LEHRBUCH

Peter Marwedel

Eingebettete Systeme

Grundlagen Eingebetteter Systeme in Cyber-Physikalischen Systemen

2. Auflage

OPEN ACCESS

Springer Vieweg

Eingebettete Systeme

Peter Marwedel

Eingebettete Systeme

Grundlagen Eingebetteter Systeme in Cyber-Physikalischen Systemen

2. Auflage

Peter Marwedel
TU Dortmund, Lehrstuhl für Eingebettete Systeme
Dortmund, Deutschland

ISBN 978-3-658-33436-9 ISBN 978-3-658-33437-6 (eBook)
https://doi.org/10.1007/978-3-658-33437-6

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Springer Vieweg
© Der/die Herausgeber bzw. der/die Autor(en) 2007, 2021
Dieses Buch ist eine Open-Access-Publikation.
Open Access Dieses Buch wird unter der Creative Commons Namensnennung 4.0 International Lizenz (http://creativecommons.org/licenses/by/4.0/deed.de) veröffentlicht, welche die Nutzung, Vervielfältigung, Bearbeitung, Verbreitung und Wiedergabe in jeglichem Medium und Format erlaubt, sofern Sie den/die ursprünglichen Autor(en) und die Quelle ordnungsgemäß nennen, einen Link zur Creative Commons Lizenz beifügen und angeben, ob Änderungen vorgenommen wurden.
Die in diesem Buch enthaltenen Bilder und sonstiges Drittmaterial unterliegen ebenfalls der genannten Creative Commons Lizenz, sofern sich aus der Abbildungslegende nichts anderes ergibt. Sofern das betreffende Material nicht unter der genannten Creative Commons Lizenz steht und die betreffende Handlung nicht nach gesetzlichen Vorschriften erlaubt ist, ist für die oben aufgeführten Weiterverwendungen des Materials die Einwilligung des jeweiligen Rechteinhabers einzuholen.
Die Wiedergabe von allgemein beschreibenden Bezeichnungen, Marken, Unternehmensnamen etc. in diesem Werk bedeutet nicht, dass diese frei durch jedermann benutzt werden dürfen. Die Berechtigung zur Benutzung unterliegt, auch ohne gesonderten Hinweis hierzu, den Regeln des Markenrechts. Die Rechte des jeweiligen Zeicheninhabers sind zu beachten.
Der Verlag, die Autoren und die Herausgeber gehen davon aus, dass die Angaben und Informationen in diesem Werk zum Zeitpunkt der Veröffentlichung vollständig und korrekt sind. Weder der Verlag, noch die Autoren oder die Herausgeber übernehmen, ausdrücklich oder implizit, Gewähr für den Inhalt des Werkes, etwaige Fehler oder Äußerungen. Der Verlag bleibt im Hinblick auf geografische Zuordnungen und Gebietsbezeichnungen in veröffentlichten Karten und Institutionsadressen neutral.

Planung: Sybille Thelen
Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft Springer Fachmedien Wiesbaden GmbH und ist ein Teil von Springer Nature.
Die Anschrift der Gesellschaft ist: Abraham-Lincoln-Str. 46, 65189 Wiesbaden, Germany

Dieses Buch ist meiner Familie gewidmet.

Vorwort

Warum sollte man dieses Buch lesen?

Während dieses Buch geschrieben wurde, d.h. im Jahr 2020, wurden intelligente Systeme (auch *smarte* Systeme genannt) zunehmend verfügbar. Solche Systeme benutzen Computer und andere Formen der Informations- und Kommunikationstechnologie (IKT), um den Menschen ihre Dienste bereitzustellen, zum Teil mit Hilfe verschiedener Formen künstlicher Intelligenz (KI). Beispielsweise sind kürzlich vorgestellte Autos zunehmend in der Lage, autonom zu fahren. In der Luftfahrt und bei schienengebundenem Verkehr sind Systeme ohne Piloten bzw. Fahrer angekündigt oder bereits verfügbar. Das Stromnetz wird intelligenter und dasselbe gilt für Gebäude. Alle diese Systeme basieren auf einer Kombination von IKT und physischen Systemen, die cyber-physikalische Systeme (engl. *Cyber-Physical System* (CPS)) genannt werden. Solche Systeme sind „ingenieurmäßig konstruierte Systeme, die auf der Synergie von rechnenden und physischen Komponenten aufbauen und davon abhängen" [413]. Aufgrund der engen Integration mit der physischen Welt müssen diese Systeme sehr verlässlich sein.

Das englische Original *physical* kann dabei im Deutschen sowohl mit „physisch" wie auch mit „physikalisch" übersetzt werden. „Physisch" trifft die Intention häufig besser, da wir meist auf eine Integration mit der Welt der (physischen) Dinge zielen und nur eine Integration mit einem Teil der Physik anstreben. Wir werden deshalb die Übersetzung „physisch" verwenden, sofern wir nicht explizit einen Bezug zur Physik meinen. Außerdem benutzen wir die Bezeichnung „cyber-physikalisches System", da sie verbreiteter ist und besser klingt.

Physische Dinge spielen auch eine entscheidende Rolle bei der Definition des Begriffs des „Internets der Dinge" (engl. *Internet of Things* (IoT)). IoT „beschreibt die um sich greifende Gegenwart einer Vielfalt von Geräten – wie Sensoren, Aktuatoren und Mobiltelefonen –, die mittels eindeutiger Adressierung in der Lage sind, miteinander zu interagieren und zu kooperieren, um gemeinsame Ziele zu erreichen" [185]. Sensor-Netzwerke und E-Roller, die anhand im Internet verzeichneter, GPS-basierter Standorte eingesammelt werden, sind Beispiele des Internets der Dinge.

Beide Begriffe, CPS und IoT, sind Verallgemeinerungen des früher geprägten Begriffs der „eingebetteten Systeme” (ES). Unter dem Begriff „eingebettete Systeme” verstehen wir dabei informationsverarbeitende Systeme, die in ein umgebendes Produkt (wie ein Auto oder ein Flugzeug) eingebettet sind [372]. Verglichen mit dem Begriff der eingebetteten Systeme betonen die Begriffe CPS und IoT stärker die physischen Objekte.

Der starke Anstieg in der Verfügbarkeit von eingebetteten Systemen wurde bereits 2001 vorhergesagt: „Die Informationstechnologie (IT) befindet sich an der Schwelle einer weiteren Revolution. .... vernetzte Systeme von eingebetteten Rechnern ... haben das Potential, die Art und Weise, in der Personen mit ihrer Umgebung interagieren, radikal zu verändern, indem sie eine Reihe von Geräten und Sensoren miteinander verbinden und erlauben, Informationen in nicht gekannter Weise zu sammeln, zu teilen und zu verarbeiten. Die Benutzung ... überall in der Gesellschaft kann bisherige Meilensteine in der Revolution der Informations(nutzung) als Zwerge erscheinen lassen.” Dieses Zitat aus einem Bericht des *National Research Councils* in den USA [411] beschreibt die dramatischen Auswirkungen der Informationstechnologie in eingebetteten Systemen sehr schön. Diese Revolution hatte bereits einen großen Einfluss und sie setzt sich fort.

Begriffe wie das durchdringende und allgegenwärtige Rechnen (engl. *pervasive* und *ubiquitous computing*), die umgebende Intelligenz (engl. *ambient intelligence*) sowie „Industrie 4.0” [68] beziehen sich ebenfalls auf den dramatischen Einfluss der Änderungen, die durch Informationstechnologie hervorgerufen werden.

Der Bedeutung eingebetteter/cyber-physikalischer Systeme und des Internets der Dinge wird bislang in Lehrplänen kaum Rechnung getragen. Dabei verlangt der Entwurf der erwähnten Systeme interdisziplinäres Wissen und Fähigkeiten jenseits der Grenzen zwischen traditionellen Fachgebieten. Eine solche breite Ausbildung erscheint allerdings als schwierig, v.a. aufgrund des großen Bereichs an relevanten Gebieten. Dieses Buch zielt auf den Wissenserwerb in einem Kernbereich der relevanten Gebiete. Dabei ist es bereits schwierig, einen solchen Kernbereich zu identifizieren. Dieses Buch soll eine Abhilfe schaffen. Es enthält Stoff für einen ersten Kurs über solche Systeme und bietet einen Überblick über Schlüsselkonzepte für eine Integration der IKT mit physischen Objekten. Es betrachtet Hardware- wie auch Softwareaspekte. Dies geschieht in Übereinstimmung mit den ARTIST[1] Richtlinien für Curricula in eingebetteten Systemen: „Die Entwicklung von eingebetteten Systemen kann die Charakteristika der zugrunde liegenden Hardware nicht ignorieren. Das Zeitverhalten, der Speicherbedarf, die Stromaufnahme und physikalische Ausfälle sind wichtig” [85].

Dieses Buch ist als Lehrbuch konzipiert, aber es enthält eine größere Anzahl an Referenzen als übliche Lehrbücher und soll auch helfen, das Themengebiet zu strukturieren. Daher soll es auch für Lehrende an Hochschulen und Ingenieure nützlich sein. Für Studierende wird der Zugang zu relevanten Informationsquellen durch die umfangreichen Referenzen erleichtert.

---

[1] ARTIST ist das Akronym für ein europäisches Exzellenznetzwerk zu eingebetteten Systemen (siehe https://artist-embedded.org und http://www.emsig.net).

Das Buch konzentriert sich auf die grundlegenden Eigenschaften von Software und Hardware. Bestimmte kommerzielle Produkte und Werkzeuge werden nur dann erwähnt, wenn sie herausragende Eigenschaften besitzen. Auch dies entspricht den ARTIST-Richtlinien: „Grundlagen sind in der Fortbildung (im Betrieb) nur sehr schwer anzueignen, wenn sie nicht von Anfang an gelernt wurden; wir müssen daher einen Schwerpunkt (in der Hochschulausbildung) darauf legen" [85]. Daher geht dieses Buch über die Programmierung von Mikrocontrollern hinaus und präsentiert die Grundlagen eingebetteter Systeme, die für den Entwurf von CPS- und IoT-Systemen benötigt werden. Mit diesem Ansatz wollen wir verhindern, dass das vorgestellte Material schnell an Aktualität verliert.

Die vorgeschlagene Positionierung des vorliegenden Lehrbuchs in Lehrplänen der Informatik und verwandten Gebieten der IKT wird in einer Veröffentlichung [373] erläutert. Ein Hauptziel dieses Buches ist es, einen Überblick über den Entwurf eingebetteter Systeme zu geben und die wichtigsten Themen darin miteinander in Beziehung zu setzen. Damit vermeiden wir ein in den ARTIST-Richtlinien erwähntes Problem: „Der Mangel an Reife des Gebiets führt zu einer großen Vielfalt an industriellen Praktiken, die vielfach aus kulturellen Gewohnheiten resultieren. ... Lehrpläne ... konzentrieren sich auf eine Technik und präsentieren keine ausreichend breite Perspektive. ... Im Ergebnis hat die Industrie Schwierigkeiten, angemessen ausgebildete Ingenieure zu finden, denen Entwurfsalternativen geläufig sind" [85].

Das Buch soll weiterhin die Lücke zwischen praktischen Erfahrungen mit der Programmierung von Mikrocontrollern und eher theoretischen Themen schließen. Zudem sollen Studierende und Lehrende dazu motiviert werden, sich tiefer in die Thematik einzuarbeiten. Während eine Reihe von Themen in diesem Buch ausführlich behandelt werden, können andere nur kurz angerissen werden. Diese kurzen Abschnitte wurden aufgenommen, um verwandte Problemstellungen miteinander in Beziehung zu setzen, ohne auf jede der Problemstellungen im Detail einzugehen. Dieser Ansatz erlaubt es Lehrenden auch, entsprechende Querverweise zu ergänzendem Material ihrer Wahl hinzuzufügen. Aufgrund der großen Zahl an Referenzen kann das Buch auch als umfassendes Tutorial benutzt werden, das Hinweise für weiteres Lesen enthält. Diese Hinweise können auch dazu anregen, das Buch in Laboren und Projekten oder als Anfangspunkt für eigene Forschungsprojekte zu nutzen.

Das Buch beschreibt Spezifikationstechniken, Hardwarekomponenten, Systemsoftware, Abbildung von Anwendungen auf Plattformen, Bewertung und Überprüfung von Entwurfszielen sowie ausgewählte Optimierungen und Testmethoden. Dabei behandelt es eingebettete Systeme und ihre Schnittstelle zur physikalischen Umgebung aus einer breiten Perspektive, aber es kann nicht jedes verwandte Gebiet abdecken. Rechtliche, soziale und ökonomische Aspekte, Mensch-Maschine-Schnittstellen, Datenanalyse, anwendungsspezifische Aspekte und eine detaillierte Darstellung physikalischer Themen wie auch die Kommunikationstechnik gehen über den Themenkreis dieses Buches hinaus. Das Internet der Dinge wird nur mit seinen Bezügen zu eingebetteten Systemen behandelt.

Wer sollte dieses Buch lesen?

Dieses Buch ist für die folgenden Leserkreise gedacht:

- Informatik- und Elektrotechnik-Studierende sowie Studierende anderer IKT-naher Studiengänge, die sich im Bereich eingebetteter/cyber-physikalischer Systeme spezialisieren wollen. Das Buch ist für Studierende im dritten Studienjahr geeignet, die über Grundkenntnisse der Rechner-Hard- und Software verfügen. Daher zielt dieses Buch hauptsächlich auf Studierende kurz vor dem Abschluss des Bachelorstudiums ab[2]. Es kann aber auch in weiterführenden Kursen zum Einsatz kommen, wenn die bisherigen Kurse den Entwurf eingebetteter Systeme nicht behandelt haben. Dieses Buch soll den Weg für **weiterführende Themen** bereiten, die in **aufbauenden Kursen** behandelt werden können. Dabei setzt das Buch Grundkenntnisse der Informatik voraus. Elektrotechnik-Studierende müssen unter Umständen auf zusätzliche Unterlagen zurückgreifen, um alle Themen dieses Buchs in vollem Umfang zu verstehen. Dies wird aber zum Teil dadurch aufgewogen, dass einige der Inhalte dieses Buches Studierenden der Elektrotechnik bereits bekannt sein sollten.
- Ingenieure, die bisher Hardwaresysteme entwickelt haben und sich in Zukunft mehr mit der Softwareseite eingebetteter Systeme befassen müssen. Dieses Buch sollte eine ausreichende Grundlage liefern, um die relevanten technischen Veröffentlichungen zu verstehen.
- Doktoranden, die schnell einen umfassenden Überblick über die wichtigsten Konzepte eingebetteter Systeme erhalten wollen, bevor sie ein spezielles Forschungsgebiet wählen.
- Professoren, die einen neuen Lehrplan für eingebettete Systeme erstellen wollen.

Wie unterscheidet sich dieses Buch von früheren Auflagen?

Die **erste englische Ausgabe** dieses Buches wurde 2003 veröffentlicht. Das Gebiet der eingebetteten Systeme entwickelt sich rasant fort, daher sind seit dem Erscheinen der ersten Ausgabe viele neue Erkenntnisse hinzugekommen. Zudem hat sich die Bedeutung zwischen verschiedenen behandelten Bereichen verschoben. In einigen Fällen war eine intensivere Auseinandersetzung mit einem Themengebiet wünschenswert. Neue Entwicklungen wurden zum Teil bereits in der ersten deutschen Übersetzung des Buches 2007 berücksichtigt, in der **zweiten englischen Auflage** (erschienen 2010/2011) übernommen und ausgebaut.

Im letzten Jahrzehnt haben sich weitere technologische Änderungen ergeben. So gab es die klare Umstellung von einzelnen Rechnerkernen zu Mehrkern-Systemen. Außerdem haben cyber-physikalische Systeme und das Internet der Dinge mehr Aufmerksamkeit erfahren. Der Verbrauch an elektrischer Energie, das thermische

[2] Dies passt zum Lehrplan, den T. Abdelzaher in einem Bericht über CPS-Ausbildung [412] veröffentlicht hat.

Verhalten, die Betriebs- und die Informationssicherheit sind zunehmend wichtiger geworden. Dadurch wurde es dadurch notwendig, eine **dritte Auflage des englischen Originals** zu publizieren. Die technischen Änderungen beeinflussten mehrere Kapitel sehr stark. In die Kapitel wurden solche Aspekte eingebetteter Systeme aufgenommen, die Grundlagen für den Entwurf von CPS- und IoT-Systemen bilden. In das Kapitel über Spezifikationen und Modellierung wurden partielle Differentialgleichungen und *Transaction-Level Modeling* (TLM) aufgenommen. Die Verwendung dieses Buchs für das Konzept der Lehre mit vertauschten Rollen des Lernens zu Hause und in der Hochschule (Stichwort: *flipped classroom*) führte zur Betrachtung weiterer Details, v.a. im Bereich der Spezifikationen. Das Kapitel über Hardware wurde um Mehrkern-Systeme und mehr Informationen zur Umwandlung zwischen analogen und digitalen Signalen (einschließlich der Pulsbreitenmodulation) erweitert und der Abschnitt über Speicher umgeschrieben. Die Beschreibungen von *Field Programmable Gate Arrays* (FPGAs) wurden erweitert und eine Einführung in Aspekte sicherer Hardware eingefügt. Das Kapitel über Systemsoftware wurde ergänzt durch einen Abschnitt über Linux in eingebetteten Systemen und mehr Informationen zu Zugriffsprotokollen auf Ressourcen. Im Kontext der Systembewertung wurden Unterabschnitte über Qualitätsmetriken, Sicherheit, Energiemodelle und thermisches Verhalten eingefügt. Das Kapitel über die Abbildung von Anwendungen auf Ausführungsplattformen wurde komplett restrukturiert: es wurde eine Klassifikation von *Scheduling*-Verfahren eingeführt und *Scheduling*-Verfahren für Mehrkern-Systeme hinzugefügt. Die Beschreibung des Hardware/Software-Codesigns wurde gestrichen. Das Kapitel über Optimierungen wurde aktualisiert und die Abbildungen wurden überarbeitet. Übungsaufgaben und eine klarere Unterscheidung zwischen Definitionen, Theoremen, Beweisen, Code und Beispielen wurden hinzugefügt.

Aufgrund der zunehmenden Bedeutung des Zugriffs auf Wissen mit Hilfe des Internets wurde auf der Basis der dritten Auflage eine **vierte Auflage** im Rahmen einer *Open Access*-Lizenz erstellt. Damit werden die Inhalte des Lehrbuchs u.a. für Studierende kostenfrei verfügbar. Für die Publikation der vierten Auflage wurden alle Kapitel wurden sorgfältig durchgesehen und ggf. aktualisiert. Fehler in der dritten Auflage wurden korrigiert. Die Beschreibung des springenden Balls wurde erweitert. Die Darstellung von Daten- und Informationssicherheit wurde restrukturiert. Der Bezug zur Datenanalyse und zur künstlichen Intelligenz tritt nunmehr deutlicher hervor. Literaturhinweise wurden aktualisiert. Die Beschreibung der Begriffe Task, Prozess, *Thread* und Job wurde – soweit möglich – präzisiert. Aufgrund der Erweiterungen ist es typischerweise nicht mehr möglich, das gesamte Buch in einem Kurs im Bachelorstudium zu behandeln. Vielmehr können Lehrkräfte eine Untermenge des Stoffes auswählen, die lokalen Anforderungen und Vorlieben entspricht.

Aufgrund eines relativ großen Interesses an der deutschen Übersetzung der ersten englischen Auflage wurde die vorliegende **deutsche Version der vierten englischen Ausgabe** erstellt. Hierfür wurden teilweise Übersetzungen der ersten Auflage durch Lars Wehmeyer und der zweiten Auflage durch Michael Engel benutzt. Zur Vermeidung sprachlich unschöner Mischungen werden deutsche Begriffe benutzt, soweit möglich. Dies machen wir sogar dann, wenn umgangssprachlich fremdsprachliche Begriffe benutzt werden (Beispiele: Fließband statt *pipeline*, Verdrängung statt

preemption). Wenn deutsche Begriffe ungebräuchlich sind und passende fremdsprachliche Begriffe bereits Teil der deutschen Sprache geworden sind, werden sie wie deutsche Worte gesetzt (Beispiele: Hardware, Software, Task, Job, Cache). Die Verwendung fremdsprachlicher Begriffe, die noch nicht Teil der deutschen Sprache geworden sind, ließ sich aber nicht vermeiden. Diese werden kursiv gesetzt. Sie werden groß geschrieben, wenn sie wie ein deutsches Substantiv benutzt werden (Beispiele: *Schedule, Thread, Chip, Timer, Layout, Flash, Deadlock*) oder wenn die Zusammensetzung einer Abkürzung klargemacht werden soll (Beispiel: *Rate Monotonic* (RM)). Sie werden klein geschrieben, wenn sie weiterhin als Fremdwort erscheinen (Beispiel: *preemption*). Soweit englische Begriffe mit Artikeln versehen werden müssen, werden diese wie bei den korrespondierenden deutschen Worten gewählt (Beispiele: **die** Task als zu **die** Aufgabe korrespondierend, **der** SPM als zu **der** ...-Speicher korrespondierend).

Zur Kennzeichnung von Hervorhebungen wird ausschließlich Fettdruck genutzt. Zitate sind durch Anführungszeichen ausgewiesen. Namen, die in diesem Buch ohne Hinweis auf Urheberrechte benutzt werden, können dennoch rechtlich geschützt sein.

Viel Spaß beim Lesen dieses Buchs!

Dortmund, *Peter Marwedel*
März 2021

Danksagungen

Die Publikation dieses Buches unter einer ***Open Access*-Lizenz** wurde durch die Förderung des Sonderforschungsbereichs (SFB) 876 zur Verfügbarkeit von Information durch Analyse unter Ressourcenbeschränkung (siehe https://sfb876.tu-dortmund.de) von der Deutschen Forschungsgemeinschaft (DFG) ermöglicht.

Die aktuelle Auflage wurde v.a. von Herrn Michael Engel und Herrn Heiko Falk **zur Korrektur gelesen**. Michael Engel war während der Durchführung von Kursen immer wieder eine große Hilfe, so auch bei der Erstellung der youtube-Videos.

In das Buch sind **Ergebnisse aus verschiedenen geförderten Projekten** eingeflossen. Ich bedanke mich deshalb v.a. bei der Deutschen Forschungsgemeinschaft (DFG) für die Finanzierung des Sonderforschungsbereichs SFB 876, des Projekts Ma 943/10 (FEHLER) sowie älterer Projekte. Die Europäische Kommission hat die Projekte MORE, Artist2, ArtistDesign, Hipeac(2), PREDATOR, MNEMEE und MADNESS gefördert. Diese Projekte boten eine ausgezeichnete Basis für die Arbeiten an der dritten und vierten Auflage dieses Buches. Die Fa. Synopsys® Inc. stellte für Forschungsarbeiten den Virtualizer™ zur Verfügung.

Ich bedanke mich ferner bei den Kollegen R. Dömer, D. Gajski, N. Dutt (UC Irvine), A. B. Kahng, R. Gupta (UC San Diego), W. Kluge, R. von Hanxleden (U. Kiel), P. Buchholz, M. Engel, H. Krumm, O. Spinczyk (TU Dortmund), W. Müller, F. Rammig (U. Paderborn), W. Rosenstiel (U. Tübingen), L. Thiele (ETH Zürich), R. Wilhelm (U. des Saarlandes), G. C. Buttazzo (U. Pisa), H. Kopetz (TU Wien), J. P. Hayes (U. Michigan) und H. Takada (U. Nagoya) für **Diskussionen und Beiträge**, die in das Buch eingeflossen sind. Für Korrekturen und Beiträge bedanke ich mich bei meinen Doktoranden sowie bei David Hec, Thomas Wiederkehr, Thorsten Wilmer und Henning Garu. Selbstverständlich ist der Autor für alle verbleibenden Fehler selbst verantwortlich.

Dieses Buch ist mit dem LaTeX-Satzsystem und der TeXnicCenter-Benutzerschnittstelle entstanden. Graphische Darstellungen wurden teilweise mit GNU Octave, mit einer Varianten des xfig-Editors sowie mit PowerPoint® erzeugt. Zur Darstellung von Programmcode wurde der Font Inconsolata zi4(varl,varqu) benutzt, der von Raph Levien, Kirill Tkachev, Michael Sharpe und mirabilos entworfen wurde. Ich danke allen Autoren für die **Bereitstellung der Software**.

Mein Dank gilt auch an alle, zu deren Lasten die Arbeitszeit für diese deutsche Übersetzung ging.

Inhaltsverzeichnis

Vorwort vii

Danksagungen xii

Inhaltsverzeichnis xv

Über den Autor xxi

Häufig benutzte mathematische Symbole xxiii

1 Einleitung 1
1.1 Geschichte der Begriffe 1
1.2 Potential 4
1.3 Herausforderungen 9
1.4 Gemeinsame Eigenschaften 18
1.5 Lehrplan-Integration von Eingebetteten Systemen, CPS und IoT . . 20
1.6 Entwurfsflüsse 23
1.7 Struktur dieses Buches 27
1.8 Aufgaben 29

2 Spezifikation und Modellierung 31
2.1 Anforderungen 31
2.2 Berechnungsmodelle 39
2.3 Frühe Entwurfsphasen 45
2.3.1 Anwendungsfälle 46
2.3.2 *(Message) Sequence Charts* 47
2.3.3 Differentialgleichungen 50
2.4 Kommunizierende endliche Automaten 53
2.4.1 Zeitgesteuerte Automaten 53
2.4.2 StateCharts 55
2.4.3 Synchrone Sprachen 66
2.4.4 Nachrichtenaustausch am Beispiel von SDL 68
2.5 Datenfluss 75
2.5.1 Überblick 75
2.5.2 Kahn-Prozessnetzwerke 76

2.5.3 SDF . . . 78
2.5.4 Simulink . . . 82
2.6 Petrinetze . . . 84
2.6.1 Einführung . . . 84
2.6.2 Bedingungs/Ereignis-Netze . . . 86
2.6.3 Stellen/Transitions-Netze . . . 87
2.6.4 Prädikat/Transitions-Netze . . . 92
2.6.5 Bewertung . . . 94
2.7 Diskrete, ereignisbasierte Sprachen . . . 96
2.7.1 Simulationszyklus diskreter Ereignissysteme . . . 96
2.7.2 Mehrwertige Logik . . . 97
2.7.3 Transaktionsbasierte Modellierung . . . 103
2.7.4 SpecC . . . 105
2.7.5 SystemC . . . 107
2.7.6 VHDL . . . 109
2.7.7 Verilog and SystemVerilog . . . 120
2.8 Von-Neumann-Sprachen . . . 122
2.8.1 CSP . . . 122
2.8.2 Ada . . . 123
2.8.3 Kommunikationsbibliotheken . . . 125
2.8.4 Weitere Sprachen . . . 126
2.9 Ebenen der Hardware-Modellierung . . . 127
2.10 Vergleich der Berechnungsmodelle . . . 130
2.10.1 Kriterien . . . 130
2.10.2 *Unified Modeling Language* (UML) . . . 133
2.10.3 Ptolemy II . . . 135
2.11 Aufgaben . . . 135

3 Hardware eingebetteter Systeme **141**
3.1 Einleitung . . . 141
3.2 Eingabe – Schnittstelle zwischen physischer und Cyber-Welt . . . 143
3.2.1 Sensoren . . . 143
3.2.2 Zeitdiskretisierung: *Sample-and-hold*-Schaltungen . . . 145
3.2.3 Fourier-Approximation von Signalen . . . 146
3.2.4 Wertediskretisierung: A/D-Wandler . . . 150
3.3 Verarbeitungseinheiten . . . 157
3.3.1 Anwendungsspezifische integrierte Schaltkreise . . . 157
3.3.2 Prozessoren . . . 158
3.3.3 Rekonfigurierbare Logik . . . 181
3.4 Speicher . . . 184
3.4.1 Zielkonflikte . . . 184
3.4.2 Speicherhierarchien . . . 185
3.4.3 Registersätze . . . 186
3.4.4 Caches . . . 186
3.4.5 *Scratchpad*-Speicher . . . 188

3.5 Kommunikation . . . 189
3.5.1 Anforderungen . . . 190
3.5.2 Elektrische Robustheit . . . 191
3.5.3 Echtzeitgarantien . . . 193
3.5.4 Beispiele . . . 194
3.6 Ausgabe – Schnittstelle zwischen Cyber- und physischer Welt . . . 196
3.6.1 D/A-Wandler . . . 197
3.6.2 Abtasttheorem . . . 200
3.6.3 Pulsbreitenmodulation . . . 204
3.6.4 Aktuatoren . . . 206
3.7 Elektrische Energie . . . 206
3.7.1 Energieerzeugung . . . 207
3.7.2 Energiespeicherung . . . 208
3.7.3 Effiziente Nutzung elektrischer Energie . . . 211
3.8 Sichere Hardware . . . 214
3.9 Aufgaben . . . 216

4 Systemsoftware **221**
4.1 Eingebettete Betriebssysteme . . . 222
4.1.1 Allgemeine Anforderungen . . . 222
4.1.2 Echtzeitbetriebssysteme . . . 226
4.1.3 Virtuelle Maschinen . . . 230
4.2 Protokolle für Ressourcen-Zugriffe . . . 231
4.2.1 Prioritätsumkehr . . . 231
4.2.2 Prioritätsvererbung . . . 233
4.2.3 *Priority Ceiling*-Protokoll . . . 236
4.2.4 *Stack Resource Policy*-Protokoll . . . 238
4.3 ERIKA . . . 240
4.4 Linux für eingebettete Systeme . . . 243
4.4.1 Struktur und Größe von Linux für eingebettete Systeme . . . 244
4.4.2 Echtzeiteigenschaften . . . 246
4.4.3 Dateisysteme für *Flash*-Speicher . . . 248
4.4.4 Verringerung des Hauptspeicherbedarfs . . . 249
4.4.5 uClinux – Linux für Systeme ohne MMU . . . 250
4.4.6 Evaluation der Nutzung von Linux in eingebetteten Systemen 252
4.5 Hardware-Abstraktionsschicht . . . 253
4.6 *Middleware* . . . 253
4.6.1 OSEK/VDX COM . . . 254
4.6.2 CORBA . . . 254
4.6.3 POSIX *Threads* (*Pthreads*) . . . 255
4.6.4 UPnP und DPWS . . . 256
4.7 Echtzeitdatenbanken . . . 256
4.8 Aufgaben . . . 257

5 Bewertung und Validierung 261
5.1 Einleitung . . . 261
5.1.1 Begriffe . . . 261
5.1.2 Multikriterielle Optimierung . . . 262
5.1.3 Relevante Kriterien . . . 265
5.2 Performanzbewertung . . . 266
5.2.1 Frühe Phasen . . . 266
5.2.2 Größtmögliche Ausführungszeiten . . . 267
5.2.3 Realzeitkalkül . . . 274
5.3 Qualitätsmetriken . . . 278
5.3.1 Näherungsweises Rechnen . . . 278
5.3.2 Einfache Qualitätskriterien . . . 278
5.3.3 Kriterien für die Datenanalyse . . . 281
5.4 Modelle des Energieverbrauchs und der Leistungsaufnahme . . . 283
5.4.1 Allgemeine Eigenschaften . . . 283
5.4.2 Energiemodell für Speicher . . . 285
5.4.3 Energiemodell für Maschinenbefehle . . . 286
5.4.4 Energiemodell für Prozessoreinheiten . . . 287
5.4.5 Energiemodell für Prozessor und Speicher . . . 287
5.4.6 Energiemodell für eine Anwendung . . . 289
5.4.7 Energiemodell für mehrere Anwendungen und Hardware-*Multithreading* . . . 290
5.4.8 Energiemodell für Android-Mobiltelefone . . . 292
5.4.9 Größtmöglicher Energieverbrauch . . . 294
5.5 Thermische Modelle . . . 294
5.5.1 Stationäres Verhalten . . . 295
5.5.2 Transientes Verhalten . . . 297
5.6 Verlässlichkeits- und Risikoanalyse . . . 302
5.6.1 Aspekte der Verlässlichkeit . . . 302
5.6.2 Informationssicherheit . . . 302
5.6.3 Betriebssicherheit . . . 303
5.6.4 Zuverlässigkeit . . . 305
5.6.5 Fehlerbaumanalyse, Fehlermöglichkeits- und Einflussanalyse 311
5.7 Simulation . . . 313
5.8 *Rapid Prototyping* und Emulation . . . 315
5.9 Formale Verifikation . . . 316
5.10 Aufgaben . . . 318

6 Abbildung von Anwendungen 321
6.1 Problemdefinition . . . 321
6.1.1 Präzisierung des Entwurfsproblems . . . 321
6.1.2 Typen von *Scheduling*-Problemen . . . 324
6.2 *Scheduling* für Einzelprozessoren . . . 330
6.2.1 *Scheduling* ohne Reihenfolgebeschränkungen . . . 330
6.2.2 *Scheduling* mit Reihenfolgebeschränkungen . . . 336

6.2.3 Periodisches *Scheduling* ohne Reihenfolgebeschränkungen . 338
6.2.4 Periodisches *Scheduling* mit Reihenfolgebeschränkungen . . 346
6.2.5 Sporadische Ereignisse 346
6.3 *Scheduling* für unabhängige Jobs auf identischen Multiprozessoren . 346
6.3.1 Partitioniertes *Scheduling* 347
6.3.2 Globales *Scheduling* mit dynamischen Prioritäten 350
6.3.3 Globales *Scheduling* für feste Job-Prioritäten 353
6.3.4 Globales *Scheduling* für feste Task-Prioritäten 356
6.4 Abhängige Jobs auf homogenen Multiprozessor-Systemen 358
6.4.1 *As-Soon-As-Possible-Scheduling* 358
6.4.2 *As-Late-As-Possible-Scheduling* 360
6.4.3 *List-Scheduling* 361
6.4.4 Optimales *Scheduling* mit Ganzzahliger Programmierung . . 363
6.5 Abhängige Jobs auf heterogenen Multiprozessoren 365
6.5.1 Problem-Beschreibung 365
6.5.2 Statisches *Scheduling* mit lokalen Heuristiken 365
6.5.3 Statisches *Scheduling* mit Ganzzahliger Programmierung . . 369
6.5.4 Statisches *Scheduling* mit Evolutionären Algorithmen 369
6.5.5 Dynamisches und Hybrides *Scheduling* 374
6.6 Aufgaben 375

7 Optimierung 379
7.1 *High-Level*-Optimierungen 380
7.1.1 Einfache Schleifentransformationen 380
7.1.2 Kachel-/Blockweise Verarbeitung von Schleifen 382
7.1.3 Aufteilen von Schleifen 384
7.1.4 Falten von Feldern 386
7.1.5 Wandlung von Gleitkomma- in Festkommadarstellung . . . 387
7.2 Nebenläufigkeit von Tasks 388
7.3 Compiler für eingebettete Systeme 392
7.3.1 Einleitung 392
7.3.2 Energiegewahre Übersetzung 393
7.3.3 Speicherarchitektur-gewahre Übersetzung 394
7.3.4 Zusammenführung von Compilern und Zeitanalyse 403
7.4 Energieverwaltung und thermisches Management 405
7.4.1 Dynamische Spannungsskalierung (DVS) 405
7.4.2 Dynamische Leistungsverwaltung (DPM) 409
7.4.3 Verwaltung des thermischen Verhaltens 409
7.5 Aufgaben 410

8 Test 413
8.1 Anwendungsbereich 413
8.2 Testverfahren 415
8.2.1 Testmustererzeugung für Modelle auf Gatterebene 415
8.2.2 Selbsttestprogramme 416

8.3 Auswertung von Testmustermengen und Systemrobustheit 417
8.3.1 Fehlerüberdeckung 417
8.3.2 Fehlersimulation 417
8.3.3 Fehlerinjektion 418
8.4 Entwurf für Testbarkeit 419
8.4.1 Motivation 419
8.4.2 *Scan*pfad-Entwurf 420
8.4.3 Signaturanalyse 421
8.4.4 Erzeugung von Pseudozufalls-Testmustern 423
8.5 Aufgaben 423

A Ganzzahlige Lineare Programmierung 425

B Kirchhoffsche Gesetze und Operationsverstärker 427

C Seitenadressierung und Speicherverwaltungseinheiten 433

Literaturverzeichnis 435

Sachverzeichnis 461

Über den Autor

Peter Marwedel wurde in Hamburg geboren. Er erhielt 1974 den Doktortitel in Physik von der Universität Kiel. Von 1974 bis 1989 war er Mitarbeiter des Instituts für Informatik und Angewandte Mathematik an dieser Universität. 1987 habilitierte er sich in Informatik. Von 1989 bis 2014 leitete er den Lehrstuhl für „Technische Informatik und Eingebettete Systeme" der Fakultät für Informatik der TU Dortmund. Seit 1997 ist er Vorstands-Vorsitzender des ICD e.V., eines *spin-offs*, das sich auf den Technologietransfer spezialisiert hat. Peter Marwedel war 1985/1986 als Gastprofessor an der Universität Paderborn und 1995 an der *University of California at Irvine*. Von 1992 bis 1995 hatte er das Amt des Dekans der Fakultät für Informatik inne. Er leitete einen Bereich des Exzellenznetzwerks *ArtistDesign* für eingebettete Systeme und Echtzeitsysteme. Außerdem war er bis zum Jahr 2015 Stellvertretender Sprecher des Sonderforschungsbereiches SFB 876 über Verfügbarkeit von Information durch Analyse unter Ressourcenbeschränkung (siehe http://www.sfb876.tu-dortmund.de).

1975 begann er im Rahmen des MIMOLA-Projekts mit der Arbeit an der *High-Level*-Synthese (v.a. von *Very Long Instruction Word*-Maschinen) und der Synthese von Selbsttestprogrammen für Prozessoren. Später beschäftigte er sich mit retargierbaren und ressourceneffizienten Compilern für eingebettete Prozessoren, speziell mit der Ausnutzung von Speicherhierarchien zwecks Minimierung von Energieverbrauch und größtmöglicher Laufzeit. Weitere Projekte behandelten die automatische Parallelisierung von Software für Mehrkern-Systeme sowie Wechselwirkungen zwischen der Fehlertoleranz und dem Einhalten von Realzeitschranken. Er initiierte im Rahmen des SFBs 876 Arbeiten zur Abbildung von statistischen Optimierungsverfahren auf Mehrkern-Systeme. Für die Lehre entwickelte Peter Marwedel das vorliegende Lehrbuch sowie begleitende youtube-Videos. Sommerschulen belegen sein Interesse an der Lehre. Er hat im Jahr 2003 den Lehrepreis seiner Universität verliehen bekommen.

Peter Marwedel ist IEEE *Fellow* und *Fellow* der Tagung DATE. Zu den weiteren Preisen gehören der *ACM SIGDA Distinguished Service Award*, der *EDAA Lifetime Achievement Award* und der *ESWEEK Lifetime Achievement Award*.

Er ist verheiratet und hat zwei Töchter und einen Sohn. Wandern, photographieren, Rad fahren und Modellbahnen zählen zu seinen Hobbys.

E-mail: peter.marwedel@tu-dortmund.de
Webseiten: http://ls12-www.cs.tu-dortmund.de/~marwedel,
https://www.marwedel.eu/peter

Häufig benutzte mathematische Symbole

Aufgrund des breiten Themenspektrums in diesem Buch könnte es leicht passieren, dass dasselbe Symbol für verschiedene Zwecke benutzt wird. Aus diesem Grund wurden die Symbole so gewählt, dass Verwechselungen vermieden werden können. Die nachfolgende Tabelle soll helfen, eine konsistente Notation zu verwenden.

a	Gewichtsfaktor
a	Zuordnung (engl. *allocation*)
A	Verfügbarkeit (engl. *availability*, siehe Zuverlässigkeit)
A	Fläche (engl. *area*)
A	Ampere
$b_{..}$	Bandbreite der Kommunikation
B	Bandbreite der Kommunikation
c_R	Charakteristischer Vektor eines Petrinetzes
c_p	spezifische Wärmekapazität
c_v	volumetrische Wärmekapazität
C_i	Ausführungszeit, Rechenzeit
C	Kapazität
C	Bedingungen in Petrinetzen
C_{th}	Wärmekapazität
°C	Grad Celsius
d_i	absolute *Deadline*
D_i	relative *Deadline*
$e(t)$	Eingangssignal
e	Eulersche Zahl ($\approx$ 2,71828)
E	Energie
E	Kante(n) eines Graphen
f	Frequenz
$f()$	allgemeine Funktion
f	Wahrscheinlichkeitsdichte
f_i	Fertigstellungszeit von Task/Job i
F	Wahrscheinlichkeitsverteilung
F	Flussrelation in Petrinetzen
g	Erdbeschleunigung
g	Verstärkung eines Operationsverstärkers
$g(t)$	Signal
G	Graph
h	Höhe
$h(t)$	Signal
i	Index, v.a. von Tasks/Jobs
I	Strom
j	Index, abhängige(r) Task/Job
J	Menge von Jobs
J	Joule
J_j	Job j
J	Jitter

k	Index, v.a. von Prozessoren
k	Boltzmann-Konstante ($\approx 1{,}3807 * 10^{-23}$ J/K)
K	Kelvin
l	Index, v.a. von Prozessoren
l_i	Schlupf (engl. *laxity*) von Task/Job i
L	Prozessortyp
L	Länge/Dicke eines Leiters
L_i	Verspätung (engl. *lateness*) von Task τ_i
L_{max}	maximale Verspätung
m	Anzahl von Prozessoren
m	Masse
m	Meter
m	milli-Prefix
M	Markierung eines Petrinetzes
MS_{max}	*Makespan*
n	Index
n	Anzahl von Tasks/Jobs
N	Netz
$\mathbb{N}$	natürliche Zahlen
$O()$	Landau-Symbol
p_i	Priorität von Task τ_i
p_i	Platz i eines Petrinetzes
P	Leistung (engl. *power*)
$P(S)$	Operation auf einem Semaphor
Q	Auflösung (engl. *resolution*)
Q	elektrische Ladung (engl. *charge*)
r_i	Bereitstellungszeit (engl. *release time*)
R	Zuverlässigkeit (engl. *reliability*)
R_{th}	thermischer Widerstand (engl. *thermal resistance*)
$\mathbb{R}$	reelle Zahlen
s	Index, v.a. von Zeiten
s	Stoßzahl (engl. *restitution*)
s_j	Startzeit von Task/Job j
s	Sekunde
S	Zustand
S	Semaphor
$\mathcal{S}$	*Schedule*
S_j	Größe eines Speichers j
t	Zeit (*time*)
t_i	Transition i eines Petrinetzes
T	Periode
T_i	Periode von Task τ_i
u_i	Auslastung von Task τ_i
$U..$	Auslastung (engl. *utilization*)
U_{max}	Maximale Auslastung

v Geschwindigkeit (engl. *velocity*)
V Knoten (engl. *vertex*) eines Graphen
V Spannung (engl. *voltage*)
V Volt
V_t Schwellenspannung (engl. *threshold voltage*)
$V(S)$ Operation auf einem Semaphor
$\mathcal{V}$ Volumen
$w(t)$ Signal
$W(p,t)$ Gewicht in einem Petrinetz
W Watt
x Eingabevariable
$x(t)$ Signal
$x_{..}, X_{..}$ Entscheidungsvariablen
$y(t)$ Signal
$y_{..}, Y_{..}$ Entscheidungsvariablen
$z(t)$ Signal
Z Zeitgeber
Z Große Impedanz
$\mathbb{Z}$ Ganze Zahlen
$\alpha^{..}$ Ankunftskurve im Realzeitkalkül
α Häufigkeit des Schaltens
α 1. Komponente in Pinedos Triplet-Notation
$\beta^{..}$ Bearbeitungs-/Übertragungsleistung im Realzeitkalkül
β 2. Komponente in Pinedos Triplet-Notation
β Kehrwert der maximalen Prozessorauslastung
$\gamma^{..}$ Arbeitslast im Realzeitkalkül
γ 3. Komponente in Pinedos Triplet-Notation
Δ Zeitintervall
θ Temperatur
κ thermische Leitfähigkeit
λ Ausfallrate
π Zahl Pi ($\approx$ 3,1415926)
π Menge von Prozessoren
π_i Prozessor i
ρ Massendichte
τ_i Task τ_i
τ Menge von Tasks
ξ Schwelle für *RM-US Scheduling*

Kapitel 1
Einleitung

In diesem Kapitel werden wir einige grundlegende Begriffe, die im Zusammenhang mit eingebetteten Systemen benutzt werden, zusammen mit der geschichtlichen Entwicklung vorstellen, sowie auf Möglichkeiten, Herausforderungen und gemeinsame Eigenschaften von eingebetteten und cyber-physikalischen Systemen eingehen. Außerdem werden Aspekte der Ausbildung, Entwurfsabläufe und die Struktur dieses Buches vorgestellt werden.

1.1 Geschichte der Begriffe

Unter dem Begriff „Informationsverarbeitung" stellte man sich bis zum Ende der achtziger Jahre des letzten Jahrhunderts unweigerlich Großrechner und riesige Bandlaufwerke vor. Die Miniaturisierung ermöglichte später die Einführung der Informationsverarbeitung mit Hilfe von PCs (engl. *Personal Computers*). Büroanwendungen dominierten, aber einige Rechner wurden auch für Steuerungen und Regelungen innerhalb von Rückkopplungsschleifen in der physischen Umgebung eingesetzt.

Später schuf Mark Weiser den Begriff des **allgegenwärtigen Rechnens** (engl. *ubiquitous computing*) [572]. Dieser Begriff bezieht sich auf Weisers Vorhersage, künftig Rechner und Information überall und zu jeder Zeit zur Verfügung zu haben. Weiser sagte auch vorher, dass Rechner künftig in Produkte integriert werden würden, sodass sie unsichtbar werden würden. Daher schuf er den Begriff des **unsichtbaren Computers**. In ähnlicher Weise sorgte die vorhergesagte Durchdringung unseres täglichen Lebens mit rechnenden Geräten zur Einführung der Begriffe **durchdringendes Rechnen** (engl. *pervasive computing*) und **umgebende Intelligenz** (engl. *ambient intelligence*). Die drei Begriffe legen das Schwergewicht auf nur leicht unterschiedliche Aspekte künftiger Informationstechnologie. Der Begriff „allgegenwärtiges Rechnen" ist verbunden mit dem langfristigen Ziel, Information jederzeit und überall bereitzustellen, wohingegen der Begriff „durchdringendes Rechnen" mehr mit den praktischen Aspekten und der Nutzung der bereits vorhandenen Technologie assoziiert wird. Der Begriff „umgebende Intelligenz" wird

© Der/die Autor(en) 2021

P. Marwedel, *Eingebettete Systeme*, https://doi.org/10.1007/978-3-658-33437-6_1

in der Regel mit der Nutzung der Informationstechnologie in künftigen Wohnhäusern und intelligenten Bürogebäuden in Verbindung gebracht. Ein Teil der Visionen ist durch die Verbreitung von kleinen mobilen Geräten in Kombination mit dem mobilen Internet bereits Wirklichkeit geworden und diese Verbreitung ist durchaus „durchdringend" in dem Sinne, dass sie viele Bereiche des täglichen Lebens bereits beeinflusst hat. Von der „Künstlichen Intelligenz" wird ebenfalls ein großer Einfluss erwartet.

Dabei wird häufig nicht bewusst wahrgenommen, dass die Miniaturisierung auch die Integration der Informationsverarbeitung mit der Umgebung vorangebracht hat. Ein derart integriertes informationsverarbeitendes System kann als eingebettetes System (ES) gemäß nachfolgender Definition bezeichnet werden.

Definition 1.1 (Marwedel [372]): „Eingebettete Systeme sind informationsverarbeitende Systeme, die in umgebende Produkte integriert sind."

Beispielsweise finden wir eingebettete Systeme in Autos, Schienenfahrzeugen, Flugzeugen, der Telekommunikation und bei der Fertigungsautomatisierung. Angekündigte Produkte wie selbstfahrende Autos und Schienenfahrzeuge machen dabei klar, dass die Miniaturisierung bei eingebetteten Systemen ähnliche Veränderungen erwarten lässt wie die Verfügbarkeit von Mobilfunkgeräten. Für die genannten Beispiele gibt es dabei eine Vielzahl ähnlicher Anforderungen, wie u.a. ein vorhersagbares Echtzeitverhalten, Verlässlichkeit und Effizienz. Für diese Systeme ist der Bezug zur Umgebung sehr wichtig. Dieser Bezug wird in dem nachfolgenden Zitat betont [330]:

"Embedded software is software integrated with physical processes. The technical problem is managing time and concurrency in computational systems".

Obiges Zitat könnte als Definition des Begriff „eingebettete Software" benutzt werden und könnte zu einer Definition des Begriffs „eingebettete Systeme" verallgemeinert werden, indem wir einfach das Wort „Software" durch das Wort „ System" ersetzen.

Der starke Bezug zur realen Umgebung wurde noch mehr betont durch die Einführung des Begriffs „cyber-physikalisches System". Dieser kann wie folgt definiert werden.

Definition 1.2 (Lee [331]): „**Cyber-physikalische Systeme** (engl. *Cyber-Physical Systems* (CPS)) bestehen aus der Integration von Berechnungen und physikalischen Prozessen."

Eine Betonung der angesprochenen Integration ist sinnvoll, denn in einer Welt voller Anwendungen auf Servern, PCs und Mobilfunkgeräten wird sie häufig ignoriert. Für cyber-physikalische Systeme können wir erwarten, dass Modelle auch die Umgebung beschreiben. In diesem Sinne können wir uns vorstellen, dass cyber-physikalische Systeme eingebettete Systeme (d.h. den informationsverarbeitenden Teil) wie auch die (dynamische) physische Umgebung enthalten, oder CPS = ES + (dynamisches) physisches bzw. physikalisches System. Dies spiegelt sich auch in Abb. 1.1 wider.

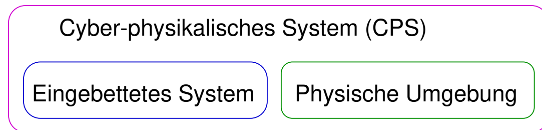


Abb. 1.1 Beziehung zwischen eingebetteten Systemen und physischer Umgebung

In ihrem Aufruf für Projektvorschläge nennt die *National Science Foundation* (NSF) in den USA dabei auch die Verbindung (und damit die Kommunikation) verteilter Systeme (gemäß [413] in deutscher Übersetzung): „Die sich herausbildenden cyber-physikalischen Systeme werden koordiniert, verteilt und verbunden sein und sie müssen robust und reagierend sein."

Vernetzung (und damit die Kommunikation) wird auch im acatech-Bericht über CPS deutlich angesprochen (gemäß [6] in deutscher Übersetzung): Cyber-physikalische Systeme „stellen vernetzte, Software-intensive eingebettete Systeme in einer Regelschleife dar, die vernetzte und verteilte Dienste anbieten."

In einem Aufruf für Projektvorschläge der Europäischen Kommission [155] werden Verbindungen und Zusammenarbeit von Komponenten erwähnt: „Cyber-physikalische Systeme (CPS) beziehen sich auf die nächste Generation von IKT-Systemen, die verbunden und zusammenarbeitend sind – auch über das Internet der Dinge – und die Bürgern und Firmen einen breiten Bereich von neuartigen Anwendungen und Diensten anbieten."

Dabei hat die Europäische Kommission bereits früher die Bedeutung der Kommunikation hervorgehoben, wie es in Abb. 1.2 zu sehen ist.

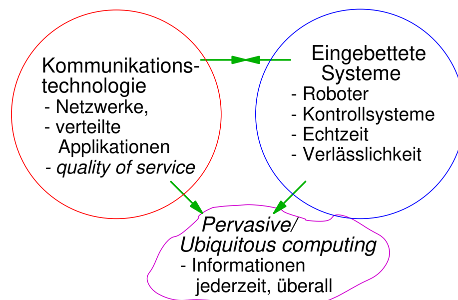


Abb. 1.2 Bedeutung der Kommunikation (© Europäische Kommission)

Aus diesen Zitaten wird klar, dass die Autoren unter dem Begriff CPS nicht nur die Integration der Cyber- und der physischen Welt verstehen, sondern dass es auch einen starken Kommunikationsaspekt gibt. Tatsächlich wird der Begriff CPS nicht immer konsistent benutzt: einige Autoren betonen die Integration mit der physischen Umgebung, andere betonen die Kommunikation.

Die Kommunikation wird noch expliziter in dem Begriff des „Internets der Dinge" (engl. *Internet of Things* (IoT)), der wie folgt definiert werden kann:

Definition 1.3 ([185]): Das **Internet der Dinge** „beschreibt die um sich greifende Gegenwart einer Vielfalt von Geräten – wie Sensoren, Aktuatoren und Mobiltelefonen –, die mittels eindeutiger Adressierung in der Lage sind, miteinander zu interagieren und zu kooperieren, um gemeinsame Ziele zu erreichen.”

Der Begriff verbindet die physische Umgebung mit dem Internet, um Sensoren-Information im Internet verfügbar zu machen und um Aktuatoren aus dem Internet heraus zu steuern. Es wird erwartet, dass das Internet der Dinge die Kommunikation zwischen Milliarden von Geräten erlauben wird. Diese Vision beeinflusst viele Industriezweige.

Die Nutzung der IoT-Technologie für die Produktionstechnik ist als „Industrie 4.0” [68] bezeichnet worden. Ziel ist eine flexiblere Produktion, bei welcher der gesamte Lebenszyklus durch diese Technologie unterstützt wird.

Insgesamt bleibt es etwas dem Einzelnen überlassen, die Verbindung zwischen physischen Umgebungen und der Cyber-Welt als CPS oder als IoT zu bezeichnen. Wir können aber erwarten, dass CPS und IoT gemeinsam betrachtet den größten Teil der künftigen Anwendungen der Informationstechnologie ausmachen, insbesondere wenn wir die Systeme mit intelligenten Algorithmen ausstatten.

Der Entwurf dieser künftigen Anwendungen setzt die Kenntnis der fundamentalen Entwurfstechniken für eingebettete Systeme voraus. Dieses Buch behandelt genau diese fundamentalen Techniken, indem es allgemeine Schnittstellen zwischen den eingebetteten Systemen und der CPS-oder IoT-Umgebung vorstellt. Diese fundamentalen Techniken werden beim Entwurf von CPS- und IoT-Systemen benötigt, auch wenn das nicht immer wieder in den einzelnen Kapiteln wiederholt wird. Für anwendungsspezifische Information sei jedoch auf spezialisiertere Quellen verwiesen.

1.2 Potential

Die nachfolgende Liste verdeutlicht das riesige Potential für Anwendungen der Informationsverarbeitung in CPS- und IoT-Systemen und sie gibt einen Eindruck von der Variationsbreite der verschiedenen Bereiche:

- **Transportwesen und Mobilität:**
  - **Automobilelektronik:** Moderne Autos sind in technisch fortgeschrittenen Ländern nur verkäuflich, wenn sie über eine umfangreiche Ausstattung an elektronischen Komponenten verfügen [416]. Beispiele dafür sind *Airbag*-Steuergeräte, Motorsteuerungen, Navigationssysteme, Antiblockiersysteme (ABS), elektronische Stabilitätsprogramme (ESP), Klimaanlagen, Diebstahlschutz, Fahrer-Assistenzsysteme und viele mehr. Ein Trend geht hin zu stärker autonom fahrenden Autos. Eingebettete Systeme können nicht nur den Komfort und die Sicherheit eines Autos verbessern, sie können potentiell auch dazu beitragen, die Umweltbelastung von Automobilen zu reduzieren. E-Mobilität ist ohne umfangreiche elektronische Ausstattung nicht vorstellbar.

- **Luftfahrt:** Ein nennenswerter Anteil der Gesamtkosten von Flugzeugen wird für informationsverarbeitende Systeme, wie z.B. Flugkontrollsysteme, Kollisionsvermeidungssysteme, Piloteninformationssysteme, Autopiloten usw. aufgewendet. Die Verlässlichkeit ist von äußerster Wichtigkeit[1]. Eingebettete Systeme können die von Flugzeugen verursachten Emissionen (wie z.B. Kohlendioxid) reduzieren. Autonomes Fliegen wird auch mehr und mehr zu einer Realität, zumindest für spezielle Anwendungsbereiche.
- **Schienenverkehr:** Die Lage im Eisenbahnbereich ist ähnlich der Situation bei Autos und Flugzeugen. Auch hier machen Sicherheitseinrichtungen einen beträchtlichen Anteil der Gesamtkosten aus und die Verlässlichkeit ist ausgesprochen wichtig. Neue Signaltechniken zielen darauf, den Zugverkehr bei hohen Geschwindigkeiten und kurzen Abständen zwischen den Zügen zu sichern. Das *European Train Control System* (ETCS) [444] ist ein Schritt in diese Richtung. Autonomer Schienenverkehr wird in begrenzten Kontexten (wie z.B. bei *Shuttle*-Zügen auf Flughäfen) bereits eingesetzt.
- **Maritime Systeme und Schiffbau:** CPS- und IoT-Systeme finden auch im Bereich moderner Schiffe und anderer maritimer Systeme Anwendung, z.B. für die Navigation, für Sicherheitskonzepte, für allgemeine Betriebsoptimierung und für die Dokumentation der Abläufe (siehe z.B. http://www.smtcsingapore.com/ und https://dupress.deloitte.com/dup-us-en/focus/internet-of-things/iot-in-shipping-industry.html).
- **Neue Mobilitätskonzepte:** Der Einsatz von Informationstechnologie und deren Komponenten ermöglicht neue Mobilitätskonzepte. *E-Bikes* lassen auch ungeübte Personen größere Strecken per Fahrrad überwinden. Das subtile Wechselspiel zwischen menschlicher Muskelkraft und Kraft des Elektromotors macht *E-Bikes* zu einem Musterbeispiel für cyber-physikalische Systeme, ähnlich wie E-Roller. Das abendliche Einsammeln von E-Rollern anhand einer Standortliste im Internet gibt dem Begriff „Internet der Dinge” eine besonders anschauliche Bedeutung. Auch bei neuen Sammelbus-, Mietwagen- und Fahrdienst-Konzepten sind CPS- und IoT-Techniken für eine Realisierung wichtig.

• **Maschinenbau** (einschl. Fabrikautomatisation): Maschinen und Fertigungsanlagen sind Bereiche, in denen eingebettete/cyber-physikalische Systeme traditionell schon seit Jahrzehnten zum Einsatz kommen. CPS- und IoT-Technologie kann genutzt werden, um die Produktion weiter zu optimieren und sie ist ein Schlüssel zu einer flexibleren Produktion, die das Ziel der „Industrie 4.0”-Initiativen ist [68]. Die weitere Automatisierung wird durch die Logistik ermöglicht, die ihrerseits hierzu CPS- und IoT-Technologien einsetzt [298]. Es gibt viele Möglichkeiten, CPS- und IoT-Technologien in der Logistik einzusetzen. Beispielsweise erlaubt die *Radio Frequency Identification* (RFID)-Technologie die einfache Identifikation jedes Objektes, über Einbindung in ein Netz ggf. weltweit. Die mobile Kommunikation erlaubt eine nie gekannte Interaktion aller an einer Fertigung Beteiligten.

[1] Das Beispiel der Boeing 737 MAX [420] macht die Gefahren sehr deutlich.

- **Robotik:** Die Robotik ist ein weiterer traditioneller Einsatzbereich für eingebettete/cyber-physikalische Systeme. Bei Robotern sind insbesondere die mechanischen Aspekte von Interesse. Es wurden Roboter nach dem Vorbild von Menschen oder Tieren entwickelt. Abb. 1.3 zeigt einen solchen Roboter.

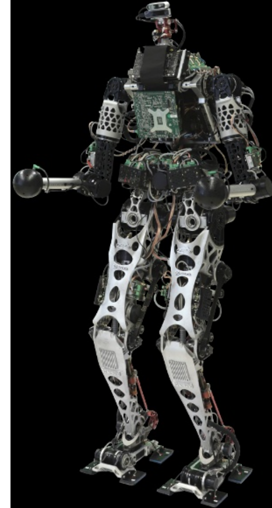


Abb. 1.3 Humanoider Roboter „Lola" (mit Genehmigung durch D. Rixen, Lehrstuhl für Angewandte Mechanik, TU München), © TU München

- **Energieversorgung und das intelligente Netz:** In der Zukunft wird die elektrische Energieversorgung voraussichtlich sehr viel stärker dezentralisiert sein. Dies macht es schwer, die Stabilität der Stromversorgung sicherzustellen. Nur mit IKT-Systemen kann es gelingen, eine ausreichende Stabilität zu erreichen. Informationen zur intelligenten Versorgung mit elektrischer Energie können beispielsweise unter den Adressen https://www.smartgrid.gov/the_smart_grid und http://www.smartgrids.eu/ gefunden werden.
- **Bauingenieurwesen:** CPS-Geräte können im Bauingenieurwesen eingesetzt werden. Beispielsweise kann die Statik von natürlichen und künstlichen Strukturen (z.B. von Bergen, Vulkanen, Brücken und Dämmen) überwacht werden. Die Abb. 1.4 zeigt beispielsweise den Damm der Möhnetalsperre. Mit Hilfe von CPS-Technologie ist es möglich, bei Gefahren rechtzeitige Warnungen zu verbreiten[2].

- **Intelligente Gebäude:** Intelligente Gebäude sind einer der Bereiche des Bauingenieurwesens. Informationsverarbeitung kann verwendet werden, um den Komfort in Gebäuden zu verbessern, den Energieverbrauch von Gebäuden zu senken und die Sicherheit zu erhöhen. Dafür müssen bisher nicht zusammenhängende Teilsysteme miteinander verbunden werden. Der Trend geht dahin, die Klimasteuerung, die Beleuchtung, die Zugangskontrolle sowie die Verwaltung und Verteilung von Informationen in ein einziges System zu integrieren. Damit können

[2] Der Dammbruch von Brumadinho (siehe https://de.wikipedia.org/wiki/Dammbruch_von_Brumadinho) zeigt, wie es nicht ablaufen soll.

Abb. 1.4 Beispiel eines zu beobachtenden Dammes, des Möhnesee-Dammes ©P. Marwedel

z.B. Klimaanlagen in leeren Räumen mit geringerer Leistung betrieben werden und gleichzeitig die Beleuchtung nur bei Bedarf aktiviert werden. Die durch die Klimatechnik erzeugten Geräusche können automatisch auf das notwendige Minimum reduziert werden. Die intelligente Verwendung von Verdunkelungen kann zudem Beleuchtungs- und Klimaverhältnisse optimieren. Freie Räume können an zentralen Stellen im Gebäude angezeigt werden, damit können die Planung von kurzfristig anberaumten Besprechungen oder die Reinigung vereinfacht werden. In Notfallsituationen kann eine Liste von belegten Räumen am Gebäudeeingang angezeigt werden (solange der Strom nicht ausgefallen ist). Durch diese Maßnahmen lässt sich Energie für Kühlung, Heizung und Beleuchtung einsparen und zudem kann die Sicherheit des Gebäudes verbessert werden. Anfangs werden solche Systeme nur in hochtechnisierten Bürogebäuden vorhanden sein. Ein Trend hin zu energieeffizienten Gebäuden ist aber auch beim Entwurf von Wohnhäusern zu erkennen. Eines der Ziele ist dabei der Entwurf sogenannter **Niedrigenergiehäuser** bzw. **Nullenergiehäuser** (Gebäude, die so viel Energie erzeugen, wie sie verbrauchen) [427]. Solche Gebäude könnten einen deutlichen Beitrag zur Verringerung der globalen Kohlendioxid-Emissionen und damit der globalen Erwärmung leisten.
- **Agrartechnik:** Es gibt viele IoT-Anwendungen in der Landwirtschaft, wie beispielsweise die Sicherstellung der Nachvollziehbarkeit des Lebensweges von Nutztieren. Im Fall des Ausbruchs ansteckender Krankheiten können so unmittelbar Maßnahmen getroffen werden [516][3]. Auch kann der Einsatz von Düngemitteln mittels IoT-Technologie dem Boden angepasst werden.
- **Gesundheitswesen:** Produkte des Gesundheitswesens gewinnen insbesondere durch die zunehmende Alterung der Bevölkerung an Bedeutung. Das Potential zur Innovation beginnt bei neuen Sensoren, die Krankheiten schneller und verlässlicher detektieren können. Neue **Datenanalysetechniken** können benutzt werden, um erhöhte Risiken zu erkennen und die Chancen auf Heilung zu verbessern. Dafür kommen i.d.R. auch neue Algorithmen zum Einsatz, beispielsweise solche der

[3] Die Bedeutung der Nachvollziehbarkeit von Kontakten im Allgemeinen, über die Tierzucht hinaus, wurde in der Corona-19-Krise besonders deutlich.

Künstlichen Intelligenz (KI). KI bietet auch die Möglichkeit einer personalisierten Medizin, um die Therapien durch angepasste Medikamente zu unterstützen. Neue Geräte können beispielsweise Behinderten helfen. Auch können Operationen roboterassistiert durchgeführt werden. CPS- und IoT-Technologien können dazu dienen, Ergebnisse einer Therapie besser zu überwachen und damit Ärzten die Möglichkeit zu geben, die Wirksamkeit einer Behandlung zu überprüfen. Diese Überwachung schließt auch entfernt lebende Patienten ein. Verfügbare Daten können in Patienteninformationssystemen gespeichert werden. Listen von Projekten in diesem Bereich können beispielsweise unter http://cps-vo.org/group/medical-cps und unter http://www.nano-tera.ch/program/health.html gefunden werden.

- **Wissenschaftliche Experimente:** Viele der gegenwärtigen wissenschaftlichen Experimente, insbesondere in der Physik, erfordern eine Erfassung der experimentellen Daten mit IKT-Geräten, d.h. der Cyber-Welt. Die Kombination von physikalischen Experimenten und der Cyber-Welt kann als Spezialfall cyberphysikalischer Systeme gesehen werden.
- **Öffentliche Sicherheit:** Das Interesse an Maßnahmen zur Steigerung der öffentlichen Sicherheit wächst im Angesicht von Gewaltakten und Pandemien. CPS- und IoT-Systeme können die öffentliche Sicherheit stärken. Dies schließt die Erkennung oder Authentisierung von Personen über verschiedene Sensoren ein.
- **Katastrophenschutz:** Im Fall größerer Katastrophen wie Erdbeben oder Überflutungen ist es wesentlich, Leben zu retten und Überlebenden zu helfen. Hierzu sind flexible Kommunikationsstrukturen erforderlich.
- **Militärische Anwendungen:** Informationsverarbeitende Komponenten sind seit vielen Jahren ein wichtiger Bestandteil militärischer Systeme. Einige der allerersten Rechner wurden verwendet, um militärische Radarsignale zu analysieren. Sie sind auch heute wichtiger Teil von Abwehrsystemen.
- **Telekommunikation:** Die Mobiltelefonie ist einer der am stärksten wachsenden Märkte der vergangenen Jahre. Wichtige Aspekte von Mobiltelefonen sind der Entwurf der Funkkomponenten, die digitale Signalverarbeitung und ein niedriger Energieverbrauch. Telekommunikation ist auch ein wesentlicher Aspekt für IoT-Technologien.
- **Unterhaltungselektronik:** Video- und Audiosysteme sind ein wichtiger Bereich der Elektronikindustrie. Diese Geräte verwenden einen stetig ansteigenden Anteil an informationsverarbeitenden Komponenten. Neue Dienste und eine gesteigerte Qualität werden dabei durch neuartige Methoden der digitalen Signalverarbeitung ermöglicht. Viele (vor allem hochauflösende) Fernseher, *Smartphones* und Spielkonsolen verwenden leistungsstarke Prozessoren und Speichersysteme. Diese stellen eine spezielle Klasse eingebetteter Systeme dar. Die Betriebssicherheit und Realzeitbedingungen sind bei dieser Klasse nicht ganz so dringend zu berücksichtigen wie bei anderen Klassen. Allerdings sind teilweise auch Realzeitbedingungen einzuhalten, beispielsweise um eine bestimmte Bildwiederholrate zu erreichen oder um Zeitbedingungen in Kommunikationsprotokollen zu garantieren. Auch sind Ressourcen wie die elektrische Energie oder die Kommunikationsbandbreite gerade bei mobilen Geräten extrem wichtig und damit liegt

hinsichtlich der Verfügbarkeit von Ressourcen eine Situation ähnlich der anderer eingebetteter Systeme vor.

Die große Menge von Beispielen zeigt eindrucksvoll die große Vielfalt von eingebetteten Systemen, die in CPS oder IoT-Systemen eingesetzt werden. Weitere Anwendungen werden in einem Bericht zu den Möglichkeiten und den Herausforderungen des Internets der Dinge aufgeführt [516]. Wir gehen davon aus, dass viele der künftigen Anwendungen der IKT-Technologie sich in solchen Systemen finden werden. Aus obiger Liste schließen wir, **dass praktisch alle Ingenieursdisziplinen betroffen sein werden**.

Die lange Liste der Anwendungen führt auch zu einer entsprechenden ökonomischen Bedeutung dieser Systeme. Laut acatech-Bericht [6] wurden zum Zeitpunkt der Berichterstellung 98% aller Mikroprozessoren in solchen Systemen eingesetzt. In gewisser Hinsicht sind eingebettete Systeme eine Voraussetzung für viele Produkte und sie haben einen Einfluss auf das Marktvolumen aller oben genannten Bereiche. Das gesamte Marktvolumen zu beziffern ist aber sehr schwer, denn es liegt deutlich über dem Marktvolumen der eingesetzten IKT-Komponenten. Es würde in die Irre führen, wenn man sich nur auf den Marktwert der Halbleiter beziehen würde, denn deren Wert macht nur einen Bruchteil des gesamten Wertes aus.

Die ökonomische Bedeutung von CPS und IoT spiegelt sich wider in Aufrufen zur Einreichung von Projektvorschlägen von fördernden Institutionen, beispielsweise in den USA [116] und in Europa [156].

1.3 Herausforderungen

Leider geht der Entwurf von eingebetteten Systemen und deren Integration in CPS- und IoT-Systeme einher mit einer großen Zahl von schwierigen Entwurfsaspekten, wie u.a. den nachfolgend aufgeführten:

- Cyber-physikalische Systeme müssen **verlässlich** sein.

  Definition 1.4: Ein System ist **verlässlich**, wenn es den vorgesehenen Dienst mit hoher Wahrscheinlichkeit bereitstellt und keinen Schaden anrichtet.

  Verlässlichkeit ist erforderlich, weil diese Systeme mit der physischen Umgebung verbunden sind und einen direkten Einfluss auf diese Umgebung haben. Dieser Aspekt muss während des gesamten Entwurfsprozesses beachtet werden.
  Verlässlichkeit umfasst eine Reihe von Teilaspekten:

  1. **Informationssicherheit** (engl. ***security***)[4]: Informationssicherheit hat den Schutz von Informationen als Ziel. Dabei können Informationen sowohl auf Papier, in Rechnern oder auch in Köpfen gespeichert sein.

[4] Im Deutschen werden die hier angeführten Begriffe *„safety"* und *„security"* meist beide durch das Wort „Sicherheit" wiedergegeben. Wir verwenden hier den Begriff „Informationssicherheit" für *security* und „Betriebssicherheit" für *safety*.

Definition 1.5 ([256, 75]): Die Informationssicherheit ist definiert als die Gewährleistung von Vertraulichkeit, Integrität und Verfügbarkeit.

Diese Gewährleistung kann durch Diebstahl oder Schäden aufgrund von Angriffen von außen gefährdet sein. Solche Angriffe können erfolgen, wenn CPS- oder IoT-Systeme mit der Umgebung verbunden werden. Cyber-Kriminalität und Cyber-Kriegsführung sind Beispiele solcher Angriffe mit einem Potential für erhebliche Schäden. Je mehr Komponenten man verbindet, umso mehr Angriffe und Schäden werden möglich. Dies ist zu einem schwerwiegenden Problem beim Entwurf und der Verbreitung von IoT-Systemen geworden.
Die einzig sichere Lösung ist es, die Komponenten nicht mit der äußeren Umgebung zu verbinden, was aber der ursprünglichen Intention der Nutzung widerspricht. Zur Gewährleistung der Informationssicherheit sollen nach Ravi et al. [301] (mindestens) die folgenden Anforderungen erfüllt werden:
– Ein **Benutzer-Identifikationsprozess** überprüft Identitäten, bevor Benutzer ein System verwenden dürfen.
– Ein **sicherer Netzwerkzugang** erlaubt eine Netzwerkverbindung nur dann, wenn das kommunizierende Gerät autorisiert ist.
– Eine **sichere Kommunikation** erfüllt dafür notwendige Eigenschaften.
– Eine **sichere Speicherung** verlangt Vertraulichkeit und Integrität der Daten.
– **Informationssicherheit der Inhalte** verlangt Einschränkungen hinsichtlich der Nutzung der Information.

2. **Vertraulichkeit** ist eine der Anforderungen an die Informationssicherheit. Sie kann wie folgt definiert werden:

   Definition 1.6 ([256, 75]): „**Vertraulichkeit** ist der Schutz vor unbefugter Preisgabe von Informationen. Vertrauliche Daten und Informationen dürfen ausschließlich Befugten in der zulässigen Weise zugänglich sein".

   Vertraulichkeit wird üblicherweise mit Techniken zur Konstruktion informationssicherer Systeme hergestellt, z.B. durch Verschlüsselung.
3. **Betriebssicherheit (engl. *safety*):**

   Definition 1.7 ([251]): Betriebssicherheit kann definiert werden als „Abwesenheit nicht annehmbarer Risiken einer physischen Verletzung oder eines Schadens der Gesundheit von Personen, weder direkt noch indirekt durch einen Schaden am Eigentum oder der Umgebung".
   „**Funktionelle Betriebssicherheit** ist der Teil der gesamten Betriebssicherheit, die davon abhängt, dass ein System oder Gerät auf Eingaben korrekt reagiert".

   „Im Kontext von Computersystemen wird der Begriff auch zur Abgrenzung von einer Sicherheit gegen äußere Angriffe, etwa durch Schadsoftware verwendet. Betriebssicherheit bezieht sich dann im Unterschied dazu auf Gefahren durch Ausfälle, die ohne Fremdeinwirkung auftreten, also etwa Hardware- oder Stromausfälle, fehlerhaft geschriebene Software oder Bedienungsfehler" [574].

4. **Zuverlässigkeit:** Dieser Begriff bezieht sich auf Fehlfunktionen von Systemen, die daraus resultieren, dass Komponenten nicht gemäß ihrer Spezifikation arbeiten, z.B. weil sie „kaputt" gehen. Zuverlässigkeit kann definiert werden als die Wahrscheinlichkeit, dass ein System den erwarteten Dienst für eine gewisse Zeitdauer erbringt[5] und sie kann als ein Aspekt der Betriebssicherheit gesehen werden. Im Rahmen der Beurteilung der Zuverlässigkeit betrachten wir keine Angriffe auf das System von außen, sondern nur Effekte, die im System während seines normalen Betriebs entstehen.
5. **Wartbarkeit:** Die Wartbarkeit ist die Wahrscheinlichkeit, dass ein ausgefallenes System innerhalb einer bestimmten Zeitspanne repariert werden kann.
6. **Verfügbarkeit:** Die Verfügbarkeit ist die Wahrscheinlichkeit, dass das System verfügbar ist. Sowohl Zuverlässigkeit wie auch Wartbarbeit müssen hoch sein, um eine hohe Verfügbarkeit zu erzielen und damit zur Verlässlichkeit beizutragen.

Entwickler eingebetteter Systeme könnten sich nun anfangs nur auf die Funktionalität eines Systems konzentrieren und versuchen, Verlässlichkeit zu einem funktionierenden Entwurf hinzuzufügen. Dieser Ansatz funktioniert in den meisten Fällen jedoch nicht, da bestimmte Entwurfsentscheidungen verhindern, dass beispielsweise die erforderliche Zuverlässigkeit im Nachhinein erreicht werden kann. Wenn die physikalische Aufteilung eines Systems falsch gewählt wurde, kann es unmöglich sein, ein redundantes System zu entwerfen. Daher darf „Zuverlässigkeit nicht erst nachträglich zu einem System hinzugefügt werden", vielmehr muss sie von Anfang an berücksichtigt werden [304]. Es müssen gute Kompromisse gefunden werden, um ein ausreichendes Maß an Betriebssicherheit und Informationssicherheit zu finden [297].
Auch perfekt entworfene Systeme können ausfallen. Dies kann daran liegen, dass die zu Grunde liegenden Annahmen über die Auslastung und möglicherweise auftretende Fehler nicht korrekt waren [304]. So kann ein System beispielsweise ausfallen, wenn es außerhalb des ursprünglich angenommenen Temperaturbereichs betrieben wird.

- Wenn wir uns die Schnittstelle zwischen der physischen und der Cyber-Welt genauer ansehen, stellen wir fest, dass physikalische und Cyber-Modelle nicht richtig zusammenpassen. Die nachfolgende Liste zeigt Beispiele:

  – Viele CPS-Systeme müssen Zeitschranken einhalten. Die Verletzung von Zeitschranken kann zu einem ernsten Qualitätsverlust für bereitgestellte Dienste führen (wenn beispielsweise die Audio- oder die Videoqualität beeinträchtigt ist) oder dem Benutzer einen Schaden zufügen (wenn beispielsweise Autos, Eisenbahnen oder Flugzeuge nicht in der erwarteten Weise funktionieren). Einige Zeitschranken heißen harte Zeitschranken:

  Definition 1.8 (Kopetz [304]): Eine Zeitschranke heißt **harte** Zeitschranke, wenn ihr Nicht-Einhalten eine Katastrophe verursachen kann. Alle anderen Zeitschranken heißen **weiche** Zeitschranken.

[5] Eine formale Definition dieses Begriffs wird in Definition 5.36 auf Seite 307 gegeben.

Viele der aktuellen informationsverarbeitenden Systeme verwenden Techniken, um die **durchschnittliche** Geschwindigkeit der Informationsverarbeitung zu erhöhen. So verbessern beispielsweise Caches die durchschnittliche Leistung eines Systems. In anderen Fällen wird zuverlässige Kommunikation realisiert, indem bestimmte Übertragungen wiederholt werden. Im Durchschnitt haben diese Wiederholungen nur einen (hoffentlich kleinen) Verlust an Übertragungsleistung zur Folge, auch wenn die Kommunikationsverzögerung für eine bestimmte Nachricht einige Größenordnungen über der normalen Verzögerung liegen kann. Im Kontext von Echtzeitsystemen sind Diskussionen über durchschnittliche Leistung oder Verzögerung eines Systems aber nicht akzeptabel. „**Eine garantierte Systemantwort muss ohne Verwendung statistischer Argumente erklärt werden können**" [304].
Viele Modellierungstechniken der Informatik benutzen keine echte Zeit. Häufig wird die Zeit ohne Benutzung physikalischer Einheiten modelliert, womit keine Unterscheidung zwischen Picosekunden und Jahrhunderten gemacht wird. Die resultierenden Probleme wurden von Edward Lee sehr klar benannt: „Das Fehlen der Zeit in Kernmodellen (der Informatik) ist aus Sicht der Software eingebetteter Systeme ein Fehler" [333].

- Viele eingebettete Systeme sind **hybride Systeme**, sie beinhalten analoge und digitale Bestandteile. Analoge Komponenten verwenden kontinuierliche Signale über einem kontinuierlichen Zeitbereich, wogegen digitale Komponenten diskrete Signalwerte über einem diskreten Zeitbereich verwenden. Viele physikalische Einheiten werden von einem Paar dargestellt, bestehend aus einer reellen Zahl und einer Einheit. Die Menge der reellen Zahlen ist überabzählbar. In der Cyber-Welt ist die Menge der darstellbaren Werte endlich. Daher können fast alle physikalischen Größen auf Computern nur approximiert werden. Bei der Simulation von physikalischen Systemen auf Computern nehmen wir üblicherweise an, dass uns diese Approximation sinnvolle Ergebnisse liefert. Taha [522] hat die Konsequenzen der Nicht-Verfügbarkeit von reellen Zahlen in der Cyber-Welt beschrieben.
- Physikalische Systeme können den sogenannten **Zeno-Effekt** aufweisen. Der Zeno-Effekt kann anhand des Beispiels des springenden Balls eingeführt werden. Angenommen, wir lassen einen Ball von einer bestimmten Höhe auf den Boden fallen. Nachdem wir den Ball loslassen, wird er beginnen zu fallen und dabei durch die Erdbeschleunigung beschleunigt werden. Wenn er auf dem Boden auftrifft, wird er aufgrund der Annahme eines **teilweise elastischen Verhaltens** reflektiert werden, d.h. er bewegt sich in die umgekehrte Richtung. Wir nehmen an, dass die Reflexion eine bestimmte Dämpfung besitzt und dass der Betrag der Geschwindigkeit unmittelbar nach der Reflexion gegenüber der Geschwindigkeit unmittelbar vor der Reflexion um einen Faktor s reduziert ist. Aufgrund dieser Annahme sprechen wir von einem **teilelastischen Stoß**. s heißt **Stoßzahl**. Aufgrund der Dämpfung wird der Ball nach dem Stoß nicht wieder die anfängliche Höhe erreichen. Außerdem wird die Zeit bis zum zweiten Aufprallen auf dem Boden kürzer sein als beim ersten Mal. Dieser Vorgang wird sich wiederholen, mit kürzeren und kürzeren Ab-

ständen zwischen den Stößen. Aufgrund des idealen Modells teilelastischer Stöße wird sich dieser Vorgang aber immer wiederholen. Abb. 1.5 zeigt ein Weg-Zeit-Diagramm (die Höhe als Funktion der Zeit) des teilelastischen Balls.

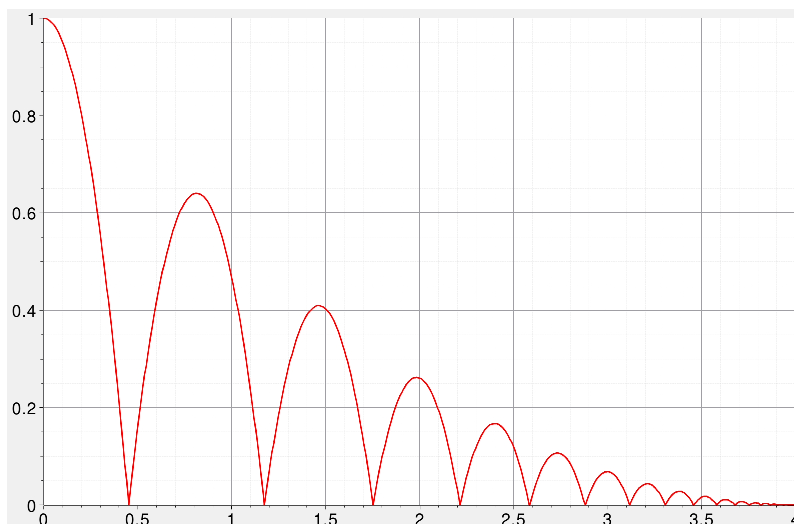


Abb. 1.5 Weg-Zeit-Diagramm eines teilelastischen Balls (© Openmodelica)

Sei nun Δ ein beliebiges Zeitintervall mit einem beliebigen Anfangszeitpunkt. Gibt es eine obere Schranke für die Anzahl der Stöße in diesem Zeitintervall? Nein, eine solche Schranke gibt es nicht, denn im idealen Modell ereignen sich die Stöße immer wieder, aber die Zeitabstände werden immer kürzer. Dies ist ein spezieller Fall des Zeno-Effekts. Ein System zeigt einen **Zeno-Effekt**, wenn es möglich ist, eine unbegrenzte Anzahl von Ereignissen in einem Intervall von endlicher Länge zu haben [404]. Mathematisch gesehen ist das möglich, denn eine unendliche Reihe kann zu einem endlichen Wert konvergieren. In diesem Fall konvergiert die unendliche Zeitreihe der Zeitpunkte der Reflexionen zu einer endlichen Zeit. In Digitalrechnern können wir eine unbegrenzte Zahl von Ereignissen nur approximieren. Weitere Details zu diesem Beispiel werden ab Seite 50 diskutiert werden. Wichtig ist, dass wir in einem Digitalrechner eine unbegrenzte Zahl von Ereignissen nur approximieren können.

– Viele eingebettete Systeme enthalten Regelschleifen wie in Abb. 1.6.

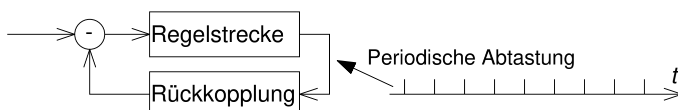


Abb. 1.6 Regelschleife

Die Theorie von Regelschleifen basierte ursprünglich auf analogen, zeitkontinuierlichen Rückkopplungen. Für digitale, zeitdiskrete Rückkopplungen wurde standardmäßig über Jahrzehnte hinweg eine periodische Abtastung eingesetzt und diese funktionierte hinreichend. Dennoch ist die periodische Abtastung nicht der bestmögliche Weg. Wir können Ressourcen einsparen, indem wir die Abtastintervalle in Zeiten relativ konstanter Signale vergrößern. Dies ist die Idee der **adaptiven Abtastung**. Adaptive Abtastung ist Gegenstand aktiver Forschung [210].

- Traditionelle sequentielle Programmiersprachen sind nicht der beste Weg, nebenläufige, zeitbehaftete Systeme zu beschreiben.
- Traditionellerweise liefert die Überprüfung einer korrekten Implementierung einer Spezifikation ein Boolesches Ergebnis: entweder ist die Implementierung korrekt oder nicht. Allerdings sind zwei physikalisch vorhandene Implementierungen nie exakt identisch, sie könnten beispielsweise in ihren Abmessungen geringfügig voneinander abweichen. Dies führt zu einer Unschärfe in der Beurteilung und die Boolesche Verifikation müsste durch eine **unscharfe Verifikation** (engl. *fuzzy verification*) ersetzt werden [446, 184].
- Nach Edward Lee [332] kann die Kombination eines deterministischen physikalischen Modells und eines deterministischen Cyber-Modells möglicherweise ein **nicht-deterministisches Modell** sein. Nicht-deterministische Abtastung kann dafür eine Ursache sein.

Insgesamt beobachten wir eine Diskrepanz zwischen der physischen und der Cyber-Welt. Dementsprechend suchen wir nach geeigneten Modellen für cyber-physikalische Systeme, aber wir können nicht erwarten, dass wir diese Diskrepanz vollkommen beseitigen können.

• Eingebettete Systeme müssen **effizient mit Ressourcen umgehen**. Dazu müssen wir uns der verwendeten Ressourcen gewahr sein. Die folgenden Metriken können verwendet werden, um die Effizienz eingebetteter Systeme zu bewerten:

1. **Energie:** Die elektronische Datenverarbeitung (EDV) benötigt, wie der Name es erwarten lässt, für die Verarbeitung von Daten elektrische Energie. Die Menge der benötigten Energie wird häufig „verbrauchte Energie” genannt. Streng genommen ist das falsch, denn die Energie bleibt insgesamt erhalten. Tatsächlich wandeln wir (nur) elektrische Energie in eine andere Form der Energie, üblicherweise in thermische Energie. Für eingebettete Systeme ist die Verfügbarkeit **elektrischer** Leistung und Energie (als Integral der Leistung über der Zeit) ein entscheidender Faktor. Dies wurde bereits im Rahmen einer niederländischen Technologievorhersage bemerkt: „(elektrische) Leistung wird als die wichtigste Randbedingung bei eingebetteten Systemen betrachtet” [150].
   Warum sollten wir uns der Menge der umgewandelten **Energie gewahr** sein, d.h. uns um diese Menge kümmern (und sie auch so sparsam wie möglich einsetzen)? Dafür gibt es viele Gründe, ein großer Teil davon ist für alle Systemklassen anwendbar, aber es gibt auch weniger relevante Kombinationen von Gründen und Systemklassen, wie in Tabelle 1.1 zu sehen ist.

Tabelle 1.1 Abhängigkeit der Relevanz von Energieeinsparungsgründen von der Systemklasse

	Während der Nutzung relevant?		
Systemklasse	Netzbetrieb	Akkubetrieb	ohne Netz
Beispiel	Fabrik	Laptop	Sensor-Netzwerk
Globale Erwärmung	ja	kaum	nein
Kosten der Energie	ja	kaum	üblicherweise nicht
Höhere Performanz	ja	ja	ja
Laufzeit ohne Netz	nein	ja	ja
Thermische Effekte, Überhitzung	ja	ja	ja
Vermeiden hoher Ströme, Metallwanderung	ja	ja	ja
Kaum Energie vorhanden	nein	selten	ja

Die globale Erwärmung ist natürlich ein sehr wichtiger Grund, um des Verbrauchs an elektrischer Energie gewahr zu sein. Allerdings ist die Menge an verfügbarer Energie bei mobilen Systemen üblicherweise sehr klein und daher ist der Beitrag dieser Systeme zur globalen Erwärmung ebenfalls klein[6].
Die Kosten der Energie sind relevant, wenn die benötigte Menge an Energie teuer ist. Bei Systemen mit Netzbetrieb kann dies aufgrund großer Mengen an Energie passieren. Für Systeme ohne Netzbetrieb sind diese Mengen meist klein, aber es kann Fälle geben, in denen selbst kleine Mengen teuer sind.
Eine größere Performanz (Rechenleistung) verlangt – sofern nicht Fortschritte in der Technologie diese ermöglichen – i.d.R. nach mehr Energie.
Die Laufzeit ohne Stromnetz ist z.B. für Mobiltelefone und Laptops relevant.
Thermische Effekte werden wichtiger, weil zu heiße Systeme nicht benutzbar sind und weil die Zuverlässigkeit von Systemen mit steigenden Temperaturen sinkt. Es kann sogar notwendig sein, Teile von Silizium-*Chips* abzuschalten, d.h. von der Stromversorgung zu trennen, um eine Überhitzung zu vermeiden. Diese Notwendigkeit ist unter dem Begriff ***dark silicon*** bekannt geworden [153]. Daher müssen thermische Effekte ebenfalls betrachtet werden.
Hohe Stromdichten können dazu führen, dass Metalle in *Chips* sich nicht mehr wie Festkörper verhalten, sondern wandern.
In einigen Fällen, wie z.B. räumlich entfernt aufgestellten Sensorknoten, ist Energie eine wirklich sehr knappe Ressource.
Es ist interessant anzusehen, in welchen Fällen ein Grund für effiziente Nutzung der Energie nicht so relevant ist. Bei Systemen, die mit dem Stromnetz verbunden sind, ist Energie i.d.R. keine knappe Ressource. Bei Systemen, die nicht mit dem Stromnetz verbunden sind, ist der Einfluss auf die globale Erwärmung gering, weil die Kapazität von Stromspeichern begrenzt ist.

[6] Dies sei durch ein konkretes Zahlenbeispiel belegt: ein derzeit als groß empfundener Akku eines Mobiltelefons hätte beispielsweise eine Kapazität von 3600 mAh und für die Spannung nehmen wir eine mittlere Spannung von 4 V an. Damit ergibt sich eine Energie von 14,4 Wh. Eine vollständige Ladung entspricht damit dem Verbrauch eines i.d.R. permanent eingeschalteten Internet-Zugangsknotens in ca. 1-2,5 Stunden bzw. eines Fernsehgerätes im Bruchteil einer Stunde.

Die Bedeutung der Energie- und Leistungseffizienz wurde zunächst bei eingebetteten Systemen wahrgenommen und wurde später für allgemeine Rechneranwendungen zur Kenntnis genommen. Die Bedeutung der Energieeffizienz führte zu Initiativen wie der *green computing initiative* [11].

Allgemein sollte nicht vergessen werden, dass es nicht nur auf den Energieverbrauch während des Betriebes ankommt. Vielmehr hat auch die Herstellung von Systemen einen erheblichen Anteil am Energieverbrauch. Es sollte daher der komplette Lebenszyklus eines Produktes im Rahmen eines *Life Cycle Assessments* (LCA) bewertet werden [375]. Dementsprechend kann man den Einfluss auf die Umwelt reduzieren, indem man seltener ein neues Gerät kauft.

2. **Laufzeit-Effizienz:** Eingebettete Systeme sollten die verfügbare Hardwarearchitektur so gut wie möglich ausnutzen. Eine ineffiziente Nutzung von Ausführungszeit (z.B. verschwendete Prozessorzyklen) sollte vermieden werden. Dies impliziert eine Optimierung der Ausführungszeiten über alle Ebenen hinweg, von den Algorithmen bis zu Hardwareimplementierungen.
3. **Codegröße:** Bei manchen eingebetteten Systemen muss der Code für die Ausführung von Software in dem System selbst gespeichert werden. Hierfür kann es aber sehr strenge Randbedingungen geben. Dies gilt insbesondere für *Systems on a Chip* (SoCs), d.h. Systeme, bei denen die gesamte Informationsverarbeitung auf einem *Chip* integriert ist. Wenn auch der Speicher mit auf diesem *Chip* integriert ist, muss er sehr effizient genutzt werden. Beispielsweise kann es medizinische Geräte geben, die im menschlichen Körper implantiert werden. Aufgrund der Beschränkungen der Größe und der Kommunikationsmöglichkeiten müssen diese Geräte sehr kompakt sein.

   Allerdings kann sich die Bedeutung dieser Einschränkung ändern, wenn das dynamische Laden von Code akzeptabel wird oder wenn größere Speicherdichten (gemessen in Bits pro Volumeneinheit) verfügbar werden. *Flash*-Speicher und neue Speichertechnologien können potentiell einen großen Einfluss haben.
4. **Gewicht:** Tragbare Systeme dürfen nicht schwer sein. Ein geringes Gewicht ist oft ein wichtiges Argument für den Kauf eines bestimmten Gerätes.
5. **Kosten:** Eingebettete Systeme für den Massenmarkt, wie Konsumelektronik, werden in hohen Stückzahlen hergestellt. Bei diesen Geräten ist die Wettbewerbsfähigkeit besonders wichtig und daher ist eine effiziente Nutzung der Hardwarekomponenten und des Budgets für Softwareentwicklung erforderlich. Zur Implementierung der benötigten Funktionalität sollte eine minimale Menge an Ressourcen verwendet werden. Wir sollten also die Anforderungen mit der geringsten Menge an Hardwareressourcen und Energie erfüllen können. Um den Energieverbrauch zu reduzieren, sollten Taktfrequenzen und Versorgungsspannungen so gering wie möglich sein. Zudem sollten nur die unbedingt notwendigen Hardwarekomponenten vorhanden sein. Komponenten, welche die maximale Ausführungszeit (engl. *Worst Case Execution Time* (WCET)) nicht verbessern (wie z.B. viele Caches oder virtuelle Speicherverwaltungseinheiten), können in manchen Fällen weggelassen werden.

Um geforderte Effizienzkriterien zu erfüllen, kann der Softwareentwurf nicht unabhängig von der darunterliegenden Hardware erfolgen. Daher müssen in allen Entwurfsschritten sowohl die Software wie auch die Hardware berücksichtigt werden. Dies ist jedoch schwierig, da solche integrierten Ansätze normalerweise nicht gelehrt werden. Die Zusammenarbeit zwischen Elektrotechnik und Informatik hat den dafür erforderlichen Umfang noch nicht erreicht.
Die beste Energieeffizienz würde bei einer Abbildung der Spezifikationen auf Hardware erzielt werden. Hardwareimplementierungen sind aber sehr teuer und erfordern eine lange Entwurfsdauer. Daher bieten Hardwareentwürfe nicht die benötigte Flexibilität, um Entwürfe bei Bedarf anzupassen. Wir benötigen also einen guten Kompromiss zwischen Effizienz und Flexibilität.
- CPS- und IoT-Systeme sammeln sehr häufig große Mengen an Daten. Diese Daten müssen gespeichert werden und sie müssen analysiert werden. Daher gibt es eine starke Verbindung zwischen der Datenanalyse bzw. Maschinellem Lernen und CPS/IoT. Dies ist genau der Gegenstand unseres Sonderforschungsbereichs SFB 876[7]. Der SFB 876 hat seinen Schwerpunkt beim Maschinellen Lernen unter Ressourcen-Beschränkungen.
- **Einfluss jenseits technischer Fragen**: Aufgrund des großen Einflusses auf die Gesellschaft müssen auch rechtliche, ökonomische, soziale, menschliche und umwelttechnische Folgen betrachten werden:
  - Die Integration vieler Komponenten, möglicherweise von verschiedenen Herstellern, lässt die Frage nach der Haftung aufkommen. Diese Frage wird beispielsweise für autonom fahrende Autos diskutiert. Auch muss geklärt werden, wer welche Rechte hat. Es kann nicht akzeptiert werden, dass alle Rechte einer Firma gehören.
  - Die sozialen Folgen der neuen IKT-Geräte müssen auch bedacht werden. Dies hat zur Einführung des Begriffs des *Cyber Physical Social Systems* (CPSS) geführt [140]. Gegenwärtig werden diese Folgen häufig erst lange nach der Verfügbarkeit der Geräte bemerkt, wie das beispielsweise bei Mobiltelefonen der Fall ist.
  - Benutzerfreundliche Mensch-Maschine-Schnittstellen müssen verfügbar sein, um nicht größere Teile der Gesellschaft von der Anwendung auszuschließen.
  - Der Beitrag zur globalen Erwärmung und der entstehende Abfall sollte möglichst gering sein und einen akzeptablen Umfang nicht übersteigen. Dasselbe gilt für den Ressourcen-Verbrauch.
- Reale Echtzeitsysteme sind hochgradig **nebenläufig** (engl. *concurrent*). Die Verwaltung der Nebenläufigkeit ist eine weitere große Herausforderung.
- CPS- und IoT-Systeme bestehen typischerweise aus **heterogenen** Hard- und Softwarekomponenten von verschiedenen Herstellern und sie müssen in einer sich verändernden Umgebung funktionieren. Die Heterogenität führt zu Herausforderungen für die korrekte Kooperation der verschiedenen Komponenten. Es ist nicht ausreichend, nur den Software- oder den Hardwareentwurf zu betrachten. Auf-

[7] Siehe http://www.sfb876.tu-dortmund.de.

grund der Entwurfskomplexität muss ein hierarchischer Ansatz gewählt werden. Reale eingebettete Systeme bestehen aus vielen Komponenten und wir interessieren uns daher für den **kompositionellen** Entwurf. Das bedeutet, wir möchten die Folgen der Integration von Komponenten analysieren [214]. Beispielsweise möchten wir wissen, ob wir ein Navigationssystem zu den Informationsquellen im Auto hinzufügen können, ohne den Kommunikationsbus zu überlasten.
- Der Entwurf von CPS-Systemen erfordert Wissen aus vielen Gebieten. Es ist schwierig, Mitarbeiter zu finden, die in allen relevanten Bereichen ein ausreichendes Wissen haben und schon der Austausch von Wissen zwischen diesen Bereichen ist eine Herausforderung. Eine noch größere Herausforderung ist es, ein Ausbildungsprogramm für den Entwurf von CPS-Systemen zu entwickeln, da es enge Obergrenzen für den studentischen Arbeitsaufwand gibt [379]. Insgesamt wäre es erforderlich, die Wände zwischen den Disziplinen und Fakultäten niederzureißen, oder sie zumindest niedriger zu gestalten.

Eine Liste der Herausforderungen findet sich auch im Bericht über IoT von Sundmaeker et al. [516].

1.4 Gemeinsame Eigenschaften

Zusätzlich zu den aufgeführten Herausforderungen gibt es weitere gemeinsame Eigenschaften von eingebetteten, cyber-physikalischen und IoT-Systemen, unabhängig vom Anwendungsbereich.

- Diese Systeme werden oft durch **Sensoren**, die Informationen über diese Umgebung sammeln, und **Aktuatoren**, welche die Umgebung steuern, mit der physischen Umgebung verbunden. Beim Internet der Dinge werden diese Komponenten mit dem Internet verbunden.

  Definition 1.9: Aktuatoren sind Komponenten, die numerische Werte in physikalische Effekte verwandeln.

- Üblicherweise sind eingebettete Systeme reaktive Systeme, die wie folgt definiert werden können:

  Definition 1.10 (Bergé [566]): „Ein reaktives System ist ein System, das sich in ständiger Interaktion mit seiner Umgebung befindet und mit einer Geschwindigkeit reagiert, die durch diese Umgebung vorgegeben wird."

  Reaktive Systeme modellieren wir als Systeme, die sich in einem Zustand befinden und auf eine Eingabe warten. Für jede Eingabe wird eine bestimmte Berechnung durchgeführt und eine Ausgabe sowie ein neuer Zustand erzeugt. Daher stellen endliche Automaten sehr gute Modelle für solche Systeme dar. Mathematische Funktionen zur Beschreibung der von Algorithmen zu lösenden Probleme wären hier ein unpassendes Modell.

- Eingebettete Systeme sind **in der Lehre und in der öffentlichen Diskussion unterrepräsentiert**. Reale eingebettete Systeme sind komplex. Daher wird eine umfangreiche Ausstattung benötigt, um das Thema realistisch zu unterrichten. Allerdings kann die Lehre des CPS-Entwurfs aufgrund des sichtbaren Einflusses auf das physische Verhalten auch attraktiv sein.
- Diese Systeme sind meist **für eine bestimmte Anwendung ausgelegt**. Prozessoren, die Steuerungssoftware in einem Auto oder einem Zug ausführen, werden i.d.R. nur diese Software ausführen. Es wird niemand versuchen, ein Computerspiel oder eine Tabellenkalkulation auf diesen Prozessoren auszuführen. Dafür gibt es hauptsächlich zwei Gründe:

  1. Die Ausführung zusätzlicher Programme würde diese Systeme weniger verlässlich machen.
  2. Die Ausführung zusätzlicher Programme ist nur möglich, wenn es ungenutzte Ressourcen, wie z.B. Speicher, gibt. In einem effizienten System sollten aber keine ungenutzten Ressourcen vorkommen.

  Allerdings ändert sich die Situation langsam. Beispielsweise zeigt die AUTOSAR-Initiative mehr Dynamik in der Automobilindustrie [27].
- Die meisten eingebetteten Systeme verwenden weder Tastaturen, Mäuse noch große Bildschirme für ihre Benutzerschnittstellen. Stattdessen existiert eine **dedizierte Benutzerschnittstelle**, bestehend aus Schaltern, Lenkrädern, Pedalen usw. Daher nimmt der Benutzer meist nicht wahr, dass in solchen Systemen Informationen verarbeitet werden. Aus diesem Grund wird diese neue Ära der Rechner auch als Ära des **verschwindenden Rechners** bezeichnet.

Die Tabelle in 1.2 stellt einige herausragende Unterschiede zwischen dem Entwurf PC-ähnlicher Systeme und dem Entwurf eingebetteter Systeme dar, die bei der Abbildung von Anwendungen auf Hardwareplattformen relevant sind.

Tabelle 1.2 Unterschiede zwischen PC/Server-ähnlichen und eingebetteten Systemen

	Eingebettet	PC/Server-ähnlich
Architekturen	Häufig heterogen, sehr kompakt	Meist homogen, nicht kompakt (x86 usw.)
x86-Kompatibilität	Weniger wichtig	Sehr wichtig
Festgelegte Architektur?	Selten	Ja
Berechnungs-modell (MoC)	C+div. Modelle (Datenfluss, diskrete Ereignisse, ...)	Meist von Neumann (C, C++, Java)
Optimerungsziele	Mehrere (Energie, Größe, ...)	Durchschnittliche Performanz überwiegt
Sicherheitskritisch?	Möglicherweise	Kaum
Echtzeitsystem?	Häufig	Selten
Anwendungen zur Entwurfszeit bekannt	Die meisten oder alle	Nur einige (z.B. WORD)

Die Kompatibilität mit von x86-PCs her bekannten Befehlssätzen ist für eingebettete Systeme weniger wichtig, da es normalerweise möglich ist, Softwareanwendungen für eine gegebene Architektur neu zu übersetzen. Sequenzielle Programmiersprachen eignen sich nicht gut für die Aufgabe, nebenläufige Echtzeitsysteme zu beschreiben. Hier könnten andere Ansätze zur Modellierung von Anwendungen vorteilhafter sein. Während des Entwurfs eingebetteter/cyber-physikalischer Systeme müssen verschiedene Ziele berücksichtigt werden. Außer der durchschnittlichen Rechenleistung (Performanz) sind hier die größtmögliche Ausführungszeit (WCET), der Energieverbrauch, das Gewicht, die Zuverlässigkeit, die Betriebstemperaturen usw. mögliche Optimierungsziele. Das Einhalten von Echtzeitbedingungen ist für cyber-physikalische Systeme sehr wichtig, aber nur selten für PC-Systeme. Das Einhalten von Zeitbedingungen kann nur dann zur Entwurfszeit verifiziert werden, wenn alle Anwendungen auch zur Entwurfszeit bekannt sind. Zudem muss bekannt sein, welche Anwendungen gleichzeitig ablaufen sollen. So müssen Entwickler beispielsweise sicherstellen, dass eine GPS-Anwendung, ein Telefonanruf und Datenübertragungen gleichzeitig stattfinden können, ohne dass Sprachinformationen verloren gehen. Bei PC-Systemen ist ein solches Wissen so gut wie nie verfügbar und man arbeitet mit *best effort*-Ansätzen, d.h. man versucht, Software möglichst schnell ablaufen zu lassen, ohne ein bestimmtes Verhalten zu garantieren.

Warum versuchen wir, alle verschiedenen Arten eingebetteter Systeme in einem Buch zu behandeln? Der Grund dafür ist, dass die informationsverarbeitenden Komponenten in all diesen Systemen über viele gemeinsame Eigenschaften verfügen, auch wenn die Systeme selbst physisch sehr unterschiedlich sind.

Nicht jedes eingebettete System wird über alle der oben genannten Eigenschaften verfügen. Wir können den Begriff „eingebettetes System" auch wie folgt definieren:

Definition 1.11: Informationsverarbeitende Systeme, welche die meisten der oben genannten Eigenschaften erfüllen, werden **eingebettete Systeme** genannt.

Diese Definition beinhaltet eine gewisse Unschärfe. Es scheint aber weder notwendig noch möglich, diese Unschärfe zu beseitigen.

1.5 Lehrplan-Integration von Eingebetteten Systemen, CPS und IoT

Leider werden eingebettete Systeme im *Computer Science Curriculum*, veröffentlicht im Jahr 2013 von *ACM* und der *IEEE* Computer Society [10], so gut wie nicht berücksichtigt. Die wachsende Bedeutung eingebetteter Systeme führt aber zu einem gesteigerten Bedarf an Ausbildung in diesem Bereich. Diese Ausbildung soll dabei helfen, die Beschränkungen aktuell verfügbarer Entwurfsmethoden zu überwinden. Eine Übersicht über Anforderungen und Ansätze zur Lehre im Bereich CPS wurden von den *National Academies of Sciences, Engineering and Medicine* [410] sowie von Marwedel et al. [379] sowie publiziert. Es gibt beispielsweise Bedarf an besse-

ren CPS-Modellen, Spezifikationssprachen, Entwurfstechniken für unterschiedliche Zielkriterien, Synthese- und Verifikationswerkzeugen, Echtzeitbetriebssystemen sowie anderer Systemsoftware. Dieses Buch soll helfen, wesentliche Fragestellungen zu vermitteln und ein Sprungbrett für weitere Forschung in diesem Gebiet sein.

Zusätzliche Informationen zum Buch gibt es unter folgender Adresse:

http://ls12-www.cs.tu-dortmund.de/~marwedel/es-book/

Diese Seite beinhaltet Verweise auf Foliensätze, Simulationswerkzeuge, Fehlerkorrekturen, Vorlesungsvideos und weiteres zugehöriges Material. Videos sind direkt verfügbar von

https://www.youtube.com/user/cyphysystems

Leser und Leserinnen, die Fehler entdecken oder Vorschläge zur Verbesserung des Buchs machen möchten, wenden sich bitte per E-Mail an:

peter.marwedel@tu-dortmund.de

Aufgrund der Verfügbarkeit dieses Buches und der Videos ist es zu empfehlen, Lehre nach dem *flipped classroom*-Konzept [376] zu erproben. Bei diesem Konzept werden die Funktionen von Stoffvorstellungen in der Hochschule und Vertiefungsphasen zu Hause weitgehend vertauscht: Ein erstes Kennenlernen des Stoffes findet anhand der Videos oder des Buches zu Hause statt, ein Vertiefen erfolgt in Gruppenarbeit während der Anwesenheit in der Hochschule. Auf diese Weise werden die Fähigkeiten zur Problemlösung, zum *Teamwork* und die sozialen Kompetenzen gestärkt. Auch wird die Verfügbarkeit des Internets genutzt, um Lehrmethoden für eingeschriebene Studierende zu verbessern. In Pandemiephasen, wie der von COVID-19, ist es vorteilhaft, dass Anwesenheitsphasen an der Hochschule auf das notwendige Minimum beschränkt werden können. Vom *flipped-classroom*-Konzept zu unterscheiden sind *Massive Open Online Courses* (MOOCs), die v.a. entfernt lebende Studierende ansprechen sollen. Zu lösende Aufgaben können entweder diesem Buch oder anderen Büchern (wie z.B. [593, 81, 174]) entnommen werden.

Beim *flipped classroom*-Konzept können Laborphasen vollständig genutzt werden, um praktische Erfahrungen mit CPS zu bekommen. Zu diesem Zweck sollte ein Kurs, der dieses Buch benutzt, um ein spannendes Labor ergänzt werden, in dem beispielsweise kleine Roboter (wie Lego *Mindstorms*™) oder Mikrocontroller (wie *Raspberry Pi*, *Arduino* oder *Odroid*) benutzt werden. Für die handelsüblichen Mikrocontroller steht meist auch Lehrmaterial zur Verfügung. Eine andere Möglichkeit besteht in der Nutzung von Werkzeugen zur Simulation von endlichen Automaten. Empfohlen wird eine Ergänzung um eine Lehrveranstaltung zur Datenanalyse bzw. zum Maschinellen Lernen [205, 188, 559, 453].

Voraussetzungen

Das Buch setzt ein grundlegendes Verständnis verschiedener Bereiche voraus:

- Programmierung von Rechnern (mit Grundlagen des *Software Engineerings*) einschließlich erster Erfahrungen mit der Programmierung von Mikrocontrollern,
- Algorithmen (Graphenalgorithmen und Optimierungsalgorithmen wie *Branch-and-Bound*) einschließlich des Konzeptes der NP-Vollständigkeit,
- Rechnerorganisation, z.B. auf der Ebene des einführenden Buchs von J.L. Hennessy und D.A. Patterson [213] einschließlich endlicher Automaten,
- Grundlagen von Betriebssystemen (BS),
- Rechnernetz-Grundlagen (wichtig für das Internet der Dinge),
- grundlegende mathematische Konzepte (z.B. Tupel, Integrale, lineare Algebra),
- elektrotechnische (ET) Grundlagen sowie Digitalschaltungen wie Gatter und Register.

Diese Voraussetzungen lassen sich in verschiedene Kurse einteilen, wie in Abb. 1.7 dargestellt. Fehlende Grundkenntnisse im Bereich elektrischer Netzwerke, von

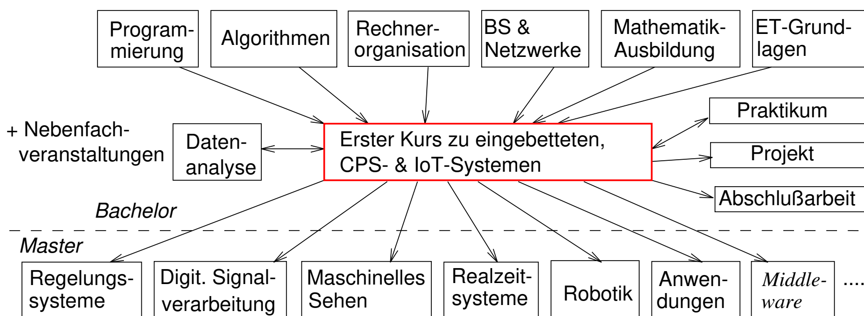


Abb. 1.7 Positionierung der einzelnen Themen dieses Buches

Operationsverstärkern, von Speicherverwaltung und von Ganzzahliger Linearer Programmierung können durch Lesen der Anhänge dieses Buches teilweise ausgeglichen werden. Kenntnisse in Statistik und Fourier-Reihen sind nützlich.

Empfohlene zusätzliche Lehrveranstaltungen

Das Buch sollte durch aufbauende Kurse ergänzt werden, die speziellere Kenntnisse in einigen der folgenden Gebiete vermitteln (siehe die untere Zeile von Abb. 1.7)[8]:

- Regelungstechnik,
- Digitale Signalverarbeitung,
- Bilderkennung,

[8] Die Aufteilung in grundlegende und weiterführende Lehrveranstaltungen kann sich dabei zwischen einzelnen Universitäten unterscheiden.

- Echtzeitsysteme, Echtzeitbetriebssysteme und *Scheduling*,
- Robotik,
- Anwendungsgebiete wie Telekommunikation, Kraftfahrzeuge, medizinische Geräte und Gebäudeautomatisierung,
- *Middleware*,
- Spezifikationssprachen und -modelle für eingebettete Systeme,
- Sensoren und Aktuatoren,
- Verlässlichkeit von Rechnersystemen,
- energiesparende Entwurfstechniken,
- physikalische Aspekte von CPS,
- rechnergestützte Entwurfswerkzeuge für anwendungsspezifische Hardware,
- formale Verifikation von Hardwaresystemen,
- Test von Hardware- und Softwaresystemen,
- Leistungsbewertung von Rechensystemen,
- *ubiquitous computing*,
- Kommunikationstechniken für das Internet der Dinge (IoT),
- gesellschaftliche Bedeutung und Wirkung eingebetteter, cyber-physikalischer und IoT-Systeme sowie
- rechtliche Aspekte derselben Systeme.

1.6 Entwurfsflüsse

Der Entwurf der betrachteten Systeme stellt eine recht komplexe Aufgabe dar, die in eine Reihe von Unteraufgaben aufgeteilt werden muss. Diese Unteraufgaben müssen nacheinander und einige müssen mehrfach ausgeführt werden.

Der Entwurfsinformationsfluss beginnt mit Ideen im Kopf von Personen. Diese Ideen sollten bereits Wissen über den Anwendungsbereich beinhalten. Die Ideen müssen in einer Entwurfsspezifikation festgehalten werden. Zudem sind üblicherweise Hardware- und Systemsoftwarekomponenten vorhanden, diese sollten möglichst wiederverwendet werden (siehe Abb. 1.8). Bei diesem Diagramm (wie auch in weite-

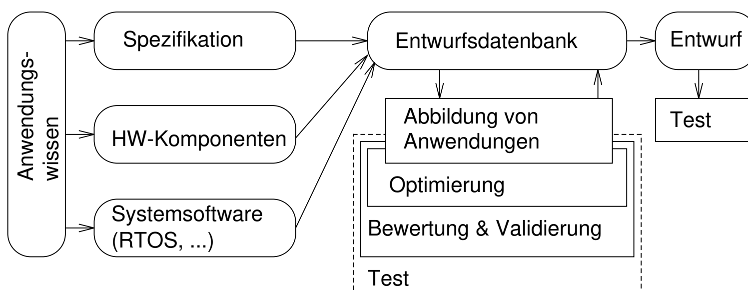


Abb. 1.8 Vereinfachter Entwurfsfluss

ren ähnlichen Diagrammen in diesem Buch) verwenden wir **Rechtecke mit abgerundeten Ecken zur Darstellung gespeicherter Informationen** und **herkömmliche Rechtecke für Transformationen von Informationen**. Insbesondere kann die Speicherung zentral in einer **Entwurfsdatenbank** erfolgen. Diese Datenbank ermöglicht es, einen Überblick über die Entwurfsmodelle zu behalten. Normalerweise sollten die Entwurfsdaten einer Versionsverwaltung unterliegen, die z.B. mittels der Werkzeuge CVS [87], SVN [108] oder git (siehe https://www.git-scm.com) verfügbar ist. Eine gute Entwurfsdatenbank sollte auch eine Schnittstelle zur Entwurfsverwaltung besitzen, die es erlaubt, einen Überblick über die Anwendbarkeit von Entwurfswerkzeugen und -folgen zu erhalten. Diese Schnittstelle kann als graphische Benutzerschnittstelle (engl. *Graphical User Interface* (GUI)) realisiert werden, um die Benutzung zu vereinfachen. Die Entwurfsdatenbank und die Benutzerschnittstelle können zu einer **integrierten Entwicklungsumgebung** (engl. *Integrated Development Environment* (IDE)) zusammengefasst werden, die auch *design framework* genannt wird (siehe z.B. [346]). Eine integrierte Entwicklungsumgebung verwaltet die Abhängigkeiten zwischen den Werkzeugen und der Entwurfsinformation.

Die Verwendung der Datenbank ermöglicht es, Entwurfsentscheidungen in einem iterativen Ablauf zu treffen. Für jeden Schritt muss die Information über das Entwurfsmodell abgerufen und anschließend bewertet werden.

In den einzelnen Entwurfsdurchläufen werden **Anwendungen** auf Ausführungsplattformen **abgebildet**, zudem werden neue (partielle) Entwurfsinformationen erzeugt. Dies beinhaltet die Abbildung von Operationen auf nebenläufige Tasks, die Abbildung von Operationen auf Hardware oder Software (Hardware/Software-Partitionierung genannt), die Übersetzung und das *Scheduling*.

Entwürfe sollten in Hinblick auf verschiedene Ziele wie Rechenleistung (Performanz), Verlässlichkeit, Energieverbrauch, thermisches Verhalten usw. **bewertet** (**evaluiert**) werden. Der aktuelle Stand der Technik garantiert nicht, dass irgendeiner dieser Entwurfsschritte korrekt ist. Daher ist es auch erforderlich, einen Entwurf zu **validieren**. Die Validierung umfasst den Abgleich von Zwischenversionen und der endgültigen Version von Entwurfsbeschreibungen mit anderen Beschreibungen. Jeder neue Entwurf sollte bewertet und validiert werden.

Da die Effizienz eingebetteter Systeme besonders wichtig ist, sind **Optimierungen** erforderlich. Dabei ist eine Vielzahl von Optimierungen verfügbar, beispielsweise *High-Level*-Transformationen (wie z.B. erweiterte Schleifentransformationen) und Energieoptimierungen.

Wenn Überlegungen zur Testbarkeit schon in frühen Entwurfsphasen berücksichtigt werden sollen, müssen Tests in die dargestellten Entwurfsiterationen integriert werden. In Abb. 1.8 wurde die Testerzeugung als optionaler Schritt während der Entwurfsiterationen hinzugefügt (dargestellt durch das gestrichelte Rechteck). Wenn die Testerzeugung nicht Bestandteil der einzelnen Entwurfsiterationen ist, dann muss sie nach Fertigstellung des Entwurfs durchgeführt werden. Am Ende jedes einzelnen Schrittes sollte die Datenbank aktualisiert werden.

Die Details des Informationsflusses zwischen Datenbank, Anwendungsabbildung, Bewertung, Validierung, Optimierung, Testbarkeitsüberlegungen und der Speicherung von Entwurfsinformationen können durchaus abweichen. Abhängig von der

konkret verwendeten Entwurfsmethode können diese Aktionen auf vielfältige Weise miteinander verknüpft sein. Dieses Buch gibt einen breiten Überblick über den Entwurf eingebetteter Systeme und ist nicht auf bestimmte Entwurfsflüsse und Werkzeuge festgelegt. Daher haben wir keine feste Liste von Entwurfsschritten vorgegeben. Wir können die in Abb. 1.8 enthaltenen Schleifen für jede Entwurfsumgebung „abrollen" und einzelnen Entwurfsschritten Namen zuordnen.

Dies führt bei Einsatz von SpecC [173] zu dem in Abb. 1.9 dargestellten Entwurfsfluss. Der gezeigte Fall beinhaltet eine bestimmte Menge an Entwurfsschritten, wie

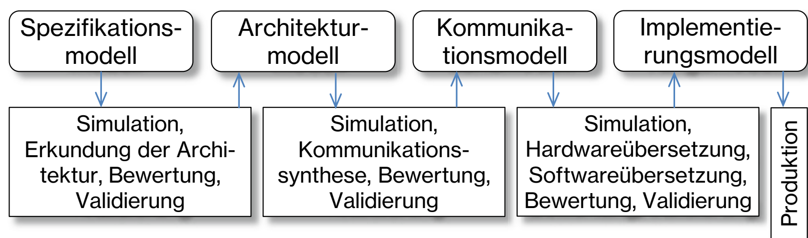


Abb. 1.9 Entwurfsfluss für die SpecC-Werkzeuge (vereinfacht)

Architekturerkundung, Kommunikationssynthese und Software- wie Hardwareübersetzung. Die genaue Bedeutung dieser Begriffe ist für dieses Buch nicht relevant. In Abb. 1.9 sind Validierung und Bewertung für jeden der Schritte explizit als Bestandteil eines größeren Schrittes dargestellt.

Eine weitere Instanz von Abb. 1.8 ist in Abb. 1.10 dargestellt. Dies ist der Entwurfsfluss des V-Modells [550], der für viele deutsche IT-Projekte, speziell im öffentlichen Sektor und darüber hinaus, verbindlich ist. Abb. 1.10 zeigt die durch-

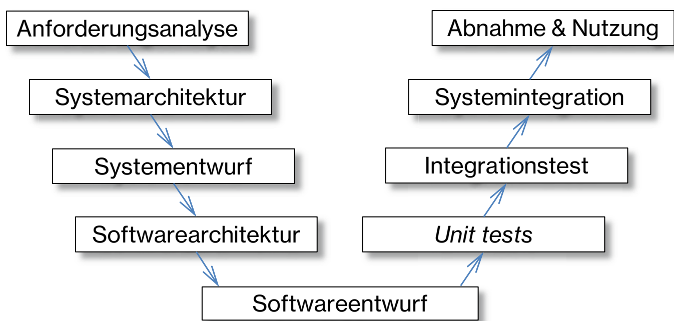


Abb. 1.10 Entwurfsfluss für das V-Modell

zuführenden Schritte im Einzelnen. Diese Schritte entsprechen bestimmten Phasen des Software-Entwicklungsprozesses (auch hier ist die genaue Bedeutung im Rah-

men dieses Buches nicht relevant). Dieses Diagramm fasst Entwurfsentscheidungen, Bewertung und Validierung in einem einzelnen Schritt zusammen. Anwendungswissen, Systemsoftware und Systemhardware werden nicht explizit dargestellt. Das V-Modell beinhaltet auch ein Modell der Integrations- und Testphase (rechter Zweig des Diagramms). Damit wird das Testen mit in die Integrationsphase einbezogen. Das dargestellte Modell entspricht dem V-Modell Version „97". Das aktuellere V-Modell XT ermöglicht eine verallgemeinerte Menge an Entwurfsschritten. Diese Erweiterung passt sehr gut zu unserer Interpretation von Entwurfsflüssen in Abb. 1.8. Andere iterative Ansätze sind das **Wasserfallmodell** und das **Spiralmodell**. Weitere Informationen zum *Software-Engineering* für eingebettete Systeme finden sich in dem Buch von J. Cooling [109].

Unser allgemeiner Entwurfsfluss passt auch zu Flussmodellen, die im Hardwareentwurf zum Einsatz kommen. Ein weit verbreitetes Modell ist das Y-Diagramm (engl. *Y-chart*) von Gajski und Kuhn [171] (siehe Abb. 1.11). Die Autoren un-

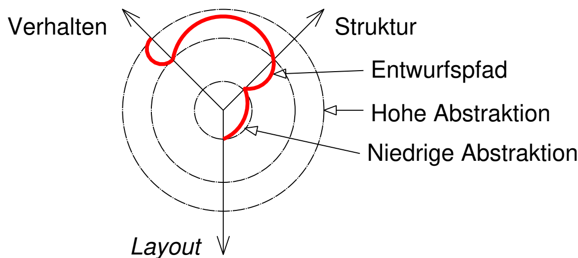


Abb. 1.11 Gajskis Y-Diagramm und Entwurfspfad (rot, fett)

terscheiden drei Dimensionen von Entwurfsinformationen: Verhalten, Struktur und *Layout*. Die erste Dimension stellt ausschließlich das Verhalten dar. Ein Modell auf hoher Abstraktionsebene würde lediglich das Gesamtverhalten beschreiben, wogegen detailliertere Modelle das Verhalten einzelner Komponenten beschreiben würden. Modelle der zweiten Dimension beinhalten Strukturinformationen wie z.B. Informationen über Hardwarekomponenten. In dieser Dimension könnten Beschreibungen auf hoher Abstraktionsebene Prozessoren entsprechen, Beschreibungen auf unterster Ebene entsprächen Transistoren. Die dritte Dimension stellt die geometrische *Layout*-Information eines *Chips* dar. Entwurfspfade beginnen meist mit einer grobgranularen Verhaltensbeschreibung und enden mit einer feingranularen geometrischen Beschreibung. Entlang dieses Pfades entspricht jeder Schritt einer Iteration unseres generischen Entwurfsflussmodells. Im Beispiel von Abb. 1.11 erfolgt zu Beginn eine Verfeinerung im Verhaltensbereich. Der zweite Entwurfsschritt bildet das Verhalten auf Strukturelemente ab usw. Schließlich entsteht eine detaillierte geometrische Beschreibung des *Chip-Layouts*.

Diese drei Beispiele zeigen, dass der iterative Fluss von Abb. 1.8 in einer Reihe von Entwurfsflüssen verwendet wird. Die Beschaffenheit der einzelnen Iterationen aus

Abb. 1.8 wird dabei oft unterschiedlich betrachtet. Idealerweise würden wir zunächst die Eigenschaften unseres Systems beschreiben wollen und dann den Rest von einem geeigneten leistungsfähigen Werkzeug erledigen lassen. Eine solche automatische Erzeugung von Entwurfsdetails wird **Synthese** genannt.

Definition 1.12 (Marwedel [371]): „**Synthese** ist der Vorgang, die Beschreibung eines Systems in Form von Komponenten auf einer niederen Abstraktionsebene aus einer Beschreibung des erwarteten Verhaltens auf einer höheren Ebene erzeugen zu lassen."

Bei einer automatischen Synthese soll dieser Vorgang automatisch ablaufen. Manuelle Entwurfsschritte können vermieden werden, wenn automatische Synthese erfolgreich ist. Das Ziel der automatischen Synthese wurde von Gajski [172] im „*describe-and-synthesize*"-Paradigma verfolgt. Dieses Paradigma steht im Gegensatz zu dem weiter verbreiteten „*specify-explore-refine*"-Paradigma, auch bekannt als „entwerfe und simuliere". Der zweite Begriff betont, dass der manuelle Entwurf meist mit Simulation kombiniert werden muss, um beispielsweise Entwurfsfehler zu finden. Simulation ist hier wichtiger als bei der automatischen Synthese.

1.7 Struktur dieses Buches

Die Struktur dieses Buches folgt dem oben gezeigten Entwurfsfluss. Kapitel 2 gibt einen Überblick über Spezifikationstechniken, -sprachen und -modelle. Wichtige Hardwarekomponenten eingebetteter Systeme und die Schnittstelle zwischen der physischen Umgebung und der Informationsverarbeitung werden im Kapitel 3 vorgestellt. Kapitel 4 behandelt Systemsoftwarekomponenten, insbesondere eingebettete Betriebssysteme. Kapitel 5 beschreibt die Grundzüge der Bewertung und Validierung eingebetteter Systeme. Die Abbildung von Anwendungen auf Ausführungsplattformen ist einer der wichtigsten Schritte im Entwurfsprozess eingebetteter Systeme. Dabei ist das *Scheduling* ein wesentlicher Teil einer solchen Abbildung. Dafür werden Standard-Techniken im Kapitel 6 beschrieben. Um effiziente Entwürfe zu erzeugen, werden viele Optimierungstechniken benötigt. Eine Auswahl von Techniken aus der großen Menge verfügbarer Optimierungen wird im Kapitel 7 vorgestellt. Kapitel 8 enthält eine kurze Einführung in den Test kombinierter Hardware/Software-Systeme. Der Anhang beinhaltet eine Beschreibung einer Standard-Optimierungstechnik, einige Grundlagen zum Verständnis einer der Schaltungen aus Kapitel 3 und Grundwissen zur Speicherorganisation.

In einigen Fällen kann es erforderlich sein, spezielle Hardware für eine bestimmte Anwendung zu entwerfen oder eine Prozessorarchitektur entsprechend zu optimieren. Dieses Buch befasst sich jedoch nicht mit dem Hardwareentwurf. Coussy and Morawiec [113] geben einen Überblick über *High-Level*-Hardwaresynthese-Techniken.

Der Inhalt dieses Buchs unterscheidet sich vom Inhalt der meisten anderen Bücher zum Entwurf eingebetteter oder cyber-physikalischer Systeme. Der Schwer-

punkt vieler solcher Bücher lag früher auf der Verwendung von Mikrocontrollern, deren Speicher, Ein/Ausgabegeräten und der *Interrupt*-Struktur. Es gibt viele derartige Bücher [176, 37, 175, 177, 280, 318, 426]. Wir sind der Auffassung, dass es die steigende Komplexität eingebetteter Systeme erfordert, diesen Schwerpunkt so zu erweitern, dass zumindest unterschiedliche Spezifikationsansätze, Grundlagen von Hardwarekomponenten, die Abbildung von Anwendungen auf Ausführungsplattformen sowie die Bewertung, Validierung und Optimierungstechniken vorgestellt werden. Im vorliegenden Buch betrachten wir alle diese Bereiche. Das Ziel ist es, Studierenden eine Einführung zu eingebetteten und cyber-physikalischen Systemen zu geben, die sie in die Lage versetzt, die verschiedenen Themenbereiche einschätzen zu können.

Für weiterführende Informationen möchten wir auf eine Reihe von Quellen verweisen, von denen einige auch bei der Vorbereitung dieses Buches zum Einsatz kamen:

- Symposien mit Schwerpunkt eingebettete/cyber-physikalische Systeme sind die *Embedded Systems Week* (siehe http://www.esweek.org) und die *Cyber-Physical Systems Week* (siehe http://www.cpsweek.org).
- Die Webseite der amerikanischen virtuellen CPS-Organisation liefert zahlreiche Hinweise auf laufende Arbeiten und deren Ergebnisse [115].
- Die Webseiten einer Interessengruppe der ACM [9] befassen sich mit eingebetteten Systemen.
- Die Webseiten des europäischen Exzellenznetzwerks für eingebettete Systeme und Echtzeitsysteme [24] beinhalten viele Links zum Thema.
- Ein Buch von Edward Lee et al. befasst sich auch mit physikalischen Aspekten von cyber-physikalischen Systemen [335].
- Die Workshopreihe *Workshop on Embedded and Cyber-Physical Systems Education* (WESE) behandelt aktuelle Ansätze für die Lehre zu eingebetteten Systemen; Ergebnisse des Workshops im Jahr 2018 sind in [89] zu finden. Die Lehre ist auch Gegenstand des ersten und einzigen Workshops über Lehre im Bereich CPS [425].
- Andere Informationsquellen zu eingebetteten Systemen sind Bücher von Laplante [323], Vahid [552], die ARTIST Roadmap [63], das „*Embedded Systems Handbook*" [614] sowie Bücher von Gajski et al. [174] und Popovici et al. [457].
- Eine große Anzahl von Quellen befasst sich mit Spezifikationssprachen. Zu diesen zählen ältere Bücher von Young [609], Burns and Wellings [79], Bergé [566] und de Micheli [124]. Weiterhin ist eine große Menge an Informationen über neue Sprachen wie SystemC [408], SpecC [173] und Java [573, 131, 71] verfügbar.
- Echtzeit-*Scheduling* wird umfassend in den Büchern von Buttazzo [81], von Krishna und Shin [311] sowie von Baruah et al. [38] behandelt.
- Ansätze für den Entwurf und die Verwendung von Echtzeit-Betriebssystemen (RTOS) werden in einem Buch von Kopetz vorgestellt [304].
- Robotik ist ein eng mit eingebetteten und cyber-physikalischen Systemen verbundenes Gebiet. Wir empfehlen das Buch von Siciliano et al. [487].
- Es gibt Bücher und Artikel, die auf das Internet der Dinge fokussiert sind [185, 193, 192].

- Sprachen und Verifikation sind der Schwerpunkt eines Buches von Haubelt und Teich [207].

1.8 Aufgaben

Die folgenden Aufgaben sollten entweder zu Hause oder während einer Anwesenheitsphase nach dem *flipped classroom*-Konzept [376] bearbeitet werden:

1.1: Nennen Sie einige der Definitionen des Begriffs „eingebettetes System”!

1.2: Definieren Sie den Begriff „cyber-physikalisches System”! Sehen Sie einen Unterschied zwischen den Begriffen „eingebettetes System” und „cyber-physikalisches System”?

1.3: Was ist das Internet der Dinge (IoT)?

1.4: Was ist das Ziel der Initiative „Industrie 4.0”?

1.5: Auf welche Weise deckt dieses Buch den Entwurf von CPS- und IoT-Systemen ab?

1.6: In welchen Anwendungsbereichen sehen Sie Potential für den Einsatz von CPS- und IoT-Systemen? Wo erwarten Sie große Veränderungen durch die IT?

1.7: Warum sind eingebettete Systeme und cyber-physikalische Systeme wichtig?

1.8: Welche Herausforderungen müssen wir bewältigen, wenn wir das Potential von eingebetteten und cyber-physikalischen Systemen ausnutzen wollen?

1.9: Was ist eine harte Zeitschranke? Was ist eine weiche Zeitschranke?

1.10: Was ist der Zeno-Effekt?

1.11: Was ist adaptives Abtasten?

1.12: Nach welchen Kriterien können wir eingebettete Systeme bewerten?

1.13: Aus welchen Gründen und für welche Systeme betrachten wir die Energieeffizienz?

1.14: Was sind die Hauptunterschiede zwischen Anwendungen auf PCs und Anwendungen in eingebetteten Systemen?

1.15: Was ist ein reaktives System?

1.16: Auf welcher Webseite gibt es ergänzendes Material zu diesem Buch?

1.17: Vergleichen Sie das Curriculum an Ihrer Hochschule mit dem Programm, das hier in der Einleitung angegeben ist! Welche Voraussetzungen fehlen an Ihrer Hochschule? Welche fortgeschrittenen Kurse sind verfügbar?

1.18: Was ist das *flipped classroom*-Konzept?

1.19: Wie modellieren wir Entwurfsflüsse?

1.20: Was ist das V-Modell?

1.21: Wie definieren Sie „Synthese"?

Open Access Dieses Kapitel wird unter der Creative Commons Namensnennung 4.0 International Lizenz (http://creativecommons.org/licenses/by/4.0/deed.de) veröffentlicht, welche die Nutzung, Vervielfältigung, Bearbeitung, Verbreitung und Wiedergabe in jeglichem Medium und Format erlaubt, sofern Sie den/die ursprünglichen Autor(en) und die Quelle ordnungsgemäß nennen, einen Link zur Creative Commons Lizenz beifügen und angeben, ob Änderungen vorgenommen wurden.

Die in diesem Kapitel enthaltenen Bilder und sonstiges Drittmaterial unterliegen ebenfalls der genannten Creative Commons Lizenz, sofern sich aus der Abbildungslegende nichts anderes ergibt. Sofern das betreffende Material nicht unter der genannten Creative Commons Lizenz steht und die betreffende Handlung nicht nach gesetzlichen Vorschriften erlaubt ist, ist für die oben aufgeführten Weiterverwendungen des Materials die Einwilligung des jeweiligen Rechteinhabers einzuholen.

Kapitel 2
Spezifikation und Modellierung

Wie können wir das System, welches wir entwerfen wollen, beschreiben und wie können wir partielle Entwürfe darstellen? Eine Antwort auf diese Fragen gibt dieses Kapitel, in dem wir Modelle und Beschreibungstechniken vorstellen, mit denen sowohl die Spezifikation wie auch Zwischenschritte dargestellt werden können. Als erstes werden wir Anforderungen an Modellierungstechniken erfassen. Danach werden wir beliebte Berechungsmodelle zusammen mit zugehörigen Sprachen vorstellen. Dazu gehören Modelle für frühe Entwurfsphasen, Automaten-basierte Modelle, Datenfluss, Petrinetze, diskrete Ereignismodelle, von-Neumann-Sprachen und Abstraktionsebenen für die Hardware-Modellierung. Schließlich werden wir verschiedene Berechnungsmodelle vergleichen und Übungsaufgaben vorstellen.

2.1 Anforderungen

In Übereinstimmung mit dem vereinfachten Entwurfsfluss aus Abb. 1.8 werden wir zunächst Anforderungen und Ansätze für die Spezifikation eingebetteter und cyber-physikalischer Systeme präsentieren. Spezifikationen solcher Systeme stellen **Modelle** des zu entwerfenden Systems bereit. Das führt zu der Frage, was ist eigentlich ein Modell? Eine mögliche Definition stammt von Jantsch.

Definition 2.1 (Jantsch [269]): „Ein Modell ist die Vereinfachung einer anderen Einheit, die ein physischer Gegenstand oder ein anderes Modell sein kann. Das Modell enthält genau diejenigen Merkmale und Eigenschaften der modellierten Einheit, die für die gegebene Aufgabe relevant sind. Ein Modell ist minimal im Hinblick auf eine Aufgabe, wenn es keine anderen Eigenschaften enthält als jene, die für die Aufgabe relevant sind".

Modelle werden in **Sprachen** beschrieben. Dabei sollten Sprachen die folgenden Eigenschaften darstellen können[1]:

[1] Diese Liste verwendet Informationen aus den Büchern von Burns et al. [79], Bergé et al. [566] und Gajski et al. [172]

© Der/die Autor(en) 2021

P. Marwedel, *Eingebettete Systeme*, https://doi.org/10.1007/978-3-658-33437-6_2

- **Hierarchie:** Menschen sind im Allgemeinen nicht dazu in der Lage, Systeme mit vielen Objekten (wie z.B. Zustände oder Komponenten) zu verstehen, in denen die Objekte in komplexen Zusammenhängen miteinander stehen. Um ein reales System zu beschreiben, benötigt man mehr Objekte als ein Mensch erfassen kann. Die Einführung von Hierarchieebenen (in Verbindung mit **Abstraktion**) ist einer der besten Wege, dieses Problem zu lösen. Hierarchie kann so eingesetzt werden, dass ein Mensch immer nur eine kleine Anzahl von Objekten auf einmal überschauen muss.
  Es gibt zwei Arten von Hierarchien:
  - **Verhaltenshierarchien:** Verhaltenshierarchien enthalten Objekte, die notwendig sind, um das Verhalten des Gesamtsystems zu beschreiben. Beispiele hierfür sind Zustände, Ereignisse und Ausgabesignale.
  - **Strukturelle Hierarchien:** Strukturelle Hierarchien beschreiben, wie das Gesamtsystem aus einzelnen physischen Komponenten zusammengesetzt ist.
    Eingebettete Systeme können z.B. aus Prozessoren, Speichern, Aktuatoren und Sensoren bestehen. Prozessoren wiederum enthalten Register, Multiplexer und Addierer. Multiplexer bestehen aus Gattern.
- **Komponentenbasierter Entwurf** [489]**:** Es muss „einfach” sein, das Verhalten eines Systems vom Verhalten seiner Komponenten abzuleiten. Wenn zwei Komponenten miteinander verbunden werden, sollte das sich ergebende neue Verhalten vorhersagbar sein. Wenn wir beispielsweise eine zusätzliche Komponente wie ein Navigationssystem in ein Auto integrieren möchten, sollte der Einfluss dieser zusätzlichen Komponente auf das Verhalten des Gesamtsystems (unter Berücksichtigung von Bussen usw.) vorhersagbar bleiben.
- **Nebenläufigkeit:** Viele praxisrelevante Systeme sind verteilte, nebenläufige Systeme, die aus mehreren Komponenten bestehen. Daher muss Nebenläufigkeit oder Parallelität einfach zu spezifizieren sein. Leider kann der Mensch nebenläufige Vorgänge nur schwer verstehen. Daher sind viele der Probleme, die bei realen Systemen auftreten, eine Folge von mangelndem Verständnis möglicher Abläufe in parallelen Systemen.
- **Synchronisation und Kommunikation:** Komponenten eines Systems müssen miteinander kommunizieren und die Verwendung von Ressourcen synchronisieren können. Ohne Kommunikation wäre eine Kooperation zwischen Komponenten nicht realisierbar und jede Komponente müsste für sich alleine verwendet werden. Weiterhin muss es eine Möglichkeit geben, die Benutzung von Ressourcen auszuhandeln. So ist es zum Beispiel häufig notwendig, gegenseitigen Ausschluss zu realisieren.
- **Zeitverhalten:** Viele eingebettete Systeme müssen Echtzeitanforderungen erfüllen. Explizite Zeitbedingungen sind daher eine der wichtigsten Eigenschaften eingebetteter Systeme, was durch den Begriff „cyber-physikalische Systeme” noch besser zum Ausdruck kommt. Zeit ist eine der wichtigsten Dimensionen der Physik. Daher **müssen** Zeitbedingungen in der Spezifikation eingebetteter und cyber-physikalischer Systeme erfasst werden.

Die Standardtheorien der Informatik modellieren Zeit allerdings nur auf eine sehr abstrakte Art. Die $\mathcal{O}$-Notation[2] ist ein Beispiel dafür. Sie gibt nur Wachstumsraten von Funktionen wieder und wird häufig genutzt, um die Laufzeiten von Algorithmen zu modellieren. Dabei ist sie aber nicht dazu in der Lage, echte Ausführungszeiten zu ermitteln. Physikalische Größen besitzen Einheiten, die $\mathcal{O}$-Notation jedoch nicht. Damit kann sie nicht zwischen Femtosekunden und Jahrhunderten unterscheiden. Ähnlich verhält es sich mit Eigenschaften der Terminierung von Prozessen. Es gibt Beweise, die zeigen, dass ein bestimmter Algorithmus *irgendwann einmal* terminiert. Bei Echtzeitsystemen hingegen müssen wir nachweisen, dass ein Algorithmus innerhalb einer vorgegebenen Zeitspanne terminiert, aber der Algorithmus als Ganzes muss evtl. ausgeführt werden, bis die Spannungsversorgung ausgeschaltet wird.
Burns und Wellings [79] definieren die folgenden vier Kontexte für die Angabe von Zeiten in Spezifikationssprachen:

- Methoden zur **Messung der vergangenen Zeit**:
  Viele Anwendungen müssen überprüfen, wie viel Zeit seit dem Ende einer bestimmten Berechnung vergangen ist. Dies ließe sich durch einen Zeitgeber (engl. *timer*) realisieren.
- Eine Möglichkeit, **Prozesse[3] um eine bestimmte Zeit zu verzögern**:
  Echtzeitsprachen stellen üblicherweise Möglichkeiten zur Verfügung, die Ausführung hinauszuzögern. Leider garantieren die verbreiteten Softwareimplementierungen eingebetteter Systeme keine präzisen Verzögerungen. Wir nehmen an, dass ein Prozess τ um eine Zeit Δ verzögert werden soll. Meist wird dies implementiert, indem der Zustand von Prozess τ im Betriebssystem von „laufend" oder „bereit" auf „blockiert" gesetzt wird. Am Ende des Zeitintervalls wird der Zustand von τ dann von „blockiert" auf „bereit" gesetzt. Dies hat aber nicht zur Folge, dass der Prozess sofort danach ausgeführt wird. Wenn gleichzeitig ein Prozess mit höherer Priorität läuft oder keine Verdrängung (kein Anhalten eines laufenden Prozesses, engl. *preemption*) verwendet wird, ist die wirkliche Verzögerung des Prozesses größer als Δ.
- Eine Möglichkeit, ***Timeouts*** anzugeben:
  In vielen Situationen muss auf das Eintreten eines bestimmten Ereignisses gewartet werden. Wenn dieses Ereignis nicht innerhalb einer bestimmten Zeitspanne auftritt, möchten wir darüber informiert werden. So könnten wir beispielsweise auf eine Antwort von einer Netzwerkverbindung warten. Wir würden gerne informiert werden, wenn diese Antwort nicht innerhalb einer bestimmten Zeit Δ angekommen ist. Dies ist der Zweck von ***Timeouts***. Die meisten Echtzeitsprachen besitzen auch ein *Timeout*-Sprachelement. Die Implementierung von *Timeouts* bringt dieselben Probleme mit sich, wie sie bereits für Verzögerungen beschrieben wurden.
- Methoden, um **Zeitschranken** und **Abläufe (*Schedules*)** anzugeben:

[2] Gemäß Seite 22 hier als bekannt vorausgesetzt.

[3] Prozesse sind in Ausführung befindliche Programme, siehe Definition 2.3.

Viele Anwendungen müssen bestimmte Berechnungen in begrenzter Zeit ausführen. So müssen beispielsweise die *Airbags* eines Autos innerhalb von rund zehn Millisekunden zünden, nachdem ein Sensor einen Unfall angezeigt hat. Hier müssen wir garantieren, dass die Software in der vorgegebenen Zeit entscheidet, ob die *Airbags* gezündet werden sollen oder nicht. Wenn die *Airbags* zu spät zünden, könnten Insassen verletzt werden. Leider erlauben es die meisten Sprachen nicht, Zeitschranken anzugeben. Wenn sie angegeben werden können, dann zumeist in separaten Steuerungsdateien oder in Dialogfenstern. Die Situation wäre aber immer noch schwierig zu beherrschen, wenn sich Zeitschranken angeben ließen. Der Grund hierfür ist, dass moderne Hardware kein gut vorhersagbares Zeitverhalten besitzt. Caches, blockierte *Pipelines*, spekulative Ausführung, Verdrängung von Prozessen, Unterbrechungen (engl. *interrupts*) usw. haben alle einen nur schwer vorhersagbaren Einfluss auf die Ausführungszeit. Entsprechend ist die **Zeitanalyse** eine sehr schwierige Entwurfsaufgabe.

- **Zustandsorientiertes Verhalten:** In Kapitel 1 wurde schon erwähnt, dass Automaten eine gute Darstellungsmöglichkeit für reaktive Systeme sind. Aus diesem Grund sollte es einfach sein, zustandsorientiertes Verhalten, wie es endliche Automaten zeigen, zu beschreiben. Klassische Automatenmodelle sind allerdings nicht ausreichend, da sie weder Zeitbedingungen noch Hierarchie unterstützen.
- **Ereignisbehandlung:** Da eingebettete Systeme oft reaktive Systeme sind, müssen Mechanismen zur Beschreibung von Ereignissen existieren. Solche Ereignisse können externe (von der Umwelt erzeugte) oder interne (von Komponenten des Systems erzeugte) Ereignisse sein.
- **Ausnahmeorientiertes Verhalten:** In vielen realen Systemen treten Ausnahmen auf. Um verlässliche Systeme entwerfen zu können, muss die Behandlung solcher Ausnahmesituationen einfach zu beschreiben sein. Es ist nicht ausreichend, wenn man die Ausnahmebehandlung z.B. für jeden einzelnen Zustand angeben muss, wie das etwa bei klassischen Automatenmodellen der Fall ist.

Beispiel 2.1: In Abb. 2.1 soll die Eingabe k das Auftreten einer Ausnahme darstellen. Die Angabe einer solchen Ausnahme für jeden Zustand lässt das Diagramm

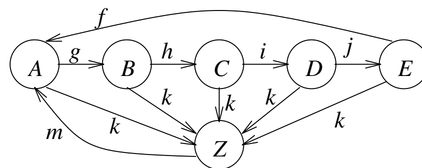


Abb. 2.1 Zustandsdiagramm mit Ausnahme k

komplex werden. Für größere Diagramme mit vielen Übergängen würde die Situation noch unübersichtlicher werden. Auf Seite 56 werden wir zeigen, wie man alle diese Transitionen durch eine einzige ersetzen kann (siehe Abb. 2.12). ∇

- **Programmiersprachenelemente:** Programmiersprachen sind eine weit verbreitete Methode zur Beschreibung von Berechnungsvorschriften. Daher sollten Elemente von Programmiersprachen in der Spezifikation verwendet werden können. Die bekannten Zustandsdiagramme erfüllen diese Anforderung nicht.
- **Ausführbarkeit:** Eine Spezifikation stimmt nicht automatisch mit der ursprünglichen Idee des Entwicklers überein. Das Ausführen der Spezifikation stellt eine Möglichkeit der Plausibilitätsprüfung für den Entwurf dar. Spezifikationen, die Programmiersprachenelemente verwenden, sind in diesem Zusammenhang von Vorteil.
- **Unterstützung für den Entwurf großer Systeme:** Eingebettete Systeme verwenden immer größere und komplexere Programme. Die Softwaretechnologie verfügt über Mechanismen wie z.B. Objektorientierung, um solche großen Systeme handhabbar zu machen. Solche Mechanismen sollten auch in der Spezifikationsmethodik zum Einsatz kommen.
- **Unterstützung von spezifischen Anwendungsgebieten:** Es wäre wünschenswert, wenn ein und dieselbe Spezifikationstechnik für alle möglichen eingebetteten Systeme verwendet werden könnte, da dies den Aufwand für die Entwicklung der Techniken minimieren würde. Allerdings ist die mögliche Bandbreite von Anwendungsgebieten sehr groß (siehe Abschnitt 1.2). Es ist daher kaum zu erwarten, dass eine einzige Sprache die Belange aller Anwendungsbereiche gleichermaßen gut abdecken kann. Beispielsweise können kontrollflussdominierte, datenflussdominierte, zentralisierte oder verteilte Anwendungsgebiete von spezifischer Werkzeugunterstützung für den jeweiligen Bereich profitieren.
- **Lesbarkeit:** Selbstverständlich muss eine Spezifikation von Menschen gelesen werden können. Ansonsten wäre es nicht möglich zu überprüfen, ob die Spezifikation tatsächlich die Absichten des Entwicklers des Systems wiedergibt. Alle Entwurfsdokumente sollten zudem maschinenlesbar sein, damit sie mit Computern verarbeitet werden können. Spezifikationen sollten also in Sprachen festgehalten werden, die sowohl von Menschen wie auch von Computern lesbar sind.
  Anfangs können solche Spezifikationen eine natürliche Sprache wie Deutsch, Englisch oder Japanisch verwenden. Auch diese natürlichsprachige Beschreibung sollte in einem Entwurfsdokument festgehalten werden, damit die resultierende Implementierung mit dieser Ausgangsspezifikation verglichen werden kann. Natürliche Sprachen sind aber für spätere Entwurfsphasen unzulänglich, da ihnen wichtige Anforderungen an Spezifikationstechniken fehlen: es ist notwendig, Spezifikationen auf Vollständigkeit und Widerspruchsfreiheit zu prüfen und es sollte zudem möglich sein, systematisch Implementierungen aus den Spezifikationen ableiten zu können. Natürliche Sprachen erfüllen diese Anforderungen nicht.
- **Portierbarkeit und Flexibilität:** Spezifikationen sollten unabhängig von der für die Implementierung verwendeten spezifischen Hardwareplattform sein, sodass man sie leicht für verschiedene Zielplattformen einsetzen kann. Idealerweise sollte eine Änderung der Hardwareplattform keinen Einfluss auf die Spezifikation haben. Hier müssen im praktischen Einsatz gegebenenfalls kleine Änderungen akzeptiert werden.

- **Terminierung:** Es sollte möglich sein, anhand der Spezifikation Prozesse zu identifizieren, die terminieren. Daher möchten wir Spezifikationen verwenden, für die das Halteproblem (das Problem, herauszufinden, ob ein gegebener Algorithmus terminieren wird oder nicht, siehe z.B. [494]) entscheidbar ist.
- **Unterstützung für Nicht-Standard-Ein/Ausgabegeräte:** Viele eingebettete Systeme verwenden andere Ein- und Ausgabegeräte als die vom PC her bekannten Tastaturen und Mäuse. Es sollte möglich sein, die Eigenschaften solcher Geräte in bequemer Weise zu spezifizieren.
- **Nicht-funktionale Eigenschaften:** Reale Systeme besitzen eine Reihe nichtfunktionaler Eigenschaften, wie etwa Fehlertoleranz, Größe, Erweiterbarkeit, Lebenserwartung, Energieverbrauch, Gewicht, *Recycling*-Fähigkeit, Benutzerfreundlichkeit, elektromagnetische Verträglichkeit (EMC) und so weiter. Es ist nicht zu erwarten, dass sich all diese Eigenschaften auf eine formale Weise definieren lassen.
- **Unterstützung für den Entwurf verlässlicher Systeme:** Spezifikationstechniken sollten den Entwurf von verlässlichen Systemen unterstützen. Beispielsweise sollte die Spezifikationssprache eine eindeutige Semantik haben und den Einsatz formaler Verifikationstechniken erlauben. Außerdem sollte es möglich sein, Anforderungen an Betriebs- und Informationssicherheit zu beschreiben.
- **Keine Hürden für die Erzeugung von effizienten Implementierungen:** Da eingebettete Systeme effizient sein müssen, sollte die Spezifikationssprache eine effiziente Realisierung des Systems nicht behindern oder unmöglich machen.
- **Geeignetes Berechnungsmodell** (engl. *Model of Computation* (MoC)): Ein häufig verwendetes Berechnungsmodell ist das sequenzielle Ausführungsmodell nach von Neumann in Kombination mit einem Kommunikationsverfahren. Im MoC nach von Neumann sind Spezifikationen üblicherweise in Tasks, Prozesse oder *Threads* gegliedert, die wie folgt definiert werden können:

  Definition 2.2 ([394]): Eine **Task** kann allgemein definiert werden als „eine zugewiesene Menge an Arbeit, die häufig in einer bestimmten Zeit zu erledigen ist".

  Im Kontext von eingebetteten Systemen verstehen wir unter einer Task i.d.R. bestimmte Berechnungen, die auszuführen sind.

  Definition 2.3 ([525]): Ein **Prozess** ist ein in Ausführung befindliches Programm.

  Eine Präzisierung dieses Begriffs wird in der Definition 4.1 gegeben werden. Tasks sind teilweise abstrakter beschrieben als die Prozesse, sie sind dann auf konkrete Prozesse innerhalb eines Betriebssystems abzubilden. Allerdings werden die Begriffe Task und Prozess auch teilweise austauschbar benutzt. Eng verwandt mit dem Begriff „Prozess" ist der Begriff des *Threads*.

  Definition 2.4: Ein ***Thread*** ist ein „leichtgewichtiger" Prozess. Das bedeutet, dass die Umschaltung zwischen der Ausführung von *Threads* mit weniger Aufwand verbunden ist als bei der Umschaltung zwischen allgemeinen Prozessen.

In der Regel können *Threads* untereinander über einen Zugriff auf gemeinsamen Speicher kommunizieren.

Eine Präzisierung dieses Begriffs wird in der Definition 4.2 gegeben werden. Das Modell nach von Neumann weist allerdings besonders für eingebettete Systeme eine Reihe schwerwiegender Probleme auf, wie z.B.:

- Es fehlen Möglichkeiten zur Beschreibung von Zeitverhalten.
- Berechnungen nach dem von-Neumann-Modell basieren implizit auf Zugriffen auf globalen gemeinsamen Speicher (wie in Java). Dabei muss der exklusive Zugriff auf die gemeinsam verwendeten Ressourcen sichergestellt sein. Ansonsten würden Anwendungen mit mehreren *Threads*, die zu beliebigen Zeiten verdrängt werden können, ein nur schwer vorhersagbares Verhalten zur Folge haben[4]. Die Verwendung von Funktionen, die exklusiven Zugriff ermöglichen, kann aber sehr schnell zu Verklemmungen (engl. *Deadlocks*) führen. Diese sind nur schwer zu erkennen und können in einem System viele Jahre lang unentdeckt bleiben.

  Beispiel 2.2: Edward Lee [330] zeigt hierzu ein sehr drastisches Beispiel. Er analysierte Implementierungen eines einfachen *Observer Patterns* in Java. Dieses erfordert, dass Änderungen eines Wertes von einem Erzeuger an eine Menge an angemeldeten Beobachtern weitergeleitet werden. Dieses Verhalten kommt in eingebetteten Systemen sehr häufig vor, ist aber in einer von-Neumann-basierten Umgebung mit mehreren verdrängbaren *Threads* nur sehr schwer korrekt zu implementieren. Lee's Code für eine möglichen Implementierung des *Observer Patterns* in Java in einer Umgebung mit mehreren *Threads* sieht wie folgt aus:

```
public synchronized void addListener(listener) {...}
public synchronized void setValue(newvalue) {
  myvalue=newvalue;
  for (int i=0; i<mylisteners.length; i++) {
    myListeners[i].valueChanged(newvalue);
  }
}
```

  Die Methode `addListener` meldet neue Beobachter an, die Methode `setValue` leitet neue Werte an angemeldete Beobachter weiter. In einer Umgebung mit mehreren *Threads* können normalerweise *Threads* zu jedem beliebigen Zeitpunkt verdrängt werden, was eine nicht vorhersagbare Ausführungsreihenfolge der *Threads* zur Folge hat. Das Hinzufügen von Beobachtern, während `setValue` gerade ausgeführt wird, kann zu Komplikationen führen, d.h. wir würden nicht wissen, ob der neue Wert bereits den Beobachter erreicht hat. Zudem stellt die Menge der Beobachter eine globale Datenstruktur dieser Klasse dar. Daher sind diese Methoden synchronisiert, um zu vermeiden, dass die Menge der Beobachter geändert wird, während bereits einige Werte weitergeleitet werden. So kann nur eine der beiden Methoden gleichzeitig aktiv

[4] Beispiele dafür sind in den meisten Kursen zu Betriebssystemen zu finden.

sein. Dieser gegenseitige Ausschluss ist notwendig, um die unerwünschte Vermischung der Ausführung verschiedener Methoden in einer Umgebung mit mehreren *Threads* zu verhindern. Warum ist dieser Code nun problematisch? Der Grund dafür ist, dass `valueChanged` versuchen könnte, exklusiven Zugriff auf eine bestimmte Ressource (z.B. `R`) zu erhalten. Wenn diese Ressource bereits von einer anderen Methode (z.B. `A`) belegt ist, wird dieser Zugriff verzögert, bis `A` die Ressource `R` freigibt. Wenn nun `A` die Methode `addListener` oder `setValue` (möglicherweise indirekt) aufruft, bevor `R` freigegeben wurde, entsteht eine Verklemmung zwischen diesen Methoden: `setValue` wartet auf `R`, die Freigabe von `R` erfordert, dass `A` fortfahren kann, `A` kann aber nicht fortfahren, bevor sein Aufruf der Methode `setValue` oder `setListener` zurückkehrt. Dadurch entsteht die Verklemmung.

Dieses Beispiel verdeutlicht, dass Verklemmungen als Folge der Verwendung mehrerer *Threads* entstehen können, wenn diese beliebig verdrängt werden können und damit gegenseitigen Ausschluss für den Zugriff auf kritische Ressourcen benötigen. Lee hat gezeigt [330], dass viele der vorgeschlagenen „Lösungen" des Problems selbst wieder problematisch sind. Also ist auch dieses sehr einfache Verhalten in einer von-Neumann-Umgebung mit mehreren *Threads* nur schwer korrekt zu implementieren. Offenbar können Menschen Nebenläufigkeit nur sehr schlecht verstehen und es besteht das Risiko, dass mögliche Verklemmungen übersehen werden, auch wenn der Code gründlichst inspiziert wurde. ∇

Lee kam daher zu der drastischen Schlussfolgerung, dass „nichttriviale, mit *Threads*, Semaphoren und *Mutexen* entwickelte Software für Menschen unverständlich ist" und dass „*Threads* als ein Modell für Nebenläufigkeit nur sehr schlecht zu eingebetteten Systemen passen. ... sie funktionieren nur gut ... wenn *best-effort Scheduling*-Verfahren ausreichend sind" [333].

Die Gründe der Entstehung von Verklemmungen wurden detailliert im Zusammenhang mit Betriebssystemen untersucht (z.B. in [507]). Es gibt hier vier wohlbekannte Bedingungen, die erfüllt sein müssen, damit eine Verklemmung eintreten kann: gegenseitiger Ausschluss, kein Entzug von Ressourcen, das Halten von Ressourcen, während auf eine weitere Ressource gewartet wird und schließlich die gegenseitige Abhängigkeit von *Threads*. Das erwähnte Beispiel erfüllt alle vier Kriterien. Die Betriebssystemtheorie stellt keine allgemeingültige Lösung für dieses Problem bereit. Seltene Verklemmungen mögen bei einem PC annehmbar sein, in sicherheitskritischen Systemen sind sie aber keinesfalls akzeptabel.

Wir möchten nun Systeme so spezifizieren, dass wir uns keine Sorgen um mögliche Verklemmungen machen müssen. Daher erscheint es sinnvoll, Berechnungsmodelle zu betrachten, die nicht auf von-Neumann-Prinzipien basieren und damit das Problem von Verklemmungen umgehen. Wir beginnen die Betrachtung solcher Berechnungsmodelle im folgenden Abschnitt. Darin zeigen wir, dass das *Observer*-Muster mit anderen Berechnungsmodellen einfacher korrekt implementierbar ist.

Die Liste von Anforderungen an Modellierungssprachen lässt erkennen, dass es wohl nie eine einzige formale Sprache geben wird, die alle diese Anforderungen erfüllt. In der Praxis muss man daher in der Regel mit Kompromissen leben und eine Kombination verschiedener Sprachen einsetzen, von der jede für die Beschreibung eines bestimmten Problemkreises geeignet ist. Die Auswahl der Spezifikationssprache, die für ein bestimmtes Projekt verwendet wird, hängt hauptsächlich vom Anwendungsbereich und von der Umgebung, in der das System entwickelt werden soll, ab. Wir stellen im Folgenden einige Sprachen vor, die für den Entwurf eines praxisnahen Systems in Frage kommen. Diese Sprachen dienen als Beispiele, um die wichtigsten Eigenschaften des zugehörigen Berechnungsmodells zu beschreiben.

2.2 Berechnungsmodelle

Berechnungsmodelle (engl. *Models of Computation* (MoCs)) beschreiben Mechanismen, welche die Durchführung von Berechnungen beschreiben. Im Allgemeinen müssen wir dabei von Systemen ausgehen, die aus verschiedenen Komponenten bestehen. Mittlerweile ist es üblich, streng zwischen den Berechnungen in den einzelnen Komponenten und der Kommunikation zu trennen. Diese Unterscheidung erleichtert die Wiederbenutzung von Komponenten in verschiedenen Kontexten und sie ermöglicht ein *plug-and-play*-Konzept für Systemkomponenten. Dementsprechend definieren wir Berechnungsmodelle wie folgt [329, 268, 269, 270]:

Definition 2.5: Berechnungsmodelle (MoCs) definieren

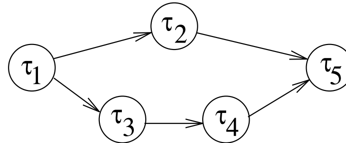
- **Komponenten** und die Organisation des Ablaufs von Berechnungen in diesen Komponenten: Prozeduren, Prozesse, Funktionen oder endliche Automaten sind mögliche Komponenten.
- **Kommunikationsprotokolle**: Diese Protokolle beschreiben die Kommunikationsmöglichkeiten zwischen den Komponenten. Asynchroner Nachrichtenaustausch und *Rendez-Vous*-basierte Kommunikation sind Beispiele für solche Protokolle.

Beziehungen zwischen Komponenten können mit Hilfe von Graphen beschrieben werden. In solchen Graphen bezeichnen wir die Berechnungen als **Tasks**. Die Beziehungen zwischen Tasks werden wir als **Task-Graph** und in bestimmten Kontexten auch als **Prozessnetz** bezeichnen. Knoten in Graphen repräsentieren Komponenten, die bestimmte Berechnungen ausführen. Die Berechnungen bilden Eingabedatenströme auf Ausgabedatenströme ab. Sie werden in einigen Fällen in Hochsprachen implementiert. Typische Tasks enthalten (möglicherweise nicht-terminierende) Schleifen. In jedem Durchlauf einer solchen Schleife lesen sie Daten von ihren Eingängen, verarbeiten diese Daten und erzeugen Daten in den Ausgabedatenströmen. Kanten beschreiben die Beziehungen zwischen Komponenten. Im Folgenden definieren wir diese Graphen nun genauer.

Die offensichtlichste Beziehung zwischen Tasks ist die kausale Abhängigkeit: Viele Berechnungen können erst dann ausgeführt werden, wenn andere Berech-

nungen abgeschlossen sind. Solche **Reihenfolgebeschränkungen** implizieren eine **Präzedenzrelation** und sie werden typischerweise in **Abhängigkeitsgraphen** als Kanten dargestellt. Abb. 2.2 zeigt einen Abhängigkeitsgraph für eine Menge von Berechnungen.

Abb. 2.2 Abhängigkeitsgraph



Definition 2.6: Ein Abhängigkeitsgraph ist ein gerichteter azyklischer Graph (engl. *Directed Acyclic Graph* (DAG)) $G = (\tau, E)$ mit der **Knotenmenge** τ und der **Kantenmenge** E. $E \subseteq \tau \times \tau$ stellt eine Relation auf τ dar. Wenn $(\tau_1, \tau_2) \in E$, dann wird τ_1 ein **direkter Vorgänger** von τ_2 genannt, entsprechend heißt τ_2 **direkter Nachfolger** von τ_1. Sei nun E^* die transitive Hülle von E. Wenn $(\tau_1, \tau_2) \in E^*$, dann ist τ_1 **Vorgänger** von τ_2 und τ_2 ist **Nachfolger** von τ_1.

Solche Abhängigkeitsgraphen sind Spezialfälle von Task-Graphen. Task-Graphen können mehr Informationen enthalten als in Abb. 2.2 dargestellt. Beispielsweise können Task-Graphen die Abhängigkeitsgraphen um folgende Eigenschaften erweitern:

1. **Zeit-Informationen**: Tasks können Ankunftszeiten, *Deadlines*, Perioden und Ausführungszeiten haben. Um diese graphisch darzustellen, kann es sinnvoll sein, sie in den Graphen darzustellen. In diesem Buch werden wir diese Informationen aber unabhängig von den Graphen repräsentieren.
2. **Unterscheidung von verschiedenen Beziehungsarten** zwischen Berechnungen: Präzedenzrelationen modellieren lediglich Reihenfolgebeschränkungen für die Ausführung. Auf einer detaillierteren Ebene kann es hilfreich sein, zwischen Bedingungen für das *Scheduling* und für die Kommunikation zwischen Berechnungen zu unterscheiden. Kommunikation kann wieder durch Kanten beschrieben werden. Es können allerdings weitere Informationen zu jeder dieser Kanten bekannt sein, etwa der Zeitpunkt der Kommunikation und die ausgetauschte Datenmenge. Präzedenzrelationen können als eigene Kantenart betrachtet werden, da es Situationen geben kann, in denen Tasks nacheinander ausgeführt werden müssen, obwohl sie keine Informationen miteinander austauschen.
   In Abb. 2.2 sind Ein- und Ausgaben (engl. *Input/Output* (I/O)) nicht explizit dargestellt. Implizit wird angenommen, dass Knoten ohne Vorgänger im Graphen irgendwann eine Eingabe erhalten. Weiter wird angenommen, dass sie nach einer gewissen Zeit eine Ausgabe für den Nachfolgerknoten erzeugen. Diese Ausgabe ist erst dann verfügbar, wenn die Berechnung abgeschlossen ist. Es ist oft nützlich, Ein- und Ausgaben eindeutiger zu beschreiben. Um dies zu erreichen, wird ein weiterer Relationstyp benötigt. Wir verwenden die Notation von Thoen [538], in der Kreise mit Punkt Ein- und Ausgaben darstellen, wie in Abb. 2.3 zu sehen.

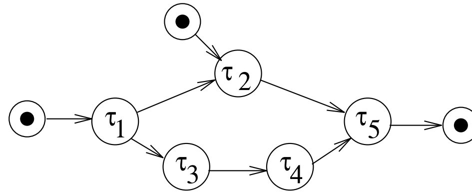


Abb. 2.3 Task-Graph mit Ein/Ausgabeknoten und -kanten

3. **Exklusiver Zugriff auf Ressourcen:** Berechnungen können einen exklusiven Zugriff auf eine Ressource anfordern, etwa auf ein Ein/Ausgabegerät oder auf einen Speicherbereich zur Kommunikation. Informationen über eventuell notwendige exklusive Zugriffe sollten beim *Scheduling* berücksichtigt werden. Durch die Verwendung dieser Information kann man z.B. das Problem der Prioritätsumkehr vermeiden (siehe Seite 233). Solche Informationen über exklusiven Zugriff können auch in Task-Graphen dargestellt werden.
4. **Periodische Abläufe:** Viele Berechnungen, insbesondere im Bereich der digitalen Signalverarbeitung, sind periodisch. Man muss also genauer zwischen einer Task und ihrer Ausführung unterscheiden – letztere wird häufig als **Job** bezeichnet [349][5]. Task-Graphen für solche *Schedules* sind unendlich groß. Abb. 2.4 zeigt einen Task-Graphen mit den Jobs J_{n-1} bis J_{n+1} einer periodischen Task.

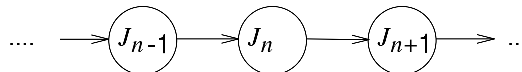


Abb. 2.4 Task-Graph mit Jobs (Ausführungen) einer periodischen Task

5. **Hierarchische Knoten**: Die Komplexität der Berechnungen, die in einem Knoten ausgeführt werden, kann sehr unterschiedlich sein. Einerseits können die beschriebenen Programme sehr groß sein und Tausende von Codezeilen enthalten. Andererseits kann man Programme in kleine Programmteile aufteilen, sodass im Extremfall jeder Knoten nur genau einer Operation entspricht. Die Komplexität von Knoten im Graphen heißt **Granularität**. Die Frage, welche Granularität man verwenden sollte, lässt sich nicht allgemeingültig beantworten. Für einige Zwecke sollte die Granularität so grob wie möglich sein, etwa wenn die Knoten Prozesse eines Echtzeitbetriebssystems darstellen. In diesem Fall sollten die Knoten große Programmteile enthalten, um die Anzahl der Kontextwechsel zwischen den Prozessen gering zu halten. Für andere Anwendungen kann es sinnvoll sein, wenn jeder Knoten nur eine Operation enthält, etwa wenn die Knoten entweder als Hardware oder als Software realisiert werden sollen. Wenn eine bestimmte Operation (z.B. die häufig vorkommende Diskrete Cosinus-Transformation (DCT)) auf eine Spezialhardware abgebildet werden kann, dann sollte sie nicht in einem großen Knoten mit vielen anderen Operationen versteckt sein. Vielmehr

[5] Präzisierungen erfolgen in den Definitionen 4.4 und 6.1.

sollte die DCT als einzelner Knoten modelliert werden. Hierarchische Knoten helfen dabei, häufige Wechsel der Granularität zu vermeiden. Auf einer hohen Hierarchieebene können die Knoten dann z.B. komplexe Aufgaben beschreiben, auf einer niederen Ebene Basisblöcke[6] oder auf einer noch niedrigeren Ebene einzelne arithmetische Operationen. Abb. 2.5 zeigt eine hierarchische Version des Abhängigkeitsgraphen aus Abb. 2.2, in dem ein hierarchischer Knoten durch ein Rechteck dargestellt wird.

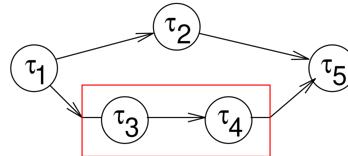


Abb. 2.5 Hierarchischer Task-Graph

Wie oben beschrieben, lassen sich Berechnungsmodelle anhand der Kommunikationsmodelle (dargestellt durch Kanten im Task-Graphen) und der Berechnungsmodelle innerhalb der Komponenten (dargestellt durch die Knoten des Task-Graphen) klassifizieren. Im Folgenden führen wir einige bekannte Beispiele für Berechnungsmodelle auf:

- **Kommunikationsmodelle**: Wir unterscheiden zwischen zwei Kommunikationsparadigmen: **Gemeinsamer Speicher** und **Nachrichtenaustausch**. Weitere existierende Kommunikationsmodelle (wie z.B. verschränkte Zustände in der Quantenmechanik [62]) werden in diesem Buch nicht betrachtet.

  – **Gemeinsamer Speicher** (engl. *shared memory*)
  Bei diesem Modell greifen kommunizierende Komponenten auf gemeinsamen Speicher zu. Zugriffe auf gemeinsamen Speicher sollten geschützt werden, solange nicht ausschließlich Lesezugriffe erfolgen. Sobald ein Schreibzugriff verwendet wird, muss exklusiver Zugriff auf den Speicher für alle beteiligten Komponenten garantiert werden. Die Programmabschnitte, die den exklusiven Zugriff benötigen, werden **kritische Abschnitte** genannt (siehe auch Lee's Beispiel 2.2). Zu den Methoden, die exklusiven Zugriff sicherstellen, gehören Semaphore, bedingte kritische Abschnitte und Monitore. Die einzelnen Techniken sind im Detail in Betriebssystem-Lehrbüchern, wie z.B. von Stallings [507], beschrieben. Kommunikation über gemeinsamen Speicher kann sehr schnell sein. Allerdings ist die Implementierung für Multiprozessor-Systeme schwierig zu realisieren, wenn kein physikalisch gemeinsamer Speicher vorhanden ist und dieser daher simuliert werden muss.

[6] Basisblöcke sind Codeblöcke von maximaler Länge, die – außer möglicherweise an ihrem Ende – keinen Sprungbefehl enthalten und in die nicht von anderen Codeblöcken aus hinein verzweigt wird.

- **Nachrichtenaustausch** (engl. *message passing*): Beim Nachrichtenaustausch kommunizieren Prozesse durch das Versenden von Nachrichten miteinander. Diese Methode lässt sich auch dann einfach implementieren, wenn kein gemeinsamer Speicher zur Verfügung steht. Allerdings ist Nachrichtenaustausch meist langsamer als Kommunikation über gemeinsamen Speicher. Für diese Kommunikationsmethode wird zwischen den folgenden drei Techniken unterschieden:
  - Beim **asynchronen Nachrichtenaustausch**, auch **nicht-blockierende** Kommunikation genannt, kommunizieren die Komponenten durch Senden von Nachrichten über Kanäle, die Nachrichten puffern können. Der Sender muss also nicht warten, bis der Empfänger bereit ist, die Nachricht zu empfangen. Dies entspricht dem Versenden eines Briefs oder einer Email. Hier kann potenziell das Problem auftreten, dass die Nachrichten gespeichert werden müssen und die Nachrichtenpuffer überlaufen können. Verschiedene Sprachen, wie z.B. SDL (siehe Seite 68) und SDF (siehe Seite 78), nutzen diese Kommunikationsmethode.
  - Bei **synchronem Nachrichtenaustausch**, auch **blockierende** Kommunikation oder ***Rendez-Vous*-Kommunikation** genannt, kommunizieren die Komponenten mittels atomarer, unverzögerter Aktionen, die *Rendez-Vous* genannt werden. Derjenige Kommunikationspartner, der die entsprechende Codestelle zuerst erreicht, muss auf den anderen Partner warten. Dies entspricht einem Treffen oder einem Telefonanruf. Beispiele hierfür sind CSP (siehe Seite 122) und Ada (siehe Seite 123).
  - **Erweitertes *Rendez-Vous*** (auch entfernter Methodenaufruf) erweitert die *Rendez-Vous*-Kommunikationsmethode dahingehend, dass der Sender nur dann mit seiner Programmausführung fortfahren darf, wenn eine Bestätigung für die versendete Nachricht erhalten wurde. Der Empfänger muss diese Nachricht aber nicht unmittelbar nach Empfang der Nachricht senden, vielmehr kann er z.B. die empfangenen Nachrichteninhalte vor dem Senden der Bestätigung überprüfen und, falls erforderlich, bei Fehlern eine andere Antwort senden.

- **Organisation von Berechnungen innerhalb der Komponenten:**
  - **Differentialgleichungen:** Mit Differentialgleichungen lassen sich analoge Schaltkreise und physikalische Systeme modellieren. Damit finden sie Anwendung in der Modellierung cyber-physikalischer Systeme.
  - **Endliche Automaten** (engl. *Finite State Machines* (FSMs)): Dieses Modell basiert auf einer endlichen Menge von Zuständen, Ein- und Ausgaben sowie Transitionen zwischen Zuständen. Es kann vorkommen, dass mehrere endliche Automaten kommunizieren müssen, diese bilden dann **kommunizierende endliche Automaten** (engl. *Communicating Finite State Machines* (CFSMs)).
  - **Datenfluss:** Im Datenflussmodell triggert die Verfügbarkeit von Daten (d.h. der Operanden) die Ausführung von Operationen.
  - **Diskretes Ereignismodell:** In diesem Modell tragen alle Ereignisse einen vollständig geordneten Zeitstempel, der die Zeit angibt, zu der das Ereig-

nis stattfindet. Simulatoren für diskrete Ereignismodelle verfügen über eine nach der Zeit sortierte globale Ereigniswarteschlange. Einträge in der Warteschlange werden in genau dieser Reihenfolge abgearbeitet. Der Nachteil dieses Modells ist, dass es auf globalen Ereigniswarteschlangen basiert. Dies macht es schwierig, das semantische Modell auf parallele Implementierungen abzubilden. Beispiele sind u.a. SystemC (siehe Seite 107), VHDL (siehe Seite 109) und Verilog (siehe Seite 120).
  – **Von-Neumann-Modell:** Dieses Modell basiert auf der sequenziellen Ausführung von Folgen einfacher Berechnungsoperationen.

- **Kombinierte Modelle:** Real existierende Sprachen verbinden meist ein bestimmtes Kommunikationsmodell mit einem Modell der Berechnungen in den Komponenten. So kombiniert StateCharts (siehe Seite 55) endliche Automaten mit gemeinsamem Speicher, SDL (siehe Seite 68) verbindet endliche Automaten für die Berechnung in den Komponenten mit asynchronem Nachrichtenaustausch für die Kommunikation. Ada (siehe Seite 123) und CSP (siehe Seite 122) kombinieren das Modell der von-Neumann-Befehlsausführung mit synchronem Nachrichtenaustausch. Tabelle 2.1 gibt einen Überblick über die in diesem Kapitel behandelten kombinierten Modelle. Diese Tabelle führt für die meisten Berechnungsmodelle auch eine Beispielsprache auf.

Tabelle 2.1 Überblick über betrachtete Berechnungsmodelle und Sprachen

Kommunikation/Organisation der Komponenten	Gemeinsamer Speicher (*shared memory*)	Nachrichtenaustausch synchron	Nachrichtenaustausch asynchron
Undefinierte Komponenten	Text oder Grafik, Anwendungsfälle (*use cases*)	(*Message*) *sequence charts*	
Differentialgleichungen	Modelica, Simulink®, VHDL-AMS		
Kommunizierende endliche Automaten (CFSMs)	StateCharts		SDL
Datenfluss	*Scoreboarding*, Tomasulo-Algorithmus		Kahn-Netzwerke SDF
Petrinetze		C/E-Netze, P/T-Netze, ...	
Diskrete Ereignis (DE)-Modelle *	VHDL, Verilog, SystemC	(Nur experimentelle Systeme) Verteilte DE in Ptolemy	
Von-Neumann-Modell	C, C++, Java	C, C++, Java, ... mit Bibliotheken; CSP, Ada	

* Die Klassifikation von VHDL, Verilog und SystemC basiert auf deren Implementierung in Simulatoren. Nachrichtenaustausch kann in diesen Sprachen auf dem Simulationskern aufbauend implementiert werden.

Unter den Sprachen mit einem definierten Berechnungsmodell finden wir für Differentialgleichungen als Beispiele die Sprachen Modelica [400], Simulink® [533] und die Erweiterung der Hardwarebeschreibungssprache VHDL zu VHDL-AMS [246].

Scoreboarding und der Tomasulo-Algorithmus sind Datenfluss-getriebene Berechnungsmodelle für das dynamische *Scheduling* in der Rechnerarchitektur. Sie werden in den Rechnerarchitektur-Büchern von Hennessy und Patterson [212] beschrieben und sind nicht Teil dieses Buches.

Jedes der Berechnungsmodelle hat Vor- und Nachteile für bestimmte Anwendungsgebiete. Es kann daher schwierig sein, das „beste" Berechnungsmodell für eine bestimmte Anwendung zu wählen. Eine Möglichkeit zur Vereinfachung der Auswahl ist es, Berechnungsmodelle zu kombinieren (wie z.B. in Ptolemy [460, 120]). Modelle können auch aus einem Berechnungsmodell in ein anderes überführt werden. Nicht-von-Neumann-Modelle werden häufig in von-Neumann-Modelle übersetzt. Die Unterschiede zwischen den einzelnen Modellen verschwimmen, wenn es einfache Möglichkeiten gibt, sie ineinander umzuwandeln.

Entwürfe, die von nicht-von-Neumann-Modellen ausgehen, werden oft **modellbasierte Entwürfe** genannt [422]. Der grundlegende Gedanke hinter dem modellbasierten Entwurf ist es, ein abstraktes Modell des zu entwerfenden Systems zu erstellen. Die Eigenschaften des Systems können dann auf dieser Abstraktionsebene analysiert werden, ohne sich mit konkreter Software befassen zu müssen. Software wird erst generiert, nachdem das Verhalten des Modells genau analysiert wurde, diese Software wird dann automatisch erzeugt [477]. Der Begriff „modellbasierter Entwurf" ist allerdings nicht genau definiert. Er wird meist mit Modellen von Regelungssystemen verbunden, die Komponenten herkömmlicher Regelungssysteme wie Integratoren, Differentiatoren usw. enthalten. Diese Sichtweise erscheint aber zu sehr eingeschränkt, da wir z.B. auch mit einem abstrakten Modell eines Geräts aus dem Bereich der Konsumelektronik beginnen könnten.

Im Folgenden stellen wir verschiedene Berechnungsmodelle vor und verwenden dabei existierende Sprachen zur Beschreibung ihrer Eigenschaften. Edwards [147] gibt einen ähnlichen (aber kürzeren) Überblick über dieses Thema. Eine ausführlichere Abhandlung ist in [187] zu finden.

2.3 Frühe Entwurfsphasen

Die anfänglichen Gedanken zu einem System werden meist sehr formlos festgehalten, vielleicht auf einem Blatt Papier. Meist existieren in dieser frühen Phase des Entwurfs dabei nur Beschreibungen des Systems in einer natürlichen Sprache wie Englisch oder Japanisch. Diese oft sehr informellen Beschreibungen sollten in maschinenlesbarer Form erfasst werden, beispielsweise mit einer Textverarbeitung, und dann durch ein Verwaltungswerkzeug gespeichert werden. Ein gutes Werkzeug könnte dabei Verbindungen zwischen den Anforderungen modellieren sowie eine Abhängigkeitsanalyse und eine Versionsverwaltung anbieten. DOORS® [229] ist ein Beispiel solcher Werkzeuge.

2.3.1 Anwendungsfälle

Für viele Entwürfe ist es nützlich, Wissen über die potenziellen Verwendungsmöglichkeiten des Systems nutzen zu könnnen. Dieses Wissen wird in **Anwendungsfällen** (engl. *use cases*) festgehalten. Anwendungsfälle beschreiben mögliche Anwendungen des Systems; hierfür können unterschiedliche Darstellungsarten zum Einsatz kommen.

Das Ziel des UML-Standards [435, 166, 208] ist es, einen systematischen Ansatz für die ersten Spezifikationsphasen zu bieten. UML steht für „*Unified Modeling Language*" und wurde von führenden Softwareexperten entwickelt. Die verfügbaren kommerziellen Werkzeuge dienen hauptsächlich der Unterstützung des Softwareentwurfsprozesses. UML stellt dafür eine standardisierte Form von Anwendungsfällen zur Verfügung.

Anwendungsfälle besitzen weder ein genau spezifiziertes Berechnungsmodell noch ein präzise spezifiziertes Kommunikationsmodell. Häufig wird behauptet, dies sei eine absichtliche Einschränkung, um zu vermeiden, dass sich Entwickler in den frühen Entwurfsphasen zu viele Gedanken über Details machen.

Beispiel 2.3: In Abb. 2.6 sind einige Anwendungsfälle für einen Anrufbeantworter zu sehen[7]. Es gibt fünf Anwendungsfälle für den Eigentümer des Anrufbeantworters und einen für einen potentiellen Anrufer. Wir müssen sicherstellen, dass alle sechs Fälle richtig implementiert werden.

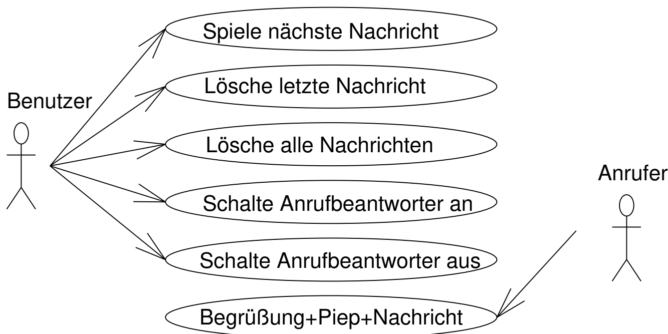


Abb. 2.6 Beispiel für Anwendungsfälle ∇

Anwendungsfälle ermöglichen es, verschiedene Klassen von Benutzern und Anwendungen, die das System unterstützen soll, voneinander zu unterscheiden. So ist es möglich, die Erwartungen an ein System auf sehr abstrakter Ebene festzuhalten.

[7] Wir gehen davon aus, dass UML ausführlich in einer Vorlesung über *Software-Engineering* behandelt wird. Daher wird das Thema UML in diesem Buch nur kurz angerissen.

2.3.2 (Message) Sequence Charts

Wenn eine etwas detailliertere Betrachtungsweise gewünscht ist, können die zwischen den einzelnen Komponenten ausgetauschten Nachrichtenfolgen angegeben werden. Diese werden benötigt, um eine bestimmte Art der Verwendung des Systems zu realisieren. ***Sequence charts*** (SCs) – früher auch ***message sequence charts*** (MSCs) genannt – ermöglichen dies. *Sequence charts* verwenden ein zweidimensionales Diagramm. In einer Dimension (üblicherweise in der Vertikalen) werden Abfolgen dargestellt, während die verschiedenen Kommunikationskomponenten in der zweiten Dimension angeordnet sind. SCs beschreiben Teilordnungen von Nachrichtenübertragungen. Sie beschreiben damit ein mögliches Systemverhalten.

SCs sind als Teil von UML standardisiert. Im Vergleich zu UML 1.0 wurden SCs in UML 2.0 durch Elemente erweitert, die genauere Beschreibungen ermöglichen.

Beispiel 2.4: Abb. 2.7 zeigt einen der Anwendungsfälle des Anrufbeantworters. Ge-

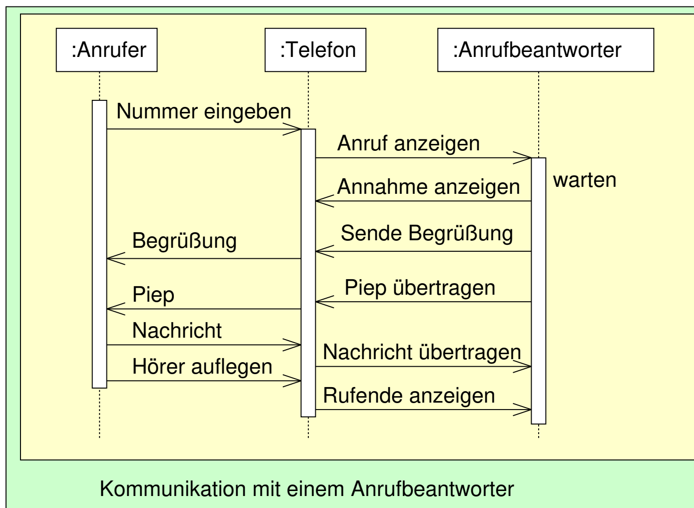


Abb. 2.7 Anrufbeantworter in UML™

strichelte Linien sind sogenannte „*life lines*" (Lebenslinien). Die Abfolge der Nachrichten entspricht ihrer Reihenfolge entlang der Lebenslinie. Bei diesem Beispiel gehen wir davon aus, dass alle Informationen in Form von Nachrichten versendet werden. Die Pfeile im Diagramm stehen für asynchrone Nachrichten. Damit kann ein Sender mehrere Nachrichten hintereinander aussenden, ohne auf eine Bestätigung warten zu müssen. Die Rechtecke oberhalb der Lebenslinien stellen die jeweils aktive Komponente dar. Im Beispiel wartet der Anrufbeantworter darauf, dass der Benutzer den Hörer des Telefons innerhalb einer bestimmten Zeit abnimmt. Wenn dies nicht geschieht, nimmt der Anrufbeantworter selbst das Gespräch an und spielt

eine Begrüßungsnachricht ab. Daraufhin kann der Anrufer seine Nachricht hinterlassen. Alternative Abläufe (z.B. das vorzeitige Beenden des Anrufs, weil der Anrufer auflegt oder der Fall, dass der Angerufene das Telefon abnimmt) sind hier nicht dargestellt. ∇

Komplexe kontrollflussabhängige Vorgänge lassen sich mit SCs nicht darstellen. Hierfür müssen andere Berechnungsmodelle verwendet werden. Oft müssen bestimmte Vorbedingungen erfüllt sein, damit SCs angewendet werden können. *Live Sequence Charts* [118] beinhalten Methoden zur Beschreibung solcher Vorbedingungen. Zudem verfügen sie über die Möglichkeit, zwischen notwendigen und optionalen Folgen zu unterscheiden und enthalten einige andere Erweiterungen.

Weg-Zeit-Diagramme (engl. *Time Distance Diagrams* (TDDs) oder auch *Distance Time Diagrams*) sind eine oft verwendete Variante von SCs. Die zeitliche Dimension von TDDs stellt nicht nur logische Abfolgen sondern real ablaufende Zeit dar, häufig auch horizontal aufgetragen. In einigen Varianten modelliert die räumliche Dimension auch den geographischen Abstand zwischen den Komponenten. TDDs sind gut geeignet, um Fahrpläne von Verkehrsmitteln darzustellen.

Beispiel 2.5: Abb. 2.8 zeigt ein Weg-Zeit-Diagramm, mit der Zeit als vertikaler Dimension. Es bezieht sich auf Züge, die zwischen Amsterdam, Köln, Brüssel und Paris verkehren. Die Züge können entweder von Köln oder von Amsterdam über Brüssel nach Paris fahren. Aachen ist als zusätzlicher Halt auf der Strecke zwischen

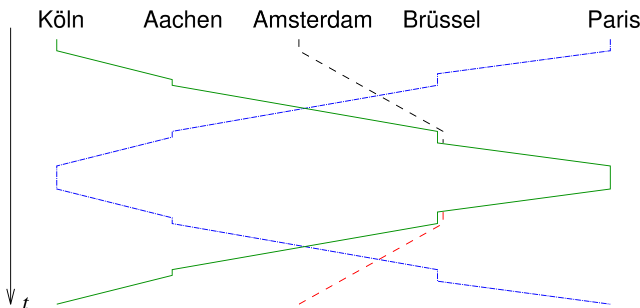


Abb. 2.8 Weg-Zeit-Diagramme

Köln und Brüssel dargestellt. Vertikale Segmente zeigen die Zeit, die der Zug in einem Bahnhof verbringt. In Brüssel gibt es eine zeitliche Überlappung zwischen den Zügen aus Richtung Köln und aus Richtung Amsterdam. Es gibt einen zweiten Zug zwischen Paris und Köln, der keine Beziehung zu einem Zug von/nach Amsterdam hat. Dieses und weitere Beispiele lassen sich mit dem Simulationswerkzeug levi simulieren [498]. ∇

Beispiel 2.6: Ein komplexeres und realistischeres Beispiel ist in Abb. 2.9 dargestellt. Es beschreibt den Zugverkehr im Bereich des Schweizer Lötschbergs [225].

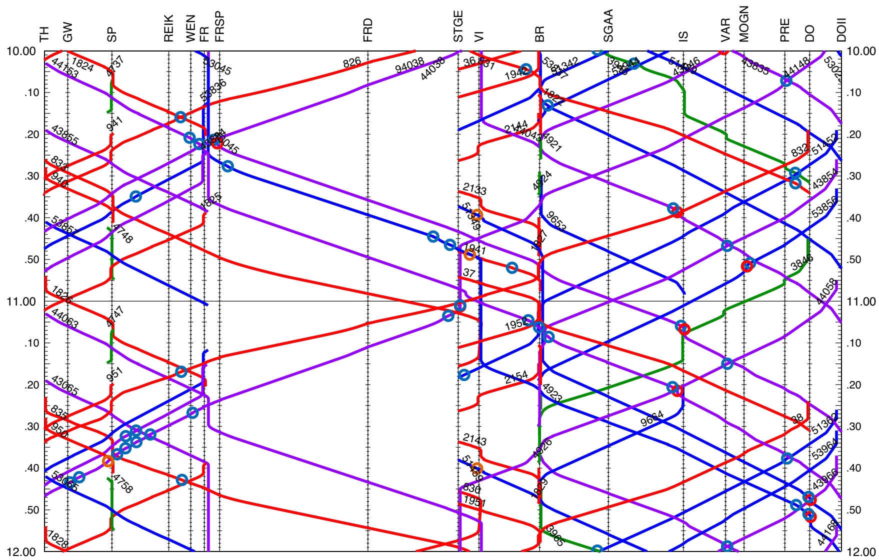


Abb. 2.9 Zugverkehr am Lötschberg (mit freundlicher Genehmigung durch N. Huerlimann, Open-Track Railway Technology GmbH, © OpenTrack Railway Technology GmbH)

Die Namen verschiedener Bahnhöfe sind in der Horizontalen gezeigt. Die vertikale Dimension entspricht der Zeit. Schnelle und langsame Züge können anhand der Steigung der Linien unterschieden werden. Bei langsamen Zügen ist die Steigung groß und enthält möglicherweise sogar vertikale Segmente, was einem Aufenthalt im Bahnhof entspricht. Bei schnellen Zügen sind die Linien fast horizontal. Die Züge halten jeweils nur an einigen der Bahnhöfe. In dem gezeigten Beispiel ist es nicht ersichtlich, ob Überlappungen an Bahnhöfen zufällig erfolgen oder ob eine echte Synchronisation für Umsteigeverbindungen benötigt wird. Außerdem sind zulässige Abweichungen nicht ersichtlich. ∇

SCs und TDDs werden in der Praxis sehr häufig verwendet. Sie sind z.B. für Anwendungen des Internets der Dinge sehr wichtig. SCs enthalten im Unterschied zu TDDs keine Information über physikalische Zeiten. TTDs können dagegen zeitliche Abläufe darstellen, geben aber keine genauen Aussagen zur Synchronisation.

UML wurde ursprünglich nicht für Echtzeitanwendungen entworfen. UML 2.0 enthält **Zeitdiagramme**, welche die Angabe der realen Zeit erlauben. Diese Diagramme erlauben Bezüge zu physikalischen Zeiten, ähnlich wie TDDs. Bestimmte UML-„Profile" (siehe Seite 134) stellen zusätzliche Annotationen für Zeitangaben zur Verfügung [369].

2.3.3 Differentialgleichungen

Differentialgleichungen können in der Sprache der Mathematik beschrieben werden. Varianten davon können als Eingabe für Entwurfswerkzeuge genutzt werden, die entsprechende Modelle unterstützen. Als Beispiel für eine solche Variante nutzen wir Modelica [400], eine Sprache, die auf die Modellierung cyber-physikalischer Systeme zielt. Modelica besitzt sowohl eine graphische wie auch eine textuelle Darstellungsform. Mit der graphischen Form kann man Systeme als Menge verbundener Blöcke beschreiben. Jeder Block wiederum kann durch Gleichungen modelliert werden. Verbindungen zwischen den Blöcken symbolisieren gemeinsame Variablen im Sinne der Mathematik. Die Informationen zu jedem der Blöcke können zusammen mit der Information zu Verbindungen in ein globales Gleichungssystem transformiert werden. Dieser Vorgang kann als „flachklopfen" (engl. *flattening*) der Hierarchie bezeichnet werden. Wie in der Mathematik haben Gleichungen (und Verbindungen) eine bidirektionale Bedeutung.

Beispiel 2.7: Das nachfolgende Modell[8] stellt den springenden Ball dar, der auf Seite 12 vorgestellt wurde:

```
model StickyBall
  type Height = Real(unit = "m");
  type Velocity = Real(unit = "m/s");
  parameter Real s = 0.8 "Restitution";
  parameter Height h0 = 1.0 "Initial height";
  constant Velocity eps = 1e-3 "small velocity";
  Boolean stuck;
  Height h;
  Velocity v;
initial equation
  v = 0;
  h = h0;
  stuck = false;
equation
  v = der(h);
  der(v) = if stuck then 0 else -9.81;
  when h <= 0.0 then
     stuck = abs(v) < eps;
     reinit(v, if stuck then 0 else -s*v);
  end when;
end StickyBall;
```

Im Gleichungsteil wird die Geschwindigkeit `v` als Ableitung der Höhe `h` definiert. Die Ableitung der Geschwindigkeit wird gleich der Erdbeschleunigung gesetzt, solange der Ball noch nicht auf der Oberfläche klebt. Für diese Gleichungen gibt es Anfangs- bzw. Randbedingungen im **`initial equation`** Teil. Die mathematischen Gleichungen können numerisch integriert werden. Diese Prozedur wird in der Beschreibung des Abprallens des Balls ausgenutzt: **`when`**-Klauseln können benutzt

[8] Dieses Modell wurde aus einem Modell von Tiller [541] abgeleitet.

werden, um Ereignisse auszulösen, die während der Berechnung auftreten. Im Beispiel wird ein Ereignis ausgelöst, wenn die Höhe des Balls gleich Null oder kleiner wird. Wenn dieses Ereignis ausgelöst wird und wenn die Geschwindigkeit noch groß genug ist, dann wird die Geschwindigkeit invertiert und aufgrund des Modells des teilelastischen Stoßes um die Stoßzahl s reduziert. Die **`reinit`**-Klausel definiert eine weitere Randbedingung.

Wenn allerdings die Geschwindigkeit kleiner als `eps` ist, nehmen wir an, dass der Ball klebt und die Geschwindigkeit wird auf Null gesetzt, wodurch alle künftigen Aktivitäten unterdrückt werden. Das resultierende Modell kann beispielsweise mit OpenModelica[9] simuliert werden.

Das Weg-Zeit-Diagramm des Balls kann wie folgt berechnet werden: nach dem Loslassen aus der Anfangshöhe fällt der Ball mit einer gleichmäßig beschleunigten Bewegung und besitzt dabei zur Zeit t die folgende Geschwindigkeit und die folgende zurückgelegte Wegstrecke:

$$v = gt \tag{2.1}$$

$$x = \frac{g}{2}t^2 \tag{2.2}$$

Dies gilt bis der Ball bei $x = h_0$ den initialen Stoß erfährt (0-te Reflektion). Wir nennen die dazugehörige Zeit t_0 und die Geschwindigkeit v_0. Aus den Gleichungen (2.1) und (2.2) ergeben sich

$$v_0 = gt_0 \tag{2.3}$$

$$h_0 = \frac{g}{2}t_0^2 \tag{2.4}$$

und daraus

$$t_0 = \frac{v_0}{g} \tag{2.5}$$

$$t_0 = \sqrt{\frac{2}{g}h_0} \tag{2.6}$$

$$v_0 = \sqrt{2gh_0} \tag{2.7}$$

Nach dem Stoß steigt der Ball in die Höhe und besitzt dabei die Geschwindigkeit

$$v = -sv_0 + gt \tag{2.8}$$

bis er die Geschwindigkeit 0 erreicht. Die Zeit bis dahin nennen wir t'_1. Aus der Gleichung (2.8) folgt

$$0 = -sv_0 + gt'_1 \quad \text{bzw.}$$

$$t'_1 = s\frac{v_0}{g} \tag{2.9}$$

[9] Siehe https://openmodelica.org/.

Danach fällt der Ball wieder nach unten und benötigt dabei dieselbe Zeit, wie für das Steigen. Damit erfolgt der nächste Stoß zu einer Zeit

$$t_1 = 2t_1' = 2s\frac{v_0}{g} \tag{2.10}$$

nach dem initialen Stoß.

Aus der Rechnung ist zu sehen, dass die Zeit für das Steigen oder das Fallen aufgrund der Teilelastizität jeweils um den Faktor s kürzer ist als für das Steigen oder Fallen in der vorherigen Iteration. Daher ereignet sich die Reflexion $n \geq 0$ jeweils zum Zeitpunkt

$$t_n = \frac{v_0}{g} + \frac{2v_0}{g}\sum_{k=1}^{n} s^k = \frac{2v_0}{g}\sum_{k=0}^{n} s^k - \frac{v_0}{g} \tag{2.11}$$

Unter der Voraussetzung $s < 1$ konvergiert diese geometrische Reihe gemäß der Theorie konvergenter Reihen zu

$$t_{final} = \lim_{n\to\infty} \frac{2v_0}{g}\sum_{k=0}^{n} s^k - \frac{v_0}{g} = \frac{2v_0}{g(1-s)} - \frac{v_0}{g} \tag{2.12}$$

Dies bedeutet: auch wenn der Ball nicht klebrig ist, gibt es eine obere Schranke für die Zeitpunkte der Reflexionen. Es gibt dann aber keine Schranke für die Anzahl der Reflexionen. Dies entspricht der aus der Mathematik bekannten Tatsache, dass unendliche Reihen zu einem endlichen Wert konvergieren können[10].

Die Benutzung von Gleichungen einschließlich Ableitungen in Modelica bringt uns in die Nähe der Sprache der Mathematik und der Physik. Mit der Einführung von Ereignissen erhalten wir allerdings wieder sequentielles Verhalten. Die implizite numerische Integration führt auch zum Risiko von Problemen mit der numerischen Genauigkeit. Dementsprechend wurde auch die Formulierung `h <= 0.0` gewählt, da wir aufgrund von Genauigkeitsproblemen den Fall `h = 0.0` verpassen können. Genauigkeitsprobleme gibt es auch für das publizierte Modell des nicht-klebrigen Balls [541]: aufgrund beschränkter Genauigkeit liefert Openmodelica eine Lösung, bei welcher der Ball bei großen Zeiten `t` den Boden durchquert! Dieses Problem wird dadurch verursacht, dass für sehr kurze Abstände zwischen den Reflexionen keine Ereignisse erzeugt werden.

Dieses Beispiel zeigt sehr schön die Vorteile und Einschränkungen von Modelica: auf der einen Seite kann man mit Modelica sogar den physikalischen Anteil cyber-physikalischer Systeme modellieren. Auf der anderen Seite benutzen wir nicht genau die Sprache der Mathematik, was zu möglichen Modellierungsproblemen beispielsweise aufgrund numerischer Ungenauigkeiten führt. ∇

[10] Auf dieser Tatsache ist das Paradoxon von Achilles und der Schildkröte [585] aufgebaut, das ebenfalls über eine konvergierende unendliche Reihe modelliert werden kann.

2.4 Kommunizierende endliche Automaten

In den nachfolgenden Abschnitten werden wir nur digitale Systeme betrachten. Verglichen mit den frühen Entwurfsphasen benötigen wir genauere Modelle unseres zu entwerfenden Systems. Auf den Seiten 18 und 34 haben wir bereits erwähnt, dass wir zustandsorientiertes Verhalten beschreiben müssen. Klassisch werden dazu endliche Automaten (kurz Automaten, engl. *Finite State Machines* (FSMs)) verwendet, die über eine Zustandsmenge, ein Eingabealphabet, Ausgaben sowie die Berechnung des Folgezustands modelliert werden. Abb. 2.10 (identisch zu Abb. 2.1) zeigt eine graphische Darstellung als klassisches Zustandsdiagramm.

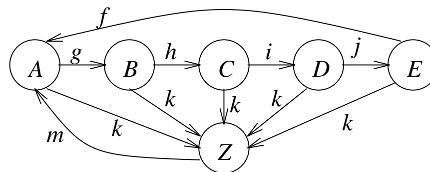


Abb. 2.10 Zustandsdiagramm eines Automaten

Zustände werden durch Kreise gekennzeichnet. Wir betrachten hier nur Automaten, die sich jederzeit in genau einem der möglichen Zustände befinden. Solche Automaten werden **deterministisch** genannt. Die Kantenbeschriftung stellt Ereignisse dar. Angenommen, ein Automat befindet sich in einem bestimmten Zustand und es tritt ein Ereignis ein, das einer der von diesem Zustand ausgehenden Kanten entspricht. Dann geht der Automat über in den Zustand, der sich am Ende der Kante befindet. Automaten können implizit getaktet sein. Solche Automaten werden **synchrone Automaten** genannt. Bei diesen finden Zustandsübergänge nur bei Taktübergängen statt. Automaten können auch Ausgaben erzeugen (in Abb. 2.10 nicht dargestellt). Weitere Informationen über klassische Automaten finden sich z.B. bei Kohavi [302].

2.4.1 Zeitgesteuerte Automaten

Klassische Automaten verfügen über keinerlei Möglichkeiten, Zeit zu modellieren. Sie wurden daher um eine Möglichkeit ergänzt, Zeitinformationen darzustellen. Zeitgesteuerte Automaten (engl. *timed automata*) sind Automaten, die durch reelle Variablen erweitert werden. „Die Variablen modellieren die logischen Uhren des Systems. Diese werden beim Systemstart mit Null initialisiert und laufen dann synchron. An den Kanten angegebene Zeitbeschränkungen schränken das Verhalten des Automaten ein. Ein Übergang über eine Kante kann stattfinden, wenn der aktuelle Zeitwert der Uhr dem an der Kante angegebenen entspricht. Uhren können bei einer Transition auf Null zurückgesetzt werden" [44].

Beispiel 2.8: Abb. 2.11 zeigt ein Beispiel für einen zeitgesteuerten Automaten. Der Anrufbeantworter befindet sich normalerweise im links dargestellten Anfangszustand.

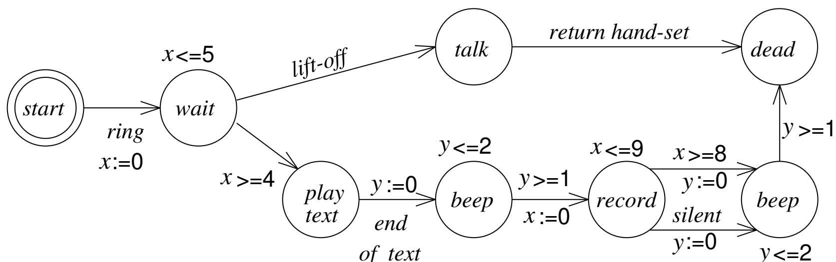


Abb. 2.11 Bearbeitung ankommender Anrufe bei einem Anrufbeantworter

Wenn ein Anruf ankommt, wird die Uhr x auf 0 zurückgesetzt und der Automat wechselt in den Wartezustand *wait*. Wenn der Angerufene den Anruf annimmt, kann ein Gespräch stattfinden, bis der Hörer aufgelegt wird. Ansonsten kann ein Übergang zum Zustand *play text* stattfinden, wenn die Zeit den Wert 4 erreicht hat.

Sobald der Übergang stattgefunden hat, wird eine aufgezeichnete Nachricht abgespielt und mit einem Piepton beendet. Die Uhr y garantiert, dass der Ton mindestens eine Zeiteinheit lang ist. Danach wird die Uhr x wieder auf 0 zurückgesetzt und der Anrufbeantworter beginnt mit der Aufnahme. Wenn die Zeit den Wert 8 erreicht hat oder der Anrufer nichts sagt, kann der nächste Piepton abgespielt werden, der auch wieder mindestens eine Zeiteinheit dauert. Danach findet ein Übergang in den Schlusszustand statt. In diesem Beispiel werden Übergänge durch Eingaben (wie *lift-off*) oder durch **Uhren-Einschränkungen** (engl. *clock constraints*) ausgelöst. ∇

Zeitbedingungen beschreiben Übergänge, die stattfinden **können**, aber nicht müssen. Um sicherzustellen, dass ein Übergang wirklich stattfindet, können zusätzlich **Stelleninvarianten** (engl. *location invariants*) definiert werden. Die Stelleninvarianten $x \leq 5$, $x \leq 9$ und $y \leq 2$ werden im Beispiel verwendet, damit Transitionen spätestens eine Zeiteinheit nach der sie auslösenden Bedingung aktiv werden. Im Beispiel dienen zwei Uhren der Veranschaulichung, eine Uhr wäre ausreichend.

Formal lassen sich zeitgesteuerte Automaten wie folgt definieren [44]: Sei C eine Menge reellwertiger, nicht negativer Variablen, die Uhren darstellen. Sei Σ ein endliches Alphabet möglicher Eingaben.

Definition 2.7: Eine **Zeitbedingung** ist eine konjunktive Formel atomarer Bedingungen der Form $x \circ n$ oder $(x - y) \circ n$ mit $x, y \in C, \circ \in \{\leq, <, =, >, \geq\}$ und $n \in \mathbb{N}$.

Die verwendeten Konstanten n der Bedingungen müssen ganzzahlig sein, auch wenn die Uhren reellwertig sind. Eine Erweiterung auf rationale Konstanten wäre einfach, da diese sich einfach durch Multiplikation in ganze Zahlen wandeln lassen. Sei $B(C)$ die Menge an Zeitbedingungen.

Definition 2.8 (Bengtson [44]): Ein **zeitgesteuerter Automat** ist ein Tupel (S, s_0, E, I) mit:

- einer endlichen Menge an Zuständen S,
- einem Anfangszustand s_0,
- der Kantenmenge $E \subseteq S \times B(C) \times \Sigma \times 2^C \times S$. $B(C)$ modelliert die konjunktive Bedingung, die gelten muss, und Σ modelliert den Eingang, der für die Aktivierung eines Übergangs notwendig ist. 2^C stellt die Menge an Uhrvariablen dar, die zurückgesetzt werden, wenn die Transition aktiv wird.
- $I : S \rightarrow B(C)$ ist die Menge der Invarianten für jeden einzelnen Zustand. $B(C)$ stellt die für einen bestimmten Zustand S zutreffende Invariante dar. Diese Invariante wird durch eine konjunktive Formel beschrieben.

Diese erste Definition wird meist erweitert, um parallel arbeitende zeitgesteuerte Automaten beschreiben zu können. Zeitgesteuerte Automaten mit einer großen Anzahl von Uhren sind meist schwer zu verstehen. Weitere Details über zeitgesteuerte Automaten finden sich z.B. in Veröffentlichungen von Dill et al. [133] und Bengtsson et al. [44].

Die Simulation und Verifikation von zeitbehafteten Automaten ist mit dem populären Werkzeug UPPAAL möglich[11]. UPPAAL unterstützt Nebenläufigkeit und Datenvariablen.

Zeitgesteuerte Automaten erweitern klassische Automaten um Zeitinformation. Sie erfüllen damit aber viele unserer anderen Anforderungen an Spezifikationstechniken nicht. Insbesondere verfügen sie in der hier beschriebenen Standardform weder über Hierarchie noch über Nebenläufigkeit.

2.4.2 StateCharts: implizite Kommunikation über gemeinsamen Speicher

Die hier vorgestellte StateCharts-Sprache ist ein bekanntes Beispiel einer automatenbasierten Sprache, die hierarchische Modelle und Nebenläufigkeit unterstützt. Sie beinhaltet zudem eingeschränkte Möglichkeiten zur Angabe von Zeitinformationen.

StateCharts wurde 1987 von David Harel [204] vorgestellt und später detaillierter beschrieben [141]. Den Namen hat Harel angeblich so gewählt, weil es „die einzige unbenutzte Kombination von *flow* oder *state* mit *diagram* oder *chart* “ war.

Modellierung von Hierarchie

Die StateCharts-Sprache beschreibt erweiterte endliche Automaten, sie ist daher gut dazu geeignet, zustandsorientiertes Verhalten abzubilden. Die wichtigste Erweite-

[11] Die akademische Version ist unter http://www.uppaal.org und die kommerzielle Version ist unter http://www.uppaal.com erhältlich.

rung gegenüber klassischen Automaten ist das Konzept der **Hierarchie**. Hierarchie wird mit Hilfe von **Superzuständen** (engl. *superstates*) eingeführt.

Definition 2.9: Zustände, die andere Zustände enthalten, heißen **Superzustände**

Definition 2.10: Zustände, die in anderen Zuständen enthalten sind, heißen **Unterzustände** (engl. *substates*) der Superzustände.

Beispiel 2.9: Abb. 2.12 zeigt ein StateCharts-Beispiel. Es ist eine hierarchische Version von Abb. 2.10. Der Superzustand S beinhaltet die Unterzustände A, B, C, D

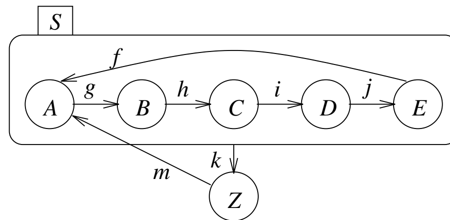


Abb. 2.12 Hierarchisches Zustandsdiagramm

und E. Angenommen, der Automat befindet sich in Zustand Z (wir bezeichnen Z in diesem Fall auch als **aktiven Zustand**). Wenn dann die Eingabe m am Automaten erfolgt, ist A der nächste Zustand. Wenn sich der Automat in Zustand S befindet und es folgt die Eingabe k, so ist Z der nächste Zustand – unabhängig davon, in welchem der Unterzustände A, B, C, D oder E sich der Automat tatsächlich befindet. In diesem Beispiel sind alle in S enthaltenen Zustände nicht-hierarchische Zustände. ∇

Im Allgemeinen könnten die Unterzustände von S auch selbst wieder Superzustände sein, die weitere Unterzustände enthalten. **Immer, wenn ein Unterzustand eines Superzustands aktiv ist, ist auch der zugehörige Superzustand aktiv.**

Definition 2.11: Jeder Zustand, der nicht aus anderen Zuständen besteht, heißt **Basiszustand**.

Der endliche Automat in Abb. 2.12 kann sich zu einem bestimmten Zeitpunkt nur in genau einem der Unterzustände des Superzustands S befinden. Solche Superzustände heißen **ODER-Superzustände**[12].

Definition 2.12: Superzustände S heißen **ODER-Superzustände**, wenn das System, das S enthält, sich in genau einem Unterzustand von S befindet, solange es sich in S befindet.

In Abb. 2.12 könnte die Eingabe k einer Ausnahme entsprechen, wegen der Zustand S verlassen werden muss. Das Beispiel zeigt bereits, wie solche Ausnahmebehandlungen durch Verwendung von Hierarchie kompakt dargestellt werden können.

[12] Gemeint ist hier immer ein sich gegenseitig ausschließendes ODER.

StateCharts erlaubt es, Systeme hierarchisch zu beschreiben, indem eine Beschreibung des Systems Untersysteme enthält, die wiederum Beschreibungen von Unterzuständen enthalten können und so weiter. Das **hierarchische** Gesamtsystem kann somit in Form eines **Baumes** dargestellt werden. Die **Wurzel** dieses Baumes entspricht dem gesamten System und die inneren Knoten entsprechen hierarchischen Beschreibungen (die im Falle von StateCharts Superzustände heißen). Die **Blätter** der Hierarchie sind nicht-hierarchische Beschreibungen (die im Falle von StateCharts **Basiszustände** heißen).

Bisher haben wir explizite, direkte Kanten verwendet, die zu Basiszuständen führen, um den neuen Zustand zu bestimmen. Der Nachteil dieser Art der Modellierung liegt darin, dass man die internen Strukturen der Superzustände nicht vor der Umgebung verstecken kann. In einem echten hierarchischen Modell sollte es möglich sein, die interne Struktur zu verstecken, sodass diese später beschrieben oder sogar geändert werden kann, ohne eine Auswirkung auf die Umgebung zu haben. Dies wird durch zusätzliche Mechanismen ermöglicht, die den neuen Zustand bestimmen.

Der erste zusätzliche Mechanismus ist der **Standardzustand** (engl. *default state*). Er kann in Superzuständen verwendet werden, um anzuzeigen, welche der Unterzustände betreten werden, wenn der Superzustand betreten wird. In den Diagrammen wird der Standardzustand mit Hilfe einer Kante bezeichnet, die von einem kleinen ausgefüllten Kreis zum jeweiligen Zustand geht.

Beispiel 2.10: Abb. 2.13 zeigt ein Zustandsdiagramm, das den Standardzustands-Mechanismus verwendet. Es ist äquivalent zum Diagramm in Abb. 2.12. Der kleine ausgefüllte Kreis stellt dabei selber keinen Zustand dar.

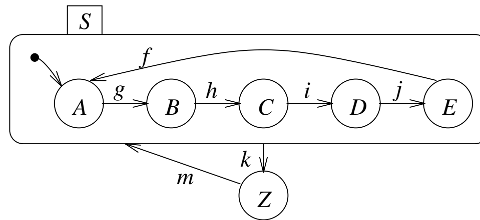
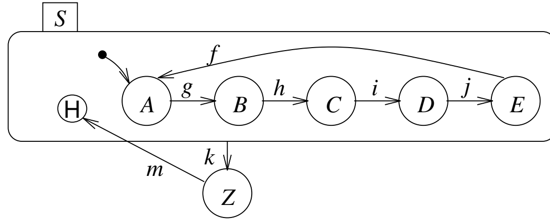


Abb. 2.13 Zustandsdiagrammmit Standardzustand ∇

Ein weiterer Mechanismus, um den nächsten Zustand anzugeben, ist der sogenannte ***History*-Zustand**. Mit Hilfe dieses Konstrukts ist es möglich, in den letzten Unterzustand zurückzukehren, der aktiv war, bevor der Superzustand verlassen wurde. Der *History*-Mechanismus wird durch den Buchstaben H in einem Kreis dargestellt. Um den aktiven Zustand für den ersten Wechsel in einen Superzustand zu kennzeichnen, wird der *History*-Mechanismus oft mit einem Standardzustand kombiniert.

Beispiel 2.11: Eine Kombination von *History*- und Standardzustand wird in Abb. 2.14 gezeigt. Das Verhalten des endlichen Automaten hat sich nun verändert: Sei der Automat zunächst im Zustand Z und es ereigne sich die Eingabe m. Dann ist

Abb. 2.14 Zustandsdiagramm mit *History*-Mechanismus und Standardzustand



A der nächste Zustand, wenn der Superzustand S zum ersten Mal betreten wird. Ansonsten wird der zuletzt aktive Unterzustand betreten. Dieser Mechanismus hat viele Anwendungen. Wenn z.B. die Eingabe k eine Ausnahme darstellt, könnte die Eingabe m verwendet werden, um in den Zustand vor der Ausnahme zurückzukehren. Die Zustände A, B, C, D und E könnten den Zustand Z auch wie eine Prozedur aufrufen. Nachdem diese „Prozedur" Z abgearbeitet ist, kehrt der Automat zum aufrufenden Zustand zurück. Auf diese Weise fügen wir StateCharts ein Element üblicher Programmiersprachen hinzu.

Der Automat aus Abb. 2.14 kann auch wie in Abb. 2.15 dargestellt werden. In diesem Fall wurden die Darstellungen für den Standardzustand und den *History*-Mechanismus kombiniert.

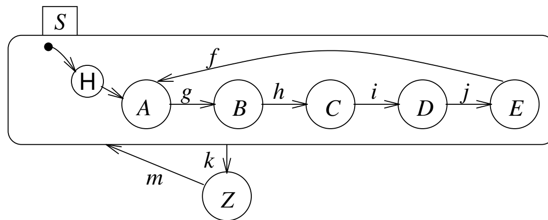


Abb. 2.15 Kombination der Symbole für *History*- und Standardzustand ∇

Eine Spezifikationstechnik muss auch in der Lage sein, Nebenläufigkeit und Parallelität darzustellen. Zu diesem Zweck gibt es in StateCharts eine zweite Art von Superzuständen, die sogenannten **UND-Superzustände**.

Definition 2.13: Superzustände S heißen **UND-Superzustände**, wenn das System, das S enthält, sich in allen Unterzuständen von S gleichzeitig befindet, solange es sich in S befindet.

Beispiel 2.12: Das Diagramm für einen Anrufbeantworter in Abb. 2.16 enthält einen UND-Superzustand. Ein Anrufbeantworter muss normalerweise zwei Aufgaben par-

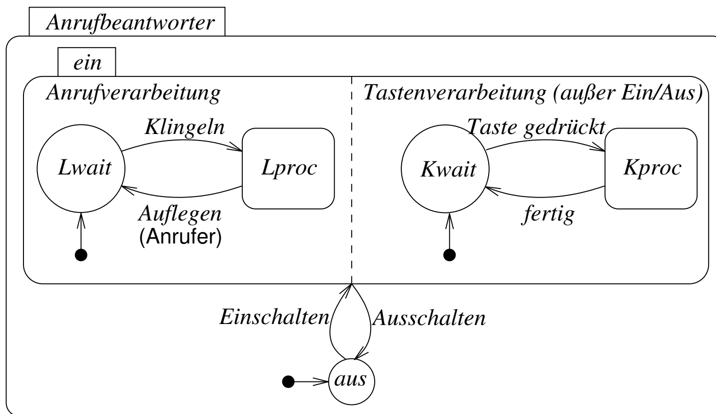


Abb. 2.16 Anrufbeantworter

allel ausführen: er wartet auf ankommende Anrufe und überprüft gleichzeitig, ob der Benutzer etwas auf den Eingabetasten eingegeben hat. In Abb. 2.16 heißen die entsprechenden Zustände *Lwait* und *Kwait*. Ankommende Anrufe werden im Zustand *Lproc* abgearbeitet, während Reaktionen auf die Eingabetasten im Zustand *Kproc* erzeugt werden. Der Zustand *Lproc* wird verlassen, wenn der Anrufer auflegt. Legt der Angerufene auf, so wird nicht automatisch zum Zustand *Lwait* zurückgekehrt. Bei unerwünschten Anrufen kann er also bei diesem Modell nicht einfach den Anfangszustand wiederherstellen.

Der Ein/Ausschalter wird fürs Erste so modelliert, dass die von ihm erzeugten Ereignisse (*Einschalten* und *Ausschalten*) separat dekodiert werden und somit nicht in den Zustand *Kproc* gewechselt wird. Wird der Anrufbeantworter ausgeschaltet, so werden die beiden im *ein*-Zustand enthaltenen Zustände verlassen. Sie werden erst wieder betreten, wenn der Anrufbeantworter wieder eingeschaltet wird. Zu diesem Zeitpunkt werden dann die Standardzustände *Lwait* und *Kwait* betreten. Solange die Maschine eingeschaltet ist, befindet sie sich immer in den beiden Unterzuständen des Zustands *ein*. ∇

Für UND-Superzustände können die Unterzustände, die als Reaktion auf ein Ereignis betreten werden, unabhängig voneinander beschrieben werden. Jede beliebige Kombination von *History*-und Standardzuständen sowie expliziten Transitionen ist möglich. Für das Verständnis von StateCharts ist es wichtig zu begreifen, dass immer **alle** Unterzustände betreten werden, auch wenn es nur eine einzige explizite Transition zu einem der Unterzustände gibt. Analog dazu gilt, dass eine Transition aus einem UND-Superzustand heraus immer dazu führt, dass **alle** seine Unterzustände verlassen werden.

Beispiel 2.13: Als Beispiel modifizieren wir unseren Anrufbeantworter so, dass der Ein/Ausschalter wie alle anderen Bedientasten im Zustand *Kproc* dekodiert und behandelt wird (siehe Abb. 2.17). Wenn der Anrufbeantworter ausgeschaltet wird,

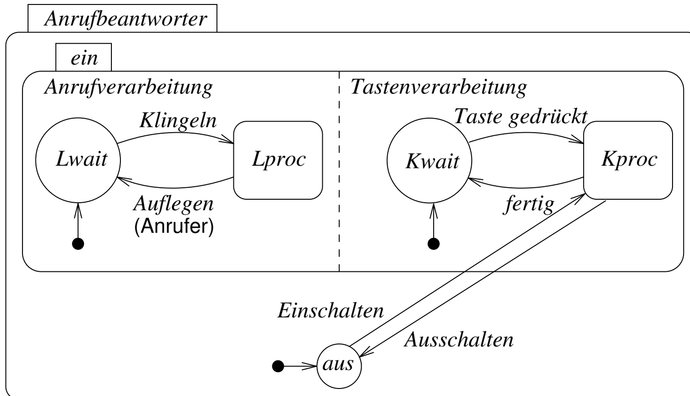


Abb. 2.17 Anrufbeantworter mit veränderter Ein/Ausschalter-Modellierung

findet eine Transition in den *aus*-Zustand statt. Dieser Übergang führt dazu, dass auch der Zustand, der auf ankommende Anrufe wartet, verlassen wird. Das Wiedereinschalten der Maschine führt dazu, dass eben dieser Zustand auch wieder mit betreten wird. ∇

UND-Superzustände sind der wichtigste Mechanismus, um Nebenläufigkeit in StateCharts zu beschreiben. Jeder Unterzustand kann als eigener Automat angesehen werden. Diese Automaten kommunizieren miteinander und werden damit zu **kommunizierenden endlichen Automaten** (engl. *Communicating Finite State Machines* (CFSMs)). Dieser Begriff wurde als Titel dieses Abschnitts gewählt.

Zusammenfassend können wir festhalten: **Zustände in StateCharts sind entweder UND-Superzustände, ODER-Superzustände oder Basiszustände.**

Zeitgeber

Da es notwendig ist, in eingebetteten Systemen Zeitbedingungen zu modellieren, bietet StateCharts die sogenannten *Timer* oder **Zeitgeber** an. Zeitgeber werden durch das gezackte Symbol in Abb. 2.18 dargestellt.
Wenn das System für die im *Timer* angegebene Zeitdauer im *Timer*-Zustand war, wird ein *Timeout* ausgelöst und das System verlässt diesen Zustand. Zeitgeber können auch hierarchisch verwendet werden.

Ein Zeitgeber könnte beispielsweise in einer tieferen Hierarchiestufe des Anrufbeantworters verwendet werden, um das Verhalten des Zustands *Lproc* zu beschreiben.

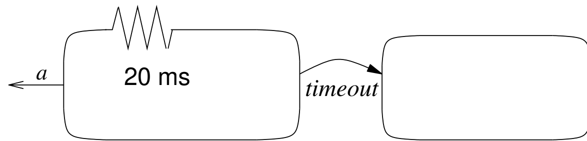


Abb. 2.18 Zeitgeber in StateCharts

Abb. 2.19 zeigt eine mögliche Beschreibung für diesen Zustand. Die Spezifikation ist damit leicht anders als die in Abb. 2.11. Da das Auflegen des Anrufers in

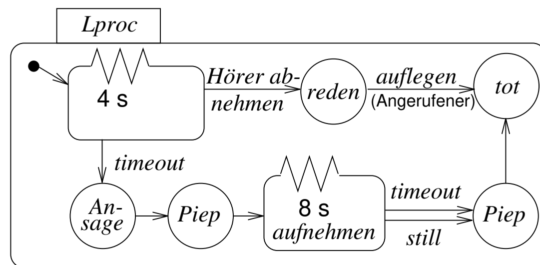


Abb. 2.19 Behandlung von eingehenden Anrufen in *Lproc*

Abb. 2.16 in Form einer Ausnahmebehandlung realisiert ist, wird der Zustand *Lproc* immer erst dann verlassen, wenn der Anrufer auflegt. Wenn allerdings der Angerufene auflegt, hat der Entwurf des Zustands *Lproc* einen Schönheitsfehler: wenn der Angerufene zuerst auflegt, ist das Telefon solange tot (und still), bis der Anrufer ebenfalls aufgelegt hat.

StateCharts beinhalten noch weitere Sprachelemente, die beispielsweise im Buch von Harel [204] zu finden sind. Zusammen mit Drusinsky gibt Harel [141] in einem Artikel eine genauere Beschreibung der Semantik von StateMate, einer StateCharts-Implementierung.

Kantenbeschriftungen und die *StateMate*-Semantik

Die Ausgabe der erweiterten Automatenmodelle wurde bislang noch nicht betrachtet. Solche Ausgaben können mit Hilfe von Kantenbeschriftungen realisiert werden. Die allgemeine Form einer Kantenbeschriftung ist „*Ereignis* [*Bedingung*] / *Reaktion*“. Alle drei Bestandteile der Beschriftung sind optional. Die **Reaktion** beschreibt die Reaktion des Automaten auf den Zustandsübergang. Mögliche Reaktionen beinhalten das Erzeugen von Ereignissen oder die Zuweisung von Variablenwerten. *Bedingung* und *Ereignis* beschreiben zusammen die Eingaben an den Automaten. Die **Bedingung** beschreibt das Überprüfen von Werten von Variablen oder des Zustands des Gesamtsystems. Der **Ereignisteil** symbolisiert, auf welches Ereignis zu prüfen ist. Solche Ereignisse können entweder intern oder extern erzeugt werden. Interne Ereignisse werden als Ergebnis von Zustandsübergängen erzeugt und in der

Reaktion des Übergangs beschrieben. Externe Ereignisse werden üblicherweise in der Systemumgebung beschrieben. Beispiele:

- *Einschalten* / *Ein*:=1 (Test auf ein Ereignis und Wertzuweisung an eine Variable),
- [*Ein*=1] (Überprüfung eines Variablenwerts),
- *Ausschalten* [*not in Lproc*] / *Ein*:=0 (Ereignistest, Überprüfung eines Zustands, Variablenzuweisung. Die Zuweisung wird nur durchgeführt, wenn das Ereignis stattgefunden hat und die Bedingung erfüllt ist).

Die Semantik der Kantenbeschriftungen kann nur im Zusammenhang mit der generellen Semantik von StateCharts erklärt werden. Die StateMate-Implementierung von StateCharts [141] geht von einer schrittweisen Ausführung von StateCharts-Beschreibungen aus, wie in Abb. 2.20 gezeigt. Schritte werden jedes Mal ausgeführt,

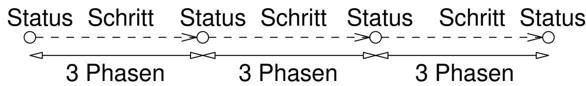


Abb. 2.20 Schritte während der Ausführung eines StateMate-Modells

wenn ein Ereignis eintritt oder sich ein Variablenwert geändert hat. Unter dem **Status** verstehen wir die Menge aller Variablenwerte, die Menge der erzeugten Ereignisse, die Menge der aktuellen Zustände und die aktuelle Zeit.

Das Konzept der Schritte erlaubt es, die Semantik von **Ereignissen** genauer zu definieren. **Die Sichtbarkeit von Ereignissen ist auf denjenigen Schritt beschränkt, der auf den Schritt folgt, in dem das Ereignis erzeugt wurde.** Somit verhalten sich Ereignisse wie Bitwerte, die bei einer Taktflanke in Registern abgelegt werden und einen Einfluss auf die bei der nächsten Taktflanke gespeicherten Werte haben. Ereignisse „leben" nur bis zum nächsten Schritt bzw. bis zur nächsten Taktflanke.

Im Gegensatz dazu behalten Variablen ihren Wert, bis ihnen ein neuer Wert zugewiesen wird. Nach der StateMate-Semantik sind neue Werte von Variablen in allen Teilen des Modells sichtbar, und zwar ab demjenigen Schritt, der auf den Schritt folgt, in dem die Wertzuweisung erfolgt ist. Das bedeutet, dass die Semantik von StateMate neue Variablenwerte zwischen zwei Schritten überall im Modell propagiert und sichtbar macht. Jeder Schritt besteht aus drei Phasen:

1. In der ersten Phase wird der Einfluss von externen Bedingungen und Ereignissen ausgewertet. Das beinhaltet auch die Auswertung von Funktionen, die von externen Ereignissen abhängen. In dieser Phase gibt es keine Zustandsübergänge. In unseren einfachen Beispielen wird diese Phase nicht unbedingt benötigt.
2. Die nächste Phase besteht darin, die Menge der Zustandsübergänge zu bestimmen, die im aktuellen Schritt ausgeführt werden sollen. Variablenzuweisungen werden ausgewertet, aber die neuen Werte werden nur temporären Hilfsvariablen zugewiesen.

3. In der dritten Phase finden die Zustandsübergänge statt und die Variablen erhalten ihre neuen Werte.

Die Aufteilung in die Phasen 2 und 3 ist besonders wichtig, um ein deterministisches und reproduzierbares Verhalten von StateCharts-Modellen zu erreichen.

Beispiel 2.14: Wir betrachten das StateCharts-Modell in Abb. 2.21.

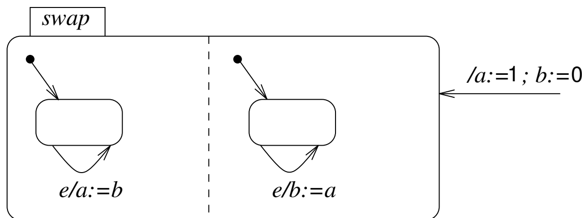


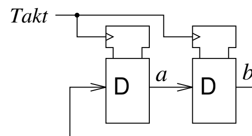
Abb. 2.21 Gegenseitige, abhängige Wertzuweisung

In der zweiten Phase werden bei Eintreten des Ereignisses *e* die neuen Werte für *a* und *b* zunächst in temporären Variablen, z.B. *a'* und *b'*, zwischengespeichert. In der letzten Phase werden diese Zwischenwerte dann in die eigentlichen Variablen kopiert:

```
phase 2: a':=b; b':=a;
phase 3: a:=a'; b:=b';
```

Dadurch werden die Werte der beiden Variablen jedes Mal vertauscht, wenn das Ereignis *e* eintritt. Dieses Verhalten entspricht zwei über Kreuz verbundenen

Abb. 2.22 Über Kreuz verbundene Register



Registern (eines für jede Variable), die an der gleichen Taktleitung angeschlossen sind (siehe Abb. 2.22) und modelliert das Verhalten eines getakteten Schaltwerks, das diese zwei Register enthält[13]. ∇

[13] Wir verwenden in diesem Buch durchgehend in allen Schaltungen die IEEE-Standardsymbole [239] für Gatter und Register. Die Symbole in Abb. 2.22 stellen getaktete Register vom D-Typ dar.

Ohne die Aufteilung in zwei Phasen würde beiden Variablen derselbe Wert zugewiesen. Das Ergebnis würde von der Reihenfolge der Ausführung der Wertzuweisungen abhängen. Diese Aufteilung in (mindestens) zwei Phasen ist typisch für Sprachen, die das Verhalten von synchroner Hardware nachbilden. Eine ähnliche Aufteilung findet sich auch in VHDL (siehe Seite 118). Durch diese Trennung hängen die Ergebnisse nicht von der Reihenfolge ab, in der Teile des Modells in der Simulation ausgeführt werden. Dies ist eine extrem wichtige Eigenschaft. Ohne diese würden verschiedene Simulationsläufe unterschiedliche Ergebnisse erzeugen, die alle als korrekt angesehen werden können. Dies wäre eine sehr verwirrende Eigenschaft des Modells und entspricht nicht dem, was wir von einer Simulation einer realen Schaltung mit festem Verhalten erwarten.

- Kahn [279] nennt diese Eigenschaft „***determinate***".
- In anderen Veröffentlichungen wird die Eigenschaft **deterministisch** genannt. Dieser Begriff ist aber mit mehreren Bedeutungen überladen:
  - Der Begriff bezeichnet nichtdeterministische endliche Automaten, die sich zugleich in mehreren Zuständen befinden können [222].
  - Sprachen können nichtdeterministische Operatoren besitzen. Diese Operatoren besitzen mehrere zulässige Implementierungen.
  - Viele Autoren bezeichnen ein System als nichtdeterministisch, wenn sein Verhalten von einer Eingabe zur Laufzeit abhängt.
  - In der Bedeutung von Kahns Begriff „*determinate*".

In diesem Buch verwenden wir den Begriff „deterministisch" im Sinne von Kahns Begriff „*determinate*". StateMate-Modelle können nur deterministisch sein, wenn es keine anderen Gründe für undefiniertes Verhalten gibt. So könnten z.B. Konflikte zwischen verschiedenen Transitionen erlaubt sein, wie in Abb. 2.23 gezeigt. Wenn in

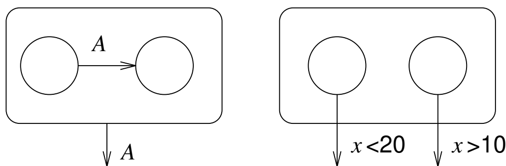


Abb. 2.23 **Links**: Konflikt zwischen Schachtelungstiefen; **rechts**: Konflikt auf derselben Tiefe

Abb. 2.23 (links) das Ereignis A eintritt, während sich das System im linken Zustand befindet, müssen wir bestimmen, welche Transition ausgeführt werden soll. Wenn diese Konflikte auf beliebige Weise aufgelöst würden, wäre das Verhalten nichtdeterministisch. Meist werden deshalb Prioritäten definiert, damit diese Art von Konflikt aufgelöst werden kann. In Abb. 2.23 (rechts) liegt ein Konflikt beispielsweise für den Fall $x = 15$ vor. Solche Konflikte sind nur schwer zu erkennen. Um deterministisches Verhalten zu erreichen, dürfen keine Konflikte vorhanden sein, die in einer beliebigen Weise aufgelöst werden.

Es mag aber auch Fälle geben, in denen wir nichtdeterministisches Verhalten beschreiben wollen, z.B. wenn von zwei verschiedenen Eingängen gelesen werden

kann. In einem solchen Fall möchten wir explizit angeben können, dass diese Entscheidung zur Laufzeit stattfinden wird (vergleiche die **select**-Anweisung von Ada auf Seite 124).

Andere Implementierungen hierarchischer StateCharts zeigen meist nicht dieses von StateMate realisierte deterministische Verhalten. Diese Implementierungen entsprechen einer Softwaresicht hierarchischer StateCharts. In solchen Implementierungen werden Wahlmöglichkeiten meist nicht explizit beschrieben.

Bewertung und Erweiterungen

Implizit geht StateCharts gemäß StateMate-Semantik also von einem *Broadcast*-Mechanismus zum Aktualisieren von Variablenwerten aus. Dadurch lassen sich StateCharts oder StateMate-Modelle leicht auf *shared memory*-Systemen implementieren, eignen sich aber weniger für verteilte und nachrichtenbasierte Systeme. Somit setzen Sprachen wie StateCharts implizit die *shared memory*-Semantik voraus, auch wenn dies nicht explizit festgelegt wird. Bei verteilten Systemen ist es sehr schwierig, alle Variablenwerte zwischen zwei Schritten zu aktualisieren. Aufgrund dieses *Broadcast*-Mechanismus sind StateCharts mit der beschriebenen Semantik nicht geeignet, um verteilte Systeme zu beschreiben. Das Hauptanwendungsgebiet von StateCharts sind daher lokale Systeme, die hauptsächlich Steuerungsaufgaben durchführen. Die Möglichkeit, Hierarchieebenen beliebig zu verschachteln und mit frei wählbaren UND- und ODER-Superzuständen zu versehen, ist der Hauptvorteil von StateCharts. Ein weiterer Vorteil ist die ausreichend präzise Definition der Semantik von StateCharts [141] gemäß StateMate-Implementierung. Außerdem ist eine Vielzahl von kommerziellen Programmen erhältlich, die StateCharts verwenden. StateMate [230] und StateFlow [383] sind Beispiele für solche kommerziellen Anwendungen. Viele dieser Programme sind in der Lage, aus StateCharts-Modellen äquivalente Beschreibungen in C oder VHDL (siehe Seite 109) zu erzeugen. Aus VHDL kann mit Hilfe von Synthesewerkzeugen direkt Hardware erzeugt werden. Aus diesem Grund bieten Programme, die auf StateCharts basieren, einen vollständigen Entwurfspfad von der StateCharts-Beschreibung bis zur fertigen Hardware. Generierte C-Programme können übersetzt und ausgeführt werden, was auch Softwarerealisierungen ermöglicht.

Leider ist die Effizienz dieser automatisch erzeugten Realisierungen häufig ein Problem. So könnten in einer automatisch erzeugten Software etwa Unterzustände von UND-Superzuständen auf Unix-Prozesse abgebildet werden. Dieses Konzept ist für die Simulation von StateCharts geeignet, nicht jedoch für eine effiziente Realisierung auf kleinen, leistungsschwachen Prozessoren. Der Produktivitätsgewinn durch objektorientierte Programmierung kann mit StateCharts nicht ausgeschöpft werden, da Objektorientierung nicht unterstützt wird. Außerdem ist es durch den *Broadcast*-Mechanismus kaum für verteilte Anwendungen geeignet. StateCharts unterstützen keine Programmiersprachenkonstrukte, um komplexe Berechnungsvorschriften zu realisieren. Außerdem können sie keine Hardwarestrukturen oder nicht-funktionales Verhalten beschreiben.

Kommerzielle Implementierungen von StateCharts bieten üblicherweise einige Mechanismen, um die Nachteile von StateCharts zu umgehen. So kann etwa C-Code verwendet werden, um Programmiersprachenkonstrukte zu unterstützen. In StateMate stellen die sogenannten *Module Charts* Hardwarestrukturen dar.

In Bezug auf Zeitbedingungen ermöglichen StateCharts nur die Angabe von *Timeouts*. Es gibt keine weitere direkte Möglichkeit, andere Zeitbedingungen anzugeben.

UML beinhaltet eine Version von StateCharts und erlaubt damit die Modellierung endlicher Automaten. UML Version 1 nennt diese ***state diagrams***, während sie ab UML 2.0 ***state machine diagrams*** genannt werden. Leider unterscheidet sich die Semantik der UML-Diagramme von StateMate, da UML die drei beschriebenen Simulationsphasen nicht vorgibt.

2.4.3 Synchrone Sprachen

Motivation

Es ist schwierig, komplexe Systeme mit Hilfe endlicher Automaten zu beschreiben, da diese keine komplexen Berechnungen enthalten. Standard-Programmiersprachen dagegen können komplexe Berechnungen beschreiben, garantieren dafür aber meist nicht die Ausführungsreihenfolge mehrerer *Threads*. In einer *Multi-Threaded*-Umgebung mit präemptivem (d.h. verdrängendem) *Scheduling* sind viele verschiedene Ablaufreihenfolgen der einzelnen Berechnungen möglich. Es ist schwer, alle möglichen Abläufe in einem solchen nebenläufigen System zu verstehen. Einer der Hauptgründe dafür ist, dass im Allgemeinen mehrere verschiedene Ausführungsreihenfolgen möglich sind, die Reihenfolge der Ausführung ist also nicht festgelegt. Allerdings kann diese Reihenfolge durchaus das Ergebnis beeinflussen. Das sich daraus ergebende nichtdeterministische Verhalten kann eine Reihe von Problemen zur Folge haben, wie z.B. das Problem, einen bestimmten Entwurf zu verifizieren. Bei verteilten Systemen mit unabhängigen Uhren ist deterministisches Verhalten nur schwer realisierbar. Bei nicht verteilten Systemen können wir aber versuchen, die Probleme einer unnötig nichtdeterministischen Semantik zu vermeiden.

Synchrone Sprachen verbinden endliche Automaten und Programmiersprachen zu einem Modell. Sie können komplexe Berechnungen beschreiben, folgen aber dem zu Grunde liegenden Ausführungsmodell endlicher Automaten. Dabei beschreiben sie nebenläufig arbeitende Automaten. Deterministisches Verhalten wird durch die folgende wichtige Eigenschaft erreicht: „wenn Automaten parallel kombiniert werden, dann besteht eine Transition des zusammengesetzten Automaten aus der ‚gleichzeitigen' Transition aller einzelnen Automaten" [198]. Das bedeutet, dass man nicht alle möglichen Abfolgen von Zustandsübergängen aller Automaten betrachten muss, wie dies bei unabhängigen Takten in den Unterautomaten der Fall wäre. Stattdessen kann man die Existenz eines einzigen globalen Taktes annehmen. Bei jedem Taktsignal werden alle Eingaben berücksichtigt, neue Ausgaben und Zustände wer-

den berechnet und die entsprechenden Zustandsübergänge ausgeführt. Dazu ist ein schneller *Broadcast*-Mechanismus notwendig, der alle Teile des Modells erreicht. Diese idealisierte Betrachtung von Gleichzeitigkeit hat den Vorteil, dass dadurch **deterministisches Verhalten** garantiert wird. Es stellt eine Einschränkung des allgemeinen Modells kommunizierender Automaten dar, bei dem jeder Automat seinen eigenen Takt haben darf. Synchrone Sprachen stellen das Prinzip der Gleichzeitigkeit in synchroner Hardware dar und repräsentieren die Semantik von Sprachen für Industriesteuerungen wie IEC 60848 [232] und STEP 7 [488]. Ein Überblick über synchrone Sprachen ist bei Potop-Butucaru et al. [458] zu finden.

Beispiele für synchrone Sprachen: Esterel, Lustre und SCADE

Ein deterministisches Verhalten für alle Sprachkonstrukte zu garantieren, war eines der Hauptziele bei der Entwicklung der synchronen Sprachen Esterel [154, 61] , Lustre [200] und Quartz [480].

Esterel ist eine reaktive Sprache: wenn Esterel-Modelle mit einem Eingabeereignis aktiviert werden, reagieren sie mit der Erzeugung eines Ausgabeereignisses. Esterel ist eine synchrone Sprache: es wird angenommen, dass alle Reaktionen ohne Zeitverzögerung abgeschlossen werden, und dass es ausreichend ist, das Verhalten zu diskreten Zeitpunkten zu analysieren. Dieses idealisierte Modell vermeidet die Probleme von überlappenden Zeiträumen und von Ereignissen, die ankommen, während die vorhergehende Reaktion noch nicht abgeschlossen war. Wie andere Sprachen auch, hat Esterel einen Parallelisierungsoperator, der als || geschrieben wird. Wie in StateCharts findet Kommunikation über einen *Broadcast*-Mechanismus statt. Im Gegensatz zu StateCharts findet die Kommunikation allerdings augenblicklich, ohne jede Verzögerung, statt. Das bedeutet, dass alle Signale, die zu einem bestimmten Zeitpunkt erzeugt werden, in genau diesem Zeitpunkt auch von allen anderen Teilen des Modells gesehen werden. Wenn diese anderen Teile sensitiv auf die erzeugten Signale reagieren, reagieren sie auch in genau diesem einen Zeitpunkt. Dabei können mehrere Runden von Auswertungen erforderlich sein, bis schließlich ein stabiler Zustand erreicht wird. Die dadurch maximal mögliche Antwortzeit wird beispielsweise von Boldt et al. berechnet [56]. Diese Weiterleitung von Werten während ein und derselben makroskopischen Zeiteinheit entspricht der Erzeugung des nächsten Zustands für den gleichen Zeitpunkt in StateCharts, wobei der *Broadcast* hier augenblicklich geschieht. Für weitere aktuelle Informationen zu Esterel verweisen wir auf die Web-Seite [154].

Esterel und Lustre verwenden eine unterschiedliche Syntax, um kommunizierende endliche Automaten zu beschreiben. Esterel sieht dabei wie eine imperative Programmiersprache aus, wogegen Lustre eher an eine Datenflusssprache erinnert (siehe Seite 75 für eine Beschreibung von Datenflussmodellen). Eine grafische Version von Esterel ist mit SyncCharts verfügbar. Alle diese Implementierungen verwenden die Semantik der zu Grunde liegenden kommunizierenden endlichen Automaten. Die kommerziell verfügbare graphische Sprache SCADE [18] verbindet Elemente al-

ler drei Sprachen. Die SCADE Suite® wird für sicherheitskritische Komponenten eingesetzt, z.B. von Airbus.

Aufgrund der drei Simulationsphasen kann auch StateMate als synchrone Sprache angesehen werden. StateMate ist deterministisch, wenn Konflikte aufgelöst werden. Laut Halbwachs ist „StateMate fast eine synchrone Sprache, die einzige Eigenschaft, die StateMate fehlt, ist der augenblickliche *Broadcast*-Mechanismus" [199].

2.4.4 Nachrichtenaustausch am Beispiel von SDL

Spracheigenschaften

Wegen der Kommunikation über gemeinsamen Speicher und aufgrund des *Broadcast*-Mechanismus sind StateCharts für die Modellierung verteilter Systeme ungeeignet. Nachrichtenaustausch ist das für verteilte Systeme besser geeignete Kommunikationsparadigma. Wir stellen daher nun eine zweite Sprache vor, die auch auf kommunizierenden endlichen Automaten basiert, aber asynchronen Nachrichtenaustausch nutzt.

Wir benutzen die Sprache SDL (*Specification and Description Language*) als ein Beispiel. SDL wurde speziell für verteilte Anwendungen entwickelt. Die erste Entwicklung begann in den siebziger Jahren des letzten Jahrhunderts, die formale Semantik von SDL wurde in den späten achtziger Jahren definiert. Die Sprache ist von der ITU (*International Telecommunication Union*) standardisiert worden. Der erste Standard, „*Z.100 Recommendation*" wurde 1980 veröffentlicht und u.a. in den Jahren 1992, 1999, 2011 und 2016 aktualisiert [483]. Die Aktualisierung von 1999 ist als SDL-2000 bekannt.

Viele Anwender bevorzugen graphische Spezifikationssprachen, während andere textuelle Spezifikationen vorziehen. SDL bietet beide Möglichkeiten. Die Basiselemente von SDL sind sogenannte Prozesse. SDL-Prozesse stellen erweiterte endliche Automaten dar. Abb. 2.24 zeigt die Symbole, die in der graphischen Repräsentation von SDL verwendet werden.

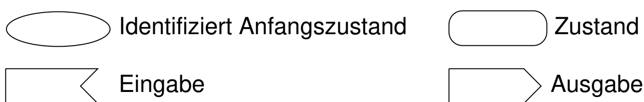


Abb. 2.24 Symbole in der graphischen Darstellung von SDL

Beispiel 2.15: Als Beispiel soll gezeigt werden, wie man das Zustandsdiagramm in Abb. 2.25 in SDL darstellen kann. Abb. 2.25 unterscheidet sich von Abb. 2.13 auf Seite 57 durch das Hinzufügen von Ausgabewerten, das Entfernen des Zustands Z und die veränderte Wirkung des Signals k.

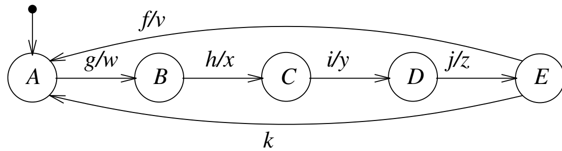


Abb. 2.25 Ein endlicher Automat

Abb. 2.26 zeigt die zugehörige graphische SDL-Darstellung. Offensichtlich ist die Darstellung äquivalent zum Zustandsdiagramm in Abb. 2.25.

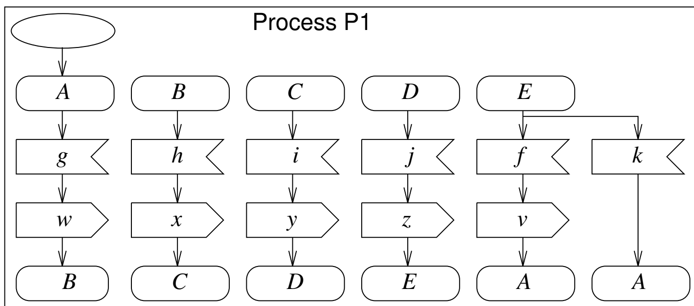


Abb. 2.26 SDL-Darstellung von Abb. 2.25 ∇

Als Erweiterung gegenüber klassischen endlichen Automaten können SDL-Prozesse Operationen auf Daten ausführen. Variablen können lokal für einen Prozess deklariert werden. Ihr Typ kann entweder vordefiniert werden oder er wird durch die SDL-Operation festgelegt. SDL unterstützt abstrakte Datentypen (ADTs). Die Syntax der Deklarationen und Operationen ist anderen Programmiersprachen sehr ähnlich. Abb. 2.27 zeigt Beispiele für Deklarationen, Zuweisungen und Entscheidungen in SDL.

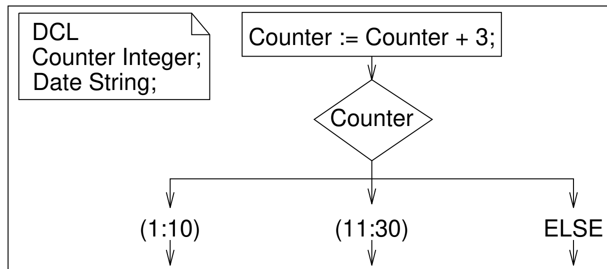


Abb. 2.27 Deklarationen, Zuweisungen und Entscheidungen in SDL

SDL enthält auch Programmiersprachenelemente wie Prozeduren. Prozeduraufrufe können auch graphisch dargestellt werden. Objektorientierte Elemente wurden 1992 in die Sprache integriert und in SDL-2000 erweitert.

Erweiterte endliche Automaten sind nur die Basiselemente von SDL-Beschreibungen. Im Allgemeinen besteht eine SDL-Beschreibung aus einer Menge von interagierenden SDL-Prozessen oder Automaten. Prozesse können Signale an andere Prozesse senden. Die Semantik der Interprozesskommunikation in SDL basiert auf *First-In First-Out-* (FIFO) **Warteschlangen**, die mit jedem Prozess verbunden sind. Pro Prozess gibt es dabei genau eine Warteschlange. Ein Signal, das an einen bestimmten Prozess geschickt wird, landet in dessen FIFO-Warteschlange (siehe Abb. 2.28).

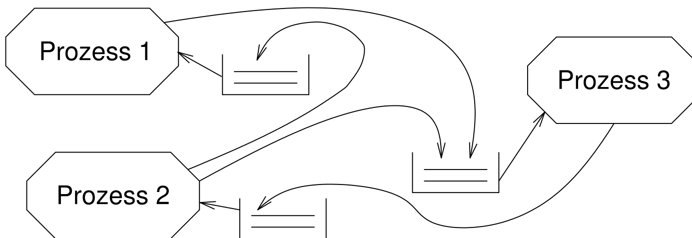


Abb. 2.28 SDL-Interprozesskommunikation

Jeder SDL-Prozess entnimmt den nächsten verfügbaren Eintrag aus der FIFO-Warteschlange und prüft, ob er einem der Eingabewerte für den aktuellen Prozess und Zustand entspricht. Wenn das so ist, wird die zugehörige Transition ausgeführt und eine entsprechende Ausgabe erzeugt. Ein Eintrag in der FIFO-Warteschlange wird ignoriert, wenn er keinem der Eingabewerte entspricht (es sei denn, der sogenannte SAVE-Mechanismus wird verwendet). Die FIFO-Warteschlangen haben im Modell eine unendliche Kapazität. Das bedeutet, dass in der Semantik von SDL-Beschreibungen keine FIFO-Überläufe betrachtet werden. In echten Systemen dagegen müssen die FIFO-Warteschlangen natürlich eine endliche Länge haben. Dies ist eines der Probleme von SDL: um korrekte Realisierungen von einer SDL-Spezifikation abzuleiten, muss man garantierte sichere obere Schranken für die Länge der FIFO-Warteschlangen bestimmen.

Prozess-Interaktionsdiagramme können verwendet werden, um zu visualisieren, welche Prozesse mit welchen anderen kommunizieren. Prozess-Interaktionsdiagramme beinhalten **Kanäle**, die zum Senden und Empfangen von Signalen verwendet werden. Bei SDL beschreibt der Begriff „Signal" Ein- und Ausgaben der modellierten Automaten.

Beispiel 2.16: Abb. 2.29 zeigt ein Prozess-Interaktionsdiagramm B1 mit den Kanälen Sw1 und Sw2 sowie den lokalen Signalen A und B. In Klammern stehen die Namen der Signale, die über den jeweiligen Kanal transportiert werden.

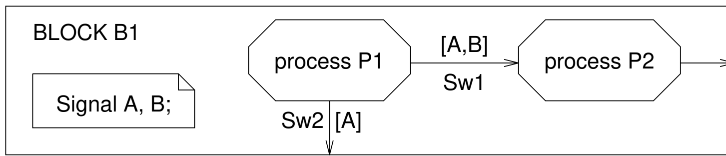


Abb. 2.29 Prozess-Interaktionsdigramm ∇

Es gibt drei Möglichkeiten, den Empfänger eines Signals anzugeben:

1. **Durch Prozess-Identifikatoren:** durch die Angabe eines empfangenden SDL-Prozesses im graphischen Ausgabesymbol (siehe Abb. 2.30 (links)).

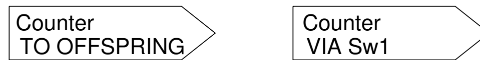


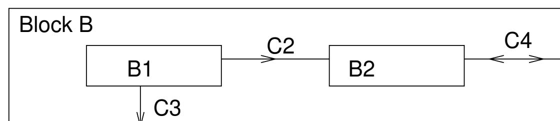
Abb. 2.30 **Links**: Empfänger über Prozess identifiziert; **rechts**: Empfänger über Kanal identifiziert

   Da Prozesse dynamisch generiert werden können, die Anzahl von Prozessen also nicht bereits zur Übersetzungszeit festgelegt werden muss, gibt der Wert OFFSPRING die Prozesse an, die von einem Prozess dynamisch generiert wurden.
2. **Explizit:** durch die Angabe eines Kanalnamens (siehe Abb. 2.30 (rechts)). Sw1 ist der Name eines Kanals.
3. **Implizit:** Wenn es eine direkte Zuordnung von Signalen zu Kanälen gibt, wird bei Angabe eines Signals der jeweilige Kanal verwendet. Beispiel: in Abb. 2.29 wird das Signal B implizit immer über Kanal Sw1 laufen.

Ein SDL-Prozess kann nicht innerhalb eines anderen SDL-Prozesses definiert werden, Prozesse können also nicht verschachtelt werden. Sie können aber hierarchisch in sogenannten **Blöcken** gruppiert werden. Blöcke auf der höchsten Hierarchiestufe heißen **Systeme**. Systeme besitzen keine Kanäle an ihrem Rand, sofern die Umgebung auch als Block modelliert wird. Blöcke auf der untersten Hierarchiestufe heißen **Prozess-Interaktionsdiagramme**. **Prozess-Interaktionsdiagramme** befinden sich eine Ebene oberhalb der Blätter einer hierarchischen Beschreibung.

Beispiel 2.17: Block B1 kann in dazwischenliegenden Blöcken (etwa in B in Abb. 2.31) verwendet werden.

Abb. 2.31 SDL-Block



Auf der obersten Ebene der Hierarchie haben wir das System (siehe Abb. 2.32).

Abb. 2.32 SDL-System

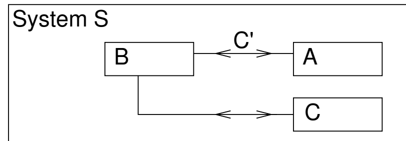
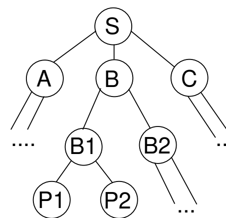


Abb. 2.33 zeigt die mit Blockdiagrammen der Abbildungen 2.29, 2.31 und 2.32 modellierte Hierarchie.

Abb. 2.33 SDL-Hierarchie



Das Beispiel zeigt, dass Prozess-Interaktionsdiagramme sich eine Stufe oberhalb der **Blätter** der hierarchischen Beschreibung befinden. Systeme stellen die Wurzel der Hierarchie dar. ∇

Einige Einschränkungen bei der Modellierung von Hierarchie wurden in SDL-2000 beseitigt. In SDL-2000 wurde die Ausdruckskraft von Blöcken und Prozessen angeglichen und durch ein allgemeines **Agentenkonzept** ersetzt.

Zur Modellierung von Zeit enthält SDL **Zeitgeber** (engl. ***Timer***). Zeitgeber können lokal für Prozesse deklariert werden. Sie können mit Hilfe von SET-Anweisungen gesetzt und zurückgesetzt werden. Diese Anweisungen besitzen zwei Parameter; eine absolute Zeit und einen Zeitgebernamen. Die absolute Zeit definiert, wann die Zeit abläuft. Die eingebaute Funktion now kann benutzt werden, um anzugeben, wann die SET-Anweisung auszuführen ist. Ein Signal wird in der Eingabeschlange gespeichert, wenn ein Zeitgeber abgelaufen ist. Der Name dieses Signals wird über den zweiten Parameter des Aufrufs von SET bereit gestellt. Das Signal wird typischerweise einen Zustandsübergang im Automaten auslösen. Allerdings kann dieser Übergang durch andere Einträge in der Eingabe-Warteschlange, die zuerst verarbeitet werden müssen, verzögert werden. Daher ist dieses Zeitgeber-Konzept nicht für harte, sondern für weiche Zeitschranken geeignet, wie sie in der Telekommunikation vorkommen. Eine zweite eingebaute Funktion expirytime kann benutzt werden, um einige der Einschränkungen von now zu überwinden.

Zeitgeber können mit der Funktion RESET zurückgesetzt werden. Damit wird das Zählen gestoppt. Das Signal wird aus der Eingabe-Warteschlange entfernt, wenn es

sich bereits dort befinden sollte. Zu Beginn der Ausführung von SET wird implizit ein RESET ausgeführt.

Beispiel 2.18: Abb. 2.34 zeigt die Verwendung eines Zeitgebers Z. Das Diagramm entspricht dem in Abb. 2.26, mit dem Unterschied, dass der Zeitgeber Z beim Übergang von Zustand D in Zustand E auf den Wert der aktuellen Zeit now + T gesetzt wird. Für die Transition von E nach A ist damit ein *Timeout* von T Zeiteinheiten

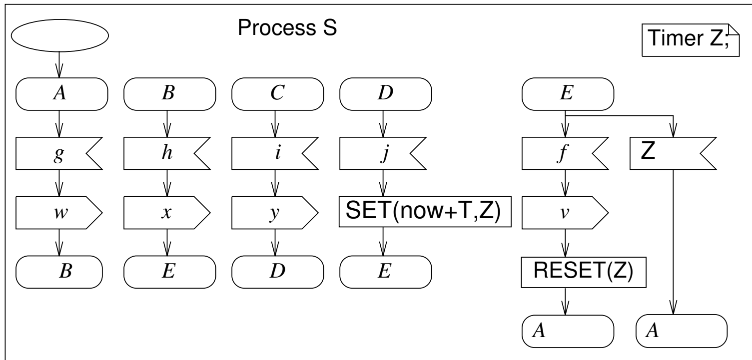
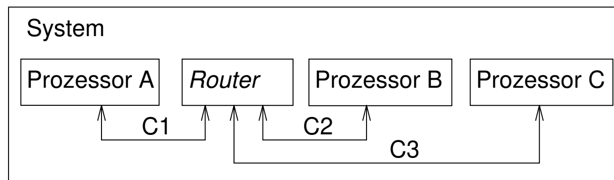


Abb. 2.34 Benutzung des Zeitgebers Z

definiert. Wenn diese Zeit verstrichen ist, bevor das Signal f eintrifft, findet ein Übergang in den Zustand A statt, bei dem kein Ausgabesignal v erzeugt wird. Eine streng periodische Verarbeitung mit einer Periode T ist auf diese Weise schwer zu erreichen, weil die anderen Einträge in der Eingabewarteschlange die Ausführung verzögern können. ∇

Beispiel 2.19: SDL kann verwendet werden, um Protokollstapel in Rechnernetzwerken zu beschreiben. Abb. 2.35 zeigt drei Prozessoren, die durch einen *Router* verbunden werden. Die Kommunikation zwischen Prozessoren und dem *Router* basiert

Abb. 2.35 Kleines Rechnernetz, beschrieben in SDL



auf FIFO-Warteschlangen. Damit bietet SDL einen eingebauten Kommunikationsmechanismus, der realen Netzwerkprotokollen entspricht.

Sowohl die Prozessoren als auch der Router implementieren Protokolle mit verschiedenen Ebenen (siehe Abb. 2.36). Jede Ebene beschreibt die Kommunikation mit einer abstrakteren Schicht. Das Verhalten jeder Ebene wird meistens als end-

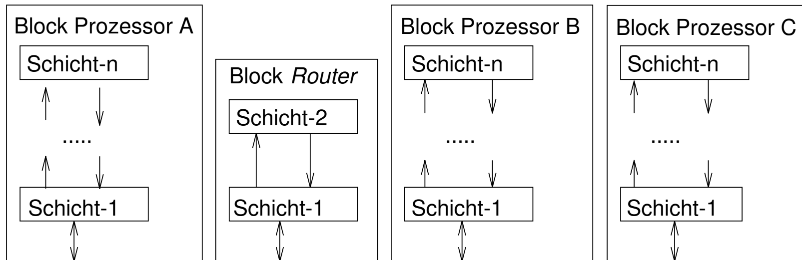


Abb. 2.36 Protokoll-Stapel in SDL

licher Automat modelliert. Die genaue Beschreibung dieser Automaten hängt vom verwendeten Netzwerkprotokoll ab und kann sehr komplex werden. Typischerweise wird das Überprüfen und Behandeln von Fehlerbedingungen sowie das Sortieren und Weiterleiten von Informationspaketen realisiert. ∇

Verfügbare Werkzeuge für SDL enthalten Schnittstellen zu UML (siehe Seite 133) und SCs (siehe Seite 47). Eine umfassende Liste verfügbarer Werkzeuge wird im SDL-Forum zur Verfügung gestellt [482]. Estelle [74] ist eine weitere Sprache, die zur Beschreibung von Kommunikationsprotokollen entworfen wurde. Ähnlich zu SDL stellt Estelle Kommunikation über Kanäle und FIFO-Puffer zur Verfügung. Versuche, Estelle und SDL zu vereinen, sind fehlgeschlagen.

Bewertung von SDL

SDL ist hervorragend zur Modellierung von verteilten Anwendungen geeignet und die Sprache ist weiterhin als Referenz für asynchronen Nachrichtenaustausch nützlich. SDL ist nicht in allen Situationen deterministisch, da die Reihenfolge, in der gleichzeitig ankommende Signale in die FIFO-Warteschlangen eingereiht werden, nicht spezifiziert ist. Alle möglichen Reihenfolgen sind damit möglich. Verlässliche Implementierungen erfordern das manchmal schwierige Bestimmen einer oberen Schranke für die Länge der FIFO-Warteschlangen. Hierzu gibt es umfangreiche Literatur. Die Modellierung von Zeitbedingungen ist ausreichend für weiche Zeitschranken, aber nicht für harte Echtzeitbedingungen. Hierarchien werden nicht so wie in StateCharts unterstützt. Es gibt keine Beschreibung von nicht-funktionalen Eigenschaften und keine vollständige Unterstützung für Programmiersprachenkonstrukte, obwohl dies von den aktuellen Standards angestrebt wird. SDL wurde zum Beispiel für die Spezifikation von ISDN verwendet. Aktuell scheint es, dass das Interesse an SDL zu einem generellen Interesse an Systembeschreibungssprachen verallgemeinert wird, wie dies auch im SDL-Forum zum Ausdruck kommt [483].

2.5 Datenfluss

2.5.1 Überblick

Viele echte Anwendungen lassen sich sehr „natürlich” durch Datenflüsse beschreiben. Datenflussmodelle beschreiben den Weg, auf dem Daten von Komponente zu Komponente fließen [146]. Jede Komponente transformiert die Daten dabei auf eine bestimmte Art. Eine mögliche Definition von Datenfluss ist die folgende:

Definition 2.14 ([582]): Datenflussmodellierung „ist ein Vorgang, bei dem identifiziert, modelliert und dokumentiert wird, wie sich Daten in einem Informationssystem bewegen. Die Datenflussmodellierung betrachtet Prozesse (Aktivitäten, die Daten von einer Form in eine andere transformieren), Datenspeicher (Bereiche, in denen Daten aufbewahrt werden), externe Einheiten (die Daten an ein System senden oder Daten von diesem empfangen) und Datenflüsse (Wege, auf denen Daten fließen können)”.

Ein **Datenflussprogramm** wird als gerichteter Graph angegeben, bei dem die Knoten, auch **Aktoren** genannt, Berechnungen und die Kanten Kommunikationskanäle darstellen. Jeder Aktor führt funktionale Berechnungen aus, also Berechnungen, die alleine auf den Eingabewerten durchgeführt werden. Jeder Prozess in einem Datenflussgraphen ist in eine Folge von atomaren „Ausführungen” (engl. *firings*) aufgeteilt. Jede Aktivierung erzeugt und verbraucht Marken (engl. *tokens*). Von-Neumann-Programme geben eine totale Ordnung für die Ausführung von Befehlen vor. Datenflussprogramme vermeiden eine unnötige Vorgabe einer solchen totalen Ordnung.

Beispiel 2.20: Abb. 2.37 zeigt als Beispiel den Datenfluss in einem *Video-on-demand*-System [299]. Kunden betreten das System über die Netzwerkschnittstelle. Ihr Zugriffswunsch wird der Warteschlange der Kunden hinzugefügt.

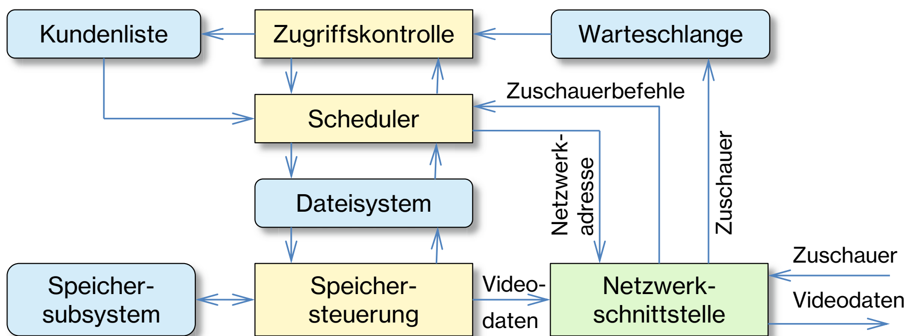


Abb. 2.37 *Video on demand*-System (blau: Speicher, gelb: Verarbeitung, grün: E/A)

Nach dem Passieren der Zugriffskontrolle werden die Videowünsche für den Zugriff auf das Dateisystem eingeplant. In Kooperation mit der Speichersteuerung stellt das Dateisystem die Videos dem Kunden zur Verfügung. ∇

Für allgemeine Datenflussmodelle lassen sich geforderte Eigenschaften eines Systems nur schwer beweisen. Daher kommen meist eingeschränkte Modelle zum Einsatz.

Ein spezieller Typ von Datenfluss wird benutzt, um in Rechnerarchitekturen **dynamisches *Scheduling*** von Befehlen zu realisieren. Dies bedeutet, dass die Reihenfolge der Ausführung von Maschinenbefehlen nicht unbedingt ihrer Anordnung im Speicher entspricht, sondern dass diese Reihenfolge unter Beachtung von Datenabhängigkeiten geändert werden kann. Man spricht deshalb auch von *out-of-order scheduling*. Es gibt zwei sehr bekannte Algorithmen für diese Form des dynamischen *Schedulings*: *scoreboarding* und den Tomasulo-Algorithmus [544]. Beide Algorithmen werden in Büchern zur Rechnerarchitektur im Detail vorgestellt (siehe z.B. Hennessy et al. [212]). Aus diesem Grund werden sie in diesem Buch nicht behandelt. Es gibt allerdings Varianten dieser Algorithmen, die auf Task-Ebene angewandt werden (siehe z.B. Wang et al. [561]).

2.5.2 *Kahn-Prozessnetzwerke*

Kahn-Prozessnetzwerke (KPN) [279] sind ein Spezialfall solcher Datenflussmodelle. KPNs bestehen aus Knoten und Kanten. Knoten entsprechen von einer Task bzw. einem Prozess ausgeführten Berechnungen. Wie alle Datenflussgraphen stellen auch KPN-Graphen nur die durchzuführenden Berechnungen und deren Abhängigkeiten voneinander dar, nicht aber die Reihenfolge, in der die Berechnungen durchgeführt werden müssen (im Gegensatz zu Spezifikationen in von-Neumann-Sprachen wie C). Die Kanten stellen Kommunikationskanäle mit potenziell unendlich großen FIFOs dar. Auch wenn die Berechnungs- und Kommunikationszeiten variieren können, ist doch sichergestellt, dass Kommunikation innerhalb endlicher Zeit stattfindet. Schreibvorgänge in KPNs sind nicht-blockierend, da angenommen wird, dass die FIFOs eine entsprechende Größe aufweisen. Leseoperationen müssen einen bestimmten Kanal angeben, von dem gelesen werden soll. Dabei kann ein Knoten vor dem Leseversuch nicht überprüfen, ob Daten zur Verfügung stehen. Auch kann ein KPN-Prozess nicht auf Daten von mehr als einem Port gleichzeitig warten. Leseoperationen werden blockiert, wenn ein KPN-Prozess versucht, aus einer leeren FIFO-Warteschlange zu lesen. Nur ein einziger KPN-Prozess darf aus einer bestimmten Warteschlange lesen, ebenso darf nur ein einziger Prozess in eine bestimmte Warteschlange schreiben. Wenn ein Prozess seine Ausgabedaten also an mehrere Nachfolger senden will, müssen die Daten innerhalb des Prozesses dupliziert werden. Es gibt keine anderen Methoden für die Kommunikation zwischen KPN-Prozessen.

Im folgenden Beispiel inkrementieren bzw. dekrementieren die Prozesse τ1 und τ2 den jeweils vom Kommunikationspartner erhaltenen Wert:

```
process τ1(in int u, out int v){
  int i;
  i = 0;
  for (;;) {
   send(i,v);                                        /* sende i über Kanal v */
   i = wait(u);                                        /* lese i von Kanal u */
   i = i-1;
  }
}
process τ2(in int v, out int u){
  int i;
  for (;;) {
   i = wait(v);
   i = i+1;
   send(i,u);
  }
}
```

Eine graphische Darstellung dieses KPNs ist in Abb. 2.38 zu sehen.

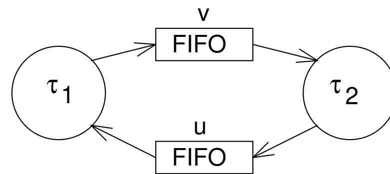


Abb. 2.38 Graphische Darstellung eines KPNs

In diesem Beispiel werden die FIFOs nicht wirklich benötigt, da es hier nicht vorkommen kann, dass sich Nachrichten in den Kanälen anstauen. Dieses und weitere Beispiele lassen sich mit der Software levi simulieren [496].

Die Beschränkungen für Lese- und Schreibvorgänge ergeben eine **sehr schöne Eigenschaft von KPNs**: die Reihenfolge, in der ein Knoten Daten empfängt, wird fest durch die Folge von Leseoperationen vorgegeben und hängt damit nicht von der Reihenfolge ab, in der die Daten über die Kanäle übertragen werden. Damit ist die Abfolge von Operationen nicht von der Geschwindigkeit der Knoten abhängig, welche die Daten produzieren. **Für eine gegebene Menge an Eingabedaten erzeugen KPNs stets dasselbe Ergebnis, unabhängig von der Geschwindigkeit der Knoten.** Diese Eigenschaft ist z.B. für Simulationen wichtig. Das Ergebnis der Simulation ist unabhängig davon, wie schnell das KPN simuliert wird. Auch der Einsatz von Hardwarebeschleunigern in einigen Knoten und einer verteilten Ausführung führen nicht zu unterschiedlichen Ergebnissen im Vergleich zu einem zentralen Simulationsansatz. Diese Eigenschaft wird „deterministisch"[14] genannt. Die von SDL

[14] Wieder im Sinne von „*determinate*".

bekannten FIFO-Konflikte existieren hier nicht. Diese schöne Eigenschaft hat zur Folge, dass KPNs häufig als interne Repräsentation innerhalb eines Entwurfsflusses verwendet werden.

KPNs können durch einen *merge*-Operator erweitert werden (ähnlich der **select**-Anweisung in Ada, siehe Seite 124). Dieser erlaubt es, Leseaufträge für mehrere Kanäle in eine Warteschlange einzureihen und darauf zu warten, dass einer der Kanäle Daten erzeugt. Ein solcher Operator führt ein nichtdeterministisches Verhalten ein: wenn von mehreren Kanten gleichzeitig Daten eintreffen, dann ist die Reihenfolge der Abarbeitung dieser Daten nicht mehr festgelegt. Diese Erweiterung ist in der Praxis nützlich, sie macht aber die schönste Eigenschaft der KPNs zunichte.

Im Allgemeinen benötigen KPNs ein *Scheduling* zur Laufzeit, da sich ihr genaues Verhalten nur schwer voraussagen lässt. Die Ursache dafür ist, dass wir keinerlei Annahmen über die Geschwindigkeiten von Kanälen und Knoten machen. Da aber in frühen Entwurfsphasen keinerlei Ausführungszeiten bekannt sind, ist dieses Modell für diese Phasen sehr gut geeignet.

KPNs sind Turing-vollständig. Das bedeutet: alles, was mit einer Turing-Maschine (dem Standard-Modell der Berechenbarkeit) berechnet werden kann, kann auch mit einem KPN berechnet werden. Der Beweis basiert darauf, dass KPNs eine Obermenge der *Boolean Dataflow* (BDF)-Netzwerke sind und nach Buck [73] können BDF-Netzwerke Turing-Maschinen simulieren. Eine Einschränkung der Anwendbarkeit von KPNs ergibt sich allerdings aus der Tatsache, dass die Anzahl der Prozesse in KPNs fest ist, sich also nicht zur Laufzeit ändert.

Im allgemeinen Fall ist es unentscheidbar, ob FIFOs endlicher Länge für ein gegebenes KPN-Modell ausreichen. Eine Reihe praktisch anwendbarer *Scheduling*-Algorithmen für KPNs werden in [294] beschrieben. Auch gibt es für einige spezielle Fälle Beweise der Beschränkheit der Größe von FIFOs [99]. Beispielsweise können Schranken für den Spezialfall *Polyhedral Process Networks* (PPNs) bestimmt werden. Bei PPNs müssen die Grenzen aller Schleifen zur Compilezeit bekannt sein. Derin [125] nutzt Wissen über den Programmcode der Knoten für eine dynamische Verlagerung von Prozessen zwischen Prozessoren.

2.5.3 SDF

Wenn wir Beschränkungen für das *Timing* von Knoten und Kanälen zulassen, wird das *Scheduling* deutlich einfacher und außerdem lassen sich benötigte Puffergrößen bestimmen. Dies ist beim SDF-Modell der Fall [334]. SDF bedeutete ursprünglich ***Synchronous** Data Flow* bzw. synchroner Datenfluss, wird aber heute teilweise als ***Static** Data Flow* gedeutet.

Das SDF-Modell [334] lässt sich am besten anhand seiner graphischen Darstellung erklären. Die Darstellung basiert auf einem gerichteten Graphen bestehend aus Knoten und gerichteten Kanten. Die Knoten werden auch als Aktoren (engl. *actors*) bezeichnet. Die Kanten können Marken speichern, wobei die Speicherkapazität standardmäßig unbegrenzt ist. Im Allgemeinen werden manche der Kanten anfangs

Marken enthalten. Zu jeder Kante gehört eine **Anzahl von konsumierten Marken** (Zahl am Pfeilende der graphischen Darstellung) und eine **Anzahl von erzeugten Marken** (Zahl am anderen Ende der Kanten). Die Ausführung eines SDF-Modells geschieht taktgesteuert auf der Basis eines impliziten Taktgebers. Ein Aktor ist ausführungsbereit, wenn für jede eingehende Kante die Zahl der vorhandenen Marken mindestens gleich der Zahl der zu konsumierenden Marken ist.

Beispiel 2.21: Abb. 2.39 (links) zeigt einen SDF-Graphen. A und B stellen die Berechnungen dar. Für Aktor B gibt es eine ausreichende Anzahl von Marken auf der eingehenden Kante, sodass dieser ausführungsbereit ist. Aktor A ist nicht ausführungsbereit. Bei jedem Takt können die ausführungsbereiten Aktoren ausgeführt

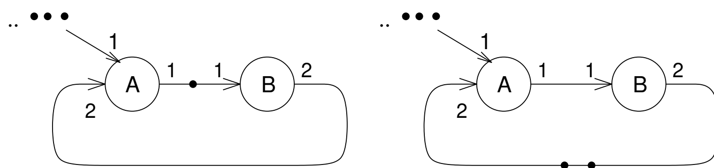


Abb. 2.39 SDF-Graph: **links**: Anfangssituation; **rechts**: nach Ausführung von B

werden, sie müssen es aber nicht. Eine Ausführung bewirkt, dass für die Eingangskanten die Anzahl der Marken um die zu konsumierenden Marken reduziert werden. Ebenso bewirkt eine Ausführung, dass für die Ausgangskanten die Anzahl der aktuellen Marken um die Anzahl zu erzeugenden Marken erhöht werden. Im Beispiel ist die Veränderung der Anzahl der Marken in der Abb. 2.39 (rechts) zu sehen. ∇

In der Praxis stellen Marken Daten dar, Aktoren repräsentieren Berechnungen und Kanten benötigen im Allgemeinen FIFO-Puffer. Diese Puffer implizieren, dass SDF **asynchronen Nachrichtenaustausch** benutzt. Anstelle der standardmäßig unbegrenzten Pufferkapazitäten können wir begrenzte Pufferkapazitäten mit Hilfe von Rückwärtskanten modellieren. Die anfängliche Zahl der Marken auf den Rückwärtskanten entspricht dabei der Kapazität der FIFO-Puffer. Dies wird in Abb. 2.40 gezeigt. Die zwei gezeigten Modelle sind dabei äquivalent.

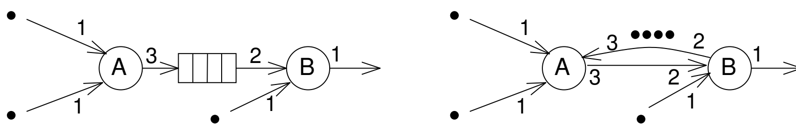


Abb. 2.40 Ersetzen von expliziten FIFO-Puffern durch Rückwärtskanten

Beispielsweise wird die erste Ausführung von A drei Marken von der Rückwärtskante konsumieren, sodass nur noch eine Marke auf dieser Kante verbleibt, was dem einen leeren FIFO-Platz nach der ersten Ausführung von A auf der linken Seite entspricht.

Aufgrund der statischen Anzahl der erzeugten und der konsumierten Marken auf jeder Kante ist es möglich, eine Ausführungsreihenfolge und den Speicherbedarf zur Entwurfszeit zu bestimmen. Damit kann eine komplexe Bestimmung einer Ausführungsreihenfolge zur Laufzeit (ein dynamisches *Scheduling*) vermieden werden. Es ist möglich, aus SDF-Graphen periodische Ausführungsreihenfolgen zu erzeugen.

Beispiel 2.22: Wir betrachten Ausführungsreihenfolgen (engl. *schedules*) von SDF-Graphen anhand des Modells in Abb. 2.41. Angenommen, es gäbe anfänglich sechs

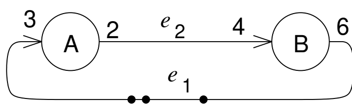


Abb. 2.41 SDF-Schleife

Marken auf der Kante e_1. Tabelle 2.2 (links) zeigt die resultierende Ausführungsreihenfolge. Aufgrund der begrenzten Anzahl anfänglich vorhandener Marken ist nur eine sequentielle Ausführung möglich. Nunmehr nehmen wir an, es gäbe neun

Tabelle 2.2 *Schedule* für SDF-Schleife: **links**: 6 initiale Marken, **rechts**: 9 initiale Marken

Takt	Marken auf Kanten		Nächste Aktion
	e_1	e_2	A oder B
0	6	0	A
1	3	2	A
2	0	4	B
3	6	0	A
4	3	2	A

Takt	Marken auf Kanten		Nächste Aktion
	e_1	e_2	A oder B oder (A und B)
0	9	0	A
1	6	2	A
2	3	4	A und B
3	6	2	A
4	3	4	A und B

initiale Marken auf der Kante e_1. Unter der Annahme, dass **alle Aktoren so früh wie möglich ausgeführt werden**, ergibt sich die Ausführung gemäß Tabelle 2.2 (rechts). Unter dieser Annahme werden A und B gleichzeitig ausgeführt. ∇

Bei der Erzeugung von Ausführungsreihenfolgen könnten wir auch Zielfunktionen und Beschränkungen berücksichtigen, wie beispielsweise eine beschränkte Anzahl von Prozessoren [57].

In obigem Beispiel führten die Kantengewichte 2, 3, 4 und 6 zu unterschiedlichen Ausführungsraten der Aktoren A und B. Im Allgemeinen erlauben Kantengewichte die Modellierung von Mehrraten-Signalverarbeitungsanwendungen, d.h. Anwendungen, bei denen bestimmte Signale mit Raten erzeugt werden, die ein Vielfaches der Raten anderer Signale sind. Beispielsweise könnten in einem Fernsehgerät manche

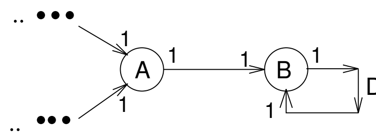
Berechnungen mit einer Rate von 100 Hz erzeugt werden, wohingegen andere Signale mit einer Rate von 50 Hz erzeugt werden. Abgesehen von der Initialisierungsphase und über längere Perioden betrachtet, muss dabei in einem SDF-Graphen die Anzahl der Marken, die an eine Kante gesendet werden gleich der Anzahl der Marken sein, die von dieser Kante wieder abgezogen werden. Ansonsten würden sich Marken in den FIFO-Puffern ansammeln und keine endliche FIFO-Kapazität würde jemals ausreichen. Sei n_s die Anzahl von Marken, die durch die Ausführung eines sendenden Aktors s erzeugt werden und sei f_s die Ausführungsrate. Sei n_r die Anzahl von Marken, die durch die Ausführung eines empfangenden Aktors r konsumiert werden und sei f_r die Ausführungsrate. Dann muss gelten

$$n_s * f_s = n_r * f_r \tag{2.13}$$

In Tabelle 2.2 ist diese Bedingung für den stationären Zustand erfüllt.

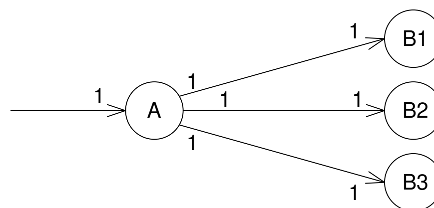
SDF-Graphen können auch Verzögerungen enthalten, was durch das das Symbol D auf einer Kante dargestellt wird (siehe Abb. 2.42).

Abb. 2.42 Verzögerung in SDF



Das *Observer*-Muster, für das wir bei der Modellierung mit von-Neumann-Sprachen ein Problem hatten (siehe Seite 37), kann in SDF relativ leicht korrekt modelliert werden (siehe Abb. 2.43). Es gibt kein Risiko von *Deadlocks*. Allerdings erlaubt es SDF auch nicht, zur Laufzeit neue Beobachter hinzuzufügen.

Abb. 2.43 *Observer*-Muster in SDF



Der Buchstabe „S" in SDF stand ursprünglich für das Adjektiv *synchronous*, da ausführbereite Aktoren synchron ausgeführt werden können. Allerdings zeigen die Ausführungsreihenfolgen in Tabelle 2.2, dass das gleichzeitige Ausführen aller Aktoren tatsächlich eher selten sein kann. Deswegen wird das „S" inzwischen auch so interpretiert, dass es für das Adjektiv *static* steht, denn eine mögliche statische Planung der Ausführungsreihenfolge ist in der Tat ein wesentliches Merkmal von SDF.

SDF-Modelle sind deterministisch (siehe Haubelt et al. [207], Abschnitt 2.2), eignen sich aber nicht für die Modellierung von Kontrollflüssen wie z.B. Sprüngen. Einige Erweiterungen von SDF-Modellen wurden vorgeschlagen (siehe z.B. Stuijk [515]):

- Eine Ergänzung sind **Modi**, die den Zuständen eines zugehörigen endlichen Automaten entsprechen. Für jeden Modus könnte jeweils ein anderer SDF-Graph zutreffend sein. Ein Übergang zwischen diesen Modi könnte dann durch bestimmte Ereignisse erfolgen.
- **Homogene synchrone Datenflussgraphen** (engl. *Homogeneous Synchronous Data Flow* (HSDF)) sind ein Spezialfall von SDF-Graphen. Bei diesen beträgt die Anzahl der pro Ausführung erzeugten und verbrauchten Marken jeweils 1.
- Bei **zyklo-statischem Datenfluss** (engl. *Cyclo-Static Data Flow* (CSDF)) kann die Anzahl der pro Ausführung erzeugten und verbrauchten Marken zeitlich variieren, aber es muss insgesamt ein periodisches Verhalten vorliegen.

2.5.4 Simulink

Graphenstrukturen für Berechnungen werden auch häufig in der Regelungstechnik eingesetzt. In diesem Bereich ist die Simulink®-Komponente von MATLAB® weit verbreitet. MATLAB/Simulink [529] ist ein Modellierungs- und Simulationswerkzeug, das auf mathematischen Prinzipien beruht. Abb. 2.44 zeigt ein Beispiel für ein Simulink-Modell [366].

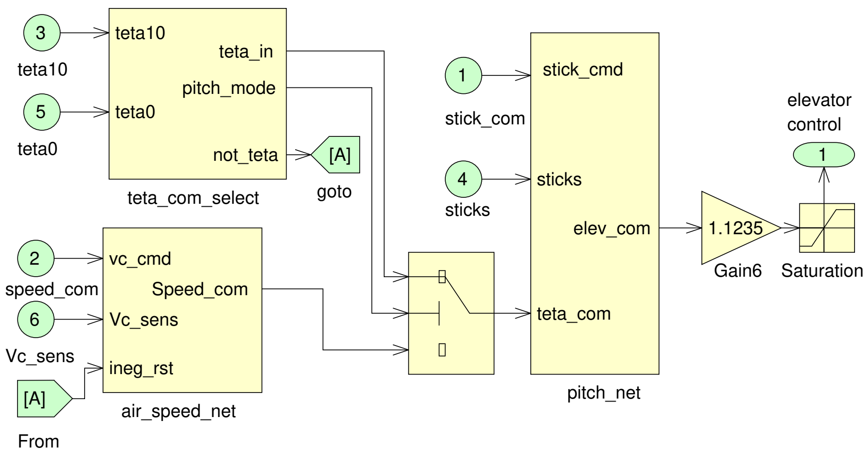


Abb. 2.44 Simulink®-Modell

Der Verstärker Gain6 und die Sättigungskomponente Saturation zeigen, dass hier analoge Modellierungskomponenten einbezogen werden können. Die „Schaltung" könnte hier auch Symbole enthalten, die für analoge Komponenten wie Integratoren oder Differentiatoren stehen. Der in der Mitte der Abbildung zu sehende Schalter zeigt, dass sich mit Simulink auch Kontrollflüsse modellieren lassen.

Diese graphische Darstellung ist intuitiv und ermöglicht es Ingenieuren der Regelungstechnik, einen Schwerpunkt auf die Regelungsfunktion zu legen, ohne den für die Implementierung dieser Funktion benötigten Code zu berücksichtigen. Die graphischen Symbole verdeutlichen dabei, dass analoge Schaltungen traditionell als Komponenten von Regelkreisentwürfen vorkommen. Ein Hauptziel ist es, aus solchen Entwürfen Software zu erzeugen. Dieser Ansatz wird meistens mit dem Begriff **modellgetriebener Entwurf** verbunden.

Die Semantik von Simulink-Modellen entspricht der Simulation auf einem digitalen Computer. Damit ist das Verhalten ähnlich zu dem analoger Schaltungen, aber möglicherweise nicht identisch dazu. Was ist also im Endeffekt die Semantik eines Simulink-Modells? Marian und Ma [366] beschreiben die Semantik wie folgt: „Simulink verwendet ein idealisiertes Zeitmodell für die Ausführung von Blöcken (Knoten) und die Kommunikation. Beide werden zu fest vorgegebenen Zeitpunkten in der Simulation mit unendlicher Geschwindigkeit ausgeführt. Danach wird die Simulationszeit um genau bestimmte Zeiteinheiten erhöht. Zwischen den einzelnen Zeitschritten sind alle Werte auf den Kanten konstant". Das bedeutet, dass das Modell schrittweise ausgeführt wird. In jedem Schritt wird die Funktion der Knoten (in Nullzeit) berechnet und die neuen Ergebnisse an den mit diesen verbundenen Eingängen zur Verfügung gestellt. Diese Erklärung legt keinen Abstand zwischen den Zeitschritten fest. Auch ist nicht direkt offensichtlich, wie sich ein solches System in Software implementieren lässt, da es vorkommen kann, dass auch nur sich langsam verändernde Ausgangswerte regelmäßig neu berechnet werden müssen.

Dieser Ansatz ist gut geeignet, um physische Systeme, z.B. Verkehrsmittel, zu modellieren und dann deren Verhalten zu simulieren. Auch digitale Signalverarbeitungssysteme können bequem in MATLAB modelliert werden. Um eine reale Implementierung eines MATLAB/Simulink-Modells zu erhalten, muss es erst in eine Sprache übersetzt werden, die von Hardware- oder Softwareentwurfssystemen unterstützt wird, also beispielsweise in C oder VHDL.

Die Komponenten von Simulink-Modellen stellen einen Spezialfall von **Aktoren** dar. Wir können dabei annehmen, dass Aktoren auf Eingaben warten und ihre Operationen erst durchführen, wenn alle benötigten Eingaben vorhanden sind. SDF ist ein weiteres Beispiel für aktorbasierte Sprachen. In **aktorbasierten Sprachen** ist es, im Gegensatz zu von-Neumann-Sprachen, nicht erforderlich, den Aktoren explizit einen Kontrollfluss zuzuweisen.

2.6 Petrinetze

2.6.1 Einführung

Eine sehr umfassende Beschreibung von Kontrollflüssen ist mit den als Petrinetzen bezeichneten Berechnungsgraphen möglich. Genau genommen modellieren Petrinetze **ausschließlich** Kontrollmechanismen und Kontrollabhängigkeiten. Die gleichzeitige Modellierung von Daten ist nur durch Erweiterungen von Petrinetzen realisierbar. Der Schwerpunkt von Petrinetzen liegt also auf der Modellierung kausaler Abhängigkeiten.

Im Jahr 1962 veröffentlichte Carl Adam Petri seine Methoden zur Darstellung kausaler Abhängigkeiten [450], die unter dem Namen **Petrinetze** bekannt wurden. Petrinetze gehen nicht von einer globalen Synchronisation aus und sind daher besonders gut zur Modellierung verteilter Systeme geeignet.

Bedingungen, **Ereignisse** und eine **Flussrelation** sind die Kernelemente von Petrinetzen. Bedingungen sind entweder erfüllt oder nicht erfüllt, Ereignisse können eintreten. Die Flussrelation beschreibt die Bedingungen, die erfüllt sein müssen, damit Ereignisse eintreten können. Außerdem beschreibt sie, welche Bedingungen wahr werden, wenn ein Ereignis eintritt. Graphische Darstellungen von Petrinetzen verwenden typischerweise Kreise, um Bedingungen darzustellen und Rechtecke, um Ereignisse darzustellen. Pfeile stellen die Flussrelation dar.

Beispiel 2.23: Abb. 2.45 zeigt eine Modellierung von Zugverkehr als Petrinetz. Es beschreibt den gegenseitigen Ausschluss von Zügen auf einem eingleisigen Streckenabschnitt, der in beide Richtungen befahren werden muss. Ein *Token* oder eine Marke

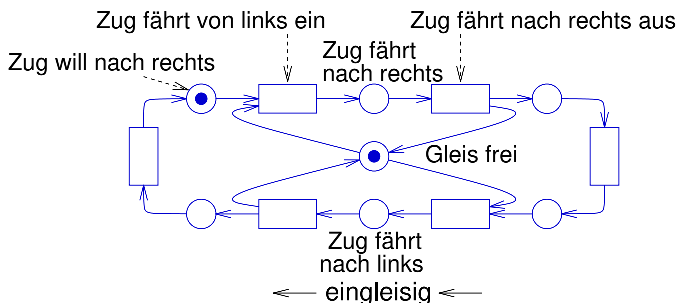


Abb. 2.45 Eingleisiger Abschnitt einer Bahnstrecke

wird verwendet, um die Kollision von Zügen zu vermeiden, die in unterschiedliche Richtungen fahren. Im Beispielnetz wird das *Token* durch eine erfüllte Bedingung in der Mitte des Modells dargestellt. Der kleine, im großen Kreis enthaltene ausgefüllte Kreis symbolisiert die Situation, in der die Bedingung erfüllt ist (im Beispiel: der Abschnitt ist frei). Dementsprechend kennzeichnet eine weitere Marke die Tatsache,

dass ein Zug nach rechts fahren möchte. Damit sind die beiden Bedingungen zum Schalten des Ereignisses „Zug fährt nach rechts“ erfüllt. Wir nennen diese Bedingungen **Vorbedingungen**. Wenn die Vorbedingungen eines Ereignisses erfüllt sind, kann das Ereignis eintreten.

Nach dem Eintreten des Ereignisses ist die Marke nicht mehr verfügbar und es wartet kein Zug mehr auf den eingleisigen Abschnitt. Folglich sind die Vorbedingungen nicht mehr erfüllt und die ausgefüllten Kreise verschwinden (siehe Abb. 2.46). Andererseits fährt jetzt ein Zug auf dem Abschnitt von links nach rechts. Somit ist

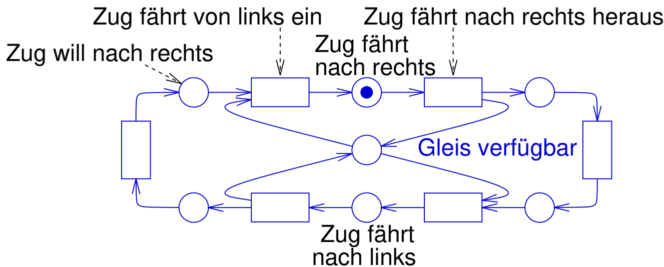


Abb. 2.46 Verwendung der Ressource „Gleisabschnitt“

die zugehörige Bedingung erfüllt. Eine Bedingung, die nach dem Eintreten eines Ereignisses erfüllt ist, heißt **Nachbedingung**. Im Allgemeinen kann ein Ereignis nur dann eintreten, wenn alle seine Vorbedingungen erfüllt sind. Wenn das Ereignis eingetreten ist, sind die Vorbedingungen nicht mehr erfüllt, stattdessen werden die Nachbedingungen erfüllt. Die Vor- und Nachbedingungen eines Ereignisses werden durch Pfeile dargestellt.

Ein Zug, der den eingleisigen Streckenabschnitt verlässt, stellt die Marke in der Mitte des Modells wieder her (siehe Abb. 2.47).

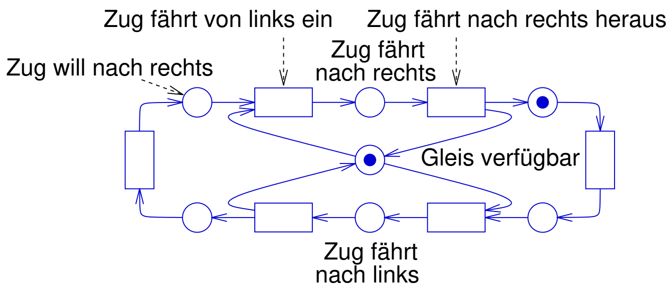


Abb. 2.47 Freigabe der Ressource „Gleisabschnitt“

Wenn zwei Züge gleichzeitig den eingleisigen Abschnitt benutzen wollen (siehe Abb. 2.48), kann nur einer der beiden in den Abschnitt eintreten.

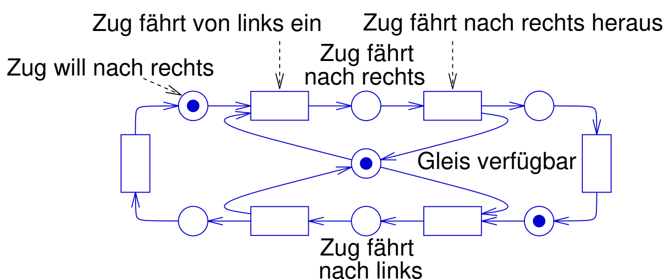


Abb. 2.48 Konflikt um die Ressource „Gleisabschnitt" ∇

In solchen Situationen wird das nächste Ereignis nichtdeterministisch ausgewählt. Netzanalysen müssen dabei alle möglichen Folgen von Ausführungen von Ereignissen berücksichtigen. Bei Petrinetzen modellieren wir absichtlich Nichtdeterminismus.

Ein wichtiger Vorteil von Petrinetzen liegt darin, dass sie als Grundlage für formale Beweise von Systemeigenschaften dienen können. Es existieren bereits einige Standardmethoden, um solche Beweise automatisch zu erzeugen. Um diese Beweise zu ermöglichen, benötigen wir eine formalere Definition von Petrinetzen. Wir betrachten drei Klassen von Petrinetzen: Bedingungs/Ereignis-Netze, Stellen/Transitions-Netze und Prädikat/Ereignis-Netze.

2.6.2 Bedingungs/Ereignis-Netze

Als erste Klasse von Petrinetzen werden wir Bedingungs/Ereignis-Netze formal definieren.

Definition 2.15: $N = (C, E, F)$ heißt **Netz**, genau dann, wenn die folgenden Bedingungen erfüllt sind:

1. C und E sind disjunkte Mengen.
2. $F \subseteq (E \times C) \cup (C \times E)$ ist eine zweistellige Relation, genannt Flussrelation.

Die Menge C heißt die Menge der Bedingungen und die Menge E ist die Menge der Ereignisse (auch Transitionen genannt).

Definition 2.16: Sei N ein Netz und sei $x \in (C \cup E)$. Dann heißt $^\bullet x := \{y | yFx, y \in (C \cup E)\}$ der **Vorbereich** von x. Wenn x ein Ereignis ist, dann heißt $^\bullet x$ auch „Menge der **Vorbedingungen** von x".

Definition 2.17: Sei N ein Netz und sei $x \in (C \cup E)$. Dann heißt $x^\bullet := \{y | xFy, y \in (C \cup E)\}$ der **Nachbereich** von x. Wenn x ein Ereignis ist, dann heißt $x^\bullet$ auch „Menge der **Nachbedingungen** von x".

Die Begriffe Vorbedingung und Nachbedingung werden in den Fällen bevorzugt, in denen diese Mengen tatsächlich Bedingungen $\in C$ darstellen, also $x \in E$ gilt.

Definition 2.18: Sei $(c, e) \in C \times E$. (c, e) heißt **Schleife**, wenn $cFe \wedge eFc$.

Definition 2.19: Sei $(c, e) \in C \times E$. N heißt **rein**, wenn F keine Schleifen enthält (siehe Abb. 2.49 (links)).

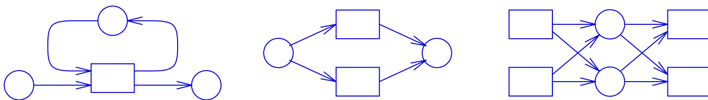


Abb. 2.49 Netze, die nicht rein (**links**) und nicht einfach sind (**Mitte** und **rechts**)

Definition 2.20: Ein Netz heißt **einfach**, wenn keine zwei Transitionen t_1 und t_2 die gleiche Menge von Vor- und Nachbedingungen haben (siehe Abb. 2.49 (Mitte) und (rechts)).

Einfache Netze ohne isolierte Elemente, die einige zusätzliche Anforderungen erfüllen, heißen **Bedingungs/Ereignis-Netze**. Bedingungs/Ereignis-Netze sind ein Sonderfall von bipartiten Graphen (Graphen mit zwei unterschiedlichen, disjunkten Mengen von Knoten). Die zusätzlich notwendigen Bedingungen werden wir hier nicht weiter vertiefen, da wir uns im Weiteren mit allgemeineren Klassen von Netzen beschäftigen.

2.6.3 Stellen/Transitions-Netze

Bei Bedingungs/Ereignis-Netzen gibt es höchstens eine Marke (engl. *token*) pro Bedingung. Für viele Anwendungen ist es nützlich, diese Einschränkung aufzuheben und mehrere Marken pro Bedingung zu erlauben. Netze, die mehrere Marken pro Bedingung erlauben, heißen Stellen/Transitions-Netze (engl. *place/transition nets*). Stellen entsprechen den bisher verwendeten Bedingungen, Transitionen entsprechen den Ereignissen. Die Anzahl von Marken pro Stelle heißt **Belegung**. Mathematisch betrachtet ist eine Belegung eine Abbildung von der Menge der Stellen auf die Menge der natürlichen Zahlen, die um ein besonderes Symbol erweitert wird: ω steht für unendlich.

Sei $\mathbb{N}_0$ die Menge der natürlichen Zahlen inklusive der 0. Ein Stellen/Transitionen-Netz kann formal wie folgt definiert werden:

Definition 2.21: (P, T, F, K, W, M_0) heißt **Stellen/Transitions-Netz** genau dann, wenn

1. $N = (P, T, F)$ ein Netz ist, mit den Stellen $p \in P$ und den Transitionen $t \in T$,
2. die Abbildung $K : P \rightarrow (\mathbb{N}_0 \cup \{\omega\}) \setminus \{0\}$ die Kapazität der Stellen darstellt (wobei ω eine unbeschränkte Kapazität symbolisiert),
3. die Abbildung $W : F \rightarrow (\mathbb{N}_0 \setminus \{0\})$ das Gewicht der Kanten des Graphen darstellt
4. und die Abbildung $M_0 : P \rightarrow \mathbb{N}_0 \cup \{\omega\}$ die initiale Belegung der Stellen mit Marken darstellt.

Kantengewichte haben einen Einfluss auf die Anzahl der Marken, die benötigt werden, bevor eine Transition schalten kann und geben auch die Anzahl von Marken an, die von einer schaltenden Transition erzeugt werden. Sei $M(p)$ die aktuelle Belegung der Stelle $p \in P$ und sei $M'(p)$ die Markierung nach dem Schalten einer Transition $t \in T$. Das Gewicht der Kanten, die zu Vorbedingungen von t gehören, gibt die Anzahl der Marken an, die aus den Vorbedingungs-Stellen abgezogen werden. Entsprechend stellt das Gewicht der Kanten, die zu Nachbedingungen führen, die Anzahl der Marken dar, die den Stellen in den Nachbedingungen hinzugefügt werden. Formal wird die neue Belegung M' wie folgt berechnet:

$$M'(p) = \begin{cases} M(p) - W(p,t), & \text{falls } p \in {}^\bullet t \setminus t^\bullet \\ M(p) + W(t,p), & \text{falls } p \in t^\bullet \setminus {}^\bullet t \\ M(p) - W(p,t) + W(t,p), & \text{falls } p \in {}^\bullet t \cap t^\bullet \\ M(p) & \text{sonst} \end{cases}$$

Abb. 2.50 zeigt ein Beispiel, wie sich das Schalten von Transition t_j auf die aktuelle Markierung auswirkt.

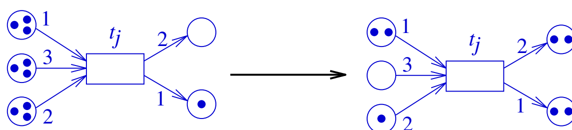


Abb. 2.50 Erzeugung einer neuen Markierung

Für unbeschriftete Kanten vereinbart man ein Gewicht von 1 und Knoten ohne entsprechende Angabe haben eine unbeschränkte Kapazität ω.

Jetzt müssen noch die beiden Bedingungen beschrieben werden, die erfüllt sein müssen, bevor eine Transition $t \in T$ schalten kann:

- in allen Stellen p in der Menge der Vorbedingungen muss die Anzahl der Marken mindestens so groß sein wie das Gewicht der Kante von p nach t und
- in allen Stellen p in der Menge der Nachbedingungen muss die Kapazität groß genug sein, um die von t neu erzeugten Marken aufzunehmen.

Formal kann dies wie folgt definiert werden:

Definition 2.22: Transition $t \in T$ heißt **M-aktiviert**, wenn sie die folgende komplexe Bedingung erfüllt:

$$(\forall p \in {}^{\bullet}t : M(p) \geq W(p,t)) \wedge (\forall p' \in t^{\bullet} : M(p') + W(t,p') \leq K(p'))$$

Aktivierte Transitionen können schalten, müssen dies aber nicht zwangsläufig. Wenn mehrere Transitionen aktiviert sind, ist die Reihenfolge ihres Schaltens nicht deterministisch definiert.

Der Einfluss einer schaltenden Transition t auf die Markenzahl kann bequem durch einen der Transition zugeordneten Vektor $\underline{t}$ beschrieben werden, der wie folgt definiert ist:

$$\underline{t}(p) = \begin{cases} -W(p,t), & \text{falls } p \in {}^{\bullet}t \setminus t^{\bullet} \\ +W(t,p), & \text{falls } p \in t^{\bullet} \setminus {}^{\bullet}t \\ -W(p,t) + W(t,p), & \text{falls } p \in {}^{\bullet}t \cap t^{\bullet} \\ 0 & \text{sonst} \end{cases}$$

Beim Schalten einer Transition t ergibt sich dann die neue Markenzahl M' für alle Stellen p wie folgt:

$$M'(p) = M(p) + \underline{t}(p)$$

Unter Benutzung der Vektoraddition können wir verkürzt schreiben:

$$M' = M + \underline{t}$$

Aus der Menge aller Vektoren können wir eine sogenannte Inzidenzmatrix $\underline{N}$ bilden, welche als Spalten die Vektoren der verschiedenen Transitionen enthält:

$$\underline{N} : P \times T \rightarrow \mathbb{Z}; \quad \forall t \in T : \underline{N}(p,t) = \underline{t}(p)$$

Mit Hilfe dieser Matrix kann man auf standardisierte Weise formale Beweise von Systemeigenschaften führen. Beispielsweise kann es Teilmengen der Stellen geben, in denen sich die Gesamtzahl der Marken unabhängig von den schaltenden Transitionen nicht verändert [468]. Solche Stellenmengen konstanter Markensumme nennen wir **S-Invarianten**. Um solche S-Invarianten zu finden, betrachten wir zunächst eine Transition t_j und suchen Stellenmengen $R \subseteq P$, für die das Schalten der Transition die Markenzahl nicht verändert. Für diese muss gelten:

$$\sum_{p \in R} \underline{t_j}(p) = 0 \tag{2.14}$$

Abbildung 2.51 zeigt ein Beispiel für eine Transition, bei der für die drei Stellen die Markensumme konstant bleibt.

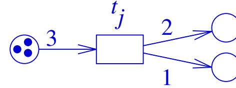


Abb. 2.51 Transition mit konstanter Markensumme

Zur Vereinfachung der Summenschreibweise in Gleichung (2.14) führen wir nunmehr den sogenannten charakteristischen Vektor $\underline{c}_R$ einer Stellenmenge R ein:

$$\underline{c}_R(p) = \begin{cases} 1 \text{ wenn } p \in R \\ 0 \text{ wenn } p \notin R \end{cases}$$

Mit dieser Definition können wir Gleichung (2.14) umschreiben zu:

$$\sum_{p \in R} \underline{t}_j(p) = \sum_{p \in P} \underline{t}_j(p) \cdot \underline{c}_R(p) = \underline{t}_j \cdot \underline{c}_R = 0 \tag{2.15}$$

Dabei kennzeichnet · das Skalarprodukt. Wir suchen jetzt Stellenmengen, für die das Schalten **aller** Transitionen die Markensumme konstant lässt. Dann muss die Gleichung (2.15) statt für eine Transition t_j für alle Transitionen gelten:

$$\begin{aligned} \underline{t}_1 \cdot \underline{c}_R &= 0 \\ \underline{t}_2 \cdot \underline{c}_R &= 0 \\ &\dots \\ \underline{t}_n \cdot \underline{c}_R &= 0 \end{aligned} \tag{2.16}$$

Das Gleichungssystem (2.16) lässt sich mit der transponierten Inzidenzmatrix $\underline{N}^T$ zusammenfassen zu:

$$\underline{N}^T \cdot \underline{c}_R = 0 \tag{2.17}$$

Das Gleichungssystem (2.17) ist ein lineares homogenes Gleichungssystem. Die Matrix $\underline{N}$ beschreibt die Kantengewichte des Petrinetzes. Gesucht sind Vektoren $\underline{c}_R$, welche dieses Gleichungssystem lösen. Da die Lösungsvektoren charakteristische Vektoren sein müssen, können wir als Komponenten der Vektoren nur 0 und 1 erlauben[15]. Das Lösen derartiger Gleichungssysteme ist komplexer als das Lösen von Gleichungssystemen mit reellwertigen Lösungsvektoren. Dennoch können durch das Lösen von Gleichung (2.17) Aussagen über Eigenschaften eines Petrinetzes gewonnen werden. In unserem Beispiel ändert sich die Anzahl der Züge, die zwischen Köln und Paris verkehren, nicht. Das gleiche gilt für die Züge zwischen Amsterdam und Paris. In Modellen, die den Zugriff auf gemeinsame Ressourcen modellieren, kann beispielsweise der gegenseitige Ausschluss nachgewiesen werden.

[15] Wenn wir gewichtete Markensummen betrachten, können wir natürliche Zahlen als Lösungen erlauben.

Beispiel 2.24: Wir betrachten nun ein größeres Beispiel: Wieder geht es um die Synchronisation von Zügen, in diesem Fall werden die Hochgeschwindigkeitszüge vom Typ „Thalys" modelliert, die zwischen Amsterdam, Köln, Brüssel und Paris verkehren. Es gibt unabhängige Zugteile, die von Amsterdam und Köln nach Brüssel fahren. Dort werden die Zugteile verbunden und fahren dann gemeinsam nach Paris weiter. Auf dem Rückweg von Paris werden die Züge in Brüssel wieder getrennt. Wir nehmen an, dass die Züge in Paris Anschlüsse an Züge aus dem Pariser Süden sicherstellen müssen. Das zugehörige Petrinetz zeigt Abb. 2.52.

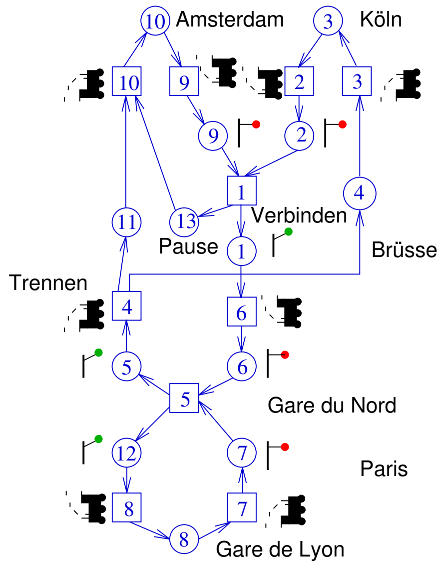


Abb. 2.52 Modell der Thalys-Züge zwischen Amsterdam, Köln, Brüssel und Paris

Sofern die Bedingungen gemäß Stellen 3 und 10 erfüllt sind, modellieren sie Züge, die in Amsterdam bzw. Köln warten. Die Transitionen 9 und 2 bilden Züge nach, die von diesen Städten aus nach Brüssel fahren. Nach der Ankunft in Brüssel enthalten die Stellen 9 und 2 jeweils eine Marke. Transition 1 symbolisiert das Verbinden der beiden Zugteile. Die Bedingung gemäß Stelle 13 ist erfüllt, sofern einer der beiden Lokführer in Brüssel eine Pause hat, während der andere nach Paris weiterfährt. Transition 5 stellt die Anschlussbedingungen mit anderen Zügen im Gare du Nord von Paris dar. Diese Anschlüsse verbinden den Gare du Nord mit anderen Pariser Bahnhöfen (wir haben hier lediglich den Gare de Lyon als Beispiel gezeigt, obwohl es in Paris noch weitere Bahnhöfe gibt). Selbstverständlich fahren die Thalys-Züge nicht mit Dampflokomotiven – die verwendeten Symbole sind aber eingängiger als die von modernen Hochgeschwindigkeitszügen. Tabelle 2.3 enthält die Matrix N^T für dieses Beispiel.

Tabelle 2.3 N^T für das Thalys-Beispiel

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}
t_1	1	-1							-1				1
t_2		1	-1										
t_3			1	-1									
t_4				1	-1						1		
t_5					1	-1	-1					1	
t_6	-1					1							
t_7							1	-1					
t_8								1				-1	
t_9									1	-1			
t_{10}										1	-1		-1

In diesem Beispiel ist in Zeile 2 dargestellt, dass das Schalten von t_2 die Anzahl der Marken bei p_2 um eine erhöht und die Anzahl der Marken bei p_3 um eine vermindert. Unter Einsatz von Methoden der linearen Algebra können wir zeigen, dass die folgenden vier Vektoren Lösungen dieses linearen Gleichungssystems darstellen:

$c_{R,1}$ = (1,1,1,1,1,1,0,0,0,0,0,0,0)
$c_{R,2}$ = (1,0,0,0,1,1,0,0,1,1,1,0,0)
$c_{R,3}$ = (0,0,0,0,0,0,0,0,1,1,0,0,1)
$c_{R,4}$ = (0,0,0,0,0,0,1,1,0,0,0,1,0)

Diese Vektoren entsprechen den Orten entlang der Gleise für Züge aus Köln und Amsterdam sowie den Orten entlang des Weges für Lokführer von Zügen aus Amsterdam und den Orten entlang der Gleise innerhalb von Paris. Damit können wir zeigen, dass die Anzahl von Zügen und Lokführern entlang der Gleise konstant ist (was wir natürlich erwartet haben). Dieses Beispiel zeigt, dass S-Invarianten als Standardtechnik eingesetzt werden können, um Eigenschaften von Systemen zu beweisen. ∇

2.6.4 Prädikat/Transitions-Netze

Sowohl Bedingungs/Ereignis-Netze als auch Stellen/Transitions-Netze können für größere Beispiele schnell sehr groß und unübersichtlich werden. Eine Verringerung der Größe ist häufig durch den Einsatz von Prädikat/Transitions-Netzen möglich.

Beispiel 2.25: Wir werden dies am Beispiel der „dinierenden Philosophen“ demonstrieren. Das Problem geht von einer Anzahl von Philosophen aus, die an einem runden Tisch essen. Vor jedem Philosophen steht ein Teller mit Spaghetti (siehe Abb. 2.53). Zwischen den Tellern befindet sich jeweils nur eine Gabel. Jeder Philosoph ist entweder mit Essen oder mit Nachdenken beschäftigt. Essende Philosophen benötigen dafür die beiden Gabeln, die neben ihrem Teller liegen. Folglich kann ein Philosoph nur dann essen, wenn seine beiden Nachbarn gerade nicht essen.

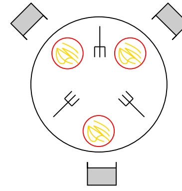


Abb. 2.53 Das Problem der dinierenden Philosophen

Diese Situation kann, wie in Abb. 2.54 gezeigt, in einem Bedingungs/Ereignis-Netz modelliert werden.

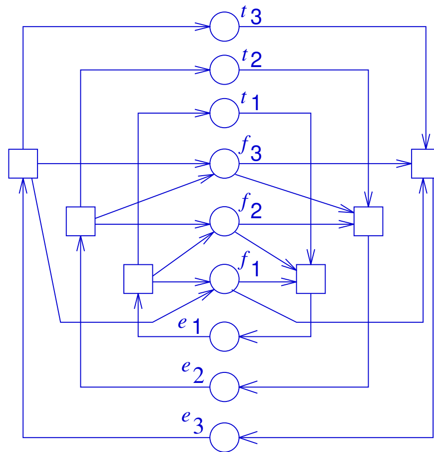


Abb. 2.54 Bedingungs/Ereignis-Netz des Philosophen-Problems

Die Bedingungen t_j entsprechen dem „denkenden“, die Bedingungen e_j dem „essenden“ Zustand und die Bedingungen f_j stellen die verfügbaren Gabeln dar. In Anbetracht der geringen Größe des zugrundeliegenden Problems ist dieses Netz bereits recht groß. Das Netz lässt sich verkleinern, wenn man ein Prädikat/Transitions-Netz verwendet. Abb. 2.55 zeigt ein Modell des Philosophen-Problems als Prädikat/Transitions-Netz. Bei Prädikat/Transitions-Netzen haben Marken eine Identität und können unterschieden werden. Man kann sich dies auch als Kennzeichnung der Marken durch **Farben** vorstellen. Daher heißen diese Netze im Englischen auch *Coloured Petri Nets* (CPN) [273]. Die Identität wird in Abb. 2.55 benutzt, um die drei Philosophen p_1 bis p_3 zu unterscheiden und um die Gabel f_3 zu identifizieren. Des weiteren können Kanten mit Beschriftungen versehen werden, die Variablen und Funktionen repräsentieren. Im Beispiel werden Variablen verwendet, um die Identität der Philosophen zu beschreiben. Die Funktionen $l(x)$ bzw. $r(x)$ beschreiben die linke bzw. rechte Gabel von Philosoph x. Diese beiden Gabeln werden als

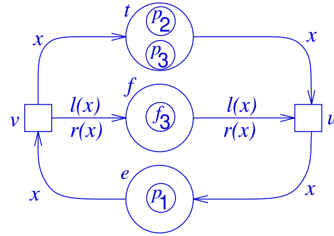


Abb. 2.55 Prädikat/Transitions-Netz des Philosophen-Problems

Vorbedingung für die Transition u benötigt und sie werden als Nachbedingung beim Schalten der Transition v wieder zurückgegeben. Dieses Modell kann einfach durch das Hinzufügen weiterer Marken auf den Fall von $n > 3$ Philosophen erweitert werden. Im Gegensatz zu dem Netz in Abb. 2.54 muss die eigentliche Struktur des Netzes dazu nicht verändert werden. ∇

2.6.5 Bewertung

Der Hauptvorteil von Petrinetzen ist ihre Stärke bei der Modellierung kausaler Abhängigkeiten. Standard-Petrinetze bieten keine Unterstützung für Zeitbedingungen. Alle Entscheidungen können lokal getroffen werden, indem Transitionen mit ihren Vor- und Nachbedingungen analysiert werden. Aus diesem Grund können sie zur Modellierung von geographisch verteilten Systemen verwendet werden. Außerdem gibt es starke theoretische Grundlagen für die Betrachtung von Petrinetzen, was formale Beweise von Systemeigenschaften vereinfacht. Petrinetze sind nicht notwendigerweise deterministisch: unterschiedliche Schaltreihenfolgen können zu unterschiedlichen Ergebnissen führen. Die Beschreibungsstärke von Petrinetzen entspricht von der Stärke her der anderer Berechnungsmodelle inklusive endlicher Automaten.

In manchen Zusammenhängen sind die Stärken von Petrinetzen aber auch ihre Schwächen. Wenn Zeitbedingungen zu berücksichtigen sind, können Standard-Petrinetze nicht verwendet werden. Außerdem bieten diese kein Hierarchiekonzept und keine Programmiersprachenkonstrukte an, von objektorientierten Konzepten ganz abgesehen. In der Regel ist es schwierig, Daten in Petrinetzen darzustellen.

Es gibt Erweiterungen von Petrinetzen, die einige dieser Schwächen beheben. Allerdings gibt es keine universelle Petrinetz-Erweiterung, die alle Anforderungen, die am Anfang dieses Kapitels aufgestellt wurden, erfüllt. Trotzdem haben sich Petrinetze aufgrund der zunehmenden Anzahl verteilter Systeme in der ganzen Welt zunehmend verbreitet.

UML beinhaltet erweiterte Petrinetze unter der Bezeichnung **Aktivitätsdiagramme**. Diese ergänzen Petrinetze um Symbole, die Entscheidungen kennzeichnen (wie in gewöhnlichen Flussdiagrammen). Die Platzierung der Symbole erfolgt ähnlich zu SDL.

Beispiel 2.26: Ein Beispiel dafür ist in Abb. 2.56 zu sehen. Das Beispiel zeigt die während eines Standardisierungsprozesses durchzuführenden Vorgänge. Verzwei-

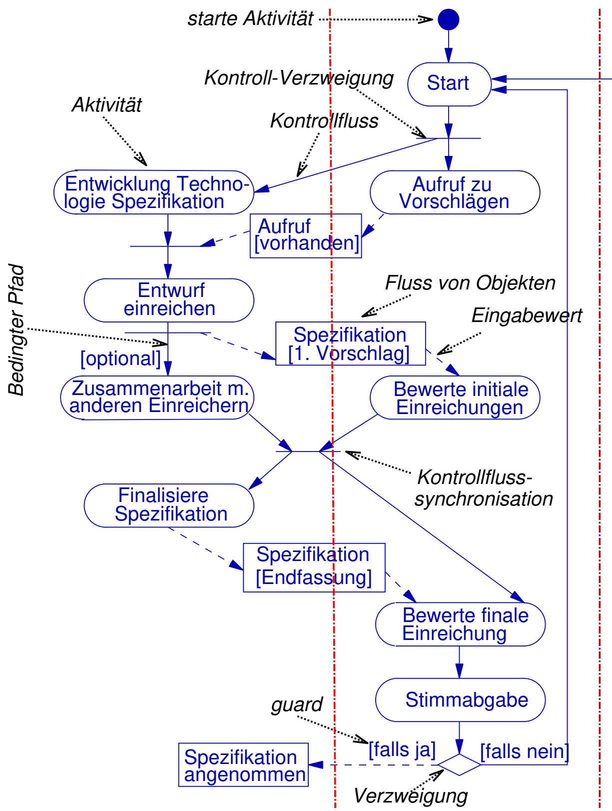


Abb. 2.56 Aktivitätsdiagramm [300]

gungen und Vereinigungen des Kontrollflusses entsprechen den Transitionen in Petrinetzen, deren Symbole sie auch verwenden (horizontale Balken). Die am unteren Ende der Abbildung zu sehende Raute steht für eine Entscheidung. Aktivitäten lassen sich in Bahnen ordnen (Flächen zwischen den gestrichelten Linien), damit lassen sich die unterschiedlichen Zuständigkeiten und die ausgetauschten Dokumente visualisieren. ∇

Es ist interessant zu sehen, dass Petrinetze ursprünglich nicht weit verbreitet waren und erst Jahre nach ihrer Erfindung durch die Verwendung in UML zu einer häufig angewendeten Modellierungstechnik wurden.

2.7 Diskrete, ereignisbasierte Sprachen

2.7.1 *Simulationszyklus diskreter Ereignissysteme*

Das diskrete, ereignisbasierte Berechnungsmodell basiert auf der Simulation der Erzeugung und Verarbeitung von Ereignissen im Zeitverlauf. Es verwendet eine Warteschlange zukünftiger Ereignisse, die nach der Zeit sortiert sind, zu der sie bearbeitet werden sollen. Die Semantik definiert, dass Ereignisse, die zum aktuellen Zeitpunkt stattfinden sollen, aus der Warteschlange entfernt werden, die ihnen zugeordneten Aktionen ausgeführt werden und bei Bedarf neue Ereignisse zur Warteschlange hinzugefügt werden können. Die Zeit schreitet weiter fort, wenn für den aktuellen Zeitpunkt keine weitere Aktion definiert ist. Dies ist das Grundmodell der Simulation:

```
loop
   Hole das nächste Ereignis aus der Warteschlange;
   Führe die Aktion aus, die mit dem Ereignis definiert ist
   (z.B. eine Variablenzuweisung; dabei können neue Ereignisse entstehen);
until  das Abbruchkriterium ist erfüllt;
```

Verbreitete Hardwarebeschreibungssprachen (engl. *Hardware Description Languages* (HDLs)) basieren meist auf dem diskreten Ereignismodell. Wir werden HDLs als ein bekanntes Beispiel der Modellierung auf der Basis eines diskreten Ereignissystems nutzen.

Beispiel 2.27: Die Anwendung des allgemeinen Simulationsschemas kann am Beispiel der Simulation eines RS-Flipflops demonstriert werden (siehe Abb. 2.57).

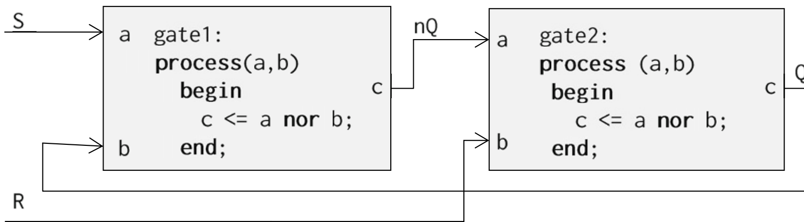


Abb. 2.57 RS-Flipflop aus zwei NOR-Gattern

Die Schaltung besteht aus zwei kreuzweise verbundenen NOR-Gattern. Die Abbildung zeigt auch dazu gehörigen Code in einer HDL, in diesem Fall VHDL.

Tabelle 2.4 enthält eine repräsentative Folge von Ein- und Ausgabewerten zu dieser Schaltung. Wir nehmen an, dass das Flipflop zunächst gesetzt ist und dass dieser Zustand gehalten wird, d.h. `Q` ist `'1'` und `R = S = '0'`. Prozesse `gate1` und `gate2` beschreiben das Verhalten der beiden NOR-Gatter. Diese Prozesse sind

Tabelle 2.4 Folge von Werten an Ein- und Ausgängen eines RS-Flipflops

	$t < 0$	t=0	$t > 0$		
R	'0'	'1'	'1'	'1'	'1'
S	'0'	'0'	'0'	'0'	'0'
Q	'1'	'1'	'0'	'0'	'0'
nQ	'0'	'0'	'0'	'1'	'1'

zunächst inaktiv, während sie auf Ereignisse an den Eingängen `a` oder `b` warten. Dieses Warten wird durch die Listen `(a,b)` ausgedrückt. Man sagt, `gate1` und `gate2` seien **sensitiv** bezüglich der Einträge in der Liste.

Angenommen, zur Zeit 0 ändern wir den Wert am Eingang `R`, dem Rücksetzeingang, auf `'1'`. Wir erwarten, dass das Flipflop zurückgesetzt wird. In Form von Ereignissen passiert dies wie folgt: Die Änderung am Eingang `R` ist ein Ereignis, welches in der Warteschlange gespeichert wird. Dieses Ereignis wird sofort verarbeitet, denn es ist das einzige Ereignis in der Warteschlange. Dieses Ereignis weckt `gate2`, da dieses sensitiv ist bezüglich Änderungen an seinem Eingang `b`. `gate2` wird daraufhin die `nor`-Funktion berechnen (mit einem Ergebnis von `'0'`) und wird sodann die Zuweisung `c <='0'` ausführen. Diese Schreibweise bezeichnet eine sogenannte **Signalzuweisung**. Dies bedeutet, dass die neuen Werte zunächst nur in der Warteschlange gespeichert werden. Die tatsächliche Zuweisung zur Variablen auf der linken Seite erfolgt erst, wenn der Zeitpunkt zur Bearbeitung dieses Eintrags in der Warteschlange erreicht ist. In unserem Beispiel wird ein Ereignis erzeugt, welches verlangt, dass Ausgang `c` von `gate2` auf `'0'` gesetzt wird und dieses Ereignis wird in der Warteschlange gespeichert.

Dieses Ereignis wird sofort aus der Warteschlange geholt, da es das einzige Ereignis ist. Das Ereignis wird Ausgang `c` auf `'0'` setzen. Diese Änderung wird `gate1` wecken, da `gate1` darauf sensitiv ist. `gate1` wird folglich ebenfalls die `nor` -Funktion berechnen. Diese Berechnung führt zu einem Ereignis, als dessen Wirkung Ausgang `c` von `gate1` auf `'1'` zu setzen ist. Dieses Ereignis wird ebenfalls in der Warteschlange gespeichert.

Dieses Ereignis wird ebenfalls sofort verarbeitet und es führt zum gewünschten Setzen des Werts am Ausgang. Diese Änderung wird `gate2` erneut wecken. `gate2` berechnet wieder den Wert `'0'` am Ausgang. Der weitere Verlauf wird etwas davon abhängen, auf welche Weise man erkennt, dass sich ein stabiler Zustand eingestellt hat und dass keine weiteren Ereignisse zu erzeugen sind.

Im Beispiel hätten wir echte Verzögerungen um physikalische Zeiten hinzufügen können und so eine Übersicht über die verstrichene Zeit gehabt. Insgesamt approximiert diese ereignisbasierte Simulation das Verhalten eines echten Flipflops. ∇

2.7.2 Mehrwertige Logik

Welche Werte sollten wir für die Simulation im obigen Beispiel benutzen? In diesem Buch beschränken wir uns auf die Beschreibung von eingebetteten Systemen, die mit

binärer Logik implementiert werden. Trotzdem ist es ratsam oder sogar notwendig, mehr als zwei Signalwerte zur Modellierung solcher Systeme zu verwenden. Beispielsweise könnte ein System elektrische Signale unterschiedlicher Stärke enthalten, und die Stärke und der Wert des Signals, das entsteht, wenn man mehrere solcher Signale zusammenschaltet, müssen berechnet werden. Im Folgenden werden wir deshalb zwischen dem **Wert** und der **Stärke** eines Signals unterscheiden. Während Ersteres eine Abstraktion der Signalspannung darstellt, ist Letzteres eine Abstraktion der Impedanz (des Innenwiderstands) der Spannungsquelle. Wir werden diskrete Wertemengen verwenden, um die Signalwerte und Signalstärken zu beschreiben. Unbekannte elektrische Signale werden durch spezielle Signalwerte modelliert.

In der Praxis verwenden elektronische Entwurfssysteme eine Vielzahl von Wertemengen. Einige Systeme erlauben nur zwei Werte, andere 9 oder sogar 46. Das Ziel bei der Entwicklung solcher diskreter Wertemengen ist es, das Lösen von Netzwerkgleichungen (z.B. nach Kirchhoff) zu vermeiden, aber existierende Systeme trotzdem ausreichend genau zu modellieren.

Im Folgenden zeigen wir einen Weg, wie man systematisch zu Wertemengen kommt und diese zueinander in Relation setzt. Wir werden die Stärke der elektrischen Signale als Hauptunterscheidungsmerkmal für verschiedene Wertemengen verwenden. Eine systematische Methode, wie man solche Wertemengen aufbauen kann, die CSA-Theorie, wurde von Hayes vorgestellt [209]. Später werden wir zeigen, wie die normalerweise von VHDL-Modellen verwendete Wertemenge als Spezialfall erzeugt werden kann.

Eine Signalstärke (Zwei Logikwerte)

Im einfachsten Fall werden lediglich zwei Logikwerte verwendet. Sie heißen `'0'` und `'1'`. Diese Werte werden als gleich stark angenommen. Das bedeutet, wenn zwei Kabel miteinander verbunden werden, von denen eines den Wert `'0'`, das andere den Wert `'1'` hat, so kann man den Wert des entstehenden Signals nicht bestimmen.

Eine einzige Signalstärke kann ausreichend sein, um ein System zu modellieren, bei dem nie zwei Signale mit unterschiedlichen Werten verbunden werden und in dem sich keine Signale mit unterschiedlichen Stärken in einer bestimmten Hardware-Einheit treffen.

Zwei Signalstärken (Drei und Vier Logikwerte)

In vielen Schaltungen gibt es den Fall, dass ein elektrisches Signal nicht aktiv von einem Ausgang getrieben wird. Dies kann vorkommen, wenn eine Leitung weder mit Masse noch mit der Versorgungsspannung oder einem anderen Knoten der Schaltung verbunden ist.

Beispielsweise können Systeme sogenannte *Open-Collector*-Ausgänge haben (siehe Abb. 2.58 (links))[16]. Der Ausgang A ist effektiv abgeschaltet, wenn der *Pull-Down*-Transistor PD nicht leitet. Bei Tristate-Ausgängen (siehe Abb. 2.58 (rechts))

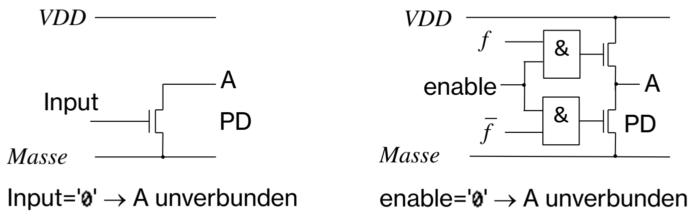


Abb. 2.58 Abschaltbare Ausgänge: **links**: *Open-Collector*-Ausgang; **rechts**: Tristate-Ausgang

wird ein Wert '0' des enable-Signals eine '0' an den Ausgängen der UND-Gatter (mit & gekennzeichnet) erzeugen und beide Transistoren nichtleitend machen[17]. Damit können wir mit geeigneten Eingangssignalen Ausgänge effektiv vom Rest der Schaltung trennen.

Offensichtlich ist die Signalstärke eines solchen unverbundenen Ausgangs die kleinste Signalstärke, die man sich vorstellen kann. Insbesondere ist diese Signalstärke kleiner als die von '0' und '1'. Außerdem ist der Signalwert eines solchen Signals unbekannt. Diese Kombination von Signalstärke und Signalwert wird durch den Logikwert mit dem Namen 'Z' dargestellt. Wenn ein Signalwert 'Z' mit irgendeinem anderen Signal zusammengeschaltet wird, dominiert immer das andere Signal. Wenn also beispielsweise zwei *Tristate*-Ausgänge an denselben Bus angeschlossen sind und einer der Ausgänge steuert ein 'Z' bei, so ist der Ergebniswert auf dem Bus immer der Wert des zweiten Ausgangs (siehe Abb. 2.59).

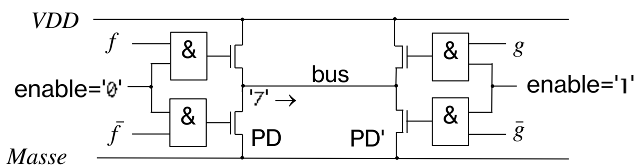


Abb. 2.59 Der rechte Ausgang dominiert den Bus

Was passiert, wenn die beiden Ausgänge in Abb. 2.59 versuchen, starke Signale mit unterschiedlichen Werten beizusteuern? Zur Behandlung solcher Situationen

[16] Schaltpläne sollen den Studenten helfen, die Signalwerte zu verstehen und es nicht schwieriger machen. Studierende, denen Schaltpläne fremd sind, können einfach die Logikwerte betrachten.

[17] In der Praxis können *Pull-Up*-Transistoren Verarmungstransistoren sein und die *Tristate*-Ausgänge können invertierend sein.

werden dreiwertige Logikmengen {'0', '1', 'Z'} in der Regel durch einen vierten Wert mit dem Namen 'X' ergänzt. 'X' beschreibt den unbekannten Signalwert der gleichen Stärke wie '0' und '1'. Genauer ausgedrückt wird der Wert 'X' verwendet, um unbekannte Signale darzustellen, die entweder einen der Werte '0' oder '1' haben oder aber eine Spannung darstellen, die keinem dieser beiden Werte entspricht[18].

Wenn mehrere Signale verbunden werden, dann müssen wir den resultierenden Signalwert berechnen. Dies kann man tun, indem man die partielle Ordnung der vier Signalwerte '0', '1', 'Z' und 'X' gemäß ihrer Signalstärke ausnutzt. Diese partielle Ordnung ist im **Hasse-Diagramm** in Abb. 2.60 dargestellt: Die Kanten in

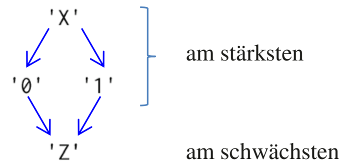


Abb. 2.60 Partielle Ordnung der Wertemenge {'0', '1', 'Z', 'X'}

der Abbildung zeigen die Dominanz von Signalwerten. Die Kanten definieren eine Relation $>$ auf den Elementen. Wenn $a > b$, dann dominiert a das Signal b (oder b wird von a dominiert). '0' und '1' dominieren 'Z' , wohingegen 'X' alle anderen Signalwerte dominiert. Ausgehend von der Relation $>$ definieren wir eine Relation $\geq$. Die Aussage $a \geq b$ gilt genau dann, wenn $a > b$ oder wenn $a = b$.

Wir definieren weiter eine Operation *sup* auf zwei Signalen, die das **Supremum** der beiden Signalwerte zurückliefert.

Definition 2.23: Seien a und b zwei Signalwerte einer partiell geordneten Menge $(S, \geq)$. Das **Supremum** $c \in S$ ist das schwächste Signal, für das $c \geq a$ und $c \geq b$ gilt.

So ist z.B. *sup* ('Z' ,'0') = '0', *sup*('Z' ,'1')='1', *sup* ('0','1') = 'X' usw.

Lemma 2.1: *Seien a und b zwei Signale einer partiell geordneten Menge, wobei die partiell geordnete Menge wie oben gewählt sei. Dann berechnet die sup-Funktion den Signalwert, der sich ergibt, wenn man die beiden Signale miteinander verbindet.*

Sup entspricht dem ***connect***-Element der **C**SA-Theorie.

Drei Signalstärken (Sieben Signalwerte)

Für viele Schaltungen sind zwei Signalstärken nicht ausreichend. Ein häufig vorkommender Fall, der mehr Werte benötigt, ist die Verwendung von sogenannten Verarmungstransistoren (engl. *depletion transistors*, siehe Abb. 2.61).

[18] Es gibt auch andere Interpretationen von 'X' [65], aber die oben genannte Interpretation ist für unsere Zwecke am Besten geeignet.

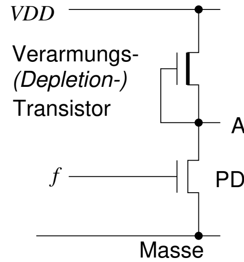


Abb. 2.61 Ausgang mit Verarmungstransistoren

Der Effekt von Verarmungstransistoren ist vergleichbar mit der Verwendung eines Widerstands, der eine hochohmige Verbindung zur Versorgungsspannung VDD herstellt. Verarmungstransistoren dienen genau wie der *Pull-Down-Transistor* PD als Signaltreiber für den Knoten A der Schaltung und der Signalwert am Knoten A kann mit Hilfe einer Supremumsbildung berechnet werden. Der *Pull-Down*-Transistor liefert einen Treiberwert von '0' oder 'Z' , je nachdem, wie der Eingabewert von PD ist. Der Verarmungstransistor liefert ein Signal, das schwächer ist als '0' und '1'. Der Signalwert entspricht jedoch einem Wert von '1'. Wir bezeichnen den Signalwert, den der Verarmungstransistor liefert, mit 'H' und wir nennen ihn „schwache logische Eins“. Analog kann es auch „schwache logische Nullen“ geben, die durch 'L' gekennzeichnet werden. Der Wert, der entsteht, wenn man eine Verbindung zwischen 'H' und 'L' herstellt, heißt „schwach logisch undefiniert“, geschrieben als 'W' . Daraus ergeben sich drei Signalstärken und die sieben Signalwerte {'0', '1', 'Z', 'X', 'H', 'L', 'W'}. Die Berechnung des resultierenden Wertes basiert wieder auf der partiellen Ordnung aus diesen sieben Signalwerten, die in Abb. 2.62 gezeigt wird. Diese Ordnung definiert wieder eine Operation *sup*, die das schwächste

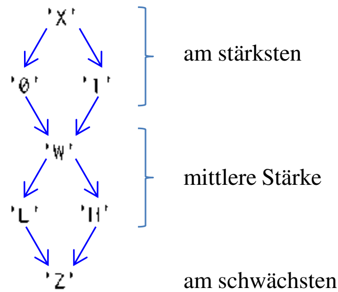


Abb. 2.62 Partielle Ordnung für die Wertemenge {'0', '1', 'Z', 'X', 'H', 'L', 'W'}

Signal zurückliefert, das mindestens so stark ist wie eines der beiden Argumente. Beispielsweise gilt *sup*('H', '0')='0', *sup*('H', 'Z')='H', *sup*('H', 'L') = 'W'.

'0' und 'L' kennzeichnen den gleichen Signalwert, aber mit unterschiedlicher Signalstärke. Das gleiche gilt für '1' und 'H'. Geräte, welche die Stärke eines Signals erhöhen, heißen **Verstärker**. Geräte, welche die Stärke eines Signals erniedrigen, heißen **Abschwächer** (engl. *attenuator*).

Vier Signalstärken (Zehn Signalwerte)

In einigen Fällen sind auch drei Signalstärken nicht ausreichend. Beispielsweise gibt es Schaltungen, die Verbindungsleitungen als Ladungsspeicher verwenden. Solche Leitungen werden während einer bestimmten Betriebsphase mit den Logikwerten `'0'` oder `'1'` vorgeladen. Diese Ladungen können den (hochohmigen) Eingang einiger Transistoren steuern. Werden diese Leitungen mit einer Signalquelle (außer einer mit dem Wert `'Z'`) verbunden, so verlieren sie ihre Ladung und das Signal der zweiten Quelle dominiert.

Beispiel 2.28: In Abb. 2.63 wird beispielsweise ein Bus von einem speziellen Ausgang getrieben. Angenommen, der Bus stellt eine hoch-kapazitive Belastung dar,

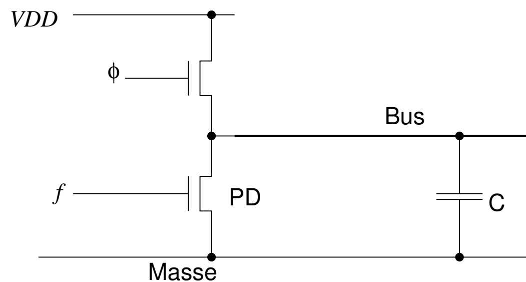


Abb. 2.63 Vorladen eines Busses

äquivalent zu einem Kondensator `C` mit hoher Kapazität. Mit dieser gehen wir wie folgt um: Während die Funktion f noch den Wert `'0'` liefert, wird ϕ auf den Wert `'1'` gesetzt, wodurch der Kondensator `C` aufgeladen wird. Danach wird ϕ auf `'0'` gesetzt. Wenn der Wert von f bekannt wird und wenn er gleich `'1'` ist, wird der Bus entladen. ∇

Dieses sogenannte Vorladen oder *Precharging* wird verwendet, da das Aufladen eines Busses hoher Kapazität mit einem Ausgang wie dem in Abb. 2.61 gezeigten ein langsamer Prozess ist, weil der Widerstand des Verarmungstransistors relativ groß ist. Die Entladung durch normale „*Pull-Down*-Transistoren" `PD` erfolgt dagegen schneller.

Um solche Fälle modellieren zu können, brauchen wir Signalwerte, die schwächer sind als `'H'` und `'L'`, aber stärker als `'Z'`. Wir nennen solche Werte „sehr schwache Signalwerte" und schreiben sie als `'h'` und `'l'`. Der zugehörige sehr schwache undefinierte Signalwert heißt `'w'`. Damit erhalten wir zehn Signalwerte {`'0'`, `'1'`, `'L'`, `'H'`, `'l'`, `'h'`, `'X'`, `'W'`, `'w'`, `'Z'`}. Mit Hilfe der Signalstärken kann wieder eine partielle Ordnung auf diesen Werten definiert werden (siehe Abb. 2.64). *Precharging* ist durchaus risikobehaftet: nachdem ein vorgeladener Bus einmal entladen ist, z.B. durch ein transientes Signal, kann er in derselben Taktperiode nicht wieder aufgeladen werden.

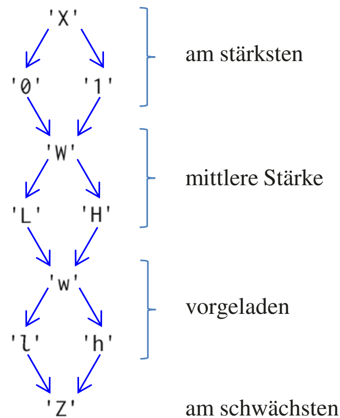


Abb. 2.64 Partielle Ordnung der Wertemenge {'0', '1', 'Z ', 'X', 'H', 'L', 'W', 'h', 'l', 'w'}

Fünf Signalstärken

Bislang haben wir die Versorgungsspannungen nicht betrachtet. Diese sind stärker als alle bisher betrachteten Signale. Signalwertemengen, welche die Versorgungsspannungen beinhalten, haben zur Definition einer 46-elementigen Signalwertmenge geführt, die auch Signalflanken erfasst [106]. Solche Modelle sind allerdings nicht sehr weit verbreitet.

2.7.3 Transaktionsbasierte Modellierung

Simulation auf der Basis diskreter Ereignissysteme erlaubt es uns, Zeitverhalten zu simulieren. Allerdings stellt sich die Frage, wie genau wir die Zeit modellieren können. Ein sehr genaues Modell, welches das Zeitverhalten der Signal präzise erfasst, wird lange Simulationszeiten verursachen. Insbesondere benötigen wir sehr lange Simulationszeiten, wenn wir elektrische Schaltkreise simulieren wollen. Kürzere Simulationszeiten sind möglich, wenn wir zyklengenaue Simulationen verwenden, bei denen wir synchrone (getaktete) Schaltungen taktgenau simulieren. Noch kürzere Simulationszeiten sind möglich, wenn wir noch gröbere Zeitmodelle verwenden. In diesem Zusammenhang hat die sogenannte transaktionsbasierte Modellierung (engl. *Transaction-Level Modeling* (TLM)) eine große Beachtung gefunden. TLM kann wie folgt definiert werden:

Definition 2.24 (Grötker et al. [191]): „**Transaktionsbasierte Modellierung** ist ein Ansatz zur Modellierung digitaler Systeme auf hoher Ebene, bei dem die Details der Kommunikation zwischen den Modulen von den Details der Implementierung der funktionellen Einheiten und der Kommunikationsarchitektur getrennt sind. Kommunikationsmechanismen wie Busse oder FIFOs werden als Kanäle modelliert und werden den Modulen als SystemC-Schnittstellenklassen angeboten. Transak-

tionsanforderungen ergeben sich aus Aufrufen der Schnittstellenfunktionen dieser Kanalmodelle, welche die Details des Informationsaustausches kapseln. Auf der Transaktionsebene liegt die Betonung mehr auf der Funktionalität des Datentransfers – welche Daten werden woher und wohin transportiert – und weniger bei der tatsächlichen Implementierung, d.h. beim tatsächlich für den Transfer benutzten Protokoll. Dieser Ansatz macht es für den Systemdesigner einfacher, beispielsweise mit verschiedenen Busarchitekturen, die alle gemeinsame Schnittstellen haben, zu experimentieren, ohne dabei die Modelle abzuändern, die mit diesen Bussen interagieren. Dabei ist vorausgesetzt, dass diese Modelle mit dem Bus (nur) über die gemeinsamen Schnittstellen kommunizieren."

Eine detaillierte Unterscheidung zwischen verschiedenen Zeitmodellen wurde von Cai and Gajski [83] beschrieben. Sie unterscheiden zwischen Zeitmodellen für die Kommunikation und für Berechnungen[19]. Cai and Gajski betrachten verschiedene Fälle von Zeitmodellen, abhängig davon, wie präzise Kommunikation und Berechnungen modelliert werden.

Sowohl für Berechnungen wie auch für die Kommunikation differenzieren wir zwischen keiner Zeitmodellierung, näherungsweiser Zeitmodellierung und zyklengenauer Zeitmodellierung. In der Abb. 2.65 markieren Kreuze unausgewogene Kombinationen von Zeitmodellen, die von Cai and Gajski nicht betrachtet wurden. Damit

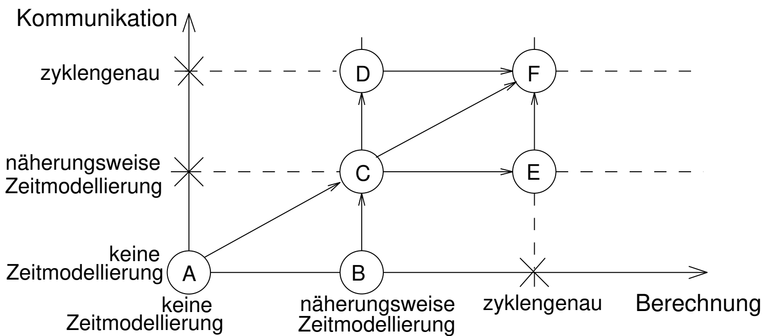


Abb. 2.65 Unterschiede zwischen verschiedenen Zeitmodellen

verbleiben noch sechs mögliche Fälle [83]:

A Keine Zeitmodellierung: wir betrachten nur die Funktionalität und betrachten überhaupt keine Zeiten. Solche Modelle sind für frühe Entwurfsphasen angebracht. Man kann sie **Spezifikationsmodell** nennen.

B Wir können im Spezifikationsmodell reine Verhaltensbeschreibungen durch Beschreibungen von Komponenten mit einem groben Zeitmodell ersetzen.

[19] Dies entspricht derselben Unterscheidung, die wir in Tabelle 2.1 auf Seite 44 vorgenommen haben.

Beispielsweise könnten wir die größtmögliche Laufzeit für die Ausführung von Code auf einem Prozessor kennen. Die Kommunikation würden wir immer noch abstrakt modellieren. Als Ergebnis würden wir den Knoten B in Abb. 2.65 erhalten. Man kann ein solches Modell ***component-assembly model*** nennen.

C In einem Modell vom Typ B könnten wir abstrakte Kommunikation durch Kommunikation mit einem näherungsweisen Zeitmodell ersetzen. Dies bedeutet, dass wir Zugriffskonflikte und ihren Einfluss auf das Zeitverhalten modellieren, aber wir modellieren nicht jedes Signal und wir beziehen uns nicht auf Taktzyklen. Ein solches Modell kann ***bus-arbitration model*** genannt werden.

D In einem Modell vom Typ C könnten wir näherungsweise Zeitinformation für die Kommunikation durch taktgenaue Zeitinformation ersetzen. Damit würden wir in der Simulation eine Übersicht über benötigte Taktzyklen bekommen. Wir könnten ggf. sogar die echte, physikalische Zeit verwenden. Das resultierende Modell, in Abb. 2.65 als Knoten D bezeichnet, können wir ***bus-functional model*** [83] nennen.

E In einem Modell vom Typ C können wir näherungsweise Zeitmodelle für Berechnungen durch ein taktgenaues Modell der Berechnungen ersetzen. Damit können wir z.B. Speicherreferenzen im Detail erfassen. Das resultierende Modell können wir **zyklengenaues Berechnungsmodell** nennen.

F Der mit F bezeichnete Knoten wird erhalten, wenn Kommunikation und Berechnungen zyklengenaue Zeiten verwenden. Wir können das Modell **Implementierungsmodell** nennen.

Entwurfsprozeduren müssen das Diagramm in Abb. 2.65 vom Knoten A zum Knoten F durchlaufen.

2.7.4 SpecC

Die SpecC-Sprache [173] zeigt sehr schön, wie eine Unterscheidung zwischen Kommunikation und Berechnung aussehen kann und sie zeigt die TLM-Prinzipien. SpecC modelliert Systeme als hierarchische Netzwerke von in ihrem Verhalten beschriebenen Komponenten, die über Kanäle kommunizieren. SpecC-Beschreibungen bestehen aus Verhalten (*behaviors*), Kanälen (*channels*) und Schnittstellen (*interfaces*). Verhalten enthält *Ports*, lokal instantiierte Komponenten, private Variablen und Funktionen sowie eine nach außen sichtbare `main`-Funktion. Kanäle kapseln die Kommunikation. Sie enthalten Variablen und Funktionen, die zur Definition von Kommunikationsprotokollen verwendet werden. Schnittstellen verbinden Verhalten und Kanäle. Sie deklarieren die Kommunikationsprotokolle, die innerhalb eines Kanals definiert werden. SpecC kann Hierarchien mit Hilfe von geschachteltem Verhalten modellieren.

Beispiel 2.29: Abb. 2.66 [173] zeigt eine Komponente G, die Unterkomponenten g1 und g2 enthält.

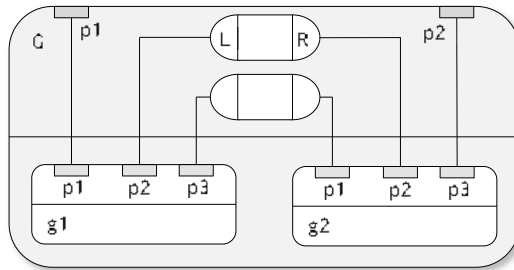


Abb. 2.66 Strukturelle Hierarchie in SpecC

Die strukturelle Hierarchie wird in dem folgenden SpecC-Modell beschrieben:

```
01: interface L { void Write(int x); };
02: interface R { int Read(void); };
03: channel H implements L,R {
04:   int Data; bool Valid;
05:   void Write(int x) { Data=x; Valid=true; }
06:   int Read(void) {
07:     while (!Valid) waitfor (10);
08:     return (Data);
09:   }
11: };
11: behavior G1(in int p1, L p2, out int p3) {
12:   void main (void) { /*...*/ p2.Write(p1); } };
13: behavior G2(in int p1, R p2, out int p3) {
14:   void main(void) { /*...*/ p3=p2.Read(); } };
15: behavior G(in int p1, out int p2) {
16:   int h1;   H h2;   G1 g1(p1, h2, h1);   G2 g2(h1, h2, p2);
17:   void main (void) {
18:     par { g1.main(); g2.main(); }
19:   }
20: };
```

Die parallele Ausführung der Teilkomponenten wird durch das Schlüsselwort **par** in Zeile 18 ausgedrückt. In Zeile 16 ist zu sehen, dass Teilkomponenten über die Ganzzahl h1 und durch den Kanal h2 kommunizieren. Das im Kanal H implementierte Schnittstellenprotokoll, das aus Lese- und Schreiboperationen besteht (siehe Zeilen 05 und 06) kann geändert werden, ohne die Verhalten G1 and G2 zu ändern. Beispielsweise kann die Kommunikation bitseriell oder parallel sein und die Wahl ändert nicht die Modelle von G1 und G2. Dieses Merkmal ist für die Wiederverwendung von Hardwarekomponenten und geistigem Eigentum (engl. *Intellectual Property* (IP)) unverzichtbar. Das vorgestellte SpecC-Modell enthält keine Zeitinformation. Es ist daher ein Spezifikationsmodell (Modelltyp A in Abb. 2.65). ∇

Der Entwurfsablauf für SpecC wurde bereits in Abb. 1.9 auf Seite 25 gezeigt. Der Weg in der Abbildung 2.65 ist A, B, D, F [83]. Auf Spezifikationsebene kann SpecC

jede Form der Kommunikation verwenden. Üblicherweise wird der Nachrichtenaustausch benutzt. Das Kommunikationsmodell in SpecC hat das Kommunikationsmodell in SystemC 2.0 beeinflusst.

SpecC basiert auf C++-Syntax. Diese Wahl wurde aus dem folgenden Grund getroffen: Es gibt den Trend, mehr und mehr Funktionalität in Software zu realisieren und dafür die Sprache C zu benutzen. Beispielsweise implementieren eingebettete Systeme Standards wie MPEG 1/2/4 oder Dekoder für Mobilfunkstandards wie GSM, UMTS oder LTE. Diese Standards sind häufig in Form von „Referenzimplementierungen" verfügbar, die aus C-Programmen bestehen, die nicht auf Effizienz optimiert sind, sondern die v.a. die notwendige Funktionalität bereit stellen. Der Nachteil von Entwurfsmethodiken, die auf speziellen HDLs (wie VHDL oder Verilog, siehe unten) basieren, ist, dass die Standards in der jeweiligen HDL neu geschrieben werden müssen. Außerdem verlangt die gemeinsame Simulation von Hardware und Software das Verbinden von Hardware- und Softwaresimulatoren. Üblicherweise führt dies zu einem Verlust an Simulationseffizienz und inkonsistenten Benutzerschnittstellen. Auch müssten die Designer mehrere Sprachen lernen.

Daher gab es das Bemühen, Hardwarestrukturen in Softwaresprachen zu modellieren. Dazu mussten einige fundamentale Probleme gelöst werden:

- Die in der Hardware übliche Nebenläufigkeit (engl. *concurrency*) muss in Software modelliert werden.
- Es gibt die Notwendigkeit, die Zeit darzustellen.
- Mehrwertige Logik, wie oben beschrieben, sollte benutzt werden können.
- Das deterministische Verhalten fast aller sinnvollen Hardwareschaltungen sollte sichergestellt sein.

Für die SpecC-Sprache wie auch für andere HDLs wurden diese Probleme gelöst.

2.7.5 SystemC

TLM-Modellierung und die Trennung zwischen Kommunikation und Berechnung sind auch in der Sprache SystemC™ verfügbar. SystemC basiert wie SpecC auf C und C++. Zur abstrakten Modellierung der Kommunikation bietet SystemC Kanäle, *Ports* und Schnittstellen, ähnlich wie SpecC. Damit wird die TLM-Modellierung erleichtert.

SystemC™ [521, 244] ist eine C++-Klassenbibliothek. Bei Verwendung von SystemC können Spezifikationen in C oder in C++ geschrieben werden, wobei an den entsprechenden Stellen Referenzen auf die Klassenbibliothek eingesetzt werden.

SystemC beinhaltet die gleichzeitige Ausführung mehrerer Prozesse. Die Ausführung von Prozessen wird über Sensitivitätslisten und Aufrufe von wait -Instruktionen gesteuert. Sensitivitätslisten können dynamisch sein, d.h. die Liste der Signale, auf deren Änderungen reagiert wird, kann sich während der Ausführung verändern.

SystemC hat auch ein Modell für die Zeit. SystemC 1.0 verwendet Gleitkommazahlen, um die Zeit darzustellen. In SystemC 2.0 wird eine ganzzahlige Darstellung

bevorzugt. SystemC 2.0 unterstützt auch physikalische Einheiten wie Pikosekunden, Nanosekunden, Mikrosekunden usw.

Die Datentypen von SystemC beinhalten alle üblichen Hardware-Typen: vierwertige Logik (`'0'`, `'1'`, `'X'` und `'Z'`) sowie Bitvektoren unterschiedlicher Länge werden unterstützt. Die Beschreibung von Anwendungen aus der digitalen Signalverarbeitung wird durch die Verfügbarkeit von Typen für Festkommazahlen vereinfacht.

Deterministisches Verhalten (siehe Seite 66) wird im Allgemeinen nicht garantiert, sondern nur durch Verwendung eines bestimmten Modellierungsstils erreicht. Mit Hilfe eines Kommandozeilenschalters kann der Simulator angewiesen werden, Prozesse in unterschiedlichen Reihenfolgen auszuführen. So kann der Benutzer überprüfen, ob die Simulationsergebnisse von der Reihenfolge der Prozessausführungen abhängen. Für Modelle mit realistischer Komplexität kann man allerdings nicht nachweisen, ob sie deterministisch sind – man kann nur zeigen, dass ein Modell nichtdeterministisch ist.

Montoreano beschreibt die TLM-Modellierung mit SystemC [402]. Er unterscheidet zwischen nur zwei Typen von TLM-Modellen:

- **Schwach zeitbehaftete Modelle**: sie werden wie folgt beschrieben [402]: „Diese Modelle besitzen eine geringe Abhängigkeit zwischen Daten und Zeitinformation, und sie können die gewünschten Daten und Zeitinformation bereitstellen, wenn Transaktionen gestartet werden. Die Erzeugung einer Antwort (z.B. auf eine Leseanforderung) setzt nicht voraus, dass die (Simulations-)zeit fortschreitet. Der Wettbewerb um Ressourcen und deren Verteilung werden in diesen Modellen üblicherweise nicht erfasst. Aufgrund der geringen Abhängigkeit und minimalen Umschalten des Kontextes können diese Modelle die schnellsten sein und sie sind besonders bei der Softwareentwicklung auf einer virtuellen Plattform nützlich."
- **Modelle mit approximierten Zeiten**: sie werden wie folgt beschrieben [402]: „Bei diesen Modellen kann die Bereitstellung einer Antwort von dem Feuern interner oder externer Ereignisse und/oder auch der fortschreitenden Zeit abhängen. Wettbewerb um Ressourcen und deren Zuordnung können mit diesem Stil leicht modelliert werden. Diese Modelle benötigen mehrere Kontextwechsel in der Simulation, um die verschiedenen Transaktionen vor ihrer Ausführung zu synchronisieren oder zu ordnen, was zu einem Verlust an Simulationsgeschwindigkeit führt."

Für den praktischen Entwurf ist es wichtig, aus den Modellen heraus mit Syntheseverfahren eine Implementierung in Hard- oder Software erzeugen zu können. Die Synthese von Hardware aus SystemC-Modellen ist verfügbar [216, 217]. Es gibt auch kommerzielle Werkzeuge zur Synthese aus SystemC heraus. Eine synthetisierbare Untermenge der Sprache ist definiert worden [8]. Für kommerzielle Werkzeuge wird erwartet, dass sie mindestens aus der synthetisierbaren Untermenge eine Implementierung erzeugen können. In einem Buch werden Methodik und Anwendungen SystemC-basierten Entwurfs vorgestellt [408]. Gegenwärtig (im Jahr 2020) ist SystemC 2.3.1 die jüngste Version von SystemC [7].

2.7.6 VHDL

Einführung

VHDL ist eine weitere Sprache, die auf der Modellierung mittels diskreter Ereignisse basiert. Im Gegensatz zu SpecC und SystemC unterstützt sie keine klare Unterscheidung zwischen Kommunikation und Berechnung, was die Wiederverwendung von Komponenten erschwert. Allerdings wird VHDL durch viele kommerzielle und akademische Werkzeuge unterstützt und ist weit verbreitet. Es ist ein (weiteres) Beispiel einer Hardwarebeschreibungssprache. Nachdem wir ein erstes Beispiel ereignisbasierter Modelle bereits auf Seite 97 vorgestellt haben, wollen wir uns jetzt näher mit VHDL beschäftigen.

VHDL verwendet **Prozesse**, um parallele Ausführung zu beschreiben. Jeder VHDL-Prozess modelliert eine Komponente einer möglicherweise nebenläufigen Hardware. Für einfache Komponenten kann ein einzelner VHDL-Prozess ausreichend sein. Um komplexere Hardwarekomponenten zu modellieren, benötigt man in der Regel mehrere Prozesse. VHDL-Prozesse kommunizieren über **Signale** miteinander. Signale entsprechen in etwa den physischen Verbindungen in der Hardware, also z.B. Leitungen oder Drähten.

Die Ursprünge von VHDL reichen bis in die achtziger Jahre des letzten Jahrhunderts zurück. Bis dahin wurden die meisten Systeme mit Hilfe graphischer HDLs beschrieben. Der meistverwendete Baustein einer solchen Darstellung war ein einzelnes Gatter. Zusätzlich zu einer graphischen Darstellung kann man auch eine textuelle HDL-Beschreibung erstellen. Die Stärke von textuellen Darstellungen liegt in der Möglichkeit, relativ leicht komplexe Berechnungen mit Variablen, Schleifen, Funktionsparametern und Rekursionen darstellen zu können. Folglich wurden graphische Darstellungen fast vollständig von textuellen HDLs ersetzt, als die digitalen Systeme immer komplexer wurden. Die textuellen HDL-Darstellungen waren ursprünglich ein Forschungsthema an Universitäten. Mermet et al. [393] geben einen Überblick über die in Europa im Laufe der achtziger Jahre entwickelten Sprachen. MIMOLA war eine dieser Sprachen und der Autor dieses Buches hat zu ihrem Entwurf und ihrer Anwendung beigetragen [380, 374]. Textuelle Sprachen wurden beliebt, als VHDL und der Konkurrent Verilog (siehe Seite 120) eingeführt wurden.

VHDL wurde im Rahmen des VHSIC-Programms des US-amerikanischen Verteidigungsministeriums entwickelt. VHSIC steht für *Very High Speed Integrated Circuits*. Ursprünglich wurde der Entwurf von VHDL (*VHSIC Hardware Description Language*) von drei Firmen durchgeführt: IBM, Intermetrics und Texas Instruments. Eine erste Version von VHDL wurde 1984 veröffentlicht. Später wurde VHDL dann von der IEEE unter dem Namen IEEE 1076 standardisiert. Die erste IEEE-Version von VHDL wurde im Jahre 1987 verabschiedet und in den Jahren 1993, 2000, 2002 und 2008 aktualisiert [238, 240, 241, 242, 243]. VHDL-AMS erlaubt zusätzlich die Modellierung analoger und gemischt analog/digitaler (engl. *mixed-signal*) Systeme durch die Erweiterung der Sprache um Differentialgleichungen [246]. Der Entwurf von VHDL nahm Ada (siehe Seite 123) als Ausgangspunkt, da beide Sprachen für das US-amerikanische Verteidigungsministerium entworfen wurden. Weil Ada auf

PASCAL basiert, hat VHDL einige syntaktische Eigenheiten von PASCAL übernommen. Allerdings ist die Syntax von VHDL im Allgemeinen sehr viel komplexer, sodass man aufpassen muss, um sich nicht verwirren zu lassen. In diesem Buch werden nur einige Konzepte von VHDL vorgestellt, die auch in anderen Sprachen vorkommen oder aber nützlich sind. Eine vollständige Beschreibung des VHDL-Sprachstandards ist nicht Bestandteil dieses Buches. Ihn kann man von der IEEE erhalten [243].

Entities **und** ***Architectures***

Wie alle anderen HDLs auch, beinhaltet VHDL die für die Modellierung nebenläufiger Operationen benötigte Unterstützung. In VHDL heißen die modellierten Hardwarekomponenten ***Design Entity*** oder **VHDL *Entity***. Innerhalb von *Entities* werden VHDL-**Prozesse** verwendet, um Nebenläufigkeit darzustellen. Die VHDL-Sprachbeschreibung definiert, dass *Entities* aus zwei Typen von Bestandteilen bestehen: einer ***Entity*-Deklaration** und einer (oder mehrerer) **Architekturen** (engl. ***Architectures*** (siehe Abb. 2.67)).

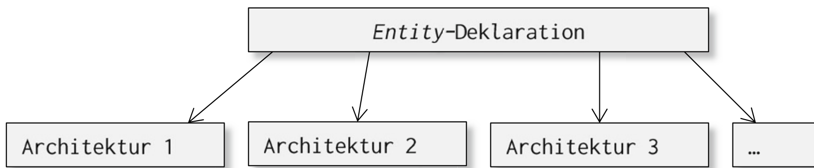


Abb. 2.67 *Entity* bestehend aus einer *Entity*-Deklaration und Architekturen

Für jede *Entity* wird im Standardfall die zuletzt analysierte Architektur verwendet. Es kann aber auch angegeben werden, dass eine andere Architektur verwendet werden soll.

Beispiel 2.30: Als Beispiel betrachten wir einen Volladdierer. Volladdierer haben drei Eingänge und zwei Ausgänge (siehe Abb. 2.68).

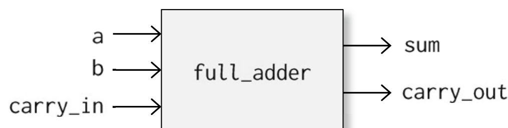


Abb. 2.68 Ein Volladdierer und seine Schnittstellensignale

Eine *Entity*-Deklaration, die Abb. 2.68 entspricht, könnte etwa wie folgt aussehen:

```
entity full_adder is                                  -- Entity-Deklaration
  port (a, b, carry_in: in BIT;                                 -- Eingänge
        sum, carry_out: out BIT);                               -- Ausgänge
end full_adder;
```

Kommentare werden durch einen doppelten Bindestrich (`--`) eingeleitet und enden am Zeilenende. ∇

Architekturen bestehen aus Architekturköpfen und Architekturrümpfen. Man kann verschiedene Stile von Rümpfen unterscheiden, nämlich strukturelle Rümpfe und Verhaltensrümpfe. Wir zeigen die Unterschiede zwischen diesen beiden Modellierungsarten am Beispiel des Volladdierers. Verhaltensrümpfe enthalten nur die Informationen, die benötigt werden, um aus Eingabesignalen und internem Zustand (wenn es einen gibt) die Ausgabesignale und den neuen internen Zustand zu berechnen. Dies beinhaltet auch das zeitliche Verhalten.

Beispiel 2.31: Das folgende Beispiel zeigt einen Verhaltensrumpf:

```
architecture behavior of full_adder is                        -- Architektur
begin
  sum       <= (a xor b) xor carry_in after 10 ns;
  carry_out <= (a and b) or (a and carry_in) or
               (b and carry_in) after 10 ns;
end behavior;
```

VHDL-basierte Simulatoren können die Ausgabesignale graphisch als Signalverläufe darstellen. Diese ergeben sich, wenn Werte an die Eingabeports des Volladdierers gelegt werden.

Im Gegensatz zu Verhaltensrümpfen beschreiben strukturelle Architekturrümpfe die Art und Weise, in der *Entities* aus einfacheren *Entities* zusammengebaut sind. Beispielsweise kann der Volladdierer als *Entity* modelliert werden, die aus drei Komponenten besteht (siehe Abb. 2.69). Diese Komponenten heißen `i1`, `i2` und `i3` und sind vom Typ `half_adder` oder `or_gate`.

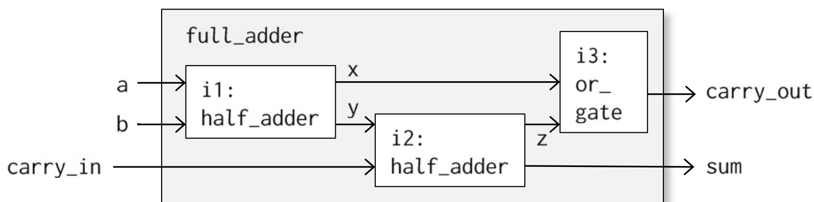


Abb. 2.69 Schematische Darstellung des Strukturrumpfes des Volladdierers

In der 1987er-Version von VHDL mussten diese Komponenten in einer sogenannten *Component*-Deklaration vorgestellt werden. Diese Deklaration ist der *Forward*-Deklaration anderer Programmiersprachen sehr ähnlich und dient auch genau dem

gleichen Ziel: sie stellt genügend Informationen über eine Komponente zur Verfügung, die zu dem Zeitpunkt evtl. noch nicht in der VHDL-Datenbank gespeichert ist (was bei sogenannten *Top-Down-Designs* vorkommen kann). Ab der 1993er-Version von VHDL sind solche Deklarationen nicht mehr notwendig, wenn die benötigten Komponenten bereits in der Komponentendatenbank abgelegt wurden.

Verbindungen zwischen lokalen Komponenten und den *Ports* der *Entity* werden mit Hilfe von ***Port Maps*** beschrieben. Der folgende VHDL-Code beschreibt den Strukturrumpf aus Abb. 2.69:

```
architecture structure of full_adder is                    -- Architektur-Kopf
  component half_adder
      port (in1, in2: in BIT; carry: out BIT; sum: out BIT);
  end component;
  component or_gate
      port (in1, in2: in BIT; o: out BIT);
  end component;
  signal x, y, z: BIT;                                      -- lokale Signale
 begin                                                            -- port map
   i1: half_adder                             -- Instanz i1 vom Typ half_adder
       port map (a, b, x, y);                   -- Verbindungen zwischen Ports
   i2: half_adder port map (y, carry_in, z, sum);                -- Instanz i2
   i3: or_gate    port map (x, z, carry_out);
 end structure;
```

∇

Zuweisungen

Beispiel 2.31 enthält mehrere Zuweisungen. Zuweisungen sind Spezialfälle von Anweisungen. In VHDL gibt es zwei verschiedene Arten von Zuweisungen:

- **Variablenzuweisungen:** Die Syntax für Variablenzuweisungen lautet

  Variable := *Ausdruck*

  Wenn der Kontrollfluss eine solche Zuweisung erreicht, wird der entsprechende Ausdruck berechnet und der Variablen zugewiesen. Diese Zuweisungen verhalten sich wie Zuweisungen in normalen Programmiersprachen.
- **Signalzuweisungen**: Signalzuweisungen, die bereits auf den Seiten 97 und 111 erwähnt wurden, werden nebenläufig ausgeführt. Signale und Signalzuweisungen wurden eingeführt, um elektrische Signale realer Hardwaresysteme modellieren zu können. Signale ordnen Werte bestimmten Zeitpunkten zu. Diese Abbildung wird in VHDL durch **Signalverläufe** dargestellt, die aus Signalzuweisungen berechnet werden. Die Syntax für Signalzuweisungen lautet

```
signal <= expression;
signal <= transport expression after delay;
signal <= expression after delay;
signal <= reject time inertial expression after delay;
```

Wenn der Kontrollfluss eine solche Zuweisung erreicht, wird der Ausdruck berechnet und dazu verwendet, zukünftige Werte des Signalverlaufes vorauszusagen. In VHDL wird Signalen ein sogenannter Signaltreiber zugeordnet. Wenn es mehrere Beiträge zu ein- und demselben Signal gibt (z.B. wegen einer leitenden Verbindung), dann wird aus diesen Beiträgen mit Hilfe einer **Auflösungsfunktion** (engl. *resolution function*) ein resultierender Wert berechnet. Auf diese Weise wird die *sup*-Funktion, die im Kontext der CSA-Theorie vorgestellt wurde, berechnet, wenn es entsprechende Verbindungen gibt.
Um zukünftige Werte zu berechnen, **verwenden Simulatoren eine Warteschlange von Ereignissen, die nach dem aktuellen Zeitpunkt eintreten sollen**. Diese Warteschlange ist aufsteigend nach der Zeit sortiert, zu der diese zukünftigen Ereignisse (z.B. die Aktualisierung eines Signals) stattfinden sollen. Die Ausführung einer Signalzuweisung führt dazu, dass Einträge in dieser Warteschlange erzeugt werden. Jeder dieser Einträge enthält den zugehörigen Ausführungszeitpunkt, das betroffene Signal und den zuzuweisenden Wert. Bei Signalzuweisungen, die keinerlei `after`-Bedingung beinhalten (erste mögliche Syntaxform der obigen Liste), verwendet der Eintrag die aktuelle Simulationszeit als Zeitpunkt, zu dem diese Zuweisung stattfinden soll. In diesem Fall findet die entsprechende Änderung nach einer unendlich kleinen Zeitspanne, δ-Verzögerung genannt (siehe unten), statt. Dies ermöglicht es, die Signale zu aktualisieren, ohne die makroskopische Sicht der Zeit zu verändern.
Bei Signalzuweisungen, die einen `transport`-Präfix verwenden (zweite Form der Liste) wird die Signalzuweisung um die angegebene Dauer verzögert. Diese Form der Zuweisung folgt dem sogenannten **Transportverzögerungsmodell**. Dieses Modell simuliert das Verhalten einfacher Leitungen: diese verzögern (in erster Näherung) Signale. Auch kurze Impulse setzen sich entlang von Leitungen fort. Das Transportverzögerungsmodell kann auch für logische Schaltungen verwendet werden, auch wenn sein Hauptzweck die Modellierung von Leitungen ist.

Beispiel 2.32: Modell eines ODER-Gatters mit Transportverzögerung:

```
c <= transport a or b after 10 ns;
```

Ein solches Modell würde auch kurze Impulse weiterleiten (siehe Abb. 2.70).

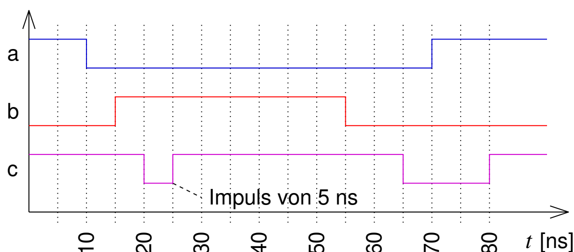


Abb. 2.70 Gatter mit Transportverzögerung

Das Ausgangssignal c enthält einen kurzen Impuls von 5 ns, der bei einer Trägheitsverzögerung (siehe unten) unterdrückt werden würde. ∇

Signalzuweisungen mit Angabe der Transportverzögerung löschen alle Einträge in der Warteschlange, die dem Zeitpunkt der berechneten Aktualisierung und den darauf folgenden Zeitpunkten entsprechen. Wenn wir also zuerst eine Zuweisung mit recht großer Verzögerung und danach eine mit kleinerer Verzögerung ausführen, dann wird der Eintrag, der durch die erste Zuweisung entstanden ist, gelöscht.
Bei Signalzuweisungen mit einer **after**-Bedingung, die keinen **transport**-Präfix enthalten, wird eine **Trägheitsverzögerung** angenommen. Dieses Verzögerungsmodell entspricht der Tatsache, dass auch echte Schaltungen über eine gewisse „Trägheit" verfügen. Das bedeutet, dass kurze Impulsspitzen unterdrückt werden. Bei der dritten angegebenen Zuweisungsform werden alle Signaländerungen unterdrückt, die kürzer als die angegebene Verzögerungsdauer anliegen. Die vierte Form entfernt dagegen alle Signaländerungen, die kürzer als die angegebene Zeit *time* anliegen, aus dem vorhergesagten Signalverlauf. Die etwas subtilen Regeln für das Löschen von Signaländerungen in der Liste künftiger Ereignisse sollen hier nicht wiedergegeben werden.

Beispiel 2.33: Wir modellieren ein einfaches ODER-Gatter mit Trägheitsverzögerung:

```
c <= a or b after 10 ns;
```

Im Modell der Trägheitsverzögerung werden kurze Signalspitzen unterdrückt (siehe Abb. 2.71).

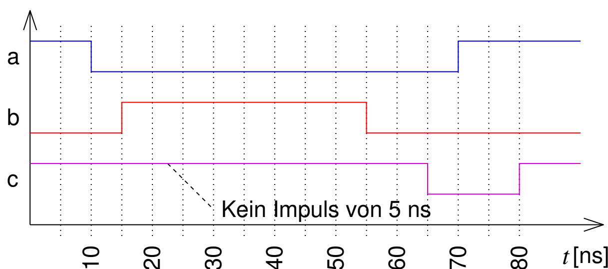


Abb. 2.71 Gatter mit Trägheitsverzögerung

Für das Ausgabesignal c gibt es keinen kurzen Impuls von 5 ns, aber der Impuls mit einer Länge von 15 ns erreicht den Ausgang. ∇

VHDL-Prozesse

Zuweisungen sind nur eine Kurzform von VHDL-Prozessen. Mehr Kontrolle über die Auswertung von Signalen hat man mit expliziten Prozessen. Die allgemeine Syntax für Prozesse ist wie folgt:

```
 Marke:                                              -- optional
process
  Deklarationen                                      -- optional
 begin
   Anweisungen                                       -- optional
 end process;
```

Zusätzlich zu Zuweisungen können Prozesse **wait**-Anweisungen enthalten. Diese Anweisungen können verwendet werden, um einen Prozess zu suspendieren. Es gibt die folgenden verschiedenen **wait**-Anweisungen:

```
wait on  Signalliste;        -- suspendiere bis eines der Signale sich ändert
wait until  Bedingung;          -- suspendiere bis die Bedingung erfüllt ist
wait for  Dauer;                      -- suspendiere für eine bestimmte Zeit
wait;                                               -- suspendiere unbegrenzt
```

Alternativ zur Verwendung expliziter **wait**-Anweisungen kann eine Liste von Signalen im Prozesskopf angegeben werden. Diese bewirken, dass der Prozess jedes Mal aktiviert wird, wenn mindestens eines der Signale in der Liste seinen Wert ändert.

Beispiel 2.34: Das folgende Modell eines UND-Gatters führt seine Anweisungen einmal aus und startet jedes Mal von vorne, wenn einer der Eingänge seinen Wert ändert:

```
process(x,y) begin
  prod <= x and y;
end process;
```

Dieses Modell entspricht

```
process begin
  prod <= x and y;
  wait on x,y;
end process;
```

mit einem expliziten **wait**-Statement am Ende. ∇

Der VHDL-Simulationszyklus

Im Standard-Dokument [241] zu VHDL wird die Ausführung eines VHDL-Prozesses wie folgt beschrieben: „Die Ausführung eines Modells besteht aus einer

Initialisierungsphase, danach folgt die **wiederholte Ausführung von Prozessen**, die in der Beschreibung des Modells enthalten sind. Jede solche Wiederholung heißt **Simulationszyklus**. In jedem Zyklus werden die Werte aller Signale in der Prozess-Beschreibung berechnet. Falls aufgrund dieser Ergebnisse ein Ereignis auf einem bestimmten Signal stattfindet, so werden Prozesse, die auf dieses Signal sensitiv sind, aufgeweckt und innerhalb desselben Simulationszyklus ausgeführt."

Die Initialisierungsphase berücksichtigt die angegebenen Initialisierungswerte für Signale und Variablen und führt jeden Prozess einmal aus. Sie wird im Standard wie folgt beschrieben[20]:

- „Der treibende Wert und der effektive Wert jedes explizit deklarierten Signals werden berechnet und der aktuelle Wert des Signals wird auf den effektiven Wert gesetzt. Es wird angenommen, dass das Signal diesen Wert bereits für eine unendlich lange Zeit vor dem Start der Simulation hatte. ...
- Jeder ... Prozess des Modells wird ausgeführt, bis er suspendiert wird. ...
- Die Zeit des nächsten Simulationszyklus T_n (der in diesem Fall der erste Simulationszyklus ist) wird anhand der Regeln von Schritt e des Simulationszyklus (s.u.) berechnet."

Jeder Simulationszyklus beginnt damit, dass die aktuelle Zeit auf denjenigen nächsten Zeitpunkt gesetzt wird, an dem Änderungen berücksichtigt werden müssen. Diese Zeit T_n wurde entweder während der Initialisierungsphase oder während der letzten Ausführung eines Simulationszyklus berechnet. Die Simulation wird beendet, wenn die aktuelle Zeit den Maximalwert $TIME'HIGH$ erreicht. Im Originaldokument wird der Simulationszyklus wie folgt beschrieben: „Ein Simulationszyklus besteht aus den folgenden Schritten:

a) Die aktuelle Zeit T_c wird auf T_n gesetzt. Die Simulation ist beendet, wenn $T = TIME'HIGH$ ist und wenn es keine aktuellen Treiber oder aufgeweckte Prozesse zur Zeit T_n gibt.
b) Jedes aktive explizite Signal im Modell wird aktualisiert. (Daraus können sich Ereignisse ergeben)" ...
Im vorhergehenden Zyklus wurden neue zukünftige Werte für einige der Signale berechnet. Wenn T_c dem Zeitpunkt entspricht, an dem diese Werte gültig werden, werden sie jetzt zugewiesen. Man beachte, dass neue Werte für Signale niemals sofort innerhalb des Simulationszyklus zugewiesen werden. Sie werden frühestens vor dem nächsten Simulationszyklus zugewiesen. Signale, die ihren Wert verändern, verursachen sogenannte Ereignisse, die ihrerseits wiederum die Ausführung von Prozessen anstoßen können, die auf diese Signale sensitiv sind.
c) „Jeder Prozess P, der gerade auf ein Signal S sensitiv ist, wird aufgeweckt, wenn in diesem Simulationszyklus ein Ereignis auf dem Signal S stattgefunden hat."
d) „Jeder ... Prozess, der im aktuellen Simulationszyklus aufgeweckt wurde, wird solange ausgeführt, bis er suspendiert wird."

[20] Implizit deklarierte Signale und sogenannte „*postponed processes*“, die in der 1997er-Version von VHDL eingeführt wurden, werden hier nicht betrachtet.

e) „Die Zeit T_n des nächsten Simulationszyklus wird bestimmt als frühester Zeitpunkt, an dem

1. TIME'HIGH erreicht wird (das Ende der Simulationszeit),
2. ein Signaltreiber aktiv wird (der nächste Zeitpunkt, an dem ein Treiber einen neuen Wert angibt) oder
3. ein Prozess aufgeweckt wird" (diese Zeit wird durch **`wait for`**-Anweisungen bestimmt).

„Wenn $T_n = T_c$, dann ist der nächste Simulationszyklus (wenn es einen gibt) ein Delta-Zyklus."

Die iterative Form von Simulationszyklen wird in Abb. 2.72 gezeigt.

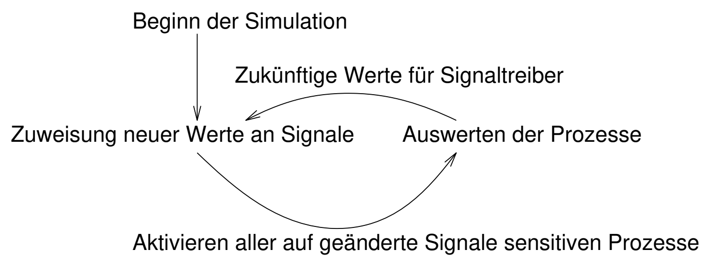


Abb. 2.72 VHDL Simulationszyklen

Delta (δ)-Simulationszyklen sind eine Quelle vieler Diskussionen. Ihr Sinn ist es, eine infinitesimal kleine Verzögerung zu erzeugen, selbst wenn der Benutzer keine Verzögerung angegeben hat.

Beispiel 2.35: Wir kommen auf unser Flipflop-Beispiel zurück. Abb. 2.73 zeigt das Flipflop mit Standard-Schaltkreissymbolen.

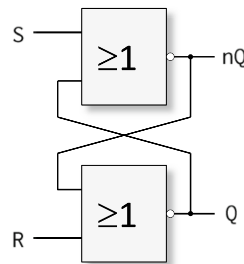


Abb. 2.73 RS-Flipflop

Das Flipflop wird in VHDL wie folgt modelliert:

```
entity RS_Flipflop is
 port (R: in    BIT;                              -- Rücksetz-Eingang
       S: in    BIT;                                  -- Setz-Eingang
       Q: inout BIT;                                      -- Ausgang
       nQ:inout BIT;);                        -- invertierter Ausgang
end RS_Flipflop;
architecture one of RS_Flipflop is
 begin
  process: (R,S,Q,nQ)
   begin
      Q <= R nor nQ; nQ <= S nor Q;
    end process;
 end one;
```

Die *Ports* `Q` und `nQ` müssen vom Typ **`inout`** sein, da sie auch intern gelesen werden, was mit *Ports* vom Typ **`out`** nicht möglich ist. Tabelle 2.5 zeigt die Simulationszeitpunkte, an denen die Signale dieses Modells aktualisiert werden.

Tabelle 2.5 δ-Zyklen für ein RS-Flipflop

	<0ns	0ns	0ns+δ	0ns+$2*\delta$	0ns+$3*\delta$
R	'0'	'1'	'1'	'1'	'1'
S	'0'	'0'	'0'	'0'	'0'
Q	'1'	'1'	'0'	'0'	'0'
nQ	'0'	'0'	'0'	'1'	'1'

In jedem Zyklus werden Aktualisierungen durch eines der Gatter geleitet. Die Simulation endet nach drei δ-Zyklen. Dabei ändert der letzte Zyklus nichts mehr, da `Q` bereits den Wert `'0'` hat. ∇

δ-Zyklen entsprechen einer infinitesimal kleinen Zeiteinheit, die in Wirklichkeit immer vorhanden ist. δ-Zyklen stellen sicher, dass die Simulation die Kausalität erhält.

Die Ergebnisse hängen nicht von der Reihenfolge ab, in der Teile eines Modells während der Simulation ausgewertet werden. Dies wird durch die Trennung der Berechnung neuer Werte für Signale und deren eigentlicher Zuweisung erreicht. In einem Modell, das die folgenden Zeilen enthält,

```
a <= b;
b <= a;
```

werden die Signale `a` und `b` stets vertauscht. Wenn die Zuweisungen sofort ausgeführt würden, so würde das Ergebnis von der Reihenfolge abhängen, in der die beiden Wertzuweisungen ausgeführt werden (siehe dazu auch Seite 63). **VHDL-Modelle sind daher deterministisch**. Dies erwarten wir auch von der Simulation einer echten Schaltung mit festem Verhalten.

Es kann unbeschränkt viele δ-Zyklen geben, bevor die Zeit T_c fortschreitet. Die Möglichkeit unbegrenzter Schleifen kann verwirrend sein. Eine Option zur Vermeidung dieser Schleifen wäre es, Verzögerungen von Null, wie wir sie in unserem Modell benutzt haben, zu vermeiden.

Die Weiterleitung von Werten mittels Signalen ermöglicht auch eine einfache Implementierung des *Observer*-Musters (siehe Seite 37). Im Vergleich zu SDL ist bei VHDL die Anzahl der Beobachter variabel und hängt von der Anzahl der Prozesse ab, die auf die Änderung eines Signals warten.

Wie sieht nun das Kommunikationsmodell hinter VHDL aus? Die Beschreibung der Semantik von VHDL basiert grundlegend auf einer **einzelnen, zentralisierten** Warteschlange für zukünftige Ereignisse, die alle zukünftigen Signalwerte speichert. Der Sinn dieser Warteschlange ist **nicht**, asynchronen Nachrichtenaustausch zu implementieren. Vielmehr soll der Simulationskern Einträge sequentiell aus der Warteschlange auslesen. Versuche, VHDL-Simulation verteilt auszuführen, leiden i.d.R. an mangelnder Performanz. Alle modellierten Komponenten können auf Werte von Signalen und Variablen zugreifen, die sich in ihrem Bereich befinden, ohne nachrichtenbasierte Kommunikation zu verwenden. Daher neigen wir dazu, die Implementierung von Kommunikation in VHDL als *shared memory*-basiert anzusehen. Auf Basis des VHDL-Simulators ließe sich aber auch FIFO-basierter Nachrichtenaustausch in VHDL implementieren.

IEEE 1164

In VHDL gibt es keine vordefinierte Menge von Signalwerten, bis auf die Basisunterstützung für zweiwertige Logik. Stattdessen kann man die zu verwendende Wertemenge in VHDL selbst definieren und unterschiedliche VHDL-Modelle können unterschiedliche Signal-Wertemengen verwenden.

Würde diese Fähigkeit von VHDL allerdings in dieser Weise eingesetzt, wären die Übertragbarkeit und Austauschbarkeit von VHDL-Modellen stark eingeschränkt. Um das Austauschen von Modellen zu vereinfachen, wurde eine Standard-Wertemenge definiert und von der IEEE standardisiert. Dieser Standard heißt IEEE 1164 und wird von vielen Systemen eingesetzt. IEEE 1164 hat neun Werte: `{'0', '1','L','H','X','W','Z','U','-'}`. Die ersten sieben Werte entsprechen den sieben Signalwerten von Abschnitt 2.7.2. `'U'` steht für einen nicht-initialisierten Wert. Er wird von Simulatoren für Signale verwendet, die noch nicht explizit definiert wurden.

`'-'` steht für eine beliebige Eingabe (engl. *input don't care*). Dieser Wert muss näher erläutert werden. Hardwarebeschreibungsspachen werden häufig zur Beschreibung Boolescher Funktionen verwendet. Mit Hilfe der VHDL-**`select`**-Anweisung ist das bequem möglich. Die **`select`**-Anweisung entspricht den **`switch`**- und **`case`**-Anweisungen anderer Programmiersprachen. Die Bedeutung des VHDL-**`select`** ist jedoch anders als die Bedeutung der **`select`**-Anweisung in Ada (siehe Seite 124).

Beispiel 2.36: Angenommen, wir wollen die Boolesche Funktion

$$f(a,b,c) = a\overline{b} + bc$$

beschreiben. Weiterhin nehmen wir an, dass f für den Fall $a = b = c =$ '0' undefiniert sein soll. Eine sehr bequeme Methode, dies in VHDL zu beschreiben, wäre der folgende Code-Ausschnitt:

```
f <= select a&b&c                          -- & ist die Konkatenation
     '1' when "10-"                                     -- erster  Term
     '1' when "-11"                                     -- zweiter Term
     'X' when "000"
```

So könnten gegebene Funktionen leicht in VHDL umgesetzt werden. Leider drückt die **select**-Anweisung hier etwas vollkommen Anderes aus: Da IEEE 1164 nur eine von vielen möglichen Wertemengen darstellt, kann VHDL kein Wissen über die „Bedeutung" von '-' haben. Wenn VHDL-Compiler Konstrukte wie die oben gezeigte **select**-Anweisung übersetzen, prüfen sie, ob der zu betrachtende Ausdruck (im Beispiel a&b&c) den Werten in den **when**-Teilen entspricht. Genauer gesagt überprüfen sie, ob z.B. a&b&c gleich "10-" ist. In diesem Zusammenhang verhält sich '-' wie jeder andere Logikwert auch: VHDL-Systeme überprüfen, ob c den Wert '-' hat. Da der Wert '-' niemals irgendeiner der Variablen zugewiesen wird, werden diese Bedingungen nie erfüllt. ∇

Daher ist der Wert '-' nur von begrenztem Nutzen. Die Nichtverfügbarkeit eines praktischen Wertes für eine beliebige Eingabe ist der Preis, den man für die Flexibilität bei der Definition von Wertemengen in der Programmiersprache VHDL bezahlen muss[21].

Eine positive Eigenschaft der beschriebenen allgemeinen Betrachtungen von Seite 98 bis Seite 103 ist die Tatsache, dass man daraus direkt Schlussfolgerungen bezüglich der Ausdrucks- und Modellierungsfähigkeit des Standards IEEE 1164 ziehen kann. Der IEEE-Standard basiert auf der 7-elementigen Signalwertmenge von Seite 100 und kann daher Schaltungen mit Verarmungstransistoren modellieren. Eine Modellierung von vorgeladenen Busleitungen ist damit allerdings nicht möglich[22].

2.7.7 *Verilog and SystemVerilog*

Verilog [539] ist eine weitere Hardware-Beschreibungssprache. Ursprünglich war es eine proprietäre Sprache, wurde aber später als IEEE Standard 1364 vereinheitlicht. Die Versionen heißen IEEE Standard 1364-1995 (Verilog Version 1.0) und IEEE Standard 1364-2001 (Verilog 2.0). Einige Eigenschaften von Verilog sind VHDL sehr ähnlich. Genau wie in VHDL werden Entwürfe als miteinander verbundene *Design-Entities* beschrieben, diese können wiederum verhaltensorientiert

[21] Dieses Problem wurde in VHDL 2006 beseitigt [342].

[22] Abgesehen von dem Fall, in dem man keine Verarmungstransistoren einsetzt. In diesem Fall könnte man die schwachen Werte als auf den Leitungen gespeicherte Ladungen interpretieren. Dies ist allerdings nicht sehr praktisch, da Verarmungstransistoren heute in den meisten Systemen eingesetzt werden.

beschrieben werden. Prozesse werden zur Modellierung der Nebenläufigkeit von Hardwarekomponenten verwendet. Genau wie in VHDL gibt es Bitvektoren und Zeiteinheiten. In einigen Bereichen ist Verilog allerdings weniger flexibel und konzentriert sich mehr auf bequem verwendbare Grundfunktionen. Standard-Verilog erlaubt es beispielsweise nicht, Aufzählungstypen so flexibel zu definieren wie dies im IEEE 1164-Standard der Fall ist. Die Unterstützung für vierwertige Logik ist direkt in die Verilog-Sprache eingebaut und der Standard IEEE 1364 stellt auch eine mehrwertige Logik mit acht verschiedenen Signalstärken zur Verfügung. Mehrwertige Logik ist in Verilog enger an die Sprache gebunden als in VHDL. Das Logik-System von Verilog bietet auch mehr Funktionen für die Beschreibung von Schaltungen auf der Transistor-Ebene. Trotzdem ist VHDL im Allgemeinen flexibler. So kann man in VHDL etwa Hardware-Komponenten in Schleifen instantiieren, wodurch man strukturelle Beschreibungen wie z.B. einen n-Bit-Addierer spezifizieren kann, ohne wirklich n Addierer und ihre Verbindungen explizit angeben zu müssen.

Die Anzahl der Benutzer von Verilog ist ähnlich hoch wie die von VHDL. Während VHDL in Europa beliebter ist, wird Verilog in den Vereinigten Staaten bevorzugt.

Die Versionen 3.0 und 3.1 von Verilog sind auch als SystemVerilog bekannt. Sie enthalten zahlreiche Erweiterungen zu Verilog 2.0, u.a. [245, 517]:

- zusätzliche Sprachelemente, um Verhalten zu modellieren,
- C-Datentypen wie `int` und Methoden zur Typdefinition wie `typedef` und `struct`,
- Definition der Schnittstellen von Hardwarekomponenten als eigenständige *Entities*,
- standardisierte Methoden zum Aufruf von C/C++-Funktionen und, zu einem gewissen Grad, zum Aufruf eingebauter Verilog-Funktionen aus C,
- deutlich verbesserte Unterstützung zur Beschreibung der Umgebung (der *Testbench*) für die entwickelte Hardware (die als *Circuit Under Design* (CUD) bezeichnet wird) und zur Verwendung der *Testbench*, um die CUD durch Simulation zu verifizieren,
- Klassen (wie man sie von der objektorientierten Programmierung her kennt), die in Testbenches verwendet werden können,
- dynamische Erzeugung von Prozessen,
- standardisierte Interprozesskommunikation und -synchronisation, auch mit Semaphoren,
- automatische Anforderung und Freigabe von Speicher,
- Sprachelemente, welche die Schnittstellen zu formalen Verifikationsmethoden vereinheitlichen (siehe Seite 261).

Aufgrund der Schnittstellen zu C und C++ ist es auch möglich, Verilog mit SystemC zu verbinden. Die Simulations- und formalen Verifikationsmöglichkeiten machen die Sprache beliebt.

2.8 Von-Neumann-Sprachen

Die sequenzielle Ausführung ist die gemeinsame Eigenschaft aller von-Neumann-Sprachen. Solche Sprachen erlauben zudem meist einen fast unbeschränkten Zugriff auf globale Variablen. Auch wenn der modellbasierte Entwurf mit kommunizierenden endlichen Automaten und Berechnungsgraphen für eingebettete Systeme deutlich besser geeignet ist, sind von-Neumann-Sprachen immer noch weit verbreitet. Daher können wir diese Sprachen nicht ignorieren. Die Unterscheidung zwischen KPNs und mit geeigneten Beschränkungen versehenen von-Neumann-Sprachen wird allerdings immer unschärfer. Bei KPNs wird auch von der sequenziellen Ausführung von Code in jedem der Knoten ausgegangen. Wir behalten die Unterscheidung zwischen KPNs und von-Neumann-Sprachen aber weiterhin bei, da der Schwerpunkt von KPNs auf der Modellierung der Kommunikation liegt und Details der Ausführung in den einzelnen Knoten unwichtig sind. Die ersten beiden in diesem Abschnitt behandelten Sprachen verfügen über Kommunikationsmechanismen. Der Schwerpunkt der restlichen Sprachen liegt bei Berechnungen, die Kommunikation kann durch den Einsatz verschiedener Bibliotheken realisiert werden.

2.8.1 CSP

CSP [218] (engl. *Communicating Sequential Processes*, kommunizierende sequentielle Prozesse) ist eine der ersten Sprachen, die Mechanismen für Interprozesskommunikation enthalten. Die Kommunikation basiert auf Kanälen.

Beispiel 2.37: Wir betrachten die Ein- und Ausgabe für den Kanal `c` :

```
process A                              process B
.....                                  ......
var a ..                               var b ...
 a := 3;                                ...
 c!a; -- Ausgabe auf Kanal c            c?b; -- Eingabe von Kanal c
end;                                   end;
```

Beide Prozesse warten, bis der jeweils andere Prozess bei der Eingabe- bzw. Ausgabeanweisung angekommen ist. Diese Form der Kommunikation bezeichnet man als ***Rendez-Vous*-Konzept** oder als **blockierende Kommunikation**. ∇

CSP ist deterministisch, da es, wie Kahn-Prozessnetzwerke auch, auf Eingaben von einem bestimmten Kanal wartet.

CSP war die Grundlage für die Sprache OCCAM, die als Programmiersprache für **Transputer** [436] eingeführt wurde. Die Idee, einen Prozessor mit Kommunikationskanälen zu entwerfen, wurde beim XS1-Prozessor wiederverwendet [603].

2.8.2 Ada

In den achtziger Jahren des zwanzigsten Jahrhunderts bemerkte das US-amerikanische Verteidigungsministerium, dass sowohl die Verlässlichkeit als auch die Wartbarkeit der Software in den militärischen Ausrüstungsgegenständen zu großen Problemen werden könnten, wenn nicht strenge Regelungen eingeführt würden. Es wurde entschieden, dass alle Software in der gleichen Echtzeitsprache geschrieben werden sollte. Die Anforderungen an solch eine Sprache wurden formuliert. Da keine existierende Sprache alle Anforderungen erfüllte, wurde eine neue Sprache entworfen. Die letztlich angenommene Sprache basiert auf PASCAL und heißt Ada (nach Ada Lovelace, die als die erste Programmiererin gilt). Ada'95 [288, 80] ist eine objektorientierte Erweiterung des Original-Sprachstandards. Allerdings wurden inzwischen die Vorgaben zur Nutzung von Ada wieder gelockert.

Ada hat die interessante Eigenschaft, dass man verschachtelte Prozesse (die in Ada Tasks heißen) deklarieren kann. Eine Task wird gestartet, wenn der Kontrollfluss in den Bereich verzweigt, in dem die Task deklariert wurde.

Beispiel 2.38: Das folgende Beispiel ist aus Burns et al. [79] entnommen. Darin enthält die Prozedur `example1` zwei lokale Tasks `a` und `b`.

```
procedure example1 is
   task a;
   task b;
   task body a is
     -- lokale Deklarationen für a
     begin
       -- Anweisungen für a
     end a;
   task body b is
     -- lokale Deklarationen für b
     begin
       -- Anweisungen für b
     end b;
 begin
  -- Rumpf der Prozedur example1
 end;
```

Tasks `a` und `b` werden gestartet, wenn der Kontrollfluss in den Rumpf verzweigt, d.h. sie werden vor der 1. Anweisung im Code von `example1` starten. ∇

Das Kommunikationskonzept von Ada ist ein weiteres wichtiges Konzept. Es basiert ähnlich wie das Konzept bei CSP auf dem *Rendez-Vous*-Paradigma. Wenn zwei Tasks Informationen austauschen wollen, muss diejenige Task, die den „Treffpunkt" zuerst erreicht, warten, bis auch der Kommunikationspartner den entsprechenden Punkt im Kontrollfluss erreicht hat. In der Ada-Syntax werden Prozeduren verwendet, um Kommunikation zu beschreiben. Prozeduren, die von anderen Tasks aufgerufen werden können, müssen mit dem Schlüsselwort **`entry`** gekennzeichnet werden.

Beispiel 2.39: Das folgende Beispiel ist ebenfalls aus Burns et al. [79] entnommen:

```
task screen_out is
 entry call (val: character; x, y: integer);
end screen_out;
```

Task `screen_out` enthält eine Prozedur `call`, die von anderen Prozessen aufgerufen werden kann. Eine andere Task kann diese Prozedur aufrufen, indem sie ihr den Namen der Task voranstellt:

```
 screen_out.call('Z',10,20);
```

Die aufrufende Task muss warten, bis die aufgerufene Task einen Punkt im Kontrollfluss erreicht hat, an dem sie Aufrufe von anderen Tasks annimmt. Dieser Punkt wird mit dem Schlüsselwort **accept** markiert:

```
task body screen_out is
 ...
 begin
    ...
    accept call (val: character; x, y: integer) do
    ...
    end call;
  ...
 end screen_out;
```

Offensichtlich sollte eine Task `screen_out` auch auf mehrere Aufrufe gleichzeitig warten können. Hierzu wird die Ada **select**-Anweisung verwendet. Beispiel:

```
task screen_output is
 entry call_ch(val: character; x, y: integer);
 entry call_int(z, x, y: integer);
end screen_out;
task body screen_output is
 ...
 select
   accept call_ch ...  do...
   end call_ch;
 or
   accept call_int ... do ..
   end call_int;
 end select;
 ...
```

In diesem Fall wartet `screen_out`, bis `call_ch` oder `call_int` aufgerufen wird. ∇

Durch die Verwendung der **select**-Anweisung ist Ada nichtdeterministisch. Ada war während einer Reihe von Jahren die Sprache der Wahl für die Programmierung vieler militärischer Ausrüstungsgegenstände, die in der westlichen Hemisphäre produziert wurden. Aktuelle Informationen über Ada sind auf verschiedenen Webseiten zu finden (siehe z.B. [289]).

2.8.3 *Kommunikationsbibliotheken*

Gewöhnliche von-Neumann-Sprachen verfügen über keinerlei eingebaute Kommunikationsfunktionen. Diese können aber durch Bibliotheken zur Verfügung gestellt werden. Es gibt einen Trend dahin, sowohl die Kommunikation innerhalb eines Systems wie auch Kommunikation über größere Entfernungen zu unterstützen. Zunehmend werden Internetprotokolle angewandt.

MPI

Die Programmierung von Mehrkern-Systemen ist mit dem sogenannten *Message Passing Interface* (MPI) möglich. MPI ist eine sehr häufig eingesetzte Bibliothek, die ursprünglich für Hochleistungsrechner entwickelt wurde. Sie basiert auf Nachrichtenaustausch und ermöglicht synchronen wie auch asynchronen Nachrichtenaustausch.
Synchroner Nachrichtenaustausch ist beispielsweise mit Hilfe der Bibliotheksfunktion `MPI_Send` möglich [396]:
`MPI_Send(buffer, count, type, dest, tag, comm)` mit:

- `buffer` : Adresse der zu sendenden Daten,
- `count`: Anzahl zu sendender Datenelemente,
- `type`: zu sendender Datentyp (z.B. `MPI_CHAR, MPI_SHORT, MPI_INT`),
- `dest` : Prozess-ID des Zielprozesses,
- `tag` : Nachrichten-ID (zur Sortierung ankommender Nachrichten),
- `comm` : Kommunikationskontext (Menge von Prozessen, für die das Zielfeld gültig ist) und
- Funktionsergebnis: gibt Erfolg oder Misserfolg an.

Die folgende Funktion ist eine asynchrone Bibliotheksfunktion:
`MPI_Isend(buffer, count, type, dest, tag, comm, request)` mit

- `buffer`, `count`, `type`, `dest`, `tag`, `comm`: wie oben und
- das System erzeugt zusätzlich eine eindeutige „Anfrage-Nummer”. Der Programmierer kann diese in einer Warteroutine nutzen, um festzustellen, ob die nichtblockierende Operation beendet wurde.

Die Aufteilung von Rechenressourcen zwischen verschiedenen Prozessoren muss bei MPI explizit erfolgen, das Gleiche gilt für die Kommunikation und das Verteilen von Daten. Synchronisation ist implizit durch die Kommunikation, zusätzlich ist explizite Synchronisation möglich. Demzufolge ist ein Großteil des Verwaltungscodes explizit zu implementieren, was einen großen Arbeitsaufwand für den Programmierer bedeutet. Zudem skaliert dieser Ansatz nicht gut, wenn die Anzahl der Prozessoren sich nennenswert ändert [553].

Um MPI-artige Kommunikation auf Echtzeitsysteme anwenden zu können, wurde eine Echtzeit-Version von MPI definiert, MPI/RT genannt [501]. MPI/RT deckt Funktionalität wie *Thread*-Erzeugung und -Beendigung nicht ab. MPI/RT wird als

mögliche Schicht zwischen dem Betriebssystem und Standard-(nicht-Echtzeit-) MPI angesehen.

MPI ist für eine Vielzahl von Prozessoren verfügbar und wird auch für Mehrkern-*Chips* in Betracht gezogen. Jedoch basiert es auf der Annahme, dass Speicherzugriffe schneller als Kommunikationsoperationen sind. Zudem zielt MPI hauptsächlich auf homogene Mehrprozessorsysteme ab. Beide Annahmen sind für mehrere Prozessoren auf einem *Chip* nicht notwendigerweise gültig.

MPI wurde kürzlich erweitert, um auch Kommunikation auf der Basis eines gemeinsamen Speichers abzudecken.

OpenMP

Bei OpenMP ist die Parallelität größtenteils explizit, wogegen die Aufteilung von Berechnungen, Kommunikation, Synchronisation usw. implizit stattfindet. Parallelität wird durch Pragmas ausgedrückt: beispielsweise können Schleifen mit Pragmas versehen werden, die angeben, dass sie parallelisiert werden sollten.

Beispiel 2.40: Das folgende Programm beinhaltet eine kleine parallele Schleife [439]:

```
void a1(int n, float *a, float *b) {
 int i;
 #pragma omp parallel for
 for (i=1; i<n; i++)                       /* i ist standardmäßig privat */
   b[i] = (a[i] + a[i-1]) / 2.0;
}
```

Ein einfaches Pragma reicht hier aus, um eine mögliche parallele Ausführung anzuzeigen. ∇

Dies bedeutet, dass OpenMP vom Benutzer nur einen kleinen Aufwand für die Parallelisierung verlangt. Dies bedeutet aber auch, dass der Benutzer die Partitionierung nicht kontrollieren kann [553]. Es existieren Anwendungen für Multiprozessor-Systeme auf SoCs, sogenannte MPSoCs (siehe z.B. [367]).

Weitere Techniken für die Programmierung von Mehrkern-Prozessoren werden im Abschnitt über Systemsoftware vorgestellt werden (siehe Seite 253).

2.8.4 Weitere Sprachen

Die Sprache Java ist ursprünglich nicht für eingebettete Systeme entwickelt worden. Es gab Bemühungen, die daraus resultierenden Probleme zu lösen [12, 271]. Allerdings bleiben Android und Java für Chipkarten die einzigen wichtigen Anwendungen. Der Workshop JTRES über „Java Technologies for Real-time and Embedded

Systems” (siehe http://jtres2016.compute.dtu.dk/) spiegelt den letzten Stand des Einsatzes von Java wider.

Die Sprache Pearl [127] wurde für industrielle Kontrollanwendungen entwickelt. Sie beinhaltet viele Sprachelemente, um Prozesse zu kontrollieren, auch die Modellierung von Zeit ist möglich. Pearl benötigt ein zugrunde liegendes Echtzeitbetriebssystem. Die Sprache ist in Europa sehr beliebt gewesen, was sich in der großen Anzahl von industriellen Steuerungsanwendungen zeigt, die in dieser Sprache implementiert wurden. Pearl unterstützt Semaphore, die dazu verwendet werden können, die Kommunikation über gemeinsame Puffer zu schützen.

Chill [592] wurde für Telefonvermittlungsstellen entworfen. Die Sprache wurde von der CCITT standardisiert und in Telekommunikationsgeräten verwendet. Chill ist eine Art erweitertes PASCAL.

IEC 60848 [232] und STEP 7 [488] sind auf die Anwendung in der Regelungstechnik spezialisierte Sprachen. Beide Sprachen enthalten graphische Elemente zur Beschreibung der Funktionalität eines Systems.

2.9 Ebenen der Hardware-Modellierung

In der Praxis werden Entwürfe auf unterschiedlichen Abstraktionsebenen begonnen. In einigen Fällen wird auf einer hohen Abstraktionsebene das Gesamtverhalten des zu entwerfenden Systems beschrieben. In anderen Fällen beginnt die Spezifikation mit der Beschreibung einer elektrischen Schaltung auf einer entsprechend niedrigeren Abstraktionsstufe. Für jede Abstraktionsebene gibt es eine Anzahl von Sprachen, einige Sprachen decken auch mehrere Abstraktionsebenen ab. Im Folgenden wird eine Menge von möglichen Abstraktionsebenen beschrieben. Einige der niedrigeren Abstraktionen werden hier nur der Vollständigkeit halber erwähnt – eine Spezifikation sollte nicht auf einer dieser Ebenen beginnen. Es folgt eine Liste von häufig verwendeten Bezeichnungen und Eigenschaften von Abstraktionsebenen:

- **Modelle auf Systemebene:** Der Begriff Systemebene ist nicht klar definiert. Er wird hier verwendet, um das gesamte eingebettete System und das System, in das die Informationsverarbeitung integriert ist („das Produkt“) zu beschreiben, und möglicherweise auch die physische Umwelt (z.B. die Straßenverhältnisse oder Wetterbedingungen) darstellen zu können. Offensichtlich beinhalten solche Modelle Aspekte sowohl der physischen Umgebung wie auch der Informationsverarbeitung und es kann schwierig sein, dafür passende Simulatoren zu finden. Mögliche Lösungen sind die Verwendung von VHDL-AMS (die analoge Erweiterung von VHDL), SystemC , Modelica, COMSOL (siehe https://www.comsol.com/) oder MATLAB/Simulink. MATLAB und VHDL-AMS unterstützen die Modellierung partieller Differentialgleichungen, eine der wichtigsten Anforderungen bei der Simulation physischer Systeme. Es ist eine große Herausforderung, die informationsverarbeitenden Bestandteile des Systems so zu modellieren, dass das Simulationsmodell auch zur Synthese des eingebetteten Systems verwendet

werden kann. Wenn das nicht möglich ist, muss unter Umständen eine fehleranfällige manuelle Übersetzung zwischen den verschiedenen Modellen durchgeführt werden.

- **Algorithmische Ebene:** Auf dieser Ebene werden Algorithmen simuliert, die innerhalb des eingebetteten Systems zum Einsatz kommen sollen. Beispielsweise kann ein MPEG Video-*Encoder* simuliert werden, um die Ausgabequalität der Videos zu bestimmen. Bei der Verwendung solcher Simulationen gibt es keinen Bezug zum Befehlssatz des Zielprozessors.
  Datentypen können in der Simulation durchaus noch eine höhere Wortbreite haben als in der endgültigen Implementierung. Beispielsweise verwenden die Referenzimplementierungen des MPEG-Standards doppelt-genaue Gleitkommazahlen. Das endgültige eingebettete System wird solche Datentypen kaum verarbeiten können. Wenn die Datentypen so gewählt wurden, dass jedes Bit in der Simulation genau einem Bit in der Implementierung entspricht, so spricht man von einem **bitgenauen Modell**. Die Übersetzung von nicht-bitgenauen in bitgenaue Modelle sollte durch Hilfsprogramme unterstützt werden (siehe Seite 388).

- **Befehlssatzebene:** In diesem Fall wurden die Algorithmen bereits für den Befehlssatz des zu verwendenden Prozessors (oder der Prozessoren) übersetzt. Simulationen auf dieser Ebene erlauben das Zählen der ausgeführten Instruktionen. Es gibt verschiedene Varianten der Befehlssatzebene:
  - In einem grobkörnigen Modell wird nur die Wirkung von Instruktionen simuliert, das Zeitverhalten wird vernachlässigt. Die Informationen aus Assembler-Referenz-Handbüchern (die Befehlssatzarchitektur, *Instruction Set Architecture* (ISA)) sind für die Definition eines solchen Modells ausreichend.
  - **Modellierung auf Transaktionsebene:** Bei der Modellierung auf Transaktionsebene werden Transaktionen, wie z.B. Bus-Lese- oder Schreiboperationen, sowie Kommunikationsvorgänge zwischen verschiedenen Komponenten modelliert. Diese Art der Modellierung enthält weniger Details als die zyklengenaue Modellierung (s.u.), daher können hier deutliche Vorteile bei der Simulationsgeschwindigkeit erzielt werden [105].
  - In einem feinkörnigeren Modell kann man eine **zyklengenaue Befehlssatzsimulation** erreichen. In diesem Fall kann die genaue Zyklenzahl, die zur Ausführung einer Applikation benötigt wird, bestimmt werden. Zum Aufstellen von zyklengenauen Modellen braucht man sehr genaue Informationen über die Prozessorhardware, um z.B. eine Fließband-artige Befehlsverarbeitung, Ressourcenkonflikte und Speicherwartezyklen richtig modellieren zu können.

- **Register-Transfer-Ebene (RTL):** Auf dieser Ebene werden alle Komponenten auf der Register-Transfer-Ebene modelliert. Das beinhaltet arithmetisch/logische Einheiten (ALUs), Register, Speicher, Multiplexer und Dekodierer. Modelle auf dieser Ebene sind immer zyklengenau. Die automatische Synthese aus solchen Modellen heraus ist keine große Herausforderung.
- **Modelle auf Gatterebene:** Hier enthalten die Modelle Gatter als Basisbausteine. Modelle auf Gatterebene erlauben genaue Aussagen über die Wahrscheinlich-

keiten einer Wertänderung von Signalen und können deshalb zur Energiebestimmung herangezogen werden. Die Berechnung von Verzögerungen kann hier genauer durchgeführt werden als bei einem Modell auf RTL-Ebene. Allerdings ist während der Entwurfsphase üblicherweise keine Information über die Längen von Verbindungsleitungen und somit auch nicht über deren Kapazitäten verfügbar. Daher sind Verzögerungs- und Energiewerte auch auf dieser Ebene lediglich Schätzungen.
Der Begriff „Gatterebene“ wird manchmal auch in Situationen verwendet, wo die Gatter lediglich dazu verwendet werden, um Boolesche Funktionen darzustellen. Gatter in einem solchen Modell repräsentieren aber nicht unbedingt physikalische Gatter. Es wird dann nur das Verhalten der Gatter betrachtet, nicht die Tatsache, dass sie in der späteren Realisierung physikalische Komponenten darstellen. Genauer sollten solche Modelle eigentlich „Boolesche Funktionsmodelle“ heißen[23], dieser Begriff ist aber nicht sehr weit verbreitet.
- **Modelle auf Schalterebene:** Modelle auf Schalterebene verwenden Schalter (Transistoren) als Grundbausteine. Solche Beschreibungen verwenden Modelle digitaler Signalwerte (siehe Beschreibung möglicher Wertemengen ab Seite 97). Im Gegensatz zu Modellen auf Gatterebene können die Modelle auf Schalterebene den bidirektionalen Transfer von Informationen in Schaltungen abbilden. Für die Simulation wird häufig die ternäre Simulation eingesetzt [72].
- **Modelle auf Schaltungsebene:** Die Schaltungstheorie und ihre Bestandteile (Strom- und Spannungsquellen, Widerstände, Kondensatoren, Spulen und möglicherweise Makromodelle von Halbleitern) bilden die Basis der Simulation auf dieser Ebene. Simulationen beinhalten partielle Differentialgleichungen. Diese Gleichungen sind nur dann linear, wenn das Verhalten der Halbleiter linearisiert (approximiert) wird. Die am häufigsten verwendeten Simulatoren auf der Schaltungsebene sind SPICE [556] und dessen Varianten.
- **Modelle auf Layoutebene:** Modelle auf Layoutebene zeigen das tatsächliche Layout der Schaltung. Solche Modelle beinhalten **geometrische** Informationen. Sie können nicht direkt simuliert werden, da die Geometrie keine konkreten Anhaltspunkte für das Verhalten gibt. Das Verhalten kann erschlossen werden, wenn man das Modell auf Layoutebene gemeinsam mit einem verhaltensorientierten Modell auf einer höheren Ebene kombiniert, oder indem aus dem *Layout* die Schaltung extrahiert wird, wobei man Wissen über die Darstellung von Komponenten auf der Layoutebene benötigt. In einem typischen Entwurfsablauf werden die Länge von Verbindungsleitungen und deren entsprechende Kapazitäten aus dem Layoutmodell extrahiert und in die Beschreibungen auf höherer Ebene **zurück-annotiert (engl. *back-annotated*)**. Auf diese Weise kann die Genauigkeit von Verzögerungs- und Energieschätzungen verbessert werden. Layoutinformationen können auch für die thermische Modellierung benötigt werden.
- **Modelle auf Prozess- und Schaltungs-Ebene:** Auf einer noch niedrigeren Stufe kann man den Herstellungsprozess modellieren. Aufgrund der Information aus diesen Modellen kann man Parameter (z.B. Verstärkung und Kapazität) für

[23] Diese Modelle könnten mit Hilfe von binären Entscheidungsdiagrammen (*Binary Decision Diagrams* (BDD)) [570] dargestellt werden.

die verwendeten Bauelemente (Transistoren) berechnen. Man beachte, dass die Verwendung des Begriffs „Prozess" in der Fertigungstechnologie mit unserer bisherigen Benutzung nicht kompatibel ist.

2.10 Vergleich der Berechnungsmodelle

2.10.1 Kriterien

Berechnungsmodelle lassen sich anhand verschiedener Kriterien vergleichen. So vergleicht z.B. Stuijk [515] Berechnungsmodelle nach folgenden Kriterien:

- **Ausdrucksstärke** und **Kompaktheit** zeigen an, welche Systeme modellierbar sind und wie kompakt diese Beschreibung ist.
- **Analysierbarkeit** bezieht sich auf die Verfügbarkeit von *Scheduling*-Algorithmen und darauf, ob Unterstützung für Echtzeitsysteme vorhanden ist.
- Die **Implementierungseffizienz** wird durch das benötigte *Scheduling*-Verfahren und die Codegröße beeinflusst.

Abb. 2.74 zeigt eine Klassifikation von Datenflussmodellen anhand dieser Kriterien.

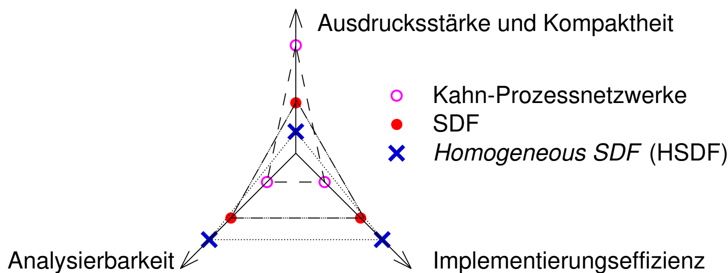


Abb. 2.74 Vergleich von Datenflussmodellen

Diese Abbildung verdeutlicht, dass Kahn-Prozessnetzwerke ausdrucksstark sind: sie sind Turing-vollständig, also kann jedes Problem, das von einer Turing-Maschine berechnet werden kann, auch von einem KPN berechnet werden. Turing-Maschinen werden als Standardmodell für universelle Computer verwendet [215]. Es ist allerdings schwierig, Terminierungseigenschaften und obere Grenzen für Puffergrößen von KPNs zu analysieren. Dagegen sind SDF-Graphen und *Cyclo-Static Data Flow* (CSDF, siehe Seite 82) nicht Turing-vollständig, da sie nicht dazu in der Lage sind, Kontrollflüsse zu modellieren. Dafür lassen sich Verklemmungseigenschaften und obere Grenzen für Puffergrößen von SDF-Graphen einfacher analysieren. Homogene

SDF-Graphen (HSDF) (siehe Seite 82) sind noch weniger ausdrucksstark, aber noch einmal einfacher zu analysieren.

Wir könnten Berechnungsmodelle auch in Hinblick auf die Arten von Prozessen, die unterstützt werden, vergleichen:

- Die **Anzahl von Prozessen** kann entweder **statisch** oder **dynamisch** sein. Eine statische Anzahl von Prozessen vereinfacht die Implementierung und ist ausreichend, wenn jeder Prozess eine bestimmte Hardwarekomponente modelliert und wir *hot plugging* (das Hinzufügen von Hardware zur Laufzeit) nicht berücksichtigen.
- Prozesse können entweder statisch **verschachtelt** sein oder alle Prozesse sind auf gleicher Ebene deklariert. Beispielsweise ermöglicht StateCharts die verschachtelte Deklaration von Prozessen, wogegen SDL (siehe Seite 68) dies nicht erlaubt. Die Verschachtelung erlaubt es, Belange zu verkapseln.
- Es gibt verschiedene Techniken zur **Prozesserzeugung**. Prozesse können durch eine Analyse der Prozessdeklarationen im Sourcecode, durch den *fork*- und *join*-Mechanismus (der z.B. von Unix unterstützt wird) und auch durch explizite Funktionen zur Prozesserzeugung erzeugt werden.

Die Ausdrucksstärke der verschiedenen datenflussorientierten Berechnungsmodelle ist auch in Abb. 2.75 [41]. dargestellt. In diesem Buch nicht aufgeführte Berechnungsmodelle sind dabei durch gestrichelte Linien dargestellt.

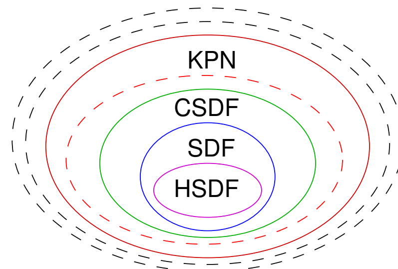


Abb. 2.75 Ausdrucksstärke von Datenflussmodellen

Keines der Berechnungsmodelle und keine der bisher vorgestellten Sprachen erfüllt alle Anforderungen an eine Spezifikationssprache für eingebettete Systeme. Tabelle 2.6 gibt einen Überblick über einige der wichtigsten Eigenschaften ausgewählter Sprachen.

SpecC und SystemC erfüllen alle aufgeführten Anforderungen. Allerdings beinhaltet diese Liste einige andere Anforderungen nicht, wie z.B. die präzise Angabe von *Deadlines*. Es ist sehr unwahrscheinlich, dass ein bestimmtes Berechnungsmodell oder eine bestimmte Sprache jemals alle Anforderungen erfüllen wird, da einige der Anforderungen im Konflikt zueinander stehen. Eine Sprache, die harte Echtzeitanforderungen unterstützt, mag weniger dafür geeignet sein, Systeme mit weniger harten Echtzeitanforderungen zu beschreiben. Eine Sprache, die für verteilte Regelungsanwendungen gedacht ist, mag sich schlecht für lokale Datenfluss-dominierte

Tabelle 2.6 Vergleich von Sprachen

Sprache	Verhaltens-Hierarchie	Strukturelle Hierarche	Programmier-sprachen-elemente	Unterstützung von Ausnahme-behandlung	Dynamische Prozess-erzeugung
StateCharts	+	-	-	+	-
VHDL	+	+	+	-	-
SDL	+-	+-	+-	-	+
Petri-Netze	-	-	-	-	+
Java	+	-	+	+	+
SpecC	+	+	+	+	+
SystemC	+	+	+	+	+
Ada	+	-	+	+	+

Anwendungen eignen. Daher ist damit zu rechnen, dass wir Kompromisse in Kauf nehmen müssen.

Welche Kompromisse kommen nun in der Praxis zum Einsatz? Programmieren in Assemblersprache war in der Anfangszeit eingebetteter Systeme sehr verbreitet. Die Programme waren klein genug, um die Komplexität der Probleme auch in Assembler zu beherrschen. Der nächste Schritt war die Verwendung von C und mit C verwandten Sprachen. Die weiter steigende Komplexität eingebetteter Systeme (siehe Seite 18) wird dazu führen, dass Sprachen auf einer höheren Ebene anstelle von C eingesetzt werden. Diese nächste Abstraktionsebene wird z.B. von objektorientierten Sprachen oder SDL abgedeckt. Sprachen wie UML werden ebenfalls benötigt, damit Spezifikationen in frühen Entwurfsphasen festgehalten werden können. Ein Trend geht hin zu modellbasiertem Entwurf [477]. In der Praxis lassen sich diese Sprachen wie in Abb. 2.76 gezeigt verwenden.

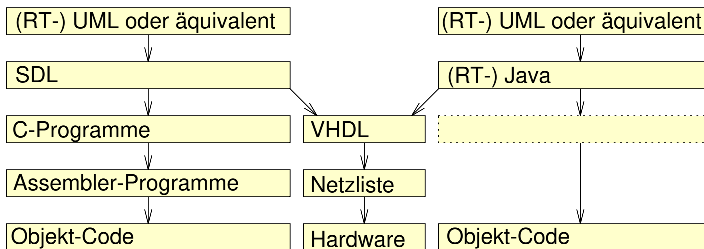


Abb. 2.76 Verwendung von Sprachen in Kombination

Nach Abb. 2.76 lassen sich Sprachen wie SDL oder StateCharts nach C übersetzen. Diese Beschreibungen können dann compiliert werden. Ein mit SDL oder StateCharts begonnener Entwurf ermöglicht auch die Implementierung von Funktionalität in Hardware, wenn Übersetzer von diesen Sprachen in eine Hardwarebeschreibungssprache wie VHDL zur Verfügung stehen. Sowohl C wie auch VHDL

werden auf absehbare Zeit noch als Zwischensprachen verwendet werden. Java benötigt keine Zwischenschritte, profitiert aber auch von ausgereiften Übersetzungsmethoden in Maschinensprache. Auf ähnliche Weise lassen sich auch Übersetzungen zwischen verschiedenen Arten von Graphen realisieren. So können SDF-Graphen in eine Unterklasse von Petrinetzen übersetzt werden [515]. Sie entsprechen auch einer Unterklasse des *computation graph model* von Karp und Miller [283]. Die Verbindung der verschiedenen Berechnungsmodelle wird durch formale Methoden vereinfacht [95].

Eine Reihe von Sprachen zum Entwurf eingebetteter Systeme werden in einem von M. Radetzki herausgegebenen Buch [464] besprochen. Popovici et al. [457] verwenden eine Kombination von Simulink und SystemC.

Weiterhin gibt es noch algebraische Sprachen wie LOTOS [257] und Z [504], die präzise Spezifikationen erlauben, die aber nicht ausführbar sind.

2.10.2 Unified Modeling Language (UML)

Die Sprache UML™ beinhaltet Diagramme, die verschiedenen Berechnungsmodellen entsprechen. Tabelle 2.7 zeigt eine Klassifikation der einzelnen bisher betrachteten UML-Diagrammarten in Hinblick auf die hier besprochenen Berechnungsmodelle.

Tabelle 2.7 In UML™ verfügbare Berechnungsmodelle

Kommunikation/ Komponenten	Gemeinsamer Speicher *shared memory*	Nachrichtenaustausch	
		synchron	asynchron
Undefinierte Komponenten	Anwendungsfälle (*use cases*)		
		Sequenzdiagramme, Zeitdiagramme	
Differentialgleichungen	-		
Endliche Automaten	Zustandsdiagramme	-	-
Datenfluss	-	Datenflussdiagramme	
Petrinetze	(nicht sinnvoll)	Aktivitätsdiagramme	
Verteilte Ereignismodelle	-	-	
Von-Neumann-Modell	-	-	

Diese Abbildung stellt dar, inwieweit UML einige der besprochenen Berechnungsmodelle abdeckt. Dabei liegt der Schwerpunkt auf frühen Entwurfsphasen. Die Semantik der Kommunikation ist meist nur unpräzise definiert, daher kann unsere Klassifikation in dieser Hinsicht nicht genau sein. Zusätzlich zu den bereits erwähnten Diagrammen lassen sich die folgenden Diagramme modellieren:

- ***Deployment*-Diagramme**: Diese Diagramme sind für eingebettete Systeme wichtig: sie beschreiben die „Ausführungsarchitektur" von Systemen (Hardware- oder Softwareknoten).
- **Paketdiagramme**: Paketdiagramme beschreiben die Partitionierung der Software in Softwarepakete. Sie ähneln den Moduldiagrammen in StateMate.

- **Klassendiagramme**: Diese Diagramme beschreiben Vererbungsbeziehungen zwischen Objektklassen.
- **Kommunikationsdiagramme** (in UML 1.0 **Kollaborationsdiagramme** genannt): Diese Graphen stellen Klassen, Relationen zwischen Klassen und zwischen diesen ausgetauschte Nachrichten dar.
- **Komponentendiagramme:** Diese stellen die Komponenten von Anwendungen oder Systemen dar.
- **Objektdiagramme**, **Interaktions-Überblicks-Diagramme**, **zusammengesetzte Strukturdiagramme**: Diese drei Arten von Diagrammen werden seltener verwendet. Einige davon sind auch Spezialfälle anderer Diagramme.

Die verfügbaren Werkzeuge erlauben es in begrenzter Weise, die Konsistenz zwischen verschiedenen Diagrammarten zu überprüfen. Eine komplette Überprüfung scheint aber unmöglich zu sein. Eine Ursache hierfür ist, dass die Semantik von UML ursprünglich nicht definiert wurde. Es wurde behauptet, dass dies absichtlich geschah, da die Beschäftigung mit genauer Semantik erst in späteren Entwurfsphasen erfolgen soll. Folglich lassen sich genaue ausführbare Spezifikationen nur dann aus einer UML-Beschreibung erzeugen, wenn UML mit einer weiteren Sprache kombiniert wird. Einige verfügbare Werkzeuge kombinieren UML mit SDL [228] und C++. Es gibt aber auch erste Ansätze, die Semantik von UML zu definieren.

Die Version 1.4 von UML war nicht für eingebettete Systeme entworfen worden. Daher fehlen dieser Version eine Reihe von Eigenschaften, die zur Modellierung eingebetteter System unerläßlich sind (siehe Seite 31). Insbesondere fehlen die folgenden Eigenschaften [387]:

- keine Partitionierung von Software in Tasks bzw. Prozesse,
- Zeitverhalten kann nicht beschrieben werden,
- die wesentlichen Hardwarekomponenten eines Systems können nicht in die Beschreibung integriert werden.

Durch die weiter zunehmende Menge an Software in eingebetteten Systemen gewinnt UML auch in diesem Bereich an Bedeutung. Es gibt daher verschiedene Vorschläge für UML-Erweiterungen, die Echtzeitanwendungen unterstützen [387, 137]. Diese wurden in der Entwurfsphase von UML 2.0 berücksichtigt. UML 2.0 beinhaltet 13 Arten von Diagrammen (im Vergleich zu 9 Arten in UML 1.4) [13]. Besondere Profile berücksichtigen die Anforderungen von Echtzeitsystemen [369]. Diese Profile beinhalten Klassendiagramme mit Beschränkungen, *Icons*, Diagrammsymbole und einige (partielle) Semantikbeschreibungen, lassen aber auch semantische Fragen offen. Es gibt UML-Profile für folgende Aspekte und Aufgaben [369]:

- *Schedulability*, Leistung, und Zeitangaben [430],
- Testen [434],
- Dienstgüte (*Quality of Service* (QoS)) und Fehlertoleranz [434],
- Systemmodellierung mit einer Sprache namens SysML [432],
- Modellierung und Analyse von eingebetteten Echtzeitsystemen (MARTE) [431],
- Interoperabilität von UML und SystemC [469],
- Wiederverwendung von *Intellectual Property* (IP) mit dem SPRINT-Profil [505].

Mit solchen Profilen lassen sich beispielsweise Sequenzdiagramme mit Zeitinformationen ergänzen. Profile können aber zueinander inkompatibel sein.

2.10.3 Ptolemy II

Modellierung, Simulation und Entwurf mit verschiedenen Berechnungsmodellen und deren Kombination sind Ziel des Ptolemy-Projekts [460]. Eine Betonung liegt auf eingebetteten Systemen, die verschiedene Technologien und dementsprechend auch MoCs mischen. Beispielsweise können analoge und digitale Elektronik, Hard- und Software, elektrische und mechanische Geräte beschrieben werden. Ptolemy unterstützt verschiedene Anwendungen, einschl. der Signalverarbeitung, Regelungstechnik, sequentielle Entscheidungsunterstützung und Benutzerschnittstellen. Ein Schwerpunkt gilt der Erzeugung von eingebetteter Software. Die Kernidee besteht darin, die Software aus demjenigen Berechnungsmodell zu erzeugen, das für eine bestimmte Anwendung am geeignetsten ist. Die Version 2 von Ptolemy (Ptolemy II) kennt die folgenden Berechnungsmodelle und zugehörigen Anwendungsgebiete (siehe auch Seite 43):

1. Kommunizierende sequenzielle Prozesse (CSP),
2. kontinuierliche Zeit (für mechanische Systeme und analoge Schaltungen),
3. diskretes Ereignismodell,
4. verteilte diskrete Ereignisse,
5. endliche Automaten (engl. *Finite State Machines* (FSM)),
6. Prozessnetzwerke (Kahn-Prozessnetzwerke, siehe Seite 76),
7. SDF (siehe Seite 68),
8. synchrone/reaktive Berechnungsmodelle.

Diese Liste macht deutlich, dass die Untersuchung verschiedener Berechnungsmodelle ein Schwerpunkt des Ptolemy-Projektes ist.

2.11 Aufgaben

Die folgenden Aufgaben sollten entweder zu Hause oder während einer Anwesenheitsphase nach dem *flipped classroom*-Konzept bearbeitet werden. Bei diesem Konzept sind die häuslichen Tätigkeiten und die Tätigkeiten an der Hochschule gegenüber einer klassischen Aufteilung weitgehend vertauscht: zu Hause erfolgt ein Studium eines technischen Gebietes und an der Hochschule wird das Gebiet in Teamarbeit diskutiert und praktisch erprobt [376].

2.1: Nennen Sie bis zu sechs Anforderungen an Spezifikations- und Modellierungssprachen für eingebettete Systeme!

2.2: Warum könnte es bei der Ausführung unserer Spezifikation zu Verklemmungen (engl. *Deadlocks*) kommen?

2.3: Was ist ein Berechnungsmodell (engl. *Model of Computation* (MoC))?

2.4: Was ist ein Job und was unterscheidet ihn von einer Task?

2.5: Welche beiden Schlüsseltechniken gibt es für die Kommunikation in Rechnern?

2.6: Welche Techniken können benutzt werden, um erste Ideen für ein zu entwerfendes System zu erfassen?

2.7: Simulieren Sie den Zugverkehr zwischen Paris, Brüssel, Amsterdam und Köln mit der levi Simulationssoftware [498]! Modifizieren Sie die vorhandenen Beispiele so, dass zwischen je zwei Bahnhöfen immer zwei Gleise existieren und zeigen Sie einen beliebigen Ablaufplan (engl. *Schedule*) für zehn Züge!

2.8: Laden Sie die OpenModelica™ Simulations-Software herunter. Entwickeln Sie ein Simulationsmodell für *Newton's cradle* (siehe z.B. https://en.wikipedia.org/wiki/Newton%27s_cradle).

2.9: Modifizieren Sie den Anrufbeantworter aus Beispiel 2.8 so, dass der Eigentümer jederzeit während des Abspielens der Ansage oder des Aufnehmens des Anrufers den Anruf entgegennehmen kann.

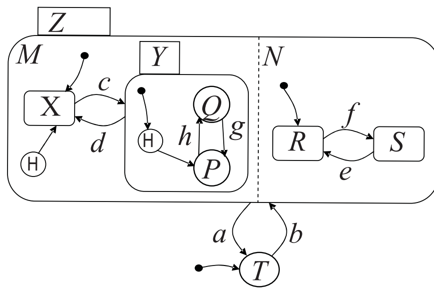
2.10: Modellieren Sie Ihre täglichen Aktivitäten mit einem zeitbehafteten Automaten! Stunden sollen einer Variablen h entsprechen, Tage einer Variablen d, wobei $d = 1$ der Montag sein soll und $d = 7$ der Sonntag.

An einem Wochenende verlassen Sie den schlafenden Zustand zwischen $h = 10$ und $h = 11$, verbringen 1-2 Stunden mit der Vorbereitung des Tages, bleiben bei einem Freund bis zu einer Zeit zwischen $h = 20$ und $h = 21$, gehen nach Hause und schlafen zwischen $h = 22$ und $h = 23$ ein. In der Woche ($d \in [1..5]$) wachen Sie zwischen $h = 7$ und $h = 8$ auf, verbringen 1-2 Stunden mit der Vorbereitung des Tages, studieren bis zu einer Zeit zwischen $h = 20$ und $h = 21$, gehen nach Hause und schlafen zwischen $h = 22$ und $h = 23$ ein. d muss am Ende eines Tages erhöht werden.

2.11: Gegeben sei das StateCharts-Modell von Abb. 2.77 (links). Weiterhin sei folgende Sequenz von Eingaben gegeben: $b\ c\ f\ h\ g\ h\ e\ a\ b\ c$. Markieren Sie im Diagramm von Abb. 2.77 (rechts) alle Zustände, in denen sich das StateCharts-Modell befindet, nachdem die angegebene Eingabe anliegt! H kennzeichnet hier den *History*-Mechanismus.

2.12: Sind StateCharts unter Verwendung der StateMate-Semantik deterministisch? Erklären Sie Ihre Antwort!

2.13: Ist die Sprache SDL deterministisch? Erklären Sie die Antwort!



	M	N	P	Q	R	S	T	X	Y	Z
(Reset)							v			
b										
c										
f										
h										
g										
h										
e										
a										
b										
c										

Abb. 2.77 StateCharts-Beispiel: **links**: graphisches Modell; **rechts**: Tabelle der Zustände

2.14: Stellen Sie sich vor, dass Sie die Besucherströme im hypothetischen *Museum of Fine Future Information Nuggets* (MUFFIN) modellieren wollen. Das Museum hat drei Ausstellungshallen. Vor jeder Halle befindet sich Platz für eine Warteschlange, von dem aus man die jeweilige Halle betreten kann. Hallenausgänge führen zu den drei Warteschlangen. Besucher können nach dem Verlassen einer Halle eine beliebige Halle als ihre nächste aussuchen. Nehmen Sie an, dass jede Halle als ein Prozess beschrieben werden kann. Die Zeit, die ein Besucher in einer Halle verbringt, ist zufällig. Wir betrachten den stabilen Zustand, in dem kein Besucher das Museum betritt oder verlässt. Modellieren Sie das Museum in SDL! Nutzen Sie explizite Prozesse und FIFO-Schlangen.

2.15: Laden Sie die levi-Software für KPNs [496] und entwickeln Sie ein verteiltes KPN-Modell zur Berechnung von Fibonacci-Zahlen. Das Modell darf nicht nur aus einem einzigen Knoten bestehen.

2.16: Welche drei Arten von Petrinetzen wurden in diesem Buch beschrieben?

2.17: Eine Art von Petrinetzen ermöglicht die Verwendung mehrerer, nicht unterscheidbarer Marken pro Position. Welche Komponenten werden in einem mathematischen Modell solcher Netze verwendet? Hinweis: N=(P, ...)

2.18: Zeichnen Sie das folgende C/E-System:

- Netz: $N = (C, E, F)$
- Bedingungen: $C = \{c_1, c_2, c_3, c_4\}$,
- Ereignisse: $E = \{e_1, e_2, e_3\}$,
- Relation: $F = \{(c_1, e_1), (c_1, e_2), (e_1, c_2), (e_1, c_3), (e_2, c_2), (e_2, c_3), (e_2, c_4), (c_2, e_3), (c_3, e_3), (c_4, e_3), (e_3, c_1), (e_3, c_4)\}$

Was ist die Vorbedingung von e_3, was ist die Nachbedingung von e_1? Ist N einfach und rein? Wenn es das nicht ist: welche Kante(n) muss man entfernen, damit N ein reines Netz wird? Geben Sie eine kurze Begründung Ihrer Aussage an!

2.19: Skizzieren Sie ein kompaktes Modell des Problems der dinierenden Philosophen!

2.20: Die CSA-Theorie beschreibt 2, 3 und 4 Stärken von Logik, die 4, 7 und 10 Logikwerten entsprechen. Wie viele Stärken und Werte werden in IEEE 1164 verwendet? Erstellen Sie ein Diagramm, das die Teilordnung der Werte aus IEEE 1164 zeigt! Welche der Werte aus IEEE 1164 sind in dem Diagramm nicht in der Teilordnung enthalten, was bedeuten diese Werte?

2.21: Gegeben sei ein Bus wie in Abb. 2.78. Welche der Werte aus IEEE 1164 liegen

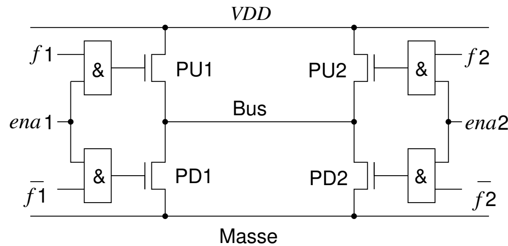


Abb. 2.78 Von Tri-State-Ausgängen getriebener Bus

auf dem Bus an, wenn beide „*enable*"-Eingänge auf `'0'` liegen ($ena1 = ena2 =$ `'0'`)? Welche der Werte aus IEEE 1164 liegen auf dem Bus an, wenn $ena1$=`'0'`, $ena2$=`'1'` und $f2$=`'1'`?

2.22: Welche der folgenden Schaltungen kann mit IEEE 1164 modelliert werden: komplementäre CMOS-Ausgänge, Ausgänge mit Verarmungstransistor (engl. *depletion transistor*), Tristate-Ausgänge, Vorladen von Bussen (wenn ebenfalls Verarmungstransistoren verwendet werden)?

2.23: Welche der folgenden Sprachen verwenden asynchronen Nachrichtenaustausch: StateCharts, SDL, VHDL, CSP, Petrinetze, MPI?

2.24: Welche der folgenden Sprachen verwenden einen *Broadcast*-Mechanismus zur Aktualisierung von Variablen: StateCharts, SDL, Petrinetze?

2.25: Welche der folgenden Diagrammarten werden von UML unterstützt: Sequenzdiagramme, Y-Diagramme, Anwendungsfälle, Aktivitätsdiagramme, Schaltpläne?

2.26: Tragen Sie in der nachfolgenden Tabelle Berechnungsmodelle der Komponenten und Kommunikationsmodelle in der linken Spalte ein. Anschließend tragen Sie bitte so viele UML-Diagramme wie möglich im Rest der Tabelle ein.

Kommunikation/Organisation der Komponenten			

Open Access Dieses Kapitel wird unter der Creative Commons Namensnennung 4.0 International Lizenz (http://creativecommons.org/licenses/by/4.0/deed.de) veröffentlicht, welche die Nutzung, Vervielfältigung, Bearbeitung, Verbreitung und Wiedergabe in jeglichem Medium und Format erlaubt, sofern Sie den/die ursprünglichen Autor(en) und die Quelle ordnungsgemäß nennen, einen Link zur Creative Commons Lizenz beifügen und angeben, ob Änderungen vorgenommen wurden.

Die in diesem Kapitel enthaltenen Bilder und sonstiges Drittmaterial unterliegen ebenfalls der genannten Creative Commons Lizenz, sofern sich aus der Abbildungslegende nichts anderes ergibt. Sofern das betreffende Material nicht unter der genannten Creative Commons Lizenz steht und die betreffende Handlung nicht nach gesetzlichen Vorschriften erlaubt ist, ist für die oben aufgeführten Weiterverwendungen des Materials die Einwilligung des jeweiligen Rechteinhabers einzuholen.

Kapitel 3
Hardware eingebetteter Systeme

In diesem Kapitel werden das Interface zwischen der physischen Umgebung und der Informationsverarbeitung (das **Cyphy-Interface**) sowie Hardware für die Verarbeitung, die Speicherung und die Kommunikation von Informationen vorgestellt. Die Notwendigkeit der Betrachtung des CyPhy-Interfaces ergibt sich dabei durch die Einbettung in ein CPS. Die übrige Hardware muss hier betrachtet werden, weil u.a. die erreichte Performanz, das Zeitverhalten, der Speicherbedarf, die Stromaufnahme und die Sicherheit von der eingesetzten Hardware abhängig sind.

Im Kontext des CyPhy-Interfaces werden in diesem Kapitel Schaltungen zur Zeit- und Wertediskretisierung sowie zu deren Umkehrung vorgestellt. Auch gehen wir auf das Abtasttheorem und seine Folgewirkungen ein. Zur Vorstellung der Informationsverarbeitung dienen v.a. verschiedene Klassen von effizienzoptimierter Hardware, insbesondere digitale Signalprozessoren (DSPs), Graphik-Prozessoren (GPU), Mehrkern-Systeme sowie programmierbare Logikschaltungen (FPGAs). Im Rahmen der Speicherung gehen wir auf Komponenten der Speicherhierarchie in einer für eingebettete Systeme angepassten Form ein. Zusätzlich bewerten wir Kommunikationstechnik im Hinblick auf ihre Eignung bei eingebetteten Systemen.

Zur Realisierung elektronischer Informationsverarbeitung wird elektrische Energie benötigt. Daher enthält das Kapitel einen Abschnitt über die Erzeugung, die Speicherung und die effiziente Nutzung der Energie in eingebetteten Systemen, einschließlich Batterie- und Verbrauchsmodellen. Das Kapitel schließt mit einer Übersicht über die Herausforderungen zur hardwaremäßigen Gewährleistung der Datensicherheit.

3.1 Einleitung

Vielfach werden Entwürfe für Hardware-Komponenten wiederverwendet. Die umfangreiche Wiederverwendung verfügbarer Hard- und Softwarekomponenten ist die Grundlage der **plattformbasierten Entwurfsmethode** (siehe Seite 322). Dieser An-

© Der/die Autor(en) 2021

P. Marwedel, *Eingebettete Systeme*, https://doi.org/10.1007/978-3-658-33437-6_3

forderung entsprechend und aufgrund des Entwurfsflusses in Abb. 3.1 beschreiben wir nun einige wichtige Grundlagen der Hardware eingebetteter Systeme.

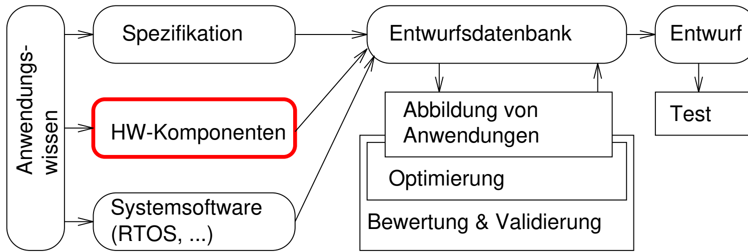


Abb. 3.1 Vereinfachter Entwurfsfluss

Die Hardware eingebetteter Systeme ist weit weniger standardisiert als jene von PCs. Die große Vielfalt von eingebetteter Systemhardware macht es unmöglich, einen umfassenden Überblick über alle Arten von Hardwarekomponenten zu geben. Wir versuchen dennoch, einen Überblick über einige wichtige Komponenten zu geben, die in den meisten Systemen vorhanden sind. In vielen cyber-physikalischen Systemen, insbesondere in Regelungssystemen, wird die Hardware in einer Schleife verwendet (siehe Abb. 3.2). Informationen über die physische Umgebung werden

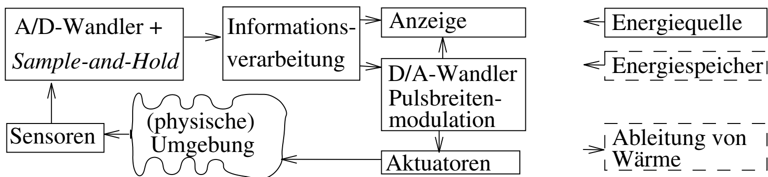


Abb. 3.2 Hardwareschleife

in dieser Schleife über **Sensoren** bereitgestellt. Sensoren erzeugen normalerweise kontinuierliche Folgen von Analogwerten. In diesem Buch beschränken wir uns auf Informationsverarbeitung, in der digitale Computer diskrete Wertefolgen verarbeiten. Die dafür erforderlichen Umwandlungen werden mit Hilfe zweier Arten von Schaltungen vorgenommen: Abtast- und Halteschaltungen (engl. *sample-and-hold circuits*) und Analog-Digital (A/D)-Wandler. Nach einer solchen Umwandlung können die Informationen digital verarbeitet werden. Die erzeugten Ergebnisse können dann angezeigt und weiterverwendet werden, um die physische Umgebung durch **Aktuatoren** zu beeinflussen. Da die meisten Aktuatoren analog arbeiten, wird hier auch eine Wandlung von digitalen in analoge Signale benötigt. Diese Wandlung

kann entweder direkt durch Digital-Analog-Wandler (DACs) oder indirekt über eine Pulsbreitenmodulation erfolgen.

Für die vorherrschende **elektronische** Datenverarbeitung benötigen wir elektrische Energie. Hierfür wird eine **Energiequelle** benötigt. Sofern unsere Energiequelle nicht permanent Energie bereitstellt, wird ein **Energiespeicher** benötigt, beispielsweise in Form aufladbarer Batterien oder Kondensatoren. Während des Betriebs wird ein großer Teil der elektrischen Energie in Wärme- oder thermische Energie (Hitze) gewandelt werden. Es kann erforderlich sein, die Wärme abzuführen, z.B. über Kühlkörper oder Lüfter.

Offenbar ist das Modell in Abb. 3.2 für Regelungsanwendungen geeignet. Für andere Arten von Anwendungen kann es als eine erste Näherung genutzt werden. Im Folgenden beschreiben wir wichtige Hardwarekomponenten cyber-physikalischer Systeme entsprechend der Schleifenstruktur von Abb. 3.2.

3.2 Eingabe – Schnittstelle zwischen physischer und Cyber-Welt

3.2.1 Sensoren

Sensoren sind wesentlicher Teil des CyPhy-Interfaces. Sensoren können für fast jede existierende physikalische Größe entworfen werden. So gibt es z.B. Sensoren für Masse, Geschwindigkeit, Beschleunigung, elektrischen Strom, Spannung, Temperatur usw. Beim Bau von Sensoren kann dabei eine große Vielfalt von physikalischen Effekten ausgenutzt werden [151]. Dazu zählen das Induktionsgesetz (Erzeugung von Spannungen in einem elektrischen Feld) und photoelektrische Effekte. Zudem existieren auch Sensoren für chemische Substanzen [152].

In den vergangenen Jahren wurde eine große Anzahl an Sensorarten entwickelt. Ein großer Teil der Fortschritte beim Entwurf intelligenter Systeme ist der modernen Sensortechnologie zu verdanken. Die Verfügbarkeit von Sensoren ermöglichte den Entwurf von Sensornetzwerken, einem Schlüsselelement des Internets der Dinge (siehe beispielsweise Tiwari [542]). Es ist unmöglich, diese Teilmenge der Technologie cyber-physikalischer Systeme umfassend zu beschreiben. Wir stellen daher hier nur einige typische Beispiele dar:

- **Beschleunigungssensoren:** Abb. 3.3 zeigt einen kleinen Sensor, der mit Hilfe der Mikrosystemtechnologie gefertigt wurde. In seiner Mitte enthält dieser Sensor eine kleine Masse. Wenn er beschleunigt wird, entfernt sich die Masse von ihrer Ursprungslage und verändert dabei den Widerstand der kleinen Drähte, die mit der Masse verbunden sind.
  Beschleunigungssensoren sind u.a. in den leistungsfähigen sogenannten inertialen Messeinheiten (engl. *Inertial Measurement Units* (IMUs)) enthalten (siehe z.B. Siciliano et al. [487], Abschnitt 20.4). Sie enthalten Kreisel und Beschleunigungssensoren und können bis zu sechs Freiheitsgrade messen, darunter die aktuelle Position (x, y, und z) wie auch die Orientierung (Roll-, Nick- und Gier-

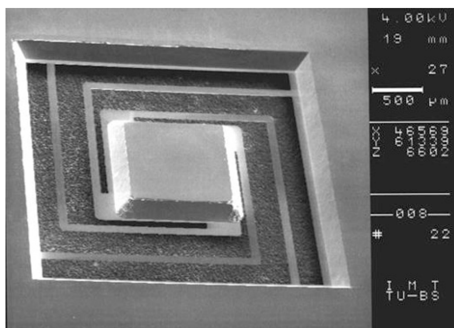


Abb. 3.3 Beschleunigungssensor (zur Verfügung gestellt von S. Büttgenbach, IMT, TU Braunschweig), ©TU Braunschweig

winkel) [580]. Sie sind in Robotern, Flugzeugen, Autos und anderen Produkten zum Zweck der Trägheitsnavigation enthalten.

- **Bildsensoren:** Es gibt zwei unterschiedliche Arten von Bildsensoren: Ladungstransportspeicher (engl. *Charge-Coupled Devices* (CCDs)) und **CMOS-Sensoren**. Beide verwenden gitterförmig angeordnete Lichtsensoren. Die Architektur von CMOS-Sensoren ähnelt der normaler Speicherbausteine: einzelne Pixel können beliebig adressiert und ausgelesen werden. CMOS-Sensoren verwenden dabei die Standard-CMOS-Technologie für integrierte Schaltungen. Damit lassen sich Sensoren und Logikschaltungen auf einem *Chip* integrieren. Dies ermöglicht „intelligente" Sensoren, bei denen Bildverarbeitungsschritte direkt auf dem Sensorchip ausgeführt werden können. CMOS-Sensoren benötigen nur eine einzige Standard-Versorgungsspannung und sind leicht in das übrige System zu integrieren. Aus diesem Grund sind CMOS-Sensoren preisgünstig.
  Im Gegensatz dazu ist die CCD-Technologie auf optische Anwendungen optimiert. Hier müssen Ladungen von einem Pixel zum nächsten übertragen werden, bis sie schließlich am Rand des Sensorgitters ausgelesen werden können. Dieser sequentielle Ladungstransfer ist auch der Namensgeber der CCDs. Die Anbindung von CCD-Sensoren ist vergleichsweise aufwendig.
  Die Auswahl des geeignetsten Bildsensors ist allerdings nicht so einfach. In den letzten Jahren wurde die Bildqualität von CMOS-Sensoren deutlich gesteigert und die qualitative Überlegenheit von CCDs ist in Frage zu stellen. Damit sind sowohl CCD- wie auch CMOS-Sensoren dazu in der Lage, eine gute Bildqualität zu liefern. Aufgrund ihrer höheren Lesegeschwindigkeit sind CMOS-Sensoren für Kameras mit elektronischem Sucher und Kameras mit Videoaufnahmemöglichkeit vorzuziehen [405]. Für preisgünstige Geräte wie auch für intelligente Sensoren werden CMOS-Sensoren ebenfalls bevorzugt eingesetzt. Viele Anwendungsbereiche für CCD-Sensoren sind verschwunden, aber für die wissenschaftliche Bildgewinnung (beispielsweise in Teleskopen) finden sie gegenwärtig weiterhin Anwendung.
- **Biometrische Sensoren:** Höhere Sicherheitsstandards haben zu einem verstärkten Interesse an Authentisierung geführt. Durch die Beschränkungen passwortbasierter Sicherheitsmethoden (z.B. durch gestohlene oder verlorene Passwörter) hat das Interesse an *Smartcards*, biometrischen Sensoren und biomedizinischer

Authentisierung deutlich zugenommen. Bei biometrischer Authentisierung wird zu erkennen versucht, ob eine bestimmte Person tatsächlich diejenige Person ist, als die sie sich ausgibt. Biometrische Authentisierungsmethoden sind z.B. Iris-*Scans*, Fingerabdrucksensoren und Gesichtserkennung. Falsche positive wie auch falsche negative Erkennungsvorgänge sind ein inhärentes Problem der biometrischen Authentisierung (siehe Definitionen auf Seite 281). Im Gegensatz zur passwortbasierten Authentisierung sind hier exakte Übereinstimmungen nicht möglich.

- **Künstliche Augen:** Projekte, die künstliche Augen entwickeln, haben nennenswerte Beachtung gefunden. Einige dieser Projekte versuchen, das Auge zu ersetzen. Andere Projekte versuchen, ein Sehen auf indirekten Wegen zu ermöglichen. Das Dobelle-Institut experimentierte beispielsweise mit einer kleinen Kamera, die mit einem Rechner verbunden war, der elektrische Impulse über einen direkten Kontakt an das Gehirn sandte [532]. In neueren Projekten wurde die Übersetzung von Bildern in Toninformationen bevorzugt. Offensichtlich ist diese Methode weit weniger invasiv.
- **RFID-Technik:** Die RFID-Technik (engl. *Radio Frequency Identification Technology*) basiert auf der Antwort eines ***Tags*** auf bestimmte Funksignale [227]. Das *Tag* besteht aus einer integrierten Schaltung und einer Antenne. Es stellt einem **RFID-Leser** seine eindeutige Identifikation zur Verfügung. Die maximal mögliche Entfernung zwischen *Tag* und Leser hängt dabei von der Art des *Tags* ab. Diese Technologie lässt sich überall da einsetzen, wo Objekte, Tiere oder Personen identifiziert werden sollen. Sie ist eine Schlüsseltechnologie für das Internet der Dinge.
- **Fahrzeugsensoren:** Heutige Autos verfügen über eine große Anzahl von Sensoren, welche das Fahren unterstützen sollen, wie beispielsweise Regensensoren, Reifensensoren, Abstandssensoren, Motorkontrollsensoren, usw.
- **Andere Sensoren:** Weiter gibt es Feuchtigkeits-, Gas- und andere Sensoren.

Möglicherweise muss Maschinelles Lernen [205, 188, 559, 453] genutzt werden, um sinnvolle Informationen aus gestörten Sensorwerten abzuleiten.

Sensoren erzeugen **Signale**. Dabei kann der Begriff „Signal" mathematisch wie folgt definiert werden:

Definition 3.1: Ein **Signal** σ ist eine Abbildung von einem Zeitbereich D_T auf einen Wertebereich D_V:

$$\sigma : D_T \rightarrow D_V$$

Signale können über einem kontinuierlichen oder einem diskreten Zeitbereich sowie über einem kontinuierlichen oder einem diskreten Wertebereich definiert sein.

3.2.2 Zeitdiskretisierung: Sample-and-hold-Schaltungen

Alle bekannten digitalen Computer arbeiten mit einem **diskreten** Zeitbereich D_T. Damit können sie diskrete Folgen oder **Ströme** (engl. *streams*) von Werten ver-

arbeiten. In einem kontinuierlichen Zeitbereich vorliegende Werte müssen daher in Werte über einem diskreten Zeitbereich konvertiert werden. Dies ist die Aufgabe von ***Sample-and-hold*-Schaltungen**. Diese sind ebenfalls wesentlicher Teil des **CyPhy-Interfaces**. Abb. 3.4 (links) zeigt eine einfache *Sample-and-hold*-Schaltung. Diese

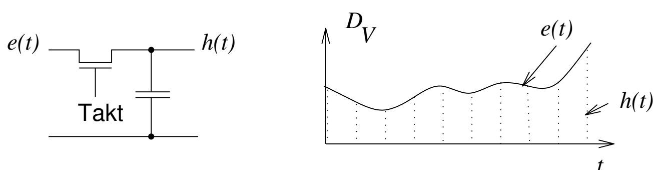


Abb. 3.4 Diskretisierung der Zeit: **links**: Schaltung; **rechts**: Signale

Schaltung besteht nur aus einem getakteten Transistor und einem Kondensator. Der Transistor arbeitet als Schalter. Wenn der Schalter durch das Taktsignal geschlossen wird, wird der Kondensator aufgeladen, sodass seine Spannung $h(t)$ praktisch der Eingangsspannung $e(t)$ entspricht. Nachdem der Schalter wieder geöffnet wurde, liegt diese Spannung so gut wie unverändert an, bis der Schalter wieder geschlossen wird. Jeder der im Kondensator gespeicherten Werte kann als ein Element einer diskreten Folge von Werten $h(t)$ angesehen werden, die aus einer kontinuierlichen Funktion $e(t)$ erzeugt wurden (siehe Abb. 3.4 (rechts)). Wenn wir nun $e(t)$ zu den Zeitpunkten $\{t_s\}$ abtasten, ist $h(t)$ nur zu diesen Zeitpunkten definiert.

Eine ideale *Sample-and-hold*-Schaltung könnte die Spannung am Kondensator in einer beliebig kurzen Zeitspanne ändern. Damit könnte die Eingangsspannung zu einem beliebigen Zeitpunkt auf den Kondensator übertragen werden und jedes Element der diskreten Folge entspräche der Eingangsspannung zu einem genau bestimmten Zeitpunkt. In der Praxis muss der Transistor aber für eine kurze Zeitspanne geschlossen bleiben, um den Kondensator wirklich laden oder entladen zu können. Während dieser Zeitspanne nähert sich die Kondensatorspannung wie bei einer üblichen Kondensatoraufladung der Eingangsspannung an.

3.2.3 Fourier-Approximation von Signalen

Können wir aus dem abgetasteten Signal $h(t)$ das ursprüngliche Signal $e(t)$ rekonstruieren? Hier möchten wir uns darauf beziehen, dass beliebige Signale durch die Summe von (möglicherweise phasenverschobenen) Sinusfunktionen unterschiedlicher Frequenzen approximiert werden können (Fourier-Approximation)[1].

[1] Die Darstellung in diesem Buch geht davon aus, dass eine umfassende Vorstellung der Fourier-Approximationstheorie nicht im Rahmen einer Vorlesung über eingebettete Systeme erfolgen kann.

Beispiel 3.1: Die Abbildungen 3.5 und 3.6 zeigen, dass selbst ein Rechtecksignal durch Sinussignale mit steigenden Frequenzen angenähert werden kann. Gleichung (3.1) ist die Basis der entsprechenden Approximation [440].

$$e'_K(t) = \sum_{k=1,3,5,7,9,...}^{K} \left(\frac{4}{\pi k} \sin(\frac{2\pi k t}{T}) \right) \tag{3.1}$$

In dieser Gleichung ist T die Periode und die Approximation wird mit steigendem K genauer. Abb. 3.5 und Abb. 3.6 visualisieren Gleichung (3.1). Der größe-

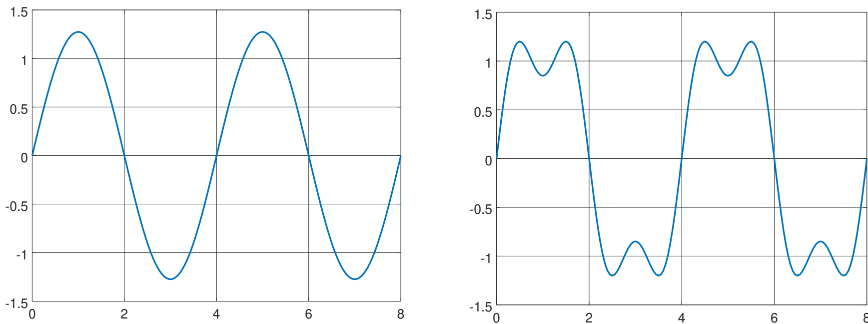


Abb. 3.5 Approximation eines Rechtecksignals durch Sinussignale für K=1 (**links**) und K=3 (**rechts**)

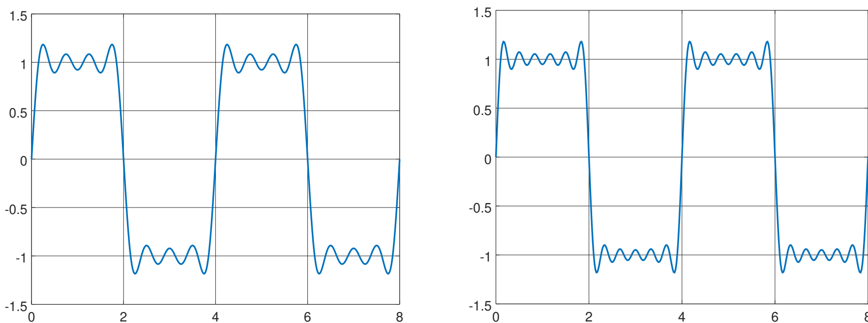


Abb. 3.6 Approximation eines Rechtecksignals durch Sinussignale für K=7 (**links**) and K=11 (**rechts**)

re Unterschied zwischen dem Rechtecksignal und seinen Approximationen an der Sprungstelle (sichtbar v.a. für K=11) heißt **Gibbs-Phänomen** [440]. ∇

Daher wird nur die Bedeutung der Theorie anhand einiger Beispiele erklärt. Es wäre für Studierende vorteilhaft, die Theorie hinter diesen Beispielen zu kennen (siehe z.B. http://www.dspguide.com).

Definition 3.2: Eine signalverarbeitende Transformation Tr wird als **linear** bezeichnet, wenn für die Signale $e_1(t)$ und $e_2(t)$ gilt:

$$Tr(e_1 + e_2) = Tr(e_1) + Tr(e_2) \tag{3.2}$$

Im Folgenden betrachten wir nur lineare Systeme. Um die oben gestellte Frage zu beantworten, betrachten wie die Auswirkung der Abtastung auf jedes einzelne Sinussignal.

Beispiel 3.2: Nehmen Sie an, unser Eingangssignal entspräche einer der beiden Funktionen e_3 oder e_4:

$$e_3(t) = \sin(\frac{2\pi t}{8}) + 0.5\sin(\frac{2\pi t}{4}) \tag{3.3}$$

$$e_4(t) = \sin(\frac{2\pi t}{8}) + 0.5\sin(\frac{2\pi t}{4}) + 0.5\sin(\frac{2\pi t}{1}) \tag{3.4}$$

Die in diesen Funktionen verwendeten Sinussignale haben Perioden von $T = 8, 4$ und 1 (dies ist leicht zu sehen, wenn man diese Sinussignale mit den Funktionen aus Gleichung (3.1) vergleicht). Eine graphische Darstellung dieser Funktionen findet sich in Abb. 3.7.

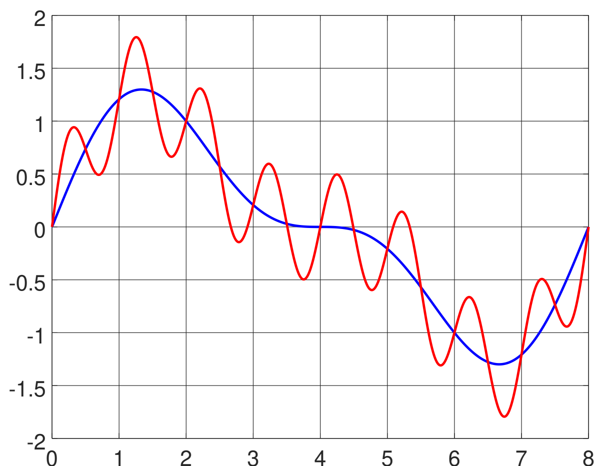


Abb. 3.7 Visualisierung der Funktionen $e_3(t)$ (blau) und $e_4(t)$ (rot)

Wir wollen diese Signale nun zu ganzzahligen Zeitpunkten abtasten. Zufälligerweise haben beide Signale dann jedes Mal, wenn sie abgetastet werden, denselben Wert. Es ist offenbar nicht möglich, zwischen $e_3(t)$ und $e_4(t)$ zu unterscheiden, wenn wir Werte zu den gezeigten Zeitpunkten abtasten und nur dieses abgetastete Signal verfügbar ist. ∇

Allgemein können wir anhand abgetasteter Signale nicht zwischen einem langsamen Signal $e_3(t)$ und einem sich schneller ändernden Signal $e_4(t)$ unterscheiden, wenn $e_3(t)$ und $e_4(t)$ zu den Abtastzeitpunkten identisch sind. Diese Tatsache, dass zwei oder mehr Ausgangssignale nach der Abtastung dieselbe Darstellung besitzen, bezeichnet man als ***Aliasing***. Wir tasten $e_4(t)$ nicht oft genug ab, um z.B. festzustellen, dass Steigungsänderungen zwischen den ganzzahligen Abtastzeitpunkten auftreten. Aus diesem Gegenbeispiel können wir folgern, dass **die Rekonstruktion des ursprünglichen Signals nicht machbar ist, solange wir keine zusätzlichen Informationen über die in dem Signal vorhandenen Frequenzen oder Wellenformen besitzen.**

Wie häufig müssen wir nun Signale abtasten, um zwischen zwei unterschiedlichen Sinuswellen unterscheiden zu können? Wir nehmen an, dass wir das Eingangssignal in konstanten Zeitabständen abtasten, sodass T_s die **Abtastperiode** ist:

$$\forall s : T_s = t_{s+1} - t_s \tag{3.5}$$

Sei

$$f_s = \frac{1}{T_s} \tag{3.6}$$

die **Abtastrate** oder **Abtastfrequenz**. Gemäß Abtasttheorie gilt [440]:

Theorem 3.1 (Abtasttheorem): *Für die o.a. Definition kann Aliasing vermieden werden, wenn Folgendes gilt:*

$$T_s < \frac{T_N}{2} \text{ mit } T_N : \text{ Periode des „schnellsten" Sinussignals bzw.} \tag{3.7}$$

$$f_s > 2 f_N \text{ mit } f_N : \text{ Frequenz des schnellsten Sinussignals} \tag{3.8}$$

Definition 3.3: f_N wird **Nyquist-Frequenz** genannt, f_s ist die **Abtastrate**.

Die Bedingung in Gleichung (3.8) wird **Abtastkriterium** oder manchmal auch **Nyquistsches Abtastkriterium** genannt.

Die Rekonstruktion von Eingangssignalen $e(t)$ aus diskreten Abtastwerten $h(t)$ kann also nur funktionieren, wenn wir sicherstellen, dass höhere Frequenzkomponenten, wie die in $e_4(t)$ vorkommende, entfernt wurden. Dies ist die Aufgabe von *Anti-aliasing*-Filtern. *Anti-aliasing*-Filter werden vor die *Sample-and-hold*-Schaltung geschaltet (siehe Abb. 3.8).

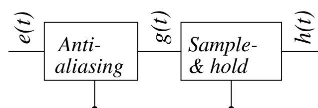


Abb. 3.8 Der *Sample-and-hold*-Schaltung vorgeschaltetes *Anti-aliasing*

Abb. 3.9 zeigt den **Amplitudengang**, d.h. das Verhältnis der Amplituden zwischen dem Ausgangs- und dem Eingangssignal als Funktion der Filterfrequenz.

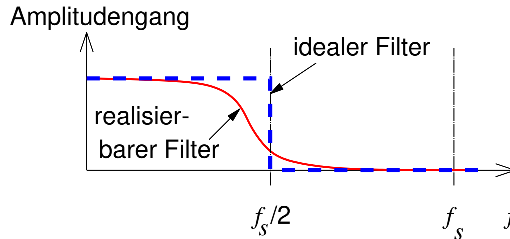


Abb. 3.9 Amplitudengänge idealer und realisierbarer *Anti-aliasing*-Filter (Tiefpassfilter)

Idealerweise würde solch ein Filter alle Frequenzen ab der halben Abtastfrequenz aufwärts entfernen und alle anderen Frequenzkomponenten unverändert lassen. Damit ließe sich das Signal $e_4(t)$ in das Signal $e_3(t)$ umwandeln.

In der Praxis existieren solche idealen Filter jedoch nicht[2]. Reale Filter dämpfen bereits Frequenzen unterhalb von $f_s/2$ und entfernen nicht alle Frequenzen oberhalb von $f_s/2$ (siehe Abb. 3.9). Gedämpfte hochfrequente Anteile sind daher auch nach der Filterung noch vorhanden. Für Frequenzen unterhalb von $f_s/2$ kann auch Überschwingen (engl. *overshooting*) auftreten, d.h. es existieren Frequenzen, für die das Eingangssignal verstärkt wird.

Der Entwurf guter *Anti-Aliasing*-Filter erfordert Spezialkenntnisse. Diese Kenntnisse werden beispielsweise beim Entwurf hochwertiger Audiogeräte eingesetzt, der üblicherweise Hörproben erfordert. Viele der empfundenen Unterschiede zwischen verschiedenen hochwertigen Audiogeräten werden dem Entwurf guter Filter zugeschrieben.

3.2.4 Wertediskretisierung: A/D-Wandler

Da wir nur digitale Computer betrachten, müssen wir auch Signale, die Zeit auf einen kontinuierlichen Wertebereich D_V abbilden, durch solche Signale ersetzen, welche die Zeit auf einen diskreten Wertebereich D'_V abbilden. Diese Wandlung von analogen zu digitalen Werten wird von Analog-Digital-Wandlern (A/D-Wandlern) realisiert. Es gibt viele Arten von A/D-Wandlern mit unterschiedlicher Geschwindigkeit und Genauigkeit. Typischerweise haben schnelle Wandler eine geringe Genauigkeit und genaue Wandler sind langsam.

In diesem Buch werden wir in den nächsten Unterabschnitten verschiedene Wandler-Typen vorstellen.

[2] Solche Filter würden voraussetzen, dass man das Signal für eine unbeschränkte Zeit kennt.

Flash-A/D-Wandler

Diese Art von A/D-Wandlern verwendet eine große Anzahl an Komparatoren. Jeder Komparator besitzt zwei Eingänge + und -. Wenn die Spannung am Eingang + die Spannung am Eingang - übersteigt, entspricht der Ausgang einer logischen `'1'`, ansonsten einer logischen `'0'`[3].

Alle - -Eingänge des A/D-Wandlers sind mit einem Spannungsteiler verbunden. Wenn die Eingangsspannung $h(t)$ den Wert $\frac{3}{4}V_{ref}$ übersteigt, erzeugt der oberste Komparator in Abb. 3.10 (links) eine `'1'`. Der an den Ausgängen der Komparatoren befindliche *Encoder* versucht, die höchstwertige `'1'` zu erkennen und kodiert diesen Fall als größtmöglichen Ausgabewert. Der Fall $h(t) > V_{ref}$ sollte normalerweise vermieden werden, da V_{ref} meist nahe der Versorgungsspannung der Schaltung liegt und Eingangsspannungen größer als die Versorgungsspannung zu elektrischen Problemen führen können. In unserem Fall erzeugen Eingangsspannungen größer als V_{ref} den größten digitalen Wert, solange der Konverter nicht wegen einer zu hohen Eingangsspannung ausfällt.

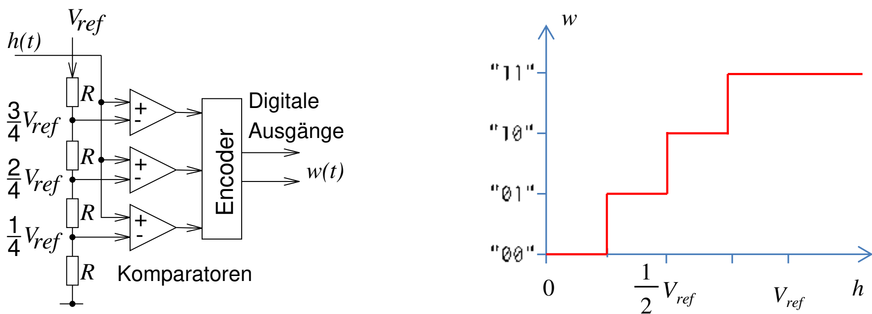


Abb. 3.10 *Flash*-A/D-Wandler: **links**: Schaltung; **rechts**: w als Funktion von h

Wenn nun der Wert der Eingangsspannung $h(t)$ kleiner als $\frac{3}{4}V_{ref}$, aber noch größer als $\frac{2}{4}V_{ref}$ ist, erzeugt der oberste Komparator in Abb. 3.10 eine `'0'`, wogegen der folgende Komparator immer noch eine `'1'` signalisiert. Der *Encoder* wird dies als den zweitgrößten Wert darstellen.

Entsprechend verhält es sich für die Fälle $\frac{1}{4}V_{ref} < h(t) < \frac{2}{4}V_{ref}$ und $0 < h(t) < \frac{1}{4}V_{ref}$, die als drittgrößter bzw. kleinster Wert dargestellt werden. Abb. 3.10 (rechts) zeigt das Verhältnis zwischen Eingangsspannungen und erzeugten digitalen Werten.

Die Ausgänge des Komparators stellen Werte auf eine besondere Art dar. Wenn der Ausgang eines bestimmten Komparators den Wert `'1'` hat, dann sind auch alle niederwertigen Ausgänge gleich `'1'`. Der *Encoder* übersetzt diese Zahlendarstellung in die gewohnte Darstellung natürlicher Zahlen. Dieser *Encoder* ist damit ein

[3] In der Praxis ist der Fall gleicher Spannungen nicht relevant, da das tatsächliche Verhalten für sehr kleine Spannungsunterschiede an den beiden Eingängen noch von vielen weiteren Faktoren abhängt (wie z.B. der Temperatur und dem Herstellungsprozess).

sogenannter „Prioritätsencoder”, der den höchstwertigen Eingang darstellt, der eine binäre '1' enthält[4].

Die Schaltung wandelt positive analoge Eingangsspannungen in digitale Werte um. Das Konvertieren sowohl positiver wie auch negativer Eingangsspannungen und die entsprechende Erzeugung von Zweierkomplementzahlen erfordert einige Erweiterungen.

Flash-A/D-Wandler sind automatisch **monoton**. Das bedeutet, dass für eine von 0 bis zum Maximum ansteigende Spannung die entsprechenden Digitalwerte auch garantiert ansteigen. Diese Eigenschaft bleibt selbst dann erhalten, wenn der reale Widerstandswert vom nominellen abweichen sollte. Allerdings hätte eine solche Abweichung einen Einfluss auf die Genauigkeit der Wandlung.

Leider bildet die Widerstandskette einen elektrisch leitenden Pfad selbst in solchen Phasen, in denen der Wandler unbenutzt ist. Deshalb ist der Wandler in der Regel nicht für stromsparende Schaltungen geeignet.

Allgemein werden A/D-Wandler durch ihre **Auflösung** gekennzeichnet. Dieser Begriff hat verschiedene miteinander verwandte Bedeutungen [15]. Die Auflösung (gemessen in Bits) ist die Anzahl an Bits, die ein A/D-Wandler erzeugt. Beispielsweise werden für viele Audioanwendungen A/D-Wandler mit einer Auflösung von 16 Bit benötigt. Die Auflösung wird aber auch in Volt angegeben, in diesem Fall kennzeichnet sie den Abstand, der zwischen zwei Eingangsspannungen vorhanden sein muss, damit die Ausgabe um 1 inkrementiert wird:

$$Q = \frac{V_{FSR}}{n} \tag{3.9}$$

mit: Q : Auflösung in Volt pro Schritt,

V_{FSR} : Differenz zwischen größter und kleinster Spannung und

n : Anzahl der Spannungsintervalle (**nicht** Anzahl der Bits).

Beispiel 3.3: Der A/D-Wandler aus Abb. 3.10 besitzt eine Auflösung von 2 Bit oder $\frac{1}{4}V_{ref}$ Volt, wenn wir V_{ref} als größten Wert annehmen. ∇

Der entscheidende Vorteil von *Flash*-A/D-Wandlern ist ihre Geschwindigkeit. Sie benötigen keinerlei Takt. Die Verzögerung zwischen Eingabe und Ausgabe ist gering und die Schaltung kann z.B. einfach für Hochgeschwindigkeits-Videoanwendungen eingesetzt werden. Der Nachteil ist die Komplexität der Hardware: wir benötigen $n-1$ Komparatoren, um zwischen n Werten unterscheiden zu können. Stellen Sie sich vor, diese Schaltung würde verwendet, um digitale Audiosignale für CD-Recorder zu erzeugen. Wir würden $2^{16}-1$ Komparatoren benötigen! Hochauflösende A/D-Wandler müssen daher anders konstruiert sein.

[4] Solche *Encoder* sind auch dabei nützlich, die höchstwertige '1' in der Mantisse einer Gleitkommazahl zu finden.

Sukzessive Approximation (Wägeprinzip)

Hochauflösende Wandler sind mit der sukzessiven Approximation möglich. Die Schaltung dafür ist in Abb. 3.11 dargestellt.

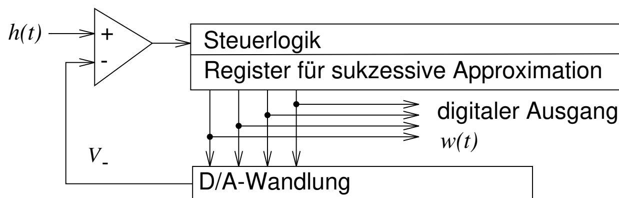


Abb. 3.11 Schaltung, die sukzessive Approximation verwendet

Die zugrunde liegende Idee ist hier die Verwendung von binärer Suche. Zu Anfang ist das höchstwertige Ausgangsbit des sukzessiven Approximations- Registers auf `'1'` und alle anderen Bits auf `'0'` gesetzt. Dieser digitale Wert wird dann in eine analoge Spannung gewandelt, die der Hälfte der maximalen Eingangsspannung entspricht[5]. Wenn $h(t)$ größer als der erzeugte analoge Wert ist, wird die `'1'` im höchstwertigen Bit beibehalten, ansonsten wird das Bit auf `'0'` gesetzt.

Dieser Vorgang wird für das nächste Bit wiederholt. Es bleibt auf `'1'`, wenn der Eingabewert entweder im zweiten oder vierten Viertel des Eingangsspannungsbereiches liegt. Der Vorgang wird dann für alle übrigen Bits ebenfalls wiederholt.

Abb. 3.12 zeigt ein Beispiel. Anfangs ist das höchstwertige Bit auf `'1'` gesetzt. Dieser Wert wird beibehalten, da die resultierende Spannung V_- kleiner als $h(t)$ ist. Anschließend wird das nächste Bit auf `'1'` gesetzt. Dieses wird zu `'0'` zurückgesetzt, da die resultierende Spannung V_- größer als $h(t)$ ist. Daraufhin werden die weiteren niederwertigen Bits verglichen. Offensichtlich muss $h(t)$ während der Umwandlung konstant bleiben. Diese Anforderung lässt sich durch die Verwendung der oben beschriebenen *Sample-and-hold*-Schaltung erfüllen. Das resultierende digitale Signal wird mit $w(t)$ bezeichnet.

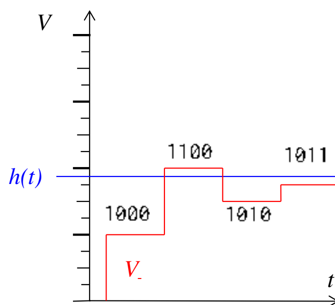


Abb. 3.12 Sukzessive Approximation

Der wesentliche Vorteil der sukzessiven Approximation liegt in ihrer Hardwareeffizienz. Es werden nur $\lceil log_2(n) \rceil$ Bits im sukzessiven Approximations-Register sowie der D/A-Wandler benötigt, um zwischen n digitalen Werten zu unterscheiden. Der Nachteil ist die geringe Geschwindigkeit, da $O(log_2(n))$ Schritte benötigt werden. Diese Wandler sind daher für Anwendungen geeignet, die eine hohe Auflösung bei mittleren Geschwindigkeiten benötigen. Beispiele hierfür sind Audioanwendungen.

---

[5] Die Wandlung von digitalen zu analogen Werten (D/A-Wandlung) kann sehr effizient implementiert werden und ist sehr schnell (siehe Seite 197).

Fließband-A/D-Wandler

Diese Wandler bestehen aus einer Folge von Stufen, von denen jede für die Wandlung von einigen wenigen Bits zuständig ist (siehe Abb. 3.13). Jede Stufe leitet die

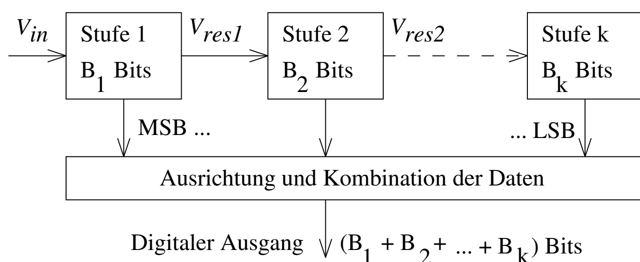


Abb. 3.13 Fließband-A/D-Wandler [292]

Differenz (den nicht gewandelten Rest) zwischen dem Eingangswert und der analogen Darstellung des erzeugten Digitalwertes an die nächste Stufe (sofern vorhanden) weiter. Beispielsweise könnte jede Stufe ein einzelnes Bit wandeln und die entsprechende Spannung von der Eingangsspannung abziehen. Die resultierende Spannung würde typischerweise wieder mit einem Faktor von zwei skaliert werden (um kleine Spannungen zu vermeiden) und an die nächste Stufe weitergeleitet werden. Üblicherweise enthält jede Stufe einen *Flash*-A/D-Wandler für einige wenige Bits und einen D/A-Wandler zur Berechnung der zu subtrahierenden Spannung. Die resultierenden Digitalwerte müssen zeitlich korrekt zugeordnet werden. Der Hardwareaufwand für diese Wandlung wächst linear mit der Anzahl der Bits. Es kann ein guter Durchsatz erzielt werden, aber die Latenzzeit ist größer als für *Flash*-A/D-Wandler.

Andere Wandler

Integrierende Wandler benutzen (mindestens) zwei Phasen für die Messung. Während der ersten Phase der Länge t_1 wird das Integral der Eingangsspannung über der Zeit berechnet[6]. Für konstante Eingangsspannungen ist der resultierende Wert V_{out} proportional zur Eingangsspannung ($V_{out} \sim V_{in} * t_1$). Während der zweiten Phase wird dieser Wert mit einer konstanten Rate reduziert und es wird mit einem Zähler gemessen, wie lange es dauert, bis der Wert 0 erreicht wird. Der erreichte Zählerstand ist proportional zur Eingangsspannung. Ein Vorteil der Schaltung liegt darin, dass ein ggf. vorhandenes Eingangsrauschen während der Integrationsphase mit hoher Wahrscheinlichkeit herausgemittelt wird. Aufgrund dieser Unterdrückung

[6] Dies kann mit einem Kondensator in der Rückkopplungsschleife eines Operationsverstärkers (siehe Seite 429) gemacht werden.

von Eingangsrauschen eignet sich dieses Prinzip der Wandlung v.a. für Wandler einer hohen Genauigkeit, wie sie z.B. in Multimetern zu finden sind.

Bei **faltenden A/D-Wandlern** wird die Eingangsspannung in 2^m Segmente aufgeteilt [100, 322]. Ein grobgranularer Wandler bestimmt das Segment der jeweils vorhandenen Eingangsspannung und bestimmt damit die signifikantesten m Bits. Ein feingranularer Wandler bestimmt den Wert innerhalb eines Segments und berechnet so die weniger signifikanten Ausgabebits.

Bei **Delta-Sigma A/D-Wandlern** werden, wie der Name es vermuten lässt, Signaldifferenzen (Δs) kodiert und aufsummiert (Σ). Eine Beschreibung dieser Wandler geht über den Themenkreis dieses Buches hinaus. Interessierte Leser können beispielsweise bei Khorramabadi [293] nachschlagen.

Vergleich von A/D-Wandlern

Abb. 3.14 bietet einen Überblick über die Beziehungen zwischen der Auflösung und der Geschwindigkeit verschiedener Typen von A/D-Wandlern auf der Basis einer

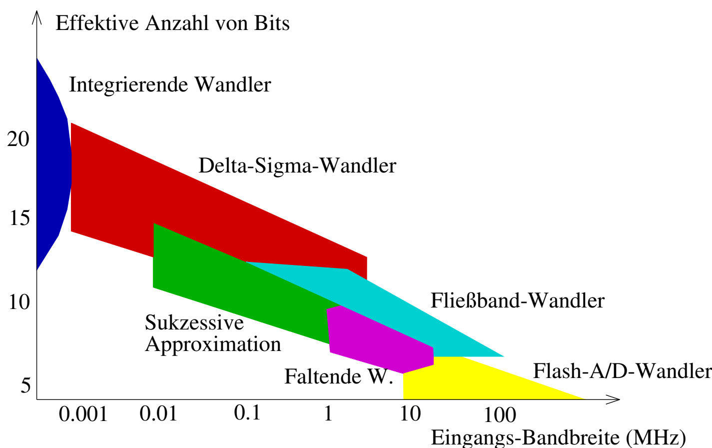


Abb. 3.14 Vergleich der Geschwindigkeiten und der Auflösung von A/D-Wandlertypen [557]

Analyse von Vogels et al. [557]. Danach sind *Flash*-A/D-Wandler offensichtlich die schnellsten, aber sie bieten nur eine begrenzte Auflösung. Fließband-A/D-Wandler sind häufig der sukzessiven Approximation überlegen. IEEE TV [437] bietet eine weitere Übersicht.

Quantisierungsrauschen

Abb. 3.15 zeigt das Verhalten des *Flash*-A/D-Wandlers für ein Eingangssignal gemäß Gleichung (3.3). Nur das Verhalten für ein positives Eingangssignal ist gezeigt. Die

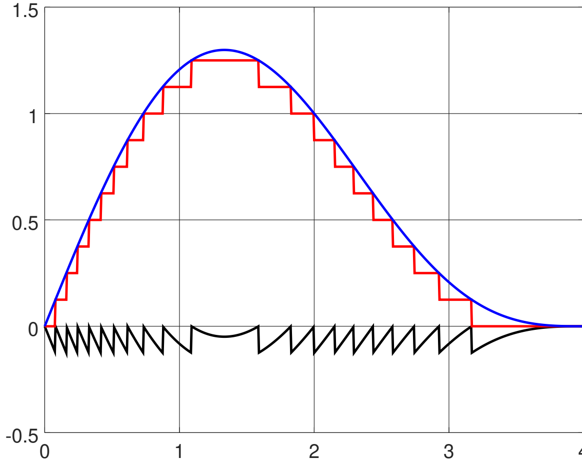


Abb. 3.15 $h(t)$ (blau), $w(t)$ (rot), $w(t) - h(t)$ (schwarz)

Abbildung enthält die Spannung, die dem bestimmten Digitalwert entspricht, die ursprüngliche Spannung und die Differenz zwischen beiden. Offensichtlich **rundet** der Konverter die digitale Darstellung entsprechend der Anzahl der verfügbaren Bits **ab** (d.h. der Digitalwert entspricht immer einer Spannung kleiner oder gleich der Eingangsspannung). Dies ist eine Konsequenz aus der Art und Weise, wie der Konverter Vergleiche durchführt. Rundende Konverter würden eine Korrektur um ein halbes Bit benötigen. Tatsächlich entspricht das Digitalsignal einem Analogwert der Summe aus dem Originalsignal und der Differenz $w(t) - h(t)$. Das heißt, es sieht so aus, als ob die Differenz zwischen den beiden Signalen auf das Eingangssignal aufaddiert wird. Diese Differenz heißt **Quantisierungsrauschen**:

Definition 3.4: Sei $h(t)$ ein Analogsignal. Sei $w(t)$ von $h(t)$ durch Quantisierung abgeleitet. Die Differenz zwischen beiden heißt **Quantisierungsrauschen**:

$$\text{Quantisierungsrauschen}(t) = w(t) - h(t) < Q \tag{3.10}$$

Definition 3.5: Das Quantisierungsrauschen lässt sich offensichtlich reduzieren, indem die Auflösung (in Bit) der A/D-Wandler erhöht wird. Der Einfluss des Quantisierungsrauschens wird meist in der Definition des Signal-Rausch-Abstands (engl. *Signal-to-Noise Ratio* (SNR)) festgehalten. Der Signal-Rausch-Abstand wird in Dezibel gemessen (Zehntel eines Bel, benannt nach Alexander G. Bell):

$$\text{SNR (in dB = Dezibel)} = 10 * log \frac{\text{Leistung des Nutzsignals}}{\text{Leistung des Rauschsignals}} \quad (3.11)$$

$$= 20 * log \frac{\text{Spannung des Nutzsignals}}{\text{Spannung des Rauschsignals}} \quad (3.12)$$

In diesem Fall haben wir ausgenutzt, dass die Energie eines Signals für jede beliebige Impedanz R gleich dem Quadrat der Spannung ist. Dezibel ist keine physikalische Einheit, da der Signal-Rausch-Abstand dimensionslos ist.

Für jedes Signal $h(t)$ beträgt die Energie des Quantisierungsrauschens $\alpha \cdot Q$, wobei $\alpha \leq 1$ von der Wellenform von $h(t)$ abhängt. Wenn $h(t)$ stets exakt durch einen digitalen Wert wiedergegeben werden kann, beträgt der Wert $\alpha = 0$. Wenn $h(t)$ andererseits immer „ein wenig" unterhalb des nächsten darstellbaren Wertes liegt, kann α nahe an 1 liegen.

Beispiel 3.4: Der Signal-Rausch-Abstand von 16-Bit CD-Audio (mit $\alpha \approx 1$) liegt in der Größenordnung von: $20 * log(2^{16}) = 96$ dB. Für qualitativ hochwertige 24-Bit CDs beträgt der Signal-Rausch-Abstand rund 144 dB. Werte von $\alpha < 1$ und Fertigungstoleranzen von A/D-Wandlern können diese Werte allerdings noch beeinflussen. ∇

3.3 Verarbeitungseinheiten

In diesem Abschnitt betrachten wir das nächste Element in der Regelschleife nach Abb. 3.2, nämlich Verarbeitungseinheiten. Für die Datenverarbeitung in eingebetteten Systemen betrachten wir anwendungsspezifische integrierte Schaltkreise (engl. *Application-Specific Integrated Circuit* (ASICs)), rekonfigurierbare Logik und verschiedene Typen von programmierbaren Prozessoren. Wir starten mit ASICs.

3.3.1 Anwendungsspezifische integrierte Schaltkreise

Für Hochleistungsanwendungen und große Stückzahlen kann es sich lohnen, dass eine anwendungsspezifische Schaltung entworfen wird. Allgemein sind ASICs sehr energieeffizient (siehe Unterabschnitt 3.7.3 auf Seite 211). Allerdings fallen hohe Kosten für den Entwurf und die Fertigung solcher *Chips* an. Die Kosten für einen Satz von Masken, mit denen die geometrischen Muster auf den *Chip* übertragen werden, sind mit zunehmender Miniaturisierung stark gestiegen[7].

Allerdings kann man diese Kosten durch die Verwendung weniger fortgeschrittener Fabrikationstechnologie und durch die Nutzung von Multi-Projekt-*Wafern* (MPW), die mehrere Schaltkreise enthalten, senken. Der Einsatz von ASICs leidet

[7] 2017 lagen die durchschnittlichen Kosten gemäß http://anysilicon.com/semiconductor-wafer-mask-costs/ für eine fortgeschrittene 28 nm-Technologie bei etwa 1,5M $.

unter langer Entwurfsdauer und einem Mangel an Flexibilität: um Entwurfsfehler zu korrigieren, wird meist ein neuer Satz Masken und ein erneuter Herstellungslauf benötigt. Daher sind ASICs nur dann angemessen, wenn spezielle Umstände vorliegen, wie beispielsweise der Bedarf an großen Stückzahlen, maximale Energieeffizienz, spezielle Spannungs- oder Temperaturbereiche, gemischte Analog-/Digitalschaltungen oder hohe Sicherheitsanforderungen. Daher behandeln wir den Entwurf von ASICs in diesem Buch nicht weiter.

3.3.2 Prozessoren

Der Hauptvorteil von Prozessoren ist ihre Flexibilität. Wenn Prozessoren eingesetzt werden, kann das gesamte Verhalten eines eingebetteten Systems alleine durch die Änderung der darauf laufenden Software verändert werden. Verhaltensänderungen können erforderlich sein, um Entwurfsfehler zu korrigieren, das System auf einen neuen oder geänderten Standard zu aktualisieren oder um dem existierenden System neue Eigenschaften hinzuzufügen. Auf diesen Gründen haben Prozessoren, insbesondere kommerziell „aus dem Regal" (engl. *Commercial Off-The-Shelf* (COTS)) verfügbare, eine weite Verbreitung gefunden.

Eingebettete Prozessoren müssen schonend mit Ressourcen umgehen, d.h. wir müssen uns darum kümmern, welche Ressourcen zur Ausführung von Anwendungen erforderlich sind. Dabei müssen sie aber nicht kompatibel zum Instruktionssatz der in PCs oder Servern verwendeten Prozessoren sein. Dadurch kann ihre Architektur von diesen Prozessoren abweichen. Die Ressourceneffizienz besitzt eine Vielzahl von Aspekten (siehe Seite 14), die als nächstes auszugsweise diskutiert werden sollen.

Energieeffizienz

Die Energie E zur Ausführung einer bestimmten Anwendung hängt eng mit der (elektrischen) Leistung $P(t)$ als Funktion der Zeit zusammen, denn es gilt Gleichung (3.13).

$$E = \int P(t)dt \tag{3.13}$$

Wir nehmen an, dass wir mit einem Entwurf mit einem Leistungsverbrauch von $P_0(t)$ starten, woraus sich nach einer Ausführungszeit von t_0 ein Energieverbrauch von

$$E_0 = \int_0^{t_0} P_0(t)dt$$

ergibt. Angenommen, ein modifizierter Entwurf führt die Applikation bereits in t_1 Zeiteinheiten aus und benötigt dabei eine Leistung $P_1(t)$. Für diesen ergibt sich ein Energieverbrauch von

$$E_1 = \int_0^{t_1} P_1(t)dt$$

Wenn $P_1(t)$ nicht viel größer ist als $P_0(t)$, dann bedeutet die Reduktion der Ausführungszeit auch eine Reduktion des Energieverbrauchs. Allerdings bedeutet eine Reduktion der Zeit nicht unbedingt auch eine Reduktion der Energie. Dies ist in Abb. 3.16 gezeigt: E_1 kann kleiner sein als E_0, aber auch größer. Deswegen ist es zur Minimierung des Energieverbrauchs notwendig, auch diesen explizit als Optimierungskriterium zu benutzen. Eine Minimierung der Laufzeit kann nicht als Ersatz dienen.

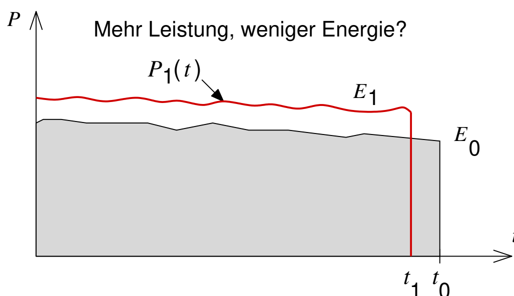


Abb. 3.16 Vergleich von Energien E_0 und E_1

Sowohl die Minimierung der Leistung wie auch die der Energie sind wichtig. Die Leistung hat einen Einfluss auf die Dimensionierung der Stromversorgung, den Entwurf der Spannungsregler, die Dimensionierung der Leitungen und die kurzfristige Kühlung. Die Minimierung des Energieverbrauchs ist insbesondere für mobile Anwendungen wichtig, da die Batterietechnologie nur langsam Fortschritte macht und da die Kosten für Energie sehr hoch sein können. Der Energieverbrauch hat auch einen Einfluss auf die Erzeugung von CO_2, wobei mobile Anwendungen aufgrund der begrenzten Batteriekapazitäten in der Regel weniger CO_2 produzieren als stationäre Anwendungen. Ein kleiner Energieverbrauch reduziert auch die Kosten für die Kühlung und erhöht die Lebensdauer des Systems (weil diese in der Regel mit der Temperatur sinkt).

Als nächstes soll gezeigt werden, dass es für CMOS-Technologie von Vorteil ist, schnelle sequentielle Berechnungen durch langsamer ausgeführte parallele Berechnungen zu ersetzen. Dazu betrachten wir zunächst die Leistungsaufnahme P von CMOS-Schaltungen. Die sogenannte **dynamische Leistungsaufnahme** ist die Leistungsaufnahme, die aufgrund des Schaltens von Transistoren und des Umladens von Leitungen entsteht (im Gegensatz zur statischen Leistungsaufnahme, die selbst ohne Schaltvorgänge entsteht). Die durchschnittliche dynamische Leistungsaufnahme P_{dyn} von CMOS kann gemäß Gleichung (3.14) modelliert werden [90]:

$$P_{dyn} = \alpha \; C_L \, V_{dd}^2 \, f \tag{3.14}$$

wobei α die Schaltaktivität, C_L die Ladekapazität, V_{dd} die Versorgungsspannung und f die Taktfrequenz darstellt. Dies bedeutet, dass die Leistungsaufnahme von CMOS-Prozessoren (mindestens) quadratisch mit der Betriebsspannung anwächst[8].

Die Verzögerung von CMOS-Schaltungen lässt sich annähern als [90]

$$\Delta = kC_L \frac{V_{dd}}{(V_{dd} - V_t)^2} \tag{3.15}$$

Dabei ist k eine Konstante und V_t die Schwellenspannung. V_t beeinflusst die Eingangsspannung, die erforderlich ist, um den Transistor einzuschalten. Bei einer maximalen Versorgungsspannung von $V_{dd,max}$=3,3 Volt könnte V_t in der Größenordnung von 0,8 Volt liegen. Folglich ist die maximale Taktfrequenz eine Funktion der Versorgungsspannung. Durch Absenken der Versorgungsspannung nimmt aber die Leistung quadratisch ab, wogegen die Laufzeit von Algorithmen nur linear steigt (wenn man die Einflüsse des Speichersystems nicht berücksichtigt).

Dieser Effekt wird ausgenutzt, um den Energieverbrauch durch eine bestimmte Menge an Operationen zu reduzieren. Dazu nehmen wir an, dass wir zunächst Operationen mit einer Spannung V_{dd}, einer konstanten Leistung P, einer Taktfrequenz f und einer Laufzeit t sequentiell ausführen und dabei eine elektrische Energie $E = P * t$ verbrauchen.

Wir stellen uns nun vor, dass wir dazu übergehen, β Operationen parallel auszuführen. Aufgrund der parallelen Ausführung können wir die Zeit zur Ausführung einer Operation jetzt um den Faktor β strecken. Damit können wir auch die Frequenz um denselben Faktor reduzieren und kommen zu einer neuen Taktfrequenz

$$f' = \frac{f}{\beta} \tag{3.16}$$

Dies erlaubt es uns, auch die Spannung auf eine neue Spannung zu reduzieren:

$$V'_{dd} = \frac{V_{dd}}{\beta} \tag{3.17}$$

Damit verringert sich die Leistung pro Operation quadratisch:

$$P^0 = \frac{P}{\beta^2} \tag{3.18}$$

Da wir β Operationen parallel ausführen, gehen wir aus von einer neuen Leistung von

$$P' = \beta * P^0 = \frac{P}{\beta} \tag{3.19}$$

Die neue Ausführungszeit t' ist aufgrund der Paralelausführung gleich der alten Zeit t, d.h. $t' = t$. Mithin ergibt sich die neue Energie als

[8] In der Praxis kann der Anstieg noch darüber liegen.

$$E' = P' * t = \frac{E}{\beta} \tag{3.20}$$

Es ist offenbar energieeffizienter, β Operationen parallel (und jede einzelne relativ langsam) auszuführen, anstatt sie sequentiell (und jede einzelne Operation relativ schnell) auszuführen. Allerdings enthält unsere Rechnung einige Annahmen und Näherungen. Auf der einen Seite gibt es Gründe für eine noch größere Energieeinsparung. Wenn es uns gelingt, die Leistung konstant zu halten anstatt sie gemäß Gleichung (3.19) durch die Parallelausführung zu erhöhen, so würde die Energie sogar quadratisch abnehmen. Sie würde ebenfalls stärker als linear abnehmen, wenn – wie dies experimentell beobachtet wurde – die Leistung sogar kubisch von der Spannung abhängt. Außerdem haben wir ignoriert, dass die Speichergeschwindigkeit häufig der begrenzende Faktor ist. Eine höhere Taktrate kann mithin bedeuten, dass der Prozessor „mit schnellerem Takt auf den Speicher wartet" und eine geringere Taktrate also die Ausführung gar nicht um den Faktor β verlangsamt. Auf der anderen Seite müssen wir β Operationen finden, die wir parallel ausführen können. Insgesamt können wir festhalten, dass parallele Ausführung ein Mittel zur energieeffizienten Berechnung ist, unabhängig von der konkreten Hardwarerealisierung.

Hardwarearchitekturen müssen in Bezug auf die Energieeffizienz optimiert werden und wir müssen sicherstellen, dass wir in der Softwareerzeugung keine Effizienz verlieren.

Es gibt eine große Anzahl von Techniken, mit denen die Energieeffizienz von Prozessoren gesteigert werden kann. Energieeffizienz sollte dabei auf verschiedenen Abstraktionsebenen betrachtet werden, vom Entwurf des Befehlssatzes bis hinab zum Entwurf der eigentlichen Chipfertigung [77]. Taktabschaltung (engl. *clock gating*) ist ein Beispiel für eine solche Technik. Die Taktabschaltung wird verwendet, um aktuell nicht benötigte Teile eines Prozessors vom Takt zu trennen. So werden beispielsweise Hardwareeinheiten für direkten Speicherzugriff (engl. *Direct Memory Access* (DMA)) oder Busbrücken nicht mit Takt versorgt, wenn sie nicht benötigt werden. Zudem existieren Ansätze, um den größten Teil eines Prozessors komplett taktlos betreiben zu können. Dabei gibt es zwei gegensätzliche Ansätze: global asynchrone, aber lokal synchrone Prozessoren (GALS) [263] und global synchrone, aber lokal asynchrone (GSLA) [117]. Weitere Informationen über stromsparende Entwurfstechniken finden sich in einem Buch von E. Macii [361] sowie in den PATMOS-Tagungsbänden (siehe http://www.patmos-conf.org/).

Auf einer vergleichsweise hohen Abstraktionsebene lassen sich mindestens drei Techniken einsetzen:

- **Parallele Ausführung:** Aufgrund von Gleichung (3.20) ist die parallele Ausführung ein gutes Mittel, um die Energieeffizienz zu erhöhen.
- **Dynamisches *Power Management* (DPM):** Bei dieser Methode verfügen Prozessoren zusätzlich zum normalen Betriebszustand über verschiedene weitere Energiesparzustände. Jeder dieser Zustände bringt eine unterschiedliche Leistungsaufnahme mit sich und benötigt unterschiedlich viel Zeit, um den Prozessor wieder in den normalen Betriebszustand zu überführen. Abb. 3.17 zeigt die drei Zustände des StrongARM SA1100-Prozessors.

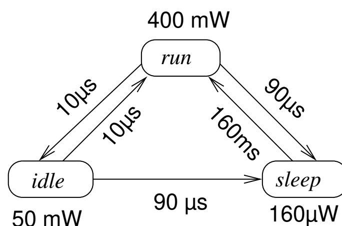


Abb. 3.17 DPM-Zustände des StrongArm-Prozessors SA1100 [45]

Der Prozessor ist im *run*-Zustand voll betriebsbereit. Im *idle*-Zustand überwacht er nur die *Interrupt*-Eingänge. Im *sleep*-Zustand wird der Prozessor zurückgesetzt und die Stromversorgung des *Chips* abgeschaltet [593]. Eine separate Stromversorgung für die Ein/Ausgabeeinheiten liefert Strom für die Energieverwaltungshardware. Der Prozessor kann durch ein vorher festgelegtes Aufweckereignis durch die Energieverwaltung neu gestartet werden. Beachtenswert ist hier der große Unterschied im Energieverbrauch zwischen dem *sleep*-Zustand und den anderen Zuständen, ebenso die große Zeitdauer, die benötigt wird, um vom *sleep*-zum *run*-Zustand umzuschalten.

- **Dynamische Spannungsskalierung:** Gleichung (3.14) wird von der sogenannten **dynamischen Spannungsskalierung** (engl. *Dynamic Voltage Scaling* (DVS)) ausgenutzt. So verfügte beispielsweise der Crusoe™-Prozessor von Transmeta [296] über 32 Spannungsstufen zwischen 1,1 und 1,6 Volt, der Takt konnte in 33 MHz-Schritten von 200 MHz bis 700 MHz variiert werden. Ein Übergang von einer Spannungs-/Frequenzkombination zur nächsten benötigte rund 20 ms. Entwurfsfragen für DVS-unterstützende Prozessoren sind in einer Veröffentlichung von Burd und Brodersen [76] beschrieben. Die Intel *SpeedStep*®-Technologie hat im Jahr 2004 sechs verschiedene Spannungs-/Frequenzkombinationen bei Pentium™M-Prozessoren realisiert [247]. Aktuelle Prozessoren erhalten umfangreichere Mechanismen zur Beeinflussung der Leistungsaufnahme.

Codegrößen-Effizienz

Für eingebettete System ist die Minimierung der Codegröße sehr wichtig, da diese nur selten über mechanische oder Festkörper-Laufwerke verfügen und die verfügbare Speicherkapazität meist auch sehr beschränkt ist[9]. Dies wird bei ***Systems-on-a-Chip*** (SoCs) um so deutlicher. SoCs implementieren Speicher und Prozessoren auf demselben *Chip*. In diesem besonderen Fall wird der Speicher auch **eingebetteter Speicher** genannt. Eingebetteter Speicher kann in der Herstellung teurer als separate Speicherchips sein, da die Herstellungsprozesse für Speicher und Prozessoren kompatibel sein müssen. Dennoch kann ein großer Anteil der gesamten Chipfläche

[9] Die Verfügbarkeit großer *Flash*-Speicher und die 3D-Integration lockern die Speicherplatzbeschränkungen ein wenig.

durch Speicher belegt sein. Es gibt verschiedene Techniken, um die Codegröße zu reduzieren:

- **CISC-Maschinen:** Standard RISC-Prozessoren wurden für maximale Geschwindigkeit und nicht für effiziente Codegrößen entworfen. Dagegen wurden frühere Prozessoren mit komplexen Befehlen (engl. *Complex Instruction Set Computers* (CISC)) auf effiziente Codegrößen hin optimiert, da sie nur über langsame Speicher verfügen konnten. Caches kamen dabei meist nicht zum Einsatz. Daher werden „altmodische" CISC-Prozessoren gerne in eingebetteten Systemen eingesetzt. Ein Beispiel dafür sind die ColdFire-Prozessoren [170], die auf der 68000er-Familie von CISC-Prozessoren der Firma Motorola basieren.
- **Komprimierungstechniken:** Durch das Ablegen von Befehlen in komprimierter Form im Hauptspeicher lassen sich die Menge an Silizium, die zum Speichern der Befehle notwendig ist, und auch die Energie, die zum Laden dieser Befehle benötigt wird, reduzieren. Durch die gesunkene Anforderung an die Bandbreite kann das Laden von Befehlen auch schneller erfolgen. Ein (hoffentlich schneller und kleiner) Dekodierer wird zwischen den Prozessor und den Befehlsspeicher geschaltet, um während des Ladens die ursprünglichen Befehle zu erzeugen (siehe Abb. 3.18 (rechts))[10]. Anstelle eines großen Speichers voller unkomprimierter Befehle speichern wir die Befehle nun in einem komprimierten Format.

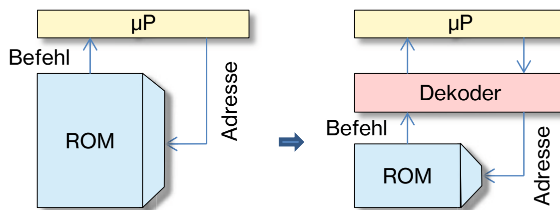


Abb. 3.18 Holen von Befehlen: **links**: unkomprimiert; **rechts**: komprimiert

Die durch Komprimierung zu erreichenden Ziele umfassen also:

- Wir möchten ROM- und RAM-Bereiche einsparen, da diese teurer als der eigentliche Prozessor sein können.
- Wir möchten eine Kodierungstechnik für Befehle und möglicherweise auch für Daten mit folgenden Eigenschaften verwenden:
  – Der Aufwand zur Laufzeit soll gering sein.
  – Die Dekodierung sollte mit einem begrenzten Kontext möglich sein (es ist beispielsweise nicht möglich, das gesamte Programm einzulesen, um das Ziel eines Sprungbefehls zu finden).

[10] Wir stellen die Funktion von Multiplexern, arithmetische Einheiten und Speichern weiterhin anhand ihrer Form (und nicht durch Beschriftung) dar, da diese Darstellung in technischer Dokumentation weit verbreitet ist. Für Speicher führen wir Symbole ein, die einen expliziten Adress-Dekodierer enthalten (in den Symbolen für ROMs auf der rechten Seite zu sehen). Diese Dekodierer kennzeichnen den Adresseingang.

– Wortgrößen von Speichern, Befehlen und Adressen sind zu berücksichtigen.
– Sprungbefehle zu beliebigen Zieladressen müssen unterstützt werden.
– Eine schnelle Kodierung ist nur erforderlich, wenn veränderliche Informationen kodiert werden. Ansonsten reicht eine schnelle Dekodierung aus.

Es gibt dabei verschiedene Variationen dieses Schemas:

– Einige Prozessoren verfügen über einen **zweiten Befehlssatz** mit einem „schmaleren" Befehlsformat. Beispielsweise verwendet der normale ARM-Befehlssatz 32 Bit breite Befehle. Die meisten ARM-Prozessoren verfügen daneben noch über einen zweiten Befehlssatz, die sogenannten THUMB-Befehle. THUMB-Befehle sind kürzer, da sie bedingte Ausführung[11] nicht unterstützen, weniger und kürzere Registerfelder besitzen und auch nur kürzere Direktoperanden (engl. *immediate values*) aufnehmen können (siehe Abb. 3.19).

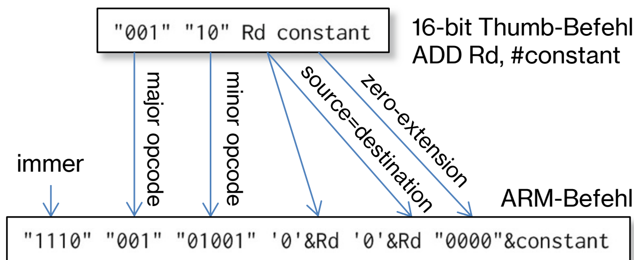


Abb. 3.19 Umkodierung von THUMB- in ARM-Befehle

THUMB-Befehle werden dynamisch in ARM-Befehle umgewandelt, während ein Programm dekodiert wird. Dabei können sie nur die Hälfte der verfügbaren Register nutzen. Den Registerfeldern von THUMB-Befehlen wird bei der Dekodierung ein `'0'`-Bit vorangestellt[12]. Im THUMB-Befehlssatz sind Quell- und Zielregister identisch und die Länge von Konstanten, die direkt im Befehl kodiert werden können, ist um 4 Bit reduziert. Während der Dekodierung wird Fließbandverarbeitung (engl. *Pipelining*) verwendet, um den Mehraufwand zur Laufzeit gering zu halten.

---

[11] Bei der bedingten Ausführung (engl. *predicated execution*) haben Befehle nur dann eine Wirkung, wenn eine im Befehl kodierte Bedingung bezüglich der Inhalte der *Condition-Code*-Register erfüllt ist. Beim ARM-Befehlssatz wird dabei die Bedingung in den höchstwertigen vier Bits des Befehlsformats gespeichert und so wird beispielsweise ein Befehl nur wirksam, wenn eine vorher geprüfte <=-Bedingung erfüllt war. Mit bedingter Ausführung können `if`-Anweisungen ohne bedingte Sprünge realisiert werden. Die bedingte Ausführung besitzt im neueren 64-Bit-Befehlssatz eine geringere Bedeutung.

[12] Unter Verwendung der VHDL-Syntax (siehe Seite 109) wird die Konkatenation durch ein &-Zeichen gekennzeichnet und Konstanten werden in Hochkommata eingeschlossen.

Ähnliche Techniken existieren auch für andere Prozessoren. Der Nachteil dieses Ansatzes ist es, dass die Entwicklungswerkzeuge (Übersetzer, Assembler, *Debugger* usw.) erweitert werden müssen, um einen zweiten Befehlssatz zu unterstützen. Daher kann dieser Ansatz zu hohen Kosten für Entwicklungswerkzeuge führen.

– Man kann in **Wörterbüchern** (engl. *dictionaries*) jedes Befehlsmuster nur ein einziges Mal speichern. Für jeden Wert des Programmzählers stellt dann eine Index-Tabelle (engl. *look-up-table*) einen Zeiger auf den entsprechenden Befehl in der Befehlstabelle, dem Wörterbuch, bereit (siehe Abb. 3.20).

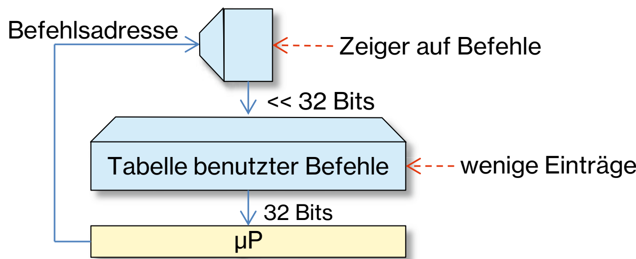


Abb. 3.20 Wörterbuch-Ansatz zur Befehlskomprimierung

Wenn nur sehr wenige unterschiedliche Befehlsmuster verwendet werden, benötigt die Befehlstabelle nur sehr wenige Einträge. Entsprechend kann die Bitbreite der Zeiger sehr schmal ausfallen. Abwandlungen dieses Schemas sind u.a. als Steuerspeicher mit zwei Ebenen (engl. *two-level control store*) [119], Nanoprogrammierung [514] oder *Procedure-Exlining* [551] bekannt.

Beszedes [52] und Latendresse [325] geben einen Überblick über bekannte Komprimierungstechniken. Außerdem publizierten Bonny et al. [58] eine Technik auf der Basis von Huffman-Kodierungen.

Laufzeiteffizienz am Beispiel Digitale Signalverarbeitung (DSP)

Um Zeitbedingungen einhalten zu können, ohne hohe Taktfrequenzen zu verwenden, können Architekturen für bestimmte Anwendungsbereiche, wie beispielsweise die digitale Signalverarbeitung (engl. *Digital Signal Processing* (DSP)), angepasst werden. Eine sehr häufige Operation der digitalen Signalverarbeitung ist das digitale Filtern. Wir erweitern nun das Verarbeitungsfließband aus Abb. 3.8 auf Seite 149 um einen solchen Filter, wie in Abb. 3.21 gezeigt.

Die Gleichung (3.21) beschreibt einen digitalen Filter, der aus einem Eingangssignal $w(t)$ das Ausgangssignal $x(t)$ erzeugt. Beide Signale sind über den (normalerweise unbegrenzten) Zeitbereich $\{t_s\}$ der Abtastwerte definiert. Wir schreiben kurz

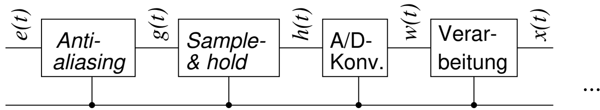


Abb. 3.21 Namenskonvention für Signale

x_s anstelle von $x(t_s)$ und $w_{s-n+k+1}$ anstelle von $w(t_{s-n+k+1})$[13]:

$$x_s = \sum_{k=0}^{n-1} w_{s-n+k+1} * a_k \tag{3.21}$$

Ein bestimmtes Ausgabeelement x_s entsteht aus dem gewichteten Mittelwert der letzten n Signalelemente von w und kann iterativ berechnet werden, wobei in jeder Iteration ein Produkt addiert wird. Prozessoren für DSP sind so ausgelegt, dass jede Iteration als ein einziger Befehl dargestellt werden kann.

Beispiel 3.5: Dies ist möglich mit den DSP-Prozessoren der Familie ADSP 2100, deren interne Architektur in Abb. 3.22 gezeigt ist.

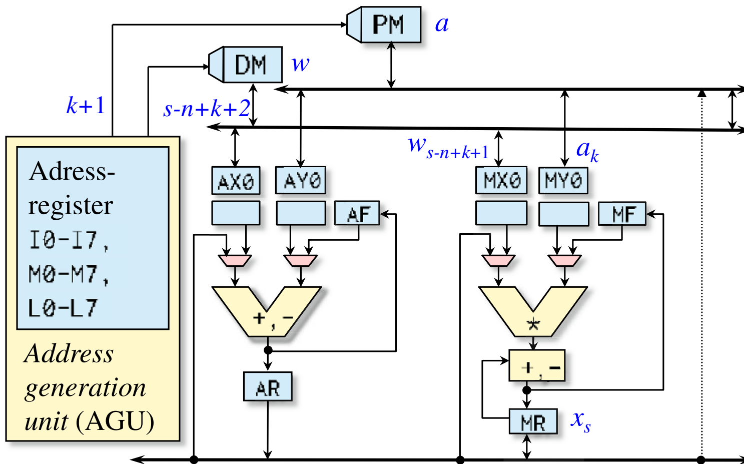


Abb. 3.22 Interne Architektur der Prozessorfamilie ADSP 2100 (vereinfacht)

Die Architektur verfügt über zwei Speicher `DM` und `PM`. Eine Adresserzeugungs-Einheit (engl. *Address Generating Unit* (AGU)) kann Zeiger für den Zugriff auf

[13] In unserer Notation ist a_0 das Gewicht des ältesten Wertes. Definiert man a_0 als Gewicht des jüngsten Wertes von w, so nimmt der erste Term die gebräuchlichere Form w_{s-k} an. Unsere Notation dient der Vorbereitung des nachfolgenden Programmcodes.

diese Speicher in Indexregistern `I0-I7` bereitstellen. Es gibt separate Einheiten für Additionen und Multiplikationen mit eigenen Parameterregistern `AX0`, `AY0`, `AF`, `MX0`, `MY0` und `MF`. Der Multiplizierer ist mit einem zweiten Addierer verbunden, um die Folge von Multiplikationen und Additionen (sog. **MAC-Operationen**) schnell berechnen zu können. Dieser Prozessor führt eine Iteration in einem Zyklus aus. Dafür wird für die Arrays w und a Speicherplatz in den beiden Speichern reserviert.

Zeiger auf die Arrayelemente werden in Indexregistern gehalten, auf die in der AGU bei jeder Iteration der Inhalt eines der ***Modify*-Register** `M0-M7` addiert wird. Dies wird meist als Nebeneffekt der Zugriffe auf die Arrays kodiert. – Teilsummen werden in `MR` gespeichert.

Würden wir jeweils das nächste eintreffende Folgenelement in einem freien Speicherelement ablegen, so würde der Speicherbedarf im Laufe der Zeit wachsen, und zwar unbegrenzt. Die Größe des Arrays w kann aber beschränkt werden, da wir nur Zugriff auf die n aktuellsten Werte benötigen. Die Wiederverwendung von Speicher ist durch Verwendung eines **Ringpuffers** und den Einsatz von Modulo-Adressierung realisierbar. Zu diesem Zweck werden die in Abb. 3.22 zu sehenden Längenregister `L0-L7` benutzt. Wenn in einem geeigneten Register der Grad n des Filters abgelegt wird, erfolgt die Adressierung des Speichers modulo n.

Die erwähnten Register haben offenbar verschiedene Aufgaben, sie sind **heterogen**. Heterogene Registersätze sind für DSP-Prozessoren charakteristisch.

Um Zyklen für das Testen auf das Schleifenende einzusparen, stellen DSP-Prozessoren häufig ***zero-overhead loop instructions*** bereit. Diese erlauben es, einen einzelnen Befehl oder eine kleine Zahl von Befehlen mehrmals zu wiederholen.

Damit kommen wir zur Vorstellung der Realisierung des Filters aus Gleichung (3.21) mit Prozessoren der ADSP 2100-Familie (adaptiert aus [14]):

```
/* äußere Schleife über Abtastzeitpunkte ts */ {
  L0 = n; L4 = n;                                    /* Ringpufferlänge */
  M1 = 1; M5 = 1;                             /* Inkrement für Indexregister */
  I0 = Adresse ältester Wert in w; I4 = Beginn der Filtertabelle a;
  MX0 = DM[I0]; MY0 = PM[I4];                          /* Lade ältestes w[] & a0 */
  MR = 0; I0 = I0 + M1; I4 = I4 + M5;        /* Ringpuffer-gewahre Addition */
  for (k=0;  k < (n − 1);  k++) {                          /* n-1 Iterationen */
     MR = MR + MX0 * MY0; MX0 = DM[I0]; MY0 = PM[I4];   /* MAC Operation */
     I0 = I0 + M1; I4 = I4 + M5;             /* Ringpuffer-gewahre Addition */
  }
  MR = MR + MX0 * MY0; x[s] = MR;                  /* MAC für jüngstes Elem. */
}
```

Die äußere Schleife entspricht den fortschreitenden Abtastzeitpunkten. Zu Beginn jeder Filterberechnung erfolgt die Initialisierung der Register. Ein einzelner Befehl realisiert den inneren Schleifenkörper, bestehend aus vier Operationen:

- Lesen von zwei Argumenten aus den Registern `MX0` bzw. `MY0`, Multiplizieren der Argumente und Addieren des Ergebnisses zur Teilsumme in Register `MR`,
- Laden der nächsten Elemente der Arrays a und w aus den Speichern `PM` und `DM` und Speichern der Werte in den Argumentregistern `MX0` und `MY0`,

- Addition von Indexregister `I0` bzw. `I4` und Modify-Register `M1` bzw. `M5` bei Berücksichtigung der Längenregister `L0` und `L4`,
- Testen auf das Schleifenende.

Für bestimmte Berechnungen kann diese (begrenzte) Form der Parallelität in vergleichsweise niedrigen Taktfrequenzen resultieren. Prozessoren, die nicht für DSP optimiert sind, würden vielleicht mehrere Befehle pro Iteration benötigen und damit ggf. eine höhere Taktfrequenz erfordern. ∇

Zusätzlich zur Realisierung von Filtern mit Hilfe eines einzigen Befehls im Schleifenkörper bieten DSP-Prozessoren noch weitere Merkmale, die spezifisch für den DSP-Anwendungsbereich sind:

- **Sättigungsarithmetik:** Sättigungsarithmetik verändert die Behandlung von Über- und Unterläufen. Die Standard-Binärarithmetik verwendet ein sogenanntes *wrap-around* bei Über- oder Unterläufen. Dabei werden die einzelnen Bits innerhalb der darstellbaren Stellen des Ergebnisses so berechnet, als gäbe es keinen Über- bzw. Unterlauf. Auftretende Überträge werden einfach ignoriert. Abbildung 3.1 zeigt ein Beispiel, bei dem zwei vorzeichenlose Vier-Bit-Zahlen addiert werden. Dabei wird ein Übertrag erzeugt, der nicht in einem der Standardregister zurückgegeben werden kann. Das Ergebnisregister enthält bei Verwendung von *wrap-around* ein Muster, das nur aus Nullen besteht. Kein Ergebnis könnte weiter vom erwarteten Ergebnis entfernt sein als dieses.

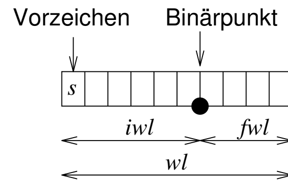
Tabelle 3.1 Vergleich von *Wrap-around* und Sättigungsarithmetik für vorzeichenlose Zahlen

	0 1 1 1
+	1 0 0 1
Standard *wrap-around* Arithmetik	1 0 0 0 0
Sättigungsarithmetik	1 1 1 1

  Bei der Sättigungsarithmetik wird ein Ergebnis erzeugt, das so nahe wie möglich am wahren Ergebnis liegt. Die Sättigungsarithmetik gibt den größten möglichen Wert zurück, wenn ein Überlauf stattgefunden hat, und den kleinsten möglichen Wert im Falle eines Unterlaufs. Dieser Ansatz ist insbesondere für Video- und Audioanwendungen sinnvoll. Der Benutzer wird im obigen Beispiel wohl kaum den Unterschied zwischen dem wahren Ergebniswert und dem größten darstellbaren Wert erkennen können. Es wäre auch sinnlos, bei einem Überlauf eine Ausnahme zu signalisieren, da es schwierig wäre, eine solche Ausnahme in Echtzeit zu behandeln. Man muss allerdings wissen, ob vorzeichenlose oder vorzeichenbehaftete Zahlen verwendet werden, um den richtigen Ergebniswert zu bestimmen.
- **Festkommaarithmetik:** Gleitkommarechnungen besitzen Eigenheiten (siehe [186]), die teilweise nicht erwünscht sind. Außerdem erhöht Gleitkommahardware die Kosten und die Leistungsaufnahme von Prozessoren. Geschätzte 80% der DSP-Prozessoren haben daher keine Gleitkommahardware [1]. Zusätzlich zur Ganzzahlverarbeitung bieten viele solcher Prozessoren aber die Verarbeitung von Festkommazahlen an. Festkommadatentypen können mit Hilfe eines

3-Tupels $(wl, iwl, sign)$ spezifiziert werden, wobei wl die Gesamtwortlänge ist, iwl die Ganzzahlwortlänge (also die Anzahl der Stellen links des Binärpunkts) und $sign \in \{s, u\}$ zwischen vorzeichenbehafteten und vorzeichenlosen Zahlen unterscheidet (siehe Abb. 3.23). Auch gibt es verschiedene Rundungsmodi (z.B. Abschneiden) und Überlaufmodi (z.B. Sättigungs- und *Wrap-around*-Arithmetik).

Abb. 3.23 Parameter eines Festkommazahlensystems



Bei Festkommazahlen bleibt die Position des Binärpunkts nach der Multiplikation erhalten (einige der niederwertigen Bits werden abgeschnitten oder gerundet). Bei Festkommaprozessoren wird diese Operation direkt von der Hardware unterstützt.

- **Echtzeitfähigkeit:** Einige der Eigenschaften moderner Prozessoren dienen dazu, die durchschnittliche Ausführungszeit von Programmen zu verbessern. In vielen Fällen ist es schwierig bis unmöglich, formal nachzuweisen, dass diese Vorkehrungen auch die längste mögliche Ausführungszeit (engl. *Worst Case Execution Time* (WCET)) verbessern. In solchen Fällen kann es besser sein, auf den Einsatz dieser Techniken zu verzichten. Beispielsweise ist es schwierig (aber nicht unmöglich, siehe [4]), eine bestimmte Leistungssteigerung durch den Einsatz eines Caches zu garantieren. Aus diesem Grunde verzichten viele eingebettete Prozessoren auf den Einsatz von Caches. Ebenso wird man virtuelle Adressierung und *demand paging* [213] (siehe auch Anhang C) üblicherweise nicht in eingebetteten Systemen finden. Techniken zur Vorhersage von *worst case*-Ausführungszeiten werden in Unterabschnitt 5.2.2 vorgestellt.

DSP-Befehle wurden aufgrund der Bedeutung der Signalverarbeitung vielen existierenden Befehlssätzen hinzugefügt.

Multimedia- und *short vector*-Befehlssätze

Die Register und Rechenwerke moderner Prozessorarchitekturen haben häufig eine Breite von mindestens 64 Bit. Man kann zwei 32-Bit-Datentypen, vier 16-Bit-Datentypen oder acht-8 Bit-Datentypen (Bytes) in ein einziges 64-Bit-Register laden (siehe Abb. 3.24).

Arithmetische Einheiten können so entworfen werden, dass die Überlaufbits an den 32-Bit-, 16-Bit- oder 8-Bit-Grenzen unterdrückt, also nicht weitergeleitet, werden. Multimediabefehlssätze nutzen diese Eigenschaft aus, um Operationen auf sogenannten gepackten Datentypen durchzuführen. Solche Befehle werden manchmal

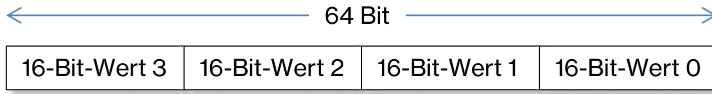


Abb. 3.24 Verwenden eines 64-Bit-Registers für gepackte Datentypen

Single Instruction, Multiple Data- (SIMD-) Befehle genannt, da ein einziger Befehl die auszuführende Operation auf mehreren Datenelementen vorgibt [212]. Bei der Verwendung von 8 Bit langen Datentypen in einem 64-Bit-Register sind Geschwindigkeitssteigerungen von etwa einem Faktor von acht gegenüber nicht-gepackten Datentypen möglich. Die Daten werden meist im Speicher gepackt abgelegt. So kann man das Entpacken und Packen vermeiden, indem man arithmetische Operationen direkt auf den gepackten Daten durchführt. Multimediabefehle können i.d.R. Sättigungsarithmetik verwenden, um eine effiziente Überlaufbehandlung zu realisieren. Der Geschwindigkeitsvorteil kann also durchaus größer sein als der Faktor, der sich nur aus der Anzahl gleichzeitig bearbeiteter Datenelemente in gepackten Datentypen ergibt. Die Vorteile, auf gepackten Daten arbeiten zu können, führten dazu, dass bei einer Reihe von Prozessoren entsprechende Befehle ergänzt wurden. So wurden z.B. für Intels Pentium®-kompatible Prozessoren [248] *Streaming SIMD Extensions* (SSE) als Befehlssatzerweiterung eingeführt. Solche neuen Befehle werden auch *short vector instructions* genannt. Sie wurden von Intel® als die sogenannten *Advanced Vector Extensions* (AVX) [249] realisiert.

Very Long Instruction Word-Prozessoren

Die Anforderungen an die Rechenleistung eingebetteter Systeme steigen, insbesondere im Multimediabereich, in dem moderne Kodierungstechniken und Kryptographie eingesetzt werden, stark an. Die in üblichen Hochleistungsrechnern verwendeten leistungssteigernden Techniken sind für eingebettete Systeme ungeeignet: da beispielsweise bei PCs die Befehlssatzkompatibilität ein wichtiger Faktor ist, verwenden die in diesen Systemen eingebauten Prozessoren viele ihrer Ressourcen und einen großen Teil der Energie dazu, parallel ausführbare Befehle in einer Anwendung zu finden. Trotzdem ist ihre Leistungsfähigkeit oft nicht ausreichend. In eingebetteten Systemen kann man ausnutzen, dass eine Befehlssatzkompatibilität mit PCs nicht erforderlich ist. Daher können Befehle verwendet werden, welche die parallel ausführbaren Operationen explizit angeben. Diese Prozessoren heißen *Explicit Parallelism Instruction Set Computers* (EPICs). Bei EPICs wird die Identifizierung von parallel ausführbaren Befehle von der Hardware zum Compiler verschoben.

Man kann sowohl Silizium als auch Energie einsparen, wenn mögliche Parallelität nicht mehr zur Laufzeit analysiert werden muss. Als Möglichkeit dazu betrachten wir sehr lange Befehlsworte (engl. *Very Long Instruction Words* (VLIW)). Bei VLIW-Prozessoren werden mehrere Operationen oder Befehle in einem sehr langen Befehlswort (das manchmal **Befehlspaket** genannt wird) kodiert und dann parallel

ausgeführt. Jede Operation wird in ein bestimmtes Feld des Befehlspakets geschrieben. Jedes dieser Felder steuert eine bestimmte Hardwareeinheit an. In Abb. 3.25 werden vier Felder verwendet, von denen jedes eine Hardwareeinheit steuert.

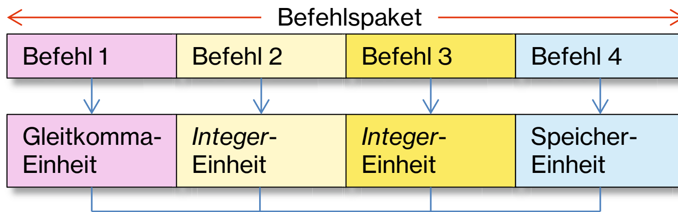


Abb. 3.25 VLIW-Architektur (Beispiel)

Der Compiler muss für VLIW-Architekturen Befehlspakete erzeugen. Dazu muss er Informationen über verfügbare Hardwareeinheiten haben, um diese auszulasten.

Die Felder in den Befehlen sind fest vorgegeben und müssen vorhanden sein, unabhängig davon, ob in einem bestimmten Zyklus ein Befehl auf der entsprechenden Einheit ausgeführt werden soll. Wenn nicht genügend Parallelität vorhanden ist, um alle Einheiten auszulasten, kann die Codedichte von VLIW-Prozessoren daher relativ niedrig sein. Dieses Problem kann durch mehr Flexibilität vermieden werden. Beispielsweise verwendet die Familie der Texas Instruments TMS 320C6xx-Prozessoren eine variable Befehlspaketgröße von bis zu 256 Bit. In jedem Befehlsfeld ist ein Bit reserviert, das angibt, ob die im nächsten Feld angegebene Operation parallel ausgeführt werden soll. Wegen der variablen Länge ihrer Befehlspakete sind die TMS 320C6xx-Prozessoren keine klassischen VLIW-Prozessoren. Aufgrund der explizit beschriebenen Parallelität sind sie aber auf jeden Fall EPICs.

Die Implementierung der Registersätze für VLIW- und EPIC-Prozessoren ist nicht einfach. Wegen der großen Anzahl parallel auszuführender Operationen wird eine große Anzahl paralleler Registerzugriffe benötigt. Man benötigt also viele Schreib- und Lese*ports* für die Register. Allerdings werden Registersätze mit vielen *Ports* zunehmend langsamer, größer und verbrauchen mehr Energie, sind also ineffizient. Daher verwenden viele VLIW/EPIC-Architekturen partitionierte Registersätze. Funktionseinheiten werden dann nur an eine Teilmenge der Register angeschlossen.

VLIW-Fließbänder

Ein potentielles Problem von VLIW- und EPIC-Architekturen ist ihre möglicherweise große *delay penalty*. Darunter versteht man Prozessorzyklen, die nicht für nützliche Operationen genutzt werden können, weil benötigte Speicherworte nicht schnell genug bereitgestellt werden können (z.B. bei Sprungbefehlen). Üblicherweise werden alle Befehlspakete in einem Fließband (engl. *Pipeline*) [212] verarbeitet. Jede der Fließbandstufen führt nur einen bestimmten Teil der Operationen des jeweiligen Befehls aus. Die Tatsache, dass ein Sprungbefehl in einem Befehlspaket

enthalten ist, kann nicht direkt in der ersten Fließbandstufe entdeckt werden. Wenn die Ausführung des Sprungbefehl endgültig abgeschlossen ist, wurden bereits weitere Befehle im Fließband bearbeitet (siehe Abb. 3.26).

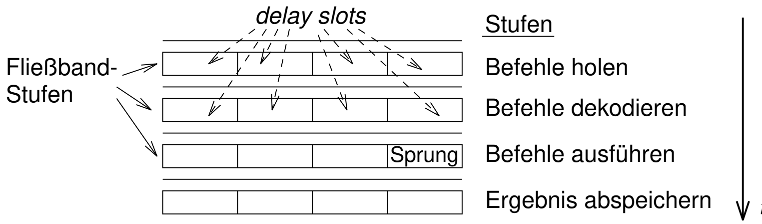


Abb. 3.26 Sprungbefehl und *delay slots*

Es gibt im Wesentlichen zwei Methoden, mit diesem potentiellen Leistungsverlust umzugehen:

1. Die Befehle werden so ausgeführt, als wären gar keine Sprungbefehle vorhanden. Dieser Fall heißt **verzögerter Sprung** (engl. *delayed branch*). Die Befehle des Befehlspaketes, die nach dem Sprung noch ausgeführt werden, heißen Befehle in ***branch delay slots***. *Delay slots* können mit Befehlen gefüllt werden, die ohne *delay slots* vor dem Sprung ausgeführt werden würden. Im Allgemeinen schwierig, alle *delay slots* mit sinnvollen Befehlen zu füllen, und so müssen einige mit *no-operation* (NOP)-Befehlen aufgefüllt werden. Der Ausdruck ***branch delay penalty*** gibt den Geschwindigkeitsverlust aufgrund dieser NOPs an.
2. Das Fließband wird angehalten, bis die ersten Befehle vom Sprungziel des Sprunges geladen worden sind. So gibt es keine *branch delay slots* und die *branch delay penalty*-Zyklen werden durch das Anhalten des Fließbandes verursacht.

Die *branch delay penalties* können sich sehr deutlich auswirken. Man kann die **bedingte Ausführung** (engl. *predicated execution*, siehe Seite 164) von Befehlen ausnutzen, um Sprünge aufgrund von **if**-Anweisungen zu vermeiden.

Der Crusoe-Prozessor™ ist ein (kommerziell nicht erfolgreiches) Beispiel eines EPIC-Prozessors für PCs [296]. Der IA-64 Befehlssatz [250] und seine Implementierung in den Itanium®-Prozessoren waren der Versuch von Intel, EPIC-Befehlssätze im PC-Markt einzuführen. Durch Probleme mit vorhandener Software hatte sich aber der Servermarkt als Haupteinsatzgebiet herausgestellt. Viele MPSoC-Systeme (siehe Seite 178) verwenden VLIW- und EPIC-Prozessoren.

Mehrkern-Prozessoren

Die bislang beschriebenen Merkmale von Einzel-Prozessoren haben geholfen, hoch performante Prozessoren auf ressourcengewahre Weise zu entwerfen. Allerdings hat

sich gezeigt, dass man beim Versuch einer weiteren Performanzsteigerung gegen die sogenannte *power wall* läuft: über Jahre wurden Taktfrequenzen zwecks Performanzsteigerung erhöht, aber eine weitere Steigerung würde in einer zu hohen Stromaufnahme und in zu heißen Schaltkreisen resultieren. Eine weitere Steigerung der Parallelität in VLIW-Prozessoren war ebenfalls nicht möglich. Aufgrund der Fortschritte in der Halbleiterfertigungstechnik ist es mittlerweile machbar, mehrere Prozessoren auf demselben Halbleiterchip zu fertigen. Wenn man mehrere Prozessoren auf demselben *Chip* fertigt, so nennt man die Prozessoren **Mehrkern-Prozessoren**. Von diesen unterscheiden muss man Multiprozessor-Systeme, wie sie in Rechenzentren seit Jahrzehnten im Einsatz sind. Verglichen mit diesen erlaubt die Integration von mehreren Kernen auf demselben *Chip* eine viel schnellere Kommunikation und eine gemeinsame Nutzung von Ressourcen (wie z.B. Caches).

Beispiel 3.6: Abb. 3.27 zeigt die Architektur des Intel® *Core*™ *Duo* [540].

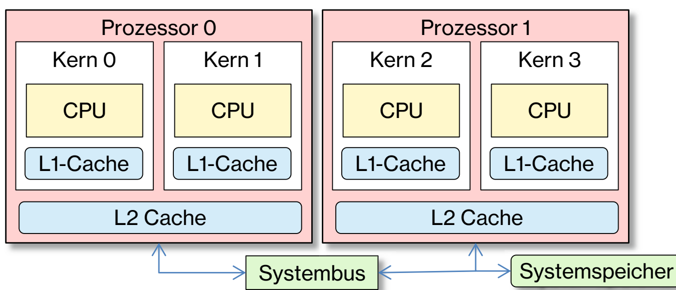


Abb. 3.27 Intel® *Core*™ *Duo*-Prozessor

In diesem Fall sind die L1-Caches den Prozessoren dediziert zugeordnet, während der L2-Cache gemeinsam genutzt wird. Selbstverständlich gibt es neben der hier aufgeführten Architektur weitere Mehrkern-Architekturen. ∇

Bei Mehrprozessor-Systemen muss die Cache-Kohärenz betrachtet werden. Das bedeutet, wir müssen wissen, ob Änderungen an Daten und möglicherweise auch an Befehlen im Cache durch einen Prozessor auch von den anderen Prozessoren gesehen werden. Protokolle für eine automatische Cache-Kohärenz sind seit vielen Jahren ein klassisches Thema in der Rechnerarchitektur [212]. Mit dem Aufkommen von Mehrkern-Prozessoren sind sie nunmehr auch ein Thema innerhalb eines *Chips*. Dabei können wir von einer wachsenden Zahl von Kernen ausgehen, sodass die Skalierbarkeit eine Rolle spielt: für wie viele Kerne können wir eine ausreichende Kommunikationsbandbreite bereitstellen, sodass Caches kohärent bleiben? Auch taucht die Frage auf, ob die Bandbreite des Speichersystems ausreichend ist, um eine wachsende Anzahl von Kernen zu versorgen.

Im Beispiel der Abbildung 3.27 sind alle Prozessoren von demselben Typ. Wir sprechen in diesem Fall von einer **homogenen** Mehrkern-Architektur. Homogene

Mehrkern-Architekturen haben den Vorteil eines begrenzten Entwurfsaufwandes, weil die Prozessoren einfach repliziert werden können. Auch kann die Software einfach von einem Prozessor zu einem anderen migrieren. Dies ist sehr hilfreich, wenn einer der Prozessoren ausfällt.

Im Unterschied zu homogenen gibt es auch heterogene Mehrkern-Architekturen mit unterschiedlichen Prozessoren. Damit können die für bestimmte Anwendungen oder bestimmte Szenarien am besten geeigneten Prozessoren gewählt werden. Meist benötigt man heterogene Mehrkern-Prozessoren, um die bestmögliche Energieeffizienz zu erreichen, da nur so eine Anpassung an Anwendungen möglich ist.

Als Kompromiss zwischen homogenen und (vollständig) heterogenen Architekturen gibt es Architekturen mit einem einheitlichen Befehlssatz (engl. *Instruction Set Architecture* (ISA)) aber unterschiedlichen internen Architekturen, sogenannte *single-ISA heterogeneous multi-cores* [317].

Beispiel 3.7: Die ARM® big.LITTLE-Architektur ist ein prominentes Beispiel dafür. Abb. 3.28 zeigt die Fließbandarchitektur des Cortex® A-15 Prozessors [165].

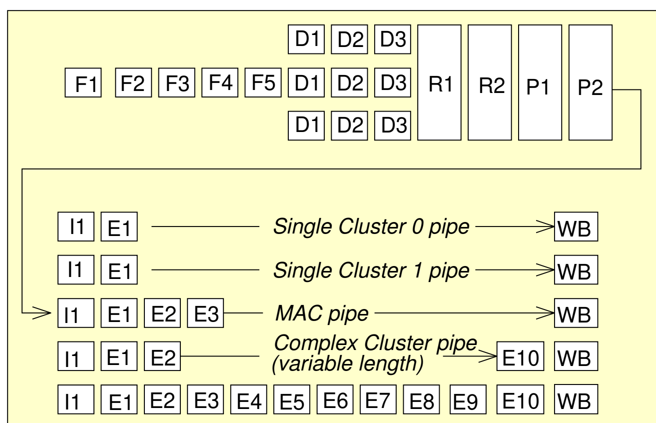


Abb. 3.28 ARM® Cortex® A-15 Fließband

Es ist ein relativ komplexes Fließband mit mehreren Fließbandstufen für das Holen von Befehlen, die Befehlsdekodierung, die Befehlsausgabe, die Befehlsausführung und das Abspeichern der Ergebnisse. Bei diesem Fließband müssen Befehle mindestens 15 Stufen durchlaufen, bevor Ergebnisse abgespeichert werden können. Eine dynamische Ablaufplanung erlaubt es, Befehle in einer Reihenfolge auszuführen, die sich von derjenigen unterscheidet, in der sie aus dem Speicher geholt werden. Dies nennt man *out-of-order execution*. Mehrere Befehle können innerhalb desselben Taktzyklus an die Ausführungseinheiten ausgegeben werden. Dies nennt man *multi instruction issue*. Insgesamt ist diese Architektur sehr performant, aber sie hat eine hohe elektrische Leistungsaufnahme.

Abb. 3.29 zeigt die Fließbandarchitektur des Cortex® A-7 Prozessors [165]. Das

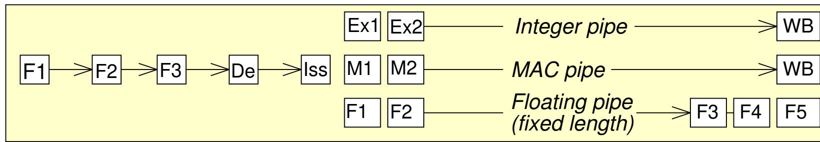


Abb. 3.29 ARM® Cortex® A-7 Fließband

Fließband ist relativ einfach. Die Befehle durchlaufen nach Flautner acht bis elf Stufen [165]. Sie werden immer in der Reihenfolge ausgeführt, in der sie aus dem Speicher geholt werden. Dies nennt man *in-order execution*. Es gibt nur eine begrenzte Anzahl von Situationen, in denen zwei Befehle in demselben Taktzyklus an die Ausführungseinheiten herausgegeben werden. Dadurch hat die Architektur eine geringe Leistungsaufnahme, aber auch nur eine begrenzte Performanz.

Abb. 3.30 [165] macht es deutlich, wie man zwischen den Optimierungskriterien Leistungsaufnahme und Performanz durch Wahl einer der Architekturen abwägen kann. Für jede der Architekturen ist dabei durch die Wahl der Betriebsspannung und der Taktfrequenz eine Flexibilität hinsichtlich dieser Kriterien gegeben.

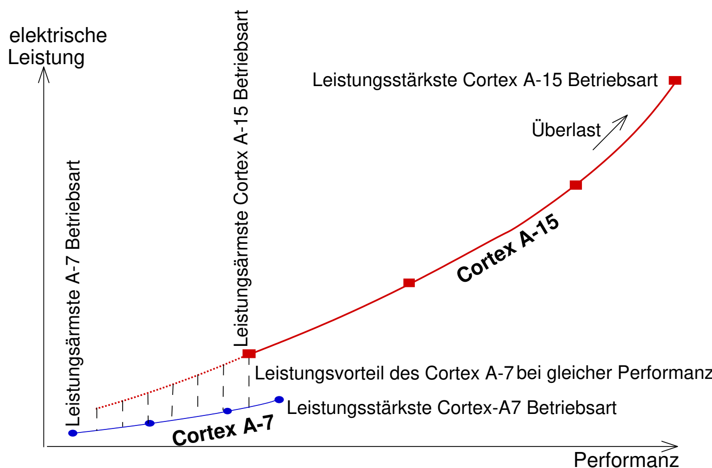


Abb. 3.30 *Tradeoff* zwischen Leistungsaufnahme und Performanz auf einem A7- oder A15-Kern

Für herausfordernde Anwendungen (wie z.B. die Videoverarbeitung) ist offensichtlich ist der Cortex® A-15-Kern besser geeignet. Für permanent aktive Anwendungen mit geringen Performanzanforderungen ist der Cortex® A-7 dagegen besser geeignet. Hätte ein Mobiltelefon mit teilweise permanent aktiven Anwendungen nur Cortex® A-15-Kerne, so wäre das eine Energieverschwendung.

Daher sind heutige Mehrkern-*Chips* typischerweise heterogen in dem Sinne, dass sie eine Mischung von hoch performanten und energieeffizienten Prozessoren enthalten, wie in Abb. 3.31 gezeigt. ∇

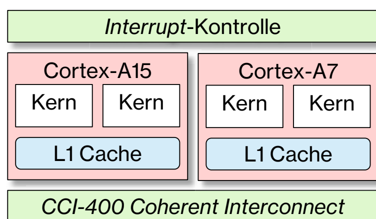


Abb. 3.31 ARM® big.LITTLE-Architektur mit Cortex® A-7- und Cortex® A-15-Kernen

Graphikprozessoren

Früher benutzten viele Rechensysteme spezielle Graphikprozessoren (engl. *Graphics Processing Units* (GPUs)), um eine ansprechende Ausgabe beispielsweise auf Bildschirmen zu erzeugen. Diese fest verdrahteten Prozessoren konnten nur Standard-Graphikalgorithmen ausführen und waren damit zu unflexibel. Daher wurden sie durch programmierbare GPUs ersetzt und führten so zu allgemeinen Berechnungen auf Graphikeinheiten (engl. *General Purpose Computation on Graphics Processing Units* (GPGPU)). Die benötigte Performanz erzielen aktuelle GPUs durch die Möglichkeit, sehr viele Berechnungen in Form von feingranularen Fäden (engl. *Threads*) gleichzeitig auszuführen. Ziel ist dabei, möglichst viele der zahlreichen gleichartigen Recheneinheiten gut auszulasten.

Beispiel 3.8: Wir betrachten die Multiplikation von zwei großen Matrizen auf einer GPU. Abb. 3.32 [212] zeigt eine Abbildung von Berechnungen auf eine GPU.

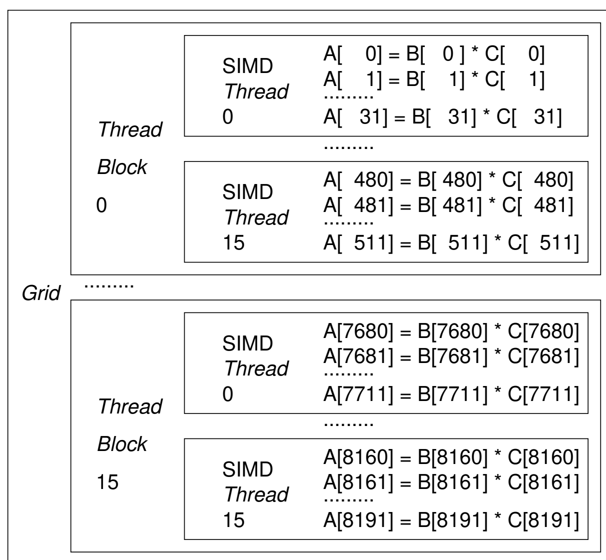


Abb. 3.32 Partitionierung einer Matrixmultiplikation zur Ausführung auf einer GPU

Die Matrix wird aufgeteilt in sogenannte *Thread*-Blöcke. Jeder *Thread*-Block kann auf einen der Kerne in einer GPU abgebildet werden. Jeder *Thread*-Block enthält eine Anzahl von *Threads* und jeder *Thread* enthält eine Anzahl von Befehlen. Die Gesamtheit der Berechnungen in Abb. 3.32 bezeichnet man als *grid*. ∇

Jeder Kern wird versuchen, durch die Ausführung von *Threads* einen Rechenfortschritt zu erzeugen. Wenn einige *Threads* blockieren, weil sie z.B. auf den Speicher warten, wird der Kern einen anderen *Thread* ausführen. Die Befehle innerhalb eines *Threads* können ebenfalls gleichzeitig ausgeführt werden, indem beispielsweise mehrere Fließbänder genutzt werden. *Thread*-Blöcke können auf aktuellen Prozessoren gleichzeitig ausgeführt werden. Ein schnelles Umschalten zwischen den *Threads* und die sich dadurch ergebende Möglichkeit, Speicherlatenz zu verstecken, sind ein wesentliches Merkmal von GPUs.

Beispiel 3.9: Abb. 3.33 zeigt die Architektur der ARM® Mali™ T880 GPU [22]. Die Architektur ist als geistiges Eigentum (engl. *Intellectual Property* (IP)) de-

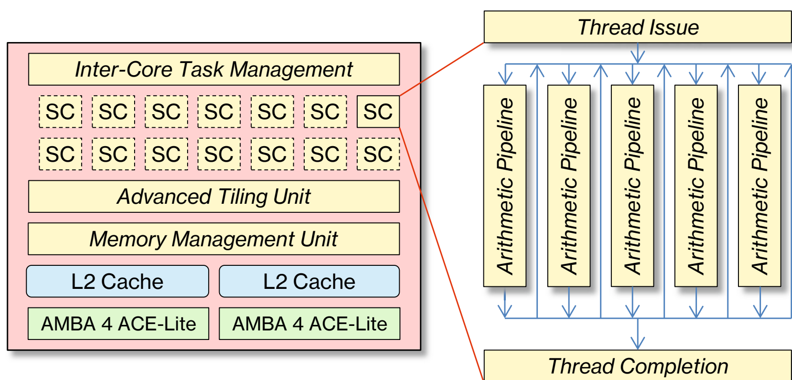


Abb. 3.33 ARM® Mali™ T880 GPU

finiert. Es gibt ein sogenanntes **synthetisierbares Modell**, aus dem heraus eine Realisierung erzeugt werden kann. In diesem Modell ist die Anzahl von SC-Kernen konfigurierbar. Jeder Kern enthält mehrere Fließbänder zur Ausführung von arithmetischen und anderen Befehlen. In jedem Taktzyklus werden so viele *Threads* wie möglich an die Ausführungseinheiten herausgegeben. Die GPU enthält auch weitere Komponenten wie z.B. eine Speicherverwaltungseinheit (siehe Anhang C), bis zu zwei Caches und eine AMBA® Bus-Schnittstelle. Die Softwareentwicklung wird durch eine Schnittstelle zur OpenGL-Bibliothek [484] und zu OpenCL (siehe https://www.khronos.org/opencl/) unterstützt. ∇

Im Allgemeinen kann auf GPUs energieeffizient gerechnet werden (siehe Unterabschnitt 3.7.3 auf Seite 211).

Multiprozessor-Systeme auf einem *Chip*

Man kann noch einen Schritt weitergehen und heterogene Mehrkern-Prozessoren zusammen mit GPUs und evtl. weiteren Komponenten auf einem *Chip* integrieren.

Beispiel 3.10: Abb. 3.34 zeigt ein heterogenes Mehrkern-System, welches eine Mali GPU [21] enthält.

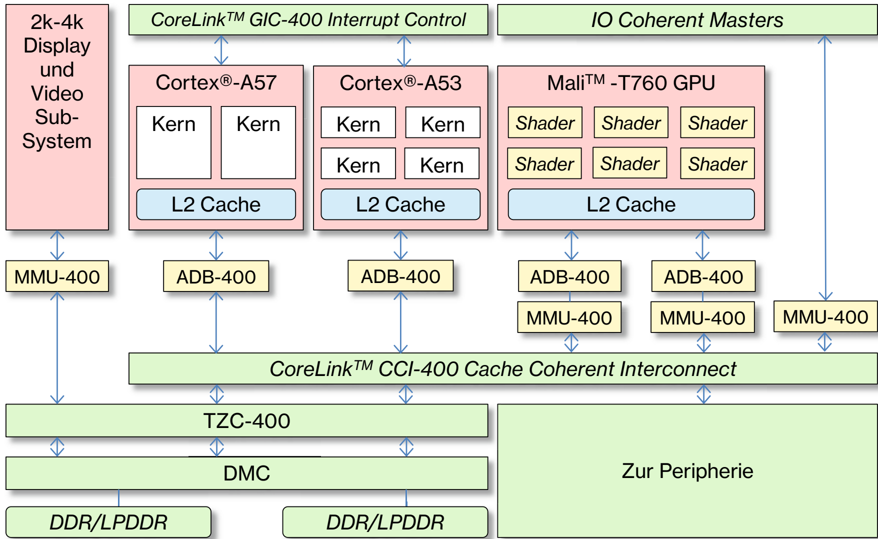


Abb. 3.34 ARM® big.LITTLE System on Chip (SoC)

Abb. 3.34 enthält nicht nur Prozessorkerne, sondern auch zusätzliche Elemente wie Speicherverwaltungseinheiten (engl. *Memory Management Units* (MMUs), siehe Anhang C) und Schnittstellen für externe Geräte. Cache-Kohärenz wird unterstützt, d.h. angeschlossene Caches werden aufgrund von Schreibvorgängen in anderen Teilen des Systems ggf. aktualisiert.

Damit soll vermieden werden, dass für diese Einheiten zusätzliche *Chips* benötigt werden. Im Endergebnis wird ein vollständiges System auf einem *Chip* integriert. Wir sprechen daher von einem System auf einem *Chip* (engl. *System-on-a-Chip* (SoC)) oder sogar von einem Mehrprozessor-System auf einem *Chip* (engl. *Multi-Processor System on a Chip* (MPSoC)). ∇

Es ist wichtig, für Anwendungen eine geschickte Zuordnung zu MPSoC-Prozessoren vorzunehmen, da gezeigt werden kann, dass so eine Energieeffizienz nahe der von ASICs erreicht werden kann. Beispielsweise wurde für IMEC's ADRES Prozessor eine Effizienz von $55 * 10^9$ Operationen pro Watt und damit 50% der Effizienz von ASICs vorhergesagt [364, 481]. Allerdings ist der Entwurfsaufwand für solche Architekturen hoch.

Beispiel 3.11: Es gibt MPSoCs, die eine Kombination von bereits vorgestellten Prozessoren enthalten: 66AK2x MPSoCs von Texas Instruments enthalten ARM®- und C66xxx-Prozessoren [530] (siehe Abb. 3.35). Damit wird die Bedeutung der vorgestellten Prozessoren unterstrichen.

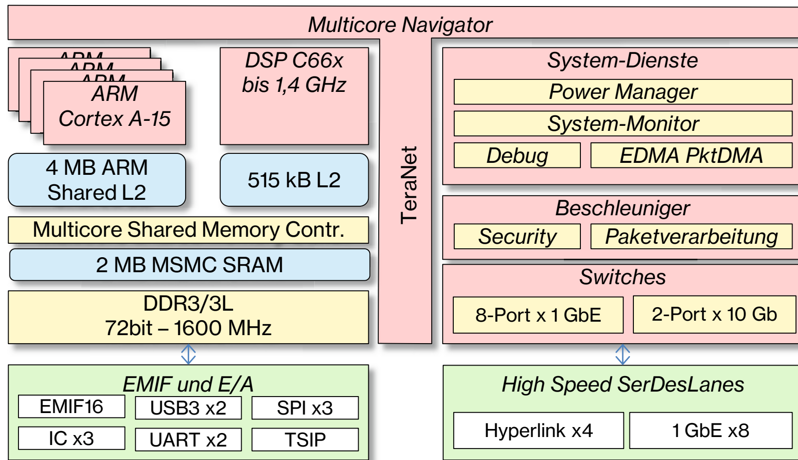


Abb. 3.35 MPSoC 66AK von Texas Instruments® mit ARM®- und C6xxx-Prozessoren ∇

Anzahl und Diversität der Komponenten kann noch größer sein, wie bei spezialisierten Prozessoren für die mobile Kommunikation oder die Bildverarbeitung.

Beispiel 3.12: Abb. 3.36 zeigt ein vereinfachtes *Layout* des *SH-MobileG1-Chips* [206]. Es werden hochspezialisierte Prozessoren benutzt, so z.B. für Bildverarbeitung

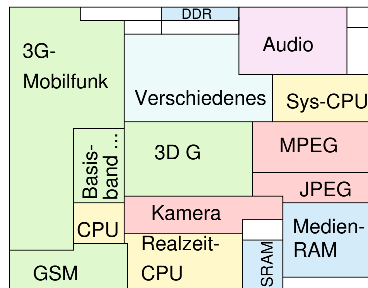


Abb. 3.36 *Layout* des *SH-MobileG1*-Chips

(rot), für GSM- und 3G Mobil-Kommunikation (grün) usw. Zur Einsparung von Energie werden unbenutzte Teile des *Chips* nicht mit Strom versorgt, wodurch diese Bereiche zu *dark silicon* werden (siehe Seite 15). ∇

Der Übergang auf spezialisierte Prozessoren erfolgt aufgrund von nachlassenden Fortschritten in der Halbleiterfertigung und beim Entwurf von parallelen Prozessoren. Deshalb soll die Performanz nunmehr mit Spezialprozessoren erhöht werden. Dies wurde mit Spezialprozessoren für das Maschinelle Lernen auch schon erreicht.

Im Jahr 2013 sagte Google vorher, dass es bald zu teuer werden würde, die benötigte Rechenleistung für Mustererkennung (z.B. für Spracherkennung) in den Rechenzentren bereitzustellen. Daher wurde die Entwicklung von Prozessoren gestartet, die für die schnelle Klassifikation von Mustern mit tiefen neuralen Netzwerken (engl. *Deep Neural Networks* (DNNs)) optimiert sein sollten. Die entsprechend entwickelte *Tensor Processing Unit* (TPU) zeigt die Abbildung 3.37.

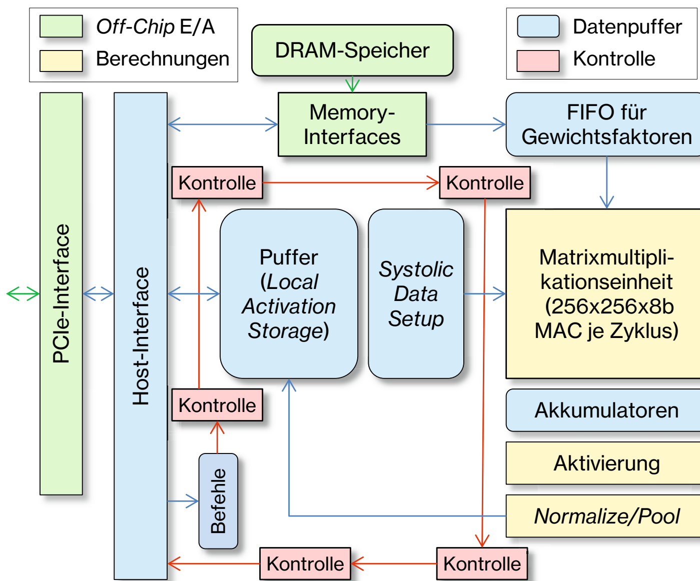


Abb. 3.37 *Tensor Processing Unit* (TPU), v1, zur schnellen Klassifikation [278, 448]

Den Kern der TPU bildet ein Feld von 256 x 256 MAC-Einheiten. Damit können 64k 8-Bit-MAC-Operationen je Taktzyklus ausgeführt werden. 16-Bit-Operationen benötigen zwei bis drei Zyklen. DNNs bestehen aus mehreren Ebenen von Berechnungen, wobei auf jeder Ebene MAC-Operationen mit Gewichtsfaktoren erforderlich sind. Diese werden realisiert, indem Eingabedaten oder Daten von Zwischenebenen durch die MAC-Matrix hindurch „gepumpt" werden. In jedem Zyklus werden 256 Ergebniswerte verfügbar. Die Version 1 der TPU ist um Faktoren von 29,2 bzw. 13,3 schneller als übliche CPUs bzw. GPUs. Der Quotient aus Performanz und Leistungsaufnahme hat sich dabei um Faktoren von 34 bzw. 16 verbessert. Aufgrund des Erfolgs hat Google eine zweite und eine dritte Generation von TPUs angekündigt [93]. Diese unterstützen auch das **Trainieren** von DNNs.

3.3.3 Rekonfigurierbare Logik

Für viele Anwendungsfälle ist ein vollständiger Hardwareentwurf und der Einsatz von anwendungsspezifischen integrierten Schaltkreisen (ASICs) zu teuer. Andererseits sind Lösungen, die auf Software basieren, häufig zu langsam oder verbrauchen zu viel Energie. Rekonfigurierbare Logik kann dann eine Lösung darstellen, wenn die verwendeten Algorithmen effizient in speziell angepasster Hardware implementiert werden können. Diese Lösung kann beinahe so schnell sein wie der Einsatz von ASICs. Im Gegensatz zu diesen kann die ausgeführte Funktion allerdings durch das Ändern von Konfigurationsdaten verändert werden. Wegen dieser Eigenschaften findet die rekonfigurierbare Logik in den folgenden Gebieten Anwendung:

- **Prototypen:** Moderne ASICs können sehr komplex sein und der Entwurfsprozess ist langwierig und teuer. Daher möchte man häufig einen Prototypen zur Verfügung haben, der für Versuche mit dem System verwendet werden kann und der sich „fast“ wie das endgültige System verhält. Der Prototyp darf teurer und größer als das endgültige System sein. Auch darf sein Stromverbrauch durchaus noch höher sein und Zeitbedingungen können entschärft werden. Es ist auch möglich, dass der Prototyp nur die wichtigsten Funktionen implementiert. Solch ein Prototyp kann dazu verwendet werden, das grundlegende Verhalten des zu entwickelnden Systems zu überprüfen.
- **Kleine Stückzahlen:** Wenn das zu erwartende Marktvolumen zu klein ist, um die hohen Entwicklungskosten für ASICs zu amortisieren, kann rekonfigurierbare Logik der richtige Ansatz für Anwendungen sein, die sich nicht allein in Software implementieren lassen.
- **Echtzeitsysteme:** Das Zeitverhalten FPGA-basierter Systeme lässt sich meist sehr genau bestimmen. Daher können FPGAs verwendet werden, um Systeme mit vorhersagbarem Zeitverhalten zu implementieren.
- **Anwendungen mit möglicher stark paralleler Verarbeitung:** Beispielsweise kann die parallele Suche nach bestimmten Mustern sehr effizient in paralleler Hardware realisiert werden. Dementsprechend wird rekonfigurierbare Logik bei der Suche nach genetischer Information, bei der Suche nach Mustern der Internet-Kommunikation, bei Börsendaten, bei der seismischen Analyse usw. eingesetzt.

Rekonfigurierbare Hardware enthält typischerweise einen Speicher, um die Konfigurationsinformation während des normalen Betriebs der Hardware abzuspeichern. Wir unterscheiden zwischen **flüchtigem** und **persistentem** Konfigurationsspeicher (engl. *volatile and persistent configuration memory*). Bei persistentem Speicher wird die Information nach einem Ausschalten der Betriebsspannung beibehalten. Bei flüchtigem Speicher geht sie dagegen verloren und muss beim Start aus einem persistenten Speicher (wie einem *Flash*-Speicher) geladen werden.

Field programmable gate arrays (FPGAs) sind die häufigste Form rekonfigurierbarer Hardware. Diese Schaltungen können „im Feld” (d.h. nach der Herstellung) mit Konfigurationsdaten programmiert werden. Sie bestehen aus einer matrixförmigen Anordnung von Verarbeitungselementen.

Beispiel 3.13: Abb. 3.38 zeigt die spaltenbasierte Struktur der Xilinx® UltraScale Architektur [602][14]. Einige Spalten enthalten Ein/Ausgabe-Schnittstellen (als *Transceiver* bezeichnet), Taktgeber und/oder Speicherschnittstellen. Andere Spalten enthalten **konfigurierbare Logikblöcke** (engl. *Configurable Logic Blocks* (CLBs)), spezielle Hardware für die digitale Signalverarbeitung und ebenfalls Speicher. CLBs sind die Schlüsselkomponenten. Sie stellen auf konfigurierbare Weise logische Funktionen bereit.

Abb. 3.38 *Layout* der spaltenbasierten Xilinx® UltraScale FPGAs

Die Architektur der Xilinx® UltraScale CLBs wird in Abb. 3.39 [601] gezeigt.

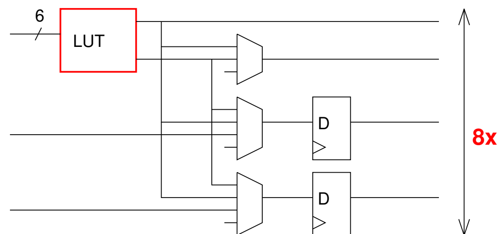


Abb. 3.39 Xilinx® UltraScale CLB (1/8 gezeigt)

In dieser Architektur enthält jeder CLB acht Blöcke. Jeder Block enthält zwei Register, Multiplexer, einzelne Logikelemente sowie RAM-Speicher, mit dem Logikfunktionen über eine Tabelle (engl. *Look-Up Table* (LUT), in der Abbildung in rot gezeigt) realisiert werden können. Jeder RAM-Speicher hat sechs Adresseingänge und zwei Ausgänge. Er kann jede Boolesche Funktion von sechs Variablen oder zwei Funktionen von fünf Variablen (wobei beide Funktionen gemeinsame Variablen haben müssen) realisieren. Mithin können alle 2^{64} bzw. 2^{32} Booleschen Funktionen von 6 bzw. 5 Variablen implementiert werden! Dies ist der Schlüssel für die **Konfigurierbarkeit**. Manche RAM-Speicher können als gewöhnliche Speicher verwendet

[14] Eine Drehung dieser Abbildung um 90 Grad würde ihre Lesbarkeit verbessern, aber sie würde der offiziellen Bezeichnung des *Layout*-Stils widersprechen.

werden. Zusätzlich können die einzelnen Logikelemente konfiguriert werden. Die Ansteuerung der Register kann ebenfalls gewählt werden, sodass diese entweder die Werte aus den RAM-Speichern oder direkte Eingaben zwischenspeichern. Die Blöcke in einem CLB können so kombiniert werden, dass sich Addierer, Multiplexer, Schieberegister oder größere Speicher ergeben. Die Konfiguration bestimmt weiterhin die Auswahl der Eingänge der Multiplexer, die Taktung von Registern und Speichern und die Verbindungen zwischen den CLBs. ∇

Gegenwärtig verfügbare FPGAs enthalten eine große Anzahl von spezialisierten Blöcken, wie z.B. DSP-Hardware, Speicher, schnelle Ein/Ausgabeschnittstellen für verschiedene Standards, die Entschlüsselung von Konfigurationsdaten, *Debugging*, Analog-Digital-Wandler, schnelle Taktgeneratoren usw.

Beispiel 3.14: Die Virtex® *UltraScale*™ VU13P Schaltkreise enthalten 1728 k LUTs, 48 Mbit verteilten RAM-Speicher, 94,5 Mbit „Block RAM", 360 Mbit „UltraRAM", ungefähr 12 k spezialisierte DSP Blöcke, 4 PCIe® Blöcke, Ethernet-Schnittstellen und bis zu 832 Ein/Ausgabe-Pins [600]. ∇

Die Integration von rekonfigurierbarer Logik mit anderen Prozessoren und Software wird bei FPGAs vereinfacht, wenn Prozessoren verfügbar sind, die direkt auf dem FPGA implementiert werden können. Hierbei gibt es sowohl *hard cores* wie auch *soft cores*. Bei *hard cores* enthält die Schaltung des FPGA bereits einen Bereich, der einen Prozessorkern mit hoher Packungsdichte beinhaltet. Dieser Bereich kann ausschließlich für diesen Prozessor verwendet werden. *Soft cores* dagegen sind als synthetisierbare Modelle verfügbar, die dann auf gewöhnliche CLBs abgebildet werden. Damit sind sie flexibler, aber auch weniger effizient als *hard cores*.

Beispiel 3.15: Der MicroBlaze-Prozessor [598] ist ein Beispiel eines *soft cores*. ∇

Beispiel 3.16: Hard cores sind auf Zynq UltraScale+ MPSoCs verfügbar. Sie enthalten bis zu vier ARM® Cortex-A53 Kerne, zwei ARM Cortex-R5 Kerne und einen Mali-400MP2 GPU-Prozessor [602]. ∇

Konfigurationen werden üblicherweise aus einer Beschreibung der Funktionalität der Hardware auf hoher Ebene, z.B. in VHDL, erzeugt. FPGA-Hersteller stellen dazu Entwicklungsumgebungen bereit. Idealerweise sollte man aus derselben Beschreibung automatisch ASICs erzeugen können. In der Praxis gelingt dies nur auf interaktive Weise. Da eine automatische Parallelisierung von Anwendungen beim gegenwärtigen Stand der Technik selten mit ausreichender Qualität möglich ist, setzt eine Ausnutzung der verfügbaren Hardwareparallelität meist voraus, dass Anwendungen manuell parallelisiert werden. Die vorhandene Hardwareparallelität wird üblicherweise nicht voll ausgenutzt, wenn Operationen lediglich den Prozessoren zugeordnet werden. FPGAs erlauben es, eine große Vielfalt an Hardwareschaltungen zu implementieren, ohne dass man andere Hardware als FPGA-Platinen benötigt.

Beispiel 3.17: Neben Xilinx® bieten auch weitere Hersteller FPGAs an. Darunter befinden sich derzeit (2020) Altera® (siehe http://www.altera.com, durch Intel® übernommen), Lattice Semiconductor (siehe http://www.latticesemi.com), Quicklogic (siehe http://www.quicklogic.com), Microsemi (siehe http://www.microsemi.com, früher Actel) sowie chinesische Hersteller.

3.4 Speicher

3.4.1 Zielkonflikte

Daten, Programme und FPGA-Konfigurationen müssen in Speichern abgespeichert werden. Speicher müssen so groß sein, wie die Anwendungen es verlangen, sie müssen die benötigte Performanz bieten, hinreichend zuverlässig sein, Zugriffe der gewünschten Granularität (Bytes, Worte, Seiten) ermöglichen und ggf. persistent speichern. Sie müssen dennoch in Bezug auf die Kosten, physikalische Größe und den Energieverbrauch effizient sein. Diese Anforderungen führen zu Zielkonflikten, was schon 1946 von Burks, Goldstine und von Neumann bemerkt wurde [78]:

„Ideally one would desire an indefinitely large memory capacity such that any particular ... word ... would be immediately available — i.e. in a time which is ... shorter than the operation time of a fast electronic multiplier. ... It does not seem possible physically to achieve such a capacity."

Zugriffszeiten für aktuelle Speicher können mit CACTI abgeschätzt werden. Diese Abschätzungen basieren auf der internen Erzeugung eines *Layouts* des Speichers und der Extraktion der Kapazitäten daraus [589].

Beispiel 3.18: Abb. 3.40 zeigt Ergebnisse von CACTI für exponentiell wachsende Größen [35]. Die Zugriffszeit wächst mit der Größe der Speicher. Abb. 3.40 enthält

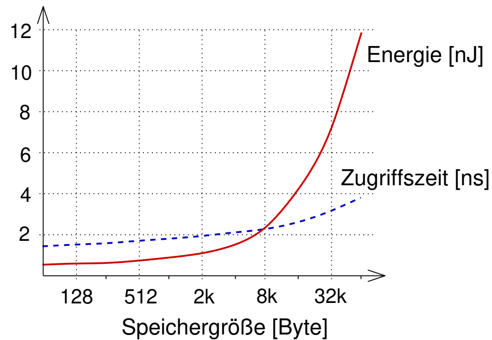


Abb. 3.40 Energieverbrauch und Zugriffszeit für Speicher gemäß CACTI

auch den jeweiligen Energieverbrauch. Danach sind große Speicher nicht energieeffizient. Die Speichergröße hat auf den Energieverbrauch tatsächlich einen größeren Einfluss als auf die Zugriffszeit. ∇

Über viele Jahre hinweg ist der Unterschied zwischen den Prozessor- und den Speichergeschwindigkeiten immer größer geworden, bis etwa ab dem Jahr 2003 die Taktraten von Prozessoren aufgrund der *power wall* weitgehend stagnierten (siehe Abb. 3.41).

Während die Geschwindigkeit von Speichern sich um einen Faktor von ca. 1,07 pro Jahr verbessert hat, wurden Prozessoren mit einem Faktor zwischen 1,5 und

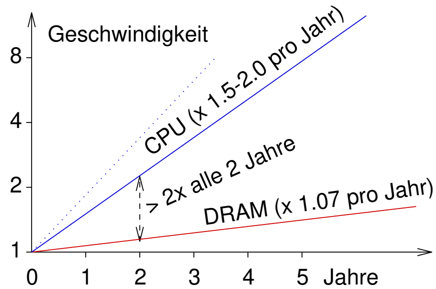


Abb. 3.41 Unterschied zwischen Prozessor- und Speichergeschwindigkeiten (bis 2003)

2 pro Jahr schneller [359]. Insgesamt ist die Lücke zwischen der Prozessor- und der Speichergeschwindigkeit sehr groß geworden und hat eine weitere Erhöhung der Performanz erschwert. Die Begrenzung der Performanz durch Speicher ist unter dem Begriff *memory wall* bekannt. Die weitere Erhöhung der Taktraten ist seit dem Jahr 2003 weitgehend zum Erliegen gekommen, aber die im Jahr 2003 vorhandene Lücke ist damit natürlich noch nicht geschlossen. Mehrkern-Systeme stellen darüber hinaus noch zusätzliche Anforderungen an das Speichersystem. Insgesamt müssen wir aufgrund der angesprochenen Zielkonflikte günstige Kompromisse zwischen den verschiedenen Anforderungen und Randbedingungen finden.

3.4.2 Speicherhierarchien

Wegen der beschriebenen Zielkonflikte schrieben Burks, Goldstine und von Neumann schon 1946 [78]: *„We are therefore forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible"*. Die genaue Struktur der Hierarchie hängt von technologischen Parametern und dem Anwendungsbereich ab. Üblicherweise können wir mindestens die folgenden Hierarchiestufen identifizieren:

- **Prozessor-Register** können als die schnellste Ebene der Speicherhierarchie betrachtet werden, wobei die Kapazität höchstens einige Hundert Worte beträgt.
- Der **Arbeitsspeicher** (oder **Hauptspeicher**) realisiert die Speichermöglichkeiten, die durch die Adressen üblicher Befehlssätze impliziert werden. Üblicherweise hat er eine Kapazität zwischen einigen Megabyte und einigen Gigabyte und er ist flüchtig.
- Typischerweise gibt es eine große Differenz in den Zugriffsgeschwindigkeiten zwischen dem Hauptspeicher und den Registern. Aus diesem Grund enthalten viele Systeme Pufferspeicher, wie z.B. Caches, *Translation Look-aside Buffers* (TLBs, siehe Anhang C) und/oder *Scratchpad Memory* (SPM). Im Unterschied zu PC-ähnlichen Systemen sollten diese Pufferspeicher möglichst ein vorhersagbares Echtzeitverhalten aufweisen. Eine Kombination von kleinen Speichern, die häufig

benutzte Informationen enthalten, mit einem großen Speicher für die übrigen Informationen ist in der Regel auch energieeffizienter als ein einzelner großer Speicher.
- Die bislang angesprochenen Speicher sind bislang in der Regel flüchtige Speicher. Zur dauerhaften Speicherung muss eine andere Technologie benutzt werden. Für eingebettete Systeme ist *Flash*-Speicher häufig die beste Lösung. In anderen Fällen können Hintergrundspeicher wie Festplatten oder Internet-basierte Lösungen (die „*Cloud*") eingesetzt werden.

Speicherhierarchien können genutzt werden, um einen Kompromiss zwischen den verschiedenen Entwurfszielen zu erreichen. Hierzu hat A. Macii beispielsweise die Speicherpartitionierung genutzt [360]. Neue Speichertechnologien, die teilweise ein permanentes Speichern erlauben, haben das Potential, die gegenwärtige Aufteilung der Hierarchien zu verändern [389].

3.4.3 Registersätze

Der vorgestellte Einfluss der Speichergröße auf die Zugriffszeiten und die Leistungsaufnahme gilt auch für sehr kleine Speicher wie Registersätze. Abb. 3.42 zeigt die Zykluszeit und die elektrische Leistung als Funktion der Größe der Speicher für eine Halbleiterfertigungstechnologie mit Strukturbreiten von $\lambda = 0,18\ \mu m$ [471]. Die elektrische Leistung ist besonders wichtig, weil Registerbänke aufgrund der häufigen Zugriffe sehr heiß werden können.

Abb. 3.42 Zykluszeit und elektrische Leistung als Funktion der Größe eines Registersatzes

3.4.4 Caches

Im Fall von Caches überprüft die Hardware, ob der Cache eine gültige Kopie der Informationen zu einer bestimmten Hauptspeicheradresse enthält. Diese Überprüfung beinhaltet einen Vergleich der *Tag*-Felder, die aus einem Teilbereich der Bits der

Hauptspeicheradressen bestehen [212]. Wenn der Cache keine gültige Kopie enthält, muss die Information aus dem Hauptspeicher automatisch nachgeladen werden.

Caches wurden ursprünglich eingeführt, um eine gute Laufzeiteffizienz zu ermöglichen. Der Name ist von dem französischen Wort *cacher* (verstecken) abgeleitet, was darauf deuten soll, dass Programmierer diesen Speicher nicht sehen oder sich um ihn kümmern müssen, weil die Aktualisierung des Speichers automatisch durch die Hardware erfolgt. Caches sind allerdings nicht mehr so versteckt, wenn große Datenmengen adressiert werden. Dies wurde sehr schön von Drepper gezeigt [139]. Drepper analysierte die Ausführungszeiten eines Programms, welches eine lineare Liste von Einträgen durchläuft. Jeder Eintrag enthielt NPAD 64-Bit-Worte sowie einen 64-Bit-Zeiger auf das Folgeelement. Für einen Pentium P4-Prozessor mit 16 kB *Level*-1 Cache und 1 MB *Level*-2 Cache wurden die Ausführungszeiten bestimmt. Dabei wurden für den *Level*-1 Cache, den *Level*-2 Cache und den Hauptspeicher Zugriffszeiten von vier, vierzehn und 200 Taktzyklen angenommen. Abb. 3.43 zeigt die durchschnittliche Anzahl von Zyklen pro Zugriff auf ein Listenelement als Funktion der Gesamtgröße der Liste für den Fall NPAD=0. Für kleine Listengrößen reichen vier

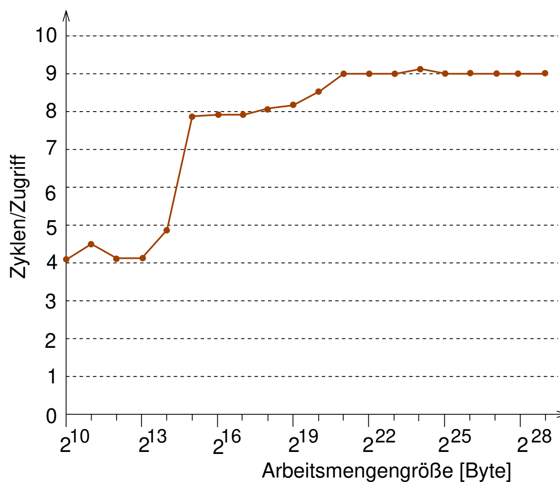


Abb. 3.43 Durchschnittliche Anzahl der Zyklen pro Zugriff für NPAD=0

Zyklen pro Element aus. Dies bedeutet, dass wir fast immer auf den *Level*-1 Cache zugreifen, da dieser für diese Listengröße ausreicht. Für größere Listen benötigen wir acht Zugriffe pro Element. In diesem Fall greifen wir fast immer auf den *Level*-2 Cache zu. Da der Cache aus Blöcken besteht, die jeweils zwei Listenelemente speichern können, wird im Mittel nur etwa bei jedem zweiten Listenelement wirklich auf den Cache zugegriffen. Für noch größere Listen steigt die Zugriffszeit auf neun Zyklen. In diesem Fall ist die Liste größer als der *Level*-2 Cache, aber das automa-

tische Vorabladen von *Level*-2 Cacheeinträgen versteckt einen Teil der Zugriffszeit des Hauptspeichers.

Abb. 3.44 zeigt die durchschnittliche Anzahl von Zyklen pro Zugriff auf ein Element als eine Funktion der Gesamtgröße der Liste für die Fälle NPAD=0, 7, 15 und 31. Für NPAD=7, 15 und 31 versagt das Vorabladen aufgrund der größeren Listenelemente. Offenbar steigen die Zugriffszeiten dramatisch an und die Cachearchitektur hat einen starken Einfluss auf die Ausführungszeit von Anwendungen. Bei größeren Caches wird sich lediglich die Größe der Anwendungen verschieben, ab der ein Anwachsen der Ausführungszeiten zu beobachten ist. Eine intelligente Ausnutzung der Speicherhierarchie kann einen großen Einfluss auf die Ausführungszeiten haben.

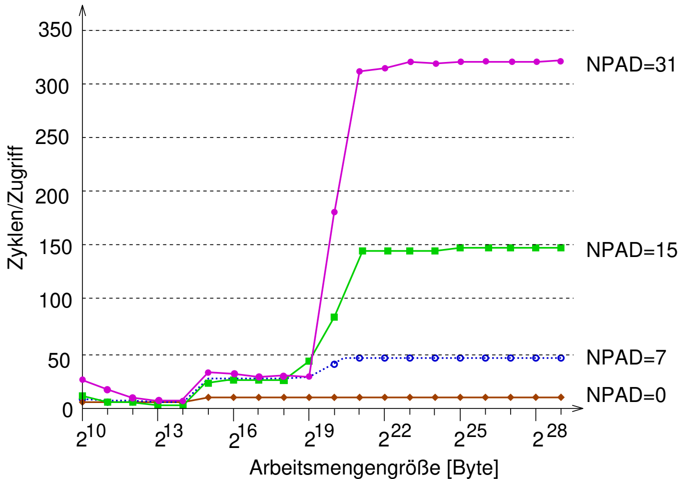


Abb. 3.44 Durchschnittliche Anzahl der Zyklen pro Zugriff für NPAD=0, 7, 15, 31

Bislang haben wir uns nur den Einfluss der Caches auf die Zugriffs- bzw. Ausführungszeiten angesehen. Aus dem Kontext der Abb. 3.40 ist aber klar, dass Caches auch die Energieeffizienz eines Speichersystems verbessern.

Es ist schwierig, bereits zur Entwurfszeit vorherzusagen, ob ein Speicherzugriff zu einem Treffer im Cache führt oder nicht, was eine präzise Vorhersage des Echtzeitverhaltens erschwert (siehe Seite 269).

3.4.5 *Scratchpad-Speicher*

Als Alternative können kleine Speicher in den Adressbereich des Systems eingeblendet werden (siehe Abb. 3.45). Solche Speicher werden ***Scratchpad*-Speicher** (SPM) genannt. Auf sie wird zugegriffen, indem eine Adresse aus dem eingeblendeten Be-

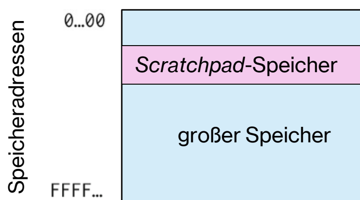


Abb. 3.45 Adressbereich mit eingeblendetem Bereich für einen *Scratchpad*-Speicher

reich benutzt wird. Im Unterschied zu Caches sind keine Gültigkeitsüberprüfungen von *Tags* zur Laufzeit notwendig. Vielmehr kann man mit einem einfachen Adressdekoder überprüfen, ob eine Adresse aus dem eingeblendetem Bereich vorliegt. SPMs werden meist zusammen mit den Prozessoren auf demselben *Chip* integriert. Sie sind daher ein Spezialfall von *On-Chip*-Speichern. Bei n-fach mengenassoziativen Caches werden bei Leseoperationen üblicherweise n Einträge parallel gelesen und der gesuchte Eintrag anschließend ausgewählt. Dies führt zu einer unnötig hohen Energieaufnahme, die für SPMs vermieden wird. Somit sind SPMs in der Regel energieeffizienter als Caches.

Abbildung 3.46 zeigt einen Vergleich zwischen der Energie pro Zugriff auf einen SPM und einen Cache. Für einen zweifach mengenassoziativen Cache unterscheiden

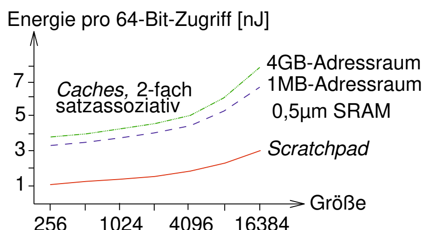


Abb. 3.46 Energie pro Zugriff für SPM und Cache

sich die Werte in etwa um einen Faktor drei. Die Energiewerte wurden mit Hilfe des CACTI-Programms berechnet [589]. Ein detaillierter Vergleich zwischen den Zielkriterien für Caches und SPMs wurde von Banakar et al. [35] publiziert.

Häufig verwendete Variablen und Befehle sollten in den Adressbereich des SPM gelegt werden. Speicherzugriffszeiten können auf vorhersehbare Weise gesenkt werden, wenn Compiler häufig benutzte Variablen oder Befehle einem SPM zuordnen. Damit können sich insbesondere bei Echtzeitsystemen Vorteile gegenüber Caches ergeben.

3.5 Kommunikation

Bevor Informationen in einem eingebetteten System verarbeitet werden können, müssen sie erst einmal verfügbar sein. Kommunikation ist insbesondere beim Internet

der Dinge sehr wichtig. Die Übertragung der Information kann über einen von vielen möglichen sogenannten **Kanälen** erfolgen. Kanäle sind abstrakte Gebilde, die durch die Eigenschaften der Kommunikation charakterisiert werden, etwa durch die maximale Übertragungsgeschwindigkeit oder die Störparameter. Die Wahrscheinlichkeit für Fehler bei der Kommunikation wird mit Hilfe von Techniken aus der Kommunikationstheorie berechnet. Die physikalische Grundlage, auf der die Kommunikation erfolgt, wird als **Medium** bezeichnet. Wichtige Medienklassen sind etwa drahtlose Medien (Radiowellenübertragung, Infrarot), optische Medien (Lichtwellenleiter) oder elektrische Leitungen.

Bei der Vielzahl von Klassen eingebetteter Systeme gibt es auch viele unterschiedliche Anforderungen an die Kommunikation. Im Allgemeinen ist das Verbinden von Hardwarekomponenten eingebetteter Systeme alles andere als einfach. Einige häufig vorkommende Anforderungen werden im nächsten Unterabschnitt aufgezählt.

3.5.1 Anforderungen

Die folgende Liste beschreibt einige Anforderungen, die bei der Kommunikation eingehalten werden müssen:

- **Echtzeitverhalten:** Diese Anforderung hat weitreichende Auswirkungen auf den Entwurf des Kommunikationssystems. Einige kostengünstige Lösungen, wie etwa Ethernet, erfüllen diese Anforderung nicht.
- **Effizienz:** Die Verbindung zweier Hardwarekomponenten kann recht teuer sein. Beispielsweise ist eine direkte Punkt-zu-Punkt-Verbindung in einem großen Gebäude nahezu unmöglich. Im Automobilbereich hat sich gezeigt, dass separate Kabel, welche die Steuereinheiten mit der externen Peripherie verbinden, sehr teuer und schwer sind. Einzelne Kabel machen es auch schwierig, neue Komponenten anzuschließen. Die Kosten haben auch einen Einfluss auf den Entwurf der Stromversorgung externer Geräte. Häufig wird eine zentrale Stromversorgung eingesetzt, um Kosten zu sparen.
- **Angemessene Bandbreite und Kommunikationslatenz:** Die geforderten Bandbreiten eingebetteter Systeme unterscheiden sich sehr stark. Die zur Verfügung stehende Bandbreite muss den Anforderungen genügen, ohne das System unnötig zu verteuern.
- **Unterstützung für ereignisgesteuerte Kommunikation:** Systeme, die auf dem regelmäßigen Abfragen von Geräten (sogenanntes *Polling*) basieren, besitzen ein sehr gut vorhersagbares Echtzeitverhalten. Die Latenzen können bei dieser Art der Kommunikation allerdings zu groß sein. Oft wird eine schnelle, ereignisgesteuerte Kommunikation benötigt. Notfallbedingungen sollten beispielsweise sofort weitergeleitet werden und nicht solange unbemerkt bleiben, bis ein zentrales System alle Geräte nach neuen Nachrichten abfragt.
- **Robustheit:** Eingebettete Systeme sollen bei extremen Temperaturen oder in der Nähe von Quellen elektromagnetischer Strahlung eingesetzt werden. Automotoren können extremen Temperaturen ausgesetzt sein, wie beispielsweise einem

Bereich von -20 bis +180 °C. Spannungspegel und Taktfrequenzen können von solch hohen Temperaturschwankungen beeinflusst werden. Trotzdem muss eine verlässliche Kommunikation gewährleistet werden.

- **Sicherheit:** Entsprechend der bisher beschriebenen Anforderungen (siehe Seite 9) ist klar, dass CPS-Systeme betriebssicher und informationssicher sein müssen. Die Informationssicherheit beinhaltet auch die Vertraulichkeit. Um sicherzustellen, dass private und vertrauliche Informationen geheim bleiben, kann es notwendig sein, bei der Kommunikation Verschlüsselungstechniken einzusetzen. Die Betriebssicherheit enthält die nachfolgend beschriebenen Aspekte der Fehlertoleranz, Diagnosefähigkeit und Wartbarkeit.
- **Fehlertoleranz:** Trotz aller Bemühungen, eine robuste Kommunikation zu erreichen, können Fehler auftreten. Cyber-physikalische Systeme sollten auch nach einem solchen Fehler funktionsfähig bleiben. Neustarts, wie sie bei PCs vorkommen, sind inakzeptabel. Wenn eine Kommunikation also fehlgeschlagen ist, sind Wiederholungsversuche notwendig. Hier entsteht ein Konflikt mit der ersten Anforderung: wenn man mehrere Kommunikationsversuche zulässt, ist es schwierig, das Echtzeitverhalten zu garantieren.
- **Diagnosefähigkeit, Wartbarkeit:** Cyber-physikalische Systeme müssen in einer annehmbar kurzen Zeit repariert werden können.

Diese Anforderungen an die Kommunikation sind eine direkte Folge der allgemeinen Eigenschaften eingebetteter und cyber-physikalischer Systeme, die in Kapitel 1 gezeigt wurden. Wegen der Konflikte zwischen einigen der Anforderungen müssen in der Praxis Kompromisse eingegangen werden. Beispielsweise kann es unterschiedliche Kommunikationsmodi geben: einen Modus mit hoher Bandbreite, der Echtzeitverhalten garantiert, aber keine Fehlertoleranz bietet (diese Übertragungsart wäre etwa geeignet für Multimediadaten), und einen fehlertoleranten Modus mit geringerer Bandbreite, der z.B. für kurze Nachrichten verwendet wird, die nicht verlorengehen dürfen.

3.5.2 *Elektrische Robustheit*

Es gibt einige grundlegende Techniken, um elektrische Robustheit zu erreichen. Digitale Kommunikation innerhalb eines Chips findet häufig mit **asymmetrischen Signalen** statt (engl. *asymmetric* oder *single-ended signaling*). Dabei werden die Signale über eine einzige Leitung geschickt (siehe Abb. 3.47).

Solche Signale werden durch eine Spannung gegenüber einer einheitlichen Masse dargestellt, seltener durch unterschiedliche Ströme. Eine einzige Masseleitung ist ausreichend für eine größere Anzahl solcher Signale. Asymmetrische Signale sind allerdings sehr anfällig für externe Störeinflüsse. Wenn solche Störungen, die z.B. durch einen anlaufenden Motor verursacht werden können, die Spannung beeinflussen, können Nachrichten verfälscht werden. Es ist auch schwierig, einen genau definierten Massepegel zwischen verschiedenen kommunizierenden Einheiten be-

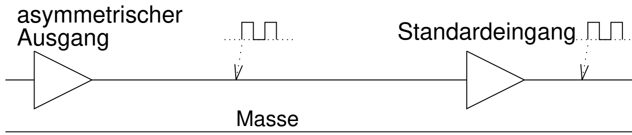


Abb. 3.47 Asymmetrische Signale

reitzustellen, da die Masseleitungen selber auch einen gewissen Widerstand und eine Induktivität haben.

Dies ist bei der **symmetrischen** oder **differentiellen Signalübertragung** anders. Bei der symmetrischen Übertragung benötigt jedes Signal zwei Leitungen (siehe Abb. 3.48). Bei der symmetrischen Signalübertragung werden binäre Werte wie folgt

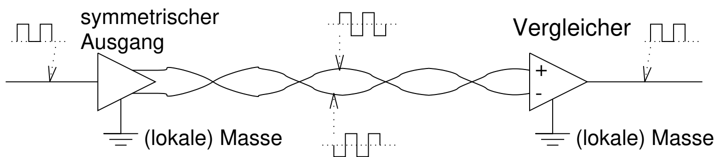


Abb. 3.48 Symmetrische Signalübertragung

kodiert: wenn die Spannung auf der ersten Leitung gegenüber der zweiten positiv ist, wird dies als `'1'` dekodiert, sonst ist der übertragene Wert eine `'0'`. Die beiden Leitungen werden üblicherweise miteinander verdrillt und bilden sogenannte *twisted pairs*. Es gibt lokale Massepegel, aber ein Spannungsunterschied zwischen diesen lokalen Massepegeln stört die Kommunikation nicht. Vorteile der symmetrischen Signalübertragung sind u.a.:

- Störungen wirken prinzipiell auf beide Leitungen in ähnlicher Weise ein. Beim Vergleich werden daher die meisten Störungen entfernt.
- Der Logikwert hängt ausschließlich von der Polarität der Spannung zwischen den beiden Leitungen ab. Die Stärke der Spannung kann durch Reflexionen oder den Leitungswiderstand verändert werden – dies hat jedoch keinen Einfluss auf den übermittelten Wert.
- Signale verursachen keine Ströme auf den Masseleitungen, wodurch die Qualität der Masseleitungen weniger kritisch ist.
- Es wird kein gemeinsames Massepotential benötigt. Daher entfällt auch die Notwendigkeit, eine große Anzahl von Kommunikationspartnern mit einem hochqualitativen gemeinsamen Masseanschluss zu versehen.
- Als Konsequenz der bisher genannten Eigenschaften erreicht die symmetrische Signalübertragung i.d.R. eine höhere Übertragungsrate als die asymmetrische Übertragung.

Allerdings müssen für differentielle Signale stets zwei Leitungen für jedes Signal zur Verfügung stehen. Außerdem benötigt man negative Spannungen, wenn man nicht nur jeweils entgegengesetzte Logikpegel der asymmetrischen Signale verwendet. Differentielle Signalübertragung wird beispielsweise in Ethernet-basierten Netzwerken sowie beim *Universal Serial Bus* (USB) verwendet.

3.5.3 Echtzeitgarantien

Computer verwenden für die interne Kommunikation dezidierte Punkt-zu-Punkt-Verbindungen oder gemeinsame Busse. Punkt-zu-Punkt-Verbindungen können ein gutes Echtzeitverhalten besitzen, erfordern aber eine große Anzahl an Verbindungen, zudem kann es zu Überlastsituationen bei den Empfängern kommen. Verbindungen mit gemeinsamen Bussen sind einfacher zu realisieren. Solche Busse verwenden meist eine prioritätenbasierte Zuteilung, wenn mehrere gleichzeitige Zugriffe auf den Bus vorliegen (siehe z.B. [212]). Das Zeitverhalten prioritätenbasierter Zuteilung ist schlecht vorhersagbar, da Konflikte zur Entwurfszeit nur schwer zu erkennen sind. Prioritätsbasierte Schemata können auch zum „Verhungern" führen (Kommunikation mit niedriger Priorität kann von solcher mit höherer Priorität vollständig blockiert werden). Um dieses Problem zu umgehen, kann *Time Division Multiple Access* (TDMA) verwendet werden. Dabei wird die Kommunikationszeit in Rahmen (engl. *Frames*) aufgeteilt. Jeder Rahmen beginnt mit einem Zeitintervall zur Synchronisation und evtl. einer Lücke, die es dem Sender erlaubt, sich abzuschalten (siehe Abb. 3.49, [303]).

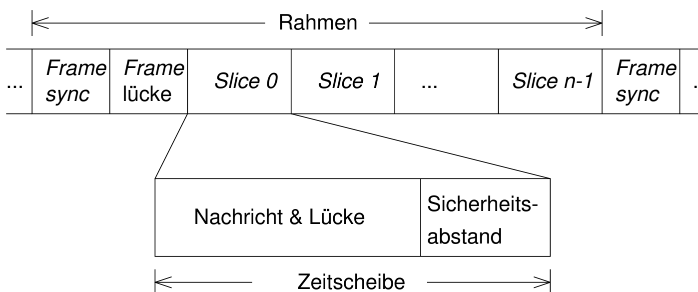


Abb. 3.49 TDMA-basierte Kommunikation

Nach dieser Lücke folgen eine Anzahl an Zeitscheiben (engl. *slices*), von denen jede eine Nachricht enthält. Auch jede Zeitscheibe enthält wieder eine Lücke und einen Sicherheitsabstand (engl. *guard time*), um die Taktratenunterschiede der Kommunikationsteilnehmer auszugleichen. Zeitscheiben sind jeweils einem Kommunikationsteilnehmer zugeordnet. Es existieren Abwandlungen dieses Schemas; so

ist es z.B. möglich, nicht verwendete Zeitscheiben zu vermeiden oder auch, eine Zeitscheibe mehreren Kommunikationsteilnehmern zuzuordnen. TDMA reduziert die maximal verfügbare Menge an Daten pro Rahmen und Kommunikationsteilnehmer, garantiert dafür aber eine bestimmte Bandbreite für jeden Teilnehmer. Das Verhungern kann dabei vermieden werden. Der AMBA-Bus von ARM®[20] beinhaltet ein Verfahren zur TDMA-basierten Buszuteilung.

Die meisten Rechnernetzwerke basieren auf den Ethernet-Standards. Für die 10 MBit/s- und 100 MBit/s-Versionen kann es zwischen den Kommunikationspartnern zu Kollisionen bei der Datenübertragung kommen. Das bedeutet, dass mehrere Partner zur gleichen Zeit versuchen, Daten zu übertragen, wodurch sich die Signale auf den Leitungen stören. Jedes Mal, wenn das passiert, müssen die Kommunikationspartner die Kommunikation einstellen, einige Zeit warten und es dann erneut versuchen. Die Wartezeit wird zufällig bestimmt, um die Wahrscheinlichkeit für einen erneuten Konflikt beim nächsten Versuch zu reduzieren. Diese Methode heißt ***Carrier-Sense Multiple Access/Collision Detect*** (CSMA/CD). Bei Verwendung von CSMA/CD können die Kommunikationszeiten sehr lang werden, da Konflikte sich häufig wiederholen können, obwohl dies nicht sehr wahrscheinlich ist. Daher kann CSMA/CD nicht eingesetzt werden, wenn Echtzeitbedingungen eingehalten werden müssen. Das Problem kann mit Hilfe von CSMA/CA (***Carrier-Sense Multiple Access/Collision Avoidance***) vermieden werden. Kollisionen werden bei diesem Verfahren komplett vermieden, anstatt sie nur zu entdecken und darauf zu reagieren. Bei CSMA/CA gibt es für jeden Kommunikationspartner eine Priorität. Das Übertragungsmedium wird den Partnern während sogenannter Arbitrierungsphasen zugeordnet, darauf folgen dann die eigentlichen Kommunikationsphasen. Während der Arbitrierungsphase geben die Partner ihren Kommunikationswunsch auf dem Übertragungsmedium bekannt. Wenn ein anderer Teilnehmer mit einer höheren Priorität ebenfalls senden möchte, müssen alle anderen ihren Kommunikationswunsch sofort zurückziehen.

Unter der Annahme, dass es eine obere Schranke für die Zeit zwischen den Arbitrierungsphasen gibt, garantiert CSMA/CA ein vorhersagbares Echtzeitverhalten für den Kommunikationspartner mit der höchsten Priorität. Für die anderen Teilnehmer kann das Echtzeitverhalten nur dann garantiert werden, wenn die höher priorisierten Partner nicht andauernd senden möchten.

Hochgeschwindigkeitsversionen von Ethernet ($\geq$ 1 GBit/s) basieren auch auf der Vermeidung von Kollisionen. TDMA-Schemata werden auch für drahtlose Kommunikation verwendet. So setzen Mobilfunk-Standards wie GSM ein TDMA-Verfahren zum Zugriff auf das Kommunikationsmedium ein.

3.5.4 Beispiele

- **Sensor-Aktuator-Busse:** Sensor-Aktuator-Busse ermöglichen die Kommunikation zwischen einfachen Geräten wie etwa Schaltern und Lampen und den datenverarbeitenden Geräten. Die Anzahl der angesteuerten Geräte kann sehr groß

werden, daher müssen die Kosten der Verdrahtung in diesem Fall besonders beachtet werden.

- **Feldbusse:** Feldbusse sind den Sensor-Aktuator-Bussen ähnlich, sie unterstützen jedoch im Allgemeinen höhere Datenraten. Beispiele für Feldbusse sind u.a. folgende:

  - Der **CAN** (***Controller Area Network***)**-Bus:** Dieser Bus wurde 1981 von Bosch und Intel entwickelt, um Steuereinheiten und Peripherie zu verbinden. Er ist insbesondere im Automobilbereich beliebt, da man mit dem CAN-Bus eine große Anzahl von Leitungen durch einen einzigen Bus ersetzen kann. Wegen der Größe des Automobilmarktes sind CAN-Komponenten verhältnismäßig preiswert und werden daher auch in anderen Bereichen, etwa zur Heimautomatisierung und in Fabriksteuerungen eingesetzt. CAN basiert auf einer differentiellen Signalübertragung und einer Arbitrierung mittels CSMA/CA. Die Kodierung der Signale erfolgt ähnlich wie bei der seriellen RS-232-Übertragung früherer PCs mit Modifikationen für differentielle Signalübertragung. Die CSMA/CA-basierte Buszuteilung verhindert nicht das Verhungern. Dies ist ein inhärentes Problem des CAN-Busses. Es gibt Erweiterungen.
  - Das ***Time-Triggered-Protocol*** **(TTP)** für fehlertolerante Sicherheitssysteme, z.B. *Airbags* in Autos [305].
  - **FlexRay**™ ist ein TDMA-Protokoll, das vom FlexRay-Konsortium (bestehend aus BMW, Daimler-Chrysler, General Motors, Ford, Bosch, Motorola und Philips Semiconductors) ausgearbeitet wurde und später zum ISO-Standard ISO 17458-1:2013 [254] wurde.
    Das FlexRay-Protokoll sieht statische und dynamische Buszuteilungsphasen vor. Die statische Phase verwendet ein TDMA-ähnliches Schema. Sie kann für echtzeitkritische Kommunikation eingesetzt werden und ist in der Lage, Verhungern zu vermeiden. Die dynamische Phase stellt dagegen eine hohe Bandbreite für nicht echtzeitkritische Kommunikation zur Verfügung.
    Die levi-Simulation ermöglicht es, das Protokoll zu simulieren [495].
  - **LIN** (*Local Interconnect Network*) ist ein preisgünstiger Kommunikationsstandard zur Verbindung von Sensoren und Aktuatoren im Automobilbereich [347].
  - **MAP** ist ein Bus, der für Automobilfabriken entworfen wurde.
  - Der ***European Installation Bus*** (EIB) wurde für die Heimautomatisierung entwickelt.

- Der ***Inter-Integrated Circuit*** **(I^2C) - *Bus*** ist ein einfacher kostengünstiger Bus, der für die Kommunikation über kurze Distanzen (Meter-Bereich) mit relativ niedrigen Datenraten entworfen wurde. Der Bus benötigt nur vier Leitungen: je eine für Masse, SCL (Takt), SDA (Daten) und die Betriebsspannung. Daten- und Taktleitungen sind *Open Collector*-Leitungen (siehe Seite 99). Folglich können verbundene Geräte diese Leitungen gegen Masse ziehen. Separate Widerstände werden benötigt, um diese Leitungen ggf. mit einer schwachen `'1'` im Sinne der CSA-Theorie (siehe Seite 98) zu verbinden. Die Standardgeschwindigkeit ist

100 kb/s, aber es existieren auch Versionen mit 10 kb/s und bis zu 3,4 Mb/s. Die Spannung auf der Betriebsspannungsleitung kann zwischen den verschiedenen Schnittstellen unterschiedlich sein. Es ist lediglich definiert, wie `'0'` und `'1'` -Pegel relativ zur Betriebsspannung erkannt werden. Der Bus wird von verschiedenen Microcontroller-Platinen unterstützt.

- **Drahtgebundene Multimediakommunikation**: Diese erfordert höhere Datenraten. Beispiel: **MOST** (*Media Oriented Systems Transport*) ist ein Kommunikationsstandard für Multimedia- und Infotainmentsysteme im Automobilbereich [403]. Standards wie z.B. IEEE 1394 (*FireWire*) können ebenfalls für diesen Zweck verwendet werden.
- Die **Drahtlose Kommunikation** wird immer beliebter.
  - **Mobile Kommunikation** bietet wachsende Datenraten an. Mit HSPA (High Speed Packet Access) werden 7 MBit/s erreicht. *Long-Term Evolution* (LTE) bietet etwa 10-fach höhere Datenraten. Für 5G-Netzwerke werden zwischen 50 MBits/s und mehr als einem Gigabit/s bei geringer Latenz erwartet.
  - **Bluetooth** ist ein Standard, um Geräte wie Mobiltelefone, Kopfhörer, drahtlose Lautsprecher und Audiosysteme in Fahrzeugen miteinander zu verbinden.
  - *Wireless Local Area Networks* (WLANs): Die drahtlose Ethernet-Version wurde von der IEEE als 802.11 standardisiert. Es gibt diverse Erweiterungen. Dieser Standard wird in lokalen Netzwerken verwendet.
  - **ZigBee** (siehe http://www.zigbee.org) ist ein Kommunikationsprotokoll zur Erzeugung persönlicher lokaler Netzwerke mit geringer Funkenergie. Es gibt Anwendungen beim Internet der Dinge und in der Heimautomatisierung.
  - *Digital European Cordless Telecommunications* (DECT) ist ein Europäischer Standard für Schnurlostelefone. Er wird in der ganzen Welt benutzt, bis auf die in Nord-Amerika unterschiedlichen Frequenzen (siehe https://en.wikipedia.org/wiki/Digital_Enhanced_Cordless_Telecommunications).

3.6 Ausgabe – Schnittstelle zwischen Cyber- und physischer Welt

Zum **CyPhy-Interface** gehören auch die Ausgabegeräte, wie beispielsweise die folgenden:

- **Anzeigen:** Die Technologie von Anzeigen (engl. *display technology*) ist sehr wichtig, deshalb gibt es darüber sehr viele Informationen [503]. Große Forschungs- und Entwicklungsanstrengungen haben zu neuen Anzeigetechnologien wie etwa organischen Anzeigen (engl. *organic displays*) [345] geführt. Organische Anzeigen können mit einer sehr hohen Dichte hergestellt werden. Im Gegensatz zu *Liquid Crystal Displays* (LCDs) benötigen sie keine Hintergrundbeleuchtung und polarisierende Filter, da sie selber Licht aussenden können. Deshalb werden grundlegende Veränderungen des Anzeigegerätemarktes erwartet.
- **Elektromechanische Geräte:** Diese Geräte beeinflussen ihre Umgebung durch Motoren und andere elektromechanische Bauteile.

Es werden sowohl analoge als auch digitale Ausgabegeräte verwendet. Im Falle analoger Ausgabegeräte muss die digitale Information zuerst mittels eines Digital-Analog-Wandlers (D/A-Wandlers) in analoge Werte konvertiert werden. Diese Konverter befinden sich entlang des Pfades von den analogen Eingängen bis hin zu ihren Ausgängen. Abb. 3.50 zeigt die Namenskonventionen für die Signale, die wir im Folgenden verwenden. Zweck und Funktion dieser Signale werden jeweils im Zusammenhang erläutert.

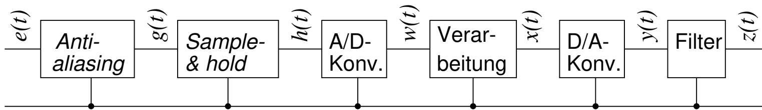


Abb. 3.50 Namenskonventionen für Signale zwischen analogen Ein- und Ausgängen

3.6.1 D/A-Wandler

D/A-Wandler sind ebenfalls Teil des **CyPhy-Interfaces** cyber-physikalischer Systeme. Abbildung 3.51 zeigt den Schaltplan eines einfachen D/A-Wandlers.

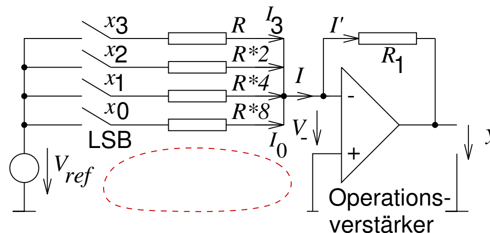


Abb. 3.51 D/A-Wandler

Die Grundidee hinter diesem Wandler ist, zunächst einen Strom zu erzeugen, der proportional zu dem Wert eines digitalen Signals x ist. Solch ein Strom kann aber nur schwer von einem nachgeschalteten System direkt verwendet werden. Daher wird dieser Strom in eine proportionale Spannung y gewandelt. Diese Wandlung übernimmt ein Operationsverstärker (in Abb. 3.51 als Dreieck dargestellt). Wesentliche Eigenschaften von Operationsverstärkern werden in Anhang B erklärt.

Wie berechnen wir nun die Ausgangsspannung y? Wir betrachten die vier Widerstände auf der linken Seite der Abb. 3.51. Der Strom durch jeden Widerstand ist Null, wenn das entsprechende Element des digitalen Signals x = `'0'` ist. Wenn das Bit den Wert `'1'` hat, entspricht der Strom durch den Widerstand dem Gewicht

dieses Bits, da die Widerstandswerte entsprechend gewählt werden. Wir betrachten nun die durch die gestrichelte rote Linie markierte Masche in Abb. 3.51. Wir können die Kirchhoffsche Maschenregel (siehe Anhang B) auf die Masche anwenden, die vom niederwertigsten Bit x_0 von x gesteuert wird. Wir beginnen den Durchlauf durch die Masche beim Widerstand und durchlaufen die Masche anschließend im Uhrzeigersinn. Der zweite Term ist die Spannung V_- zwischen den Eingängen des Operationsverstärkers. Sie wird positiv gezählt, da wir in Richtung des Pfeils vorgehen. Der dritte Term ist negativ, da wir entgegen der Pfeilrichtung laufen. Damit ist

$$x_0 * I_0 * 8 * R + V_- - V_{ref} = 0 \tag{3.22}$$

V_- ist ungefähr 0 (siehe Anhang B). Daraus folgt

$$I_0 = x_0 * \frac{V_{ref}}{8 * R} \tag{3.23}$$

Entsprechende Gleichungen gelten für die Ströme I_1 bis I_3 durch die anderen Widerstände. Wir können jetzt die Kirchhoffsche Knotenregel (siehe Anhang B) auf den Knoten anwenden, der alle Widerstände verbindet. An diesem Knoten muss der Ausgangsstrom gleich der Summe der Eingangsströme sein. Damit ergibt sich

$$I = I_3 + I_2 + I_1 + I_0 \tag{3.24}$$

$$\begin{aligned} I &= x_3 * \frac{V_{ref}}{R} + x_2 * \frac{V_{ref}}{2 * R} + x_1 * \frac{V_{ref}}{4 * R} + x_0 * \frac{V_{ref}}{8 * R} \\ &= \frac{V_{ref}}{R} * \sum_{i=0}^{3} x_i * 2^{i-3} \end{aligned} \tag{3.25}$$

Jetzt können wir die Maschenregel auf die Masche anwenden, die aus R_1, y und V_- besteht. Da V_- ungefähr 0 ist, folgt:

$$y + R_1 * I' = 0. \tag{3.26}$$

Wir können die Knotenregel auf den Knoten anwenden, der I, I' und den invertierenden Signaleingang des Operationsverstärkers verbindet. Der Strom an diesem Eingang ist praktisch gleich Null. Die Ströme I und I' sind damit gleich ($I = I'$).

$$y + R_1 * I = 0 \tag{3.27}$$

Aus den Gleichungen (3.25) und (3.27) folgt:

$$y = -V_{ref} * \frac{R_1}{R} * \sum_{i=0}^{3} x_i * 2^{i-3} = -V_{ref} * \frac{R_1}{8 * R} * nat(x) \tag{3.28}$$

nat ist die natürliche Zahl, die durch den Bitvektor x dargestellt wird. Offensichtlich ist die Spannung am Ausgang proportional zur durch x dargestellten Zahl.

Positive Ausgangsspannungen und Bitvektoren, die Zahlen im Zweierkomplement darstellen, erfordern kleine Erweiterungen des D/A-Wandlers. Ähnlich wie bei Analog/Digitalwandlern ist die erforderliche Genauigkeit der Widerstände ein potentielles Problem, v.a. bei großen Wortbreiten von x. Als Alternative wird daher die Pulsbreitenmodulation verwendet (siehe Unterabschnitt 3.6.3).

$y(t)$ ist eine Funktion über einem diskreten Zeitbereich: sie stellt eine **Folge** von Spannungspegeln dar. Im Beispiel ist sie nur über ganzzahlige Zeitwerte definiert. Dies ist in Anwendungen unpraktisch, da wir meist den Ausgang der Schaltung aus Abb. 3.51 kontinuierlich betrachten. Daher werden D/A-Wandler oft durch eine ***zero-order hold*-Funktionalität** erweitert. Diese bewirkt, dass der Wandler den vorherigen Wert beibehält, bis der folgende Wert gewandelt wurde. Der D/A-Wandler aus Abb. 3.51 verhält sich in der Tat genau so, wenn wir die Stellung der Schalter bis zum nächsten diskreten Zeitpunkt nicht ändern. Damit liegt am Ausgang des Wandlers eine Treppenfunktion $y'(t)$ an, die der Folge $y(t)$ entspricht[15]. $y'(t)$ ist eine Funktion über dem kontinuierlichen Zeitbereich.

Wir betrachten als Beispiel das Ausgangssignal der Wandlung des Signals von Gleichung (3.3) mit einer Auflösung 0,125. Für diesen Fall zeigt Abb. 3.52 $y'(t)$. $y'(t)$ ist als Treppenfunktion einfacher zu visualisieren als die Funktion $y(t)$, die nur für diskrete Zeitpunkte definiert ist.

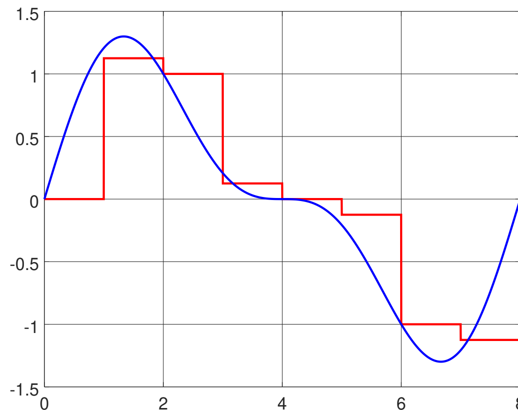


Abb. 3.52 $y'(t)$ (rot) erzeugt aus $e_3(t)$ (Gl. (3.3), blau), zu ganzzahligen Zeitpunkten abgetastet

D/A-Wandler ermöglichen eine Wandlung von zeit- und wertediskreten Signalen in Signale über kontinuierlichen Zeit- und Wertebereichen. Dabei stellen aber offensichtlich weder $y(t)$ noch $y'(t)$ die Werte des Eingangssignals zwischen den Abtastzeitpunkten dar.

[15] In der Praxis sind die Übergänge von einem Schritt zum nächsten wegen von Null verschiedener Anstiegs- und Fallzeiten des Signals nicht ideal, sondern benötigen eine gewisse Zeit zur Stabilisierung.

3.6.2 Abtasttheorem

Nehmen wir an, dass die Prozessoren in der beschriebenen Hardwareschleife eingehende Daten von den A/D-Wandlern unverändert zu den D/A-Wandlern weiterleiten. Wir könnten uns auch vorstellen, dass wir die Werte $x(t)$ auf einer CD speichern und daraus ein hervorragendes analoges Audiosignal erzeugen wollen. Wäre es möglich, die ursprüngliche analoge Spannung $e(t)$ (siehe Abb. 3.8, Abb. 3.21 und Abb. 3.50) an den Ausgängen des D/A-Wandlers wieder zu erzeugen?

Offensichtlich ist diese Rekonstruktion nicht möglich, wenn *Aliasing* gemäß Abb. 3.7 auf Seite 148 auftritt[16]. Wir nehmen also an, dass die Abtastrate mehr als doppelt so groß wie die höchste Frequenz ist, die bei der Aufspaltung des Eingangssignals in Sinuswellen auftritt (Abtastkriterium, siehe Gl. (3.8)). Erlaubt uns die Einhaltung dieses Kriteriums die Rekonstruktion des ursprünglichen Signals? Schauen wir uns das einmal genauer an!

Wenn man an einen D/A-Wandler eine diskrete Folge digitaler Werte anlegt, so wird daraus eine Folge analoger Werte erzeugt. Die Werte des Eingangssignals, die zwischen den Abtastzeitpunkten lagen, werden vom D/A-Wandler nicht erzeugt. Die beschriebene einfache *zero-order*-Haltefunktion (wenn vorhanden) würde nur eine Treppenfunktion erzeugen. Dies scheint darauf hinzudeuten, dass zur Rekonstruktion von $e(t)$ eine unendlich hohe Abtastrate erforderlich ist, damit alle Zwischenwerte erzeugt werden können.

Es könnte aber eine Art geschickter Interpolation geben, die Werte zwischen den Abtastzeitpunkten aus den Werten an den Abtastzeitpunkten berechnet. In der Tat zeigt die Abtasttheorie [440], dass ein zeitkontinuierliches Signal $z(t)$ aus einer Folge $y(t)$ analoger Werte erzeugt werden kann.

Seien $\{t_s\}, s = ..., -1, 0, 1, 2, ...$ die Abtastzeitpunkte des Eingangssignals. Wir nehmen eine konstante Abtastrate von $f_s = \frac{1}{T_s}$ $(\forall s : T_s = t_{s+1} - t_s)$ an. Die Abtasttheorie beschreibt die Approximation $z(t)$ von $e(t)$ mit Hilfe von $y(t)$ wie folgt:

$$z(t) = \sum_{s=-\infty}^{\infty} \frac{y(t_s) sin \frac{\pi}{T_s}(t - t_s)}{\frac{\pi}{T_s}(t - t_s)} \tag{3.29}$$

Diese Gleichung heißt **Shannon-Whittaker-Interpolation**. $y(t)$ ist der Anteil des Signals y zum Abtastzeitpunkt t_s. Der Einfluss dieses Anteils sinkt umso stärker, je weiter t von t_s entfernt ist. Dabei wird diese Absenkung durch die sogenannte *sinc*-Funktion gewichtet:

$$sinc(t - t_s) = \frac{sin(\frac{\pi}{T_s}(t - t_s))}{\frac{\pi}{T_s}(t - t_s)} \tag{3.30}$$

[16] Die Rekonstruktion mag möglich sein, wenn zusätzliche Informationen über das Signal vorliegen, z.B. wenn wir nur bestimmte Signaltypen betrachten.

Sie sinkt nicht monoton als eine Funktion von $|t - t_s|$. Dieser Gewichtungsfaktor wird verwendet, um Werte zwischen den Abtastzeitpunkten zu berechnen. Abb. 3.53 zeigt den Gewichtungsfaktor für den Fall $T_s = 1$.

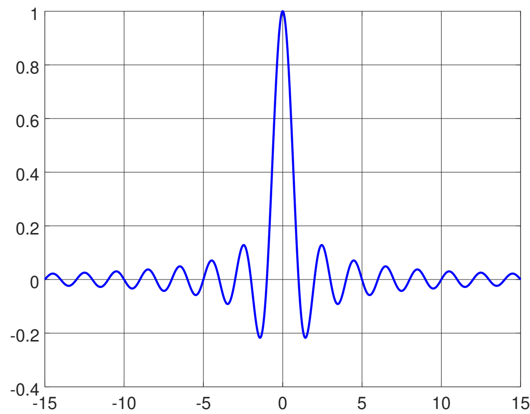


Abb. 3.53 Visualisierung der für die Interpolation verwendeten Gl. (3.30)

Mit der *sinc*-Funktion können wir die einzelnen Terme der Summe aus Abb. 3.29 berechnen. Abb. 3.54 und Abb. 3.55 zeigen die sich ergebenden Terme, wenn $e(t) = e_3(t)$ und die Verarbeitung der Daten diese unverändert lässt ($x(t) = w(t)$).

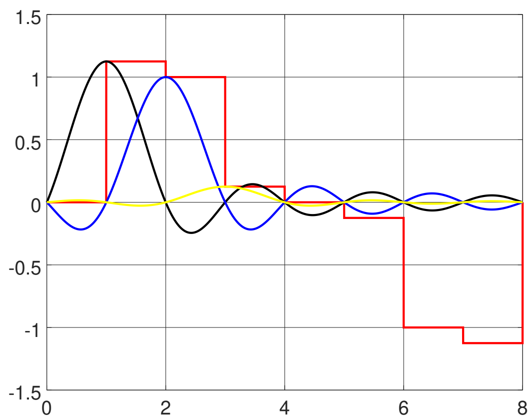


Abb. 3.54 $y'(t)$ (rot) und die ersten drei Terme von Gl. (3.29)

Zu jedem Abtastzeitpunkt t_s (in unserem Fall nimmt t_s nur ganzzahlige Werte an) wird $z(t_s)$ nur aus dem entsprechenden Wert $y(t_s)$ berechnet, da die *sinc*-Funktion in diesem Fall für alle anderen abgetasteten Werte den Wert Null annimmt. Zwischen den Abtastzeitpunkten tragen alle benachbarten diskreten Werte zum sich ergebenden Wert von $z(t)$ bei. Abb. 3.56 zeigt die sich ergebende Funktion $z(t)$, wenn $e(t) = e_3(t)$

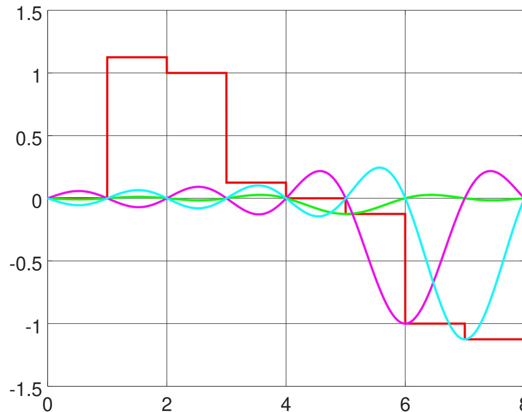


Abb. 3.55 $y'(t)$ (rot) und die letzten drei Terme von Gl. (3.29)

und die Verarbeitung der Daten wieder die Identitätsfunktion ($x(t) = w(t)$) ausführt. Diese Abbildung beinhaltet die Signale $e_3(t)$ (blau), $y'(t)$ (rot) und $z(t)$ (magenta).

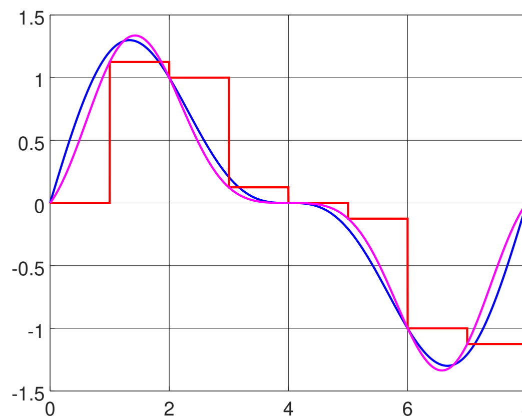


Abb. 3.56 $e_3(t)$ (blau), $y'(t)$ (rot) und $z(t)$ (magenta)

$z(t)$ entsteht, indem die Anteile aller Abtastzeitpunkte aus den Diagrammen 3.54 und 3.55 summiert werden. $e_3(t)$ und $z(t)$ sind sehr ähnlich.

Wie nahe könnten wir nun dem Ausgangssignal kommen, wenn wir Gleichung (3.29) implementieren? Die Abtasttheorie besagt (siehe z.B. [440]), dass **Gleichung (3.29) eine exakte Approximation berechnet**, wenn das Abtastkriterium eingehalten wird. Daher schauen wir uns nun an, wie wir die Gleichung (3.29) implementieren können.

Wie berechnen wir Gleichung (3.29) in einem elektronischen System? Mit einem digitalen Signalprozessor können wir diese Gleichung nicht über einem diskreten Zeitbereich berechnen, da diese Berechnung ein in der Zeit kontinuierliches Signal

erzeugen muss. Zunächst scheint die Berechnung einer solch komplexen Gleichung mit analogen Schaltungen schwierig zu sein.

Glücklicherweise ist die benötigte Berechnung eine sogenannte *Faltungsoperation* zwischen dem Signal $y(t)$ und der *sinc*-Funktion. Nach der klassischen Theorie der Fouriertransformationen entspricht eine Faltungsoperation im Zeitbereich einer Multiplikation mit einer zeitabhängigen Filterfunktion im Frequenzbereich. Diese Filterfunktion ist die Fourier-Transformierte der entsprechenden Funktion über den Zeitbereich. Daher lässt sich Gleichung (3.29) mit einem passenden Filter berechnen. Abb. 3.57 zeigt die Lage des Filters.

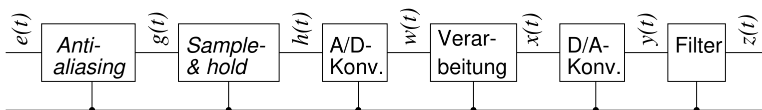
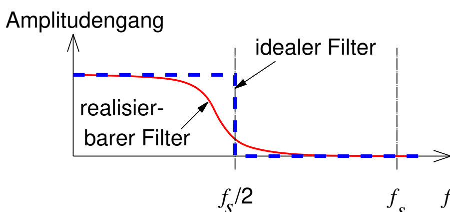


Abb. 3.57 Konvertierung des Signals $e(t)$ aus analogen Zeit- und Wertebereichen in digitale Bereiche und zurück

Nun bleibt noch die Frage zu beantworten, welche frequenzabhängige Filterfunktion die Fourier-Transformierte der *sinc*-Funktion ist. Die Berechnung der Fourier-Transformierten der *sinc*-Funktion ergibt eine Tiefpass-Filterfunktion [440]. Also müssen wir „nur" das Signal $y(t)$ mit einem Tiefpassfilter bearbeiten, um die Gleichung (3.29) zu berechnen. Damit filtern wir Frequenzen, wie es für den „idealen Filter" von Abb. 3.58 gezeigt ist. Hier fällt auf, dass die Darstellung der Funktion $y(t)$ als Summe von Sinuswellen einige sehr hohe Frequenzkomponenten benötigen würde. Daher entfernt eine solche Filterung Signalanteile mit sehr hohen Frequenzen, obwohl wir bereits einen *Anti-Aliasing*-Filter am Eingang vorgesehen hatten.

Abb. 3.58 Tiefpassfilter: ideal (*blau, gestrichelt*) und real (*rot, durchgezogene Linie)*



Es gibt noch ein weiteres Problem: ideale Tiefpassfilter existieren nicht. Daher müssen wir Kompromisse eingehen und Filter entwerfen, welche die Eigenschaften eines Tiefpassfilters annähern. Genau genommen müssen wir mit verschiedenen Unvollkommenheiten zurecht kommen, die eine präzise Rekonstruktion des Eingangssignals verhindern:

- Ideale Tiefpassfilter lassen sich nicht realisieren. Daher müssen wir Näherungen solcher Filter verwenden. Der Entwurf guter Kompromisse ist eine Kunst, die z.B. bei Audiogeräten weit verbreitet ist.
- Aus demselben Grund können wir Eingangsfrequenzen oberhalb der Nyquist-Frequenz nicht vollständig ausfiltern.
- Der Einfluss der Wertequantisierung wird in Abb. 3.56 deutlich. Durch die Wertequantisierung ist $e_3(t)$ manchmal verschieden von $z(t)$. Das von A/D-Wandlern eingebrachte Quantisierungsrauschen kann während der Erzeugung der Ausgabe nicht entfernt werden. Das Signal $w(t)$ am Ausgang des A/D-Wandlers wird daher durch das Quantisierungsrauschen gestört sein. Dieser Effekt betrifft allerdings nicht das Signal $h(t)$ am Ausgang von *Sample-and-hold*-Schaltungen.
- Gleichung (3.29) verwendet eine unendliche Summe, die auch Werte an zukünftigen Zeitpunkten beinhaltet. In der Praxis lassen sich Signale um eine endliche Zeit verzögern, um eine endliche Anzahl „zukünftiger" Werte zu erhalten. Unendliche Verzögerungen sind unmöglich. In Abb. 3.56 haben wir die Beiträge von Abtastzeitpunkten außerhalb des Diagramms nicht berücksichtigt.

Die Funktionalität der Tiefpassfilter zeigt die Leistungsfähigkeit analoger Schaltungen: es wäre im digitalen Bereich nicht möglich, das Verhalten analoger Filter zu implementieren, da die diskretisierten Zeit- und Wertebereiche hier Einschränkungen erfordern.

Viele Autoren waren an der Entwicklung der Abtasttheorie beteiligt. Damit sind auch viele Namen mit dem Abtasttheorem verbunden. Zu den wichtigsten Namen zählen Shannon, Whittaker, Kotelnikov, Nyquist und Küpfmüller. Daher sollte die Tatsache, dass ein Ursprungssignal rekonstruiert werden kann, einfach „Abtasttheorem" genannt werden, da es unmöglich ist, das Theorem nach allen beteiligten Wissenschaftlern zu benennen.

3.6.3 Pulsbreitenmodulation

In der Praxis besitzt die beschriebene Erzeugung von Analogsignalen einige Nachteile:

- Die Realisierung von D/A-Wandlern mit einer Menge von Widerständen ist schwierig. Die Genauigkeit der Widerstände muss exzellent sein. Die Abweichung des Widerstandes für das signifikanteste Bit von seinem nominellen Wert muss kleiner sein als die Auflösung des Wandlers. Beispielsweise muss die Abweichung bei einem 14-Bit-Wandler kleiner als 0,01% sein. Diese Genauigkeit ist schwer zu erreichen, insbesondere über den vollständigen Temperaturbereich. Wenn diese Genauigkeit nicht erreicht wird, ist der Wandler nicht linear, möglicherweise noch nicht einmal monoton.
- Es werden analoge Leistungsverstärker benötigt, um ausreichend Leistung für Motoren, Lampen, Lautsprecher usw. zu erzeugen. Analoge Leistungsverstärker, insbesondere die sogenannten Klasse-A Verstärker, sind sehr energieineffizient,

denn sie enthalten einen immer leitenden Pfad zwischen den beiden Anschlüssen der Betriebsspannung. Dieser Pfad bewirkt eine ständige Leistungsaufnahme, unabhängig vom Ausgabesignal. Das Verhältnis zwischen der tatsächlich genutzten (Wechselstrom-) Leistung und der verbrauchten elektrischen Leistung wäre daher sehr klein. Insgesamt betrachtet wäre die Effizienz von Audioverstärkern für kleine Lautstärken entsetzlich schlecht.

- Analoge Schaltkreise können auf ansonsten digitalen *Chips* nicht so einfach integriert werden. Externe analoge Komponenten würden die Kosten sehr in die Höhe treiben.

Daher ist die Pulsbreitenmodulation (engl. *pulse width modulation* (PWM)) sehr populär. Mit PWM starten wir mit einem digitalen Wert und generieren ein digitales Rechtecksignal, dessen Tastverhältnis dem zu wandelnden Wert entspricht. Abb. 3.59 zeigt digitale Signale mit Tastverhältnissen von 25% und 75%. Solche Signale können durch Fourierreihen wie in Gleichung (3.1) dargestellt werden. Für die Anwendung von PWM versuchen wir, die Effekte höherer Frequenzen zu eliminieren.

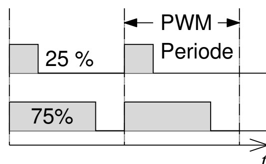


Abb. 3.59 Tastverhältnisse

PWM-Signale können erzeugt werden, indem wir einen Zähler mit einem Wert vergleichen, der in einem programmierbaren Register gespeichert ist (siehe Abb. 3.60). Eine hohe Spannung wird immer erzeugt, wenn der Wert im Zähler den Wert im Register übersteigt. Sonst wird eine Spannung nahe Null generiert. Das Taktsignal des Zählers muss programmierbar sein, damit die Basisfrequenz der PWM-Signale gewählt werden kann. In unserer Schaltung haben wir angenommen, dass die PWM-Frequenz für alle Ausgänge gleich ist. Die Register müssen mit den Werten geladen werden, die zu wandeln sind, typischerweise entsprechend der Abtastrate der Analogsignale.

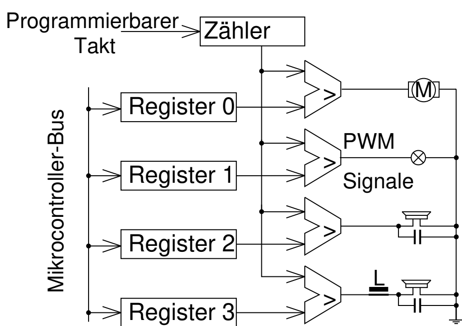


Abb. 3.60 Hardware für PWM-Ausgabe

Der Aufwand für die Filterung der höheren Frequenzanteile hängt von der Anwendung ab. Im Fall eines Motors geschieht eine entsprechende Mittelwertbildung im Motor selbst, da die sich bewegenden Teile eine gewisse Masse haben und da in der Regel Induktivitäten im Motor ebenfalls filternd wirken. Aus diesem Grund werden keine externen Komponenten benötigt (siehe Abb. 3.60). Im Falle von Lampen erfolgt die Mittelwertbildung im menschlichen Auge, solange die Frequenzen nicht zu niedrig sind. Einfache Klingeln kann man auch direkt antreiben. In anderen Fällen kann das Filtern von hohen Frequenzanteilen erforderlich sein. So kann die elektromagnetische Strahlung durch hohe Frequenzanteile Störungen bewirken und Audioanwendungen können ebenfalls eine Filterung verlangen. In Abb. 3.60 wurden

zwei Kondensatoren und eine Spule zur Filterung von hohen Frequenzanteilen vorgesehen. In unserem Beispiel gibt es vier PWM-Ausgänge. Es ist üblich, mehrere PWM-Ausgänge zu haben. Atmel 32-Bit AVR Microcontroller in der AT32UC3A-Serie haben beispielsweise sieben PWM-Ausgänge [26]. In der Praxis gibt es viele Optionen für das detaillierte Verhalten von PWM-Hardware.

Kompromisse sind erforderlich für die Wahl der Basisfrequenz (dem Reziproken der Periode) des PWM-Signals und des Filters. Die Basisfrequenz muss größer sein als der höchste Frequenzanteil im zu konvertierenden Signal. Höhere Basisfrequenzen machen den Entwurf des Filters – sofern erforderlich – einfacher. Zu hohe Basisfrequenzen erzeugen allerdings mehr elektromagnetische Störungen und einen hohen Energiebedarf, da jedes Schalten Energie verbrauchen wird. Typischerweise benutzt man Basisfrequenzen, die zwischen einem Faktor 2 und 10 größer sind als die größte Frequenz des zu wandelnden Signals.

3.6.4 Aktuatoren

Es gibt eine Vielzahl von Aktuatoren [151], von riesigen Aktuatoren, die in der Lage sind, Gewichte von mehreren Tonnen zu bewegen, bis hin zu winzigen Aktuatoren mit einer Größe im µm-Bereich.

Als Beispiel nennen wir hier eine bestimmte Art von Aktuatoren, die zukünftig an Bedeutung gewinnen werden: die Mikrosystemtechnologie erlaubt die Herstellung von winzigen Aktuatoren, die beispielsweise in den menschlichen Körper eingepflanzt werden können. Mit Hilfe solcher Miniaktuatoren kann die Menge von Medikamenten, die in den Körper eingebracht werden, genau an den aktuellen Bedarf angepasst werden. Damit ist eine viel bessere Medikamentenversorgung möglich als mit herkömmlichen Injektionen.

Beispiel 3.19: Abb. 3.61 zeigt einen Miniaturmotor, der mit Mikrosystemtechnologie hergestellt wurde. Seine Größe bewegt sich im µm-Bereich. Der drehbare Teil in der Mitte wird durch dreiphasige elektrostatische Kräfte gesteuert [478]. ∇

Aktuatoren sind für das Internet der Dinge wichtig. Es ist unmöglich, einen Überblick über alle verfügbaren Aktuatoren zu geben.

3.7 Elektrische Energie

Die allgemeinen Randbedingungen und Ziele für den Entwurf von eingebetteten und cyber-physikalischen Sytemen (siehe Seiten 9 bis 18 und Tabelle 1.2) müssen auch für den Entwurf von Hardware eingehalten werden. Unter den verschiedenen Zielen konzentrieren wir uns auf jene, welche elektrische Energie betreffen. Gründe hierfür sind in Tabelle1.1 auf Seite 15 angegeben.

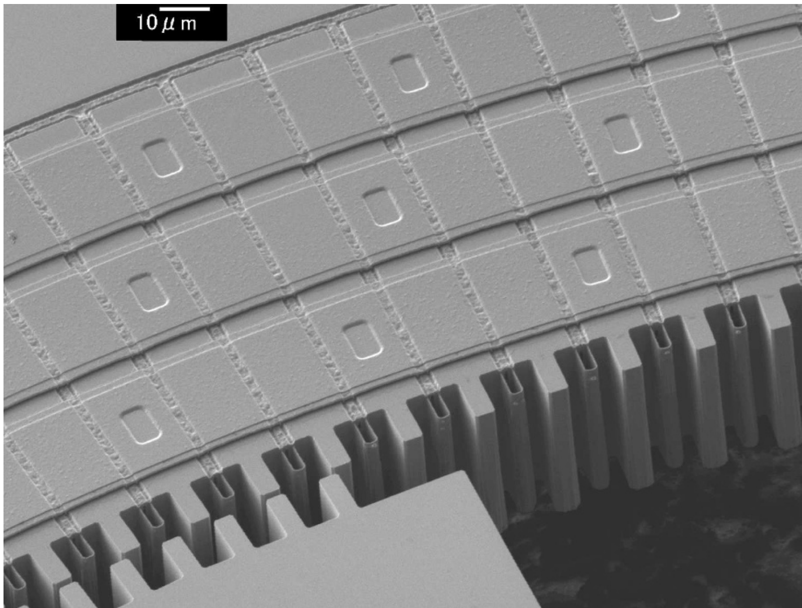


Abb. 3.61 Detail eines rotierenden Schrittmotors; oben: fester Teil, unten: beweglicher Teil; © Sarajlic et al. [478]

3.7.1 Energieerzeugung

Bei Geräten, die mit den Stromnetz verbunden sind, ist elektrische Energie relativ leicht verfügbar. Bei allen anderen Geräten muss die Energie mit anderen Techniken bereit gestellt werden. Insbesondere trifft dieses auf Sensornetzwerke zu, wo die Energie eine besonders knappe Ressource ist.

Batterien speichern Energie in Form von chemischer Energie. Ihre wesentliche Einschränkung ist, dass sie dorthin getragen werden müssen, wo die Energie gebraucht wird. Wenn wir diese Einschränkung vermeiden wollen, dann müssen wir Energie „ernten" (im Englischen als *energy harvesting* oder *energy scavenging* bezeichnet). Es gibt sehr viele Techniken, Energie zu ernten [575, 569], aber die Menge an erreichbarer Energie ist typischerweise sehr begrenzt:

- **Photovoltaik** ermöglicht die Umwandlung von Licht in elektrische Energie. Die Umwandlung basiert üblicherweise auf dem photovoltaischen Effekt von Halbleitern. Abb. 3.62 zeigt Beispiele.
- Der **piezoelektrische Effekt** kann benutzt werden, um mechanische Verformung in elektrische Energie zu wandeln. Piezoelektrische Zünder nutzen diesen Effekt.
- **Thermoelektrische Generatoren** (TEGs) erzeugen elektrische Energie beim Vorliegen von Temperaturgradienten. Mit ihnen kann z.B. die Wärme des menschlichen Körpers für die Energieerzeugung genutzt werden.

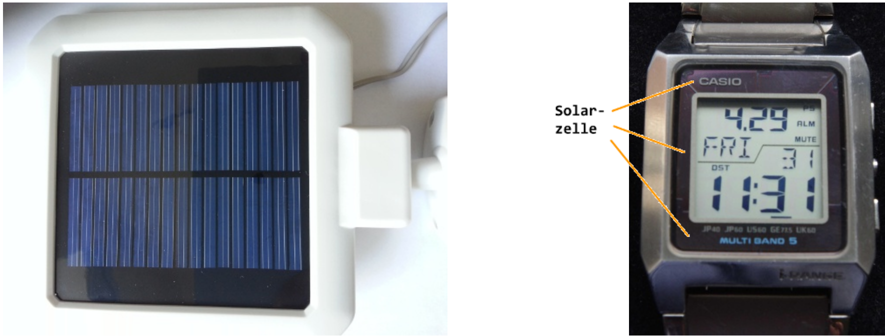


Abb. 3.62 Photovoltaisches Material: **links**: Solarmodul; **rechts**: Solaruhr

- **Kinetische Energie** kann in elektrische Energie gewandelt werden. Dies wird beispielsweise bei manchen Armbanduhren ausgenutzt.
- **Elektromagnetische Strahlung** kann ebenfalls in elektrische Energie gewandelt werden.
- Viele weitere physikalische Effekte erlauben die Konversion anderer Formen von Energie in elektrische Energie.

3.7.2 Energiespeicherung

In vielen Anwendungen eingebetteter Systeme ist keine permanente Versorgung mit Energie garantiert. Es kann aber möglich sein, Energie zu speichern. Beispielsweise gibt es folgende Methoden der Energiespeicherung:

1. Nicht wiederaufladbare Batterien können nur einmal benutzt werden und werden nicht weiter betrachtet.
2. Kondensatoren sind eine bequeme Möglichkeit, elektrische Energie zu speichern. Sie lassen sich extrem schnell aufladen, bieten sehr hohe Ausgangsströme, sind fast frei von Verlusten und lassen sich sehr häufig aufladen. Leider speichern sie nur eine begrenzte Menge an Energie, wobei mit Superkondensatoren (engl. *supercaps*) durchaus erhebliche Mengen an Energie gespeichert werden können.
3. Wiederaufladbare Batterien können wie Kondensatoren Energie speichern, allerdings in der Regel mit einer geringeren Zahl von Ladezyklen. Die Speicherung basiert auf chemischen Prozessen und die Entladung basiert auf der Umkehrung dieser Prozesse.

Aufgrund ihrer Bedeutung für eingebettete Systeme betrachten wir hier wiederaufladbare Batterien. Wenn wir Quellen von Energie in unserem Systemmodell einschließen wollen, dann benötigen wir Modelle von wiederaufladbaren Batterien. Verschiedene Modelle können benutzt werden. Sie unterscheiden sich im Umfang der eingeschlossenen Details und es gibt nicht das eine Modell, das alle Anforderungen erfüllt [467]. Die folgenden Modelle werden häufig benutzt:

- **Chemische und physikalische Modelle**: diese beschreiben die chemischen oder physikalischen Vorgänge im Detail. Solche Modelle können Differentialgleichungen mit vielen Parametern beinhalten. Diese Modelle sind für die Hersteller von Batterien gut geeignet, aber für Designer von eingebetteten Systemen, die in der Regel diese Parameter nicht kennen, typischerweise zu komplex.
- **Einfache empirische Modelle**: solche Modelle basieren auf einfachen Gleichungen, deren Parameterwerte bestimmt wurden. Das Gesetz von Peukert [451] ist ein beliebtes Modell. Danach berechnet sich die Lebensdauer einer Batterie zu:

$$\text{Lebensdauer} = C/I^{\alpha} \quad (3.31)$$

wobei $\alpha > 1$ das Ergebnis einer empirischen Anpassung ist. Das Gesetz von Peukert spiegelt wider, dass hohe Ströme typischerweise zu einer Verkürzung der Batterielebensdauer führen. Andere Details des Verhaltens von Batterien sind in diesem Modell nicht enthalten.
- **Abstrakte Modelle** modellieren mehr Details als die einfachen empirische Modell, aber gehen nicht auf die Ebene der chemischen Prozesse. Wir werden zwei solcher Modelle präsentieren:

  – Das elektrische Model von Chen and Ricón [96], das in Abb. 3.63 zu sehen ist. Nach diesem Modell steuert ein Ladestrom I_{Batt} eine Stromquelle im linken

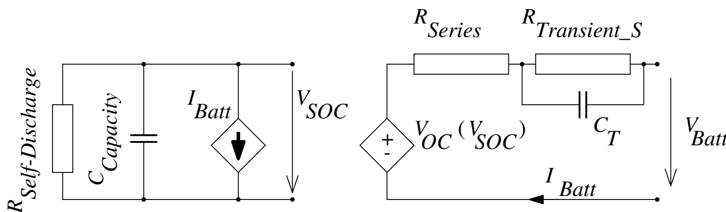


Abb. 3.63 Batteriemodell von Chen et al. (vereinfacht)

Teil des Schaltplans. Der von der Stromquelle erzeugte Strom ist gleich dem von rechts hereinkommenden Ladestrom. Dieser Strom lädt den Kondensator $C_{Capacity}$. Die jeweilige Ladung Q auf dem Kondensator wird *State of Charge* (SoC) genannt. Diese Ladung spiegelt sich in der Spannung V_{SOC} über den Anschlüssen des Kondensators wider, da für die Ladung Q die Kondensatorgleichung $Q = C_{Capacity} * V_{SOC}$ gilt. Der Widerstand $R_{Self-Discharge}$ modelliert die Selbstentladung des Kondensators, die selbst dann stattfindet, wenn kein Strom über die Anschlüsse des Kondensators abfließt.

Als nächstes betrachten wir die Spannung an den Anschlüssen des Kondensators, wenn der Strom durch diese Anschlüsse Null ist (die sogenannte **Leerlaufspannung** (engl. *open terminal output voltage*)). Die Spannung an den Anschlüssen wird typischerweise nichtlinear von V_{SOC} abhängen. Diese Abhängigkeit kann durch eine nichtlineare Funktion $V_{OC}(V_{SOC})$ modelliert wer-

den. V_{OC} nimmt mit steigendem Ausgangsstrom ab. Für einen konstanten Entladestrom modelliert die Reihenschaltung $R_{Series} + R_{Transient_S}$ den entsprechenden Spannungsverlust. Für kurze Stromspitzen ist ausschließlich R_{Series} relevant, da solche Spitzen durch C_T gepuffert werden. Für längere Entladephasen bestimmt die Zeitkonstante $R_{Transient_S} * C_T$ die Geschwindigkeit des Übergangs von dem Spannungsabfall über R_{Series} zum Spannungsabfall über $R_{Series} + R_{Transient_S}$. Der ursprüngliche Vorschlag von Chen et al. enthält ein zweites Widerstand/Kondensator-Paar, um das Verhalten bei Entladephasen genauer zu modellieren. Insgesamt beschreibt dieses Modell die Wirkung von hohen Entladeströmen auf die Ausgangsspannung, die Selbstentladung und die nichtlineare Abhängigkeit der Ausgangsspannung vom Ladezustand relativ gut. Einfachere Varianten dieses Modell existieren, sind aber nicht in der Lage, alle drei Effekte zu beschreiben.

– Gebräuchliche Batterien besitzen einen Erholungseffekt: wenn der Entladevorgang für eine Weile unterbrochen wird, wird zusätzliche Ladung verfügbar und die Spannung steigt meist auch wieder an. Dieser Effekt wird in Chens Modell nicht erfasst. Erfasst wird er im kinematischen Batteriemodell (KiBaM) von Manwell et al. [365]. Die Bezeichnung spiegelt die Analogie wieder, auf der es basiert. Im Modell gehen wir von zwei Behältern mit Ladung aus, wie in Abb. 3.64 gezeigt. Der rechte Behälter enthält eine Ladung y_1, die sofort verfügbar

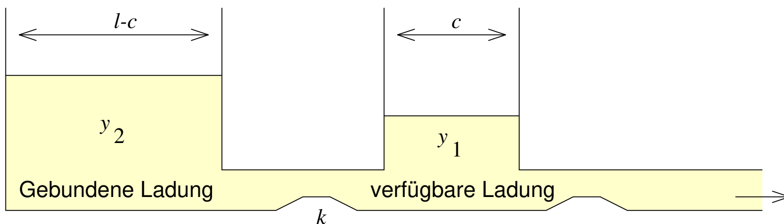


Abb. 3.64 Kinematisches Batteriemodell

ist. Der linke Behälter enthält eine Ladung y_2, die in der Batterie existiert, die aber in den rechten Behälter fließen muss, bevor sie genutzt werden kann. Eine Phase hoher Entladeströme könnte den rechten Behälter fast vollständig entleeren. Es dauert dann eine Weile, bevor wieder mehr Ladung zur Verfügung steht. Die Geschwindigkeit der Erholung wird durch den Parameter k bestimmt, der im mechanischen Modell dem Durchmesser des Rohrs entspricht, welches beide Behälter verbindet. Der elektrische Strom wird bei diesem Modell über die entsprechenden mechanischen Größen berechnet. Dieses Modell beschreibt die Erholung der Batterien mit einiger Genauigkeit, aber es kann Selbstentladung und das Verhalten bei Stromimpulsen im Gegensatz zu Chens Modell nicht erfassen. Darüber hinaus werden auch die Alterung oder temperaturabhängige Effekte nicht modelliert. Aus dem kinematischen Modell heraus kann man bestimmen, wie man die gespeicherte Energie am besten nutzt. Beispiels-

weise wurde gezeigt, dass es sinnvoll ist, bei der drahtlosen Datenübertragung Intervalle vorzusehen, in denen die Übertragung abgeschaltet wird, weil sich so die Batterien erholen können [144].
Insgesamt zeigen die beiden Modelle, dass man Modelle wählen muss, welche die Effekte, die man berücksichtigen möchte, auch erfassen.

- Es kann **gemischte Modelle** geben, die teils auf abstrakten und teils auf chemisch/physikalischen Modellen basieren.

3.7.3 *Effiziente Nutzung elektrischer Energie*

Energieeffizienz von Hardwareplattformen

Die Hardwaretechnologien, welche in diesem Kapitel betrachtet wurden, unterscheiden sich hinsichtlich ihrer Energieeffizienz, wie in Abb. 3.65 zu sehen ist.

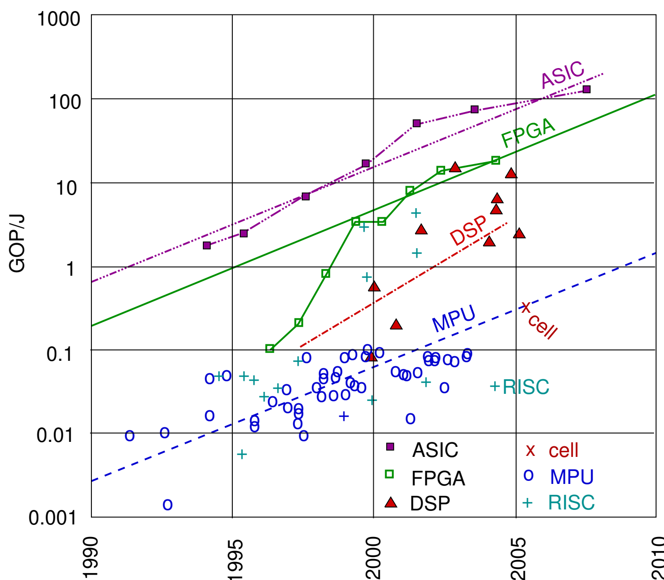


Abb. 3.65 Energieeffizienz (©De Man und Philips)

Die Abbildung zeigt einen Vergleich der Energieeffizienz als Anzahl der Operationen per Einheit der Energie (GOP/J) als Funktion der Hardwaretechnologie (ASICs, Prozessoren, FPGAs) und der Zeit, die jeweils ein Spiegelbild der verfügbaren Miniaturisierung in der Fertigungstechnologie ist[17]. In diesem Kontext

[17] Die Abbildung ist eine Approximation von Informationen von H. De Man [364] und basiert auf Informationen von Philips.

verstehen wir unter Operationen beispielsweise 32-Bit Additionen. Offensichtlich können pro Joule immer mehr Operationen ausgeführt werden, da kleinere Strukturgrößen in der Chipfertigung zu effizienteren *Chips* führen. Für ein festes Jahr und eine damit verfügbare Fertigungstechnologie ist aber die Energieeffizienz am höchsten für ASICs. Für rekonfigurierbare Schaltungen wie FPGAs (siehe Seite 181) ist die Effizienz ca. eine Größenordnung geringer. Für programmierbare Prozessoren ist sie noch geringer. Dagegen offerieren Prozessoren die größte Flexibilität hinsichtlich ihrer Programmierbarkeit. Etwas Flexibilität gibt es auch noch für FPGAs, aber diese ist begrenzt auf die Größe von Anwendungen, die sich auf FPGAs abbilden lassen. Für ASICs gibt es keine Flexibilität. Ein Abwägen zwischen Flexibilität und Effizienz existiert selbst innerhalb der Klasse der Prozessoren: Prozessoren, die für einen bestimmten Anwendungsbereich optimiert sind, wie z.B. digitale Signalprozessoren, erreichen eine Effizienz nahe der von FPGAs. Für allgemeine Prozessoren ist die Effizienz am geringsten. Dies zeigt sich in Abb. 3.65 daran, dass die Effizienz von x86-ähnlichen Prozessoren (siehe „MPU"-Einträge) geringer ist als die von DSP-Prozessoren.

Für Abb. 3.65 ist nicht genau bekannt, welche Anwendungen untersucht wurden und wie Anwendungen auf die Prozessoren abgebildet wurden. Jüngere und detailliertere Publikationen erlauben es, diese Vergleiche umfassender zu gestalten. Eine Übersicht über Vergleiche, die auch GPUs einschließt, wurde von Mittal et al. publiziert [399]. Die Übersicht enthält eine Liste von 28 Publikationen, denen zufolge GPUs in den jeweiligen Anwendungen energieeffizienter waren als CPUs sowie zwei Publikationen, in denen das Gegenteil gefunden wurde. Die Übersicht enthält auch eine Liste von 26 Publikationen, denen zufolge FPGAs energieeffizienter waren als GPUs und eine Publikation, die über das Gegenteil berichtet. Beispielsweise fanden Hamada et al. [201] für eine n-Körper-Simulation der Gravitation, dass mit FPGAs fünfzehn Mal mehr Operationen pro Watt ausgeführt werden konnten als mit GPUs. Verglichen mit CPUs betrug der Faktor sogar 34. Die genauen Faktoren hängen sicherlich von der Anwendung ab, aber als Daumenregel können wir folgendes festhalten: Wenn wir auf Spitzenwerte für die Energie- und Leistungseffizienz zielen, sollten wir ASICs benutzen. Wir sollten FPGAs nehmen, wenn ASICs z.B. aus Kostengründen ausscheiden. Wir sollten GPUs nehmen, wenn FPGAs nicht in Frage kommen, weil z.B. Kompetenz zur Programmierung fehlt. Bei Prozessoren sind heterogene Prozessoren effizienter als homogene. Für abgegrenzte Anwendungsbereiche lassen sich weitere Informationen gewinnen.

Mobiltelefone

Als einen speziellen Anwendungsbereich von eingebetteten Systemen (siehe Seiten 4 bis 9) betrachten wir nunmehr Telekommunikation und Mobiltelefone. Bei Mobiltelefonen steigen die Anforderungen an die Rechenleistung (Performanz) sehr stark an, v.a. für Multimediaanwendungen. De Man und Philips haben geschätzt, dass fortgeschrittene Multimediaanwendungen etwa 10 bis 100 Milliarden Operationen pro Sekunde benötigen. Abb. 3.65 zeigt, dass fortgeschrittene Hardwaretechnolo-

gien im Jahr 2007 etwa diese Anzahl von Operationen pro Joule (= Ws) bereit stellten. Mithin boten die effizientesten Hardwareplattformen so gerade eben die Effizienz, die benötigt wurde. Standardprozessoren (siehe Einträge für MPU und RISC) waren hoffnungslos ineffizient. Dies bedeutete auch, dass alle Möglichkeiten einer Steigerung der Energieeffizienz genutzt werden mussten. In den letzten Jahren wurde die Energieeffizienz verbessert. Allerdings wurden alle diese Verbesserungen wieder „aufgebraucht", um eine bessere Qualität bereit zu stellen, z.B. durch eine Erhöhung der Auflösung von Fotos bzw. Videos sowie durch größere Bandbreiten bei der Kommunikation.

Detaillierte Analysen des Leistungsverbrauchs wurden von Berkel [47] und von Carroll et al. [84] publiziert. Eine neuere Analyse, die auch Mobiltelefone mit LTE einbezieht, wurde von Dusza et al. [144] veröffentlicht. Es wurde eine Leistungsaufnahme von bis zu vier Watt beobachtet. Das Display war dabei für eine Leistungsaufnahme von bis zu einem Watt verantwortlich, je nach Displayhelligkeit.

Eine bessere Batterietechnologie würde es uns erlauben, über längere Zeiträume mehr Leistung zu verbrauchen, aber aus thermischen Gründen können wir in der nächsten Zeit kaum über die aktuelle Leistungsaufnahme hinaus gehen. Zur Vermeidung von Überhitzung ist es inzwischen Standard, Mobiltelefone mit Temperatursensoren auszustatten und ggf. zu drosseln. Dabei könnten größere Mobiltelefone mehr Wärme an die Umwelt abgeben und somit auch eine größere Leistung verbrauchen. Die Leistungsaufnahme sollte aber aus Umweltschutzgründen nicht zu stark steigen.

Technologievorhersagen wurden auch als sogenannte *International Technology Roadmap for Semiconductors* (ITRS) publiziert. In der ITRS Ausgabe von 2013 [262] wird explizit bestätigt, dass Mobiltelefone die technologische Entwicklung vorantreiben: „*System integration has shifted from a computational, PC-centric approach to a highly diversified mobile communication approach. The heterogeneous integration of multiple technologies in a limited space (e.g., GPS, phone, tablet, mobile phones, etc.) has truly revolutionized the semiconductor industry by shifting the main goal of any design from a performance driven approach to a reduced power driven approach. In few words, in the past performance was the one and only goal; today minimization of power consumption drives IC design*".

Sensornetzwerke

Sensornetzwerke, wie sie beim Internet der Dinge benutzt werden, sind ein zweiter Anwendungsfall. Bei Sensornetzwerken ist möglicherweise noch weniger elektrische Energie vorhanden als bei Mobiltelefonen. Daher ist die Energieeffizienz äußerst wichtig, einschließlich der Effizienz der Kommunikation [542].

3.8 Sichere Hardware

Die allgemeinen Anforderungen an eingebettete Systeme enthalten oft auch Anforderungen an die Informationssicherheit (siehe Seite 9). Insbesondere ist diese Sicherheit für das Internet der Dinge wichtig. Wenn Sicherheit einen hohen Stellenwert hat, dann muss aus diesem Grund evtl. spezielle Hardware entwickelt werden. Sicherheit muss möglicherweise v.a. für die Kommunikation und die Speicherung gewährleistet sein [310]. Sicherheit kann i.d.R. nicht unter allen Umständen garantiert werden, sondern man muss sich überlegen, welche **Angriffe** oder **Attacken** möglich sind und jeweils für solche Angriffe Abwehrmaßnahmen bereit stellen. Es muss dann der Nachweis der Sicherheit trotz dieser Attacken garantiert werden. Man kann zwischen Angriffsklassen unterscheiden (in Anlehnung an Ravi et al. [301]):

- **Softwareangriffe** basieren ausschließlich auf der Ausführung von Software. Die Entwicklung von Trojanischen Pferden ist ein Beispiel eines solchen Angriffs. Es können auch Fehler in der Software ausgenutzt werden. Pufferüberläufe sind eine häufige Ursache für eine Verletzlichkeit der Software. Ein spezieller Fall sind **Seitenkanalangriffe**, bei denen man versucht, Wege des Informationsflusses zu eröffnen, die im Entwurf eigentlich nicht vorgesehen waren. Seitenkanalangriffe auf Softwarebasis sind relativ schwierig zu realisieren. Allerdings sind beispielsweise Seitenkanalangriffe durch Zeitanalyse möglich, wenn die Ausführungszeit von Software datenabhängig ist[18]. Sicherheitsrelevante Algorithmen sollten so entworfen sein, dass die Ausführungszeit nicht von den Daten abhängt. Diese Anforderung berührt sogar die Realisierung von Rechnerarithmetik: Maschinenbefehle sollten keine datenabhängigen Ausführungszeiten haben.
- **Physikalische Angriffe** sind i.d.R. ebenfalls Seitenkanalangriffe. Man kann sie folgendermaßen klassifizieren:
  - **Physische Manipulationen**: Beispielsweise kann man einen Siliziumchip öffnen und analysieren. Der erste Schritt hierbei besteht daran, das Gehäuse zu entfernen. Als nächstes kann man den *Chip* optisch analysieren oder die Leiterbahnen mit feinen Nadeln kontaktieren. Solche Angriffe sind schwierig, aber sie verraten viele Details der *Chips*.
  - Eine **Analyse der Stromaufnahme** bietet eine weitere Möglichkeit eines physischen Seitenkanalangriffs. Die Analyse kann eine einfache Analyse (SPA) oder eine differentielle Analyse (DPA) umfassen. In manchen Fällen ist es möglich, Datenschlüssel durch eine einfache Analyse der Stromaufnahme herauszufinden. In anderen Fällen können fortgeschrittene statistische Methoden benutzt werden, um Schlüssel aus kleinen statistischen Schwankungen der Stromaufnahme herauszufinden.
  - Eine **Analyse der elektromagnetischen Strahlung** bietet eine weitere Möglichkeit, physische Seitenkanäle zu eröffnen.

---

[18] Seitenkanal-Angriffe unter Ausnutzung von Zeitinformationen wurden unter den Namen Spectre und Meltdown sehr bekannt, da v.a. moderne Prozessoren mit spekulativer Ausführung davon betroffen sind, siehe u.a. https://de.wikipedia.org/wiki/Spectre_(Sicherheitslücke).

Die Angriffe können von verschiedenen Klassen von Personen versucht werden und wiederum unterschiedliche Klassen von Personen können ein Interesse daran haben, diese Angriffe zu verhindern. Beispielsweise können Angriffe durch reguläre Benutzer gestartet werden, die unberechtigten Zugriff auf ein Netzwerk oder eine geschützte Mediendatei haben möchten.

Wir können zwischen den folgenden **Gegenmaßnahmen** unterscheiden:

- Bei der Softwareentwicklung muss man sich der Sicherheitsrisiken bewusst sein, um Gegenmaßnahmen gegen Angriffe ergreifen zu können.
- Mit physischen Mitteln wie einer Abschirmung oder Sensoren, die Manipulationsversuche erkennen, können Geräte geschützt werden.
- Geräte können so entwickelt werden, dass Datenmuster einen möglichst geringen Einfluss auf die Stromaufnahme haben. Hierfür sind teilweise spezielle Schaltungen erforderlich, wie sie meistens in komplexen *Chips* nicht eingesetzt werden.
- Logische Sicherheit, die i.d.R. mit kryptographischen Methoden erreicht wird: diese basieren entweder auf symmetrischen oder auf asymmetrischen Codes.
  - Bei symmetrischen Codes benutzen Sender und Empfänger denselben Schlüssel, um Nachrichten zu ver- und entschlüsseln.
  - Bei asymmetrischen Codes werden Nachrichten mit einem öffentlichen Schlüssel verschlüsselt und mit einem privaten Schlüssel entschlüsselt. So sind RSA- und Diffie-Hellman-Verschlüsselung asymmetrische Verschlüsselungen.
  - Werden Nachrichten *Hash Codes* zugefügt, so können Modifikationen der Nachricht erkannt werden.

  Aufgrund der begrenzten Performanz von Prozessoren in eingebetteten Systemen gibt es teilweise spezielle Maschinenbefehle zur Verschlüsselung und Entschlüsselung, welche die Effizienz steigern. Es gibt auch spezialisierte Lösungen wie die *TrustZone* von ARM. „*At the heart of the TrustZone approach is the concept of secure and non-secure worlds that are hardware separated, with non-secure software blocked from accessing secure resources directly. Within the processor, software either resides in the secure world or the non-secure world; a switch between these two worlds is accomplished via software referred to as the secure monitor (Cortex-A) or by the core logic (Cortex-M). This concept of secure (trusted) and non-secure (non-trusted) worlds extends beyond the processor to encompass memory, software, bus transactions, interrupts and peripherals within an SoC*" (siehe https://www.arm.com/products/security-on-arm/trustzone).
  Der Kalray MPPA2®-256 Mehrkern-Prozessor-*Chip* enthält 128 Krypto-Coprozessoren, die mit einer Matrix von „regulären" 64-Bit VLIW-Kernen verbunden sind (siehe http://www.kalrayinc.com/kalray/products/).

  Beim Entwurf von Gegenmaßnahmen gibt es Herausforderungen [301]:

1. Die **Performanzlücke**: aufgrund der begrenzten Performanz von eingebetteten Systemen sind fortgeschrittene Verschlüsselungstechniken möglicherweise zu langsam, v.a. wenn hohe Datenraten verarbeitet werden sollen.

2. Die **Energielücke:** fortgeschrittene Verschlüsselungstechniken benötigen eine große Menge an Energie. Diese große Menge ist in einem mobilen System möglicherweise nicht vorhanden. Insbesondere müssen *smart cards* mit einer besonders geringen Menge an Energie auskommen.
3. **Mangelnde Flexibilität:** häufig sind viele verschiedene Sicherheitsprotokolle in einem System erforderlich und diese Protokolle müssen von Zeit zu Zeit aktualisiert werden. Dies erschwert die Benutzung von speziellen Hardwarebeschleunigern für die Verschlüsselung.
4. **Manipulationsschutz:** es ist nicht einfach, Maßnahmen gegen Angriffe von außen einzubauen. Beispielsweise kann es schwierig sein, eine Stromaufnahme zu garantieren, die unabhängig von dem verarbeiteten Schlüssel ist.
5. **Verifikationslücke:** eine Überprüfung der Sicherheitsmaßnahmen erfordert einen zusätzlichen Aufwand bei Entwurf und Herstellung,
6. **Kosten:** Sicherheitsmaßnahmen erhöhen die Kosten von Systemen.

Ravi et al. haben diese Herausforderungen für ein *Secure Socket Layer* (SSL) Protokoll im Detail analysiert [301].

Weitere Informationen zu sicherer Hardware kann man beispielsweise in einem Buch von Gebotys [180] und in Tagungsbänden einer Serie von Workshops finden, die sich mit diesem Thema beschäftigen (siehe z.B. [183]).

3.9 Aufgaben

Die folgenden Aufgaben sollten entweder zu Hause oder während einer Anwesenheitsphase nach dem *flipped classroom*-Konzept [376] bearbeitet werden:

3.1: Lokal verfügbare kleine Roboter sollten benutzt werden, um die Komponenten in einer Regelschleife wie in Abb. 3.2 kennenzulernen. Diese Roboter sollten Sensoren und Aktuatoren enthalten und sie sollten ein Programm ausführen, das eine Regelschleife realisiert. Beispielsweise kann ein optischer Sensor genutzt werden, um einen Roboter einer schwarzen Linie auf dem Boden folgen zu lassen. Die Details dieser Aufgabe hängen von der Verfügbarkeit von Robotern ab.

3.2: Was ist ein „Signal”?

3.3: Mit welchem Schaltkreistyp kann der Übergang von einer kontinuierlichen zu einer diskreten Zeit bewerkstelligt werden?

3.4: Was ist der Inhalt des Abtasttheorems?

3.5: Angenommen, wir haben ein Eingangssignal x, das aus einer Summe von Sinussignalen mit einer Frequenz von 1,75 kHz und 2 kHz besteht. Wir tasten x mit einer Rate von 3 kHz ab. Können wir aus den Abtastwerten das Originalsignal rekonstruieren? Erklären Sie Ihr Ergebnis!

3.6: Der Übergang auf diskrete Werte basiert auf Analog/Digital- (A/D-) Wandlern. Entwickeln Sie einen *Flash*-A/D-Wandler für positive und negative Eingangsspannungen! Die Ausgabewerte sollten als 3-Bit-Zweierkomplementzahlen kodiert sein, sodass wir zwischen acht verschiedenen Spannungsbereichen unterscheiden können.

3.7: Angenommen, wir arbeiten mit einem 4-Bit-A/D-Wandler nach dem Wägeprinzip. Der Eingangsspannungsbereich reicht von V_{min} =1 V (="0000") bis V_{max} =4,75 V (="1111"). Welche Schritte werden benötigt, um Spannungen von 2,25 V, 3,75 V und 18 V zu wandeln? Zeichnen Sie ein Diagramm wie in Abb. 3.12, welches die sukzessive Approximation an diese Spannungen zeigt!

3.8: Vergleichen Sie den *Flash*-basierten Wandler mit dem Wandler nach dem Wägeprinzip. Nehmen Sie an, Sie möchten zwischen n verschiedenen Spannungsintervallen unterscheiden. Tragen Sie die jeweilige Komplexität in die Tabelle 3.2 mit der O-Notation ein.

Tabelle 3.2 Komplexität von A/D-Wandlern

	Flash-Konverter	Konverter nach dem Wägeprinzip
Zeitkomplexität		
Hardwarekomplexität		

3.9: Angenommen, wir legen ein Sinussignal an den Eingang des Konverters aus Aufgabe 3.6 an. Zeichnen Sie den Verlauf des Quantisierungsrauschens für diesen Fall!

3.10: Welches sind typische Eigenschaften von DSP-Prozessoren?

3.11: Welche Komponenten einhält ein FPGA (Beispiel: Xilinx-FPGAs)? Welche Komponenten werden benutzt, um Boolesche Funktion zu realisieren? Wie werden FPGAs konfiguriert? Sind FPGAs energieeffizient? Für welche Anwendungen sind FPGAs gut?

3.12: Was ist die Grundidee von VLIW-Prozessoren?

3.13: Was ist eine „*single-ISA heterogeneous multi-core architecture*“ ? Warum nutzt man solche Architekturen?

3.14: Erläutern Sie die Begriffe „GPU“ und „MPSoC“!

3.15: Einige FPGAs unterstützen die Realisierung aller Booleschen Funktionen von sechs Variablen. Wie viele solcher Funktionen gibt es? Wir ignorieren dabei, dass sich einige Funktionen nur um die Umbenennung von Variablen unterscheiden.

3.16: Im Zusammenhang mit Speichern sagen wir manchmal *small is beautiful*. Welchen Grund kann es dafür geben?

3.17: Einige Ebenen der Speicherhierarchie werden vor Programmierern versteckt. Warum sollten Programmierer sich trotzdem um die Existenz solcher Ebenen kümmern?

3.18: Was ist ein *Scratchpad*-Speicher (SPM)? Wie können wir erreichen, dass ein Speicherobjekt in einem solchen Speicher abgelegt wird?

3.19: Entwickeln Sie das folgende FlexRay™Cluster: Das Cluster besteht aus den fünf Knoten A, B, C, D und E. Alle Knoten sollen mit zwei Kanälen verbunden sein. Das Cluster benutzt eine Bus-Topologie. Die Knoten A, B und C führen eine sicherheitskritische Task aus und daher sollten ihre Anforderungen des Busses bei 20 Macroticks garantiert sein. Das nachfolgende wird erwartet:

- Laden Sie den levi FlexRay Simulator [495]! Entpacken Sie die zip-Datei und installieren Sie den Simulator!
- Starten Sie den Simulator indem Sie leviFRP.jar ausführen!
- Entwerfen Sie das Flexray-Cluster als Simulatormodell!
- Konfigurieren Sie den Kommunikationszyklus so, dass die Knoten A, B und C nach einer maximaler Verzögerung von 20 *macroticks* einen Buszugriff haben! Die Knoten D and E sollen nur das dynamische Segment nutzen.
- Konfigurieren Sie die Busanforderungen! Knoten A soll in jedem Zyklus eine Nachricht senden. Die Knoten B and C senden bei jedem zweiten Zyklus eine Nachricht. Knoten D sendet in jedem Zyklus eine Nachricht der Länge von 2 Mikrozyklen und Knoten E sendet in jedem zweiten Zyklus eine Nachricht der Länge 2 Minislots
- Starten Sie die Visualisierung und prüfen Sie, ob die Busanforderungen von A, B und C garantiert werden.
- Vertauschen Sie die Positionen von D und E! Welches Verhalten beobachten Sie?

3.20: Entwickeln Sie den Schaltplan eines 3-Bit Digital/Analog-Wandlers! Der Wandler soll Bitvektoren x, die jeweils eine positive 3-Bit-Zahl darstellen, wandeln können. Beweisen Sie, dass die Ausgangsspannung proportional zu dem durch x dargestellten Wert ist! Wie würden Sie die Schaltung modifizieren, wenn x Zweierkomplementzahlen darstellt?

3.21: Der Schaltkreis in Abb. B.4 im Anhang B ist ein Verstärker, der die Eingangsspannung V_1 verstärkt:

$$V_{out} = g_{closed} * V_1$$

Berechnen Sie die Verstärkung g_{closed} für die Schaltung in Abb. B.4 als Funktion von R und R_1!

3.22: Wie unterscheiden sich verschiedene Hardwaretechnologien hinsichtlich ihrer Energieeffizienz?

3.23: Die Energieeffizienz wird manchmal in Milliarden Operationen pro Sekunde und Watt gemessen. Wie unterscheidet sich dieses Maß von dem Maß, das in Abb. 3.65 benutzt wurde?

3.24: Warum ist es so wichtig, eingebettete Systeme zu optimieren? Vergleichen Sie verschiedene Hardwaretechnologien in Bezug auf ihre Energieeffizienz!

3.25: Angenommen, ein Mobiltelefon benutzt eine Lithiumbatterie mit einer Kapazität von 3200 mAh. Die nominelle Spannung ist 3,7 V. Wie lange würde es dauern, bis die Batterie leer ist, wenn wir konstant 1 W Leistung entnehmen? Alle sekundären Effekte (wie eine abnehmende Batteriespannung) sollen ignoriert werden.

3.26: Warum ist es schwierig, Informationssicherheit bei eingebetteten Systemen zu gewährleisten?

3.27: Was ist ein Seitenkanalangriff? Nennen Sie Beispiele!

Open Access Dieses Kapitel wird unter der Creative Commons Namensnennung 4.0 International Lizenz (http://creativecommons.org/licenses/by/4.0/deed.de) veröffentlicht, welche die Nutzung, Vervielfältigung, Bearbeitung, Verbreitung und Wiedergabe in jeglichem Medium und Format erlaubt, sofern Sie den/die ursprünglichen Autor(en) und die Quelle ordnungsgemäß nennen, einen Link zur Creative Commons Lizenz beifügen und angeben, ob Änderungen vorgenommen wurden.

Die in diesem Kapitel enthaltenen Bilder und sonstiges Drittmaterial unterliegen ebenfalls der genannten Creative Commons Lizenz, sofern sich aus der Abbildungslegende nichts anderes ergibt. Sofern das betreffende Material nicht unter der genannten Creative Commons Lizenz steht und die betreffende Handlung nicht nach gesetzlichen Vorschriften erlaubt ist, ist für die oben aufgeführten Weiterverwendungen des Materials die Einwilligung des jeweiligen Rechteinhabers einzuholen.

Kapitel 4
Systemsoftware

Zur Bewältigung der Komplexität des Entwurfes von Anwendungen eingebetteter Systeme ist die Wiederverwendung von Komponenten ein wichtiges Hilfsmittel. Im Sinne einer plattformbasierten Entwurfsmethode (siehe Seite 322) ist neben vorhandener Hardware auch Software einzusetzen. Diese Komponenten beinhalten das Wissen früherer Entwicklungen und stellen sogenanntes **geistiges Eigentum** (engl. *Intellectual Property* (IP)) dar. Zu den wiederverwendbaren Standard-Softwarekomponenten zählen Komponenten der Systemsoftware wie eingebettete **Betriebssysteme** (engl. *Operating Systems* (OS)) und sogenannte ***Middleware***. Im diesem Kapitel beschreiben wir zunächst allgemeine Anforderungen an Betriebssysteme für eingebettete Systeme. Dazu gehören insbesondere die Echtzeitfähigkeit und die Adaptierbarkeit an die jeweilige Anwendung. Beim exklusiven Zugriff auf Ressourcen kann es zur Prioritätsumkehr kommen, die für Echtzeitsysteme schwere Probleme verursachen kann. Diese können mit Zugriffsprotokollen für diese Ressourcen vermieden werden. Mit der Prioritätsvererbung, dem *Priority Ceiling*-Protokoll und dem *Stack Resource*-Protokoll stellen wir drei wichtige Protokolle vor. Ein eigener Abschnitt ist dem Echtzeitkern ERIKA gewidmet, der v.a. für Mikrocontroller gedacht ist. Ein weiterer Abschnitt ist den Anpassungen gewidmet, mit denen Linux in eingebetteten Systemen eingesetzt werden kann. Das Kapitel schließt mit Hinwei-

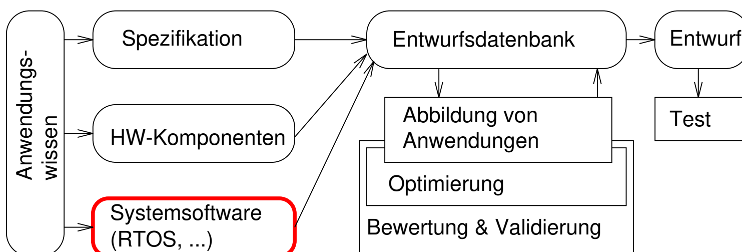


Abb. 4.1 Vereinfachter Entwurfsfluss

© Der/die Autor(en) 2021

P. Marwedel, *Eingebettete Systeme*, https://doi.org/10.1007/978-3-658-33437-6_4

sen auf weitere wiederverwendbare Softwarekomponenten, insbesondere *Hardware Abstraction Layers* (HALs), Kommunikationssoftware und Echtzeitdatenbanken. Mit der Beschreibung eingebetteter Betriebssysteme und von *Middleware* in diesem Kapitel entsprechen wir dem Entwurfsfluss (siehe auch Abb. 4.1).

4.1 Eingebettete Betriebssysteme

4.1.1 Allgemeine Anforderungen

Mit Ausnahme sehr einfacher Systeme benötigen eingebettete Anwendungen ein geeignetes Betriebssystem für die Ein/Ausgabe, für das *Scheduling* (d.h. die Ablaufplanung) und das Umschalten zwischen der Ausführung verschiedener Codeobjekte. Durch das Umschalten, Kontextwechsel (engl. *context switching*) genannt, entsteht der Eindruck, dass jedes Codeobjekt einen eigenen Prozessor besitzt. Bei Codeobjekten unterscheiden wir zwischen Prozessen und *Threads*. Zunächst definieren wir Prozesse:

Definition 4.1 (in Anlehnung an Tanenbaum [525]): Ein **Prozess** ist ein Programm (oder ein Teil eines Programms) einschließlich aller zugehörigen Speicherinhalte.

Weitere Informationen zu diesem Begriff können Kursen zu Betriebssystemen entnommen werden (siehe z.B. Roitzsch [472]). In diesem Kapitel werden wir den Begriff „Prozess" immer in dem hier definierten Sinn als ein Objekt innerhalb eines Betriebssystems (und nicht im Sinne von Prozessen in SDL, VHDL, Prozessnetzwerken oder dem Halbleiterfertigungsprozess) benutzen.

Bei Systemen mit virtuellem Speicher[1] unterscheiden wir zwischen verschiedenen Adressräumen. In solchen Systemen können wir differenzieren, ob verschiedene in Ausführung befindliche Codeobjekte jeweils in einem eigenen Adressraum ausgeführt werden oder ob sie sich einen Adressraum teilen; im ersten Fall sprechen wir von **Prozessen**, im zweiten Fall von ***Threads*** (deutsch: Fäden, auch leichtgewichtige Prozesse genannt).

Definition 4.2: Ein ***Thread*** ist ein Programm in Ausführung, welches sich von einem Prozess durch die Verwendung eines gemeinsamen Adressraums unterscheidet.

Im ersten Fall existiert ein gewisser Schutz der Objekte gegeneinander, da sich verschiedene Objekte nicht einfach den Speicher überschreiben können. Allerdings erfordert der Kontextwechsel hier auch Aufwand für den Wechsel des Adressbereichs. Im zweiten Fall existiert dieser Schutz nicht, dafür ist der Kontextwechsel schneller und die Kommunikation über gemeinsamen Speicher (engl. *shared memory*) einfach realisierbar. In Systemen mit nur einem Adressraum entfällt die Unterscheidung zwischen Prozessen und *Threads*. Betriebssysteme müssen Kommunikations- und Synchronisationsmechanismen für *Threads* und Prozesse zur Verfügung stellen.

---

[1] Siehe Anhang C.

Vertiefende Informationen über die hier angesprochenen Standardthemen im Systemsoftwarebereich finden sich in Betriebssystem-Lehrbüchern wie z.B. dem Buch von Tanenbaum [525][2].
Die folgenden Eigenschaften sind für eingebettete Betriebssysteme wesentlich:

- Die große Vielfalt eingebetteter Systeme resultiert in einer großen Vielfalt von Anforderungen an die Funktionalität eingebetteter Betriebssysteme. Aus Effizienzgründen ist es nicht möglich, ein System einzusetzen, das alle Eigenschaften zur Verfügung stellt. Die meisten Anwendungen benötigen ein kleines Betriebssystem. Daher sollten Betriebssysteme eine **flexible Maßschneiderung** auf die gegebene Anwendung zulassen. **Konfigurierbarkeit** ist daher eine der wichtigsten Eigenschaften eingebetteter Betriebssysteme. Konfigurierbarkeit lässt sich mit einer Reihe von Techniken realisieren, wie z.B.[3]:
  - **Objektorientierung** zur Ableitung geeigneter Unterklassen. Von einer allgemeinen *Scheduler*-Klasse könnten z.B. *Scheduler* mit bestimmten Eigenschaften abgeleitet werden. Allerdings erfordern objektorientierte Ansätze oft zusätzlichen Aufwand. So erzeugt zum Beispiel das dynamische Binden von Methoden zusätzlichen Aufwand zur Laufzeit. Vorschläge zur Reduktion dieses Aufwandes existieren[4]. Dennoch können der verbleibende Aufwand und die potentiell schlechte Vorhersage des Zeitverhaltens für performanzkritische Systeme möglicherweise nicht tragbar sein.
  - **Aspektorientierte Programmierung** [351]: dieser Ansatz erlaubt es, Aspekte von Software, die orthogonal zueinander sind, unabhängig voneinander zu beschreiben und automatisch in alle relevanten Teile des Programmcodes einzubauen. So könnte *Profiling*-Code in einem eigenen Modul beschrieben sein, das dann automatisch zu allen relevanten Teilen des Quellcodes hinzugefügt oder aus diesen entfernt werden kann. Die CiAO-Betriebssystemfamilie wurde auf diese Weise entworfen [352].
  - **Bedingte Übersetzung**: Hier kommt ein Makro-Präprozessor zum Einsatz, dessen Befehle `#if` und `#ifdef` verwendet werden.
  - Erweiterte **Auswertung zur Übersetzungszeit** (engl. *compile time evaluation*): ein Betriebssystem könnte konfiguriert werden, indem bestimmte Variablen vor der Übersetzung mit konstanten Werten belegt werden. Ein Compiler könnte dann das Wissen über diese Werte so weit wie möglich ausnutzen. Hier könnten auch erweiterte Compiler-Optimierungen nützlich sein. Wenn ein bestimmter Funktionsparameter beispielsweise stets einen konstanten Wert besitzt, könnte dieser Parameter aus der entsprechenden Parameterliste entfernt werden. Die sogenannte partielle Evaluation [277] stellt eine Umgebung für solche Compileroptimierungen zur Verfügung. Weiterführende Ansätze könnten zudem dynamische Daten durch statische Daten ersetzen [25]. Ein

[2] Studierende, die bisher keine Vorlesung über Betriebssysteme gehört haben, sollten eines dieser Standardwerke zu Rate ziehen, bevor sie hier weiterlesen.

[3] Diese Liste ist nach der Position der Technik im Entwicklungsprozess geordnet.

[4] https://github.com/lefticus/cppbestpractices/blob/master/08-Considering_Performance.md ist ein Beispiel dafür.

Überblick über Ansätze der Betriebssystemspezialisierung wurde von McNamee et al. veröffentlicht [388].

– ***Linker*-basierte Entfernung ungenutzter Funktionen**: Oft stehen zur *Link*-Zeit mehr Informationen darüber zur Verfügung, welche Funktionen verwendet und welche nicht verwendet werden. So kann ein *Linker* z.B. herausfinden, welche Funktionen einer Bibliothek tatsächlich benutzt werden. Entsprechend können nicht verwendete Bibliotheksfunktionen entfernt werden und weitere Spezialisierungen stattfinden [91].

Diese Techniken werden meist mit einer regelbasierten Auswahl der für das jeweilige Betriebssystem benötigten Dateien kombiniert. Die Maßschneiderung des Betriebssystems kann durch eine graphische Benutzerschnittstelle vereinfacht werden, welche die eigentlichen Konfigurationstechniken vor dem Entwickler verbirgt. Das VxWorks-Betriebssystem der Firma *Wind River* [590] wird beispielsweise über eine solche graphische Benutzerschnittstelle konfiguriert.
Eines der möglichen Probleme bei Systemen mit einer großen Anzahl abgeleiteter, maßgeschneiderter Betriebssysteme ist die Verifikation. Jedes einzelne abgeleitete Betriebssystem muss sorgfältig getestet werden. Takada erwähnt dies als ein mögliches Problem für eCos (ein *Open Source*-Echtzeitbetriebssystem der Firma Red Hat [382]), das über 100 bis 200 Konfigurationsmöglichkeiten verfügt [523]. Auch bei Linux ist das ein Problem [526]. Die Verwendung von Techniken der Software-Produktlinienentwicklung [456] kann Ansätze liefern, um dieses Problem zu lösen.

• Eingebettete Systeme nutzen eine große Vielfalt an Peripheriegeräten. Viele eingebettete Systeme besitzen weder Festplatte, Tastatur, Bildschirm oder Maus. Es gibt **kein Gerät, das von allen Varianten eines Betriebssystems unterstützt werden muss**, vielleicht mit Ausnahme des Systemzeitgebers. Anwendungen sind oft dazu entworfen, bestimmte Geräte zu nutzen. In solchen Fällen müssen Geräte nicht von verschiedenen Anwendungen gleichzeitig benutzt werden. Damit ist es auch nicht erforderlich, dass das Betriebssystem diese Geräte verwaltet. Durch die große Anzahl an Geräten wäre es auch schwierig, alle benötigten Gerätetreiber zusammen mit dem Betriebssystem zur Verfügung zu stellen. Daher ist es sinnvoll, Betriebssystem und Geräte durch die Verwendung spezieller Prozesse zu entkoppeln, anstatt die Gerätetreiber in das Betriebssystem zu integrieren. Durch die begrenzte Geschwindigkeit vieler eingebetteter Systeme ist es auch nicht notwendig, die Treiber aus Performanzgründen in das Betriebssystem zu integrieren. Dies kann zu einer geänderten Anordnung der einzelnen Softwareschichten eines Systems führen. Bei PCs wird davon ausgegangen, dass bestimmte Treiber, wie solche für Festplatten, Netzwerkkarten, oder Audioausgabe, stets verfügbar sind. Diese sind auf einer weit unten liegenden Schicht implementiert. Anwendungssoftware und *Middleware* sind oberhalb der Anwendungs-Programmierschnittstelle implementiert, wie es für alle Anwendungen üblich ist. Bei einem eingebetteten Betriebssystem hingegen werden Gerätetreiber häufig oberhalb des Kernels implementiert. Anwendungen und *Middleware* wiederum

können dann oberhalb der zugehörigen Treiber implementiert werden, d.h. nicht basierend auf einer standardisierten API (siehe Abb. 4.2).

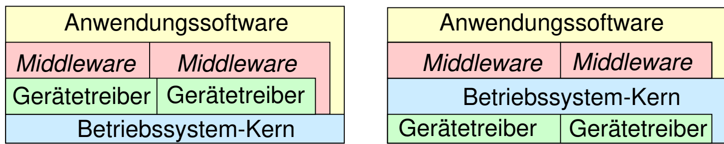


Abb. 4.2 Gerätetreiber: (**links:**) oberhalb und (**rechts:**) unterhalb des Betriebssystem-Kerns

- **Schutzmechanismen sind nicht immer erforderlich**, da eingebettete Systeme oft für einen bestimmten Zweck entworfen werden (es wird nicht angenommen, dass sie sogenanntes „Multiprogramming" unterstützen). Daher kommt es selten vor, dass ungetestete Programme geladen werden. Nach dem Testen von Software könnte davon ausgegangen werden, dass diese zuverlässig ist. Dies trifft genauso auf die Eingabe und Ausgabe zu. Im Gegensatz zu *Desktop*-Anwendungen müssen Ein/Ausgabeoperationen nicht als privilegierte Befehle implementiert werden, Prozesse dürfen eigenständig Eingaben und Ausgaben vornehmen. Dies passt gut zum vorher angesprochenen Punkt und senkt den Aufwand für E/A-Vorgänge.

  Beispiel 4.1: Sei `switch` die (in den Speicheradressbereich abgebildete) E/A-Adresse eines Schalters, der von einem Programm abgefragt werden soll. Dies kann einfach mit dem Befehl

  ```
  load register,switch
  ```

  erfolgen. Es ist nicht notwendig, dafür einen Betriebssystemdienst aufzurufen, der einen erheblichen Aufwand zum Sichern und Wiederherstellen des Prozesskontextes (Register usw.) erfordern würde. ∇

  Es ist aber ein Trend hin zu dynamischeren eingebetteten Systemen feststellbar. Zudem können Sicherheitsanforderungen Schutzvorrichtungen notwendig machen. Zu diesem Zweck wurden spezielle Speicherschutzeinheiten (engl. *Memory Protection Units* (MPUs)) entworfen, ein Beispiel dafür findet sich in [164]. Für Systeme mit einer Mischung von kritischen und unkritischen Anwendungen (engl. *mixed-criticality systems*) kann ein konfigurierbarer Speicherschutz [353] ein Ziel sein.
- **Unterbrechungen** (engl. *interrupts*) **können mit beliebigen Prozessen verbunden werden**. Über Betriebssystemdienste können wir anfordern, dass bestimmte Prozesse gestartet oder beendet werden, wenn eine bestimmte Unterbrechung auftritt. Wir könnten sogar die Startadresse eines Prozesses direkt in der *Interrupt*-Vektortabelle hinterlegen. Diese Methode ist allerdings sehr gefährlich, da das Betriebssystem dann nicht darüber informiert wäre, dass dieser Prozess tatsächlich läuft. Darunter könnte auch die Kompatibilität leiden: wenn ein bestimmter

Prozess direkt mit einer Unterbrechung verbunden ist, könnte es schwierig sein, einen weiteren Prozess hinzuzufügen, der ebenfalls von einem bestimmten Ereignis gestartet werden soll. Wenn anwendungsspezifische Gerätetreiber verwendet werden, könnten diese ebenfalls Verbindungen zwischen Unterbrechungen und Prozessen herstellen. Techniken zur Herstellung sicherer Verbindungen wurden von Hofer et al. untersucht [219].
- Viele eingebettete Betriebssysteme sind Echtzeitsysteme. Daher muss das verwendete Betriebssystem ein **echtzeitfähiges Betriebssystem** (engl. *Real-Time Operating System* (RTOS)) sein.

Weiterführende Informationen über eingebettete Betriebssysteme finden sich in einem von Bertolotti geschriebenen Buchkapitel [51]. Dieses Kapitel beinhaltet Informationen über die Architektur eingebetteter Betriebssysteme, den POSIX-Standard, *Open Source*-Echtzeitbetriebssysteme und Virtualisierung.

4.1.2 Echtzeitbetriebssysteme

Definition 4.3: Nach Takada [523] ist *„Ein Echtzeitbetriebssystem ... ein Betriebssystem, das die Erstellung von Echtzeitsystemen unterstützt"*.

Es gibt vier Eigenschaften, welche die Echtzeitfähigkeit eines Betriebssystems ausmachen[5]:

- **Das Zeitverhalten des Betriebssystems muss vorhersagbar sein.** Für jeden Dienst des Betriebssystems muss eine obere Schranke für die entsprechende Ausführungszeit garantiert sein. In der Praxis unterscheidet man zwischen verschiedenen Stufen der Vorhersagbarkeit. So kann es Mengen von Systemaufrufen geben, für die eine obere Schranke bekannt ist und deren Ausführungszeit nicht signifikant schwankt. Ein solcher Aufruf könnte „liefere die aktuelle Tageszeit" sein. Bei anderen Aufrufen könnte es dagegen starke Schwankungen geben. Aufrufe wie „reserviere 4 MB freien Speicher" könnten in diese zweite Klasse fallen. Insbesondere muss das *Scheduling*-Verfahren jedes RTOS deterministisch sein.
  Unter bestimmten Umständen kann es notwendig sein, Unterbrechungen zu sperren, um die gegenseitige Beeinflussung zwischen bestimmten Teilen des Betriebssystems zu vermeiden. Unterbrechungen können auch gesperrt werden, um Wechselwirkungen zwischen Prozessen zu vermeiden; dies ist jedoch meist von geringerer Bedeutung. Die Zeiträume, in denen Unterbrechungen gesperrt sind, sollten sehr kurz sein, um nicht vorhersagbare Verzögerungen bei der Abarbeitung kritischer Ereignisse zu vermeiden.
  Wenn ein RTOS ein Dateisystem auf einer Festplatte implementiert, kann es notwendig sein, grundsätzlich unfragmentierte Dateien (Dateien, deren Blöcke in aufeinanderfolgenden Festplattenbereichen gespeichert sind) zu implementieren, um unvorhersagbare Bewegungen der Festplattenköpfe zu vermeiden.

[5] Dieser Abschnitt beinhaltet Informationen aus dem Tutorial von Hiroaki Takada [523].

- **Das Betriebssystem muss das *Scheduling* der Prozesse durchführen.** *Scheduling* kann definiert werden als die Zuordnung von Mengen von Prozessen auf Intervalle der Ausführungszeit, mit dem Spezialfall der Abbildung auf bestimmte Startzeiten. Möglicherweise muss das Betriebssystem auch *Deadlines* bestimmter Prozesse kennen. Es gibt aber auch Fälle, in denen das *Scheduling* komplett zur Entwurfszeit stattfindet, dann muss das Betriebssystem nur Dienste bereitstellen, um Prozesse zu bestimmten Zeiten und auf bestimmten Prioritätsebenen zu starten. *Scheduling*-Algorithmen werden detailliert in Kapitel 6 behandelt.
- **Einige Dienste erfordern, dass das Betriebssystem Zeit verwaltet.** Eine solche Verwaltung ist unerlässlich, wenn die interne Abarbeitung mit einer absoluten Zeit in der physikalischen Umgebung in Verbindung steht. Physikalische Zeit wird durch reelle Zahlen beschrieben, Computer dagegen verwenden meist diskrete Zeitangaben. Die jeweiligen genauen Anforderungen können dabei variieren:

  1. Einige Systeme erfordern die Synchronisation mit globalen Zeitstandards. Diese führen eine **globale Uhrensynchronisation** durch. Dafür existieren zwei unterschiedliche Standards:
     - Die *Universal Time Coordinated* (UTC): Die UTC wird durch astronomische Standards definiert. Dieser Standard muss von Zeit zu Zeit aufgrund von Abweichungen der Erdbewegung angepasst werden. Beim Übergang von einem Jahr auf das folgende werden dabei einige Sekunden hinzugefügt. Diese Anpassungen können zu Problemen führen, da fehlerhaft implementierte Software davon ausgehen könnte, dass das folgende Jahr in derselben Nacht zweimal beginnt.
     - Die Internationale Atomzeit (französisch *temps atomic internationale* (TAI)). Dieser Standard unterliegt nicht den oben beschriebenen Beschränkungen.

     Die genaue Zeitinformation wird über eine Verbindung mit der Umgebung ermittelt. Externe Synchronisierung basiert dabei meist auf Funkübertragungsstandards, wie z.B. dem Globalen Positionierungs-System (GPS) [414], Mobilfunknetzwerken oder speziellen Langwellensendern zur Steuerung von Funkuhren [579]. Die Sender verbreiten eine Zeit auf der Basis von Atomuhren, in Mitteleuropa mittels des DCF77-Systems.
  2. Wenn eingebettete Systeme in einem Netzwerk eingesetzt werden, reicht es oft aus, Zeitinformation innerhalb des Netzwerks zu synchronisieren. Hierfür kann lokale Uhrensynchronisation zum Einsatz kommen. Damit können miteinander verbundene eingebettete Systeme versuchen, eine einheitliche Zeitbasis zu bestimmen.
  3. Es kann zudem Fälle geben, in denen es lediglich notwendig ist, präzise lokale Verzögerungen zu realisieren.

  Einige Anwendungen benötigen präzise Zeitdienste mit einer hohen Auflösung. Dies ist beispielsweise erforderlich, um zwischen ursprünglichen Fehlern und Folgefehlern in einem System unterschieden zu können. Diese können beispielsweise dazu genutzt werden, um Kraftwerke zu identifizieren, die für einen Stromausfall verantwortlich sind (siehe [428]). Die Genauigkeit von Zeitdiensten hängt davon

ab, wie diese auf einer bestimmten Ausführungsplattform unterstützt werden. Wenn die Implementierung durch Prozesse auf Anwendungsebene erfolgt, sind sie sehr unpräzise (mit einer Genauigkeit im Millisekunden-Bereich). Zeitdienste sind sehr präzise (mit Genauigkeiten im Mikrosekunden-Bereich), wenn sie von der Kommunikationshardware unterstützt werden. Weitere Informationen über Zeitdienste und Uhrensynchronisation finden sich im Buch von Kopetz [304].
- **Das Betriebssystem muss schnell sein.** Ein Betriebssystem, das alle bisher angeführten Anforderungen erfüllt, wäre nutzlos, wenn es selbst langsam wäre. Daher muss ein Betriebssystem offensichtlich schnell sein.

Jedes RTOS beinhaltet einen sogenannten **Echtzeitkern**. Dieser verwaltet die Ressourcen, die in jedem Echtzeitsystem vorhanden sind, wie z.B. den Prozessor, den Speicher und die Systemzeitgeber. Wichtige Funktionen des Echtzeitkerns sind die Verwaltung von Prozessen, Synchronisation und Kommunikation zwischen Prozessen, Zeitverwaltung und Speicherverwaltung.

Einige RTOS wurden für allgemeine eingebettete Anwendungen entworfen, während andere sich auf bestimmte Anwendungsgebiete spezialisiert haben. So sind beispielsweise OSEK/VDX® -kompatible Betriebssysteme speziell für den Einsatz in Steuergeräten im Automobilbereich gedacht. Betriebssysteme für einen bestimmten Anwendungsbereich stellen bestimmte Dienste für diesen speziellen Bereich zur Verfügung; sie können dadurch kompakter sein als Betriebssysteme, die für mehrere Anwendungsbereiche geeignet sind.

In ähnlicher Weise gibt es RTOS, die eine Standard-Programmierschnittstelle anbieten, während andere Systeme eine eigene, proprietäre API besitzen. So implementieren einige RTOS die standardisierte POSIX RT-Erweiterung für UNIX [202], andere implementieren den OSEK ISO 17356-3:2005-Standard oder die in Japan entwickelte ITRON-Spezifikation[6]. Viele Echtzeitbetriebssysteme verwenden ihre eigene API. Das erwähnte ITRON ist ein industrieerprobtes RTOS, das die Konfiguration zur *Link*-Zeit ermöglicht.

Die verfügbaren RTOS lassen sich weiterhin nach den folgenden Kriterien unterscheiden [194]:

- **Schnelle, proprietäre Kerne:** Nach Gupta sind „diese Kerne nicht für komplexe Systeme geeignet, da sie entworfen wurden, um schnell zu sein, aber nicht, um in jeder Hinsicht vorhersagbar zu sein”. Beispiele für solche Systeme sind QNX, PDOS, VCOS, VTRX32 und VxWorks.
- **Um Echtzeitfunktionen erweiterte Standard-Betriebssysteme:** Hybride Systeme wurden entwickelt, um die Vorteile komfortabler Standard-Betriebssysteme nutzen zu können. In solchen Systemen gibt es einen besonderen Echtzeitkern, der alle Echtzeitprozesse verwaltet. Das Standard-Betriebssystem wird dann als ein solcher Prozess ausgeführt (siehe Abb. 4.3). Dieser Ansatz bietet einige Vorteile: das System verfügt über eine standardisierte Betriebssystem-Programmierschnittstelle (API) und kann eine graphische Benutzerschnittstelle (GUI) sowie Dateisysteme usw. realisieren. Zudem sind Erweiterungen des

[6] Siehe http://www.ertl.jp/ITRON/ .

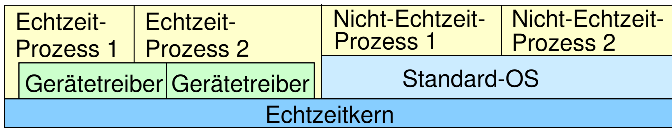


Abb. 4.3 Hybride Betriebssysteme

Standard-Betriebssystems schnell auch in eingebetteten Systemen verfügbar. Probleme, die das Standard-Betriebssystem und dessen nicht-Echtzeit-Prozesse betreffen, beeinflussen die Echtzeitprozesse nicht. Dies geht so weit, dass das Standard-Betriebssystem sogar abstürzen kann und die Echtzeitprozesse davon unbeeinflusst weiterlaufen. Ein Problem mit diesem Ansatz ist ebenfalls in Abb. 4.3 zu sehen: Gerätetreiber könnten Probleme verursachen, da das Standard-Betriebssystem über eigene Treiber verfügt. Um Wechselwirkungen zwischen Treibern für Echtzeitprozesse und denen für andere Prozesse zu vermeiden, kann es notwendig sein, die Menge der Geräte in solche, die durch Echtzeitprozesse behandelt werden und solche, die durch das Standard-Betriebssystem behandelt werden, zu unterteilen. Echtzeitprozesse können zudem keine Dienste des Standard-Betriebssystems nutzen. Damit sind alle nützlichen Eigenschaften wie ein Dateisystemzugriff und graphische Benutzerschnittstellen für diese Prozesse normalerweise nicht verfügbar. Es existieren aber erste Ansätze, die Unterschiede zwischen den beiden Arten von Prozessen zu überbrücken, ohne die Echtzeitfähigkeiten zu verlieren. RT-Linux ist ein Beispiel für ein solches hybrides Betriebssystem.

Nach Gupta [194] ist die Verwendung eines Standard-Betriebssystems „nicht der richtige Ansatz, da zu viele grundlegende und unpassende Voraussetzungen existieren wie die Optimierung für den durchschnittlichen Fall (anstelle des *worst case*), ... das Ignorieren der meisten oder der gesamten semantischen Informationen und unabhängiges *Scheduling* der CPU sowie Allokation von Ressourcen." Abhängigkeiten zwischen Prozessen kommen bei den meisten Anwendungen auf Standard-Betriebssystemen tatsächlich sehr selten vor und werden von solchen Systemen daher häufig ignoriert. Bei eingebetteten Systemen ist die Lage anders, da Abhängigkeiten zwischen Prozessen die Regel sind und daher berücksichtigt werden müssen. Dies ist jedoch nicht immer der Fall, wenn Erweiterungen von Standard-Betriebssystemen zum Einsatz kommen. Zudem betrachten Standard-Betriebssysteme Ressourcenzuteilung und *Scheduling* selten in Kombination. Integrierte Ressourcenzuteilung und *Scheduling*-Algorithmen sind aber notwendig, um das Einhalten von Zeitbedingungen garantieren zu können.

- Einige **Forschungssysteme** versuchen, die oben beschriebenen Einschränkungen zu vermeiden. Dazu zählen Melody [568] sowie (nach Gupta [194]) MARS, Spring, MARUTI, Arts, Hartos und DARK.

Takada [523] nennt leichtgewichtigen Speicherschutz, zeitlichen Schutz von Rechenressourcen (um Prozesse davon abzuhalten, länger zu rechnen als ursprünglich

geplant), RTOS für Multiprozessor-Systeme auf einem *Chip* (speziell für heterogene Multiprozessoren und mehrfädige (engl. *multi-threaded*) Prozessoren) sowie die Unterstützung für Medienströme und Dienstgüte als Forschungsfragen.

Das erwartete Wachstum des Marktes für das Internet der Dinge führt dazu, dass die Hersteller von Standard-Betriebssystemen versuchen, angepasste Versionen ihrer Produkte zu vermarkten und damit spezialisierten Anbietern wie *Wind River Systems* [591] Marktanteile abzuringen. Aufgrund zunehmender Netzanbindung von eingebetteten Systemen werden Linux und davon abgeleitete Systeme populär. Vor- und Nachteile des Einsatzes von Linux in solchen Systemen werden im Abschnitt 4.4 beschrieben.

4.1.3 Virtuelle Maschinen

In bestimmten Umgebungen kann es von Vorteil sein, mehrere Prozessoren auf einem einzelnen echten Prozessor zu emulieren. Dies wird durch **virtuelle Maschinen**, die direkt auf der Hardware ausgeführt werden, ermöglicht. Auf Basis einer solchen virtuellen Maschine können dann verschiedene Betriebssysteme realisiert werden. Damit können mehrere Betriebssysteme auf einem einzigen Prozessor implementiert werden. Bei eingebetteten Systemen muss dieser Ansatz mit Vorsicht betrachtet werden, da das Zeitverhalten hier problematischer sein kann und die zeitliche Vorhersagbarkeit verloren gehen könnte. Dennoch kann es Fälle geben, in denen dieser Ansatz nützlich ist. Beispielsweise könnte es die Anforderung geben, mehrere existierende Anwendungen, die verschiedene Betriebssysteme nutzen, auf einem einzelnen Prozessor zu integrieren. Eine vollständige Betrachtung virtueller Maschinen würde den Rahmen dieses Buches sprengen. Der interessierte Leser sei daher auf die Bücher von Smith et al. [502] und Craig [114] verwiesen. PikeOS ist ein Beispiel für ein Virtualisierungskonzept, das speziell für eingebettete Systeme entwickelt wurde [520]. PikeOS ermöglicht es, die Ressourcen eines Systems (z.B. Speicher, E/A-Geräte, CPU-Zeit) in einzelne Teilmengen aufzuteilen. PikeOS stellt einen kleinen Mikrokern zur Verfügung. Auf Basis dieses Kerns können verschiedene Betriebssysteme, Programmierschnittstellen (APIs) und Laufzeitumgebungen (engl. *Run-Time Environments* (RTEs)) implementiert werden (siehe Abb. 4.4).

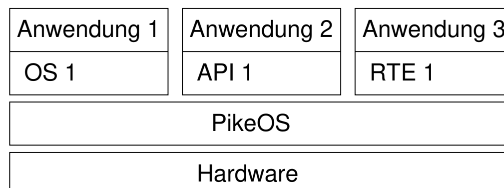


Abb. 4.4 PikeOS-Virtualisierung (©SYSGO)

4.2 Protokolle für Ressourcen-Zugriffe

In diesem Unterabschnitt benutzen wir den Begriff des **Jobs**.

Definition 4.4: Ein **Job** ist eine konkrete Ausführung einer (möglicherweise wiederholt auszuführenden) Task.

Ein Job ist eine abstraktere Sicht auf auszuführende Berechnungen als Prozesse und *Threads* in Betriebssystemen, auf die Jobs letztlich im Rahmen des Entwurfs abgebildet werden müssen. Eine Präzisierung dieser Definition wird in Definition 6.1 gegeben.

4.2.1 Prioritätsumkehr

In einigen Fällen muss sichergestellt werden, dass Jobs exklusiven Zugriff auf Ressourcen wie gemeinsame globale Variablen oder Geräte erhalten, um nichtdeterministisches oder anderweitig unerwünschtes Programmverhalten zu vermeiden. Dieser exklusive Zugriff ist besonders wichtig bei eingebetteten Systemen, z.B. zur Implementierung von Kommunikation über gemeinsamen Speicher oder für den exklusiven Zugriff auf ein bestimmtes Hardwaregerät. Die Programmabschnitte, die einen solchen exklusiven Zugriff während ihrer Ausführung benötigen, werden **kritische Abschnitte** genannt. Kritische Abschnitte sollten kurz ein. Betriebssysteme stellen meist Grundfunktionen (engl. *primitives*) zur Verfügung, um exklusiven Zugriff auf Ressourcen anzufordern und wieder freizugeben, diese werden auch *Mutexe* (für „*mutual exclusion*" = gegenseitiger Ausschluss) genannt. Jobs, die keinen exklusiven Zugriff erhalten, müssen warten, bis die entsprechende Ressource freigegeben wird. Demzufolge muss die Freigabeoperation prüfen, ob es wartende Jobs gibt und den Job mit höchster Priorität fortsetzen.

In diesem Buch nennen wir die Anforderungsoperation P(S) und die Freigabeoperation V(S), wobei S für die jeweilige Ressource steht. P(S) und V(S) sind sogenannte **Semaphor**-Operationen. Semaphore ermöglichen es, dass bis zu n (wobei n ein Parameter ist) Jobs eine bestimmte Ressource, die von S geschützt wird, gleichzeitig benutzen können. S kennzeichnet dabei eine Datenstruktur, die einen Zähler enthält, der angibt, wie viele Ressourcen noch verfügbar sind. P(S) prüft diesen Zähler und blockiert den Aufrufer, wenn alle Ressourcen bereits belegt sind. Anderenfalls wird der Zähler verändert und der Aufrufer darf fortfahren. V(S) inkrementiert die Anzahl verfügbarer Ressourcen und stellt sicher, dass ein blockierter Aufrufer (falls einer existiert) fortfahren kann. Wichtig ist, dass die Operationen P(S) und V(S) unteilbar sind, d.h. nie durch andere Operationen unterbrochen werden können. Die Bezeichnungen P(S) und V(S) entstammen der niederländischen Sprache. Wir verwenden diese Operationen nur in Form von binären Semaphoren mit $n = 1$, d.h., wir erlauben nur einem einzigen Aufrufer, die Ressource zu nutzen.

In eingebetteten Systemen sind Abhängigkeiten zwischen Jobs die Regel und keine Ausnahme. Auch ist die effektive Jobpriorität von Echtzeitanwendungen wichtiger

als bei nicht-Echtzeit-Anwendungen. Gegenseitiger Ausschluss kann zur Prioritätsumkehr führen, welche die effektive Priorität von Jobs ändert. Auch bei nicht eingebetteten Systemen kommt Prioritätsumkehr vor. Aus den vorher genannten Gründen ist das Problem der Prioritätsumkehr aber bei eingebetteten Systemen schwerwiegender.

Ein erstes Beispiel für die Folgen, die aus einer Kombination von gegenseitigem Ausschluss und Nichtentzug von Prozessorressourcen entstehen, ist in Abb. 4.5 dargestellt. In diesem Unterabschnitt nutzen wir Material aus dem Buch von Buttazzo [81].

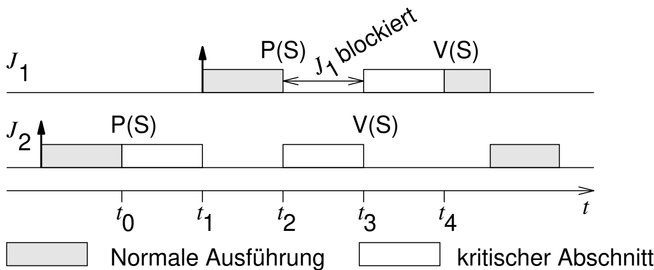


Abb. 4.5 Blockieren eines Jobs durch einen Job niedrigerer Priorität

Fettgedruckte, nach oben zeigende Pfeile kennzeichnen die Zeitpunkte, zu denen Jobs ausführbar oder „bereit" werden. Zum Zeitpunkt t_0 betritt Job J_2 einen kritischen Abschnitt, nachdem er exklusiven Zugriff auf eine Ressource mittels einer P-Operation angefordert und erhalten hat. Zum Zeitpunkt t_1 wird Job J_1 ausführbereit und verdrängt J_2. Zum Zeitpunkt t_2 erhält J_1 keinen exklusiven Zugriff auf die gerade von J_2 belegte Ressource und wird blockiert. Job J_2 kann fortfahren und gibt die Ressource einige Zeit später wieder frei. Die Freigabeoperation prüft, ob Jobs höherer Priorität warten und verdrängt J_2. Während der Zeit, in der J_1 blockiert war, hat also ein Job niederer Priorität einen Job mit höherer Priorität effektiv blockiert. Die Notwendigkeit, exklusiven Zugriff auf einige Ressourcen zur Verfügung zu stellen, ist die Hauptursache für diesen Effekt. Im Fall von Abb. 4.5 kann die Dauer der Blockierung die Länge des kritischen Abschnitts von J_2 glücklicherweise nicht überschreiten. Diese Situation ist problematisch, aber nur schwierig zu vermeiden.

Im allgemeinen Fall kann die Situation noch deutlich schlimmer sein. Dies ist z.B. in Abb. 4.6 dargestellt. Gegeben seien die Jobs J_1, J_2 und J_3. J_1 besitzt die höchste Priorität, J_2 eine mittlere und J_3 die niedrigste Priorität. Wir nehmen an, dass J_1 und J_3 exklusiven Zugriff auf eine Ressource mittels der Operation P(S) anfordern. Sei J_3 nun in seinem kritischen Abschnitt, wenn er von J_2 verdrängt wird. Wenn J_1 nun J_2 verdrängt und versucht, auf dieselbe Ressource zuzugreifen, auf die J_3 gerade exklusiven Zugriff hat, blockiert dieser und J_2 kann fortfahren. Solange J_2 fortfährt, kann J_3 die Ressource nicht freigeben. Daher blockiert J_2 effektiv J_1, obwohl die Priorität von J_1 höher als die von J_2 ist. In diesem Beispiel wird J_1 bis

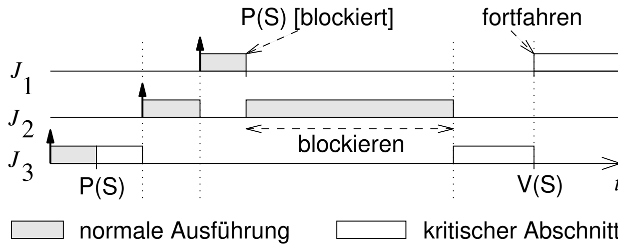


Abb. 4.6 Prioritätsumkehr mit möglicher großer Verzögerung

zum Ende der Laufzeit von J_2 blockiert. J_1 wird von einem Job niederer Priorität blockiert, der sich nicht in seinem kritischen Abschnitt befindet. Dieser Effekt wird **Prioritätsumkehr**[7] genannt. Tatsächlich findet die Prioritätsumkehr statt, obwohl J_2 nicht direkt mit J_1 und J_3 in Verbindung steht. Die Dauer der Prioritätsumkehr ist nicht durch die Länge irgendeines kritischen Abschnitts beschränkt. Dieses und weitere Beispiele können mit Hilfe der levi-Simulationssoftware simuliert werden [497].

Einer der bekanntesten Fälle von Prioritätsumkehr betraf den *Mars Pathfinder*, in dem ein exklusiver Zugriff auf einen gemeinsam benutzten Speicherbereich zur Prioritätsumkehr auf dem Mars führte [276].

4.2.2 *Prioritätsvererbung*

Ein Ansatz zum Umgang mit Prioritätsumkehr ist die Verwendung des Prioritätsvererbungs-Protokolls (engl. *Priority Inheritance Protocol* (PIP)). Dieses Protokoll ist in vielen Echtzeitbetriebssystemen verfügbar. Es funktioniert wie folgt:

- Jobs werden entsprechend ihrer aktiven Prioritäten zugeteilt. Jobs mit derselben Priorität werden auf *first-come, first-served*-Basis zugeteilt.
- Wenn ein Job J_1 die Operation P(S) ausführt und ein anderer Job J_2 bereits exklusiven Zugriff darauf hat, wird J_1 blockiert. Wenn die Priorität von J_2 niedriger als die von J_1 ist, erbt J_2 die Priorität von J_1. Damit kann J_2 die Ausführung fortsetzen. Allgemein betrachtet, erbt jeder Job die höchste Priorität aller Jobs, die durch ihn blockiert werden.
- Wenn ein Job J_2 die Operation V(S) ausführt, wird seine Priorität auf die höchste Priorität aller Jobs, die von ihm blockiert werden, reduziert. Wenn kein weiterer Job von J_2 blockiert wird, wird dessen Priorität auf den ursprünglichen Wert zurückgesetzt. Zudem wird der Job mit der höchsten Priorität, der bisher beim Zugriff auf S blockiert wurde, fortgesetzt.

[7] Einige Autoren sehen bereits den in Abb. 4.5 gezeigten Fall als Prioritätsumkehr an. Dies war auch in früheren Ausgaben dieses Buchs der Fall.

- Prioritätsvererbung ist transitiv: wenn J_x Job J_y blockiert und J_y Job J_z blockiert, dann erbt J_x die Priorität von J_z.

Wenn Jobs einer hohen Priorität durch Jobs mit einer niedrigen Priorität blockiert werden, dann geben sie ihre Priorität an die Jobs der niedrigen Priorität weiter, sodass diese ihre Semaphore so schnell wie möglich freigeben können. Im Beispiel von Abb. 4.6 würde J_3 die Priorität von J_1 erben, sobald J_1 P(S) ausführt. Dies würde das genannte Problem vermeiden, da J_2 nun J_3 nicht verdrängen würde (siehe Abb. 4.7).

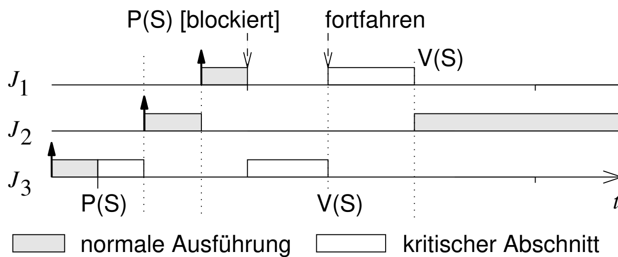


Abb. 4.7 Prioritätsvererbung für das Beispiel von Abb. 4.6

Abb. 4.8 zeigt ein Beispiel für verschachtelte kritische Abschnitte [81]. Beachten Sie, dass die Priorität von Job J_3 zum Zeitpunkt t_0 nicht auf ihren ursprünglichen Wert zurückgesetzt wird. Stattdessen wird seine Priorität auf den niedrigsten Wert der von ihm blockierten Jobs gesetzt, in diesem Fall die Priorität p_1 von J_1.

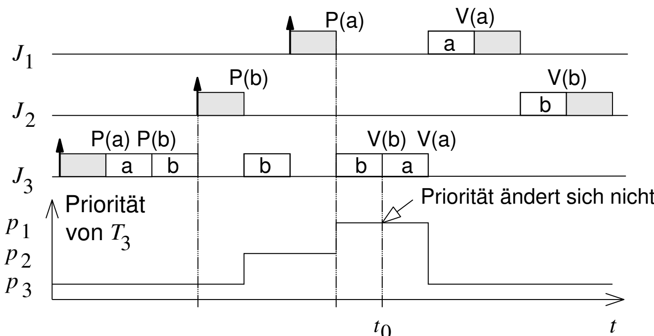


Abb. 4.8 Verschachtelte kritische Abschnitte

Die Transitivität der Prioritätsvererbung ist in Abb. 4.9 dargestellt [81]. Zum Zeitpunkt t_0 wird J_1 von J_2 blockiert, der wiederum von J_3 blockiert wird. Daher erbt J_3 die Priorität p_1 von J_1.

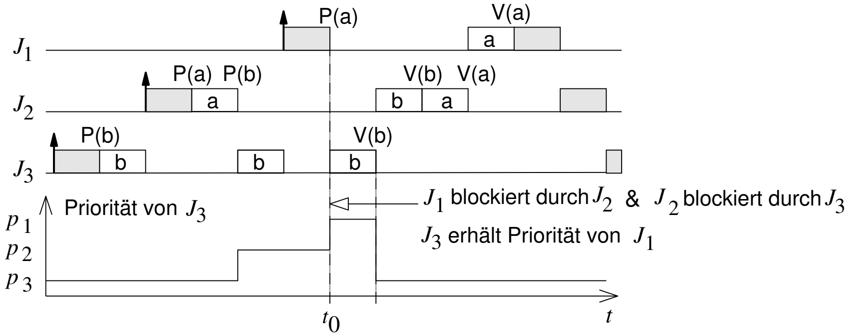


Abb. 4.9 Transitivität der Prioritätsvererbung

Prioritätsvererbung wird auch von Ada verwendet: während eines *Rendez-Vous* wird die Priorität der betroffenen Codeobjekte auf deren Maximum gesetzt.

Prioritätsvererbung löste auch das Problem des *Mars Pathfinder*: das verwendete VxWorks-Betriebssystem besitzt einen Booleschen Parameter für Aufrufe von *Mutex*-Grundfunktionen. Dieser Parameter erlaubt es, die Prioritätsvererbung zu nutzen. Im Auslieferungszustand der Software war Prioritätsvererbung deaktiviert. Das Problem auf dem Mars wurde beseitigt, indem dieser Parameter mittels der *Debugging*-Funktionalität von VxWorks modifiziert und die Prioritätsvererbung damit aktiviert wurde, als der *Pathfinder* sich bereits auf dem Mars befand [276]. Prioritätsvererbung kann mit Hilfe der levi-Simulationssoftware simuliert werden [497].

Prioritätsvererbung kann einige Problem lösen, aber bei Weitem nicht alle. So könnte es eine große Anzahl von Jobs mit hoher Priorität geben. Auch kann es *Deadlocks* geben, was anhand eines Beispiels gezeigt werden kann [81].

Beispiel 4.2: Angenommen, es gäbe zwei Jobs J_1 und J_2. Für Job J_1 nehmen wir eine Codesequenz der Form ...; P(a); P(b); V(b); V(a); ...; an. Für Job J_2 nehmen wir eine Codesequenz der Form ...; P(b); P(a); V(a); V(b); ...; an. Eine mögliche Ausführungsreihenfolge ist in Abb. 4.10 zu sehen. Wir nehmen an, dass die Priorität von J_1 höher ist als die von J_2. Daher verdrängt J_1 den Job J_2 zum Zeitpunkt t_1

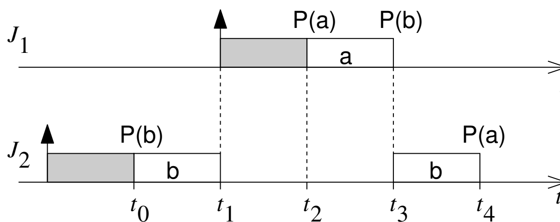


Abb. 4.10 Prioritätsvererbung mit *Deadlock*

und wird ausgeführt bis er P(b) ruft, während b durch J_2 gehalten wird. Daher wird J_2 wieder ausgeführt. Job J_2 wird blockiert, wenn er P(a) aufruft. Dieser *Deadlock* würde auch ohne Nutzung eines Protokolls zum Ressourcenzugriff existieren. ∇

4.2.3 Priority Ceiling-Protokoll

Deadlocks können mit dem *Priority Ceiling*-Protokoll (PCP) [485] vermieden werden, bei dem die Jobs zur Entwurfszeit bekannt sein müssen. **Bei PCP dürfen Jobs keinen kritischen Abschnitt betreten, wenn es bereits gesperrte Semaphoren gibt, die diesen Job irgendwann blockieren könnten.** Wenn ein Job einen kritischen Abschnitt betritt, kann er vor dem Verlassen desselben mithin nicht durch Jobs einer niedrigeren Priorität blockiert werden. Dies wird mit der Bildung eines Maximums von Prioritäten (engl. *priority ceiling*) erreicht. Jedem Semaphor S wird ein Maximum C(S) zugeordnet. Dies ist die statische Priorität des Jobs der höchsten Priorität, der S sperren kann. Das PCP-Protokoll geht wie folgt vor:

- Wir nehmen an, dass Job J läuft und Semaphor S sperren möchte. Dann kann J S nur sperren, wenn die Priorität von J den Wert C(S') des Semaphors S' übersteigt, wobei S' der Semaphor mit dem höchsten Wert von C(S) unter allen Semaphoren ist, die gegenwärtig durch von J verschiedene Jobs gesperrt werden. Wenn ein solcher Semaphor existiert, dann gilt J als durch S' und den S' haltenden Job blockiert. Wenn J durch S' blockiert wird, erhält der Job, der S' sperrt, die Priorität von J.
- Wenn ein Job J einen kritischen Abschnitt, der durch S geschützt ist, verlässt, entsperrt er S. Sofern mindestens ein Job vorhanden ist, der durch S blockiert ist, wird unter diesen der Job der höchsten Priorität aufgeweckt. Die Priorität von J wird auf die höchste Priorität unter allen Jobs gesetzt, die noch durch eine Semaphore blockiert sind, die J hält. Wenn J durch keinen anderen Job blockiert wird, wird seine Priorität wieder auf seine normale Priorität zurückgesetzt.

Beispiel 4.3: In der Abb. 4.11 werden Semaphore a, b und c benutzt[8]. Die höchste Priorität von a und b ist p_1, die höchste Priorität von c ist p_2. Zum Zeitpunkt t_2 möchte J_2 Semaphor c sperren, aber c ist schon gesperrt. Außerdem übersteigt die Priorität von J_2 nicht den Wert C(c). Trotzdem resultiert der Versuch, c zu sperren, in einem Erhöhen der Priorität von J_3 auf p_2.

Zum Zeitpunkt t_5 versucht J_1, Semaphor a zu sperren. a ist noch nicht gesperrt, aber J_3 hat b gesperrt und die gegenwärtige Priorität von J_1 übersteigt nicht den Wert C(b). Deswegen wird J_1 blockiert. Dies ist die wesentliche Eigenschaft von PCP: dieses Blockieren vermeidet ansonsten mögliche spätere *Deadlocks*. J_3 erhält die Priorität von J_1. Darin drückt sich aus, dass J_1 darauf wartet, dass Semaphor b durch J_3 freigegeben wird.

[8] Wir nutzen hier ein Beispiel von Bordoloi [59].

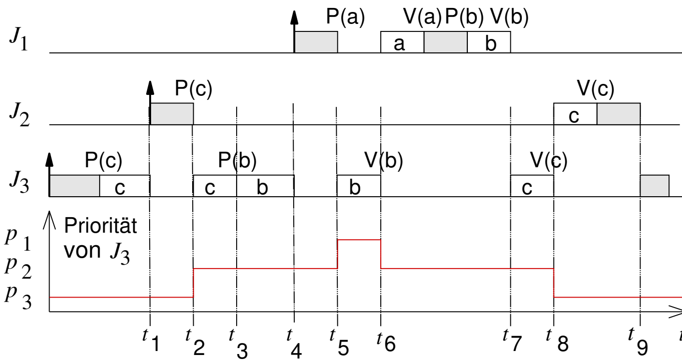


Abb. 4.11 Sperren mit PCP

Zum Zeitpunkt t_6 entsperrt J_3 Semaphor b. J_1 ist der Job der höchsten Priorität, der bislang durch b blockiert war und er wird jetzt wieder aufgeweckt. Die Priorität von J_3 fällt auf p_2. J_1 sperrt und entsperrt a und b und wird zu Ende ausgeführt. Zum Zeitpunkt t_7 ist J_2 noch durch c blockiert und unter allen Jobs mit p_2 ist J_3 der einzige, der weiter ausgeführt werden kann. Zum Zeitpunkt t_8 entsperrt J_3 Semaphor c und seine Priorität fällt auf p_3. J_2 ist nicht mehr länger blockiert und er verdrängt J_3 und sperrt c. J_3 wird erst wieder ausgeführt, wenn J_2 beendet wurde. ∇

Beispiel 4.4: Wir betrachten nunmehr ein Beispiel, das wir später für einen Vergleich mit einem erweiterten PCP-Protokoll verwenden werden. Abb. 4.12 zeigt dieses zweite Beispiel[9]. Die höchste Priorität unter allen Semaphoren ist die Priorität von

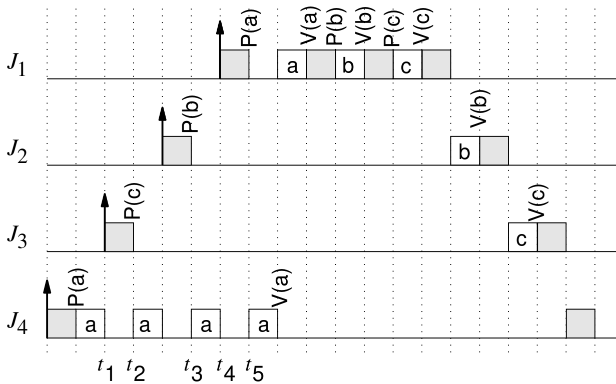


Abb. 4.12 Zweites PCP-Beispiel

[9] Dieses Beispiel wurde [59] entnommen.

J_1. Zum Zeitpunkt t_2 möchte J_3 Semaphor c sperren, aber die Priorität von J_3 ist kleiner als C(a) für die bereits gesperrte Semaphore a und J_4 erhält die Priorität von J_3. Zum Zeitpunkt t_3 gibt es einen Sperrwunsch für b, aber die Priorität von J_2 ist wieder kleiner als C(a) des bereits gesperrten Semaphors a und J_4 erhält die Priorität von J_2. Zum Zeitpunkt t_5 gibt es einen Sperrwunsch für a, aber die Priorität von J_1 übersteigt nicht C(a) und J_4 erhält die Priorität von J_1. Nachdem J_4 Semaphor a freigibt, ist kein Semaphor gesperrt und seine Priorität fällt auf seine normale Priorität. Zu dieser Zeit hat J_1 die größte Priorität und wird bis zu seinem Ende ausgeführt. Die verbleibenden Ausführungen sind durch die normalen Prioritäten bestimmt. ∇

Es kann gezeigt werden, dass PCP *Deadlocks* vermeidet (siehe [81], Theorem 7.3). Es gibt Varianten von PCP, die sich dadurch von PCP unterscheiden, dass die Prioritäten zu anderen Zeitpunkten geändert werden. Das *Distributed Priority Ceiling Protocol* (DPCP) [466] und das *Multiprocessor Priority Ceiling Protocol* (MPCP) [465] sind Erweiterungen von PCP auf Multiprozessoren.

4.2.4 Stack Resource Policy-Protokoll

Im Unterschied zu PCP unterstützt die *Stack Resource Policy* (SRP) dynamisches *Scheduling*, insbesondere kann SRP mit den dynamischen Prioritäten benutzt werden, die im EDF-*Scheduling* berechnet werden (siehe Unterabschnitt 6.2.1 auf Seite 332). Bei SRP müssen wir zwischen Jobs und Tasks unterscheiden. Tasks können sich wiederholende Berechnungen beschreiben (siehe auch Definition 2.2). Jede Berechnung bildet einen **Job** entsprechend unserer bisherigen Benutzung. Unter dem Begriff **Task** erfassen wir jetzt alle Eigenschaften, die auf eine Menge von gleichartigen Jobs zutreffen, d.h. Jobs, die periodisch denselben Code ausführen. Dementsprechend gehört zu jeder Task τ_i eine Menge von Jobs. Siehe dazu auch Definition 6.1 auf Seite 324. SRP betrachtet nicht jeden Job einer Task einzeln, sondern definiert Eigenschaften, die global auf Tasks Anwendung finden. Zusätzlich unterstützt SRP Ressourcen, die aus mehreren Einheiten bestehen, wie z.B. Speicherpuffer. Die folgenden Größen werden definiert:

- Der *preemption level* l_i einer Task τ_i gibt Informationen darüber, welche anderen Tasks durch Jobs von τ_i verdrängt werden können. Eine Task τ_i kann eine andere Task τ_j nur verdrängen, wenn $l_i > l_j$ ist. Wir verlangen: Wenn Task τ_i nach τ_j im System ankommt und τ_i hat eine höhere Priorität als τ_j, dann muss $l_i > l_j$ sein. Für EDF-*Scheduling* (siehe Seite 332) bedeutet dies, dass die *preemption levels* eine monoton fallende Folge der relativen *Deadlines* sind. Umso später die *Deadline*, umso leichter wird es sein, einen Job zu verdrängen. l_i ist ein statischer Wert.
- Die *resource ceiling* einer Ressource ist der größte *preemption level* der Tasks, die dadurch blockiert werden könnten, dass sie ihre Maximalforderung an Einheiten

dieser Ressource stellen. Die *resource ceiling* ist ein dynamischer Wert, der davon abhängt, wie viele Einheiten in den Ressourcen noch verfügbar sind.

- Die *system ceiling* ist die größte unter allen *resource ceilings* von gegenwärtig blockierten Ressourcen. Dieser Wert ist dynamisch und ändert sich mit dem Zugriff auf Ressourcen.

SRP blockiert Jobs nicht, wenn sie versuchen, eine Ressource zu blockieren, sondern wenn sie versuchen, andere zu verdrängen.

Beispiel 4.5: Abb. 4.13[10] zeigt den Unterschied zwischen PCP und SRP anhand des Beispiels aus Abb. 4.12. Im Fall von SRP gibt es zum Zeitpunkt t_1 kein Verdrängen,

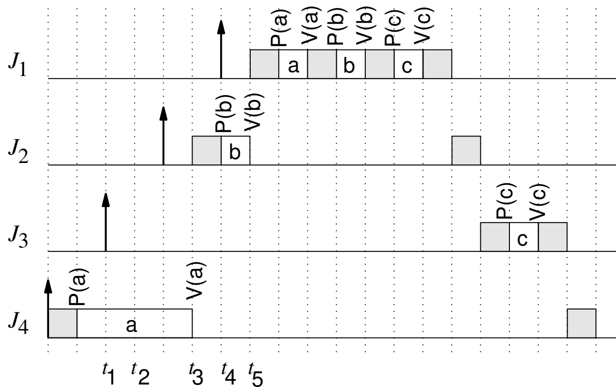


Abb. 4.13 Verhalten des SRP-Protokolls

denn der *preemption level* ist nicht größer als die *system ceiling*. Dasselbe passiert zum Zeitpunkt t_4. Insgesamt kommt es bei SRP zu deutlich weniger Verdrängungen von Jobs als bei PCP. Daher ist SRP ein sehr beliebtes Protokoll. ∇

SRP heißt ***stack** resource policy*, da Jobs nicht durch Jobs mit einem niedrigeren *preemption level* blockiert werden können und sie können nur wieder ausgeführt werden, wenn der vorherige Job zu Ende ausgeführt wurde. Daher können alle Jobs mit demselben l_i denselben *Stack*-Speicherplatz belegen. Bei sehr vielen Jobs mit demselben *preemption level* kann so viel Speicherplatz gespart werden. SRP ist auch frei von *Deadlocks* (siehe Baker [34]). Buttazzo [81] beschreibt weitere Details von SRP.

PIP-, PCP- und SRP-Protokolle wurden für Einzelprozessoren entworfen. Rajkumar et al. [466] haben eine erste Übersicht über Protokolle für den Zugriff zu Ressourcen bei Multiprozessoren veröffentlicht. Gegenwärtig hat sich hierfür noch kein Standard etabliert (siehe Baruah et al. [38], Kapitel 23).

[10] Auch diese Abbildung wurde [59] entnommen.

4.3 ERIKA

Verschiedene eingebettete Systeme (wie Automobilsteuergeräte und Haushaltsgeräte) bedingen, dass die gesamte Anwendung auf einem kleinen Mikrocontroller integriert ist[11]. Daher müssen die von der *Firmware* angebotenen Betriebssystemdienste auf solchen Systemen auf eine solche minimale Teilmenge beschränkt werden, welche die mehrfädige Ausführung von periodischen und aperiodischen Tasks unterstützt und dabei gemeinsame Ressourcennutzung unter Vermeidung von Prioritätsumkehr erlaubt.

Entsprechende Anforderungen wurden in den neunziger Jahren durch das OSEK/VDK-Konsortium veröffentlicht [578]. Diese definierten die minimale Menge an Diensten eines mehrfädigen Echtzeitbetriebssystems, die mit einer Codegröße von 1-10 kB auf einem 8-Bit Mikrocontroller implementierbar sein sollte. Im Jahr 2010 wurde die OSEK/VDX API durch das AUTOSAR-Konsortium [27] um Funktionalität zur Unterstützung von zeitlichem Schutz, *Scheduling*-Tabellen für zeitgetriggerte Systeme und Speicherschutz zum Schutz der Ausführung unterschiedlicher Anwendungen auf demselben Mikrocontroller erweitert. Dieser Abschnitt beschreibt in Kürze die Haupteigenschaften von und Anforderungen an solche Systeme, wobei der *Open Source*-Echtzeitkern ERIKA *Enterprise* als Referenzimplementierung dient [157].

Die erste Eigenschaft, die einen OSEK-Kern von anderen Betriebssystemen unterscheidet, ist, dass alle Kernobjekte *statisch* zur Übersetzungszeit definiert sind. Insbesondere erlauben die meisten dieser Systeme weder dynamische Speicherzuteilung noch die dynamische Erzeugung von Tasks[12]. Um den Benutzer bei der Konfiguration des Systems zu unterstützen, stellt der OSEK/VDX-Standard die Konfigurationssprache OIL zur Verfügung, in der die für die Anwendung zu instanziierenden Objekte definiert werden. Beim Übersetzen der Anwendung erzeugt der OIL-Compiler die Betriebssystem-Datenstrukturen und alloziert genau die benötigte Menge an Speicher. Dieser Ansatz ermöglicht es, nur genau die Daten im *Flash*-Speicher (der bei den meisten Mikrocontrollern preiswerter als RAM ist) zu allozieren, die von der Anwendung wirklich gebraucht werden.

Das zweite Unterscheidungsmerkmal eines OSEK/VDX-Systems im Vergleich zu Standard-Betriebssystemen ist die Unterstützung für **gemeinsam benutzte Stapel** (engl. *Stack Sharing*). *Stack Sharing* wird eingesetzt, da RAM-Speicher bei kleinen Mikrocontrollern sehr teuer ist. Ob *Stack Sharing* einsetzbar ist, hängt insbesondere davon ab, wie der Code der einzelnen Tasks geschrieben wurde.

In herkömmlichen Echtzeitsystemen ist die Implementierung einer periodischen Task nach dem folgenden Schema strukturiert:

```
task(x) {
  int Lokale_Variable;
  Initialisierung();
  for (;;) {
```

[11] Dieser Abschnitt wurde von G. Buttazzo und P. Gai (Pisa) beigesteuert.

[12] Im Kontext von ERIKA wird der Begriff Task ähnlich benutzt wie bisher der Begriff Prozess.

```
        Bearbeite_Instanz();
        Ende_der_Instanz();
      }
    }
```

Dieses Schema ist gekennzeichnet durch eine Endlosschleife, die eine Instanz der periodischen Task enthält, die mit einer blockierenden Operation `Ende_der_Instanz()` endet. Diese blockiert die Task bis zur nächsten Aktivierung. Wenn ein solches Programmierschema zum Einsatz kommt (bei OSEK/VDX *extended task* genannt), ist diese Task ständig, auch in Wartezeiten, auf dem Stapel präsent. In diesem Fall kann der Stapel nicht geteilt werden und es muss ein eigener Stapel-Bereich für jede Task reserviert werden.

Der OSEK/VDX-Standard sieht außerdem *basic tasks* vor. Diese sind periodische Tasks, die ähnlich zu Funktionen nach dem folgenden Schema implementiert werden:

```
int Lokale_Variable;
task(x) {
  Bearbeite_Instanz();
}
System_Initialisieren() {
  Initialisierung();
  ...
}
```

Im Vergleich zu *extended tasks* wird in *basic tasks* der persistente Zustand, der zwischen zwei Ausführungen erhalten bleiben muss, nicht auf dem Stapel, sondern in globalen Variablen gespeichert. Der Initialisierungsteil wird in die Systeminitialisierung verschoben, da Tasks nicht dynamisch erzeugt werden, sondern schon zum Start des Systems existieren. Schließlich wird keine Synchronisierungsoperation benötigt, um die Task bis zu ihrer folgenden Periode zu blockieren, da die Task jedes Mal aktiviert wird, wenn eine neue Instanz startet. Die Task selbst kann auch keine blockierende Operation aufrufen, daher kann sie entweder von höher priorisierten Tasks verdrängt werden oder bis zu ihrem Ende ablaufen. So verhält sich die Task wie eine Funktion, die einen Bereich auf dem Stapel (engl. *stack frame*) alloziert, abläuft und dann den Bereich wieder freigibt. Daher belegt die Task keinen Stapelspeicher zwischen zwei Ausführungen und der Stapel kann von allen Tasks des Systems zusammen benutzt werden. ERIKA *Enterprise* unterstützt *Stack Sharing* für alle *basic tasks* des Systems, damit für diesen Anwendungsfall die Menge an benötigtem RAM-Speicher reduziert werden kann.

Im Bereich der Taskverwaltung bieten OSEK/VDX-Kerne Unterstützung für *Scheduling* mit festen Prioritäten mit *Immediate Priority Ceiling Protocol* (deutsch unverzügliches Prioritäts-Obergrenzen-Protokoll), um Prioritätsumkehr-Probleme zu vermeiden. Die Verwendung des *Immediate Priority Ceiling*-Protokolls wird unterstützt, indem der Ressourcenverbrauch jeder Task in der OIL-Konfigurationsdatei angegeben wird. Basierend auf dieser Angabe berechnet der OIL-Compiler den maximalen Ressourcenverbrauch für jede Task.

OSEK/VDX-Systeme unterstützen auch nicht verdrängendes (engl. *non-preemptive*) *Scheduling* und Präemptionsgrenzen, um den Gesamt-Stapelverbrauch zu begrenzen. Hierbei ist der grundlegende Gedanke, dass eine Einschränkung der Verdrängung zwischen Tasks die Anzahl der gleichzeitig auf dem Systemstapel allozierten Tasks reduziert, wodurch der Gesamtspeicherverbrauch weiter sinkt. Hierbei muss aber bedacht werden, dass eine Reduktion der Verdrängungen die Planbarkeit (engl. *Schedulability*) der Taskmengen verschlechtern kann, daher muss der Grad der Verdrängung gegen die Planbarkeit des Systems und den gesamten RAM-Verbrauch des Systems abgewogen werden.

Eine weitere Anforderung an Betriebssysteme für kleine Mikrocontroller ist die **Skalierbarkeit** durch die Unterstützung reduzierter Versionen der API für Implementierungen mit kleinerem Speicherverbrauch. Bei in großen Mengen produzierten Systemen beeinflusst dieser Speicherverbrauch in der Tat wesentlich die gesamten Systemkosten. In diesem Kontext wird Skalierbarkeit durch das Konzept der *Conformance Classes* realisiert, die bestimmte Teilmengen der Betriebssystem-API definieren. Zu den *Conformance Classes* gehört ein definierter Weg, auf eine Klasse mit mehr Funktionalität zu wechseln. Dabei ist das Ziel, partielle Implementierungen des Standards mit reduziertem Speicherverbrauch zu unterstützen. Die vom OSEK/VDX-Standard (und damit von ERIKA *Enterprise*) unterstützten *Conformance Classes* sind im Einzelnen:

- BCC1: Dies ist die kleinste *Conformance Class*, sie unterstützt ein Minimum von 8 Tasks mit unterschiedlicher Priorität und eine gemeinsame Ressource.
- BCC2: Diese *Conformance Class* erweitert BCC1 um die Möglichkeit, mehr als eine Task mit derselben Priorität zu verwenden. Jede Task kann ausstehende Aktivierungen besitzen, d.h. das Betriebssystem führt über die Anzahl der schon aktivierten, aber noch nicht ausgeführten Instanzen Buch.
- ECC1: Diese *Conformance Class* fügt der Klasse BCC1 die Möglichkeit hinzu, erweiterte Tasks zu verwenden, die auf das Eintreten eines Ereignisses warten können.
- ECC2: Diese *Conformance Class* verfügt sowohl über mehrfache Aktivierungen wie über erweiterte Tasks.

ERIKA *Enterprise* erweitert diese *Conformance Classes* durch die Bereitstellung der folgenden beiden *Conformance Classes*:

- EDF: Diese *Conformance Class* verwendet einen *Earliest Deadline First* (EDF)-*Scheduler*, der für den Einsatz auf kleinen Mikrocontrollern optimiert wurde, anstelle eines *Schedulers* mit festen Prioritäten (siehe Abschnitt 6.2.1).
- FRSH: Diese *Conformance Class* erweitert den EDF-*Scheduler* durch einen Ressourcen-Reservierungs-*Scheduler*, der auf dem IRIS *Scheduling*-Algorithmus basiert [381].

Eine weitere interessante Eigenschaft von OSEK/VDX-Systemen ist es, dass das System eine API zur Unterbrechungskontrolle zur Verfügung stellt. Dies ist ein großer Unterschied im Vergleich zu POSIX-basierten Systemen, in denen Unterbrechungen ausschließliche Aufgabe des Betriebssystems sind und nicht über eine

Betriebssystem-API zur Verfügung gestellt werden. Der Grund hierfür ist, dass die Benutzer kleiner Mikrocontroller oft Unterbrechungsprioritäten direkt kontrollieren möchten; daher ist es wichtig, eine Standardmethode zur Aktivierung und Deaktivierung von Unterbrechungen zur Verfügung zu stellen. Zudem spezifiziert der OSEK/VDX-Standard zwei Arten von Unterbrechungsbehandlungsroutinen (engl. *Interrupt Service Routines* (ISR)):

- Kategorie 1: einfacher und schneller, implementiert keinen Aufruf des *Schedulers* am Ende der ISR;
- Kategorie 2: diese ISR kann einige Funktionen aufrufen, die das *Scheduling*-Verhalten ändern. Am Ende der ISR kann *Scheduling* stattfinden. ISR1 hat stets eine höhere Priorität als ISR2.

Eine wichtige Eigenschaft von OSEK/VDX-Kernen ist die Möglichkeit, den Speicherverbrauch genau zu kontrollieren. Dazu kann aus Produktionsversionen Code zur Fehlerüberprüfung entfernt werden, zudem sind spezielle Einsprungpunkte (engl. *hooks*) vorgesehen, die vom System aufgerufen werden, wenn bestimmte Ereignisse auftreten. Diese Eigenschaften erlauben es, den Speicherverbrauch einer Anwendung präzise zu steuern; die während des *Debugging* genutzte Version ist größer (und sicherer), während die Produktionsversion kleiner ist, da zu diesem Zeitpunkt die meisten Fehler im Code bereits gefunden und beseitigt sein sollten.

Der OSEK/VDX-Standard unterstützt eine Sprache namens ORTI, um die *Debugging*-Möglichkeiten zu verbessern. Diese Sprache enthält eine Beschreibung, an welchen Speicherstellen die verschiedenen Objekte des Betriebssystems alloziert wurden. Die ORTI-Datei wird normalerweise vom OIL-Compiler erzeugt. Sie wird von *Debuggern* genutzt, um detaillierte Informationen über im System definierte Betriebssystemobjekte anzuzeigen (beispielsweise könnte der *Debugger* eine Liste aller Tasks im System zusammen mit ihrem jeweiligem Status ausgeben).

Alle vom OSEK/VDX-Standard definierten Eigenschaften wurden im *Open Source*-Kern ERIKA *Enterprise* implementiert [157]. Zielplattform ist eine Reihe von Mikrocontrollern. Die endgültige Codegröße schwankt dabei zwischen 1 und 5 Kilobyte Objektcode. ERIKA *Enterprise* implementiert auch weitere Eigenschaften, wie den EDF-*Scheduler*, und stellt damit ein offenes und kostenloses Betriebssystem zur Verfügung, das zum Lernen, zum Testen und auch zum Implementieren echter Anwendungen für Industrie- und Lehrzwecke eingesetzt werden kann.

4.4 Linux für eingebettete Systeme

Die Anforderungen an die Funktionalität eingebetteter Systeme steigen zusehends. Beispiele hierfür sind Netzwerkkonnektivität (besonders für das Internet der Dinge) oder anspruchsvolle Grafikanzeigen. Zur Realisierung dieser Funktionalität muss ein typisches eingebettetes Betriebssystem durch eine große Menge an Software erweitert werden. Teile dieser komplexen Funktionen können auch in kleine eingebettete Betriebssysteme integriert werden, wie z.B. ein kleiner Internetprotokollstack

[142]. Die Integration mehrerer zusätzlicher Softwarekomponenten stellt aber eine komplexe Aufgabe dar und kann zu Beeinträchtigungen der Funktion wie auch der Sicherheit eines Systems führen.

Ein anderer Ansatz, der durch das exponentielle Wachstum von Komponentendichten in Halbleitern nach dem Mooreschen Gesetz ermöglicht wird, ist die Anpassung eines Softwaresystems, das die gewünschte Funktionalität bereits enthält und in einer großen Vielfalt von Umgebungen getestet wurde, an eingebettete Systeme. In diesem Bereich hat sich Linux[13] als Betriebssystem der Wahl für eine große Zahl komplexer eingebetteter Anwendungen etabliert. Beispiele für diese Anwendungen sind Internetrouter, GPS-Navigationssysteme, Netzwerkspeicher, Fernsehgeräte mit Internet-Zusatzfunktionen (sogenannte „*Smart-TVs*") und Mobiltelefone. Diese Anwendungen profitieren davon, dass Linux einfach portierbar ist. Das System ist mittlerweile für mehr als 30 Prozessorarchitekturen verfügbar, wie z.B. die in eingebetteten Systemen verbreiteten ARM-, MIPS- und PowerPC-Architekturen. Des Weiteren bringt Linux den Vorteil mit sich, durch seine Verfügbarkeit unter freien Softwarelizenzen die ansonsten anfallenden Lizenzkosten für kommerzielle eingebettete Systeme einzusparen.

4.4.1 Struktur und Größe von Linux für eingebettete Systeme

Genau genommen bezeichnet der Begriff „Linux" nur den eigentlichen Kern eines Linux-basierten Betriebssystems. Ein komplettes, funktionierendes Betriebssystem benötigt eine Menge an zusätzlichen Komponenten, die auf dem Linux-Kern aufbauen. Die Konfiguration eines typischen Linux-Systems, das auch die systemnahen Komponenten außerhalb des Kerns beinhaltet, ist in Abb. 4.14 dargestellt. Oberhalb des Linux-Kerns finden sich – meist dynamisch gelinkte – Bibliotheken, die grundlegende Funktionen für Systemwerkzeuge und Anwendungen bereitstellen. Gerätetreiber werden in Linux üblicherweise als dynamisch ladbare Kernmodule realisiert, ein direkter Zugriff von Anwendungen auf die Hardware ist aber auch möglich.

Die Verfügbarkeit des Linux-Quellcodes ermöglicht es, den Kern und andere Systemkomponenten an die Anforderungen einer gegebenen Anwendung und Plattform anzupassen. Daraus folgt, dass auch kleine Linux-Systeme realisierbar sind, die auf Systemen mit wenig Hauptspeicher genutzt werden können.

Eine der zentralen Komponenten eines Unix-artigen Systems wie Linux ist die C-Bibliothek libc. Diese stellt grundlegende Funktionen wie z.B. Datei-Ein/Ausgabe, Prozesssynchronisation und -kommunikation, Behandlung von Zeichenketten, arithmetische Operationen und Speicherverwaltung zur Verfügung. Die normalerweise in Linux verwendete Variante ist die GNU libc (glibc). Diese wurde für Server- und Desktop-Systeme entwickelt und stellt daher deutlich mehr Funktionalität zur Verfügung als normalerweise in eingebetteten Systemen benötigt wird. Linux-basierte

[13] Dieser Abschnitt zu Linux in eingebetteten Systemen wurde von M. Engel (NTNU Trondheim) beigesteuert.

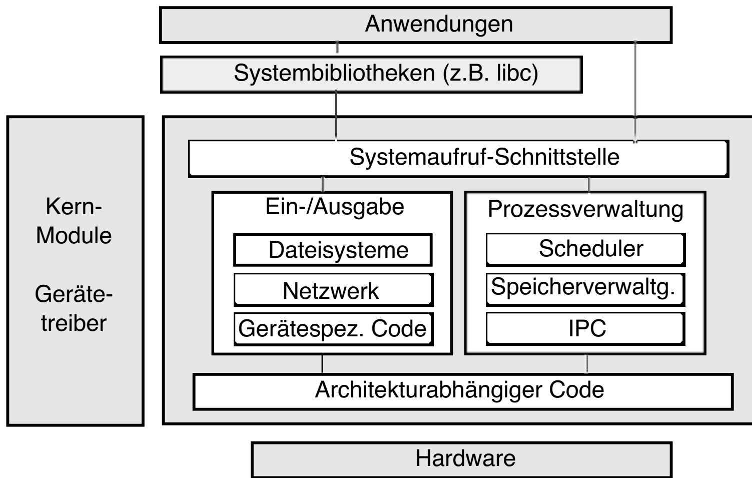


Abb. 4.14 Struktur eines typischen Linux-Systems

Android®-Systeme verwenden anstelle von libc die Bionic-Bibliothek, die von BSD-Unix-Varianten abgeleitet ist. Bionic ist dafür entworfen worden, Systeme mit geringerer Rechenleistung zu unterstützen. Dies geschieht z.B. durch die Bereitstellung einer maßgeschneiderten Version der *pthreads* Multithreading-Bibliothek, die effizient die in Android verwendete Java-VM Dalvik unterstützt. Die Codegröße von Bionic beträgt ungefähr die Hälfte einer typischen glibc-Version[14].

libc-Version	musl	uClibc	dietlibc	glibc
Größe der statischen Bibliothek	426 kB	500 kB	120 kB	2.0 MB
Größe der dynamischen Bibliothek	527 kB	560 kB	185 kB	7,9 MB
Minimale Größe C-Programm, statisch gelinkt	1,8 kB	5 kB	0,2 kB	662 kB
Minimale Größe „Hello, World", statisch gelinkt	13 kB	70 kB	6 kB	662 kB

Abb. 4.15 Größenvergleich verschiedener libc-Konfigurationen für Linux

Daneben existieren aber auch weitere deutlich kleinere Implementierungen der C-Bibliothek, wie z.B. newlib, musl, uClibc, PDCLib und dietlibc. Jede dieser Varianten ist für einen bestimmten Anwendungsfall optimiert, so ist z.B. musl für den Einsatz in statisch gelinkten Programmen optimiert. uClibc wurde ursprünglich für Linux-Systeme ohne Speicherverwaltungseinheit (MMU)[15] entwickelt, wogegen newlib eine plattformunabhängige C-Bibliothek ist, die auch für verschiedene andere Betriebssystemplattformen verfügbar ist. Die Größen der zugehörigen Binärdateien

[14] Die Größe der glibc-Bibliothek beinhaltet allerdings umfangreichere Unterstützung für Internationalisierung.

[15] Eine Einführung in MMUs findet sich in Anhang C.

bewegen sich im Bereich von 185 kB (dietlibc) bis 560 kB (uClibc). Im Vergleich dazu beträgt die typische Größe der glibc 7,9 MB (alle Werte beziehen sich auf die x86-Versionen)[16]. Abb. 4.15 gibt einen Überblick über die Größen verschiedener libc-Varianten und damit übersetzter Programme.

Neben der C-Bibliothek kann auch die Funktionalität, Größe und Anzahl der mit dem Betriebssystem mitgelieferten Systemprogramme an die Anforderungen der jeweiligen Anwendung angepasst werden. Diese Programme werden in Linux zur Steuerung des Systemstarts und -betriebs sowie zur Systemüberwachung benötigt, z.B. in Programmen zum Einbinden eines Dateisystems, zur Konfiguration von Netzwerkschnittstellen oder zum Kopieren von Dateien. Wie glibc enthält auch ein Linux-System selbst eine Menge an Programmen, die für eine große Anzahl verschiedener Anwendungsfälle gedacht sind, in den meisten Fällen aber für eingebettete Systeme nicht notwendig sind.

Eine Alternative zu dieser Menge an diversen Programmen ist *BusyBox*, eine Software, die eine Reihe von vereinfachten essentiellen Unix-Werkzeugen in einer einzigen ausführbaren Datei vereint. *BusyBox* wurde speziell für eingebettete Systeme mit sehr begrenzten Ressourcen entwickelt. Sie reduziert den zusätzlichen Speicherbedarf des ausführbaren Dateiformats und ermöglicht, dass Code von mehreren Anwendungen ohne Verwendung einer Bibliothek geteilt wird. Ein Vergleich von *BusyBox* mit weiteren Ansätzen, eine Menge an kleinen Werkzeugen für den Systembetrieb zur Verfügung zu stellen, findet sich unter [531].

4.4.2 Echtzeiteigenschaften

Eine der komplexesten Herausforderungen bei der Anpassung eines Allzweck-Betriebssystemkerns für eingebettete Systeme ist die Einhaltung von Echtzeiteigenschaften. Wie bereits in Abb. 4.3 dargestellt, ist ein verbreiteter Ansatz hierzu, den Linux-Kern und alle Linux-Prozesse als dedizierte Task eines darunterliegenden Echtzeitbetriebssystems auszuführen, die nur aktiv ist, wenn keine Echtzeittask Rechenzeit benötigt. Für Linux existieren verschiedene Implementierungen, die diesen Ansatz realisieren. RTAI (*real-time application interface*) [138] basiert auf dem Adeos-Hypervisor[17], der als Linux-Kernerweiterung implementiert ist. Adeos ermöglicht es, verschiedene priorisierte Domänen, wovon eine der Linux-Kern selbst ist, zeitgleich auf derselben Hardware auszuführen. Darauf basierend stellt RTAI eine Diensteschnittstelle (engl. *Application Programmer Interface* (API)) zur Verfügung, die z.B. die Steuerung von Unterbrechungen und Systemzeitgebern ermöglicht. Xenomai [182] wurde einige Jahre lang gemeinsam mit RTAI entwickelt, bis es im Jahr 2005 ein eigenständiges Projekt wurde. Es basiert auf einem abstrakten „nucleus"-Echtzeitkern, der Funktionalitäten wie Echtzeit-*Scheduling*, Zeitgeber, Speicherverwaltung und virtuelle Dateisysteme zur Verfügung stellt. Die

---

[16] Die Werte entstammen einem umfassenden Vergleich verschiedener libc-Varianten von Eta Labs, verfügbar unter http://www.etalabs.net/compare_libcs.html.

[17] Siehe http://home.gna.org/adeos/.

beiden Projekte unterscheiden sich in ihren Zielen und ihrer Implementierung. Allerdings unterstützen beide das *Real-Time Driver Model* (RTDM), einen Ansatz zur Vereinheitlichung von Schnittstellen für Gerätetreiber und darauf basierender Anwendungen für echtzeitfähige Linux-Systeme. Der dritte Ansatz, der ebenfalls einen unterliegenden Echtzeitkern verwendet, ist RTLinux [608], das als Projekt am *New Mexico Institute of Mining and Technology* entwickelt und später von der Firma FSMLabs, die im Jahr 2007 von WindRiver übernommen wurde, kommerzialisiert wurde. Das zugehörige Produkt wurde bereits im Jahr 2011 eingestellt. Der Einsatz von RTLinux in Produkten wurde kontrovers diskutiert, da dessen Erfinder ihr geistiges Eigentum, für das sie ein Software-Patent [607] erhielten, vehement verteidigten. Die Entscheidung, die Methoden hinter RTLinux zu patentieren, rief bei Linux-Entwicklern wenig Begeisterung hervor, was schließlich zur Entwicklung der oben erwähnten alternativen Projekte RTAI und Xenomai führte.

Ein neuerer Ansatz, Linux um Echtzeitfähigkeiten zu erweitern, ist `SCHED_DEADLINE`, das seit 2014 (Kernversion 3.14) im Linux-Kern integriert ist. Dabei handelt es sich um ein CPU-*Scheduling*-Verfahren, das auf den Algorithmen für *Earliest Deadline First* (EDF) und *Constant Bandwidth Server* (CBS) [3] basiert und die Reservierung von Ressourcen unterstützt. Das `SCHED_DEADLINE`-Verfahren kann dabei gemeinsam mit anderen Linux *Scheduling*-Verfahren aktiv sein, es hat aber stets Vorrang vor allen anderen Verfahren, um Echtzeiteigenschaften garantieren zu können.

Jede unter `SCHED_DEADLINE` eingeplante Task τ_i erhält ein Laufzeit-Budget C_i und eine Periode T_i. Dies zeigt dem Kern an, dass diese Task in einer Periode T_i jeweils C_i Zeiteinheiten auf einem beliebigen Prozessor benötigt. Bei Echtzeitanwendungen entspricht T_i der minimalen Zeit zwischen zwei aufeinanderfolgenden Aktivierungen der Task, C_i entspricht der WCET für jede Ausführung der Task. Wenn eine neue Task zum *Scheduling*-Verfahren hinzugefügt werden soll, wird ein Test auf Einplanbarkeit (*schedulability*) durchgeführt. Die Task wird nur akzeptiert, wenn der Test erfolgreich ist. Während des *Schedulings* wird eine Task suspendiert und auf die kommende Periode verlagert, wenn sie versucht, mehr Rechenzeit zu beanspruchen als ihrem Budget entspricht. Diese sogenannte *non-work conserving*-Strategie[18] ist notwendig, um die temporale Isolation zwischen Tasks sicherzustellen. Daher ist auf Einprozessorsystemen und partitionierten Mehrprozessorsystemen (mit Tasks, deren Ausführung auf eine bestimmte CPU beschränkt ist) für alle akzeptierten `SCHED_DEADLINE`-Tasks garantiert, dass sie für eine Gesamtzeit eingeplant werden, die ihrem Zeitbudget in jedem Zeitfenster entspricht.

Für den allgemeinen Fall, in dem Tasks zwischen verschiedenen Kernen eines Mehrprozessorsystems migrieren können, gilt die übliche *tardiness*-Grenze für globales EDF [128], da `SCHED_DEADLINE` globales EDF implementiert (wie im Detail in Abschnitt 6.3.3 beschrieben). Benchmarks aus [337] geben den Anteil verpasster Deadlines mit weniger als 0,2% an, wenn `SCHED_DEADLINE` auf einem Vierprozessorsystem mit einer Auslastung von 380% läuft und von weniger als 0,615% bei einer Auslastung von 390%. Die angegebenen Zahlen für ein Sechsprozessorsystem

[18] Dies bedeutet, dass der Prozessor *idle* sein kann, auch wenn Tasks ausgeführt werden könnten. Eine Definition des Begriffs findet sich in Kapitel 6 auf Seite 336.

liegen in einer ähnlichen Größenordnung. Selbstverständlich treten bei Einprozessorsystemen und Systemen, bei denen eine Task an einen Prozessorkern gebunden wurde, keine verpassten Deadlines auf.

4.4.3 Dateisysteme für Flash-Speicher

Die Anforderungen eingebetteter Systeme an permanenten Speicher unterscheiden sich von denen im Server- und Desktop-Bereich. Oft existiert eine große Menge an statischen (nur-lese) Daten, wogegen die Menge an variablen Daten in vielen Fällen sehr gering ist.

Dateisysteme können sich diese besonderen Bedingungen zu Nutze machen. Da ein Großteil der nur-lese-Daten in aktuellen eingebetteten Systemen als *Flash*-ROM realisiert ist, ist die Optimierung für diese Art von Speicher ein wichtiger Aspekt bei der Nutzung von Linux in eingebetteten Systemen. Daher wurden eine Reihe verschiedener Dateisysteme entwickelt, die speziell für die Nutzung von NAND-*Flash* basierten Speichern ausgelegt sind.

Eines der stabilsten *Flash*-spezifischen Dateisysteme ist das protokollbasierte *Journaling Flash File System version* 2 (JFFS2) [596]. JFFS2 protokolliert Änderungen an Dateien und Verzeichnissen im *Flash*-Speicher in Knoten (*nodes*). Es gibt zwei Arten von Knoten. *Inodes* (dargestellt in Abb. 4.16) bestehen aus einem *Header* mit Datei-Metadaten, gefolgt von optionalen Dateiinhalten, wogegen *dirent*-Knoten Verzeichniseinträge darstellen, die jeweils einen Namen und die Nummer einer *inode* beinhalten. Knoten werden bei ihrer Erzeugung als gültig markiert und verlieren ihre Gültigkeit, wenn eine neuere Version an einer anderen Stelle im *Flash*-Speicher erzeugt wurde. JFFS2 unterstützt die transparente Kompression von Daten durch die Speicherung komprimierter Daten als Nutzdaten der *inodes*.

Gemeinsamer Header für Knoten				Knotenspezifischer Inhalt
Bitmaske	Knotentyp	Gesamte Knotenlänge	Knotenheader CRC	Inode/Verzeichniseintrag
0x1985				

Abb. 4.16 Inhalt einer JFFS2-Inode

Verglichen mit anderen protokollbasierten Dateisystemen wie Berkeley lfs [473] existiert in JFFS2 allerdings kein zirkuläres Protokoll. Stattdessen verwendet JFFS2 Blöcke, die der Größe der löschbaren Segmente des unterliegenden *Flash*-Speichers entsprechen. Wie in Abb. 4.17 gezeigt, werden Blöcke einzeln vom Start des Blocks ab mit Knoten gefüllt.

Bereinigte (*clean*) Blöcke enthalten ausschließlich gültige Knoten, wogegen unbereinigte (*dirty*) Blöcke mindestens einen ungültigen (veralteten) Block beinhalten. Um Speicher freizugeben, sammelt ein Dienst zur automatischen Speicherbereini-

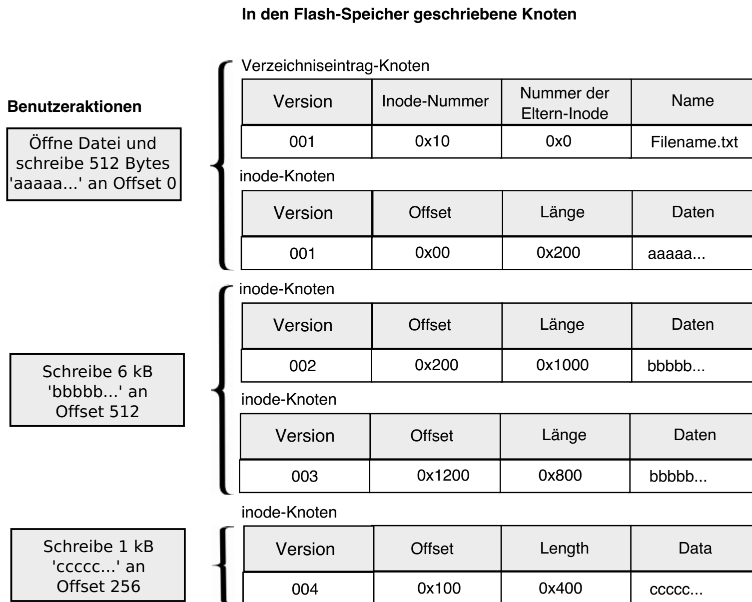


Abb. 4.17 Änderung des *Flash*-Inhalts bei Schreibvorgängen in JFFS2

gung (engl. *garbage collector*) die unbereinigten Blöcke im Hintergrund ein und gibt diese wieder frei. Gültige Knoten aus unbereinigten Blöcken werden dabei in neue Blöcke kopiert und ungültige übersprungen. Nach dem Ende des Kopiervorgangs wird der unbereinigte Block als freigegeben markiert. Der *garbage collector* kann auch bereinigte Blöcke nutzen, um die Abnutzung des *Flash*-Speichers gleichmäßig zu gestalten (engl. *wear-leveling*). Damit wird vermieden, dass Blöcke wiederholt in einem beschränkten kleinen Bereich eines größtenteils statischen Dateisystems, wie es oft in eingebetteten Systemen der Fall ist, gelöscht werden.

4.4.4 Verringerung des Hauptspeicherbedarfs

Traditionell sehen Unix-ähnliche Systeme den Hauptspeicher (RAM) als einen Cache für Sekundärspeicher auf einer Festplatte, d.h. Auslagerungsspeicher (engl. *swap space*), an [386]. Dies ist eine sinnvolle Annahme für Server- und Desktop-Systeme mit großen Festplatten und entsprechend großem Speicherbedarf. Bei eingebetteten Systemen führt dies jedoch zu einer Verschwendung von Ressourcen, da Programme, die im nichtflüchtigen Speicher eines Systems abgelegt sind, vor ihrer Ausführung zunächst in den flüchtigen Hauptspeicher kopiert werden müssen. Dies trifft auch auf den üblicherweise recht großen Betriebssystemkern selbst zu.

Um diesen doppelten Speicherbedarf zu vermeiden, wurden verschiedene Techniken zur direkten Ausführung von Programmcode aus Festwertspeichern (engl. *Execute-in-Place* (XiP)) entwickelt. Diese Technik ist bei kleineren mikrocontrollerbasierten Systemen üblich, bringt aber einige Probleme mit sich. Zum Einen muss der nichtflüchtige Speicher, der das auszuführende Programm beinhaltet, Zugriffe mit Byte- oder Wort-Granularität unterstützen. Zum Anderen werden ausführbare Programme meist in einem Datenformat wie ELF gespeichert, das Metainformationen (z.B. Symbole für *Debugging*) beinhaltet und vor der Ausführung zunächst gelinkt werden muss.

Die Unterstützung für XiP-Techniken wird meist als eigenes Dateisystem implementiert, z.B. beim *Advanced XiP Filesystem* (AXFS) [42], das zusätzlich die Kompression von nur-lese-Daten erlaubt. Die Verwendung von XiP ist besonders für den Linux-Kern selbst nützlich, da dieser normalerweise eine große Menge an nicht auslagerbarem Speicher belegt. Die Ausführung des Kerns direkt aus dem *Flash* lässt entsprechend mehr Hauptspeicher für Benutzerprozesse übrig. XiP für Benutzerprozesse selbst ist weniger nützlich, da der Linux-Kern in Systemen mit virtuellem Speicher jeweils nur die benötigten Speicherseiten einer ausführbaren Datei in den Hauptspeicher lädt. Dadurch wird der RAM-Bedarf für diese Programme automatisch minimiert.

Die Verfügbarkeit von *Flash*-Speicher mit byte- oder wortgranularem Zugriff, wie für XiP benötigt, ist für aktuelle Systeme zumeist eine Kostenfrage. Die üblicherweise verwendete NAND-*Flash*-Technologie, die z.B. in SD-Speicherkarten und SSDs zum Einsatz kommt, ist kostengünstig, erlaubt aber jeweils nur blockweisen Zugriff, ähnlich wie konventionelle Festplatten. NOR-*Flash* hingegen ist eine Technologie, die wahlfreien Zugriff erlaubt und die sich damit für die Implementierung von XiP-Techniken eignet. Allerdings ist NOR-*Flash* meist eine Größenordnung teurer als NAND-*Flash* und zudem langsamer als RAM-basierter Hauptspeicher. Entsprechend kann es für viele Systeme eine sinnvolle Entwurfsentscheidung sein, einen größeren RAM-Speicher anstelle von NOR-*Flash*-basierten XiP-Techniken zu verwenden.

4.4.5 uClinux – Linux für Systeme ohne MMU

Eine weitere Ressourcenbeschränkung findet sich schließlich in kleineren mikrocontrollerbasierten Systemen wie der Cortex-M-Serie von ARM. Die Prozessorkerne in diesen Systemen wurden für den Einsatz mit typischen Echtzeitbetriebssystemen entwickelt, die oft der einfachen Struktur eines Bibliotheks-Betriebssystems entsprechen, wie für Erika (siehe Seite 240) beschrieben. Daher fehlt diesen Mikrocontrollern wichtige Hardware zur Unterstützung von Betriebssystemen, speziell eine Speicherverwaltungseinheit (engl. *memory management unit* (MMU), siehe Anhang C). Mit einigen Einschränkungen ist es aber möglich, Linux auf Mikrocontrollern mit ausreichend großem Speicher und vergleichsweise hohen Taktfrequenzen zu nutzen. Entsprechend wurde uClinux als Abwandlung des Linux-Kerns für Systeme ohne

MMU entwickelt. Seit der Kernversion 2.5.45 ist uClinux integraler Bestandteil des regulären Linux-Kerns für verschiedene Prozessorarchitekturen wie ARM7TDMI, ARM Cortex-M3/4/7/R, MIPS, M68k/ColdFire sowie FPGA-basierte Prozessoren wie Altera Nios II, Xilinx MicroBlaze und Lattice Mico 32.

Das Fehlen einer Speicherverwaltungseinheit in uClinux-basierten Systemen bringt eine Reihe von Nachteilen mit sich. Eine direkte Folge ist der fehlende Speicherschutz, somit kann jeder Prozess beliebig den Speicher anderer Prozesse auslesen und beschreiben. Das Fehlen einer MMU hat aber auch Folgen für die Unix-eigene Methode zur Erzeugung neuer Prozesse. Normalerweise werden Prozesse in Unix als eine Kopie (Kindprozess) eines existierenden Prozesses durch den `fork()`-Systemaufruf erstellt [470]. Dieser erzeugt aus Effizienzgründen keine komplette Kopie des Speicherinhalts des aufrufenden Prozesses, sondern repliziert nur dessen Seitentabelleneinträge, die im Kindprozess auf dieselben physikalischen Seitenrahmen zeigen wie im aufrufenden Elternprozess. Wenn nun der Kindprozess Daten in seinen Speicher schreibt, wodurch sich sein Speicherinhalt von dem des Elternprozesses unterscheidet, werden nur die davon betroffenen Seitenrahmen mit einer *copy-on-write*-Strategie kopiert. Die fehlende Unterstützung für die *copy-on-write*-Semantik in MMU-losen Systemen und der in Folge benötigte hohe Aufwand, bei Aufruf von `fork()` alle Seiten des Elternprozesses zu kopieren, führen dazu, dass `fork()` in uClinux nicht zur Verfügung steht.

Stattdessen stellt uClinux den `vfork()`-Systemaufruf zur Verfügung. Dieser Systemaufruf nutzt aus, dass die meisten Unix-Prozesse direkt nach ihrer Erzeugung den Systemaufruf `exec()` verwenden, um ihren Speicherinhalt durch den einer anderen ausführbaren Datei zu ersetzen:

```
pid_t kindPID;
kindPID = vfork();
if (kindPID == 0) { // im Kindprozess
   exec("/bin/sh", "sh", 0);
}
printf("Elternprozess läuft wieder, PID des Kindprozesses ist %d", kindPID);
```

Der direkte Aufruf von `exec()` nach `fork()` bedingt, dass der Inhalt des gesamten Adressraums des neu erzeugten Prozesses auf jeden Fall ersetzt wird und nur der kleine Teil des Kindprozesses tatsächlich ausgeführt wird, der `exec()` aufruft. Verglichen mit dem Standardverhalten von Unix garantiert `vfork`, dass der Elternprozess nach dem Aufruf von fork so lange angehalten wird, bis der Kindprozess `exec()` aufgerufen hat. Dadurch ist der Elternprozess nicht dazu in der Lage, die Ausführung des Kindprozesses zu beeinflussen (z.B. durch Schreibvorgänge), bis dessen neuer Speicherinhalt geladen wurde. Zur sicheren Verwendung von `vfork()` müssen allerdings einige Einschränkungen beachtet werden. Es ist nicht erlaubt, den *Stack* des Kindprozesses zu verändern, d.h. vor `exec` dürfen keine Funktionsaufrufe erfolgen. Eine Folge davon ist, dass eine Rückkehr aus dem `vfork`-Aufruf im Falle eines Fehlers, z.B. bei zu geringem Speicher oder einem Fehler bei der Ausführung des neuen Programms, unmöglich ist, da dies den *Stack* verändern würde. Stattdessen lautet die Empfehlung, im Fehlerfall den Kindprozess mit Hilfe des `exit()`-Systemaufrufs zu beenden.

Zusammenfassend lässt sich feststellen, dass uClinux ein funktionierender Ansatz ist, einen Teil der Linux-Funktionalität auf kleinen Mikrocontrollersystemen zu realisieren. Die auf dem *Chip* verfügbare Menge an RAM ist aber aktuell auch bei größeren Mikrocontrollern auf wenige hundert Kilobyte beschränkt. Eine minimale uClinux-Version benötigt aber rund 8 MB RAM und erfordert daher einen zusätzlichen externen RAM-Baustein. Für Systeme mit geringem Hauptspeicher wird also auch in Zukunft ein traditionelles Echtzeitbetriebssystem eher das System der Wahl bleiben.

4.4.6 Evaluation der Nutzung von Linux in eingebetteten Systemen

Neben den technischen Kriterien bringt die Entscheidung, ob ein eingebettetes System auf Linux basieren soll, auch rechtliche und wirtschaftliche Fragen mit sich.

Auf der technischen Seite unterstützt Linux eine große Anzahl an Prozessorarchitekturen, SoCs, Peripherieeinheiten und Kommunikationsprotokollen, die üblicherweise bei eingebetteten Systemen zum Einsatz kommen, wie z.B. das Internet-Protokoll TCP/IP, CAN, Bluetooth® und IEEE802.15.4/ZigBee®. Linux stellt eine POSIX-ähnliche Programmierschnittstelle zur Verfügung, welche die Portierung von existierendem Code erleichtert. Dieser Code muss nicht notwendigerweise in C oder C++ geschrieben sein, sondern kann auch Skriptsprachen wie Python und Lua oder gar spezialisiertere Sprachen wie Erlang verwenden. Entwicklungswerkzeuge für Linux sind kostenfrei verfügbar und lassen sich einfach in existierende IDE-basierte Entwicklungsprozesse, z.B. mit Eclipse, und Dienste zur kontinuierlichen Integration wie Jenkins integrieren. Im allgemeinen ist die Codebasis von Linux wohlerprobt, allerdings variiert die Qualität der Unterstützung je nach Plattform. Bei Verwendung einer weniger verbreiteten Hardwareplattform wird empfohlen, die Stabilität der Unterstützung für den Prozessor und die Gerätetreiber sorgfältig zu prüfen. Ein Nachteil der Verwendung von Linux ist die Komplexität der großen Codebasis, die einen guten Einblick in und Erfahrung mit dem System erfordert, um Probleme finden und beheben zu können. Hier bieten aber verschiedene Halbleiterhersteller und Drittanbieter kommerzielle Unterstützung für eingebettetes Linux an, bis hin zur Verfügbarkeit kompletter grundlegender Systemportierungen, sogenannter *board support software packages* (BSPs), für verschiedene Referenzsysteme.

Aus einem wirtschaftlichen Blickwinkel ist der offensichtliche Vorteil von Linux, dass der Quellcode kostenfrei zur Verfügung steht. Der Linux-Kern unterliegt der GPL-Lizenz Version 2[19]. Diese erfordert, dass der Quellcode für Modifikationen der existierenden Codebasis zusammen mit dem erzeugten Binärcode zur Verfügung gestellt werden muss. Dies kann dazu führen, dass Geschäftsgeheimnisse von Hardwarekomponenten offengelegt werden oder Vertraulichkeitsvereinbarungen mit den Anbietern von Hardwarekomponenten gebrochen werden. Für bestimmte Hardware, wie z.B. Grafikkartentreiber, wird dies umgangen, indem binäre Code-„Blobs"

[19] Siehe http://www.gnu.org/licenses/gpl-2.0.html.

verwendet werden, die nur durch einen quelloffenen Lader zum Kern hinzugefügt werden. Dieser Ansatz wird aber von den Linux-Kernentwicklern abgelehnt.

Ein zunehmend kritisches Problem ist die Sicherheit von Linux-basierten eingebetteten Systemen, besonders im Internet der Dinge. Viele den Linux-Kern betreffende Sicherheitslücken finden sich auch in eingebettetem Linux. Kostengünstige Verbrauchergeräte, wie Internet-basierte Kameras, Router und Mobiltelefone, erhalten nur in Ausnahmefällen Softwareaktualisierungen, sind aber oft viele Jahre im Einsatz. Dadurch sind diese Geräte Sicherheitslücken ausgesetzt, die bereits aktiv ausgenutzt werden, z.B. für massive verteilte Angriffe auf die Verfügbarkeit von Servern (sogenannte *denial-of-service*-Angriffe (DDOS)), die von Tausenden von kompromittierten Linux-Geräten ausgehen. Entsprechend müssen für einen sicheren Betrieb von Linux-basierten Systemen die Kosten für eine kontinuierliche Aktualisierung der Software sowohl für in Produktion befindliche wie auch ausgelaufene, aber noch aktiv betriebene Geräte, in der Kalkulation berücksichtigt werden.

4.5 Hardware-Abstraktionsschicht

Hardware-Abstraktionsschichten (engl. *Hardware Abstraction Layers* (HALs)) ermöglichen es, auf Hardware über eine Hardware-unabhängige Programmierschnittstelle (API) zuzugreifen. Wir könnten uns hier z.B. eine Hardware-unabhängige Methode zum Zugriff auf Zeitgeber vorstellen, die unabhängig von den jeweiligen Adressen der Zeitgeber funktioniert. Hardware-Abstraktionsschichten kommen meist zwischen der Hardware und dem Betriebssystem zum Einsatz. Sie stellen *Software Intellectual Property* (IP) zur Verfügung, sind aber weder Teil des Betriebssystems, noch können sie als *Middleware* angesehen werden. Ein Überblick über Arbeiten in diesem Bereich ist bei Ecker, Müller und Dömer [145] zu finden.

4.6 *Middleware*

Kommunikationsbibliotheken sind ein Mittel, um Sprachen, die keine entsprechenden Eigenschaften besitzen, um Funktionalität zur Kommunikation zu erweitern. Sie stellen Kommunikationsfunktionalität auf Basis der grundlegenden Funktionalität des Betriebssystems zur Verfügung. Da sie auf dem Betriebssystem aufbauen, können sie Betriebssystem-unabhängig sein (und damit auch unabhängig von der verwendeten Prozessorhardware). Als Ergebnis entstehen damit **vernetzte cyberphysikalische Systeme**. Derartige Kommunikation wird auf für das Internet der Dinge benötigt. Ein Trend geht dahin, sowohl Kommunikation innerhalb eines lokalen Systems wie auch Kommunikation über weitere Entfernungen zu unterstützen. Auch hier nimmt die Verwendung von Internetprotokollen weiter zu. Häufig erlauben solche Protokolle die sichere Kommunikation auf der Basis von Ver- und Entschlüs-

selung (siehe Seite 214). Die entsprechenden Algorithmen sind eine spezielle Klasse von *Middleware*.

4.6.1 OSEK/VDX COM

OSEK/VDX® COM ist ein besonderer Kommunikationsstandard für das OSEK-Betriebssystem [441][20] aus dem Automobilbereich. OSEK COM stellt eine „Interaktionsschicht” als Programmierschnittstelle zur Verfügung, über die sowohl interne Kommunikation (Kommunikation innerhalb eines Steuergerätes) wie auch externe Kommunikation (Kommunikation mit anderen Steuergeräten) realisiert werden kann. OSEK COM spezifiziert dabei nur die Funktionalität der Interaktionsschicht. Entsprechende Implementierungen müssen eigenständig entwickelt werden.

Die Interaktionsschicht kommuniziert mit anderen Steuergeräten über eine „Netzwerkschicht” und eine „*Data Link*”-Schicht. Einige der Anforderungen an diese Schichten werden von OSEK COM definiert, aber diese Schichten sind nicht Teil von OSEK COM selbst. Auf diese Weise kann Kommunikation auf Basis verschiedener Netzwerkprotokolle realisiert werden.

OSEK COM ist ein Beispiel für Kommunikations-*Middleware*, die für eingebettete Systeme bestimmt ist. Ausser diesen speziell für eingebettete Systeme entwickelten *Middleware*-Systemen können auch viele für nicht eingebettete Systeme entwickelte Kommunikationsstandards für eingebettete Systeme angepasst werden

4.6.2 CORBA

CORBA® (*Common Object Request Broker Architecture*) [433] ist ein Beispiel eines solchen angepassten Standards. CORBA ermöglicht den Zugriff auf entfernte Dienste, auf entfernte Objekte kann über standardisierte Schnittstellen zugegriffen werden. Klienten kommunizieren in CORBA mit lokalen Hilfsfunktionen (engl. *stubs*), die den Zugriff auf entfernte Objekte vortäuschen. Die Klienten senden Informationen über das gewünschte Objekt sowie eventuelle Parameter an den *Object Request Broker* (ORB, siehe Abb. 4.18). Der ORB ermittelt darauf den Ort des Objektes, auf das zugegriffen werden soll, und sendet die Information über ein standardisiertes Protokoll, wie z.B. IIOP, an den Ort, an dem sich das Objekt befindet. Diese Information wird dann über ein *Skeleton* an das eigentliche Objekt weitergeleitet und die vom Objekt angeforderte Information wird gegebenenfalls wieder über den ORB zurück zum Klienten übertragen.

Der CORBA-Standard verfügt nicht über die für Echtzeitanwendungen notwendige Vorhersagbarkeit. Daher wurde ein spezieller Echtzeit-CORBA-Standard (RT-CORBA) definiert [429]. Eine sehr wichtige Eigenschaft von RT-CORBA ist es,

[20] OSEK ist ein Warenzeichen der Continental Automotive GmbH.

Abb. 4.18 Zugriff auf entfernte Objekte mit CORBA

Ende-zu-Ende-Vorhersagbarkeit von Pünktlichkeit in einem System mit festen Prioritäten zur Verfügung zu stellen. Dies beinhaltet **das Respektieren von *Thread*-Prioritäten zwischen Klient und Server, um Ressourcen-Wettstreit-Situationen aufzulösen** und ein Begrenzen der Latenzen von Operationsaufrufen. Ein besonderes Problem von Echtzeitsystemen ist es, dass *Thread*-Prioritäten möglicherweise nicht beachtet werden, wenn *Threads* exklusiven Zugriff auf Ressourcen erhalten. Das Problem der Prioritätsumkehr (siehe Seite 231) muss in RT-CORBA behandelt werden. RT-CORBA beinhaltet Vorkehrungen, um die Zeit zu begrenzen, innerhalb derer eine solche Prioritätsumkehr vorkommen kann. RT-CORBA verfügt zudem über Methoden, um *Thread*-Prioritäten zu verwalten. Diese Priorität ist unabhängig von den Prioritäten des unterliegenden Betriebssystems, auch wenn sie mit den Echtzeit-Erweiterungen des POSIX-Standards kompatibel ist [202]. Die *Thread*-Priorität von Klienten kann auf die Server-Seite übertragen werden. Prioritätsverwaltung ist auch für Funktionen verfügbar, die gegenseitigen Ausschluss beim Zugriff auf Ressourcen zur Verfügung stellen. Das ab Seite 233 beschriebene Prioritätsvererbungsprotokoll muss in RT-CORBA-Implementierungen zur Verfügung stehen. Eine Menge an bereits existierenden *Threads* reduziert dabei den Aufwand für die *Thread*-Erzeugung und Instanziierung.

4.6.3 POSIX Threads (Pthreads)

Die POSIX *Thread*-Bibliothek (*Pthreads*) ist eine Programmierschnittstelle (API) für *Threads* auf Betriebssystemebene [36]. *Pthreads* entsprechen dem Betriebssystemstandard IEEE POSIX 1003.1c. Eine Menge von *Threads* kann im selben Adressraum ablaufen, damit kann Kommunikation über gemeinsam genutzten Speicher stattfinden. Dies vermeidet die Speicherkopieroperationen, die MPI (siehe Unterabschnitt 2.8.3) normalerweise mit sich bringt. Die *Pthreads*-Bibliothek eignet sich daher zur Programmierung von Mehrkern-Prozessoren, die sich einen Adressraum teilen. Die Bibliothek beinhaltet eine Standard-API mit Mechanismen für den gegenseitigen Ausschluss. *Pthreads* verwenden vollständig explizite Synchronisation [553]. Die genaue Semantik hängt dabei vom verwendeten Speicherkonsistenzmodell ab. Es ist schwierig, Synchronisation korrekt zu implementieren. Die Bibliothek kann als Grundlage zur Implementierung anderer Programmiermodelle verwendet werden.

4.6.4 UPnP und DPWS

Universal Plug-and-Play (UPnP) ist eine Erweiterung des *Plug-and-Play*-Konzeptes von PCs auf über einem Netzwerk verbundene Geräte. Als Hauptziel wird die Anbindung von Netzwerkdruckern, Speichergeräten und *Routern* in Heim- und Büronetzwerken gesehen [438]. Aus Sicherheitsgründen werden hierbei nur Daten ausgetauscht, Code kann nicht übertragen werden.

Das *Devices Profile for Web Services* (DPWS) zielt auf eine allgemeinere Verwendbarkeit als UPnP. „Das Devices Profile for Web Services (DPWS) definiert eine minimale Menge von Implementierungsbeschränkungen, um sicheres Versenden von Nachrichten zu *Web Services*, deren Auffinden, Beschreibung und Ereignisbehandlung auf ressourcenbeschränkten Geräten zu ermöglichen" [597]. DPWS spezifiziert Dienste zum Auffinden von an einem Netzwerk angebundenen Geräten, zum Austausch von Informationen über verfügbare Dienste und zum Veröffentlichen und Abonnieren (Stichwort: *publish and subscribe*) von Ereignissen.

In Ergänzung der für Hochleistungsrechnen entworfenen Bibliotheken können einige generische Netzwerk-Kommunikationsbibliotheken eingesetzt werden. Diese wurden meist für eine lose Kopplung über Internet-basierte Kommunikationsprotokolle entworfen.

MPI (siehe Seite 125), OpenMP (siehe Seite 126), OSEK/VDX COM, CORBA, *Pthreads*, UPnP und DPWS sind Spezialfälle von Kommunikations-*Middleware* (Software, die in einer Schicht zwischen Betriebssystem und Anwendungen genutzt wird). Ursprünglich wurden sie für die Kommunikation zwischen PCs entwickelt (mit Ausnahme von OSEK/VDX COM). Es gibt jedoch Versuche, die gewonnenen Erkenntnisse und entwickelten Techniken auch für eingebettete Systeme nutzbar zu machen.

Dieser Ansatz könnte besonders für mobile Geräte wie *Smartphones* geeignet sein. Für „harte" Echtzeitsysteme mögen der notwendige Aufwand, ihre Echtzeitfähigkeiten und ihre Dienste unpassend sein.

4.7 Echtzeitdatenbanken

Datenbanken bieten eine bequeme und strukturierte Art, Informationen zu speichern und auf sie zuzugreifen. Dementsprechend besitzen Datenbanken eine API zum Schreiben und Lesen von Informationen. Eine Folge von Lese- und Schreiboperationen wird eine **Transaktion** genannt. Transaktionen können aus einem der folgenden Gründe abgebrochen werden: es könnten Hardwareprobleme, Verklemmungen, Probleme mit der Kontrolle der Nebenläufigkeit usw. auftreten. Eine übliche Anforderung ist, dass Transaktionen den Zustand der Datenbank nicht verändern, bis sie erfolgreich zu Ende gelaufen sind. Daher werden durch Transaktionen angeforderte Änderungen meist nicht realisiert, bis sie ***committed*** werden. Die meisten Transaktionen müssen dabei **atomar** sein. Dies bedeutet, dass das Endergebnis (der neue Zustand der Datenbank), das durch eine Transaktion erzeugt wurde, entweder

der Zustand ist, der durch vollständiges Abarbeiten der Transaktion entsteht oder aber der vorherige Zustand. Zudem muss der Zustand der Datenbank, der aus einer Transaktion resultiert, **konsistent** sein. Die Konsistenzanforderungen beinhalten z.B., dass Ergebnisse von Leseanforderungen derselben Transaktion konsistent sind (also keinen Zustand beschreiben, der niemals in der von der Datenbank modellierten Umgebung existiert hat). Weiterhin soll für einen anderen Benutzer der Datenbank kein Zwischenzustand, der aus einer partiellen Ausführung einer Transaktion entsteht, sichtbar sein (die Transaktionen müssen durchgeführt werden, als ob sie in **Isolation** stattfinden würden). Schließlich müssen die Ergebnisse der Transaktionen persistent sein. Diese Eigenschaft wird auch als **Dauerhaftigkeit** der Transaktionen bezeichnet. Zusammengenommen bilden die vier fett gedruckten Eigenschaften die ACID-Eigenschafen (siehe das Buch von Krishna and Shin [311], Kapitel 5).

Einige Datenbanken erfordern weiche Echtzeitbedingungen. So sind zum Beispiel die Zeitanforderungen für Flugreservierungssysteme weich. Im Gegensatz dazu kann es auch harte Beschränkungen geben. So muss beispielsweise die automatische Erkennung von Fußgängern in automobilen Anwendungen und die Zielerkennung in militärischen Anwendungen harten Echtzeitbedingungen genügen. Die obigen Anforderungen erschweren das Einhalten von Echtzeitbedingungen. Beispielsweise könnten Transaktionen mehrfach abgebrochen worden sein, bevor sie endgültig *committed* werden. Bei allen Datenbanken, die virtuellen Speicher und Festplatten zu Grunde legen, sind die Zugriffszeiten auf die Platten kaum vorhersagbar. Mögliche Lösungen liegen hier bei Hauptspeicherdatenbanken und der Verwendung von *Flash*-Speichern. Eingebettete Datenbanken sind oft klein genug, um solche Ansätze zu realisieren. In anderen Fällen kann es möglich sein, die ACID-Anforderungen zu lockern. Weitere Informationen hierzu finden sich im Buch von Krishna und Shin sowie auch bei Lam und Kuo [320].

4.8 Aufgaben

Die folgenden Aufgaben sollten entweder zu Hause oder während einer Anwesenheitsphase nach dem *flipped classroom*-Konzept [376] bearbeitet werden:

4.1: Mit welchen Methoden kann man ein Betriebssystem an die Anforderungen in einem konkreten Produkt so anpassen, dass alle Ressourcen möglichst effizient genutzt werden?

4.2: Welche Anforderungen muss ein Echtzeitbetriebssystem erfüllen? Wie unterscheiden sie sich von den Anforderungen an ein Standard-Betriebssystem? Welche Eigenschaften eines Standard-Betriebssystems wie Windows oder Linux könnten vielleicht in einem Echtzeitbetriebssystem fehlen?

4.3: Wie viele Sekunden wurden in der Silvesternacht insgesamt hinzugefügt, um den Unterschied zwischen UTC und TAI seit 1958 zu kompensieren? Suchen Sie im Internet nach einer Lösung für diese Aufgabe!

4.4: Finden Sie Prozessoren, die über eine Speicherschutzeinheit (engl. *Memory Protection Unit* (MPU)) verfügen! Wie unterscheiden sich MPUs von den häufiger anzutreffenden *Memory Management Units* (MMUs, siehe Anhang C)? Suchen Sie im Internet nach einer Lösung für diese Aufgabe!

4.5: Beschreiben Sie Klassen von eingebetteten Systemen, für die Schutz auf jeden Fall verfügbar sein sollte! Beschreiben Sie auch Klassen von eingebetteten Systemen, für die Schutz möglicherweise nicht notwendig ist!

4.6: Entwickeln Sie ein Beispiel, das die Prioritätsumkehr für ein System mit drei Tasks demonstriert!

4.7: Laden Sie das levi-Modul leviRTS von der levi-Webseite [497]. Modellieren Sie eine Jobmenge wie in Tabelle 4.1 dargestellt.

Tabelle 4.1 Jobmenge mit exklusiven Ressourcenanforderungen

Job	Priorität	Ankunft	Laufzeit	Kamera		Mikrofon	
				$t_{P,P}$	$t_{V,P}$	$t_{P,C}$	$t_{V,C}$
J_1	1 (hoch)	3	4	1	4	-	-
J_2	2	10	3	-	-	1	2
J_3	3	5	6	-	-	4	6
J_4	4 (niedrig)	0	7	2	5	-	-

$t_{P,P}$ und $t_{P,C}$ sind die Zeitpunkte relativ zur Startzeit, zu denen ein Job die exklusive Benutzung der Kamera bzw. des Mikrofons anfordert (ΔtP in levi). $t_{V,P}$ und $t_{V,C}$ sind die Zeitpunkte relativ zur Startzeit, zu denen diese Ressourcen wieder freigegeben werden. Verwenden Sie prioritätsbasiertes, präemptives Scheduling! Welches Problem tritt hier auf? Wie kann es gelöst werden?

4.8: Welche Protokolle zum exklusiven Zugriff auf Ressourcen verhindern *Deadlocks*?

4.9: Wie wird in ERIKA die Benutzung des *Stacks* optimiert?

4.10: Welche Probleme müssen gelöst werden, wenn Linux als Betriebssystem für eingebettete Systeme benutzt wird?

4.11: Welchen Einfluss hat das Prioritätsumkehr-Problem auf den Entwurf von Netzwerk-*Middleware*?

4.12: Welchen Einfluss könnte *Flash*-Speicher auf den Entwurf von Echtzeitdatenbanken haben?

Open Access Dieses Kapitel wird unter der Creative Commons Namensnennung 4.0 International Lizenz (http://creativecommons.org/licenses/by/4.0/deed.de) veröffentlicht, welche die Nutzung, Vervielfältigung, Bearbeitung, Verbreitung und Wiedergabe in jeglichem Medium und Format erlaubt, sofern Sie den/die ursprünglichen Autor(en) und die Quelle ordnungsgemäß nennen, einen Link zur Creative Commons Lizenz beifügen und angeben, ob Änderungen vorgenommen wurden.

Die in diesem Kapitel enthaltenen Bilder und sonstiges Drittmaterial unterliegen ebenfalls der genannten Creative Commons Lizenz, sofern sich aus der Abbildungslegende nichts anderes ergibt. Sofern das betreffende Material nicht unter der genannten Creative Commons Lizenz steht und die betreffende Handlung nicht nach gesetzlichen Vorschriften erlaubt ist, ist für die oben aufgeführten Weiterverwendungen des Materials die Einwilligung des jeweiligen Rechteinhabers einzuholen.

Kapitel 5
Bewertung und Validierung

Während des Entwurfs müssen wir uns immer wieder davon überzeugen, dass das geplante System voraussichtlich seine Funktion erfüllen wird und gemäß aller relevanten Bewertungskriterien die Anforderungen einhalten wird. Diesem Zweck dienen die Validierung und die Bewertung von Zwischenzuständen im Entwurfsprozess. In den Abschnitten 5.1.2 bis 5.6 geben wir einen Überblick über wichtige Bewertungstechniken. Dabei werden Bewertungstechniken für eine Anzahl von Kriterien vorgestellt, insbesondere für die Kriterien der Ausführungszeit, der Ergebnisqualität, des Energieverbrauchs, des thermischen Verhaltens und der Verlässlichkeit. Wir betrachten in diesem Rahmen ausführlich Basistechniken zur Bestimmung der größtmöglichen Ausführungszeit. Wir empfehlen, auf der Basis der vorgestellten Modelle des Energieverbrauchs jeweils ein dem vorliegenden Problem angepasstes Energiemodell zu entwickeln. Das Problem der thermischen Modellierung führen wir auf das äquivalente Problem der Modellierung von elektrischen Schaltkreisen zurück. Zur Berechnung der Verlässlichkeit stellen wir Basistechniken der statistischen Analyse der Zuverlässigkeit, die Fehlerbaumtechnik sowie die Fehlermöglichkeits- und Einflussanalyse vor. Zum Vergleich der Bewertungen gemäß verschiedener Kriterien stellen wir das Konzept der Pareto-Optimalität vor. Beginnend mit Abschnitt 5.7 werden wir einen kurzen Überblick über Validierungstechniken (wie die Simulation, das *Rapid Prototyping* und die formale Verifikation) geben.

5.1 Einleitung

5.1.1 Begriffe

Spezifikationen, Hardwareplattformen und Systemsoftware stellen die grundlegenden Zutaten zur Verfügung, die für den Entwurf eingebetteter Systeme benötigt werden. Während des Entwurfsprozesses müssen Entwürfe **bewertet** (oder „**evaluiert**") und **validiert** werden. Diese Entwurfsschritte können wie folgt definiert werden:

© Der/die Autor(en) 2021

P. Marwedel, *Eingebettete Systeme*, https://doi.org/10.1007/978-3-658-33437-6_5

Definition 5.1: Bewertung oder **Evaluation** ist der Vorgang, quantitative Informationen einiger wichtiger Kriterien (oder „Ziele") eines bestimmten (möglicherweise partiellen) Entwurfs zu berechnen.

Definition 5.2: Validierung ist die Überprüfung, ob ein bestimmter (Teil-)Entwurf für seinen Zweck geeignet ist, alle Bedingungen erfüllt und die geforderte Verarbeitungsleistung erbringt.

Definition 5.3: Validierung mit mathematischer Genauigkeit wird als **(formale) Verifikation** bezeichnet.

Bewertung und Validierung sind in verschiedenen Phasen des Entwurfsvorgangs erforderlich (siehe Abb. 5.1). Diese Aktionen und der Entwurf sollten verzahnt sein und nicht als voneinander unabhängige Aktivitäten betrachtet werden. Auch wenn

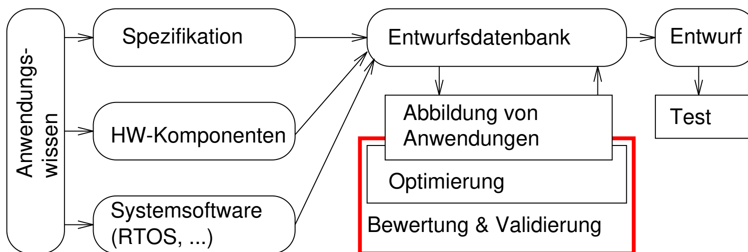


Abb. 5.1 Kontext dieses Kapitels

Bewertung und Validierung deutliche Unterschiede aufweisen, so sind sie dennoch eng miteinander verwandt. Der großen Bedeutung einer wiederholten Bewertung und Validierung wegen werden wir diese Vorgänge vor den eigentlichen Entwurfsschritten beschreiben.

5.1.2 Multikriterielle Optimierung

Bewertungen von Entwürfen werden im Allgemeinen zu einer Charakterisierung des Entwurfs anhand mehrerer Kriterien führen, wie Ausführungszeit, Energieverbrauch, Ergebnisqualität, thermisches Verhalten und Verlässlichkeit. Es ist meist nicht ratsam, alle diese Kriterien in einer einzelnen Zielfunktion (z.B. durch Verwendung eines gewichteten Mittelwerts) zusammenzufassen, da dies einige der wichtigen Eigenschaften des Entwurfs unterdrücken könnte. Es ist vielmehr ratsam, dem Entwickler eine Anzahl an Entwürfen vorzuschlagen, unter denen dieser den passenden Entwurf auswählen kann. Jedoch sollte diese Menge bereits ausschließlich „vernünftige" Entwürfe beinhalten. Das Finden solcher Mengen von Entwürfen ist der Zweck von **multikriteriellen Optimierungstechniken**.

Um multikriterielle Optimierungen durchzuführen, betrachten wir einen m-dimensionalen Raum X möglicher Lösungen des Optimierungsproblems. Diese Dimensionen könnten beispielsweise die Anzahl der Prozessoren, die Größen von Speichern und die Art und Anzahl von Bussen darstellen. Auf diesem Raum X definieren wir eine n-dimensionale Funktion, die Entwürfe in Hinblick auf mehrere Kriterien oder Ziele (z.B. Kosten und Leistung) hin evaluiert:

$$f(x) = (f_1(x), \ldots, f_n(x)) \text{ mit } x \in X$$

Sei F der n-dimensionale Werteraum dieser Ziele (der sogenannte **Zielraum**). Für jedes der Ziele ist eine Ordnung $<$ und die entsprechende $\leq$-Ordnung definiert. Im Folgenden nehmen wir an, dass wir die **Minimierung** der Ziele erreichen wollen.

Definition 5.4: Ein Vektor $u = (u_1, \ldots, u_n) \in F$ **dominiert** einen Vektor $v = (v_1, \ldots, v_n) \in F$ genau dann, wenn u in Hinblick auf mindestens ein Ziel „besser" als v und nicht schlechter als v für alle anderen Ziele ist:

$$\forall\, i \in \{1, \ldots n\} : u_i \leq v_i \quad \wedge \tag{5.1}$$

$$\exists\, j \in \{1, .., n\} : u_j < v_j \tag{5.2}$$

Definition 5.5: Ein Vektor $u \in F$ wird **indifferent** zu einem Vektor $v \in F$ genannt genau dann, wenn weder der Vektor u den Vektor v dominiert noch der Vektor v den Vektor u.

Definition 5.6: Ein Entwurf $x \in X$ heißt **Pareto-optimal** auf X genau dann, wenn es keinen Entwurf $y \in X$ gibt, so dass $u = f(x)$ von $v = f(y)$ dominiert wird.

Die vorstehende Definition definiert Pareto-Optimalität im Lösungsraum. Die folgende Definition liefert die Entsprechung für den Zielraum.

Definition 5.7: Sei $S \subseteq F$ eine Teilmenge von Vektoren im Zielraum. $v \in F$ ist eine **nicht dominierte Lösung** von S genau dann, wenn v von keinem Element $\in S$ dominiert wird. v heißt Pareto-optimal genau dann, wenn v nicht dominiert wird in Hinblick auf alle Lösungen F.

Abb. 5.2 verdeutlicht die unterschiedlichen Gebiete in einem Zielraum mit den Optimierungskriterien $O1$ und $O2$ relativ zum Entwurfspunkt (1). Abb. 5.2 (links) zeigt Pareto-Punkte und die obere rechte Fläche beschreibt Entwürfe, die von Entwurf (1) dominiert werden, da diese „schlechter" in Hinblick auf beide Ziele sind. Entwürfe im linken unteren Rechteck (wenn sie existieren würden) würden den Entwurf (1) dominieren, da sie in Hinlick auf beide Ziele „besser" wären. Entwürfe in der linken oberen und rechten unteren Ecke sind indifferent: sie sind „besser" in Hinblick auf ein Ziel und „schlechter" in Hinblick auf das andere. Abb. 5.2 (rechts) zeigt eine Menge von Pareto-Punkten, mit + markiert. Dominierte Entwurfspunkte sind alle Punkte in diesem Diagramm, die von **mindestens einem** Pareto-Punkt dominiert sind. Sie ergeben sich also als Vereinigung aller von einem Pareto-Punkt dominierten Punkte. Die **Pareto-Front** grenzt diese Punkte vom übrigen Teil des Diagramms ab. Die Pareto-Front wird durch eine Treppenfunktion dargestellt.

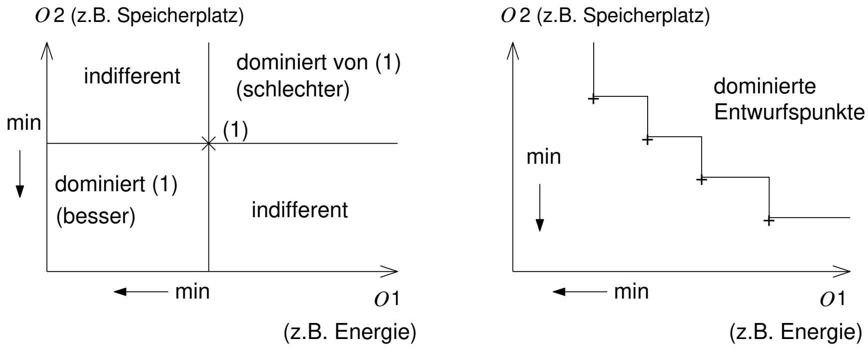


Abb. 5.2 Pareto-Optimalität: **links**: Pareto-Punkt; **rechts**: Pareto-Front

Definition 5.8: Unter Entwurfsraumexploration auf der Basis der Pareto-Optimalität verstehen wir den Vorgang, Pareto-optimale Lösungen zu finden und dem Entwerfer zu präsentieren, sodass dieser die am besten geeignete Lösung auswählen kann.

Zur Visualisierung der Bewertung anhand von Metriken in vielen Dimensionen können sogenannte Kiviat-, Spinnennetz- oder Radardiagramme benutzt werden [577]. Sie sind Erweiterungen der Abbildung 2.74 auf mehrere Dimensionen.

Beispiel 5.1: Wir können mehrere Entwürfe (z.B. für ein Mobiltelefon) ähnlich der nachfolgend beschriebenen Kriterien zu vergleichen (siehe Abb. 5.3).

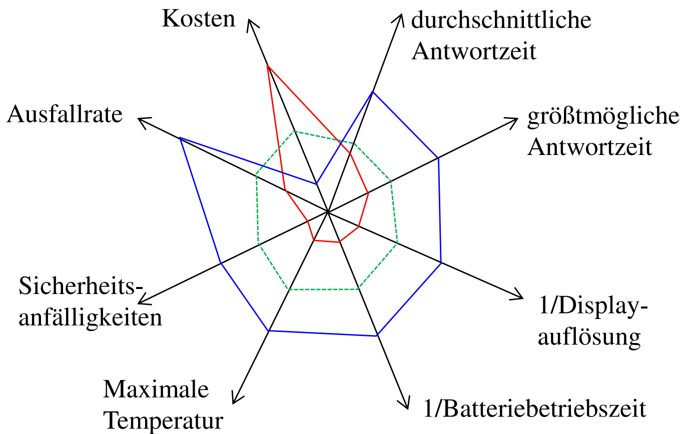


Abb. 5.3 Kiviat-Diagramm für Spitzenmodell (rot), Mittelklasse (grün, gestrichelt) und Einstiegsmodell (blau)

Einheitlich wird eine Minimierung angestrebt. Das Spitzenmodell erreicht gute Werte (außer bei den Kosten). Beim Einstiegsmodell ist es umgekehrt. ∇

5.1.3 Relevante Kriterien

Für Server und PC-ähnliche Systeme spielt die erwartete durchschnittliche Verarbeitungsleistung die wichtigste Rolle beim Entwurf eines neuen Systems. Für eingebettete und cyber-physikalische Systeme müssen mehrere Kriterien berücksichtigt werden. Die folgende Liste erläutert, ob und wo ein bestimmtes Zielkriterium in diesem Buch diskutiert wird:

1. **Durchschnittliche Verarbeitungsleistung (Performanz):** Information zu diesem Kriterium gibt es im Abschnitt 5.2. Eine Bewertung nach diesem Kriterium erfolgt häufig auf der Basis von Simulationen, die im Abschnitt 5.7 behandelt werden.
2. **Größtmögliche Laufzeit/Echtzeitverhalten:** Fundamentale Techniken zur Berechnung der größtmöglichen Laufzeit (engl. *Worst Case Execution Time* (WCET)) werden in Unterabschnitt 5.2.2 vorgestellt. Zusätzlich gibt es eine Einführung in den sogenannten *Real-Time Calculus* (RTC) im Unterabschnitt 5.2.3.
3. **Qualitätsmetriken:** Metriken zur Bewertung der Qualität (der Ausgabe) von eingebetteten Systemen werden im Abschnitt 5.3 präsentiert. Qualitätsmetriken spielen auch eine Rolle bei der Bewertung der Transformation zwischen Zahlendarstellungen, wie sie im Unterabschnitt 7.1.5 gezeigt werden.
4. **Energieverbrauch/elektrische Leistungsaufnahme:** Ein Überblick über Techniken, um nach diesem Kriterium zu bewerten, wird im Abschnitt 5.4 gegeben.
5. **Thermisches Verhalten:** Eine Einführung zu diesem Kriterium gibt es im Abschnitt 5.5.
6. **Verlässlichkeit:** Verlässlichkeit (engl. *dependability*) ist das Thema des Abschnitts 5.6, mit Unterabschnitten zur Informations- und zur Betriebssicherheit.
7. **Elektromagnetische Verträglichkeit:** Dieses Kriterium wird in diesem Buch nicht behandelt.
8. **Testbarkeit:** Die Kosten, die zum Testen eines Systems aufgewendet werden müssen, können sehr hoch sein. In Einzelfällen übersteigen sie sogar die Produktionskosten. Daher sollte auch die Testbarkeit berücksichtigt werden, vorzugsweise bereits während des Entwurfs. Testbarkeit wird im Kapitel 8 behandelt..
9. **Kosten:** Kosten in Form von Chipfläche oder Geld werden in diesem Buch nicht weiter betrachtet.
10. **Gewicht, Robustheit, Verwendbarkeit, Erweiterbarkeit, Umweltfreundlichkeit, rechtliche Fragen:** Auch diese Kriterien werden hier nicht weiter betrachtet.

Es gibt auch noch weitere Bewertungskriterien. Beispielsweise können wir gemäß der Qualitätsstandards ISO/IEC 25022 [259], ISO/IEC 25023 [260] und ISO/EIC 25024 [258] bewerten. Wir starten mit einer Betrachtung des Kriteriums „Verarbeitungsleistung” (Performanz). Dabei wird der Schwerpunkt auf die Performanz im schlimmstmöglichen Fall gelegt.

5.2 Performanzbewertung

Die Bewertung der Performanz hat zum Ziel, die Rechenleistung von Systemen vorherzusagen. Dies ist eine große Herausforderung (insbesondere für cyber-physikalische Systeme), da wir Informationen über den *worst case* anstelle von Informationen über den durchschnittlichen Fall benötigen könnten. Diese Informationen sind notwendig, um die Einhaltung von Echtzeitbedingungen garantieren zu können.

5.2.1 Frühe Phasen

Zwei unterschiedliche Arten von Techniken wurden vorgeschlagen, um bereits in frühen Entwurfsphasen Bewertungen der Performanz zu erhalten:

- **Geschätzte Kosten- und Performanzwerte:** Eine ganze Reihe von Schätzern wurde für diesen Zweck entwickelt. Beispiele sind Arbeiten von Jha and Dutt [275] für Hardware und von Jain et al. [267] und Franke [167] für Software. Die Erzeugung ausreichend genauer Abschätzungen erfordert einen beträchtlichen Aufwand.
- **Genaue Kosten- und Performanzwerte:** Wir können auch echten Software-Code (in Form einer Binärdatei) auf einer Hardwareplattform, die mit der zu entwickelnden eng verwandt ist, verwenden. Dies erfordert die aufwändige Bereitstellung eines Modells der Hardwareplattform wie auch eine rasche Abbildung der Software auf diese. Dies ist nur möglich, wenn Schnittstellen zu „Software-Synthesewerkzeugen" (Compiler) und Hardware-Synthesewerkzeugen existieren. Diese Methode kann präzisere Ergebnisse als die Abschätzung liefern, kann dabei aber auch erheblich mehr (und manchmal untragbar viel) Zeit in Anspruch nehmen.

Um ausreichend präzise Informationen zu erhalten, muss auch die Kommunikation betrachtet werden. Unglücklicherweise ist es meist schwierig, die Kommunikationskosten schon in frühen Entwurfsphasen zu berechnen.

Formale Techniken zur Performanzbewertung wurden bereits von vielen Forschern veröffentlicht. Für eingebettete Systeme sind die Arbeiten von Thiele et al., Henia und Ernst et al. und Wilhelm et al. besonders relevant (siehe z.B. [536, 211, 587]). Diese Techniken setzen einen bestimmten Grad von Wissen über Architekturen voraus. Sie sind für frühe Entwurfsphasen weniger wichtig, einige von ihnen lassen sich aber auch anwenden, ohne alle Details der Zielarchitektur zu kennen. Alle diese Ansätze modellieren echte, physikalische Zeit.

5.2.2 *Größtmögliche Ausführungszeiten*

Die Ablaufplanung oder das *Scheduling* von Tasks erfordert Wissen über deren Ausführungsdauer, insbesondere wenn das Einhalten von Zeitbedingungen garantiert werden muss, wie etwa in Echtzeitsystemen. Die größtmögliche Ausführungszeit (engl. *Worst Case Execution Time* (WCET)) ist die Ausgangsbasis der meisten *Scheduling*-Algorithmen. Einige im Zusammenhang mit der größtmöglichen Ausführungszeit wichtige Definitionen sind in Abb. 5.4 und der nachfolgenden Definition aufgeführt.

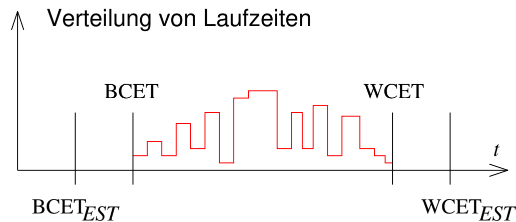


Abb. 5.4 Begriffe im Zusammenhang mit der größtmöglichen Ausführungszeit

Definition 5.9: Die **größtmögliche Ausführungszeit** (engl. *Worst Case Execution Time* (WCET)) ist das Maximum über alle Ausführungszeiten, die ein Programm benötigen kann, wenn beliebige Eingaben und ein beliebiger Anfangszustand möglich sind.

Leider ist die WCET nur sehr schwer berechenbar. Allgemein ist es sogar unentscheidbar, ob die WCET endlich ist oder nicht. Dies ist eine direkte Folge davon, dass es nicht entscheidbar ist, ob ein Programm terminiert oder nicht. Daher lässt sich die WCET nur für bestimmte Programme oder Tasks berechnen. Beispielsweise kann die WCET für Programme berechnet werden, die weder Rekursion noch Schleifen mit einer unbekannten Anzahl an Wiederholungen enthalten. Doch auch die Anwendung solcher Einschränkungen macht es meist in der Praxis unmöglich, die WCET zu berechnen. Die Auswirkungen der Fließbänder (engl. *pipelines*) von modernen Prozessorarchitekturen, die verschiedene Arten von *Hazards* (Risiken) erzeugen können und Speicherhierarchien mit begrenzter Vorhersagbarkeit von Trefferraten sind nur schwer während des Entwurfs vorhersagbar. Die Berechnung der WCET für Systeme mit Caches, Verdrängungen von Prozessen, Unterbrechungen und virtuellem Speicher ist eine noch größere Herausforderung. Daher müssen wir uns damit zufrieden geben, eine gute **obere Schranke** für die WCET angeben zu können.

Solche oberen Schranken werden als **geschätzte WCETs** (engl. *Estimated WCETs*) oder WCET$_{EST}$ bezeichnet. Diese Schranken sollten zumindest diese beiden Eigenschaften besitzen:

1. die Schranken sollten sicher sein (WCET$_{EST}$ $\geq$ WCET) und
2. die Schranken sollten gut sein (WCET$_{EST}$-WCET $\ll$ WCET).

Trotz der Verwendung des Begriffs „geschätzte" handelt es sich um sichere Schranken.

In einigen Fällen werden Architektureigenschaften, welche die durchschnittliche Ausführungszeit senken, aber nicht garantieren können, dass auch die WCET sinkt, in Echtzeitsystemen nicht weiter berücksichtigt (siehe Seite 169). Die Berechnung guter oberer Grenzen für die Ausführungszeit kann aber dennoch schwierig sein. Die oben beschriebenen Architektureigenschaften stellen auch ein Problem für die Berechnung der WCET_{EST} dar. Für Mehrkern-Systeme ist die Berechnung guter oberer Schranken noch schwieriger als für einzelne Kerne, da die zeitlichen Beeinflussungen häufig schwierig zu modellieren sind. Tatsächlich kann es mögliche Ressourcenkonflikte geben, die zur Folge haben, dass Mehrkern-Systeme größere Schranken haben als Einkern-Systeme.

Definition 5.10: Die kleinstmögliche Ausführungszeit (engl. *Best Case Execution Time* (BCET)) eines Programms ist die kürzeste Ausführungszeit, die unter allen möglichen Eingaben und Anfangszuständen möglich ist. BCET_{EST} ist eine sichere und gute untere Schranke für die Ausführungszeit.

Die Berechnung guter Grenzen für ein in einer Hochsprache wie C geschriebenes Programm ist ohne Kenntnis des erzeugten Assemblercodes und der verwendeten Architektur nicht möglich. Daher muss eine sichere Analyse den Maschinencode betrachten. Alle anderen Ansätze würden zu unsicheren Ergebnissen führen.

Im Folgenden befassen wir uns im Detail mit der Abschätzung der WCET. Diese Darstellung basiert auf der Beschreibung des Werkzeugs aiT von R. Wilhelm [587]. Die Architektur von aiT ist in Abb. 5.5 dargestellt. Unseren Erkenntnissen

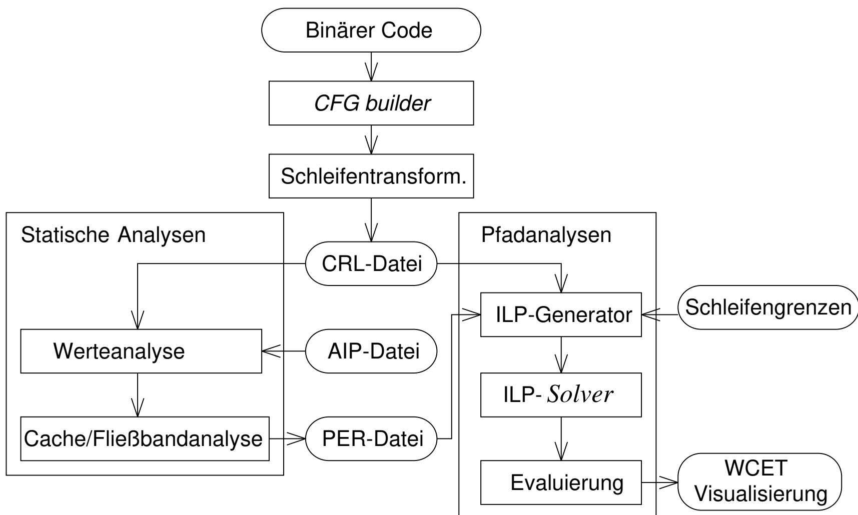


Abb. 5.5 Architektur des Timinganalyse-Werkzeugs aiT

zu Problemen der Analysierbarkeit von Hochsprachen-Code folgend, führt aiT die Analyse auf einer ausführbaren Binärdatei des zu analysierenden Codes durch. Aus diesem Code wird ein Kontrollflussgraph (engl. *Control Flow Graph* (CFG)) erzeugt. Daraufhin werden Schleifentransformationen angewandt. Diese umfassen Transformationen zwischen Scheifen und rekursiven Funktionsaufrufen sowie das virtuelle „Abrollen" von Schleifen (engl. *loop unrolling*). Dieses Abrollen ist „virtuell", da es nur intern stattfindet, ohne tatsächlich den ausführbaren Code zu verändern. Die Ergebnisse werden im CRL-Format (engl. *Control flow Representation Language*) dargestellt. Die nächste Phase setzt nun statische Analysen ein. Statische Analysen lesen eine AIP-Datei mit Annotationen (Anmerkungen) des Entwicklers ein. Diese Annotationen beschreiben schwer oder unmöglich automatisch aus der Programmstruktur zu ersehende Information (z.B. Schranken für komplexe Schleifen). Die statischen Analysen bestehen aus Werte-, Cache- und Fließbandanalysen.

Eine **Werteanalyse** berechnet maximale Intervalle für Werte in Registern und lokalen Variablen. Diese Angaben können für die Kontrollflussanalyse und die Datencacheanalyse verwendet werden. Häufig sind Werte wie Adressen genau bekannt (besonders für „sauberen" Code). Dies erleichtert die Vorhersage von Speicherzugriffen sehr.

Die nächsten Schritte sind die **Cache-** und die **Fließbandanalyse**. Nachfolgend beschreiben wir einige Details der Cacheanalyse. Wir gehen von einem n-fach mengenassoziativen Cache aus (siehe Abb. 5.6)[1]. Betrachten wir nun einen Teil (eine

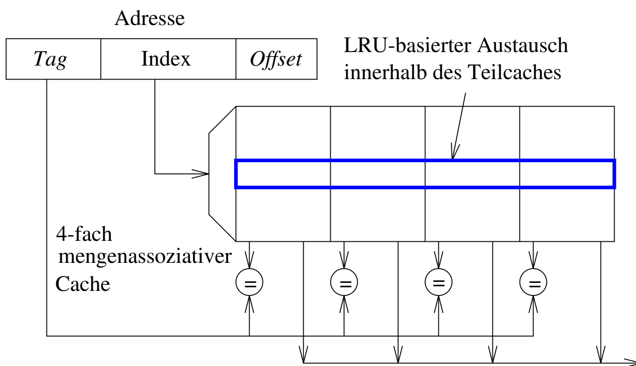


Abb. 5.6 n-fach mengenassoziativer Cache mit n=4

Zeile) des Caches mit einem bestimmten Index (in Abb. 5.6 fett und blau dargestellt). Wir nehmen an, dass die Verdrängung aus dem Cache mittels der *Least Recently Used* (LRU)-Strategie erfolgt. Damit werden für alle Zugriffe auf einen bestimmten Index die n Speicherblöcke, auf die zuletzt zugegriffen wurde, in diesem Teil des Caches gespeichert. Dabei soll die erforderliche LRU-Verwaltungshardware für

[1] Wir setzen voraus, dass der Leser mit dem Konzept von Caches vertraut ist.

jeden Index verfügbar sein, wobei alle Indizes unabhängig voneinander behandelt werden. Unter dieser Annahme sind alle Verdrängungen für einen bestimmten Index vollständig unabhängig von den Entscheidungen für andere Indizes. Diese Unabhängigkeit ist sehr wichtig, da sie es uns ermöglicht, jeden der Indizes für sich alleine zu betrachten.

Dementsprechend sehen wir uns nun das Verhalten des Caches für einen bestimmten Index und die dazu gehörige Cachezeile an. Was passiert jetzt bei einem Zugriff auf diesen Index? Als erstes betrachten wir den Fall des Zugriffs auf eine Variable e, die sich im Cache befindet. Nach dem Zugriff ist diese Variable die jüngste (siehe Abb. 5.7). In der Abbildung sind die Einträge links stets jünger als solche rechts.

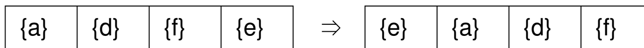


Abb. 5.7 Zugriff auf Variable e macht sie zur jüngsten

Im zweiten Fall gehen wir davon aus, dass wir einen Zugriff auf eine Variable c haben, die sich noch nicht im Cache befindet. In diesem Fall wird die älteste Variable verdrängt (siehe Abb. 5.8).

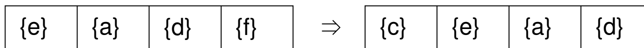


Abb. 5.8 Zugriff auf Variable c verdrängt f

Als nächstes beschäftigen wir uns mit rekonvergenten Programmpfaden, wie sie nach der Bearbeitung von bedingten Anweisungen auftreten. Welche Informationen haben wir über den Inhalt des Cacheteils nach einer solchen Vereinigung?

Wir müssen zwischen der ***may***-Analyse und der ***must***-Analyse unterscheiden. Die ***must***-Analyse gibt Auskunft darüber, welche Informationen sich sicher im Cache befinden. Sie ist geeignet, Zusicherungen bei der Bestimmung von WCET-Schranken zu geben. Die ***may***-Analyse resultiert in Aussagen darüber, welche Informationen sich möglicherweise im Cache befinden könnten. Diese Aussagen sind wichtig, um zu wissen, was sich mit Sicherheit **nicht** im Cache befindet. Mit diesem Wissen können wir BCET-Schranken bestimmen.

Betrachten wir zunächst die *must*-Analyse für rekonvergente Programmpfade. Abbildung 5.9 zeigt eine entsprechende Situation. Das Alter der Einträge wachse innerhalb eines Rechtecks von links nach rechts. Das Speicherobjekt c in Abb. 5.9 sei das jüngste Element in der Cachezeile, sofern wir über einen Programmpfad zur Rekonvergenz gelangen und a sei das jüngste Element, sofern wir über den anderen Pfad dorthin gelangen. Entsprechendes gilt für die älteren Einträge im Cache. Wir wollen jetzt im Kontext der *must*-Analyse bestimmen, was der „schlechteste“ Fall

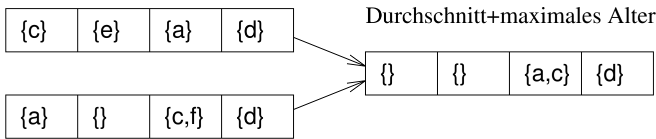


Abb. 5.9 *Must*-Analyse für LRU-Caches bei rekonvergenten Pfaden

nach der Rekonvergenz ist. Offensichtlich können wir nach der Rekonvergenz im Cache mit Sicherheit nur solche Speicherobjekte finden, die sich in der Schnittmenge der beiden ursprünglichen Cacheinhalte befanden. Als Alter müssen wir im Sinne der *worst case*-Analyse das maximale Alter annehmen. Abb. 5.9 zeigt das Ergebnis. Offensichtlich muss die Analyse für jeden Platz (jede Spalte) im Cache mit Mengen möglicher Einträge arbeiten.

Betrachten wir nun die *may*-Analyse für rekonvergente Programmpfade. Abb. 5.10 zeigt wiederum die Situation. Im resultierenden Cache können nunmehr offen-

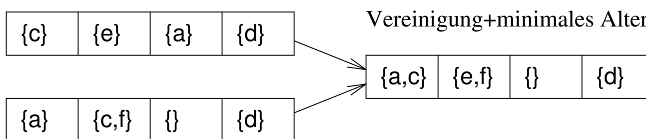


Abb. 5.10 *May*-Analyse für LRU-Caches bei rekonvergenten Pfaden

sichtlich Speicherobjekte vorhanden sein, die vor der Rekonvergenz in einem der beiden Programmpfade vorhanden waren. Also müssen wir die **Vereinigung** der Speicherobjekte betrachten. Im bestmöglichen Fall müssen wir von dem **jüngsten Alter** der Speicherobjekte ausgehen. Abb. 5.10 zeigt das Ergebnis.

Zu den statischen Analysen zählen auch Fließbandanalysen. Eine Fließbandanalyse muss sichere Schranken für die Anzahl an Zyklen berechnen, die für die Ausführung des Maschinencodes, der sich im Fließband des Prozessors befindet, benötigt wird. Details der Fließbandanalyse werden von Hahn et al. [197] sowie S. Thesing [534] beschrieben. Das Endergebnis statischer Analysen enthält Schranken für die Ausführungszeiten für jeden Basisblock eines Programms. Die Ergebnisse werden in einer PER-Datei abgelegt, wie in Abb. 5.5 zu sehen.

Die folgende Phase von aiT verwendet diese Schranken, um größtmögliche Ausführungszeiten für das gesamte Programm zu ermitteln. Dieser Schritt basiert auf einem Modell der **Ganzzahligen Linearen Programmierung** (engl. *integer linear programming* (ILP), siehe Anhang A). Dementsprechend enthält das Modell der Ausführungszeiten zwei Typen von Information:

- **Die Zielfunktion**: in unserer Anwendung der ILP-Modellierung nutzen wir die Gesamt-Ausführungszeit als Zielfunktion, die zu maximieren ist. Diese Zeit wird als Summe der Ausführungszeiten der Basisblöcke bestimmt:

$$WCET_{EST} = \sum_{\text{Basisblöcke}} e_i * f_i \tag{5.3}$$

  wobei e_i die größtmögliche Ausführungszeit von Basisblock i ist (die in der statischen Analyse berechnet wird) und f_i ist dessen Ausführungshäufigkeit. Die Ausführungshäufigkeiten können möglicherweise nicht alle vollständig automatisch bestimmt werden. Daher werden Zusatzinformationen des Entwerfers benötigt, wie z.B. Schleifengrenzen.
- **Lineare Randbedingungen**: in unserer Anwendung der ILP-Modellierung nutzen wir lineare Randbedingungen, um die Struktur des Datenflussgraphen zu repräsentieren.

Beispiel 5.2: Nachfolgend betrachten wir ein einfaches Beispiel:

```
int main() {
   int i,j=0;
   _Pragma("loopbound min 100 max 100")                /* Hinweis an aiT */
   for (i=0; i <100; i++) {
     if (i<50) j+=i;
     else j+=(i*13) % 42;
   }
   return j;
}
```

Abb. 5.11 (links) zeigt einen Kontrollflussgraphen für dieses kleine Programm. Der Kontrollfluss wurde durch start- und exit-Knoten erweitert.

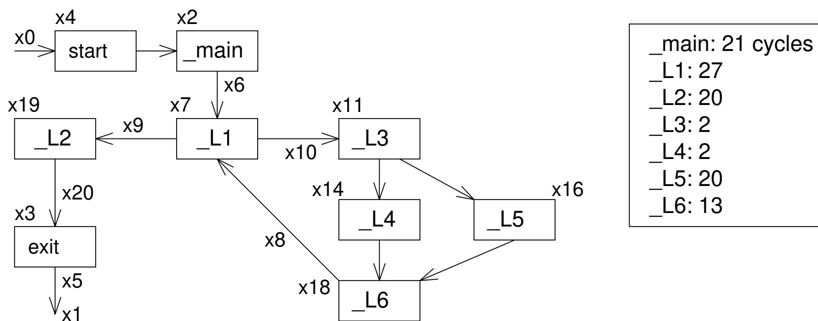


Abb. 5.11 Beispielprogramm: **links**: Kontrollflussgraph, **rechts**: WCET_{EST} der Basisblöcke

Knoten _L1 entspricht dem Test der `for`-Schleife, _L3 dem `if`-Test, _L4 und _L5 den zwei Fällen des `if`-Statements und _L6 der Rekonvergenz. Die Variablen x0

bis x20 bezeichnen die Ausführungshäufigkeiten der Blöcke und die Anzahl der Übergänge zwischen den Blöcken. Beispielsweise haben wir x6 Übergänge von Block main zum Block _L1 und führen den Zielblock x7-mal aus. Wir nehmen an, dass die Analyse der WCET für die Basisblöcke zu der Liste auf der rechten Seite von Abb. 5.11 geführt hat. Dies führt zu dem nachfolgenden Auszug aus der Liste der ILP-Randbedingungen.

```
01: 21 x2 + 27 x7 + 2 x11 + 2 x14 + 20 x16 + 13 x18 + 20 x19;/*Zielfunktion*/
02: x7 - x8 - x6 = 0;                  /* Randbedingung für Fluss nach _L1 */
03: x7 - x9 - x10 = 0;                  /* Randbedingung für Fluss von _L1 */
04: x7 - 101 x9 >= 0; /* Randbedingung für untere Schleifengrenze von _L1 */
05: x7 - 101 x9 <= 0; /* Randbedingung für obere Schleifengrenze von  _L1 */
06: x0 - x4 = 0;                                /* CFG Start-Randbedingung */
07: x2 - x4 = 0;                     /* Randbedingung für Fluss nach _main */
08: x2 - x6 = 0;                      /* Randbedingung für Fluss von _main */
09: ...
```

Zeile 01 enthält die Zielfunktion. Alle anderen Zeilen enthalten Randbedingungen, welche die Struktur des Graphen modellieren. Beispielsweise sind die Randbedingungen für den Knoten _L1 in den Zeilen 02 und 03 angegeben. Die Häufigkeit des Verzweigens in diesen Knoten (x6+x8) ist gleich der Anzahl der Ausführungen (x7). Die Häufigkeit des Verlassens dieses Knotens (x9+x10) ist ebenfalls gleich der Anzahl der Ausführungen (x7). Die Zeilen 04 und 05 korrespondieren zu den Schleifeniterationen. Die Anzahl der Iterationen wurde dem *Pragma* im Code entnommen. Zeile 06 sagt aus, dass die Anzahl der Ausführungen des start-Knotens gleich der Anzahl der Verzweigungen in den Code ist. Die übrigen Zeilen beschreiben die Struktur in ähnlicher Weise. ∇

Das so definierte ILP-Problem kann mit einem Standard-ILP-*Solver*, der die Zielfunktion maximiert, gelöst werden. Das so berechnete Maximum ist eine sichere obere Schranke für die gesamte Ausführungszeit.

Diese Technik der Modellierung von Ausführungszeiten heißt *Implicit Path Enumeration Technique* (IPET) [344]. Sie vermeidet eine vollständige Aufzählung aller möglichen Ausführungspfade. Eine solche Aufzählung würde zu i.d.R. zu sehr vielen Pfaden führen.

In aiT ist auch eine Visualisierung der Ergebnisse in Form annotierter Kontrollflussgraphen verfügbar. Der Entwickler kann diese Graphen analysieren, um das zu entwerfende System zu optimieren. aiT besitzt eine Reihe von Einschränkungen, die sich aus dem beschriebenen Vorgehen erklären lassen: so werden keine Verdrängungen durch andere Prozesse, keine Hardware-Unterbrechungen, keine Ein/Ausgaben, keine direkten Speichertransfers (DMA) und keine Interferenzen durch andere Hardwarekomponenten betrachtet.

Für die WCET-Analyse von Mehrkern-Systemen existieren nur wenige Ansätze [265, 266, 287]. Mit neuen probabilistischen Methoden [2] sollen existierende Ansätze ergänzt werden. Sie basieren meist auf der Extremwerttheorie [196].

5.2.3 Realzeitkalkül

WCET-Berechnungen erlauben es uns, obere Schranken für eine einzelne Ausführung eines Programms zu bestimmen. Die resultierenden Werte reichen aber noch nicht aus, um zu garantieren, dass ein Strom von Ereignissen von einer Hardwareplattform mit ausreichender Performanz rechtzeitig verarbeitet wird. Dies kann aber z.B. für Teile des Internets der Dinge wichtig sein.

Eine Prüfung auf eine ausreichende Verarbeitungsleistung ist mit Thieles **Realzeitkalkül** (engl. *Real-Time Calculus* (RTC)) möglich. Dieser Kalkül basiert auf einer Beschreibung der Rate der eingehenden Ereignisse[2]. Diese Beschreibung umfasst auch Fluktuationen dieser Rate. Zu diesem Zweck werden charakteristische Eigenschaften eingehender Ströme von Ereignissen durch ein Tupel von Ankunftskurven (engl. *arrival curves*) dargestellt

$$\overline{\alpha}^{\,u}(\Delta), \overline{\alpha}^{\,l}(\Delta) \in \mathbb{R} \geq 0, \Delta \in \mathbb{R} \geq 0$$

$\overline{\alpha}^u(\Delta)$ und $\overline{\alpha}^l(\Delta)$ beschreiben jeweils die maximale bzw. die minimale Anzahl von Ereignissen, die in einem Intervall der Länge Δ eingehen. Es gibt also höchstens $\overline{\alpha}^u(\Delta)$ und mindestens $\overline{\alpha}^l(\Delta)$ eingehende Ereignisse in einem Intervall $(t, t + \Delta)$ für alle $t \geq 0$. Abb. 5.12 beschreibt die Zahl möglicher Ereignisankünfte für mögliche Modelle von Ereignisströmen. Bei periodischen Ereignisströmen mit einer Periode

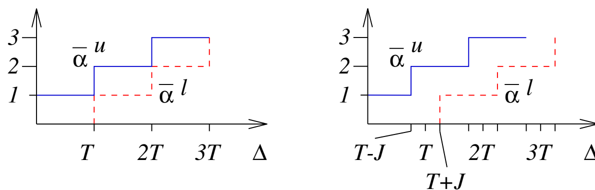


Abb. 5.12 Ankunftskurven: **links**: periodischer Strom; **rechts**: periodischer Strom mit Jitter J

T findet in einem Intervall $(0, T)$ maximal ein einzelnes Ereignis statt[3]. Entsprechend gibt es eine obere Grenze von höchstens zwei ankommenden Ereignissen im Zeitintervall $(T, 2T)$.

Betrachten wir nun die untere Schranke für das Zeitintervall $(0, T)$. Möglicherweise findet kein einziges Ereignis in diesem Intervall statt. Die untere Schranke ist also Null. Im Zeitintervall $(T, 2T)$ muss es mindestens ein Ereignis geben. Daher ist die Schranke ebenfalls eins. Für $\Delta = 0{,}5\,T$ gibt es also mindestens Null und maximal ein ankommendes Ereignis (siehe Abb. 5.12 (links)). Bei periodischen Ereignisströmen mit Jitter J werden die Kurven um diesen Betrag verschoben. Die obere Schranke ist

[2] Unsere Darstellung basiert auf Thieles Beitrag im Buch von Zurawski [536]. Entsprechende Betrachtungen auf Systemebene heißen *Modular Performance Analysis* (MPA).

[3] Wir vermeiden hier die subtile Diskussion der Unstetigkeiten für $\Delta = n * T$.

nach links verschoben und die untere Schranke nach rechts. Wir nehmen dabei an, dass der Jitter sich nicht im Laufe der Zeit aufsummiert. Wir können uns dazu vorstellen, dass die Taktfrequenz im Prinzip korrekt ist und dass nur einzelne Flanken zu früheren oder späteren Zeiten verschoben sind.

Wir benutzen Striche über den Symbolen (wie in $\overline{\alpha}$) für alle Größen, die Ereignisse (engl. *events*) bezeichnen.

Auf ähnliche Weise kann die zur Verfügung stehende Bearbeitungs- und Übertragungsleistung durch *service functions* $\beta^u(\delta)$ und $\beta^l(\delta)$ beschrieben werden mit

$$\beta^u(\Delta), \beta^l(\Delta) \in \mathbb{R} \geq 0, \Delta \in \mathbb{R} \geq 0$$

Mit dem Intervall $[\beta^l, \beta^u]$ wird modelliert, dass die Bearbeitungsleistung im Laufe der Zeit schwanken kann. Abb. 5.13 charakterisiert die Übertragungsleistung eines Busses, der nach dem *Time Division Multiple Access* (TDMA)-Prinzip (siehe Seite 193) immer nur innerhalb gewisser Zeitintervalle s für eine Übertragung genutzt werden kann. Die Zuteilung findet periodisch alle T Zeiteinheiten statt. Das

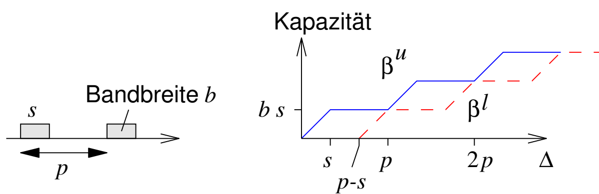


Abb. 5.13 Bearbeitungs- bzw. Übertragungsleistung (*service functions*) eines TDMA-Busses

Zeitfenster für die Zuteilung ist s Zeiteinheiten lang. Während dieses Zeitfensters erreicht der Bus eine Bandbreite von b. Die obere Grenze lässt sich ermitteln, wenn der Bus zu Beginn der Beobachtung zugeteilt wird. Der Umfang an übertragbaren Informationen steigt dann linear an. Die untere Grenze ergibt sich, wenn der Bus zu Beginn unserer Beobachtung der Länge Δ gerade freigegeben wurde. Dann müssen wir $T - s$ Zeiteinheiten warten, bis der Bus wieder zugeteilt wird.

Die Bestimmung der Funktionen $\overline{\alpha}$ und β ist nicht Teil des Realzeitkalküls, sondern sie muss mit separaten Methoden extern erfolgen. Allerdings können Schranken für innerhalb des Systems erzeugte Ereignisse aus dem Kalkül abgeleitet werden (siehe unten).

Bislang fehlt noch die Information, welche **Arbeitslast** (engl. *workload*) ein eintreffendes Ereignis erzeugt. Die Arbeitslast wird im Realzeitkalkül durch weitere Funktionen $\gamma^u(e), \gamma^l(e) \in \mathbb{R} \geq 0$ für jede Folge von e Ereignissen charakterisiert. Diese Information kann z.B. aus den Schranken für die Ausführungszeit eines Jobs bestimmt werden. Abb. 5.14 zeigt ein Beispiel für diese Funktionen. Dabei wurde angenommen, dass pro eingehendem Ereignis drei bis vier Rechenzeiteinheiten zur Bearbeitung erforderlich sind. Entsprechend schwankt der Bearbeitungsbedarf für

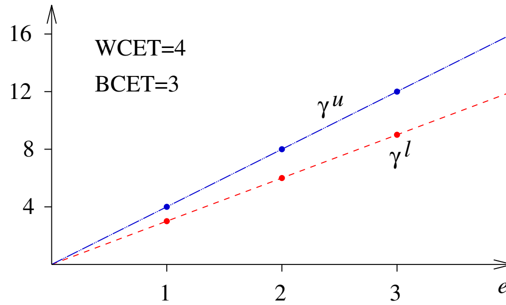


Abb. 5.14 Arbeitslast (WCET$_{EST}$ kann anstelle der WCET genutzt werden)

ein einzelnes Ereignis zwischen drei und vier Zeiteinheiten, der Bearbeitungsbedarf für zwei Ereignisse zwischen sechs und acht Zeiteinheiten usw. Die gestrichelten Linien sind kein Teil der Funktion, da diese nur an ganzzahligen Zeitpunkten definiert ist. Die Arbeitslast, die ein eingehender Strom von Ereignissen erfordert, kann nun leicht berechnet werden. Obere und untere Schranken werden beschrieben durch die Funktionen

$$\alpha^u(\Delta) = \gamma^u(\overline{\alpha}^u(\Delta)) \text{ und} \tag{5.4}$$

$$\alpha^l(\Delta) = \gamma^l(\overline{\alpha}^l(\Delta)) \tag{5.5}$$

Es sollte ausreichende Bearbeitungs- oder Kommunikationskapazität zur Verfügung stehen, um diese Arbeitslast zu bewältigen. Die Anzahl der Ereignisse, die mit der verfügbaren Bearbeitungskapazität verarbeitet werden kann, kann berechnet werden durch

$$\overline{\beta}^u(\Delta) = (\gamma^l)^{-1}(\beta^u(\Delta)) \text{ und} \tag{5.6}$$

$$\overline{\beta}^l(\Delta) = (\gamma^u)^{-1}(\beta^l(\Delta)) \tag{5.7}$$

Die Gleichungen (5.6) und (5.7) verwenden die Inverse der Funktionen γ^u und γ^l, um Schranken der verfügbaren Kapazität (gemessen in realen Zeiteinheiten) in Schranken zu wandeln, die durch die Anzahl abarbeitbarer Ereignisse beschrieben werden.

Mit dieser Information kann bestimmt werden, wie Echtzeitkomponenten einen eingehenden Ereignisstrom $[\overline{\alpha}^l, \overline{\alpha}^u]$ in einen ausgehenden Ereignisstrom $[\overline{\alpha}^{l'}, \overline{\alpha}^{u'}]$ transformieren. Ebenso kann bestimmt werden, welche Verarbeitungsleistung noch für andere Aufgaben bereitsteht. Diese verbleibende Verarbeitungsleistung ergibt sich durch die Transformation der *service curves* $[\overline{\beta}^l, \overline{\beta}^u]$ in *service curves* $[\overline{\beta}^{l'}, \overline{\beta}^{u'}]$ (siehe Abb. 5.15). Diese verbleibende Verarbeitungsleistung steht z.B. für Aufgaben zur Verfügung, die auf demselben Prozessor in Tasks mit niedrigerer Priorität gelöst werden.

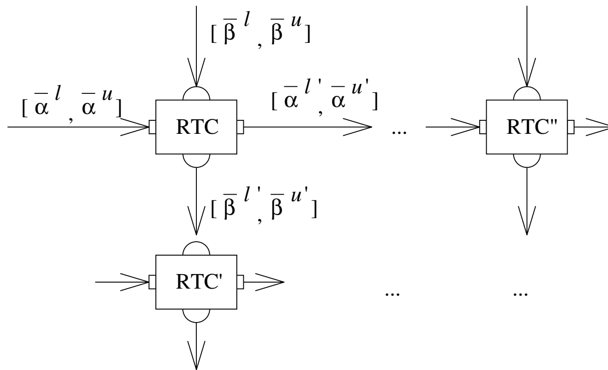


Abb. 5.15 Echtzeitkomponenten transformieren Ereignisströme und Bearbeitungskapazität

Thiele et al. geben an, wie die ausgehenden Ereignisströme und die verbleibende Bearbeitungsleistung berechnet werden können [536]:

$$\overline{\alpha}^{u'} = [(\overline{\alpha}^{u} \underline{\otimes} \overline{\beta}^{u}) \overline{\oslash} \overline{\beta}^{l}] \wedge \overline{\beta}^{u} \tag{5.8}$$

$$\overline{\alpha}^{l'} = [(\overline{\alpha}^{l} \overline{\oslash} \overline{\beta}^{u}) \underline{\otimes} \overline{\beta}^{l}] \wedge \overline{\beta}^{l} \tag{5.9}$$

$$\overline{\beta}^{u'} = (\overline{\beta}^{u} - \overline{\alpha}^{l}) \underline{\oslash} 0 \tag{5.10}$$

$$\overline{\beta}^{l'} = (\overline{\beta}^{l} - \overline{\alpha}^{u}) \overline{\otimes} 0 \tag{5.11}$$

Dabei sind die Operatoren definiert durch:

$$(f \underline{\otimes} g)(t) = inf_{0 \le u \le t}\{f(t-u) + g(u)\} \tag{5.12}$$

$$(f \overline{\otimes} g)(t) = sup_{0 \le u \le t}\{f(t-u) + g(u)\} \tag{5.13}$$

$$(f \overline{\oslash} g)(t) = sup_{u \ge 0}\{f(t+u) - g(u)\} \tag{5.14}$$

$$(f \underline{\oslash} g)(t) = inf_{u \ge 0}\{f(t+u) - g(u)\} \tag{5.15}$$

$\wedge$ bezeichnet den Minimum-Operator.

Im Wesentlichen beschreiben diese Funktionen ausgehende Ströme und Kapazitäten. Diese Gleichungen wurden aus der Kommunikationstheorie übernommen. Beweise zu diesen Gleichungen finden sich in Publikationen zum Netzwerkkalkül [328]. Die einfachste Art, diese Gleichungen zu verwenden, ist der Einsatz einer Matlab-*Toolbox* [560].

Diese Theorie ermöglicht auch die Berechnung der Verzögerung, die durch die Echtzeitkomponenten verursacht wird und der Puffergröße, die für die Zwischenspeicherung ein- und ausgehender Ereignisse benötigt wird. So lassen sich die Performanz und weitere Eigenschaften eines Systems aus Informationen über die einzelnen Komponenten berechnen.

Ein weiterer Ansatz zur Performanzanalyse wurde von Henia, Ernst et al. entwickelt. Der sogenannte SymTA/S-Ansatz [211] ersetzt die verschiedenen Kurven aus Thieles Ansatz durch Standard-Modelle von Ereignisströmen wie z.B. periodischen Ereignisströmen, periodischen Ereignisströmen mit Jitter und periodischen Ereignisströmen mit Bursts. SymTA/S unterstützt dabei insbesondere die Kombination und Integration verschiedener Analysetechniken aus dem Echtzeitbereich.

5.3 Qualitätsmetriken

5.3.1 Näherungsweises Rechnen

In manchen Anwendungen ergibt sich zur Berechnung des bestmöglichen Ergebnisses ein hoher Bedarf an Ressourcen (wie z.B. Rechenzeit, Energie, Speicher usw.). Dabei wird teilweise nicht das bestmögliche Ergebnis benötigt, da kleine Abweichungen davon nicht vom Benutzer bemerkt werden. Dies gilt beispielsweise für verlustbehaftete Audiokodierung (wie z.B. MP3) sowie verlustbehaftete Video- und Bildkodierung (wie z.B. JPEG). Dies kann in einer ressourcenbeschränkten Umgebung ausgenutzt werden, um zwischen der Qualität der Ausgabe und dem Ressourcenbedarf auszubalancieren. Dies führt uns zum **näherungsweisen Rechnen** (engl. *approximate computing*).

Definition 5.11: Unter dem **näherungsweisen Rechnen** versteht man Rechenmethoden, bei denen man eine gewisse Abweichung des Ergebnisses vom bestmöglichen Ergebnis toleriert [398].

Beim näherungsweisen Rechnen ist es notwendig, die Qualität der Ergebnisse als eines der Zielkriterien zu benutzen. Leider ist es nicht einfach, die Qualität von Ergebnissen zu bewerten und es können verschiedene Metriken zum Einsatz kommen.

5.3.2 Einfache Qualitätskriterien

Einige einfache Metriken können benutzt werden, wenn das wahre oder das bestmögliche Ergebnis bekannt sind. Gegeben seien n Folgenelemente $x_1, ..., x_n$ eines Signals x in diskreter Zeit. Weiterhin nehmen wir an, dass wir statt der echten oder der bestmöglichen Werte $x_1, ..., x_n$ näherungsweise Werte $y_1, ..., y_n$ berechnen oder messen.

Damit können wir das mittlere Fehlerquadrat (engl. *Mean-Squared Error* (MSE)) wie folgt definieren:

Definition 5.12: Das **mittlere Fehlerquadrat** ist definiert als

$$MSE(x,y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2 \tag{5.16}$$

Die Wurzel aus dem mittleren Fehlerquadrat (engl. *Root-Mean-Squared Error* (RMSE)) bildet die zweite Metrik:

Definition 5.13: Die Wurzel aus dem mittleren Fehlerquadrat (RMSE) ist definiert als

$$RMSE(x,y) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2} \tag{5.17}$$

RMSE besitzt dieselbe Dimension wie die Differenz zwischen dem echten und dem aktuellen Wert, sollte aber nicht mit dem mittleren Fehler verwechselt werden, der wie folgt definiert wird:

Definition 5.14: Der mittlere Fehler (engl. *Mean-Absolute Error* (MAE)) ist definiert durch

$$MAE(x,y) = \frac{1}{n}\sum_{i=1}^{n}|x_i - y_i| \tag{5.18}$$

Der MAE ist gleich dem RMSE, wenn die Abweichungen von y vom echten Wert x alle gleich sind. Ansonsten betont der RMSE große Abweichungen zwischen den beiden Werten (sogenannte Ausreißer).

Das Signal-zu-Rauschverhältnis (engl. *Signal-to-Noise Ratio* (SNR)) wurde bereits auf Seite 156 definiert. Als nächstes definieren wir das Spitzen-Signal-zu-Rauschverhältnis, welches dem SNR ähnlich ist. Sei x ein Signal, x_{max} dessen Maximum und y die verrauschte Approximation von x.

Definition 5.15: Das **Spitzen-Signal-zu-Rauschverhältnis** (engl. *Peak-Signal-to-Noise Ratio* (PSNR)) ist definiert durch

$$PSNR(x,y) = 10\log_{10}\left(\frac{x_{max}^2}{MSE(x,y)}\right) \tag{5.19}$$

$$= 20\log_{10}\left(\frac{x_{max}}{RMSE(x,y)}\right) \tag{5.20}$$

PSNR wird wie SNR in Dezibel (dB) gemessen.

Die genannten Metriken sind leicht zu berechnen, aber sie beziehen den Eindruck, den Menschen von den Abweichungen haben, nicht mit ein [316]. Es ist bekannt, dass manche Abweichungen zwischen den echten und den berechneten Werten kaum bemerkt werden. Dies ist die Basis der verlustbehafteten Kodierungstechniken wie MP3, JPEG oder digitaler Fernsehstandards. Keine der bislang betrachteten Metriken berücksichtigt die menschlichen Eindrücke. Als nächstes stellen wir den *Universal Image Quality Index* (UIQI) [563] vor, der Änderungen in der Struktur von Bildern

zu erfassen sucht, weil das menschliche Auge darauf sehr sensibel reagiert. Wir werden die Berechnung dieser Metrik für Grauwert-Bilder vorstellen. Verschiedene Größen müssen berechnet werden [316]:

$$\mu_x = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{5.21}$$

$$\mu_y = \frac{1}{n}\sum_{i=1}^{n} y_i \tag{5.22}$$

$$\ell(x, y) = \frac{2\mu_x\mu_y}{\mu_x^2 + \mu_y^2} \tag{5.23}$$

Die Gleichungen (5.21) und (5.22) berechnen die durchschnittliche Helligkeit jedes der Bilder und aus diesen den Wert $\ell(x, y)$. Bei Bildern gleicher Helligkeit wird $\ell(x, y)$ den Wert 1 annehmen, ansonsten einen davon verschiedenen Wert.

Außerdem betrachten wir Abweichungen. Gleichungen (5.24) und (5.25) berechnen den Kontrast jedes der Bilder und daraus den Wert $c(x, y)$.

$$\sigma_x = \sqrt{\frac{1}{(n-1)}\sum_{i=1}^{n}(x_i - \mu_x)^2} \tag{5.24}$$

$$\sigma_y = \sqrt{\frac{1}{(n-1)}\sum_{i=1}^{n}(y_i - \mu_y)^2} \tag{5.25}$$

$$c(x, y) = \frac{2\sigma_x\sigma_y}{\sigma_x^2 + \sigma_y^2} \tag{5.26}$$

Bei Bildern gleichen Kontrasts wird $c(x, y)$ den Wert 1 annehmen, ansonsten einen davon verschiedenen Wert. Gleichung (5.27) berechnet die Kreuzkorrelation zwischen den beiden Bildern:

$$\sigma_{x,y} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu_x)(y_i - \mu_y) \tag{5.27}$$

$$s(x, y) = \frac{\sigma_{x,y}}{\sigma_x\sigma_y} \tag{5.28}$$

Nimmt die gemäß Gleichung (5.28) berechnete Größe $s(x, y)$ positive Werte an, so gibt es eine gute Korrelation zwischen den beiden Bildern, ansonsten eine inverse Korrelation.

Aus den eingeführten Größen wird nunmehr eine Gesamt-Qualitätsmetrik mittels Gleichung (5.29) berechnet.

$$Q(x, y) = \frac{2\mu_x\mu_y}{\mu_x^2 + \mu_y^2} * \frac{2\sigma_x\sigma_y}{\sigma_x^2 + \sigma_y^2} * \frac{\sigma_{x,y}}{\sigma_x\sigma_y} \tag{5.29}$$

Für identische Bilder wird $Q = 1$ sein und negativ für invers korrelierte Bilder.

Es ist nicht sinnvoll, diese Qualitätsmetrik für ganze Bilder zu berechnen, da eine inverse Korrelation in einem bestimmten Block schon eine negative Korrelation für das gesamte Bild ergibt. Daher wird Gleichung (5.29) immer nur für Blöcke von Pixeln berechnet. Aus den Q-Werten verschiedener Blöcke wird die globale UIQI-Metrik berechnet.

Als Erweiterung der UIQI-Metrik kann man ein Maß für strukturelle Ähnlichkeit in Form des *Structural Similarity Index Measure* (SSIM)-Wertes berechnen [564].

Kühn [316] hat die verschiedenen Metriken verglichen und herausgefunden, dass keine dieser Metriken den anderen wirklich überlegen ist. Er empfiehlt, in der Praxis mehrere dieser Metriken zu berechnen und diese sorgfältig zu vergleichen. Eine Übersicht über einige nützliche Metriken findet sich auch bei Mittal [398].

In der digitalen Kommunikationstechnik ist das Bitfehlerverhältnis (engl. *Bit Error Ratio* (BER)) eine wichtige Metrik.

Definition 5.16: Das **Bitfehlerverhältnis (BER)** ist der Quotient aus der Anzahl der Bitfehler und der Gesamtzahl der übertragenen Bits.

5.3.3 Kriterien für die Datenanalyse

Sensoren sind häufig nicht ideal, sondern weisen Fehler auf. Auch müssen aus den Messungen von evtl. mehreren Sensoren Schlüsse gezogen werden. Hierzu werden i.d.R. Datenanalysetechniken beispielsweise aus dem Bereich des maschinellen Lernens benötigt (siehe auch Seite 17). Ergebnisse der Datenanalyse sind in der Regel mit Unsicherheiten behaftet, sei es durch Unsicherheiten bereits bei den Ausgangsdaten, sei es durch unterschiedliche Analysetechniken. In gewisser Weise ist Datenanalyse daher ein Fall von näherungsweisem Rechnen, obwohl dieser Begriff in diesem Kontext in der Regel nicht benutzt wird.

Ein sehr wichtiger Spezialfall der Datenanalyse ist die Klassifikation von Objekten. Sei X eine Menge von Objekten, die wir klassifizieren möchten. Wir beschränken uns hier auf die binäre Klassifikation.

Beispiel 5.3: Wir betrachten die Suche nach Bernstein an einem Strand. Leider sieht weißer Phosphor, der beispielsweise in Form von Rückständen von Bomben am Rand der Ostsee gefunden wird, sehr ähnlich wie Bernstein aus. Er fängt aber sehr plötzlich an, bei 1300 °C zu brennen, sobald er trocknet. Die Klassifikation von Fundstücken als Phosphor oder Bernstein ist daher sehr kritisch und mit hohen Gesundheitsrisiken verbunden. Unerfahrene Personen sollten solche Fundstücke nicht berühren. ∇

Allgemein sind bei der binären Klassifikation vier Fälle möglich:

- **richtig positiv** (engl. *true positive* (TP)): wir klassifizieren ein Objekt als positiv und das ist richtig. Im obigen Beispiel: wir halten ein Fundstück für Bernstein und es ist Bernstein.

- **falsch positiv** (engl. *false positive* (FP)): wir klassifizieren ein Objekt als positiv und das ist falsch. Im obigen Beispiel: wir halten Phosphor für Bernstein (eine gefährliche Fehleinschätzung).
- **richtig negativ** (engl. *true negative* (TN)): wir klassifizieren ein Objekt als negativ und das ist richtig. Im obigen Beispiel: wir halten ein Fundstück für Phosphor und es ist Phosphor.
- **falsch negativ** (engl. *false negative* (FN)): wir klassifizieren ein Objekt als negativ und das ist falsch. Im Beispiel: wir klassifizieren ein Fundstück als Phosphor und es ist Bernstein[4].

Absolute Zahlen (beispielsweise Anzahlen von gefundenen Objekten) müssen zueinander in Beziehung gebracht werden. Daher werden die folgenden Metriken definiert:

Definition 5.17: Unter **Genauigkeit** (engl. *precision*) verstehen wir das Verhältnis

$$p = \frac{TP}{TP + FP} \tag{5.30}$$

Im Fall der Bernsteinsuche zielen wir auf einen Wert von 1 (also auf $FP = 0$), weil wir uns nicht verbrennen möchten. Im Zweifelsfall würden wir also ein Objekt sicherheitshalber als Phosphor klassifizieren. Mithin müssen wir die Möglichkeit falsch negativer Klassifikationen (d.h. $FN > 0$) akzeptieren, wenn wir eine gute Genauigkeit erreichen wollen. Anders ist die Situation beispielsweise bei der Krankheitsdiagnose. Hier zielen wir möglichst auf $FN = 0$, um keinen Krankheitsfall zu übersehen, und akzeptieren dafür eher falsch positive Klassifikationen und dementsprechend $p < 1$.

Definition 5.18: Unter **Sensitivität** (engl. *sensitivity* oder *recall*) verstehen wir das Verhältnis

$$r = \frac{TP}{TP + FN} \tag{5.31}$$

Eine gute Genauigkeit führt wegen $FN > 0$ tendenziell zu einer schlechteren Sensitivität.

Definition 5.19: Unter **Korrektklassifikationsrate** (engl. *accuracy*) verstehen wir das Verhältnis

$$acc = \frac{TP + TN}{TP + FP + TN + FN} \tag{5.32}$$

Im Fall der Bernsteinsuche können wir eine nicht-optimale Korrektklassifikationsrate akzeptieren, um die Zahl der falsch positiven Werte so gering wie möglich zu halten. Wir können daher eine größere Zahl von falsch-negativen Klassifikationen haben.

[4] Ein anderes, aktuelles Beispiel der Anwendung der Klassifikation ist der Test auf eine vorliegende COVID-19 Infektion. In diesem Fall sind falsch negative Klassifikationen gefährlich.

Definition 5.20: Unter **Spezifizität** (engl. *specificity*) verstehen wir das Verhältnis

$$specificity = \frac{TN}{TN + FP} \tag{5.33}$$

Definition 5.21: Der *F1 score* oder das **F-Maß** ist definiert als das harmonische Mittel von Genauigkeit und Sensitivität:

$$F1 = 2\,\frac{p * r}{p + r} \tag{5.34}$$

In einem allgemeineren Kontext ist *Quality of Service* (QoS) eine weitere bekannte Metrik. Häufig bezieht sich diese Metrik auf die Qualität von Kommunikationskanälen, für die Bitfehlerquotienten, Latenzzeit und Bandbreite mögliche Kriterien sind.

In einem noch größeren Zusammenhang könnten wir nicht nur technische Parameter betrachten, sondern uns insgesamt auf die Erfahrung des Benutzers beziehen. Diese wird in der Metrik *Quality of Experience* (QoE) erfasst. Sie schließt alle Aspekte der Erfahrungen von Nutzern ein. Es gibt eine Anzahl von Metriken, die alle versuchen, die Erfahrungen von Nutzern zu messen [401].

5.4 Modelle des Energieverbrauchs und der Leistungsaufnahme

5.4.1 Allgemeine Eigenschaften

Energiemodelle und **Leistungsmodelle** sind wichtig, um die entsprechenden Ziele bewerten zu können und um Optimierungen vorzunehmen, welche diese Größen reduzieren sollen. Sie werden auch für Optimierungen verwendet, die Betriebstemperaturen verringern und die Zuverlässigkeit verbessern sollen. Ein Modell der Leistungsaufnahme wird auch im *power management* benötigt (siehe Seite 404).

Energie- und Leistungsmodelle sind eng miteinander verwandt, wie in Gleichung (3.13) zu sehen ist. Gemäß dieser Gleichung ergeben sich Energiemodelle häufig durch Integration von Leistungswerten über der Zeit. Umgekehrt kann aus Energiewerten auch mittlere Leistungsaufnahme in einem Zeitintervall bestimmt werden. Wir können zwischen zwei Klassen von Modellen unterscheiden:

1. Die Klasse der Modelle, die auf **Messungen auf realer Hardware** basiert. Messungen können sehr genau sein, aber sie können nur bei tatsächlich vorhandener Hardware durchgeführt werden. Die Messung von Spannungen ist üblicherweise sehr einfach und bedarf keiner komplexen Prozeduren.
   Die Messung von Strömen kann mit Stromzangen oder mit *Shunt*-Widerständen erfolgen.

   - **Stromzangen** müssen eines der Stromversorgungskabel umfassen. Sie messen das Magnetfeld, welches durch den Strom verursacht wird, der durch das Kabel

fließt. Der Vorteil dieses Verfahrens liegt darin, dass die Stromversorgung nicht unterbrochen werden muss und dass die Stromversorgung während der Messung unverändert bleibt. Ein Nachteil liegt darin, dass diese Messungen meist nicht sehr genau sind.

- Zur Messung des Stromes könnte man ein einfaches Strommessgerät (Ampèremeter) benutzen, welches man in die Stromzuführung einschleift, d.h. über das der gesamte Strom fließt. Ein Nachteil ist, dass die Schaltung dann bei abgeklemmtem Messgerät gar nicht arbeitet und dass lange Leitungen zum Strommessgerät möglicherweise zusätzliche Störungen verursachen. Daher arbeitet man lieber mit einem *Shunt*-Widerstand, wie in Abb. 5.16 (links) zu sehen. Aufgrund des *Shunt*-Widerstandes werden Ströme, die in das zu testende Gerät

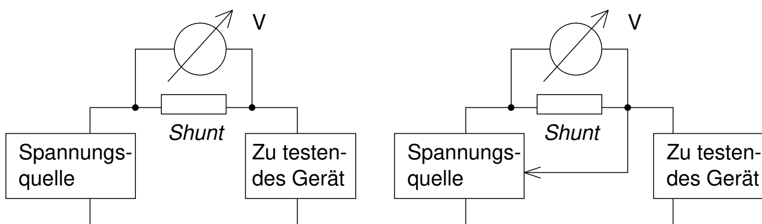


Abb. 5.16 Strommessung: **links**: 2-Draht-Verbindung; **rechts**: Rückkopplung über geregelte Spannungsquelle

fließen, einen Spannungsabfall über eben diesem Widerstand erzeugen. Diese Spannung kann gemessen werden und mit Hilfe des Ohmschen Gesetzes kann der Strom ausgerechnet werden. Die Dimensionierung des *Shunt*-Widerstandes bedarf einiger Überlegung. Wenn der Widerstandswert zu groß ist, dann wird die Spannung am zu testenden Gerät eventuell zu klein und das Gerät funktioniert möglicherweise gar nicht mehr. Wenn der Widerstandswert zu klein ist, dann wird die Spannung über dem *Shunt* zu klein, um zuverlässig gemessen zu werden und wird ggf. durch Rauschen beeinflusst werden. Es hängt vom Strom in das Gerät ab, welcher Widerstandswert geeignet ist. Wenn der Strom sich stark ändert, kann es sogar notwendig sein, mehrere *Shunt*-Widerstände zu benutzen und zwischen diesen abhängig vom Strom umzuschalten.
Das Problem des Spannungsabfalls über dem *Shunt*-Widerstand kann teilweise vermieden werden, wenn eine geregelte Spannungsversorgung benutzt wird und wenn der *Shunt*-Widerstand in die Rückkopplungsschleife des Spannungsreglers integriert wird, wie dies in Abb. 5.16 zu sehen ist. Der Spannungsregler wird dann versuchen, die Spannung über dem Gerät auf dem nominalen Wert zu halten. Allerdings fließt so auch der Strom in den Regeleineingang des Spannungsreglers über den *Shunt*-Widerstand und verfälscht so die abgelesene Spannung.

Leider gibt es in der Regel keinen separaten Stromversorgungsanschluss für jede Komponente im zu testenden Gerät und wir können daher nur die Summe des Strombedarfs der einzelnen Komponenten bestimmen. Wir können jedoch versuchen, das Gerät auf bestimmte Weise zu stimulieren, sodass wir den Strombedarf einzelner Komponenten bestimmen können.

2. Die Klasse der **Modelle**, die nur „virtuell", d.h. **auf Papier, in einem Rechner oder in Gedanken** existieren. Modelle können auch dann benutzt werden, wenn die echte Hardware nicht zur Verfügung steht und stattdessen eine Formel oder ein Rechnerprogramm benutzt wird. Solche Modelle können aber sehr ungenau sein und sie müssen daher überprüft werden, ansonsten blieben sie fragwürdig. Zwei Methoden der Überprüfung werden für viele Leistungs- und Energiemodelle eingesetzt:

   - Die Ergebnisse können mit überprüften Modellen einer niedrigeren Abstraktionsebene verglichen werden.
   - Die Ergebnisse können mit Messungen an echten Geräten verglichen werden, was zu einem hybriden Modell führt. Auf den Basis der Messungen müssen Modellparameter so gewählt werden, dass sich eine möglichst gute Übereinstimmung zwischen dem Modell und den Messungen ergibt. Häufig werden lineare Modelle gewählt und die Modellparameter werden mit der Methode der kleinsten Quadrate gewählt, was zur kleinsten Abweichung gemäß MSE-Metrik (siehe Gleichung (5.16)) führt. Ein solcher Abgleich zwischen dem Modell und den Messwerten wird durch mathematische Werkzeuge wie z.B. MATLAB® unterstützt. Eine neuere Technik hierfür besteht aus dem Einsatz von Methoden des Maschinellen Lernens. Beispielsweise beschreiben Falkenberg et al. [161] den Einsatz Maschinellen Lernens zur Bestimmung eines Modells für die Sendeleistung von Mobilfunkgeräten.

Jedes Leistungsmodell abstrahiert in gewissem Umfang von den Details einer physischen Realisierung: manche Details werden berücksichtigt, andere wiederum nicht. Bei der Wahl eines Leistungsmodells muss man sich daher fragen, ob die gerade relevanten Details berücksichtigt sind. Beispielsweise ist die Alterung häufig nicht modelliert. Aus diesem Grund müssen Modelle für die wichtigen Details ggf. miteinander zu einem Gesamtmodell kombiniert werden. Daher werden wir in diesem Abschnitt repräsentative Modelle der Leistungsaufnahme vorstellen und wir setzen darauf, dass Leser bei Bedarf aus diesen Modellen ein neues Modell für die jeweils benötigte Aufgabe erzeugen.

5.4.2 Energiemodell für Speicher

Im Abschnitt über Speicherhardware (siehe Seite 184) wurde bereits beschrieben, dass die Leistungsaufnahme und der Energieverbrauch von Caches und anderen Speichern mit CACTI [589, 409] bestimmt werden kann. CACTI basiert auf einem abstrakten *Chiplayout* des Speichers, extrahiert Kapazitäten aus diesem *Layout* und

berechnet daraus Zugriffszeiten, Zykluszeiten, Fläche, Leckströme und die dynamische Leistungsaufnahme. CACTI wurde anhand von Speichermodellen auf einer detaillierteren Ebene überprüft. Hierfür wurden SPICE-Modelle benutzt [519]. Gegenwärtig (d.h. im Jahr 2020) ist die jüngste Version von CACTI (Version 6.5) erhältlich von http://www.hpl.hp.com/research/cacti/[5]. Die letzten Erweiterungen betreffen die Modellierung von Leitungen in *Chips*, von kontextabhängigen Speicherzugriffen, von Leitungstreibern und von Leitungsempfängern. Darüber hinaus können Architektur- wie auch technologische Parameter spezifiziert werden.

5.4.3 Energiemodell für Maschinenbefehle

Eines der ersten Energiemodelle für Maschinenbefehle stammt von Tiwari [543]. Dieses Modell unterscheidet zwischen Basiskosten von Maschinenbefehlen und Kosten, die durch den Wechsel von einem Befehl zum nächsten entstehen. Basiskosten modellieren die Energie, die pro Ausführung eines Maschinenbefehls verbraucht wird, wenn eine unendliche Folge von diesen Befehlen ausgeführt wird. Basiskosten können in der Praxis bestimmt werden, indem eine große Anzahl von identischen Befehlen nacheinander ausgeführt werden und anschließend ein Sprung zurück an den Anfang erfolgt. Die Anzahl muss so groß sein, dass der Effekt des Sprungs vernachlässigt werden kann. Dabei müssen die Programme so konzipiert sein, dass keine Wartezyklen (z.B. auf den Speicher) auftreten. Dazu müssen ggf. NOOP-Befehle eingefügt werden, deren Effekt später wieder herausgerechnet wird.

Die Kosten des Wechsels zwischen verschiedenen Befehlen modellieren zusätzliche Energie, die beispielsweise dafür benötigt wird, funktionelle Einheiten an- und abzuschalten. Diese Kosten berücksichtigen den Einfluss des Anfangszustandes auf den Gesamtenergieverbrauch eines Befehls. Sie können berechnet werden, indem Programme mit alternierenden Maschinenbefehlen ausgeführt werden.

Basiskosten und Kosten für Befehlswechsel werden für Programme berechnet, die keine Cachefehler erzeugen. Der Effekt von Cachefehlern muss hinzu gerechnet werden, indem die Cachefehlerrate und die Energie für Speicherzugriffe einbezogen werden. Dabei kann die Energie für Speicherzugriffe bei nicht-homogenem Speicher von den jeweils benutzten Adressen abhängen. Bei Tiwari werden diese Adressen nicht statisch vorhergesagt. Somit kann die Energie für Speicherzugriffe bei nicht-homogenem Speicher nur durch eine Ausführung des Programms bestimmt werden.

Das Modell von Tiwari wurde u.a. auf zwei reale Systeme angewendet, nämlich einen Intel 486 DX2- und einen Fujitsu SPARClite 934-Prozessor. Messungen an den realen Systemen wurden benutzt, um das Modell zu kalibrieren.

[5] Der Zugriff über diese Adresse wird empfohlen, da es auch andere Werkzeuge mit demselben Namen gibt. Derzeit steht eine anpassbare C++-Version zur Verfügung, ein Webinterface existiert i. Ggs. zu früheren Jahren nicht mehr.

5.4.4 Energiemodell für Prozessoreinheiten

Das Energieermittlungswerkzeug Wattch [70] ermittelt Schätzungen für den Energieverbrauch von Mikroprozessorsystemen auf Architekturebene, ohne detaillierte Kenntnisse über die Schaltungs- oder Layoutebene zu benötigen. Wattch benutzt den Simulator SimpleScalar, um Prozessoren zu simulieren. SimpleScalar kann so konfiguriert werden, dass der jeweils benutzte Prozessor so genau wie möglich abgebildet wird. Die Anzahl der Fließbandstufen und die funktionellen Einheiten werden typischerweise korrekt modelliert, i. Ggs. zu mehr spezialisierten Eigenschaften. Wattch basiert auf der detaillierten Information des Energieverbrauchs der verschiedenen funktionellen Einheiten, die man im Mikroprozessor findet. Während der Ausführung merkt sich SimpleScalar, welche funktionellen Einheiten benutzt werden und berechnet daraus den Gesamt-Energieverbrauch.

Wattch benötigt wesentlich mehr Informationen über die Prozessorarchitektur als Tiwaris Model auf Befehlssatzebene. Beispielsweise enthält Wattch ein eigenes detailliertes Modell des Energieverbrauchs in Speichern. Auch wird der Takt explizit berücksichtigt, einschließlich bedingter Taktung, soweit diese eingesetzt wird. In der ursprünglichen Publikation [70] gibt es Angaben zur Überprüfung des Modells anhand von drei verschiedenen Prozessoren.

5.4.5 Energiemodell für Prozessor und Speicher

Als nächstes betrachten wir ein Energiemodell [511], welches hinsichtlich der Detaillierung der Komponenten zwischen dem von Tiwari und Wattch liegt und somit einen Kompromiss zwischen beiden darstellt. In diesem Modell betrachten wir die Summe des Energieverbrauchs in der CPU und im Speicher, jeweils aufgeteilt in den Einfluss der Befehle und der Daten:

$$E_{total} = E_{cpu_instr} + E_{cpu_data} + E_{mem_instr} + E_{mem_data} \tag{5.35}$$

Jeder der vier Terme wird aus detaillierten Gleichungen berechnet. Die folgende Notation wird in diesen Gleichungen benutzt: m ist die Anzahl der betrachteten Befehle, die Funktion $w(b)$ liefert die Anzahl von Einsen in ihrem Argument (entweder Befehl oder Datum), die Funktion $h(b_1, b_2)$ liefert den Hamming-Abstand zwischen den beiden Argumenten, dir bezeichnet die Richtung eines Datentransfers und α_i sowie β_i ($i \in \{1..10\}$) sind Konstanten, die aus der Optimierung der Modellparameter resultieren. Damit berechnen wir E_{cpu_data} wie folgt:

$$\begin{aligned} E_{cpu_data} = \sum_{i=1}^{m} \big\{ & \alpha_5 * w(DAddr_i) + \beta_5 * h(DAddr_{i-1}, DAddr_i) \\ & + \alpha_{6,dir} * w(Data_i) + \beta_{6,dir} * h(Data_{i-1}, Data_i) \big\} \end{aligned} \tag{5.36}$$

wobei $Data_i$ der Datenwert ist, der in Befehl i benutzt wird, und $DAddr_i$ ist dessen Adresse.
Der Term E_{mem_data} ist nur dann relevant, wenn tatsächlich auf den Speicher zugegriffen wird:

$$\begin{aligned}E_{mem_data} = \sum_{i=1}^{m} \Big\{ & BaseMem(DataMem, dir, Word_width) \\ & + \alpha_9 * w(DAddr_i) + \beta_9 * h(DAddr_{i-1}, DAddr_i) \\ & + \alpha_{10,dir} * w(Data_i) + \beta_{10,dir} * h(Data_{i-1}, Data_i) \Big\}\end{aligned} \tag{5.37}$$

wobei $BaseMem$ die Basiskosten für den Speicherzugriff mit Richtung dir sind.
Der Term E_{mem_instr} kann wie folgt berechnet werden:

$$\begin{aligned}E_{mem_instr} = \sum_{i=1}^{m} \Big\{ & BaseMem(InstrMem, Word_width_i) \\ & + \alpha_7 * w(IAddr_i) + \beta_7 * h(IAddr_{i-1}, IAddr_i) \\ & + \alpha_8 * w(IData_i) + \beta_8 * h(IData_{i-1}, IData_i) \Big\}\end{aligned} \tag{5.38}$$

wobei $BaseMem$ die Basiskosten für den Befehlsspeicherzugriff sind, $IAddr_i$ ist die Adresse des Befehls und $IData_i$ ist der Befehl i selbst.
Der Term E_{cpu_instr} kann aus der folgenden Gleichung berechnet werden:

$$\begin{aligned}E_{cpu_instr} = \sum_{i=1}^{m} \Big\{ & BaseCPU(Opcode_i) + FUChange(Instr_{i-1}, Instr_i) \\ & + \alpha_4 * w(IAddr_i) + \beta_4 * h(IAddr_{i-1}, IAddr_i) \\ & + \sum_{j=1}^{s} \big(\alpha_1 * w(Imm_{i,j}) + \beta_1 * h(Imm_{i-1,j}, Imm_{i,j}) \big) \\ & + \sum_{k=1}^{t} \big(\alpha_2 * w(Reg_{i,k}) + \beta_2 * h(Reg_{i-1,k}, Reg_{i,k}) \big) \\ & + \sum_{k=1}^{t} \big(\alpha_3 * w(RegVal_{i,k}) + \beta_3 * h(RegVal_{i-1,k}, RegVal_{i,k}) \big) \Big\}\end{aligned} \tag{5.39}$$

wobei $BaseCPU$ die Basiskosten für den Befehlscode $Opcode_i$ sind. $FUChange(..)$ stellt den Energieverbrauch für den Übergang von Befehl $i-1$ zu Befehl i dar. Imm modelliert den Einfluss von bis zu s Direktoperanden per Befehl. $Reg_{i,k}$ bezieht sich auf die Registernummern von bis zu t Registern pro Befehl und $RegVal_{i,k}$ modellieren bis zu t Registerwerte pro Befehl.
Zur Bestimmung der Konstanten wurden spezielle Befehlssequenzen entworfen.

Beispiel 5.4: Energieverbrauchsbestimmung von *load word* (`lw`)-Befehlen:

```
start: lw R1, address                                /* load word */
       ...                          /* lw-Befehl 50-100-fach wiederholt */
       bra start                                /* zurück zum Anfang */
```

Der Energieverbrauch des Rücksprungs kann vernachlässigt werden. Der Einfluss der Adressen, Registernummern und Registerinhalte kann durch Variation dieser Werte bestimmt werden: wir setzen diese Werte zunächst auf Null und erhöhen dann inkrementell die Anzahl der Einsen in der Bitvektordarstellung. ∇

In unseren Experimenten wurden die Konstanten durch eine lineare Regression bestimmt. Alternativ hätte Maschinelles Lernen benutzt werden können. Es ergab sich ein signifikanter Einfluss der Anzahl der Einsen in der Bitvektordarstellung der Daten, der in Tiwaris Modell nicht erfasst werden kann.

5.4.6 Energiemodell für eine Anwendung

Die Odroid XU3-Plattform [203] (siehe Abb. 5.17) besitzt mehrere Strom- und Spannungssensoren. Diese Sensoren ermöglichen eine präzise Messung der Leistungsaufnahme während der Ausführung von Anwendungen, wobei die Leistungsaufnahme der beiden Arten von ARM®-Prozessorkernen, der GPU und des DRAMs einzeln gemessen werden können. Diese Sensoren werden in mehreren Arbeiten genutzt, um verlässliche Angaben zum Energieverbrauch zu machen. Beispielsweise haben Neugebauer et al. [418] diese Plattform in ihre Exploration des Entwurfsraums für eine Anwendung integriert. Diese Exploration auf der Basis eines evolutionären Algorithmus wird in Abb. 5.18 gezeigt.

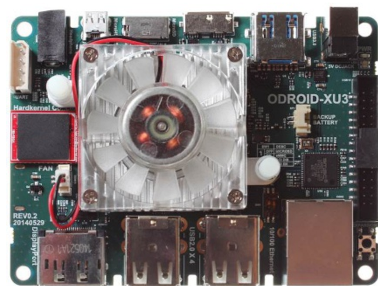


Abb. 5.17 Odroid XU3

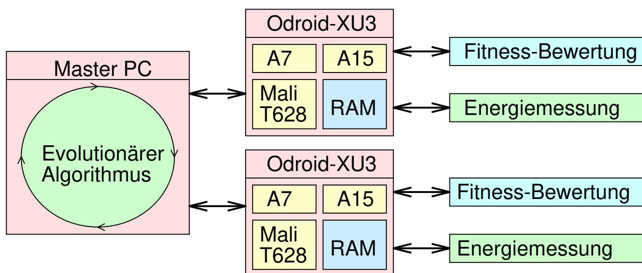


Abb. 5.18 Evolutionärer Algorithmus, Fitness-Berechnung auf der Basis von Messungen

Die Bewertung einer bestimmten Lösung des Entwurfsproblems basiert auf der Messung von Strömen während der Ausführung der Anwendung auf dem XU3. Damit ist es möglich, Energiemodelle mit unbekannter Genauigkeit zu vermeiden. Der resultierende optimierte Algorithmus wurde von Neugebauer innerhalb des cyber-physikalischen PAMONO-Systems eingesetzt [418]. PAMONO ist in der Lage, mittels des sogenannten Plasmon-Effekts Bio-Viren zu erkennen. Gerade in Pandemie-Zeiten könnte es ein großer Vorteil sein, wenn es möglich wäre, Viren innerhalb von kurzen Zeitspannen zu detektieren.

Die Odroid-XU3-Plattform ist leider nicht mehr erhältlich. Sie wurde durch eine Plattform ohne Stromsensoren ersetzt.

5.4.7 *Energiemodell für mehrere Anwendungen und Hardware-Multithreading*

Eine Analyse der Ausführung von mehreren Anwendungen wurde von Kerrison und Eder durchgeführt [291]. Sie benutzten dafür den XMOS XS1-L Prozessor, der zur Unterstützung von Echtzeitanwendungen ein *Multithreading* in Hardware realisiert. Er ist in der Lage, einen schnellen Kontextwechsel zwischen vier Anwendungen durchzuführen. Eine der wissenschaftlichen Fragestellungen war: wie viel kostet der hardwaremäßige Kontextwechsel, z.B. für das Bewertungskriterium Energieverbrauch?

Aufgrund der Verfügbarkeit von realer Hardware konnte diese Frage durch Messungen beantwortet werden. Die Leistungsaufnahme des XMOS XS1-L wurde mit einem *Shunt*-Widerstand gemessen, der in die Stromzuführung einfügt wurde und dessen Spannungsabfall mit einem INA219-Leistungsmessungs-*Chip* (siehe http://www.ti.com/product/ina219) bestimmt wurde. Die auf dem XMOS XS1-L ausgeführte Software wurde durch einen zweiten Prozessor gesteuert. Die beste Energieeffizienz wurde erreicht, wenn alle vier Hardware-*Threads* genutzt wurden. Allerdings führt das Hardware-*Multithreading* zu vielen Lade- bzw. Entlade-Operationen und einer entsprechenden Leistungsaufnahme. Diese hängt von den jeweils ausgeführten Befehlen ab.

Das Ergebnis einer Analyse der Abhängigkeit der Leistungsaufnahme bei Befehlsausführung ist für den Fall von 8-Bit-Daten in Abb. 5.19 zu sehen. Die zwei Dimensionen des Diagramms stellen die Maschinenbefehle dar, die in den geraden bzw. ungeraden *Threads* ausgeführt werden. Gestrichelte Linien grenzen Bereiche mit einer unterschiedlichen Anzahl von Operanden voneinander ab. Befehle mit drei oder mehr Operanden sind in jedem Diagramm rechts bzw. oben aufgeführt. Es ist deutlich zu sehen, dass die Leistungsaufnahme mit der Anzahl der Operanden steigt.

Abb. 5.20 zeigt die entsprechenden Ergebnisse für 16-Bit-Daten. Abb. 5.20 verdeutlicht, dass die Verarbeitung von 16-Bit-Daten mehr Leistung benötigt, als die von 8-Bit-Daten. Kerrison et al. benutzen diese Ergebnisse, um Software eingebetteter Systeme zu optimieren. So kann man mit diesen Ergebnissen beispielsweise ent-

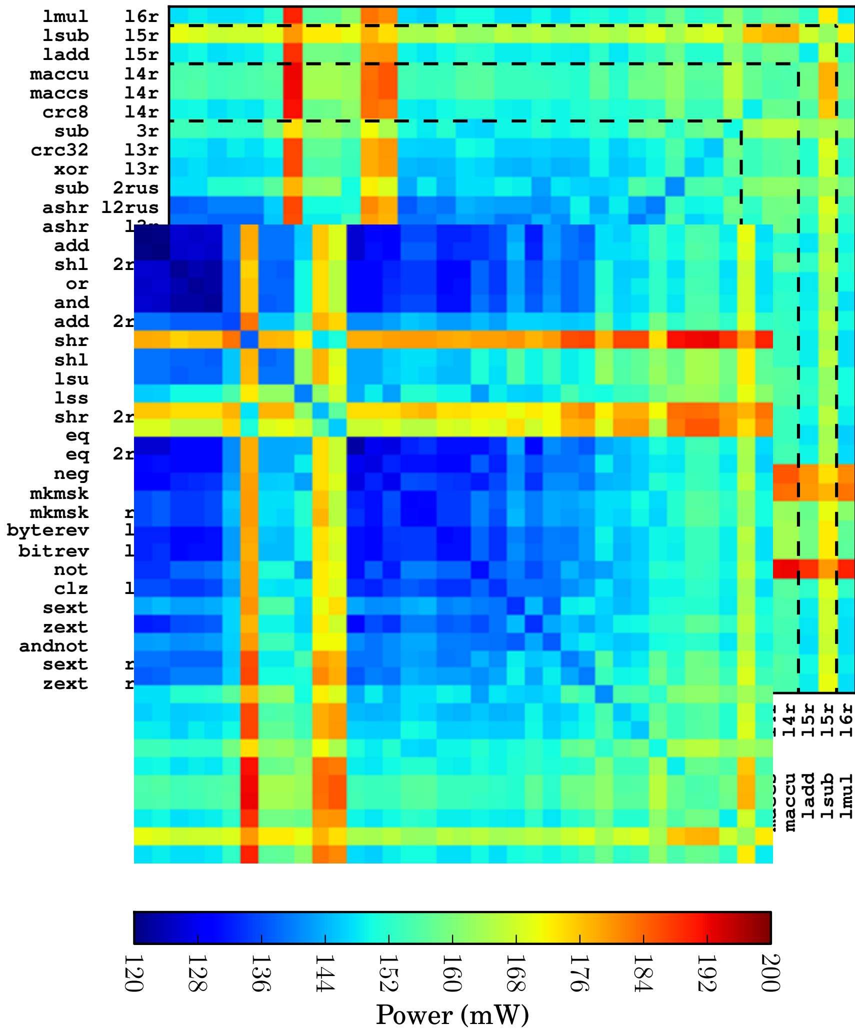


Abb. 5.19 Oben: Leistungsaufnahme für *Multithreading* bei 8-Bit-Daten, vertikal: Befehle in geraden *Threads*, horizontal: Befehle in ungeraden *Threads*; **Unten**: Farbcodierung von Temperaturen; ©Kerrison, Eder, 2008

scheiden, ob byte- oder wortweise Speicherzugriffe energetisch günstiger sind oder ob die lokale Speicherung von Registern gegenüber von Speicherzugriffen Vorteile bringt.

Abb. 5.20 **Oben**: Leistungsaufnahme für Multithreading bei 16-Bit-Daten; **Unten**: Farbcodierung von Temperaturen; ©Kerrison, Eder, 2008

5.4.8 Energiemodell für Android-Mobiltelefone

Zhang et al. [611] beschreiben eine Technik zur Erzeugung eines Energiemodells für ein Android-Mobiltelefon, die PowerBooter genannt wird. Diese Technik basiert auf der Gleichung (5.40).

$$E = (\beta_{uh} * freq_h + \beta_{ul} * freq_l) * util + \beta_{CPU} * CPU_{on}$$
$$+\beta_{br} * brightness + \beta_{Gon} * GPS_on + \beta_{Gsl} * GPS_sl$$
$$+\beta_{WiFi_l} * WiFi_l + \beta_{WiFi_h} * WiFi_h + \beta_{3G_idle} * 3G_{idle}$$
$$+\beta_{3G_FACH} * 3G_{FACH} + \beta_{3G_DCH} * 3G_{DCH} \quad (5.40)$$

mit

$\beta_{..}$: zu bestimmende Konstanten
$freq_i$: CPU-Frequenzen
$util$: CPU-Auslastung
CPU_{on} : Prozessor eingeschaltet?
$brightness$: berücksichtigt die Helligkeit der Anzeige
$GPS_{..}$: bezieht sich auf die GPS-Benutzung
$WiFi_l$: Zeit, in der WLAN im langsamen Modus ist
$WiFi_h$: Zeit, in der WLAN im schnellen Modus ist
$3G_{3G_idle}$: Zeit, in der 3G-Funk unbeschäftigt ist
$3G_{FACH}$: Zeit, in der ein gemeinsamer 3G-Kanal benutzt wird
$3G_{DCH}$: Zeit, in der ein dedizierter 3G-Kanal benutzt wird

Es wird deutlich, dass PowerBooter von den Details der Hardware abstrahiert. Die Gleichung (5.40) modelliert auch die Kommunikation, die in den bislang betrachteten Modellen nicht explizit vorkam. Die Parameter werden wie in den bislang beschriebenen Modellen durch Messungen und Parameteroptimierungen bestimmt. Die Messungen wurden mit dem *Monsoon*-Leistungsmessgerät (siehe http://www.msoon.com/LabEquipment/PowerMonitor/) vorgenommen.

Das gewonnene Modell erlaubt in Kombination mit dem Batteriemodell eine Vorhersage der Batterielaufzeit. Die resultierende Information wird einem Werkzeug mit dem Namen PowerTutor übermittelt. Mit diesem Werkzeug sollen Anwendungen an verschiedene Hardwareplattformen angepasst werden können und Anwendungsprogrammierer sollen Energiespartechniken nutzen können, ohne dass sie sich mit den Besonderheiten der jeweiligen Plattformen intensiv beschäftigen müssen.

Ein weiteres Modell des Energieverbrauchs in Mobiltelefonen wurde von Dusza et al. [144] publiziert. Darüber hinaus gibt es auch kommerzielle Werkzeuge zur Analyse des Energieverbrauchs bzw. der Leistungsaufnahme.

Alle bislang vorgestellten Energiemodelle zielen auf die Vorhersage der **durchschnittlichen** Leistungsaufnahme bzw. des **durchschnittlichen** Energieverbrauchs. Dabei erfolgt die Durchschnittsbildung i.d.R. für bestimmte Eingaben oder Anfangszustände. Durchschnittliche Werte sind hilfreich, um thermisches Verhalten oder die Batterielaufzeit über bestimmte Zeitintervalle vorherzusagen.

5.4.9 Größtmöglicher Energieverbrauch

In manchen Kontexten sind dagegen der größtmögliche Energieverbrauch oder die größtmögliche Leistungsaufnahme interessant.

Definition 5.22: Der größtmögliche Energieverbrauch (engl. *Worst Case Energy Consumption* (WCEC)) eines eingebetteten Systems ist das Maximum des Energieverbrauchs, der für alle Eingaben und alle Anfangszustände möglich ist.

Definition 5.23: Die größtmögliche Leistungsaufnahme (engl. *Worst Case Power Consumption* (WCPC)) eines eingebetteten Systems ist das Maximum der Leistungsaufnahme, die für alle Eingaben und alle Anfangszustände möglich ist.

Die größtmögliche Leistungsaufnahme spielt bei der Dimensionierung der Verbindungen und der Stromversorgung eine Rolle. Der größtmögliche Energieverbrauch ist für die Wahl eines Batteriesystems relevant. Dieses muss die WCEC-Anforderungen erfüllen. Eine sichere obere Schranke für die WCEC kann wie folgt berechnet werden:

$$\mathrm{WCEC} \leq \int_0^{\mathrm{WCET}} \mathrm{WCPC}\,\mathrm{d}t = \mathrm{WCET} * \mathrm{WCPC} \qquad (5.41)$$

Jayaseelan et al. [272], Pallister et al. [443] und Wägemann et al. [558] haben Methoden zur Bestimmungen engerer Schranken vorgeschlagen.

5.5 Thermische Modelle

Das Streben nach höherer Leistung eingebetteter Systeme macht es wahrscheinlicher, dass Komponenten im Betrieb heiß werden. Die Temperaturen der einzelnen Komponenten eines eingebetteten Systems haben einen großen Einfluss auf ihre Verwendbarkeit, z.B. auf die Qualität von Sensorwerten. Überhitzte Komponenten können auch dazu führen, dass das betroffene eingebettete System selbst ganz ausfällt oder dass andere Systeme geschädigt werden. Ein Beispiel hierfür wäre eine entstehende Brandgefahr. Aber auch wenn kein direkter Ausfall vorliegt, können überhitzte Komponenten andere Folgen haben. Beispielsweise kann die Lebensdauer eines Systems durch Überhitzung erheblich sinken (siehe Gleichung (5.73) auf Seite 309). Auch kann es notwendig sein, Teile von Silizium-*Chips* von der Stromversorgung zu trennen, um Überhitzung zu vermeiden. Dies wurde unter dem Begriff *dark silicon* bekannt [153].

Das thermische Verhalten eingebetteter Systeme hängt eng mit der Umwandlung von elektrischer Energie in Wärme zusammen. Daher sind Temperaturmodelle eng mit Energiemodellen verwandt. Grundlage für Temperaturmodelle sind physikalische Gesetze[6].

[6] Wir benutzen hier das Symbol θ zur Bezeichnung von Temperaturen, um eine Verwechslung mit Perioden zu vermeiden, für die das Symbol T verwendet wird.

5.5.1 Stationäres Verhalten

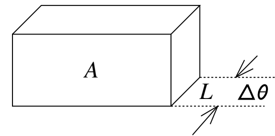


Abb. 5.21 Schicht der Dicke L

Wir betrachten eine homogene Schicht aus einem bestimmten Material der Fläche (des Querschnitts) A und der Dicke L (siehe Abb. 5.21). Wir nehmen an, dass es zwischen den gegenüberliegenden Seiten eine Temperaturdifferenz $\Delta\theta$ gibt. Weiterhin setzen wir voraus, dass sich die Wärme in alle Richtungen gleich ausbreitet (Isotropie) und dass wir uns im **stationären (eingeschwungenen) Zustand** befinden. Unsere Aussagen gelten näherungsweise für den Fall, dass die Fläche A groß gegenüber der Dicke L ist. Wir ignorieren dementsprechend Effekte, die sich an den Rändern der Fläche ergeben. Dann ist die thermische Leistung, die zwischen den gegenüberliegenden Seiten transferiert wird, gleich

$$P_{th} = \kappa \frac{\Delta\theta * A}{L} \quad \text{mit:} \tag{5.42}$$

P_{th} : transferierte thermische Leistung κ : Wärmeleitfähigkeit A : Fläche
$\Delta\theta$: Temperaturdifferenz L : Dicke

Dieser Zusammenhang heißt auch **Fouriersches Gesetz**.

Definition 5.24: Aufgrund von Gleichung (5.42) definieren wir die **Wärmeleitfähigkeit** (auch **Wärmeleitzahl**, **spezifisches Wärmeleitvermögen**, **thermische Leitfähigkeit** oder englisch *thermal conductivity* genannt) κ als die thermische Leistung P_{th}, die durch eine Schicht der Dicke 1 und der Fläche 1 fließt, wenn sich die Temperaturen an den Enden um eine Temperatureinheit unterscheiden (jeweils in den benutzten Einheiten gemessen).

Anstelle von κ wird häufig das Symbol λ benutzt. κ ist abhängig vom Material und den Umgebungsbedingungen. Tabelle 5.1 enthält Werte von κ für verschiedene Materialien und Umgebungsbedingungen. Weitere Informationen zur Abhängigkeit von den Umgebungsbedingungen sind in den zitierten Quellen aufgeführt.

Tabelle 5.1 Näherungsweise thermische Eigenschaften für Luft, Kupfer und Silizium

Material	κ: Wärmeleitfähigkeit	c_p: spezifische Wärme	c_v: Volumetrische Wärmekapazität
	(W/(K m))	(J/(K g))	(J/(K m^3))
Luft (25 °C)	0.025 [583]	1.012 [576]	$1.21 * 10^3$ [576]
Kupfer	401 [583]	0.385 [576, 567]	$3.45 * 10^6$ [576]
Silizium (≈ 26 °C)	148 [148]	0.705 [148, 567]	$1.64 * 10^6$ [148][a]

[a] Aus Gleichung (5.57) berechnet

Definition 5.25: Das **Wärmeleitvermögen** [169] (engl. *thermal conductance*) ist definiert als die thermische Energie, die durch eine Schicht pro Zeiteinheit transferiert wird, wenn die beiden Enden eine Temperaturdifferenz von einer Temperatureinheit (typischerweise in Kelvin gemessen) besitzen.

Aus Gleichung (5.42) folgt

$$\frac{P_{th}}{\Delta\theta} = \kappa * \frac{A}{L} \tag{5.43}$$

Der reziproke Wert dieser Größe heißt **thermischer Widerstand** (auch Wärmeleitwiderstand) R_{th}:

$$R_{th} = \frac{\Delta\theta}{P_{th}} = \frac{L}{\kappa * A} \tag{5.44}$$

Lemma 5.1: *Thermische Widerstände addieren sich wie elektrische Widerstände. Wir können thermisches Verhalten analog zu elektrischem Verhalten modellieren.*

Beispiel 5.5: Wir betrachten einen Mikroprozessor, der eine thermische Leistung P_{th} erzeugt, einen thermischen Widerstand $R_{th,chip}$ des Prozessorchips und thermischen Widerstand $R_{th,fan}$ des Lüfters (siehe Abb. 5.22).

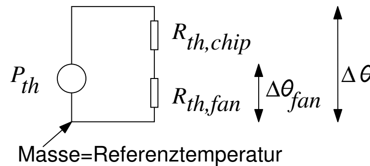


Abb. 5.22 Thermisches Modell eines Mikroprozessors mit Lüfter

Es gilt

$$\Delta\theta = R_{th} * P_{th} \tag{5.45}$$

$$R_{th} = R_{th,chip} + R_{th,fan} \tag{5.46}$$

Als Beispiel betrachten wir die folgenden Werte:

$$R_{th,chip} = 0.4\,\text{W/K} \tag{5.47}$$

$$R_{th,fan} = 0.3\,\text{W/K} \tag{5.48}$$

$$P_{th} = 10\,W \tag{5.49}$$

Daraus errechnen wir:

$$\Delta\theta = 7\,\text{K} \tag{5.50}$$

$$\Delta\theta_{fan} = 3\,\text{K} \tag{5.51}$$

∇

Die Leistungsaufnahme und das Wärmeleitvermögen liefern Anhaltspunkte für die Berechnung der sogenannten *Thermal Design Power* (TDP).

Definition 5.26 ([584]): Die *Thermal Design Power (TDP)* ist die maximale thermische Leistung einer Rechnerkomponente, für die das Kühlsystem entworfen wurde. Die TDP dient anstelle der echten Leistungsaufnahme als nominelle Referenz für den Entwurf des Kühlsystems.

Wir könnten versuchen, die TDP aus der WCPC und daraus folgend den maximalen Temperaturen sowie dem thermischen Widerstand zu berechnen. Häufig entsprechen die publizierten TDP-Werte aber nicht der WCPC, sondern sind niedriger angesetzt. Aus diesem Grund werden Temperatursensoren benötigt, um einen sicheren Betrieb zu gewährleisten. Notfalls müssen funktionelle Bereiche abgeschaltet werden, wodurch Teile von *Chips* zu *dark silicon* werden können.

5.5.2 Transientes Verhalten

Bislang haben wir immer nur den eingeschwungenen Zustand betrachtet. Im Allgemeinen muss man zeitlich veränderliche Systeme und die Speicherung mit Wärmekapazitäten betrachten.

Definition 5.27: Die **Wärmekapazität** eines Objekts ist definiert als die Menge thermischer Energie E_{th}, die je Differenz $\Delta\theta$ der Temperaturen gespeichert werden kann:

$$C_{th} = \frac{E_{th}}{\Delta\theta} \tag{5.52}$$

Primär hängt C_{th} von der Menge und des Art des Materials im Objekt ab:

$$C_{th} = c_p * m \tag{5.53}$$

wobei c_p die spezifische Wärmekapazität ist und m die Masse. Wir können Gleichung (5.53) als Definition der spezifischen Wärme betrachten:

Definition 5.28: Die **spezifische Wärmekapazität** c_p eines Objekts der Masse m ist definiert als

$$c_p = \frac{C_{th}}{m} \tag{5.54}$$

c_p hängt vom Material ab und ist temperaturabhängig, kann aber für kleine Temperaturdifferenzen als konstant angenommen werden.

In unserem Kontext ist es häufig sinnvoller, statt der Wärmekapazität pro Masseneinheit die Wärmekapazität pro Volumeneinheit zu betrachten. So ist primär i.d.R. das Volumen einer Schicht eines Materials bekannt und nur in abgeleiteter Form dessen Masse.

Definition 5.29: Die **volumetrische Wärmekapazität** c_v ist definiert als

$$c_v = \frac{C_{th}}{\mathcal{V}} \tag{5.55}$$

wobei $\mathcal{V}$ das Volumen des Objektes ist.

c_v und c_p hängen über die spezifische Dichte miteinander zusammen:

Definition 5.30: Die **spezifische Dichte** ρ ist definiert als

$$\rho = \frac{m}{\mathcal{V}} \tag{5.56}$$

Wenn wir $\mathcal{V} = m/\rho$ in die Definition von c_v einsetzen, dann folgt

$$c_v = \frac{C_{th}}{\mathcal{V}} = \frac{C_{th} * \rho}{m} = c_p * \rho \tag{5.57}$$

Damit ist es uns möglich, zwischen Tabellen für c_p und c_v umzurechnen (siehe z.B. Tabelle 5.1).

Die Korrespondenz zu elektrischen Schaltungen erlaubt es uns, auch das transiente thermische Verhalten zu berechnen.

Beispiel 5.6: Zur Demonstration der Berechnung des transienten Verhaltens erweitern wir unser Mikroprozessorbeispiel entsprechend Abb. 5.23 (links). Das resul-

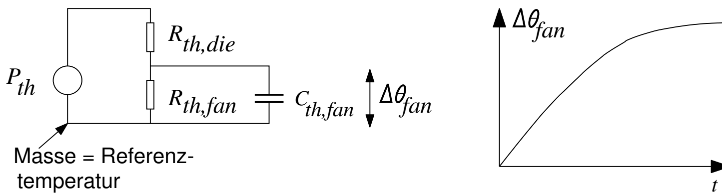


Abb. 5.23 Mikroprozessor mit Lüfter: **links**: thermisches Modell; **rechts**: transientes Verhalten

tierende thermische Verhalten ist in Abb. 5.23 (rechts) gezeigt. Das System nähert sich einem stabilen Zustand wie ein elektrisches Netzwerk aus Widerständen und Kondensatoren. ∇

Insgesamt ist es möglich, thermisches Verhalten mittels des äquivalenten elektrischen Verhaltens zu modellieren. Tabelle 5.2 zeigt die Äquivalenzen zwischen den Größen. Zur Lösung der Netzwerkgleichungen für elektrische Netzwerke können bekannte Techniken eingesetzt werden (siehe z.B. Chen et al. [97]). Allerdings gibt es kein Äquivalent zu Induktivitäten auf der thermischen Seite.

Tabelle 5.2 Äquivalenzen zwischen elektrischen und thermischen Modellen

Elektrisches Modell		Thermisches Modell	
Strom	I	Thermischer Fluss, „Leistungsfluss"	$P_{th} = \dot{Q}$
Gesamtladung	$Q = \int I \, dt$	Thermische Energie	$E_{th} = \int P_{th} \, dt$
Potential	ϕ	Temperatur	θ
Spannung=Potentialdifferenz	$V = \Delta\phi$	Temperaturdifferenz	$\Delta\theta$
Widerstand[a]	$R = \rho_{el} \, \frac{L}{A}$	Thermischer Widerstand	$R_{th} = \frac{1}{\kappa} \, \frac{L}{A}$
Ohmsches Gesetz	$V = R * I$	Δ Temperatur über R_{th}	$\Delta\theta = R_{th} * P_{th}$
Kapazität	C	Wärmekapazität	C_{th}
Ladung auf Kondensator	$Q = C * V$	Energie in Wärmekapazität	$E_{th} = C_{th} * \Delta\theta$
Kapazität eines Objekts[b]	$C = \rho_q \mathcal{V}$	Kapazität eines Objekts	$C_{th} = c_v \mathcal{V}$

[a]: ρ_{el} ist der spezifische elektrische Widerstand. [b]: ρ_q ist die Ladungsdichte.

Die Äquivalenz zwischen thermischen und elektrischen Modellen wird in Werkzeugen wie beispielsweise HotSpot [499] ausgenutzt. Abb. 5.24 zeigt ein HotSpot-Modell eines *Chips*, der auf einem Wärmeverteiler platziert ist, der wiederum auf einem Kühlkörper montiert ist [500]. Skadron et al. [500] betonen, dass innerhalb

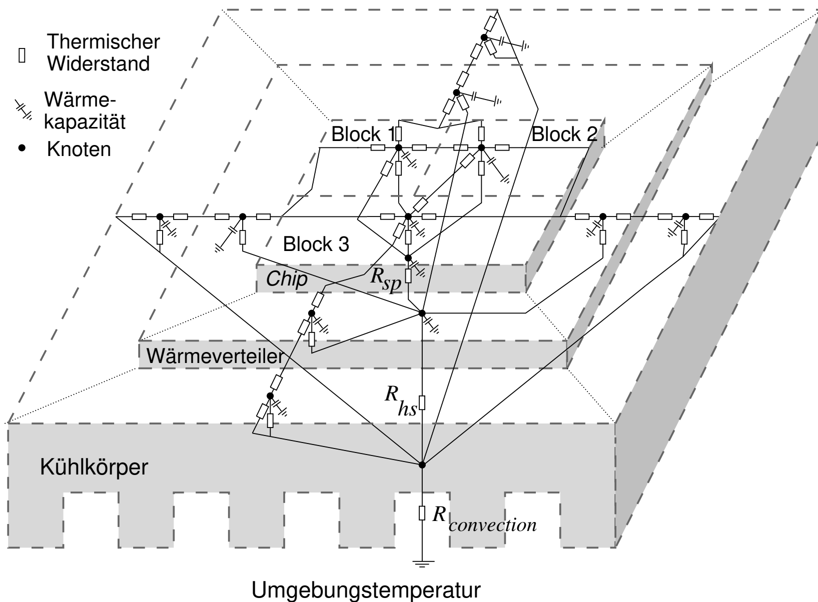


Abb. 5.24 HotSpot-Modell eines Chips auf einem Wärmeverteiler und einem Kühlkörper

eines *Chips*, eines Wärmeverteilers oder eines Kühlkörpers große Temperaturdifferenzen vorhanden sein können. Daher darf für diese Komponenten keine einheitliche Temperatur vorausgesetzt werden. In Abb. 5.24 nehmen wir an, dass der *Chip* drei Mikroarchitektur-Komponenten enthält, von denen jede eine thermische Zone definiert.

Der Wärmeverteiler und der Kühlkörper sind je als fünf thermische Zonen modelliert. Eine Zone des Wärmeverteilers befindet sich unter dem *Chip*, die vier weiteren Zonen modellieren jeweils eine Randzonen. Sie sind trapezförmig. Gepunktete Linien kennzeichnen Grenzen der Trapeze. Entsprechendes gilt für die fünf Zonen des Kühlkörpers. Die verdeckten mittleren Zonen sind aus Darstellungsgründen nicht gezeigt. Ansonsten ist jede Zone in Abb. 5.24 als Knoten in dem äquivalenten (elektrischen) Netzwerk gezeigt. Die Umgebungstemperatur wird als einheitlich vorausgesetzt. $R_{convection}$ ist der thermische Widerstand zur Umgebung. Er ist mit den fünf thermischen Zonen des Kühlkörpers verbunden. Die Verbindung zum Wärmeverteiler erfolgt über den Widerstand R_{hs}. Der Wärmeverteiler besteht wiederum aus fünf Zonen. Davon ist die mittlere über den Widerstand R_{sp} mit dem *Chip* verbunden. Die Wärmequellen auf dem *Chip* selbst sind nicht gezeigt. Für jede der Zonen gibt es eine thermische Kapazität. Diese modelliert immer die Temperaturdifferenz zur Umgebung, dementsprechend sind die Kapazitäten im elektrischen Modell mit der Masse verbunden. Außerdem gibt es für jede der Zonen ein Paar von Widerständen, die eine thermische Verbindung zu den Nachbarzonen darstellen.

In ihren Experimenten haben Skadron et al. den Wattch-Simulator (siehe Seite 287) als Wärmequelle benutzt. Wattch wiederum kann durch Mikroarchitektursimulatoren wie z.B. SimpleScalar getrieben werden. HotSpot enthält Mechanismen zur Erzeugung eines Systems aus partiellen Differentialgleichungen für Modelle wie das in Abb. 5.24 gezeigte. Diese Gleichungen werden anschließend mit Runge-Kutta-Verfahren gelöst.

Die Ergebnisse von Skadron et al. bestätigen, dass es notwendig ist, verschiedene thermische Zonen zu betrachten. Der erwartete Einfluss der Leistungsaufnahme auf die Temperatur wurde quantitativ nachgewiesen. Verschiedene Techniken zur Reduktion der Leistungsaufnahme hatten allerdings nur einen kleinen Einfluss auf kritische Temperaturen. Beispielsweise tendieren Registersätze dazu, heiß zu werden. Eine Reduktion der Leistungsaufnahme für Speicherzugriffe hilft in diesem Zusammenhang wenig und kann sogar einen negativen Einfluss haben.

Beispiel 5.7: Wir betrachten ein MPSoC-System von ST Microelectronics mit 64 P2012 Kernen [506]. Die thermische Modellierung dieses MPSoCs wurde mit dem 3D-ICE-Werkzeug [23] vorgenommen. Die Abbildungen 5.25 und 5.26 zeigen relative Temperaturen für dieses MPSoC-System[7]. Hohe Temperaturen sind in rot angezeigt, niedrige in blau. Das MPSoC-System enthält vier Cluster, jedes mit 16 Kernen. Jede Ecke des *Layouts* enthält ein Cluster. Die 16 Prozessoren sind in der Mitte der Cluster platziert. Speicher sind unterhalb und oberhalb der Prozessoren an-

[7] Reproduktion dieser Bilder mit freundlicher Genehmigung durch David Atienza (EPFL). Die Bilder wurden im Rahmen einer Kooperation zwischen EPFL und ST Microlectronics im Projekt *PRO3D: Programming for Future 3D Architectures with Many Cores* gewonnen.

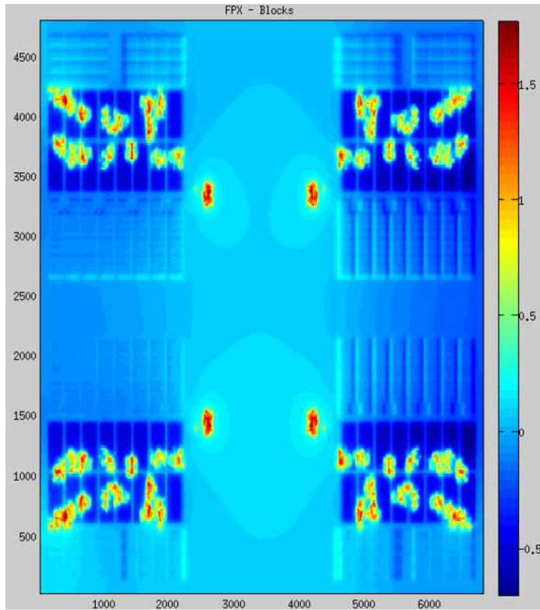


Abb. 5.25 Ergebnisse der thermischen Simulation eines MPSoCs bei 50% Auslastung

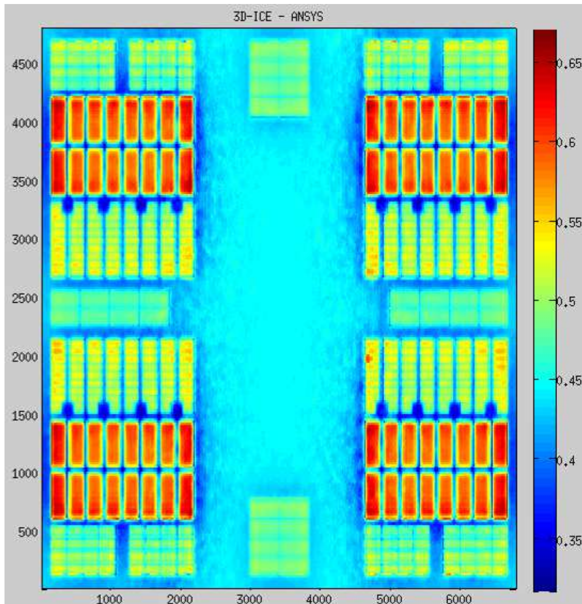


Abb. 5.26 Ergebnisse der thermischen Simulation eines MPSoCs bei 100% Auslastung

geordnet. Die Simulation bestätigt, dass die Prozessoren heißer sind als die Speicher. Die Abbildung 5.25 entspricht der Situation bei halber Prozessorauslastung. Abbildung 5.26 spiegelt die Situation bei voller Prozessorauslastung wider. Die größere Auslastung in Abb. 5.26 führt zu höheren Temperaturen. Die detaillierte Modellierung des *Layouts* hatte zur Folge, dass eine Überschätzung der Temperaturen vermieden wurde. ∇

Die Validierung von Temperaturmodellen erfordert Temperaturmessungen [395].

5.6 Verlässlichkeits- und Risikoanalyse

Als nächstes betrachten wir die Analyse der Verlässlichkeit und möglicher Risiken.

5.6.1 Aspekte der Verlässlichkeit

Eingebettete und cyber-physikalische Systeme können (wie andere Produkte auch) Schäden an Menschen, Tieren und Material verursachen. Wir haben bereits in Tabelle 1.2 auf Seite 19 darauf hingewiesen, dass solche Systeme potentiell sicherheitskritisch sind. Im Allgemeinen werden wir dies berücksichtigen müssen. Dabei ist es unmöglich, das Risiko von Schäden komplett zu eliminieren. Wir können nur dafür sorgen, dass eine ausreichende Verlässlichkeit gegeben ist. Zur Verlässlichkeit gehören die Aspekte der Betriebs- und der Informationssicherheit, die ihrerseits wiederum Aspekte wie die Zuverlässigkeit und Vertraulichkeit enthalten. Bezüglich dieser Aspekte muss offenbar auch eine Bewertung von Entwürfen erfolgen.

5.6.2 Informationssicherheit

Solange auf eingebettete und cyber-physikalische Systeme nicht elektronisch von außen zugegriffen werden konnte, galt deren Informationssicherheit nicht als ein Problem. Dies hat sich geändert, seit auf Systeme über Kommunikationskanäle zugegriffen werden kann. Eine Bewertung dieser Form der Sicherheit muss Angriffsszenarien betrachten, wie bereits in Abschnitt 3.8 erwähnt. Dazu zählen u.a. Seitenkanal-Angriffe. Wenn ein physischer Zugang möglich ist, dann müssen auch physische Angriffe betrachtet werden.

Darüber hinaus müssen die Beziehungen zwischen Ver- und Entschlüsselungsprotokollen und der erreichbaren Datenrate analysiert werden, da es leicht vorkommen kann, dass ressourcenbeschränkte Geräte nicht die erwarteten Ver- und Entschlüsselungsraten erreichen.

5.6.3 Betriebssicherheit

Betriebssicherheit soll ebenfalls Schäden abwenden. Der bestmögliche Ansatz besteht darin, die Wahrscheinlichkeit von Schäden gering zu halten, sodass sie hoffentlich einige Größenordnungen unter der Wahrscheinlichkeit des Auftretens anderer Risiken liegt.

Eine übliche Minimalanforderung an die Fertigung sicherheitskritischer Systeme ist die Einhaltung der Norm ISO 9001. Dieser Standard definiert allgemeine Anforderungen an ein Qualitätsmanagement. Die Anforderungen gemäß diesem Standard beinhalten die folgenden Prinzipien [255]: Kundenfokussierung, Führungsqualitäten, Engagement der beteiligten Personen, prozessbasierter Ansatz, Verbesserungen, evidenzbasierte Entscheidungen und ein Management von Beziehungen. Die ersten vier Prinzipien sind mehr oder weniger selbsterklärend. Das Verbesserungsprinzip schreibt vor, dass Arbeiten in den Phasen Planung, Realisierung, Überprüfen und Handeln ablaufen. In der Planungsphase sollen Ziele, Risiken und Möglichkeiten aufgestellt werden. In der Handlungsphase sollen Verbesserungen, sofern notwendig, umgesetzt werden.

Für den Entwurf sicherheitskritischer Systeme sind spezifischere Richtlinien entwickelt und als Standard IEC 61508 publiziert worden [527]. Teil 1 [233] dieses Standards definiert Standardtechniken für technische Systeme im Allgemeinen. Teil 2 [234] spezifiziert Anforderungen für elektrische, elektronische und programmierbare sicherheitskritische Systeme. Anforderungen an die Software werden in Teil 3 aufgeführt [235]. Die Teile 4 bis 6 enthalten weniger formale Empfehlungen. Diese Standards gehen davon aus, dass es nicht möglich ist, technische Systeme zu entwerfen, die zu jeder Zeit die erwarteten Dienste bereitstellen. Die Standards betonen die Verwendung von Entwurfsprozessen, welche die Ursachen von ggf. falschen Entscheidungen identifizieren können.

Der Standard IEC 61508 unterscheidet zwischen vier verschiedenen Risikoklassen, die *Safety Integrity Levels* (SIL) genannt werden. Für Geräte im Dauerbetrieb verlangt der Standard Ausfallraten von 10^{-5} bis 10^{-6} pro Stunde für SIL-1, 10^{-6} bis 10^{-7} pro Stunde für SIL-2, 10^{-7} bis 10^{-8} pro Stunde für SIL-3 und 10^{-8} to 10^{-9} pro Stunde für SIL-4 [581]. SIL-4 entspricht beispielsweise einem Zwischenfall bei 100.000 Systemen mit einer Betriebszeit von jeweils 10.000 Stunden. Diese niedrige Ausfallrate ist schwer zu erreichen und setzt typischerweise Redundanz voraus. Probleme entstehen durch den gegenwärtigen Trend hin zu gemischt-kritischen Systemen (Stichwort „*mixed-criticality*"), bei denen Subsysteme verschiedener SIL-Klassen beispielsweise auf demselben Mikroprozessor realisiert werden. Eine geeignete Abschirmung der verschiedenen Klassen voneinander ist i.d.R. schwierig.

Der Standard IEC 61508 soll in verschiedenen Branchen Anwendung finden. Für bestimmte Branchen gibt es aber spezifische Erweiterungen des Standards. Diese berücksichtigen u.a. die Zeit, die für menschliche Eingriffe zur Verfügung steht, die Möglichkeit, in einen sicheren Modus zu wechseln und die Auswirkungen von Fehlfunktionen. In einem Auto gibt es beispielsweise wenig Zeit, um zu reagieren. Allerdings können Autos abgebremst und in einem sicheren Modus an einem sicheren Ort geparkt werden (mit Ausnahme von z.B. Tunneln). Im Gegensatz dazu steht in

einem Flugzeug meist mehr Zeit für eine Reaktion zur Verfügung, aber einige der sicherheitskritischen Systeme in einem Flugzeug können nicht einfach abgeschaltet werden.

MISRA-C [397] definiert Regeln, die zu befolgen sind, wenn die Programmiersprache C für sicherheitskritische Systeme benutzt wird.

Der Standard ISO 26262 [253] wurde speziell für die Automobilindustrie entwickelt. Die Standards IEC 62279 und CENELEC 50128 zielen auf die Sicherheit von Schienenfahrzeugen [60].

Im Bereich der Luftfahrt sollten Systeme die Anforderungen gemäß der *Airworthiness Certification Specifications FAR-CS 25.1309 „ Equipment, Systems and Installations"* und *AC-AMC 25.1309 „System design and analysis"* [549] erfüllen. Für Hardware werden diese Standards ergänzt durch den Standard DO-254 und für Software durch den Standard DO-178B („*Software Considerations in Airborne Systems and Equipment Certification*") [474, 163], der in Europa auch ED-12B genannt wird. DO-178C ist ein Nachfolge-Standard zum Standard DO-178B.

Für den Bereich der Fertigung wurde Standard IEC 61511 [237] entwickelt und für Kernkraftwerke kommt IEC 61513 [236] zur Anwendung.

Aus der Betrachtung der SIL-Ebenen geht hervor, dass die Anzahl der erlaubten Ausfälle um einige Größenordnungen unter der Ausfallrate von Halbleiterchips liegt. Aus diesem Grund hat Kopetz [304] betont, dass das System als Ganzes verlässlicher sein muss als irgendeines seiner Teile und dass Sicherheitsanforderungen im Rahmen eines Entwurfs nicht nachträglich berücksichtigt werden können, sondern von Anfang an betrachtet werden müssen. Offensichtlich müssen Fehlertoleranzmaßnahmen benutzt werden. Aufgrund der niedrigen Ausfallrate werden Systeme nicht zu 100 Prozent testbar sein. Daher muss die Sicherheit mit einer Kombination von Testen und logischen Argumenten gezeigt werden. Abstraktion muss genutzt werden, um ein System mit Hilfe einer hierarchischen Menge von Verhaltensmodellen erklären zu können. Entwurfsfehler und menschliche Fehler müssen betrachtet werden.

Zur Bewältigung dieser Herausforderungen hat Kopetz zwölf Entwurfsprinzipien vorgeschlagen:

1. Sicherheitsanforderungen müssen als der wesentliche Teil der Spezifikation betrachtet werden, der den gesamten Entwurfsprozess beeinflusst.
2. Präzise Spezifikationen von Entwurfshypothesen müssen ganz am Anfang vorliegen. Dazu gehören erwartete Fehler und ihre Wahrscheinlichkeit.
3. Die Eindämmung von Fehlern in bestimmten Regionen (engl. *fault containment regions* (FCRs)) muss betrachtet werden. Fehler in einer Region sollen andere Regionen nicht beeinflussen.
4. Es muss eine konsistente Auffassung von Zeit und Zuständen geben. Ansonsten wird es nicht möglich sein, zwischen ursprünglichen und Folgefehlern zu unterscheiden.
5. Die Interna von Komponenten müssen durch gut definierte Schnittstellen versteckt werden.
6. Es muss sichergestellt werden, dass Komponenten unabhängig voneinander ausfallen.

7. Sofern nicht zwei oder mehr Komponenten das Gegenteil behaupten, sollten Komponenten sich selbst als korrekt ansehen (Prinzip des Selbstvertrauens).
8. Fehlertoleranzmaßnahmen müssen so entworfen sein, dass sie keine zusätzlichen Schwierigkeiten bei der Erklärung des Systemverhaltens erzeugen. Fehlertoleranzmaßnahmen sollten von der regulären Funktion entkoppelt sein.
9. Das System muss für die Diagnose entworfen sein. Beispielsweise muss es möglich sein, vorhandene (aber maskierte) Fehler zu identifizieren.
10. Das Benutzerinterface muss intuitiv bedienbar sein und trotz Benutzerfehlern arbeiten. Die Sicherheit sollte trotz menschlicher Fehler gewährleistet sein.
11. Alle außergewöhnlichen Vorfälle sollten protokolliert werden. Diese Vorfälle können über die normalen Schnittstellen möglicherweise nicht beobachtet werden. Die Protokolle sollten interne Effekte erfassen, die sonst ggf. durch Fehlertoleranzmaßnahmen verdeckt werden könnten.
12. Es sollte eine Strategie geben, niemals aufzugeben. Eingebettete Systeme müssen ununterbrochen ihre Dienste bereitstellen können. Die Erzeugung von *Pop-Up* Fenstern oder das vollständige Abschalten können nicht akzeptiert werden.

Definition 5.31: Ein System heißt *resilient* (elastisch), wenn Änderungen an den Entwurfsvoraussetzungen im System selbst oder seiner Umgebung für den Benutzer nur eine begrenzte Änderung zur Folge haben.

Ein sich selbst reparierendes System ist in gewissem Umfang *resilient*. Resilienz liegt außerhalb des in diesem Buch behandelten Themenkreises.

5.6.4 Zuverlässigkeit

Beim Entwurf verlässlicher Systeme muss auch die Zuverlässigkeit analysiert werden, d.h. die Wahrscheinlichkeit, dass ein anfänglich korrektes System aufgrund eines internen Fehlers seine Dienste nicht mehr anbietet. Es wird erwartet, dass diese Aufgabe künftig wichtiger und schwieriger werden wird, da abnehmende Strukturgrößen von Halbleitern deren Zuverlässigkeit reduzieren werden (siehe beispielsweise http://variability.org). Auch kann man annehmen, dass sowohl transiente wie auch permanente Fehler häufiger auftreten werden. Sinkende Strukturbreiten führen auch zu einer erhöhten Variabilität von Schaltungsparametern. Daher werden die Zuverlässigkeitsanalyse und der fehlertolerante Entwurf sehr wichtig werden [407, 179]. Diese Analyse hängt eng mit der Modellierung des thermischen Verhaltens zusammen, da hohe Betriebstemperaturen zu einer niedrigeren Lebensdauer der Geräte führen. Dieser Zusammenhang wird auf Seite 309 in Form von Gleichung (5.73) beschrieben werden. Fehler in Halbleitern können zu Systemausfällen führen.

Die Begriffe **Störung** (engl. *fault*), **Ausfall** (engl. *failure*) und die verwandten Begriffe **Fehler** (engl. *error*) und **Dienst** (engl. *service*) wurden von Laprie et al. [324, 28] definiert:

Definition 5.32: „Der **Dienst**, den ein System (in seiner Rolle als **Anbieter**) liefert, ist sein vom Benutzer beobachtbares Verhalten; ... Der erbrachte Dienst ist dabei

eine Folge von externen Zuständen des Anbieters. ... Ein **fehlerfreier Dienst** wird geliefert, wenn der Dienst die Systemfunktion implementiert".

Definition 5.33: „Ein **Dienstausfall**, auch einfach als **Ausfall** (engl. *failure*) bezeichnet, ist ein Ereignis, das auftritt, wenn der von einem System gelieferte Dienst nicht dem fehlerfreien Dienst entspricht. ... Ein Dienstausfall ist ein Übergang von fehlerfreiem Dienst hin zu fehlerhaftem Dienst".

Definition 5.34: Ein **Fehler** (engl. *error*) liegt vor, wenn einer der Systemzustände nicht korrekt ist und einen nachfolgenden Dienstausfall verursachen kann.

Definition 5.35: „Die festgestellte oder angenommene Fehlerursache wird auch **Störung** (engl. *fault*) genannt. Störungen können sowohl innerhalb wie auch außerhalb eines Systems auftreten."

Einige Störungen verursachen keinen Systemausfall.

Als Beispiel betrachten wir eine transiente **Störung**, die ein Bit im Speicher kippt. Nach diesem Ereignis ist die Speicherzelle **fehlerhaft**. Ein **Systemausfall** wird eintreten, wenn ein Systemdienst von diesem Fehler betroffen ist.

Entsprechend diesen Definitionen verwenden wir den Begriff **Ausfall**raten, wenn wir Systeme betrachten, die nicht die erwartete Funktion zeigen. Wir sprechen immer dann von **Störungen**, wenn wir die zugrunde liegenden **Ursachen**, die einen Ausfall verursachen, betrachten. Es gibt eine große Anzahl möglicher Störungsursachen, von denen einige auf die sinkenden Strukturbreiten von Halbleitern zurückzuführen sind. Fehler werden im Rest des Buches nicht mehr betrachtet.

Das Erreichen der Ausfallrate gemäß SIL-4 ist nur machbar, wenn die Bewertung des Entwurfs auch eine Analyse der Verlässlichkeit, der erwarteten Lebensdauer und verwandter Ziele umfasst. Eine solche Analyse basiert meist auf Ausfallwahrscheinlichkeiten.

Wir betrachten nun die Wahrscheinlichkeitsdichten von Ausfällen genauer. Im Folgenden bezeichne x die Zeit bis zum ersten Ausfall eines Systems. x ist eine Zufallsvariable. Sei $f(x)$ die Dichtefunktion dieser Zufallsvariablen.

Eine sehr häufig benutzte Dichtefunktion ist dabei die Exponentialverteilung mit der Dichtefunktion $f(x) = \lambda e^{-\lambda x}$. Bei dieser Dichtefunktion nimmt die Anzahl der Systemausfälle mit fortschreitender Zeit immer mehr ab. Dies modelliert ein System, bei dem es mit der Zeit immer unwahrscheinlicher wird, dass das System noch arbeitet, und ein nicht mehr arbeitendes System kann nicht ausfallen. Die Ausfallrate (implizit bezogen auf die Anzahl noch funktionierender Systeme) ist bei dieser Dichtefunktion dagegen konstant (siehe Gleichung (5.68)). Daher kann diese Dichtefunktion gut genutzt werden, wenn die (relative) Ausfallrate konstant ist. Aus Gründen der einfachen mathematischen Handhabbarkeit benutzt man diese Dichtefunktion als Referenz auch dann, wenn sie die exakte Situation nicht widerspiegelt oder wenn zunächst eine grobe Übersicht über die Verhältnisse gewünscht wird. Darüber hinaus hat diese Dichtefunktion angenehme mathematische Eigenschaften. Abb. 5.27 (links) zeigt die Dichtefunktion.

In vielen Fällen ist man mehr an den Wahrscheinlichkeiten für die Funktionsfähigkeit des Systems interessiert als an den Dichtefunktionen. Allgemein ergibt sich

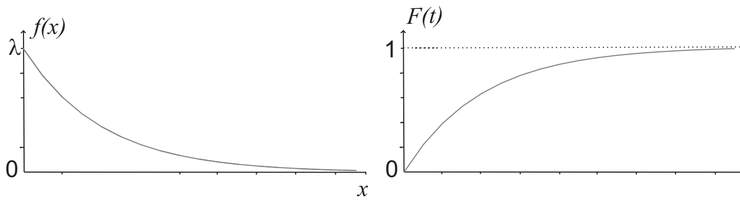


Abb. 5.27 Exponentialverteilung: **links**: Dichtefunktion; **rechts**: Wahrscheinlichkeitsverteilung

die Wahrscheinlichkeit dafür, dass ein System zum Zeitpunkt t fehlerhaft ist, durch Integration über die Dichtefunktion. Die so bestimmte Funktion ist die **Wahrscheinlichkeitsverteilung**:

$$F(t) = Pr(x \leq t) \tag{5.58}$$

$$F(t) = \int_0^t f(x)\mathrm{d}x \tag{5.59}$$

Für die Exponentialverteilung ergibt sich beispielsweise

$$F(t) = \int_0^t \lambda \mathrm{e}^{-\lambda x}\mathrm{d}x = -[\mathrm{e}^{-\lambda x}]_0^t = 1 - \mathrm{e}^{-\lambda t} \tag{5.60}$$

Abb. 5.27 (rechts) zeigt die entsprechende Funktion. Mit fortschreitender Zeit nähert sich die Wahrscheinlichkeit, dass das System fehlerhaft ist, dem Wert 1.

Definition 5.36: Unter der **Zuverlässigkeit** $R(t)$ eines Systems verstehen wir die Wahrscheinlichkeit dafür, dass die Zeit bis zum ersten Fehler größer ist als eine Zeit t:

$$R(t) = Pr(x > t), t \geq 0 \tag{5.61}$$

$$R(t) = \int_t^\infty f(x)\mathrm{d}x \tag{5.62}$$

$$F(t) + R(t) = \int_0^t f(x)\mathrm{d}x + \int_t^\infty f(x)\mathrm{d}x = 1 \tag{5.63}$$

$$R(t) = 1 - F(t) \tag{5.64}$$

$$f(x) = -\frac{\mathrm{d}R(t)}{\mathrm{d}t} \tag{5.65}$$

Für die Exponentialverteilung ergibt sich $R(t) = \mathrm{e}^{-\lambda t}$. Abb. 5.28 zeigt diese Funktion.

Nach einer Zeit $t = 1/\lambda$ beträgt die Wahrscheinlichkeit, dass das System funktioniert, noch ca. 37%.

Definition 5.37: Die **Ausfallrate** $\lambda(t)$ ist die Wahrscheinlichkeit dafür, dass ein System im Zeitintervall zwischen t und $t + \Delta t$ ausfällt.

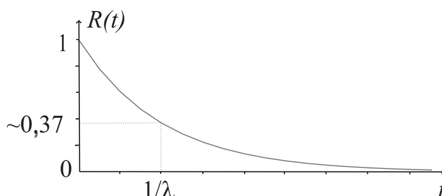


Abb. 5.28 Zuverlässigkeit für die Exponentialverteilung

$$\lambda(t) = \lim_{\Delta t \to 0} \frac{Pr(t < x \leq t + \Delta t | x > t)}{\Delta t} \quad (5.66)$$

Dabei ist $Pr(t < x \leq t + \Delta t | x > t)$ die bedingte Wahrscheinlichkeit dafür, dass das System im Zeitintervall ausfällt, unter der Annahme, dass es zur Zeit t noch funktioniert. Für bedingte Wahrscheinlichkeiten gilt die allgemeine Formel $Pr(A|B) = Pr(AB)/Pr(B)$, wobei $Pr(AB)$ die Wahrscheinlichkeit des Verbundereignisses AB ist. In diesem Fall ist $Pr(B)$ die Wahrscheinlichkeit, dass das System zur Zeit t noch funktioniert, also $R(t)$. Damit folgt aus Gleichung (5.66):

$$\lambda(t) = \lim_{\Delta t \to 0} \frac{F(t + \Delta t) - F(t)}{\Delta t R(t)} = \frac{f(t)}{R(t)} \quad (5.67)$$

Beispielsweise ergibt sich für die Exponentialverteilung[8]:

$$\lambda(t) = \frac{f(t)}{R(t)} = \frac{\lambda e^{-\lambda t}}{e^{-\lambda t}} = \lambda \quad (5.68)$$

Die Ausfallraten werden häufig in der Einheit **FIT** (*Failure unIT*) gemessen. Ein FIT entspricht dabei einem erwarteten Ausfall in 10^9 Stunden.

Die Ausfallraten vieler echter Systeme sind allerdings nicht konstant. Manche folgen der sogenannten „Badewannen“-Kurve (siehe Abb. 5.29). Bei diesem Verhal-

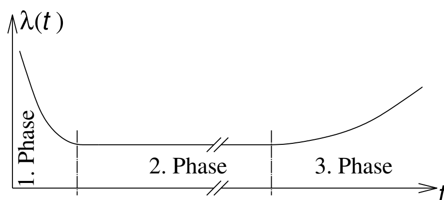


Abb. 5.29 Badewannenkurve der Ausfallraten

ten beginnen wir mit einer zu Anfang größeren Ausfallrate. Diese höhere Rate ist eine Folge von gestörten Fertigungsprozessen oder „Frühausfällen”. Die während des normalen Betriebs auftretende Rate bleibt dann im Wesentlichen konstant. Am Ende der Produktlebenszeit steigt die Rate dann wieder durch Alterungseffekte an.

[8] Dieses Ergebnis legt nahe, dass die Ausfallrate und die Konstante der Exponentialverteilung mit demselben Symbol gekennzeichnet werden.

Definition 5.38: Unter der **mittleren Zeit bis zum Ausfall** (engl. *Mean Time To Failure* (**MTTF**)) verstehen wir die mittlere Zeit bis zu einem Ausfall unter der Annahme, dass das System zunächst funktioniert. Diese Zeit ergibt sich als Erwartungswert der Zufallsvariablen x:

$$\text{MTTF} = E\{x\} = \int_0^\infty x f(x)\mathrm{d}x \tag{5.69}$$

Für die Exponentialverteilung ergibt sich beispielsweise

$$\text{MTTF} = \int_0^\infty x\lambda \mathrm{e}^{-\lambda x}\mathrm{d}x \tag{5.70}$$

Das Integral können wir mit Hilfe der Produktregel ($\int uv' = uv - \int u'v$ mit $u = x$ und $v' = \lambda \mathrm{e}^{-\lambda x}$) bestimmen. Dann ergibt sich aus der Gleichung (5.70):

$$\text{MTTF} = -[x\mathrm{e}^{-\lambda x}]_0^\infty + \int_0^\infty \mathrm{e}^{-\lambda x}\mathrm{d}x \tag{5.71}$$

$$= -\frac{1}{\lambda}[\mathrm{e}^{-\lambda x}]_0^\infty = -\frac{1}{\lambda}[0-1] = \frac{1}{\lambda} \tag{5.72}$$

Bei der Exponentialverteilung ist die mittlere Zeit bis zu einem Ausfall damit gleich dem Kehrwert der Ausfallrate.

Lemma 5.2 (Blacks Gleichung [55, 49]): *Es gilt die folgende empirische Beziehung zwischen der MTTF und den Betriebstemperaturen:*

$$\text{MTTF} = \frac{A}{j_e^n}\mathrm{e}^{\frac{E_a}{k\theta}} \tag{5.73}$$

mit

A : Konstante

j_e : Stromdichte

n : konstant (1..7), umstritten, =2 gemäß Black

E_a : Aktivierungsenergie (z.B. ≈ 0,6 eV)

k : Boltzmann-Konstante (≈ 8,617 * 10¯5 eV/K)

θ : Temperatur

Unabhängig von den Diskussionen um den korrekten Wert von n zeigt diese Gleichung, dass die Temperatur einen exponentiellen Einfluss auf die Lebensdauer des Produkts hat. Auch sind die Stromdichten wichtig: um so größer die Stromdichten, umso kürzer die Lebensdauer des Produkts.

Definition 5.39: Unter der **mittleren Zeit bis zur Reparatur** (engl. *Mean Time To Repair* (**MTTR**)) verstehen wir die mittlere Zeit bis zur Reparatur eines Ausfalls unter der Annahme, dass das System zunächst defekt ist. Die MTTR ist der Erwar-

tungswert der Verteilungsfunktion, welche die für eine Reparatur benötigten Zeiten beschreibt.

Definition 5.40: Unter der **mittleren Zeit zwischen Fehlern** (engl. *Mean Time Between Failures* (**MTBF**)) verstehen wir die mittlere Zeit zwischen zwei Fehlerereignissen.

Die MTBF ergibt sich als Summe der MTTF und der MTTR:

$$\text{MTBF} = \text{MTTF} + \text{MTTR} \tag{5.74}$$

Abb. 5.30 zeigt eine einfache Sicht dieser Gleichung. Die Zeichnung spiegelt dabei nicht wider, dass es sich bei allen Größen um statistische Werte handelt und dass daher konkrete Werte für MTBF, MTTF und MTTR abweichen können. Bei vielen

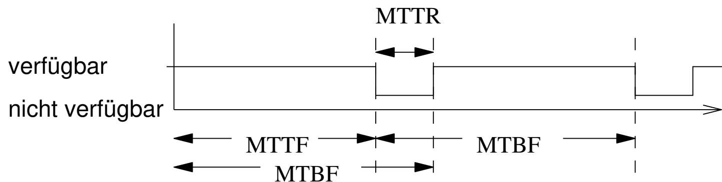


Abb. 5.30 Zur Definition von MTTF, MTTR und MTBF

Systemen werden Reparaturen nicht betrachtet. Auch sollte die MTTR sehr viel kleiner sein als die MTTF. Deswegen werden in vielen Publikationen leider die Begriffe MTBF und MTTF weitgehend synonym benutzt. Beispielsweise kann die Lebensdauer einer Festplatte als MTBF angegeben sein, obwohl sie nie repariert werden wird. Die Angabe dieser Zahl als MTTF wäre korrekter. Die MTTF beschreibt die Verlässlichkeit eines Systems aber immer noch nur sehr grob, insbesondere, wenn die Ausfallraten im Lauf der Zeit große Variationen aufweisen.

Definition 5.41: Unter der **Verfügbarkeit** (engl. *availability*) verstehen wir den Anteil der Zeit an der Gesamtzeit, zu der wir über ein System verfügen können.

Die Verfügbarkeit verändert sich im Laufe der Zeit (denken Sie an die Badewannenkurve!). Daher modellieren wir sie als Funktion $A(t)$. Es wird allerdings häufig nur die Verfügbarkeit $A = lim_{t\to\infty}A(t)$ für große Zeiten betrachtet. Daher definieren wir

$$A = \lim_{t\to\infty} A(t) = \frac{\text{MTTF}}{\text{MTBF}} \tag{5.75}$$

Beispielsweise hätte ein System, dass im Mittel 999 Tage benutzt werden kann und dann jeweils einen Tag lang repariert wird, eine Verfügbarkeit von $A = 0,999$.

Es ist schwierig, tatsächliche Störungsraten zu erhalten. In Abb. 5.31 sind einige der wenigen veröffentlichten Ergebnisse zu sehen [546]. Diese Abbildung enthält die

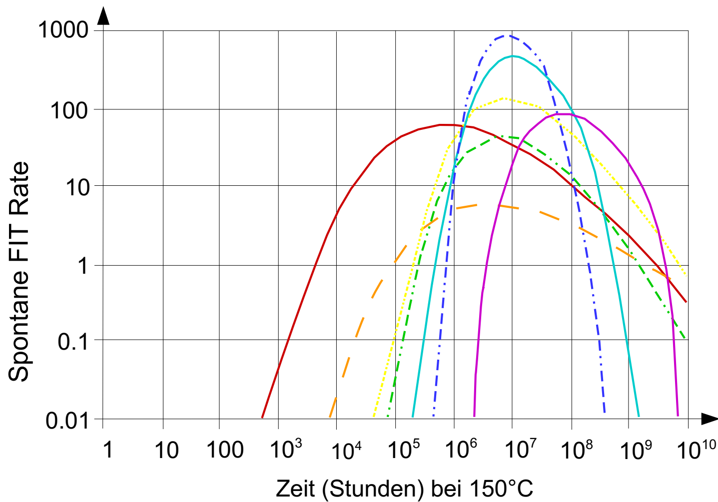


Abb. 5.31 Störungsraten von TriQuint Gallium-Arsenid-Halbleitern (mit freundlicher Genehmigung der TriQuint, Inc., Hillsboro), ©TriQuint

Ausfallraten für verschiedene Gallium-Arsenid (GaAs)-Halbleiter, wobei der „heißeste" Transistor bei einer Temperatur von 150°C betrieben wird. Dieses Beispiel soll hier zeigen, dass es Geräte gibt, für welche die Annahme konstanter Ausfallraten oder eines der Badewannenkurve entsprechenden Verhaltens zu sehr vereinfachend sind[9]. Die Angabe eines einzelnen MTTF-Wertes kann täuschen, vielmehr sollte die tatsächliche Verteilung der Ausfälle über der Zeit angegeben werden. In diesem Beispiel sind die Ausfallraten in den ersten 20 Jahren (174.300 Stunden) der Produktlebenszeit kleiner als 100 FIT, obwohl die Geräte bei hoher Temperatur betrieben werden. Die Angabe von FIT-Werten ist in der Tat sehr temperaturabhängig. Triquint verwendet Temperaturen bis zu 275°C und bekannte Temperaturabhängigkeiten, um Ausfallraten für eine Zeitdauer, welche die Testdauer übersteigt, zu berechnen. Triquint gibt an, dass ihre GaAs-Halbleiter zuverlässiger als gewöhnliche Siliziumhalbleiter sind. Ergebnisse von FIT-Tests sind auch für Xilinx-FPGAs verfügbar (siehe z.B. [599]).

5.6.5 Fehlerbaumanalyse, Fehlermöglichkeits- und Einflussanalyse

Es ist meist nicht machbar, Ausfallraten kompletter Systeme experimentell zu bestimmen. Die geforderten Ausfallraten sind zu gering, Ausfälle können auch untragbar sein. Es ist nicht möglich, 10^5 Flugzeuge für 10^4 Stunden testweise zu fliegen, um zu überprüfen, ob eine Ausfallrate von weniger als 10^{-9} erreicht wird! Der einzi-

[9] Daher wird manchmal die sogenannte **logarithmische Normalverteilung** betrachtet.

ge Ausweg ist die Verwendung einer Kombination der Prüfung von Ausfallraten von Geräten. Eine formale Ableitung aus diesen Angaben ergibt dann Garantien für den zuverlässigen Betrieb des Systems. Ausfälle, die im Entwurf begründet liegen, oder von Benutzern verursachte Ausfälle müssen ebenfalls berücksichtigt werden. Es ist mittlerweile Standard, Entscheidungsdiagramme zur Berechnung der Zuverlässigkeit eines Gesamtsystems aus den Zuverlässigkeitswerten der einzelnen Komponenten einzusetzen [261].

Schäden sind eine Folge von **Risiken** (engl. *hazards*, Möglichkeiten eines Ausfalls). Für jeden Schaden, der durch einen Ausfall verursacht werden kann, gibt es einen Wert für die Schwere des Schadens (die Kosten) und eine entsprechende Wahrscheinlichkeit. Das Risiko kann als Produkt der beiden Werte definiert werden. Informationen über Schäden, die durch Ausfall von Komponenten verursacht werden, können mit verschiedenen Methoden abgeleitet werden [143, 459]:

- **Fehlerbaumanalyse** (engl. *Fault Tree Analysis* (FTA)): Die Fehlerbaumanalyse ist eine *Top-Down*-Methode der Risikoanalyse. Die Analyse beginnt mit einem möglichen Schaden und versucht dann, Szenarien zu finden, die zu diesem Schaden führen. FTA basiert auf der Modellierung einer Booleschen Funktion, die den Betriebszustand des Systems wiedergibt (funktionierend oder nicht funktionierend). Bei der FTA werden Symbole für UND- und ODER-Gatter verwendet. ODER-Gatter werden verwendet, wenn ein einziges Ereignis einen Ausfall erzeugen kann. UND-Gatter werden verwendet, wenn mehrere Ereignisse oder Bedingungen erfüllt sein müssen, damit der Ausfall auftritt. Abbildung 5.32 zeigt ein Beispiel[10].
  FTA basiert auf einem **strukturellen** Modell des Systems, es gibt also die Aufteilung des Systems in Komponenten wieder. Die einfachen UND- und

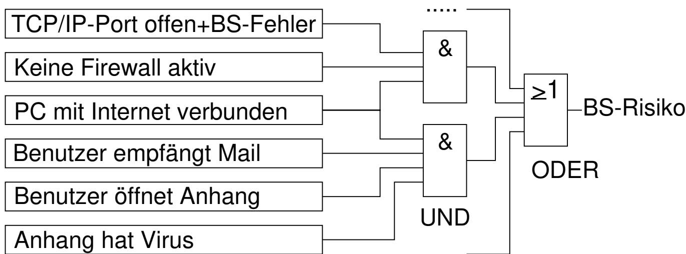


Abb. 5.32 Fehlerbaum

  ODER-Gatter können nicht alle Situationen modellieren. Beispielsweise kann man mit ihnen keine gemeinsam genutzten Ressourcen modellieren, die nur

[10] Entsprechend dem ANSI/IEEE-Standard 91 verwenden wir die Symbole &, =1 und ≥1 zur Kennzeichnung von UND-, XOR- und ODER-Gattern.

in beschränkter Form vorliegen (beispielsweise Energie oder Speicherzellen). Markov-Modelle [67] können verwendet werden, um solche Fälle zu modellieren. Markov-Modelle basieren auf der Darstellung von **Zuständen** und nicht auf der Struktur eines Systems.

- **Fehlermöglichkeits– und Einflussanalyse** (engl. *Failure Mode and Effect Analysis* (FMEA)): FMEA beginnt bei den Komponenten und versucht, deren Zuverlässigkeit abzuschätzen. Mit dieser Information wird die Zuverlässigkeit des Systems aufgrund der Zuverlässigkeit seiner Bestandteile bestimmt (entsprechend einer *Bottom-Up*-Analyse). Der erste Schritt besteht im Aufstellen einer Tabelle der Komponenten, möglicher Störungen, Störungswahrscheinlichkeiten und Auswirkungen auf das Systemverhalten. Die Risiken für das Gesamtsystem werden dann aus dieser Tabelle berechnet. Tabelle 5.3 zeigt ein Beispiel.

Tabelle 5.3 FMEA-Tabelle

Komponente	Störung	Konsequenzen	Wahrscheinlichkeit	Kritisch?
...	...	...	...	...
Prozessor	Metallwanderung	Ausfall	10^{-7} /h	ja
...	...	...	...	...

Es sind für beide Ansätze Werkzeuge verfügbar. Sowohl Fehlerbaumanalyse als auch FMEA können in sogenannten *Safety Cases* eingesetzt werden. Dabei muss eine unabhängige Instanz davon überzeugt werden, dass ein bestimmter technischer Ausrüstungsgegenstand wirklich sicher ist. In diesem Zusammenhang wird häufig gefordert, dass der Ausfall einer einzelnen Komponente nicht zu einer Katastrophe führen darf.

Dieses Buch kann hierzu nur einige Hinweise geben. Es gibt reichhaltige Literatur zum Entwurf von zuverlässigen Systemen, beispielsweise Veröffentlichungen von Huang [224], Zhuo [613] und Pan [445]. Bücher aus diesem Bereich [324, 419, 340, 513, 181] enthalten weitere Informationen.

5.7 Simulation

In diesem Kapitel haben wir uns bislang v.a. mit der Bewertung von Entwürfen beschäftigt. In diesem Abschnitt beginnen wir nunmehr, auch die Validierung von Entwürfen zu betrachten. Validierung ist für jeden Entwurfsvorgang wichtig. Kaum ein System würde wie erwartet funktionieren, wäre es nicht während des Entwurfsprozesses validiert worden. Besonders wichtig ist die Validierung für sicherheitskritische eingebettete Systeme. Theoretisch ließen sich verifizierte Werkzeuge erstellen, die immer korrekte Implementierungen aus einer Spezifikation heraus erstellen. In der Praxis funktioniert diese Verifikation von Werkzeugen aber nur in sehr einfachen Fällen. Als Folge muss jeder einzelne Entwurf validiert werden. Um die Anzahl der

erforderlichen Validierungen eines Entwurfs zu senken, könnten wir versuchen, die Validierung erst zum Ende des Entwurfsvorgangs durchzuführen. Leider funktioniert dieser Ansatz meist auch nicht, da die Unterschiede zwischen den Abstraktionsebenen der Spezifikations- und der Implementierungsbeschreibung groß sind. Daher sind Validierungen im Entwurfsablauf immer wieder erforderlich.

Es wäre vorteilhaft, eine einzelne Validierungstechnik für alle Validierungsprobleme anwenden zu können. In der Praxis löst leider keine der verfügbaren Techniken alle auftretenden Probleme, daher kommt eine Kombination von Techniken zum Einsatz. Simulationen sind eine sehr weitverbreitete Technik für die Validierung von Entwürfen. Eine Simulation besteht aus der Ausführung eines Entwurfsmodells auf einer geeigneten Hardware, üblicherweise auf digitalen Standard-Computern. Bei diesem Ansatz muss das zu validierende Modell ausführbar sein. Alle in Kapitel 2 vorgestellten ausführbaren Sprachen können zur Durchführung von Simulationen verwendet werden, und zwar auf verschiedenen Abstraktionsebenen (wie auf Seite 127 beschrieben). Die Abstraktionsebene, auf der simuliert wird, ist stets ein Kompromiss zwischen der Simulationsgeschwindigkeit und der Simulationsgenauigkeit. Je schneller die Simulation, desto weniger genau ist sie.

Bisher haben wir den Begriff „Verhalten“ im Sinne des funktionalen Verhaltens des Systems (also seiner Ein/Ausgabefunktionalität) verwendet. Es gibt auch Simulationen von anderen, nicht-funktionalen Aspekten von Entwürfen, etwa des thermischen Verhaltens oder der elektromagnetischen Verträglichkeit (engl. *Electro-Magnetic Compatibility* (EMC)) mit anderen elektronischen Geräten.

Durch die Integration der Physik müssen Simulationsmodelle möglicherweise eine große Anzahl physikalischer Effekte beinhalten. Dadurch ist es unmöglich, alle zugehörigen Ansätze zur Simulation cyber-physikalischer Systeme in diesem Buch zu behandeln. Ein Überblick über Ansätze und Themen der Simulation digitaler Systeme ist bei Law [326] zu finden. Es gibt sehr viel weiterführende Literatur zur Simulation von Systemen, insbesondere von heterogenen, cyber-physikalischen Systemen (siehe beispielsweise [363, 126, 442]). Einige Simulatoren sind auf bestimmte Anwendungsgebiete spezialisiert, andere sind breiter einsetzbar.

Für die Validierung cyber-physikalischer Systeme haben Simulationen einige schwerwiegende Einschränkungen:

- Simulationen sind typischerweise viel langsamer als das endgültige System. Wenn man also versucht, den Simulator mit der tatsächlichen Umgebung des eingebetteten Systems zu verbinden, würden viele **verletzte Zeitbedingungen** auftreten.
- Simulationen in der realen Umgebung können sogar **gefährlich** sein - wer würde schon ein Flugzeug mit instabiler, ungetesteter Steuerungssoftware fliegen wollen?
- Viele Anwendungen verwenden sehr große Datenmengen und es ist in der Regel unmöglich, in akzeptabler Zeit eine ausreichende Abdeckung dieser Daten durch Simulation zu erzielen. Multimediaanwendungen sind typische Vertreter für diesen Effekt. So benötigt etwa die Simulation der Kompression eines Videostroms sehr viel Zeit.
- Die meisten praktisch eingesetzten Systeme sind heutzutage zu komplex, um alle möglichen Fälle (Eingaben) simulieren zu können. Daher können Simulationen

zwar helfen, Fehler im Entwurf zu entdecken, sie können aber die Fehlerfreiheit des Systems nicht garantieren. Es ist nicht möglich, alle Kombinationen von Eingaben und internen Zuständen zu simulieren.

Aufgrund dieser Einschränkungen gewinnt die formale Verifikation zunehmend an Bedeutung (siehe Seite 316). Dennoch spielen hochentwickelte Simulationstechniken weiterhin eine wichtige Rolle bei der Validierung von Systemen (siehe z.B. Braun et al. [66]). Es sind sowohl akademische Lösungen wie gem5 (siehe http://gem5.org), SimpleScalar und OpenModelica wie auch kommerzielle Lösungen wie der Synopsys® Virtualizer™ (siehe http://synopsys.com) verfügbar. Netzwerke, wie sie für die Realisierung des Internets der Dinge benötigt werden, können mit verschiedenen Werkzeugen simuliert werden, wie z.B. mit OMNET++ (siehe https://omnetpp.org/).

5.8 *Rapid Prototyping* und Emulation

Simulationen beruhen auf Modellen, die Annäherungen von realen Systemen darstellen. Im Allgemeinen gibt es einige Unterschiede zwischen dem realen System und dem Modell. Wir können diese Lücke verkleinern, indem wir einige Teile des zu entwerfenden Systems präziser als in einem Simulator implementieren (z.B. als reale, physische Komponente).

Definition 5.42: Die Ausführung eines Modells eines Systems, bei der mindestens eine Komponente **nicht** durch eine Simulation auf einem Computer realisiert wird, bezeichnen wir als **Emulation** [384].

Nach M^C^Gregor [384] ist das „Überbrücken der Glaubwürdigkeitslücke nicht der einzige Grund für ein steigendes Interesse an Emulation – die obige Definition eines Emulationsmodells bleibt auch gültig, wenn sie ins Gegenteil verkehrt wird – ein Emulationsmodell ist ein Modell, in dem ein Teil des realen Systems durch ein Modell ersetzt wurde. Die Verwendung von Emulationsmodellen, um Regelungssysteme unter realistischen Bedingungen zu testen, in denen das ... (reale System) ... durch ein Modell ersetzt wird, ist von besonderem Interesse für all jene, die für die Bereitstellung, die Installation und die Inbetriebnahme verschiedenster automatisierter Systeme verantwortlich sind.”

Um die Glaubwürdigkeit noch weiter zu erhöhen, können wir weitere simulierte Komponenten durch reale Komponenten ersetzen. Diese Komponenten müssen dabei nicht die im endgültigen System eingesetzten sein. Sie können auch selbst wieder Näherungen des realen Systems sein, sollten aber präziser als eine Simulation sein.

Es ist mittlerweile üblich, über „Emulation” eines Computers auf einem anderen Computer mit Hilfe von Software zu sprechen. In diesem Zusammenhang fehlt eine präzise Definition des Begriffs „Emulation”. Dennoch passt diese Verwendung zu unserer Definition, da der emulierte Computer nicht nur simuliert wird. Vielmehr wird eine Geschwindigkeit erwartet, die die Simulationsgeschwindigkeit deutlich übersteigt.

Definition 5.43: Die Ausführung eines Modells eines Systems, in dem **keine** Komponente durch eine computerbasierte Simulation realisiert wird, heißt ***Rapid Prototyping***. Dabei werden alle Bestandteile durch realistische Komponenten dargestellt. Einige dieser Komponenten sollten nicht die endgültig verwendeten Komponenten sein, da das System ansonsten dem realen System entspräche.

In vielen Fällen sollten Entwürfe in realistischen Umgebungen getestet werden, bevor die endgültige Version produziert wird. Steuergeräte von Autos sind ein hervorragendes Beispiel hierfür. Diese Systeme sollten von Testfahrern in verschiedenen Umgebungen verwendet werden, bevor sie in die Massenproduktion gehen. Daher entwirft die Automobilindustrie Prototypen. Diese Prototypen sollten sich größtenteils wie die endgültigen Systeme verhalten, sie können dabei aber schwerer sein, mehr Energie verbrauchen oder andere Eigenschaften besitzen, die einen Testfahrer nicht stören. Der Begriff „Prototyp" kann für das gesamte System aus elektrischen und mechanischen Komponenten verwendet werden. Die Unterscheidung zwischen *Rapid Prototyping* und Emulation beginnt aber unscharf zu werden. *Rapid Prototyping* selbst ist ein umfangreiches Themengebiet, das nicht vollständig in diesem Buch behandelt werden kann.

Prototypen und Emulatoren lassen sich beispielsweise mit Hilfe von FPGAs konstruieren. Schaltschränke mit FPGA-Platinen lassen sich für Testfahrten im Kofferraum eines Autos unterbringen. Dabei ist dieser Ansatz nicht auf die Automobilindustrie beschränkt, auch in anderen Fällen werden Prototypen mit FPGAs gebaut. Kommerziell verfügbare **Emulatoren** bestehen aus einer großen Anzahl von FPGAs. Wenn solche Emulatoren verwendet werden, können Experimente mit Systemen durchgeführt werden, die sich „fast" wie die endgültigen Systeme verhalten. Es ist aber bereits bei nicht verteilten Systemen schwierig, Fehler durch Prototypen und Emulation zu finden. Bei verteilten Systemen ist die Lage noch einmal komplizierter (siehe z.B. [547]).

5.9 Formale Verifikation

Die formale Verifikation beschäftigt sich mit formalen mathematischen Beweisen, welche die Korrektheit eines Systems zeigen[11]. Zur Durchführung einer formalen Verifikation wird zuerst ein formales Modell benötigt. Wenn ein Modell erstellt wurde, können damit bestimmte Eigenschaften bewiesen werden. Dieser Schritt ist nur schwer zu automatisieren und erfordert einigen Aufwand. Wenn ein Modell verfügbar ist, können wir versuchen, bestimmte Eigenschaften zu beweisen.

Die Techniken der formalen Verifikation können anhand der verwendeten Logik klassifiziert werden:

- **Aussagen-Logik:** In diesem Fall bestehen die Modelle aus Booleschen Gleichungen. Zugehörige Werkzeuge heißen **Tautologie-*Checker*** oder **Äquivalenz-**

[11] Dieser Text zu formaler Verifikation basiert auf einer Gastvorlesung von Tiziana Margaria-Steffen an der TU Dortmund.

Checker. Sie werden verwendet, um zu entscheiden, ob zwei Darstellungen äquivalent sind oder nicht (es gibt keine unsicheren Fälle). Beispielsweise kann eine Darstellung den Gattern einer realen Schaltung entsprechen, während die andere der Spezifikation entspricht. Der Beweis der Äquivalenz der beiden Darstellungen beweist dann die Korrektheit aller durchgeführter Transformationen (beispielsweise Energie- oder Laufzeitoptimierungen). Tautologie-*Checker* können häufig mit Systemen umgehen, die für eine vollständige simulationsbasierte Validierung zu groß sind. Der Hauptgrund für die Mächtigkeit von neueren Tautologie-*Checkern* liegt in der Verwendung von binären Entscheidungsdiagrammen (engl. *Binary Decision Diagrams* (BDDs)) [570]. Die Komplexität eines BDD-basierten Äquivalenz-*Checkers* für Boolesche Funktionen wächst linear mit der Anzahl der Knoten des BDDs. Die Anzahl der Knoten eines BDDs kann zwar exponentiell mit der Anzahl der Variablen wachsen, aber für viele praktisch relevante Funktionen sind BDDs kompakt, sodass ein effizienter Vergleich möglich ist[12]. Im Gegensatz dazu ist die Äquivalenzprüfung von Funktionen in einer DNF-Darstellung (also als Summe von Produkten) NP-hart. Auf BDDs basierende Äquivalenz-*Checker* haben daher für diese Anwendung Simulatoren verdrängt, sie können mit Schaltungen umgehen, die aus mehreren Millionen Transistoren bestehen.
- Die **Prädikatenlogik erster Stufe** beinhaltet die Quantifizierung mithilfe der Existenz- ($\exists$) und All-Quantoren ($\forall$). Ein gewisses Maß an Automatisierung für die Verifikation durch Prädikatenlogik erster Stufe ist möglich. Da diese Logik im Allgemeinen nicht entscheidbar ist, können unsichere Fälle auftreten.
- **Prädikatenlogik höherer Stufe** (engl. *Higher Order Logic* (HOL)): Prädikatenlogik höherer Stufe erlaubt es, Funktionen so wie andere Objekte zu manipulieren [424]. Bei Verwendung von Logik höherer Ordnung ist die Automatisierung von Beweisen schwierig.

Die Aussagenlogik kann verwendet werden, um zustandslose Logiknetze zu verifizieren, sie kann aber nicht direkt endliche Automaten modellieren. Für kurze Eingabefolgen kann es ausreichen, die Rückkopplung des Automaten aufzutrennen und im Endeffekt mehrere Kopien dieses Automaten zu betrachten, von denen jede die Auswirkung eines Eingabemusters wiedergibt. Diese Methode ist aber nicht sinnvoll auf längere Eingabefolgen anwendbar.

Solche Folgen können mittels **Modellüberprüfung** (engl. *model checking*) verarbeitet werden. Beim *model checking* erhält das Verifikationswerkzeug zwei Eingaben:

1. Ein Modell des zu verifizierenden Systems und
2. die zu überprüfenden Eigenschaften.

Zustände können mit $\exists$ und $\forall$ quantisiert werden, Zahlen jedoch nicht. Verifikationswerkzeuge können die Eigenschaften beweisen oder widerlegen. Für letzteren Fall können sie ein Gegenbeispiel angeben. *Model checking* lässt sich leichter als

[12] Die Multiplikation ist eine prominente Ausnahme [285].

Logik erster Ordnung automatisieren. Es wurde zum ersten Mal im Jahr 1987 unter Verwendung binärer Entscheidungsdiagramme (BDDs) implementiert. *Model checking* war dazu in der Lage, verschiedene Fehler in der Spezifikation des *future bus*-Protokolls zu entdecken [104]. UPPAAL ist ein sehr viel genutztes Werkzeug für das *model checking*[13].

Diese Technik könnte beispielsweise genutzt werden, um Eigenschaften des Modells der Zugbewegungen aus Abb. 2.52 zu beweisen (siehe Seite 91). Es sollte möglich sein, das Petrinetz in ein StateChart umzuwandeln und dann nachzuweisen, dass die Anzahl der Züge zwischen Köln und Paris in der Tat konstant ist, wodurch unsere Diskussion von Ortsinvarianten von Petrinetzen von Seite 89 bestätigt würde.

Weitere Techniken werden von Haubelt und Teich beschrieben [207].

5.10 Aufgaben

Die folgenden Aufgaben sollten entweder zu Hause oder während einer Anwesenheitsphase nach dem *flipped classroom*-Konzept [376] bearbeitet werden:

5.1: Wir betrachten ein Anwendungsbeispiel der Pareto-Optimalität auf der Basis von Ergebnissen der *Task Concurrency Management* (TCM) Werkzeuge des IMEC-Forschungszentrums. In diesem Beispiel werden verschiedene Optionen betrachtet, mit denen ein MPEG-4-Abspielprogramm auf Mehrkern-Prozessoren verteilt werden können [595]. Wong et al. nehmen dabei an, dass eine Kombination von StrongARM-Prozessoren und speziellen Hardwarebeschleunigern benutzt werden soll. Vier Entwürfe erfüllen die Zeitbeschränkung von 30 ms (siehe Tabelle 5.4). Für

Tabelle 5.4 Prozessorkonfigurationen

Prozessorkonfiguration	1	2	3	4
Anzahl schneller Prozessoren	6	5	4	3
Anzahl langsamer Prozessoren	0	3	5	7
Anzahl Prozessoren insgesamt	6	8	9	10

die Kombinationen 1 und 4 erfüllt nur eine Abbildung von Tasks auf Prozessoren die Zeitbeschränkung. Für die Kombinationen 2 und 3 gibt es verschiedene zulässige Task/Prozessorzuordnungen. Diese werden in Abb. 5.33 gezeigt. Welche Fläche im Raum der Zielkriterien ist durch mindestens eine Task/Prozessorzuordnung der Konfiguration 3 dominiert? Gibt es eine Task/Prozessorzuordnung der Konfiguration 2, die nicht durch eine Task/Prozessorzuordnung der Konfiguration 3 dominiert wird? Welche Fläche im Raum der Zielkriterien dominiert mindestens eine Task/Prozessorzuordnung der Konfiguration 3?

[13] Siehe http://www.uppaal.org für die akademische und http://www.uppaal.com für die kommerzielle Version.

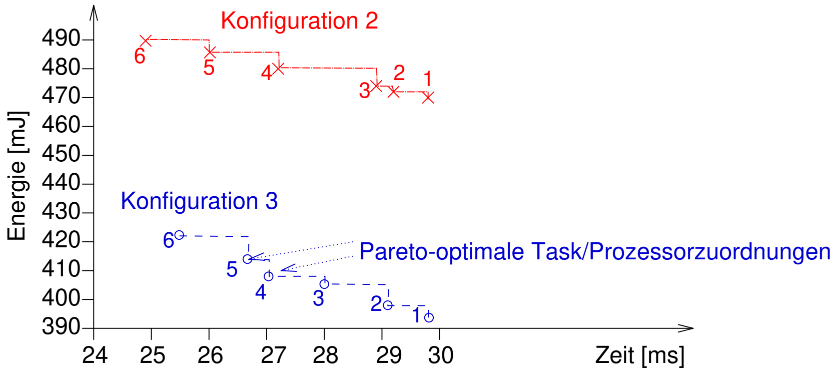


Abb. 5.33 Pareto-Punkte für Konfigurationen 2 und 3

5.2: Welche Anforderungen gibt es für die Berechnung der geschätzten größtmöglichen Ausführungszeit (WCET_{EST})?

5.3: Wir betrachten abstrakte Cachezustände bei rekonvergenten Programmpfaden. Abb. 5.34 zeigt die abstrakten Zustände vor der Verschmelzung. Nunmehr betrachten

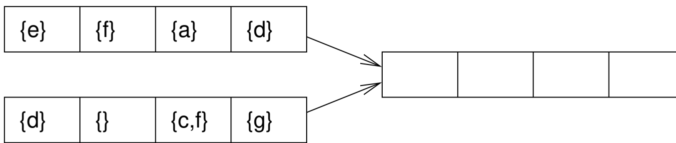


Abb. 5.34 Abstrakte Cachezustände

wir die abstrakten Zustände nach der Verschmelzung. Welcher neue Zustand ergibt sich aufgrund einer *must*-Analyse? Welcher neue Zustand ergibt sich aufgrund einer *may*-Analyse?

5.4: Betrachten Sie einen eingehenden Ereignisstrom mit *Bursts*! Der Strom ist periodisch mit einer Periode von T. Zu Beginn einer jeder Periode gehen zwei Ereignisse im Abstand von d Zeiteinheiten ein. Entwickeln Sie Ankunftskurven für diesen Strom! Die entstehenden Graphen sollen die Zeiten von 0 bis $3*T$ darstellen.

5.5: Angenommen, wir haben einen Prozessor mit einer maximalen Performanz von b.

1. Wie sehen die *service curves* aus, wenn die Performanz aufgrund von Cachekonflikten auf b' absinken kann?
2. Wie ändern sich die *service curves*, wenn ein Zeitgeber das ausgeführte Programm alle 100 ms unterbricht und wenn die Unterbrechung jeweils 10 ms andauert? Wir nehmen an, dass es keine Cachekonflikte gibt.

3. Wie ändern sich die *service curves*, wenn Sie Cachekonflikte wie unter (1.) und Unterbrechungen wie unter (2.) betrachten?

Die erzeugten Graphen sollten die Zeiten von 0 bis 300 ms darstellen.

5.6: Stellen Sie sich vor, wir wollen Bernstein sammeln. Allerdings gibt es das Risiko, dass wir stattdessen weißen Phosphor sammeln. Angenommen, wir haben 50 Objekte gesammelt und wir lassen sie zunächst alle im Wasser, um das Risiko eines Feuers zu vermeiden. Wir klassifizieren 30 Objekte als Bernstein und 20 als weißen Phosphor. Tatsächlich sind zwei der Objekte, die wir für Bernstein halten, weißer Phosphor und acht Objekte, die wir für Phosphor halten, sind tatsächlich Bernstein. Berechnen Sie die Genauigkeit (engl. *precision*), Sensitivität (engl. *sensivity*), Korrektklassifikationsrate (engl. *accuracy*) und Spezifizität (engl. *specificity*) unserer Klassifikation!

5.7: Angenommen, Sie wollen die Leistungsaufnahme Ihres Mobiltelefons mit einem *Shunt*-Widerstand messen. Die folgenden Werte sind für die Bestimmung der Leistungsaufnahme zum Zeitpunkt t relevant: Widerstand: 0,47 Ohm, Spannung des Netzteils: 5,1 V, Spannung über dem Widerstand: 0,23 V. Wie groß ist die Leistungsaufnahme zur Zeit t?

5.8: Gegeben sei eine Kupferplatte der Fläche A=10 cm^2 und Dicke 5 mm. Wie viel thermische Leistung fließt zwischen den Enden der Platte, wenn die Temperaturdifferenz zwischen den beiden Seiten 10 °C beträgt?

5.9: Wir betrachten eine Menge von Plattenlaufwerken, von denen die Hälfte nach 5000 Betriebsstunden ausgefallen ist. Wir nehmen an, dass die Ausfälle einer exponentiellen Verteilung folgen. Wie groß ist das λ dieser Verteilung?

Open Access Dieses Kapitel wird unter der Creative Commons Namensnennung 4.0 International Lizenz (http://creativecommons.org/licenses/by/4.0/deed.de) veröffentlicht, welche die Nutzung, Vervielfältigung, Bearbeitung, Verbreitung und Wiedergabe in jeglichem Medium und Format erlaubt, sofern Sie den/die ursprünglichen Autor(en) und die Quelle ordnungsgemäß nennen, einen Link zur Creative Commons Lizenz beifügen und angeben, ob Änderungen vorgenommen wurden.

Die in diesem Kapitel enthaltenen Bilder und sonstiges Drittmaterial unterliegen ebenfalls der genannten Creative Commons Lizenz, sofern sich aus der Abbildungslegende nichts anderes ergibt. Sofern das betreffende Material nicht unter der genannten Creative Commons Lizenz steht und die betreffende Handlung nicht nach gesetzlichen Vorschriften erlaubt ist, ist für die oben aufgeführten Weiterverwendungen des Materials die Einwilligung des jeweiligen Rechteinhabers einzuholen.

Kapitel 6
Abbildung von Anwendungen

Ein sehr wichtiger Schritt im Entwurfsprozess ist die Abbildung von Anwendungen auf verfügbare Hardware-Plattformen. Dazu müssen wir entscheiden, auf welchem Prozessor Anwendungen (oder Teile davon) ausgeführt werden und wie sie zeitlich eingeplant werden. Dabei sollten so viele Entscheidungen wie möglich zur Entwurfszeit getroffen werden, um Garantien für das Zeitverhalten geben zu können. Dies ist mit geeigneten *Scheduling*-Verfahren möglich. In diesem Kapitel geben wir eine Übersicht über statische Verfahren und wir klassifizieren diese anhand der Triplet-Notation von Pinedo und anderen. Wir starten dazu mit einer Vorstellung von klassischen Verfahren für Einzelprozessoren sowohl für aperiodische wie auch für periodische Tasksysteme, wobei wir u.a. die bekannten Verfahren *Earliest Deadline First* (EDF) und *Rate Monotonic Scheduling* (RMS) vorstellen. Nach einem Hinweis auf die Benutzung von *bin-packing*-Algorithmen für homogene Mehrprozessor-Systeme gehen wir auf die Betrachtung von heterogenen Mehrprozessor-Systemen über. Wir stellen jeweils Algorithmen für unabhängige und abhängige Jobs vor. Dabei zeigt sich, dass für abhängige Jobs v.a. heuristische Verfahren in Frage kommen. Das Kapitel schließt mit Hinweisen auf dynamische Verfahren.

6.1 Problemdefinition

6.1.1 Präzisierung des Entwurfsproblems

Die skizzierte Abbildung auf Hardware-Plattformen ist auch Teil des vereinfachten Entwurfsflusses in Abb. 6.1.

Die in Frage kommenden *Scheduling*-Verfahren sollten es möglich machen, ein System mit einer bestimmten Kombination von Anwendungen zu betreiben. So erwarten wir beispielsweise von einem Mobiltelefon, dass wir telefonieren können, während der *Bluetooth*-*Stack* gleichzeitig die Audiosignale an einen Kopfhörer überträgt und wir Informationen im Adressbuch nachschlagen. Parallel dazu könnte auch

© Der/die Autor(en) 2021

P. Marwedel, *Eingebettete Systeme*, https://doi.org/10.1007/978-3-658-33437-6_6

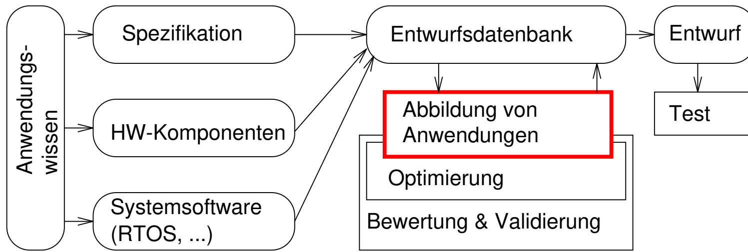


Abb. 6.1 Vereinfachter Entwurfsfluss

noch eine Dateiübertragung oder sogar eine Videoverbindung aktiv sein. Wir müssen sicherstellen, dass all diese Anwendungen zusammen verwendet werden können und dass dabei die Echtzeitbedingungen eingehalten werden (keine Tonstörungen beim Telefonieren). Dies lässt sich durch eine Analyse der Anwendungsfälle erreichen.

Eine besondere Eigenschaft von eingebetteten und cyber-physikalischen Systemen ist, dass Hardware und Software gleichermaßen während des Entwurfs berücksichtigt werden müssen. Daher wird diese Art des Entwurfs auch als *Hardware/Software-Codesign* bezeichnet. Das generelle Ziel ist hier, die richtige Kombination aus Hardware und Software zu finden, damit das effizienteste Produkt entsteht, das der Spezifikation genügt. Deshalb können eingebettete Systeme nicht durch ein Syntheseverfahren entworfen werden, welches nur eine Verhaltensbeschreibung betrachtet, vielmehr müssen die vorhandenen Komponenten mit in das Verfahren einbezogen werden. Es gibt noch weitere Gründe für diese Beschränkung: um die stetig steigende Komplexität eingebetteter Systeme beherrschen zu können und immer kürzere Entwicklungszyklen zu realisieren, ist Wiederverwendung ein unerlässliches Grundprinzip. Dieser Ansatz führte zum Begriff **plattformbasierter Entwurf**:

„Eine Plattform ist eine Familie von Architekturen, die eine Menge von Bedingungen erfüllen, welche eingeführt wurden, um die Wiederverwendung von Hardware- und Softwarekomponenten zu ermöglichen. Dabei reicht eine Hardwareplattform alleine nicht aus. Ein schneller, zuverlässiger, ableitender[1] Entwurf erfordert die Verwendung einer Anwendungsprogrammierschnittstelle (API) für die Plattform, um die Plattform in Richtung der Anwendungssoftware erweitern zu können. Allgemein ist eine Plattform eine Abstraktionsebene, die viele der möglichen Details einer unteren Ebene verdeckt. Plattformbasierter Entwurf ist eine Kompromisslösung: im *top-down* Entwurfsfluss bilden Entwickler eine Instanz der oberen Plattform auf eine Instanz der unteren Plattform ab und propagieren dabei die Entwurfsbeschränkungen” [476]. Die Abbildung ist ein iterativer Vorgang, in dem Werkzeuge zur Performanz-Bewertung die jeweils nächste durchzuführende Aufgabe bestimmen.

In diesem Buch liegt der Schwerpunkt auf dem Entwurf eingebetteter Systeme basierend auf vorhandenen Ausführungsplattformen. Dies spiegelt die Tatsache wider, dass viele moderne Systeme auf bereits existierenden Plattformen aufbauen.

[1] Gemeint ist: aus der Spezifikation ableitend.

Von den in diesem Buch beschriebenen Techniken abweichende Methoden müssen in Fällen eingesetzt werden, in denen zusätzlich auch die Ausführungsplattform entworfen werden soll. Aufgrund unseres Schwerpunktes betrachten wir die **Abbildung von Anwendungen auf Ausführungsplattformen** als das **maßgebliche Entwurfsproblem**.

Auch für den plattformbasierten Entwurf kann es eine Anzahl von Entwurfsoptionen geben. So könnten wir zwischen verschiedenen Varianten einer Plattform auswählen, bei denen jede Variante über eine unterschiedliche Anzahl an Prozessoren, unterschiedliche Geschwindigkeiten oder eine andere Kommunikationsarchitektur verfügt. Zudem könnte es verschiedene anwendbare *Scheduling*-Verfahren geben. Daher müssen die jeweils passenden Optionen ausgewählt werden.

Damit kommen wir zur folgenden Definition des Abbildungsproblems [535]:
Gegeben sind:

- eine Menge von Anwendungen,
- Anwendungsfälle, welche die Verwendung der Anwendungen beschreiben und
- eine Menge an möglichen Architekturen:
  - (möglicherweise heterogene) Prozessoren,
  - (möglicherweise heterogene) Kommunikationsarchitekturen und
  - verschiedene mögliche *Scheduling*-Verfahren.

Finde nun:

- Eine Abbildung von Anwendungen auf Prozessoren,
- dazu passende *Scheduling*-Verfahren (wenn nicht festgelegt) und
- eine Zielarchitektur (wenn nicht vorab festgelegt).

Zielsetzungen sind dabei:

- Einhalten von *Deadlines* und/oder Maximierung der Performanz sowie
- Minimierung der Kosten, des Energieverbrauchs und möglicherweise weitere Ziele.

Die Erforschung der möglichen Architekturoptionen wird als **Erkundung des Entwurfsraumes** (engl. *Design Space Exploration* (DSE)) bezeichnet. Dabei kann eine vollständig festgelegte Plattformarchitektur als Spezialfall betrachtet werden.

Ein Beispiel ist der Entwurf eines AUTOSAR-basierten automotiven Systems: In AUTOSAR [27] kommen eine Anzahl heterogener Ausführungseinheiten oder Steuergeräte (engl. *Embedded Control Units* (ECUs)) und eine Anzahl von Softwarekomponenten zum Einsatz. Die Frage, die sich nun stellt, lautet: Wie können die einzelnen Softwarekomponenten auf die Steuergeräte abgebildet werden, sodass alle Echtzeitbedingungen eingehalten werden? Dabei möchten wir eine minimale Anzahl an Steuergeräten verwenden.

Bei eingebetteten Systemen können wir annehmen, dass die Anwendungen eine Anzahl von Tasks enthalten, die wiederholt ausführungsbereit werden. Mit diesen Tasks können wir den auszuführenden Softwarecode assoziieren. Beispielsweise kann es notwendig sein, einen bestimmten Code genau einmal für jeden Eingabewert

auszuführen. Jede einzelne Task bezeichnen wir als τ_i und die Menge aller Tasks als $\tau = \{\tau_1, ..., \tau_n\}$.

Definition 6.1: Jede Ausführung einer Task heißt ein **Job** (siehe auch Definition 4.4). Zu jeder Task τ_i gehört eine Menge $J(\tau_i)$ von Jobs. Aufgrund der wiederholten Ausführungen ist die Menge der Jobs von Task τ_i möglicherweise nicht endlich.

Definition 6.2: Tasks τ_i, die alle T Zeiteinheiten ausführungsbereit werden, heißen **periodische Tasks** mit der **Periode** T.

Definition 6.3: Eine Task τ_i heißt **sporadisch**, wenn es eine untere Schranke für den Abstand der Zeitpunkte gibt, zu denen die Task ausführungsbereit wird. Für jede sporadische Task τ_i nennen wir diese Schranke auch T_i.

Diese Schranke ist wichtig: ohne eine solche Schranke könnten die Ankunftskurven für jedes Δ unbeschränkt werden. Es wäre dann nicht möglich, für eine beschränkte Menge an Ressourcen ein *Schedule* zu finden.

Definition 6.4: Tasks, die weder periodisch noch sporadisch sind, heißen **aperiodisch**.

Das Konzept der Hyperperioden ist für periodische und sporadische Task-Systeme nützlich:

Definition 6.5: Sei τ ein periodisches oder sporadisches Task-System. Seine **Hyperperiode** ist definiert als das kleinste gemeinsame Vielfache (KGV) der Perioden der einzelnen Tasks.

Wenn für eine Menge von Tasks für eine Hyperperiode ein *Schedule* existiert, dann existiert es aufgrund der wiederholten identischen *Scheduling*-Aufgaben auch für alle Hyperperioden.

6.1.2 Typen von Scheduling-Problemen

Im verbleibenden Teil dieses Kapitels wird die nachfolgende Notation für Jobs benutzt: Sei $J = \{J_i\}$ eine Menge von Jobs. Sei ferner (siehe Abb. 6.2):

- r_i der Zeitpunkt, an dem J_i ausführungsbereit wird (engl. *release time*),
- C_i die **größtmögliche Ausführungszeit** (engl. *Worst Case Execution Time* (WCET)) von J_i,
- d_i die **(absolute)** ***Deadline*** von J_i,
- D_i die **relative** ***Deadline***, d.h. die Zeit zwischen r_i und der Zeit, zu der J_i seine Ausführung beendet haben soll ($D_i = d_i - r_i$),
- l_i der **Schlupf** (engl. *laxity* oder *slack*), definiert als

$$l_i = D_i - C_i \tag{6.1}$$

  (wenn $l_i = 0$ ist, dann muss J_i sofort ausgeführt werden, nachdem er ausführungsbereit ist),

- s_i die Zeit, zu der die Ausführung von J_i startet,
- f_i die Zeit, zu der die Ausführung von J_i beendet wird (engl. *finishing time*).

In Abbildungen wie in Abb. 6.2 bedeuten nach oben zeigende Pfeile die Zeit, zu der Jobs ausführungsbereit werden und nach unten zeigende Pfeile die *Deadline* der Jobs.

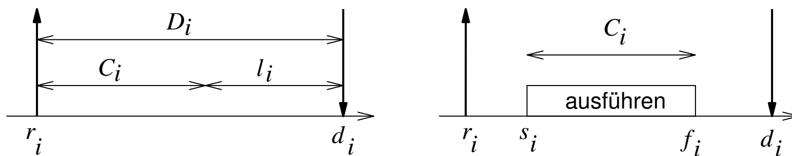


Abb. 6.2 Notation für Jobs

Klassifizieren werden wir *Scheduling*-Probleme im Folgenden gemäß der Triplet-Notation, die von Pinedo vorgestellt wurde [455] und die auf einer Notation basiert, die ursprünglich von Graham, Lawler, Lenstra und Kan vorgeschlagen wurde [190]. Diese Notation nutzt das folgende Triplet:

$$(\alpha|\beta|\gamma). \tag{6.2}$$

Das α-Feld

Das α-Feld beschreibt die Maschine, auf der Jobs ausgeführt werden sollen. Einfache *Scheduling*-Algorithmen behandeln den Fall eines einzelnen Prozessors, wohingegen komplexe Algorithmen auch Systeme mit mehreren Prozessoren behandeln können. In diesem Buch betrachten wir die folgenden Werte für das α-Feld:

- Ein Wert von 1 zeigt einen einzelnen Prozessor an.
- Ein Wert *Pm* zeigt an, dass es *m* Prozessoren gibt, die parallel benutzt werden können. Jeder Job kann mit derselben Ausführungszeit auf jedem der *m* Prozessoren ausgeführt werden. Man nennt die Prozessoren in diesem Fall *identisch* oder **homogen**. Das β-Feld kann benutzt werden, um die möglichen Zuordnungen von Jobs zu Prozessoren einzuschränken.
- Ein Wert von *Qm* bezeichnet parallele Prozessoren mit unterschiedlichen Performanzen (Rechenleistungen). Die Performanz jedes Prozessors wird durch einen Skalierungsfaktor relativ zum langsamsten der Prozessoren angegeben. Alle Skalierungsfaktoren zusammen werden durch einen Vektor $(s_1, .., s_m)$ ausgedrückt, wobei s_k der Skalierungsfaktor für Prozessor π_k ist. In diesem Fall heißen die Prozessoren **uniform**. Das uniforme Prozessormodell ist stark vereinfacht, wir werden uns kaum darauf beziehen.
- Ein Wert von *Rm* zeigt an, dass es *m* Prozessoren mit unterschiedlichen Ausführungszeiten gibt. Die Ausführungszeit von Job oder Task *i* auf Prozessor *k* ist

$C_{i,k}$. Diese Prozessoren heißen **heterogen.** Heterogene Prozessoren können für bestimmte Zielkriterien optimiert sein, beispielsweise für einen geringen Energieverbrauch oder für eine hohe Rechenleistung. Daher sind heterogene Prozessoren für eingebettete Systeme sehr wichtig. Als Sonderfall können auch spezielle Hardware-Beschleuniger modelliert werden.

Das α-Feld besteht immer nur aus einer Komponente.

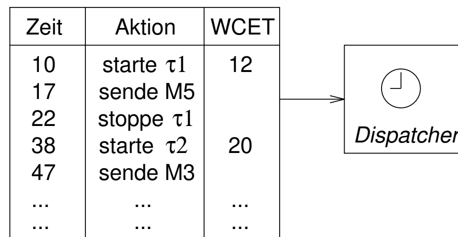
Das β-Feld

Das β-Feld beschreibt Beschränkungen für die Verarbeitung. Das β-Feld kann mehrere Komponenten enthalten. In diesem Buch betrachten wir die folgenden Werte für dieses Feld:

- Ein Eintrag r_i bezeichnet existierende *release times*, d.h. Zeiten, an denen Job i ausführungsbereit wird.
- Ein Eintrag *prmp* bedeutet, dass Verdrängung (engl. *preemption*) erlaubt ist. Wenn dieser Eintrag fehlt, wird angenommen, dass keine Verdrängung möglich ist. Nicht verdrängendes *Scheduling* basiert auf der Annahme, dass Jobs ausgeführt werden, bis sie beendet sind. Daher kann die Reaktionszeit[2] für externe Ereignisse sehr groß sein, wenn manche Jobs eine lange Ausführungszeit haben. Verdrängende *Scheduler* müssen benutzt werden, wenn manche Jobs eine lange Ausführungszeit haben oder wenn die Antwortzeit für externe Ereignisse sehr kurz sein muss. Allerdings kann *preemption* zu unvorhersehbaren Ausführungszeiten für die verdrängten Jobs führen. Daher kann es erforderlich sein, Verdrängungen zu beschränken, damit Jobs mit harten *Deadlines* ihre Zeitschranke einhalten.
- Ein anderer möglicher Eintrag bezieht sich auf die Art der Zeitschranken. Wir unterscheiden zwischen **weichen** und **harten** Zeitschranken, siehe Definition 1.8 auf Seite 11.
  Scheduling für weiche Zeitschranken basiert häufig auf Erweiterungen von Standard-Betriebssystemen. Wir werden solche Systeme in unserem Buch nicht weiter betrachten. Daher ist unsere Standardannahme, dass wir harte Zeitschranken haben.
- Einträge *periodic* und *sporadic* können beschreiben, dass wir es mit periodischen bzw. sporadischen Task-Systemen zu tun haben.
- Ein Eintrag *prec* drückt aus, dass Präzedenzrelationen (engl. *precedence constraints*) existieren. Präzedenzrelationen bewirken, dass Jobs gemäß einer bestimmten partiellen Ordnung ausgeführt werden müssen. Eine Ursache dafür kann in der Kommunikation zwischen Jobs liegen. Für eingebettete Systeme sind Präzedenzrelationen eher die Regel als eine Ausnahme.
- Für sporadische und periodische Tasks unterscheiden wir *Scheduling*-Probleme häufig anhand der *Deadlines*:

[2] Das ist die Zeit vom Auftreten des externen Ereignisses bis zum Abschluss der benötigten Reaktion.

Wir sprechen von Tasks mit **impliziten *Deadlines*** (engl. *implicit deadline tasks*), wenn $D_i = T_i$ für alle i gilt. Wir nennen diese Tasks dann auch ***Liu-and-Layland (L & L) tasks*** [348]. Dieser Fall wird durch einen Eintrag $D_i = T_i$ gekennzeichnet. Wir sprechen von Tasks mit **beschränkten *Deadlines*** (engl. *constrained deadline tasks*) wenn $D_i \leq T_i$ für alle i gilt.
Wenn es hinsichtlich der *Deadlines* keine Beschränkungen gibt, sprechen wir von Mengen von Tasks mit **beliebigen *Deadlines*** (engl. *arbitrary deadline tasks*). Diese Information hinsichtlich der *Deadlines* wird ebenfalls durch eine entsprechende Komponente im β-Feld kenntlich gemacht.
Wir können dieses Feld auch benutzen, um zu beschreiben, welche Form von *Scheduling* benutzt werden soll. Beispielsweise können wir Komponenten *fixed-job-prio* bzw. *fixed-task-prio* verwenden, um auszudrücken, dass Jobs bzw. Tasks eine feste Priorität haben sollen.
Außerdem könnten wir zwischen statischem und dynamischem *Scheduling* unterscheiden. Dynamische *Scheduler* treffen Entscheidungen zur Laufzeit. Sie sind sehr flexibel, aber sie erzeugen zur Laufzeit *Overhead*. Außerdem sind ihnen globale Zusammenhänge wie der Ressourcenbedarf und Reihenfolgebeschränkungen in der Regel nicht bekannt. Bei eingebetteten Systemen sind solche globalen Zusammenhänge meist verfügbar und sie sollten ausgenutzt werden.
Statische *Schedules* nutzen Entscheidungen zur Entwurfszeit. Sie basieren darauf, dass die Startzeiten von Jobs vorab geplant werden und dass Tabellen mit den Startzeiten einem einfachen *Dispatcher* mitgeteilt werden. Der *Dispatcher* trifft keine Entscheidungen, sondern startet lediglich die Jobs entsprechend der Zeiten, die in der Tabelle eingetragen sind. Der *Dispatcher* kann durch einen Zeitgeber getriggert werden, der ihn die Tabelle analysieren lässt. Systeme, die vollständig durch einen Zeitgeber kontrolliert werden, heißen **vollständig zeitgesteuert** (engl. *entirely time triggered* (TT *systems*)). Derartige Systeme werden im Buch von Kopetz [304] detailliert vorgestellt.
„In einem vollständig zeitgesteuerten System wird der zeitliche Ablauf aller Tasks **vorab** durch *off-line* Werkzeuge geplant. Dieser zeitliche Ablauf wird gespeichert in einer *Task-Descriptor List* (TDL), die den zyklischen Ablauf aller Aktivitäten ... enthält (siehe Abb. 6.3). Der Ablauf beachtet Reihenfolgebedingungen und den gegenseitigen Ausschluss zwischen Tasks, sodass eine explizite Koordination der Tasks durch das Betriebssystem zur Laufzeit nicht notwendig ist. Der *Dispatcher* wird durch synchronisierte Taktsignale aktiviert. Er sieht sich die TDL an und führt dann die Aktion aus, die für diesen Zeitpunkt geplant ist".



Zeit	Aktion	WCET
10	starte τ1	12
17	sende M5	
22	stoppe τ1	
38	starte τ2	20
47	sende M3	
...	...	...
...	...	...

Abb. 6.3 TDL in einem zeitgesteuerten System

Der Hauptvorteil des statischen *Schedulings* ist, dass man einfach prüfen kann, ob die Zeitschranken eingehalten sind. „Die Vorhersagbarkeit des Systemver-

haltens ist der wichtigste Aspekt, um Zeitschranken in harten Echtzeitsystemen einzuhalten; *Scheduling* vor der Ausführungszeit ist häufig die einzige praktische Methode, um in einem komplexen System Vorhersagbarkeit zu bieten" [604]. Der Hauptnachteil ist, dass die Antwortzeit auf Ereignisse recht schlecht sein kann.

- *Scheduling*-Algorithmen für Mehrprozessor-Systeme können entweder lokal auf einem Prozessor ausgeführt werden oder unter einer Menge von Prozessoren aufgeteilt werden. Wir können daher zwischen **zentralisiertem** und **verteiltem** *Scheduling* unterscheiden. Diese Unterscheidung könnte auch im β-Feld ausgedrückt werden.

Das γ-Feld

Das γ-Feld beschreibt die Zielfunktion. In diesem Buch betrachten wir die folgenden Werte für dieses Feld:

- Ein Eintrag L_{max} bedeutet, dass die maximale Verspätung zu minimieren ist.

  Definition 6.6: Die **maximale Verspätung** (engl. *maximum lateness*) ist definiert als die Differenz zwischen dem Berechnungsende f_i und der *Deadline* d_i, maximiert über alle Tasks i.

  Die maximale Verspätung ist negativ, wenn alle Tasks ihre Berechnungen vor ihrer *Deadline* beenden.
- Ein Eintrag MS_{max} bezeichnet die Minimierung des *Makespan*, d.h. der Zeit, zu der die letzte Ausführung beendet ist.

  Definition 6.7: Der ***Makespan*** ist definiert als[3]

$$MS_{max} = max_i(f_i) \tag{6.3}$$

- Zusätzlich zu den Einträgen, die Pinedo betrachtet, sind auch weitere Einträge für eingebettete Systeme relevant. Beispielsweise möchten wir vielleicht den Energieverbrauch minimieren oder wir möchten *Tradeoffs* (also z.B. Pareto-Kurven) zwischen verschiedenen Bewertungskriterien betrachten.

Es gibt eine riesige Menge an *Scheduling*-Algorithmen und eine umfassende Behandlung existierender Algorithmen wäre selbst dann nicht möglich, wenn ein vollständiges Buch oder ein ganzer Kurs zur Verfügung stünden. In einem üblichen Bachelorstudium gibt es üblicherweise nicht ausreichend Stundenvolumen, um einen Kurs vollständig dem *Scheduling* zu widmen (aber das kann für ein Masterstudium anders sein). Deshalb geben wir hier nur eine kurze Einführung in das Thema. Viele *Scheduling*-Algorithmen sind sehr komplex [38, 455] und häufig können nur annähernd optimale Ablaufpläne garantiert werden. Das wirft die Frage auf, welche Algorithmen hier präsentiert werden sollten. Wir werden in diesem Kapitel eine

[3] Pinedo bezeichnet den *Makespan* als C_{max}. Mit der Bezeichnung MS_{max} möchten wir eine Verwechslung mit Ausführungszeiten vermeiden.

Auswahl an Algorithmen präsentieren, die in eingebetteten Systemen häufig benutzt werden. Eine Übersicht über die ausgewählten Algorithmen ist in Tabelle 6.1 zu sehen. Von links nach rechts betrachtet, beziehen sich die Spalten auf das Prozes-

Tabelle 6.1 Scheduling-Algorithmen dieses Kapitels

α	β							γ	Ab-	Algorithmus
Proc.	r_i	*prmp*	*prec*	*periodic*[a]	D_i	prio	glob	Zielfkt.	schnitt	
1	-	-	-	-				L_{max}	6.2.1	*Earliest Due Date*
1	X	X	-	-				L_{max}	6.2.1	*Earliest Deadline First*
1	X	X	-	-				L_{max}	6.2.1	*Least Laxity*
1	X	-	-	-				L_{max}	6.2.1	(Theorem 6.3)
1	X	X	X	-		Job		L_{max}	6.2.2	*Latest Deadline First*
1	X	-	X	-				L_{max}	6.2.2	Spring OS [508]
1	X	X	-	X	$= T_i$	Task		$\leq D_i$	6.2.3	*Rate Monotonic*
1	X	X	-	X	$\neq T_i$	Task		$\leq D_i$	6.2.3	*Deadline Monotonic*
Pm	-	-	-	-			X	$m = \|\pi\|$	6.3.1	*Bin Packing*
Pm	-	-	-	-			X	$\sum b_i$	6.3.1	0/1 *Multi-Knapsack*
Pm	X	X	-	X	$= T_i$			$\leq D_i$	6.3.1	*First Fit Decreasing*
Pm	X	X	-	X	$= T_i$	Job	X	$\leq D_i$	6.3.2	Pfair
Pm	X	X	-	-		Job	X	$\leq D_i$	6.3.3	G-EDF, fpEDF, EDZL
Pm	X	X	-	X	$= T_i$	Task	X	$\leq D_i$	6.3.4	G-RM, RM-US, RMZL
Pm	X	X	-	X	$\neq T_i$	Task	X	$\leq D_i$	6.3.4	Dichte-basiert
Pm	-	-	X	-				MS_{max}	6.4	ASAP, ALAP
Rm[b]	-	-	X	-				MS_{max}	6.4.3	*List Scheduling*
Pm	-	-	X	-				MS_{max}	6.4.4	Ganzz.Lin.Progr.(ILP)
Rm	-	-	X	-				MS_{max}	6.5.2	HEFT, CPOP
Rm	-	-	X	-				MS_{max}	6.5.3	ILP, z.B. [362]
Rm	X	X	X	-			(X)	verschiedene	6.5.4	DOL, HOPES, MAPS,

[a] Algorithmen für aperiodische Task-Mengen können auch für periodische/sporadische Mengen benutzt werden.
[b] *List scheduling* unterstützt heterogene Prozessoren nur eingeschränkt.

sormodell, asynchrone Ankunftszeiten, Verdrängbarkeit, Reihenfolgebeschränkungen, periodische/sporadische vs. aperiodische Tasks bzw. Jobs, das *Deadline*-Modell (falls existent), Job- vs. Task-Prioritäten, globales vs. lokales *Scheduling* (für Multiprozessoren), die Zielfunktion, den Unterabschnitt im Buch und den Namen des Algorithmus. Algorithmen wie *Earliest Deadline First* sind für nicht-periodische Systeme entwickelt, aber sie können auch auf periodische/sporadische angewandt werden. Aus der ersten Spalte geht hervor, dass heterogene Multiprozessoren nur durch die Algorithmen in den letzten drei Zeilen voll unterstützt werden. Uniforme Prozessoren werden nur als Anwendung des 0/1 Multi-Knapsack-Modells erwähnt. Verdrängungen sind nutzlos, wenn alle Jobs zu derselben Zeit ausführungsbereit werden (kenntlich gemacht durch ein „-" in der zweiten Spalte). Daher gibt es in diesem Fall in der dritten Spalte kein X. Einträge in der Spalte D_i sind nur für periodische/sporadische Tasks relevant. Als Zielfunktion wird in vielen Fällen die maximale Verspätung benutzt. Für periodisches/sporadisches *Scheduling* ist die wesentliche Frage aber: gibt es ein *Schedule*, welches die *Deadline* erfüllt? *Bin packing*

zielt vom Entwurf her auf die Minimierung der Anzahl der Prozessoren. Für HEFT und CPOP ist *Makespan* die relevante Zielfunktion. Nur die letzte Zeile entspricht der Minimierung mehrerer Kriterien, entweder in der Form eines einzelnen Kriteriums zur Zeit oder in Form einer multi-kriteriellen Optimierung auf der Basis der Pareto-Optimalität.

Ähnlich wie die Beurteilung der Rechenleistung ist auch das *Scheduling* ein Vorgang, der nicht nur ein einziges Mal während des Entwurfs durchgeführt wird. Vielmehr werden *Scheduling*-Algorithmen während des Entwurfs mehrfach benötigt. Grobe Abschätzungen können sogar erforderlich werden, wenn die Spezifikation verfasst wird. Später werden evtl. genauere Vorhersagen der Ausführungszeiten benötigt. Nach der Kompilierung gibt es noch genaueres Wissen über die Ausführungszeiten und dementsprechend können noch genauere *Schedules* erzeugt werden. Schließlich kann auch möglicherweise zur Laufzeit entschieden werden, welche Task/welcher Job als nächstes auszuführen ist.

In der Praxis ist es sehr wichtig zu wissen, ob ein *Schedule* für eine gegebene Menge an Tasks und Randbedingungen existiert. Eine Menge von Tasks heißt *schedulable* bei einer gegebenen Menge von Randbedingungen, wenn ein *Schedule* für diese Menge von Tasks und Randbedingungen existiert. Für viele Anwendungen sind Tests auf die Existenz eines *Schedules* (engl. *schedulability tests*) wichtig. Tests, die immer ein exaktes Ergebnis liefern, sind in vielen Fällen NP-hart [178]. Daher werden notwendige und hinreichende Tests benutzt. Für hinreichende Tests werden hinreichende Bedingungen für die Existenz eines *Schedules* geprüft. Es gibt eine (hoffentlich kleine) Wahrscheinlichkeit dafür, dass ein *Schedule* nicht garantiert werden kann, aber dennoch ein solches existiert. Notwendige Tests basieren auf notwendigen Bedingungen. Mit ihnen kann gezeigt werden, dass kein *Schedule* existiert. Dennoch kann es Fälle geben, in denen die notwendigen Bedingungen erfüllt sind, aber trotzdem kein *Schedule* existiert.

6.2 *Scheduling* für Einzelprozessoren

Wir betrachten zunächst den Fall von Einzelprozessoren. In der Triplet-Notation entspricht dies dem Fall (1|..|..). Für diesen Abschnitt benutzen wir Material aus dem Buch von Buttazzo [81]. Für zusätzliche Referenzen sei auf dieses Buch verwiesen.

6.2.1 Scheduling ohne Reihenfolgebeschränkungen

Weiterhin betrachten wir zunächst *Scheduling* ohne Reihenfolgebeschränkungen, d.h. den Fall unabhängiger Jobs.

***Earliest Due Date*-Algorithmus**

Noch weiter einschränkend betrachten wir die Situation, in der alle Jobs gleichzeitig ausführungsbereit werden und wir die maximale Verspätung minimieren wollen. Verdrängungen sind offenbar nutzlos, wenn alle Jobs zur selben Zeit ankommen. In der Triplet-Notation betrachten wir daher den Fall $(1|..|L_{max})$. Für diesen Fall wurde von Jackson 1955 [264] eine sehr einfache Regel gefunden:

Theorem 6.1 (Jacksons Regel): *Wenn eine Menge von n unabhängigen Jobs gegeben ist, so ist jeder Algorithmus, der die Jobs in der Reihenfolge nicht-abnehmender Deadlines ausführt, optimal in Bezug auf die Minimierung der maximalen Verspätung.*

Der Algorithmus, der dieser Regel folgt, heißt ***Earliest Due Date*** **(EDD)**. Wenn die *Deadlines* vorab bekannt sind, kann EDD als ein statischer Algorithmus realisiert werden. Für EDD müssen die Jobs nach ihren *Deadlines* sortiert sein. Seine Komplexität ist aufgrund des Sortierens $O(n \log(n))$.

Beweis (der Optimalität von EDD): Sei $\mathcal{S}$ ein *Schedule*, das durch irgendeinen Algorithmus A erzeugt wurde. Wenn A nicht die Wirkung von EDD hat, dann gibt es Jobs J_a und J_b derart, dass die Ausführung von J_b derjenigen von J_a vorangeht, obwohl die *Deadline* von J_a kleiner ist als die von J_b ($d_a < d_b$). Wir betrachten nun ein *Schedule* $\mathcal{S}'$ welches aus $\mathcal{S}$ durch Vertauschen der Ausführungsreihenfolge von J_a und J_b entsteht (siehe Abb. 6.4).

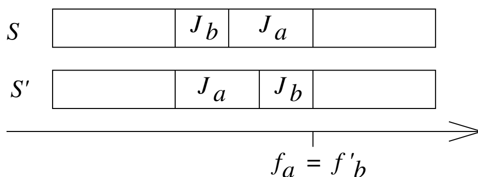


Abb. 6.4 *Schedules*$\mathcal{S}$ und $\mathcal{S}'$

Für das *Schedule* $\mathcal{S}'$ ist $L'_{max}(a, b) = max(L'_a, L'_b)$ die maximale Verspätung unter den Jobs J_a und J_b. L'_a ist die maximale Verspätung von Job J_a in *Schedule* $\mathcal{S}'$. L'_b wird entsprechend definiert. Es gibt zwei mögliche Fälle:

1. $L'_a > L'_b$: in diesem Fall haben wir
   $L'_{max}(a, b) = f'_a - d_a$
   J_a wird im neuen *Schedule* früher beendet. Deswegen gilt
   $L'_{max}(a, b) = f'_a - d_a < f_a - d_a$.
   Die rechte Seite dieser Ungleichung ist die maximale Verspätung in *Schedule* $\mathcal{S}$. Deswegen gilt das folgende:
   $L'_{max}(a, b) < L_{max}(a, b)$

2. $L'_a \leq L'_b$:
   In diesem Fall haben wir:
   $L'_{max}(a,b) = f'_b - d_b = f_a - d_b$ (siehe Abb. 6.4).
   Die *Deadline* von J_a liegt vor derjenigen von J_b. Dies führt auf
   $L'_{max}(a,b) < f_a - d_a$
   Wieder gilt:
   $L'_{max}(a,b) < L_{max}(a,b)$

Als Ergebnis kann jedes von einem EDD-*Schedule* verschiedene *Schedule* in endlich vielen Vertauschungen in ein EDD-*Schedule* transformiert werden, wobei die maximale Verspätung nur kleiner werden kann. Also ist EDD optimal für diese Klasse von *Scheduling*-Problemen. q.e.d. □

Earliest Deadline First-Algorithmus

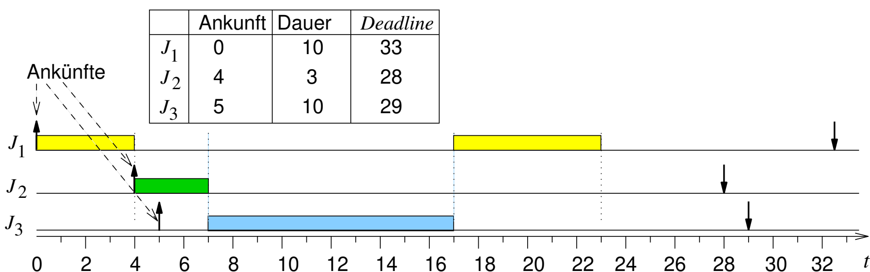
Betrachten wir nun den Fall von unterschiedlichen Ankunftszeiten für ein Einzelprozessor-System. Bei diesem Szenario kann ein Verdrängen von Jobs (engl. *preemption*) potentiell die maximale Verspätung verbessern. In der Triplet-Notation entspricht dies dem Fall ($1 \mid r_i, prmp \mid L_{max}$).

Der ***Earliest Deadline First*** **(EDF)**-Algorithmus ist optimal in Bezug auf die Minimierung der maximalen Verspätung. Er basiert auf folgendem Theorem [223]:

Theorem 6.2: *Wenn eine Menge von n unabhängigen Jobs mit beliebigen Ankunftszeiten gegeben ist, so ist ein Algorithmus, der zu jedem Zeitpunkt den ausführungsbereiten Job mit der frühesten absoluten Deadline ausführt, optimal in Bezug auf die Minimierung der maximalen Verspätung.*

Bei EDF muss jeder ankommende ausführungsbereite Job in eine Warteschlange von ausführbereiten Jobs eingefügt werden. Die Jobs in der Warteschlange sind nach ihrer *Deadline* sortiert. Wenn ein neu angekommener Job als erstes Element in die Warteschlange eingefügt wird, muss der gerade ausgeführte Job beendet werden.

Beispiel 6.1: Abb. 6.5 zeigt ein EDF-*Schedule*.



	Ankunft	Dauer	*Deadline*
J_1	0	10	33
J_2	4	3	28
J_3	5	10	29

Abb. 6.5 EDF-*Schedule*

Zum Zeitpunkt 4 hat Job J_2 eine frühere *Deadline*. Daher wird J_1 verdrängt. Zur Zeit 5 wird J_3 ausführungsbereit. Aufgrund der späteren *Deadline* wird J_2 nicht verdrängt. J_3 startet erst, wenn J_2 beendet ist. Danach wird J_3 bis zu seinem Ende ausgeführt. J_1 wird erst ausgeführt, wenn J_3 beendet ist. ∇

Wenn für die Warteschlange sortierte Listen verwendet werden, ist die Komplexität von EDF $\mathcal{O}(n^2)$. Sogenannte *Bucket*-Arrays können zur Reduktion der Laufzeit beitragen. Die Prioritäten sind offensichtlich dynamisch: sie hängen davon ab, welche *Deadline* die nächste ist. Da EDF dynamische Prioritäten benutzt, kann es es nicht mit einem Betriebssystem verwendet werden, welches nur statische Prioritäten unterstützt. Allerdings wurde gezeigt, dass man Betriebssysteme so erweitern kann, dass EDF auf Anwendungsebene simuliert wird [132].

Beweis (des Theorems 6.2): Sei $\mathcal{S}$ ein *Schedule*, welches durch einen von EDF verschiedenen Algorithmus A erzeugt wird. Sei $\mathcal{S}_{EDF}$ ein durch EDF erzeugtes *Schedule*. Wir zerlegen nunmehr die Zeitachse in diskrete Zeitabschnitte von jeweils einer Zeiteinheit[4]. Jeder Zeitabschnitt enthält die Zeiten im Intervall $[t, t+1)$. Sei $\mathcal{S}(t)$ der Job, der gemäß *Schedule* $\mathcal{S}$ im Zeitabschnitt $[t, t+1)$ ausgeführt wird. Sei $E(t)$ der Job, der zur Zeit t unter allen verfügbaren Jobs die früheste *Deadline* besitzt. Sei ferner $t_E(t)$ die Zeit ($\geq t$), zu welcher Job $E(t)$ im *Schedule* $\mathcal{S}$ seine Ausführung beginnt. *Schedule* $\mathcal{S}$ ist kein EDF-*Schedule*. Daher gibt es eine Zeit t, zu der nicht der Job mit der frühesten *Deadline* ausgeführt wird. Für t gilt $\mathcal{S}(t) \neq E(t)$ (siehe Abb. 6.6).

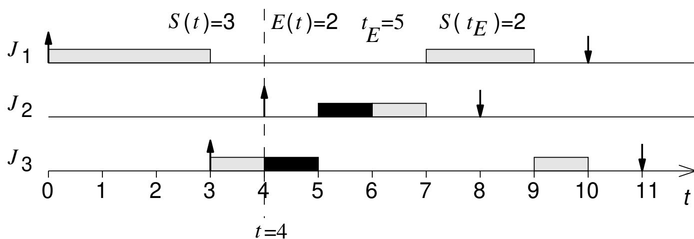


Abb. 6.6 *Schedule* $\mathcal{S}$

Mit denselben Argumenten wie bei Jacksons Regel können wir zeigen, dass die Vertauschung $\mathcal{S}(t) \neq E(t)$ wie in Abb. 6.7 nicht die maximale Verspätung erhöht.

Daher kann jedes *Schedule*, das kein EDF-*Schedule* ist, durch eine begrenzte Anzahl von Vertauschungen in ein EDF-*Schedule* umgeformt werden, ohne die maximale Verspätung zu erhöhen. Also ist EDF unter den möglichen Algorithmen optimal.

Wir können zeigen, dass das Vertauschen alle *Deadlines* einhält, sofern sie im *Schedule* $\mathcal{S}$ eingehalten wurden. Aufgrund der ursprünglichen Annahme ist die

[4] Dieser Beweis nimmt an, dass wir mit diskreten Zeiten arbeiten. Er kann auf reelle (kontinuierliche) Zeiten erweitert werden.

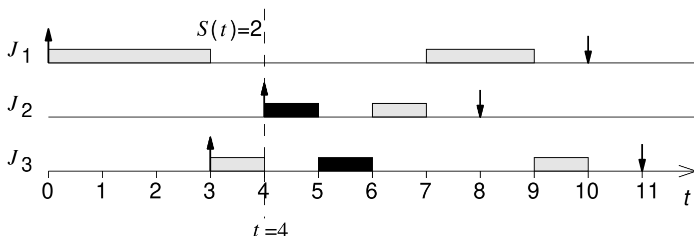


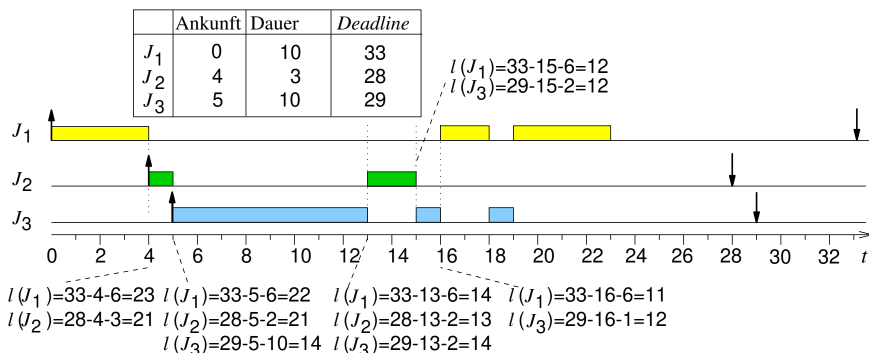
Abb. 6.7 *Schedule* nach dem Vertauschen von Jobs $\mathcal{S}(t)$ und $E(t)$

maximale Verspätung in *Schedule* $\mathcal{S}$ gleich 0. Deswegen und weil EDF ein *Schedule* mit minimaler Verspätung erzeugt, ist die maximale Verspätung des EDF-*Schedules* auch gleich Null. Daher ist das EDF-Schule für diese Problemklasse das optimale *Schedule*, das auch die *Deadlines* einhält. □

Least Laxity-Algorithmus

Wir wenden uns nunmehr der Betrachtung des Schlupfes (engl. *laxity*) zu und untersuchen den Fall $(1|r_i, prmp, ..|..)$, wobei wir ein *Schedule* finden möchten, sofern es existiert. *Least Laxity* (LL), *Least Slack Time First* (LST) und *Minimum Laxity First* (MLF) sind drei Namen für eine Schlupf-basierte *Scheduling*-Strategie [349]. Beim LL-*Scheduling* sind die Prioritäten der Jobs eine monoton fallende Funktion ihres Schlupfs (siehe Gleichung (6.1); je weniger Spielraum ein Job also hat, desto höher seine Priorität). Der Schlupf verändert sich dynamisch.

Beispiel 6.2: Abb. 6.8 zeigt ein Beispiel eines LL-*Schedules* mit den berechneten *Laxity*-Werten.



	Ankunft	Dauer	*Deadline*
J_1	0	10	33
J_2	4	3	28
J_3	5	10	29

Abb. 6.8 *Least laxity-Schedule*

Zum Zeitpunkt 4 wird Job J_1 wie oben verdrängt. Zum Zeitpunkt 5 wird Job J_2 nun auch verdrängt, weil er einen höheren Schlupf hat als Job J_3. ∇

LL ist auch ein verdrängendes (präemptives) *Scheduling*-Verfahren. Verdrängungen sind dabei nicht auf Zeitpunkte beschränkt, zu denen neue Jobs ausführungsbereit werden. Ein negativer Schlupf ist eine frühe Warnung für *Deadlines*, die verpasst werden. Man kann zeigen (in [349] ist dies als Übung dem Leser überlassen), dass LL auch ein optimales *Scheduling*-Verfahren für Einzelprozessor-Systeme in dem Sinne ist, dass es ein *Schedule* findet, wenn ein *Schedule* existiert. Wegen seiner dynamischen Prioritäten kann man es nicht in Standard-Betriebssystemen einsetzen, da diese üblicherweise nur statische Task-Prioritäten verwalten können. Im Gegensatz zu EDF-*Scheduling* erfordert LL-*Scheduling* Wissen über die Ausführungszeit. Seine Anwendungsgebiete sind daher auf solche Situationen beschränkt, in denen seine Eigenschaften vorteilhaft sind. In den Unterabschnitten 6.3.3 und 6.3.4 wird gezeigt werden, dass der Schlupf im Multiprozessor-*Scheduling* eine Rolle spielen kann.

Verfahren ohne Verdrängung

Wir betrachten nunmehr den Fall, in dem Verdrängungen (engl. *preemptions*) nicht erlaubt sind, in unserer Klassifikation als $(1|r_i|L_{max})$ bezeichnet.

Theorem 6.3: *Wenn Verdrängungen nicht erlaubt sind, dann müssen optimale Schedules den Prozessor zu gewissen Zeiten unbeschäftigt lassen, damit spät eintreffende Jobs mit frühen Deadlines rechtzeitig beendet werden können.*

Beweis: Nehmen wir an, dass ein optimaler nicht verdrängender *Scheduler* (der kein Wissen über die Zukunft hat) den Prozessor immer auslastet. Dieser *Scheduler* muss das *Schedule* in Abb. 6.9 optimal erzeugen (d.h. er muss ein *Schedule* finden, wenn eines existiert). Für das Beispiel in Abb. 6.9 nehmen wir zwei Tasks an. Sei

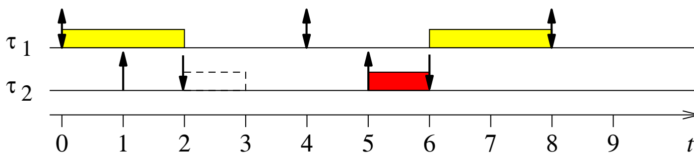


Abb. 6.9 Der *Scheduler* darf den Prozessor nicht vollständig auslasten

τ_1 eine periodische Task mit $C_1 = 2$, $T_1 = 4$, $D_1 = 4$ und $r_1 = 0$. Sei τ_2 eine sporadische Task mit $C_2 = 1$, $D_2 = 1$, $T_2 = 4$ und $r_2 = 1$, d.h. sporadisch zu Zeiten $4 * n + 1$ ausführungsbereit werdend. Wir nehmen an, dass die gleichzeitige Ausführung beider Tasks aufgrund von Ressourcenkonflikten nicht möglich ist.

Unter diesen Annahmen muss der *Scheduler* die Ausführung von Task τ_1 zum Zeitpunkt 0 starten, da er ja den Prozessor voll auslasten soll. Da der *Scheduler*

nicht-verdrängend ist, kann er τ_2 nicht starten, wenn er zum Zeitpunkt 1 ausführungsbereit ist. Daher verpasst τ_2 die *Deadline*. Wenn der *Scheduler* den Prozessor nicht sofort belegt hätte (wie in Abb. 6.9 zum Zeitpunkt 4 gezeigt), hätte er ein zulässiges *Schedule* gefunden. Daher ist dieser *Scheduler* nicht optimal. Dies ist ein Widerspruch zur Annahme, dass optimale *Scheduler* existieren, die den Prozessor immer voll auslasten. □

Abschließend stellen wir fest: um verpasste *Deadlines* zu vermeiden, muss der *Scheduler* Wissen über die Zukunft haben. Solche Algorithmen heißen **hellsehend** (engl. *clairvoyant*). Ein Algorithmus, der Prozessoren trotz der Anwesenheit ausführungsbereiter Tasks unbeschäftigt lässt, heißt **nicht arbeitserhaltend**.

Definition 6.8: Ein *Scheduling*-Algorithmus heißt **arbeitserhaltend** (engl. *work conserving*), wenn es keine Zeiten gibt, zu denen der Prozessor unbeschäftigt ist, während es eine ausführungsbereite Task gibt [121].

Wenn es vorab keine Informationen über Ankunftszeiten gibt, dann kann kein *online*-Algorithmus entscheiden, ob er den Prozessor unbeschäftigt lassen soll. Wenn Ankunftszeiten a priori bekannt sind, wird das *Scheduling*-Problem im nicht verdrängenden Fall im Allgemeinen NP-hart und *Branch-and-Bound*-Techniken werden typischerweise zur Erzeugung von *Schedules* verwendet.

6.2.2 *Scheduling mit Reihenfolgebeschränkungen*

Als nächstes betrachten wir die Ablaufplanung mit vorhandenen Reihenfolgebeschränkungen, ausgedrückt durch eine Präzedenzrelation *prec* und in der Triplet-Notation als ($1| r_i$, *prmp, prec* | L_{max}) bezeichnet.

Task-Graphen

Reihenfolgebeschränkungen werden in gerichteten azyklischen Abhängigkeitsgraphen (DAGs, siehe Definition 2.6) $G = (\tau, E)$ ausgedrückt. Die Menge τ stellt die Knoten (engl. *vertices* oder *nodes*) und $E \subseteq \tau \times \tau$ seine Kanten (engl. *edges*) des Graphen dar.

Beispiel 6.3: In der Abb. 6.10 drücken die Kanten aus, dass die Quellknoten (die ersten Komponenten der Tupel, die eine Kante repräsentieren) vor den Zielknoten (die zweiten Komponenten der Tupel, die eine Kante repräsentieren) ausgeführt werden müssen. Die Knotenbeschriftungen bezeichnen Tasks. ∇

Es gibt mehrere Gründe, Anwendungen in Form von DAGs zu beschreiben.

1. Zum einen könnte jeder Knoten einer Task entsprechen und Kanten würden Abhängigkeiten zwischen Tasks repräsentieren.

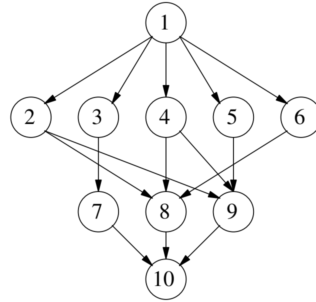


Abb. 6.10 Reihenfolgebeschränkungen

2. Zum anderen führt die Verfügbarkeit von Multiprozessoren zu der Idee der Aufspaltung von Tasks in Subtasks und der Ausführung der Subtasks in überlappender Weise auf den verschiedenen Prozessoren. Das automatische Partitionieren von Tasks in Subtasks, sodass parallele Prozessoren effizient genutzt werden können, heißt **automatische Parallelisierung**. Automatische Parallelisierung ist sogar noch schwieriger als das automatische *Scheduling* für eine gegebene Menge von Subtasks (beispielsweise weil Datenabhängigkeiten analysiert werden müssen).

Beide Fälle der Erzeugung von DAGs können auch in Kombination Anwendung finden: wir können Abhängigkeiten zwischen Tasks haben und Tasks können in Subtasks aufgespalten werden. Nachfolgend nehmen wir an, dass DAGs jede der eben beschriebenen Situationen repräsentieren können und wir sprechen in jedem Fall von Task-Graphen (siehe auch Seite 39). Für das *Scheduling* ist es unwichtig, wie der DAG erzeugt wurde.

Beispiel 6.4: Abb. 6.11 zeigt ein gültiges *Schedule* für einen einfacheren Task-Graphen einschließlich Kommunikation. Task τ_3 kann nur ausgeführt werden, wenn τ_1 und τ_2 ausgeführt wurden und Nachrichten an τ_3 geschickt haben.

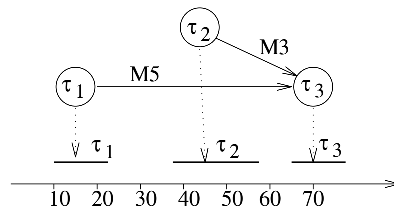


Abb. 6.11 Reihenfolgebeschränkungen und *Schedule*

∇

Latest Deadline First-Algorithmus

Ein optimaler Algorithmus, der die maximale Verspätung im Falle gleichzeitig ankommender Tasks oder Jobs minimiert, wurde von Lawler vorgestellt [327]. Dieser

Algorithmus heißt ***Latest Deadline First*** (**LDF**)-Algorithmus. LDF liest den Task-Graphen und speichert den Knoten, der die späteste *Deadline* und keine Nachfolger im Graphen hat, in einem Stapel. Dies wird für alle verbleibenden Knoten wiederholt, wobei immer der Knoten mit der spätesten *Deadline* unter allen Knoten mit bereits ausgewählten Nachfolgern gewählt wird. Dieser Knoten wird auf den Stapel gelegt. Zur Laufzeit werden die Tasks einer Reihenfolge ausgeführt, bei der wir die Einträge im Stapel von oben nach unten abarbeiten, d.h. in einer Reihenfolge, die gegenüber der Reihenfolge der Betrachtung der Knoten im Task-Graphen **umgekehrt** ist. LDF ist nicht verdrängend und optimal für Einzelprozessoren.

Beispiel 6.5: Wir betrachten das Beispiel aus Abb. 6.11. LDF würde zuerst τ_3 auf dem Stapel ablegen, denn diese Task hat keinen Nachfolger. Danach sind alle Nachfolger von τ_1 und τ_2 bereits gewählt. Es hängt von der *Deadline* ab, welcher der Knoten als nächstes gewählt wird. Der Knoten mit der späteren *Deadline* wird als erstes auf den Stapel gelegt. Zur Laufzeit wird der Stapel in der umgekehrten Reihenfolge abgearbeitet und beispielsweise mit der Bearbeitung von τ_1 begonnen. ∇

Im Falle von asynchronen Ankunftszeiten der Tasks kann man mit einem modifizierten EDF-Algorithmus ein gültiges *Schedule* bestimmen. Die Grundidee besteht darin, das Problem mit der gegebenen Menge abhängiger Tasks in eine äquivalente Menge unabhängiger Tasks mit unterschiedlichen Zeitparametern umzuwandeln [98]. Dieser Algorithmus ist wiederum optimal für Einzelprozessoren.

Wenn Tasks nicht verdrängt werden dürfen, kann der heuristische Algorithmus aus [508] verwendet werden.

6.2.3 Periodisches Scheduling ohne Reihenfolgebeschränkungen

Jetzt betrachten wir periodische Abläufe. Wir werden anstelle von Jobs überwiegend Tasks betrachten, da sich im periodischen Fall die meisten Eigenschaften für Tasks zeigen lassen. Wir beschränken uns auf Tasks ohne Reihenfolgeabhängigkeiten, was in der Triplet-Notation durch das Triplet ($1|r_i$,*prmp, periodic*| ..) ausgedrückt werden kann.

Notation

Bei periodischem *Scheduling* sind die für aperiodisches *Scheduling* relevanten Ziele nicht sehr nützlich. Beispielsweise spielt die Minimierung der totalen Dauer des *Schedules* keine Rolle, wenn wir eine unendliche Wiederholung von Jobs betrachten. Wir können bestenfalls einen Algorithmus entwerfen, der, falls ein *Schedule* existiert, dieses immer findet. Dies motiviert die Definition von Optimalität für periodische *Schedules*.

Definition 6.9: Ein periodischer *Scheduling*-Algorithmus wird als **optimal** bezeichnet, wenn er immer ein *Schedule* findet, falls eines existiert.

Definition 6.10: Für periodische und sporadische Task-Systeme $\tau = \{\tau_1, .., \tau_n\}$ definieren wir die Task-Auslastung (engl. *task utilization*) als

$$u_i = \frac{C_i}{T_i} \tag{6.4}$$

Damit benutzen wir für sporadische Systeme dieselbe Notation, obwohl T_i nur den minimalen Abstand zwischen zwei Jobs bedeutet.

Definition 6.11: Für ein Task-System $\tau = \{\tau_1 ... \tau_n\}$ mit der Auslastung u_i von Task τ_i definieren wir das Maximum U_{max} und die Gesamtauslastung U_{sum} durch:

$$U_{max} = \max_i (u_i) \tag{6.5}$$

$$U_{sum} = \sum_i u_i \tag{6.6}$$

Rate Monotonic Scheduling

Rate Monotonic Scheduling (RMS) [348] ist wohl der bekannteste *Scheduling*-Algorithmus für unabhängige periodische Tasks. RMS basiert auf den folgenden Annahmen („**RM-Annahmen**"):

1. Alle Tasks mit harter *Deadline* sind periodisch.
2. Alle Tasks sind voneinander unabhängig.
3. $D_i = T_i$ für alle Tasks.
4. C_i ist konstant und für alle Tasks bekannt.
5. Die Zeit für einen Kontextwechsel ist vernachlässigbar.
6. Für einen Prozessor und n Tasks wird die folgende Schranke bzgl. der Gesamtauslastung U_{sum} eingehalten:

$$U_{sum} = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1) \tag{6.7}$$

Abb. 6.12 zeigt Werte der Schranke in Ungleichung (6.7).

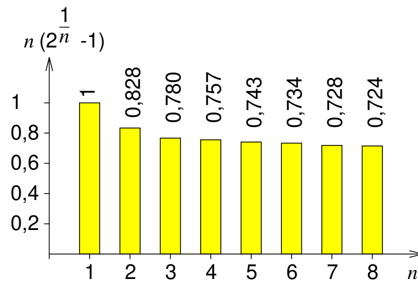


Abb. 6.12 Werte der rechten Seite von Ungleichung (6.7)

Die Schranke für die Auslastung hat für große Werte von n einen Wert von ungefähr 0,7:

$$\lim_{n\to\infty} n * (2^{1/n} - 1) = log_e(2) = ln(2) \quad \approx 0,7 \tag{6.8}$$

Beim *Rate Monotonic Scheduling* ist **die Priorität der Tasks eine monoton fallende Funktion ihrer Periode**. Anders ausgedrückt, haben Tasks mit einer kurzen Periode eine höhere Priorität, wohingegen Tasks mit langer Periode eine geringere Priorität haben. RM-*Scheduling* ist verdrängend mit **festen Prioritäten**.

Beispiel 6.6: Abb. 6.13 zeigt ein mit RMS erzeugtes *Schedule*. Pfeile mit zwei Spitzen deuten die Ankunftszeiten einer Task und die *Deadline* der vorherigen Task an. Tasks τ_1, τ_2 und τ_3 haben jeweils eine Periode von 2, 6 und 6. Die Ausführungszeiten betragen 0,5, 2 und 1,75. Task τ_1 hat die kürzeste Periode und daher die höchste Priorität. Die Task τ_2 wird mehrere Male verdrängt. Jedes Mal, wenn Task τ_1 ausführungsbereit ist, unterbricht ihr Job die gerade aktive Task. Task τ_2 hat die gleiche Periode wie Task τ_3, daher verdrängen sich diese beiden Tasks nicht.

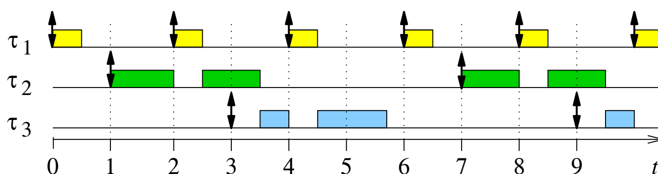


Abb. 6.13 Beispiel eines RM-*Schedules* ∇

Die Schranke in Ungleichung (6.7) erfordert, dass ein Teil der Rechenleistung des Prozessors unbenutzt bleibt, um sicherzustellen, dass alle Anforderungen rechtzeitig behandelt werden können. Warum gibt es diese Nutzungsschranke? Die Hauptursache dafür ist, dass die statischen Prioritäten des RMS bedingen, dass möglicherweise eine Task verdrängt wird, die nahe ihrer *Deadline* ist und dafür eine andere, höher priorisierte Task mit deutlich späterer *Deadline* zum Zuge kommt. Dadurch könnte die Task mit niedrigerer Priorität dann ihre *Deadline* verpassen.

Lemma 6.1: ***Wenn die oben genannten sechs RM-Annahmen*** *(siehe Seite 339)* ***erfüllt sind, werden alle Deadlines garantiert eingehalten*** *(siehe Buttazzo [81]).*

Beispiel 6.7: Die Task-Parameter der Abb. 6.14 sind: $T_1 = 5, C_1 = 3, T_2 = 8, C_2 = 3$. Die Gesamtauslastung ist: $U_{sum} = \frac{3}{5} + \frac{3}{8} = \frac{39}{40} = 0,975$. Die Schranke der Ungleichung (6.7) ist $2*(2^{\frac{1}{2}} - 1) \approx 0,828$. Die Schranke wird damit nicht eingehalten somit ist nicht garantiert, dass ein RM-*Schedule* existiert. Tatsächlich wird die *Deadline* zur Zeit 8 verpasst. Wir nehmen hier an, dass die Berechnungen, die ihre *Deadline* verpasst haben, nicht in der folgenden Periode nachgeholt werden. ∇

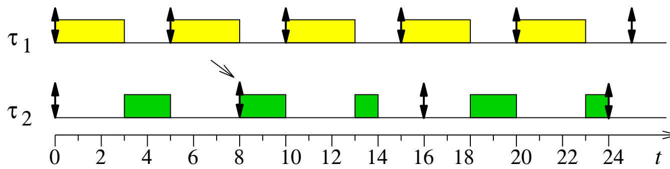


Abb. 6.14 RM-*Schedule* verpasst die *Deadline* zum Zeitpunkt 8

Solche verpassten *Deadlines* können nicht auftreten, wenn die Prozessorlast sehr gering ist, sie treten dagegen möglicherweise bei hoher Prozessorlast auf, wie in Abb. 6.14 zu sehen. Wenn die Ungleichung (6.7) erfüllt ist, ist damit garantiert, dass die Prozessorauslastung gering genug ist, um Probleme wie in Abb. 6.14 zu vermeiden. Die Ungleichung (6.7) ist hinreichend, aber nicht notwendig. Es könnte damit ein RM-*Schedule* geben, obwohl die Ungleichung nicht erfüllt ist. Es gibt weitere hinreichende Schranken [54].
RMS hat die folgenden wichtigen Vorteile:

- Unter den verdrängenden *Scheduling*-Algorithmen mit fester Priorität für Einzelprozessoren ist RMS optimal [54].
- RMS basiert auf **statischen** Prioritäten. Das ermöglicht die Verwendung von RMS in Standard-Betriebssystemen, die feste Prioritäten unterstützen.
- Wenn die sechs RM-Annahmen eingehalten werden, ist ein *Schedule* garantiert (siehe [81]).

RMS ist die Basis vieler formaler *Schedulability*-Beweise. Bei der Konstruktion von Beispielen und beim Führen von Beweisen ist es hilfreich, zu wissen, welche Situationen hinsichtlich des Findens von *Schedules* mit RMS besonders kritisch sind. Wir setzen dazu zunächst folgende Eigenschaft voraus:

Eigenschaft 6.1: Wir nehmen an, dass jeder Job beendet wird, bevor der nächste Job derselben Task ausführungsbereit wird.

Definition 6.12: Ein kritischer Zeitpunkt (engl. *critical time instant*) für eine Task τ_i ist der Zeitpunkt t, an dem ein Eintreffen der Task zur größten Antwortzeit führt.

Theorem 6.4 (*Critical instant theorem*): *Für ein Scheduling mit festen Prioritäten auf einem Einzelprozessor ist die Antwortzeit für jede Task τ_i maximiert, wenn τ_i gleichzeitig mit allen anderen Tasks einer höheren Priorität ausführungsbereit wird.*

Beweis: Wir zeigen hier den originalen Beweis von Liu and Layland [348] in deutscher Übersetzung und mit Anpassung an unsere Notation: „Sei $\tau = \{\tau_1, ..., \tau_n\}$ eine Menge von nach Prioritäten sortierten Tasks, wobei τ_n die Task mit der niedrigsten Priorität ist. Wir betrachten ein Eintreffen von τ_n zur Zeit t_1. Wir nehmen an, dass zwischen der Zeit t_1 und der Zeit $t_1 + T_n$ (der Zeit, zu der die nächste Anforderung von τ_n erfolgt) Task τ_i, $i < n$, zu den Zeitpunkten t_2, $t_2 + T_i$, $t_2 + 2T_i$, ..., $t_2 + kT_i$ ausführungsbereit wird, wie in Abb. 6.15 gezeigt. Offensichtlich wird die Verdrängung von

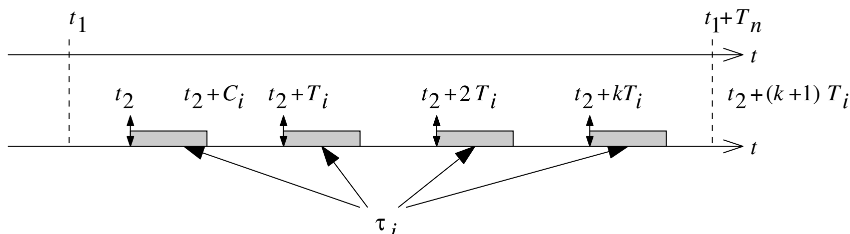


Abb. 6.15 Verzögern von Task τ_n durch höher priorisierte Task τ_i

τ_n durch τ_i eine gewisse Verzögerung bei der Ausführung der bei t_1 eintreffenden Instanz von τ_n bewirken, sofern nicht τ_n vor t_2 fertig gestellt ist. Außerdem sehen wir aus Abb. 6.15, dass ein Vorziehen der Zeit t_2 die Fertigstellung von τ_n nicht beschleunigen würde. Durch ein solches Vorziehen bleibt die Fertigstellung entweder unverändert oder sie wird verzögert. Also ist die Verzögerung in der Fertigstellung von τ_n am größten, wenn t_2 und t_1 übereinstimmen. Wir beweisen das Theorem, indem wir das Argument für alle $\tau_i, i = 2, ..., m-1$, wiederholen.” □

Implizit haben wir die Eigenschaft 6.1 im Beweis benutzt. Wenn wir den allgemeinen Fall betrachten (d.h. den Fall, in dem die Eigenschaft 6.1 nicht gilt, siehe z.B. Baker [33]), dann bleibt Theorem 6.4 gültig, aber der Beweis wird komplexer, wie von Devillers et al. [129] und Bril [69] gezeigt[5].

Das Theorem von den kritischen Zeitpunkten hilft, *Schedules* für Einzelprozessoren zu finden. Leider gilt dieses Theorem nicht für Multiprozessor-Systeme, sodass Beweise wesentlich schwieriger werden. Man sollte sich also freuen, dass dieses Theorem für Einzelprozessoren gilt!

Wir wollen uns nun andere Eigenschaften von RMS ansehen. Die freie Kapazität des Prozessors wird nicht immer benötigt.

Theorem 6.5: *Sei τ ein System mit periodischen Tasks. Wenn die Periode aller Tasks ein Vielfaches der Periode der Task mit der höchsten Priorität ist, kann τ mit RMS zeitlich eingeplant werden, wenn gilt:*

$$U_{sum} \leq 1 \tag{6.9}$$

Beispiel 6.8: Diese Voraussetzung wird z.B. erfüllt, wenn Tasks eines Fernsehers mit den Raten von 25, 50 und 100 (oder 30, 60 und 120) Hertz ausgeführt werden müssen. ∇

Beweis (von Theorem 6.5): Die Tasks seien nach Prioritäten sortiert, sodass gilt: $\forall i : T_i \leq T_{i+1}$. Wir betrachten eine Task τ_i und die Task mit der nächstniedrigeren Priorität, Task τ_{i+1} (siehe Abb. 6.16). Die zweite *Deadline* von τ_{i+1} passt sehr schön

[5] Ich verdanke diesen Hinweis J.J. Chen von der TU Dortmund.

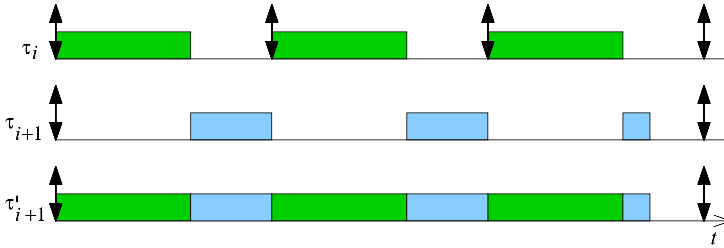


Abb. 6.16 Falten von Tasks benachbarter Prioritäten

zur vierten *Deadline* von τ_i. Deshalb können wir die Ausführungszeiten von Task τ_{i+1} in die Ausführungszeiten von Task τ_i hinein falten und eine neue Task τ'_{i+1} erzeugen, welche die Ausführungszeiten der beiden ursprünglichen Tasks enthält. Dieses Falten ist möglich, wenn die gesamte Ausführungszeit der beiden Tasks nicht die Periode von τ_{i+1} übersteigt. Dieses Verfahren kann in derselben Weise mit der Task der nächstniedrigeren Priorität wiederholt werden. Dieses Falten ist möglich, solange die Gesamtauslastung nicht größer ist als 1. □

Die Schranken in den Ungleichungen (6.7) oder (6.9) erlauben einen einfachen Test für die Existenz eines *Schedules*.

Aufgrund des Theorems der kritischen Zeitpunkte muss man beim Beweis der Optimalität von RMS nur den Fall betrachten, in dem alle Tasks gleichzeitig mit allen anderen einer höheren Priorität ausführungsbereit werden.

Earliest Deadline First Scheduling

EDF kann auch auf Mengen periodischer Tasks angewendet werden. Es reicht offensichtlich aus, das *Scheduling*-Problem für eine einzelne Hyperperiode wie ein aperiodisches *Scheduling*-Problem zu lösen. Die Lösung kann dann für alle weiteren Hyperperioden angewandt werden. So beträgt beispielsweise die Dauer der Hyperperiode für das Beispiel von Abb. 6.14 40 Zeiteinheiten. Aus der Optimalität von EDF für nicht-periodische *Schedules* ergibt sich, dass EDF auch für eine einzelne Hyperperiode optimal ist, damit also auch für das gesamte *Scheduling*-Problem. Es müssen also keine weiteren Bedingungen eingehalten werden, um die Optimalität des Verfahrens zu garantieren. Daraus folgt, dass EDF auch für den Fall $U_{sum} = 1$ optimal ist.

Beispiel 6.9: Dementsprechend wird keine *Deadline* verpasst, wenn für das Beispiel aus Abb. 6.14 EDF-*Scheduling* verwendet wird (siehe Abb. 6.17). Zum Zeitpunkt 5 unterscheidet sich das Verhalten von dem Verhalten bei RM-*Scheduling*: durch die frühere *Deadline* von τ_2 wird diese Task nicht verdrängt.

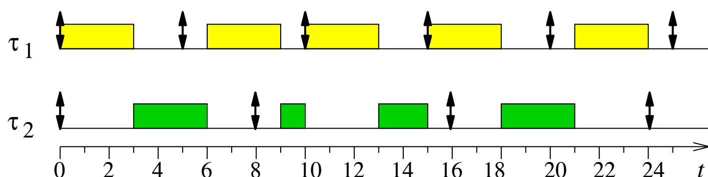


Abb. 6.17 Mit EDF erzeugtes *Schedule* für das Beispiel aus Abb. 6.14 ∇

Tasks mit expliziter *Deadline*

Nunmehr gehen wir über zur Betrachtung von Tasks, deren *Deadline* von der Periode verschieden ist. Solche Tasks heißen Tasks mit **expliziter *Deadline***. In einem solchen System ist jede Task durch ein Tripel (C_i, D_i, T_i) charakterisiert, wobei D_i die relative *Deadline* ist. Der Fall $D_i \leq T_i$ heißt *constrained deadline*-Fall. Wenn eine solche Einschränkung nicht existiert, sprechen wir von einer beliebigen *Deadline* (engl. *arbitrary deadline*). Offensichtlich ist der Fall einer expliziten *Deadline* allgemeiner als der Fall einer impliziten *Deadline* und jede Task mit impliziter *Deadline* ist auch eine Task mit expliziter *Deadline*.

Bei Tasks mit expliziter *Deadline* ist die Auslastung nur sehr begrenzt zur Charakterisierung der Rechenanforderungen geeignet. In gewissem Umfang spielt nunmehr die Dichte (engl. *density*) die Rolle, welche bislang die Auslastung spielte. Die Dichte ist wie folgt definiert:

$$dens_i = \frac{C_i}{\min(D_i, T_i)} \tag{6.10}$$

$$dens_{sum}(\tau) = \sum_{\tau_i \in \tau} dens_i \tag{6.11}$$

$$dens_{max}(\tau) = \max_{\tau_i \in \tau}(dens_i) \tag{6.12}$$

Werte der Dichte charakterisieren Rechenzeitanforderungen. Die *Demand-Bound-Function* (DBF) liefert allerdings bessere Schranken:

Definition 6.13: *Die **Demand-Bound-Function** $DBF(\tau_i, t)$ ist, für jede sporadische Task τ_i und jede reelle Zahl $t \geq 0$, die größte aufsummierte Rechenanforderung aller Jobs, die von τ_i erzeugt werden können und die sowohl den Zeitpukt der Ausführbarkeit wie auch die Deadline in einem zusammenhängenden Intervall der Länge t haben.*

Die aufsummierten Rechenanforderungen von Task τ_i in einem Intervall $[t_0, t_0 + t)$ sind maximiert, wenn einer ihrer Jobs zu Beginn des Intervalls eintrifft (d.h. zur Zeit t_0) und die nachfolgenden Jobs treffen so schnell wie erlaubt ein, d.h. an Zeitpunkten $t_0 + T_i, t_0 + 2T_i, t_0 + 3T_i, \ldots$. Diese Beobachtung führt uns zur Gleichung (6.13) [40, 38]:

$$DBF(\tau_i, t) = \max\left(0, \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) * C_i\right) \tag{6.13}$$

Die Dichte und die DBF hängen miteinander zusammen:

Lemma 6.2: *Für alle Tasks τ_i und für alle $t \geq 0$:*

$$t * dens_i \geq DBF(\tau_i, t) \tag{6.14}$$

Beweis: Wir vergleichen die graphischen Darstellungen der Dichte und der *DBF* als eine Funktion der Zeit. Abb. 6.18 zeigt die beiden Funktionen. Die linke Seite

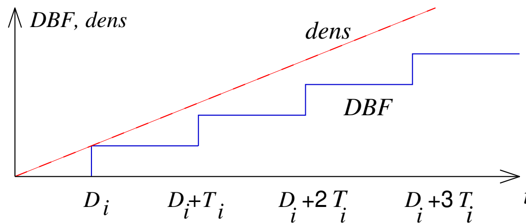


Abb. 6.18 Vergleich von Dichte und *DBF*

der Gleichung (6.14) ist als gerade Linie mit Steigung $dens_i$ dargestellt. Die *DBF* ist eine Stufenfunktion mit Stufen der Höhe C_i. Die Stufenfunktion steigt um C_i jedes Mal, wenn eine Task ausgeführt werden muss. Die erste Stufe ist bei $t = D_i$. Aufgrund der Definition der Dichte übersteigt diese Stufe nie die gerade Linie. Die nächsten Stufen gibt es bei $t = D_i + T_i$, $t = D_i + 2T_i$, $t = D_i + 3T_i$ usw. Auch diese Stufen werden die Gerade nicht überschreiten. □

EDF kann leicht auf den Fall der von den Perioden verschiedenen *Deadlines* erweitert werden. Für RMS heißt die Erweiterung *Deadline Monotonic Scheduling* (DMS).

Deadline Monotonic Scheduling

Tasks mit expliziten *Deadlines* können mit ***Deadline Monotonic Scheduling*** (DMS) zeitlich eingeplant werden. Statische Task-Prioritäten basieren bei DMS auf nicht-aufsteigenden *Deadlines*: für zwei Tasks τ_i und $\tau_{i'}$ ist die Priorität von τ_i größer als die von $\tau_{i'}$ wenn $D_i < D_{i'}$ ist.

Für *constrained deadline*-Tasks kann die Schranke in Ungleichung (6.7) zur Schranke in Ungleichung (6.15) verallgemeinert werden. Diese ist hinreichend, aber nicht notwendig [81].

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n(2^{1/n} - 1) \tag{6.15}$$

6.2.4 Periodisches Scheduling mit Reihenfolgebeschränkungen

Scheduling für abhängige Tasks ist schwerer als das *Scheduling* für unabhängige Tasks, v.a. falls keine Verdrängungen erlaubt sind (in der Triplet-Notation im Fall $(1|r_i,prec,periodic|L_{max})$). Das Entscheidungsproblem hinsichtlich der Existenz eines *Schedules* für eine Menge abhängiger Tasks und einer gegebenen *Deadline* ist NP-hart [178]. Es gibt verschiedene Strategien, um die Komplexität zu senken:

- Es werden zusätzliche Ressourcen bereit gestellt, sodass das *Scheduling* einfacher wird.
- Das *Scheduling* wird in statische und dynamische Anteile zerlegt. Bei diesem Ansatz werden so viele Entscheidungen wie möglich zur Entwurfszeit getroffen und der verbleibende Rest wird zur Laufzeit vorgenommen.

6.2.5 Sporadische Ereignisse

Prinzipiell könnte man sporadische Ereignisse mit *Interrupts* verbinden und sie jedes Mal sofort ausführen, wenn die Interrupt-Priorität die höchste im gesamten System ist. Das hätte allerdings ein unvorhersagbares Verhalten für alle anderen Tasks zur Folge. Daher werden besondere *sporadic task server* verwendet, die regelmäßig ausgeführt werden und dabei prüfen, ob ausführungsbereite sporadische Ereignisse existieren. So können sporadische Ereignisse praktisch in periodische Tasks umgewandelt werden, wodurch die Vorhersagbarkeit des Gesamtsystems deutlich verbessert wird.

6.3 *Scheduling* für unabhängige Jobs auf identischen Multiprozessoren

Aufgrund der großen Verbreitung von Mehrkern-Systemen in aktuellen eingebetteten Systemen betrachten wir als nächstes Multiprozessor-Systeme. Beim Übergang von Einzelprozessoren zu Multiprozessor-Systemen muss eine Vielzahl von Herausforderungen bewältigt werden. Zunächst betrachten wir den Fall von m identischen Prozessoren. Weiterhin gehen wir von einem Task-System $\tau = \{\tau_1, ..., \tau_n\}$ aus, bei dem jede Task i durch ihre größtmögliche Ausführungszeit C_i und – bei periodischen und sporadischen Tasks – ihre Periode charakterisiert ist. Sofern nichts anderes gesagt ist, nehmen wir an, dass die Periode auch die *Deadline* ist. Wenn die periodische

oder sporadische Natur der Tasks nicht relevant ist, können wir auch eine Menge von Jobs mit expliziten *Deadlines* d_i betrachten.

Für Multiprozessoren ist es nicht ausreichend, zu entscheiden, **wann** Tasks oder Jobs ausgeführt werden. Vielmehr müssen wir entscheiden, **wann** und **wo** wir diese ausführen wollen.

Für m identische Prozessoren gibt es die folgenden notwendigen Schranken für die Existenz eines *Schedules*:

$$\forall i : u_i \leq 1 \tag{6.16}$$

$$U_{sum} \leq m \tag{6.17}$$

6.3.1 Partitioniertes Scheduling

Unsere Darstellung in den nächsten Abschnitten basiert v.a. auf einem Buch von Baruah et al. [38] sowie auf einem Übersichtspapier von Davis et al. [121] und Folien von I. Puaut [461, 462]. Baruah et al. konzentrieren sich dabei auf sporadische Task-Systeme. Dies wird zum Teil dadurch motiviert, dass sporadische Systeme – im Gegensatz zu periodischen Task-Systemen – keine globale Zeitsynchronisation für das Bereitstellen von Jobs benötigen. Es reicht aus, wenn wir mit einem Zeitgeber sicherstellen, dass die minimalen Zeitabstände eingehalten werden.

Weiterhin beschränken wir uns zunächst auf **partitioniertes *Scheduling***. Dies bedeutet, dass jede Task einem bestimmten Prozessor zugeordnet ist. Die Verlagerung von Tasks (auf andere Prozessoren) ist nicht erlaubt. Partitioniertes *Scheduling* bei synchronen Ankunftszeiten kann mit *bin-packing* realisiert werden. *Bin-packing* [307] kann in einer auf das *Scheduling* angepassten Notation wie folgt beschrieben werden:

Definition 6.14: Sei $\tau = \{1,...,n\}$ eine Menge von Objekten, wobei jedes Objekt $i \in \tau$ eine Größe $c_i \in (0,1]$ besitzt. Sei $\pi = \{1,...m\}$ eine Menge von Behältern mit der Kapazität 1. Das Problem, eine Zuordnung $a : \tau \rightarrow \pi$ so zu finden, dass die Anzahl nicht-leerer Behälter $m \leq n$ minimal ist und dass die Kapazität der Behälter nicht überschritten wird, heißt *bin packing*-Problem.

Bin packing ist NP-hart [178]. Daher benötigen optimale Algorithmen wie der von Korf [306] große Laufzeiten. Die Formalisierung des *Scheduling*-Problems als *bin packing*-Problem zielt auf die Minimierung der Anzahl der Prozessoren m.

Für eine gegebene Anzahl m von Prozessoren ist es angemessener, *Scheduling* für synchrone Ankunftszeiten als ein Rucksack-Problem (engl. *knapsack problem)* zu modellieren, genauer gesagt, als ein 0/1-Multiples Knapsack-Problem (MKP).

Definition 6.15 (Martello [368]): Sei $\tau = \{1,...,n\}$ eine Menge von n Objekten, jedes mit einer Größe c_i und einem Nutzen b_i. Sei π eine Menge von m Rucksäcken, jeder mit einer Kapazität κ_k, mit $(m \leq n)$. Wir gehen davon aus, dass wir einen Teil der Objekte den Rucksäcken so zuordnen können $(a : \tau \rightarrow \pi)$, sodass die Größenbeschränkungen eingehalten werden:

$$\forall k : \sum_{i,a:i \to k} c_i \leq \kappa_k. \tag{6.18}$$

Das Problem der Auswahl einer Teilmenge von Objekten derart, dass der Gesamtprofit $\sum_i b_i$ für die Objekte in den Rucksäcken maximiert wird, heißt **0-1-*Multiple-Knapsack-Problem*** (MKP).

Unter Ausnutzung eines Algorithmus für das MKP können wir Jobs m Prozessoren zuordnen. Dabei würden wir evtl. nicht alle Jobs wirklich einem Prozessor zuordnen können. Bei identischen Prozessoren wären alle Kapazitäten gleich. Für uniforme Prozessoren können wir die Kapazität benutzen, um die Geschwindigkeiten bzw. die Performanz der Prozessoren zu modellieren. Das MKP ist ebenfalls NP-hart.

Aufgrund der Komplexität des *Schedulings* für synchrone Ankunftszeiten gibt es keine Hoffnung auf effiziente optimale Algorithmen für das allgemeine Problem. In der Praxis werden daher Heuristiken benutzt. Diese betrachten Tasks und Prozessoren in einer bestimmten Reihenfolge und sie unterscheiden sich in der Reihenfolge, die sie nutzen. Lopez et al. [356] haben verschiedene Heuristiken verglichen. Sie beschränken sich auf sogenannte **vernünftige** Zuordnungsalgorithmen.

Definition 6.16: Ein vernünftiger Zuordnungsalgorithmus (engl. *reasonable allocation (RA) algorithm*) ist ein Algorithmus, der nur dann einer Task keinen Prozessor zuordnet, wenn die Task auf keinen der Prozessoren der Plattform passt.

Definition 6.17: Ein vernünftiger abnehmender Zuordnungsalgorithmus (engl. *reasonable allocation decreasing (RAD) algorithm*) ist ein RA-Algorithmus, der Tasks in nicht-aufsteigender Reihenfolge der Auslastung betrachtet.

Die von Lopez et al. untersuchten Algorithmen kombinieren alle möglichen Kombinationen von zwei Eigenschaften:

1. Die Reihenfolge, in der Tasks betrachtet werden: Tasks können in absteigender Reihenfolge der Auslastung (bezeichnet als **D**), aufsteigender Reihenfolge der Auslastung (bezeichnet als **I**) und in beliebiger Reihenfolge (bezeichnet durch keinen Buchstaben) betrachtet werden.
2. Die Suchstrategie der Prozessorzuordnung. Wir nehmen an, dass die Prozessoren in einer bestimmten Weise geordnet sind. Die *first fit*-Strategie (bezeichnet als **FF**) wird dann eine Task dem ersten Prozessor zuordnen, auf den sie passt. Die *worst fit*-Strategie (bezeichnet als **WF**) wird dann eine Task dem Prozessor mit der größten verbleibenden Kapazität zuordnen. Die *best fit*-Strategie (bezeichnet als **BF**) wird dann eine Task dem Prozessor mit der kleinsten noch ausreichenden verbleibenden Kapazität zuordnen.

Es gibt insgesamt neun Kombinationen. Alle können effizient realisiert werden. Beispielsweise sieht der Algorithmus **FFD** wie folgt aus:

```
Sortiere die Task-Menge nach nicht-aufsteigenden Auslastungen u_i = C_i / T_i;
/* Annahme: Task-Menge wird entsprechend der Sortierung neu nummeriert; */
for (mt=0; mt ≤ m; mt++) K[mt] =1;      /* initialisiere die Kapazität */
for (i=1; i≤n; i++) {                                    /* für jede Task */
   for (mt=1; (u_i >K[mt]) and (mt≤m); mt++);  /* ausreichend Kapazität? */
   if (mt > m) mt=0;                        /* keine Lösung, wähle Index 0 */
   a[i]=mt;                      /* Gib Prozessorzuordnung im Array zurück */
   K[mt]=K[mt]-u_i;                  /* Aktualisiere verbleibende Kapazität */
}
```

Der heuristische Algorithmus ist sicherlich nicht optimal. Wir können uns fragen: wie weit sind wir vom Optimum entfernt? Viele Publikationen diskutieren obere Schranken für die Anzahl von zusätzlichen Prozessoren, die im Vergleich mit der minimalen Anzahl von Prozessoren beim optimalen *bin packing* benötigt werden. Die Publikation von Dosa [136] ist ein Beispiel dafür. Für Echtzeitsysteme ist eine andere Frage relevant: Gibt für eine gegebene Anzahl von Prozessoren eine obere Schranke für die Auslastung, bis zu der ein *Schedule* garantiert ist? Eine solche Schranke wurde von Lopez et al. [356] bewiesen:

Theorem 6.6: *Jeder RA-Algorithmus hat eine Auslastungsschranke nicht kleiner als*

$$U_{B1}(U_{max}) = m - (m-1)U_{max} \tag{6.19}$$

Beweis: Wenn eine Task mit einer Auslastung u_i nicht zugeordnet werden kann, dann muss jeder Prozessor bereits Tasks so zugeordnet bekommen haben, dass seine Auslastung $(1 - u_i)$ übersteigt. Die Gesamtauslastung über alle zugeordneten Tasks und τ_i einschließend muss dann größer sein als

$$m(1-u_i) + u_i = m - (m-1)u_i \tag{6.20}$$

$$\geq m - (m-1)U_{max} \tag{6.21}$$

Diese Bedingung muss erfüllt sein, damit kein *Schedule* möglich ist. □

Weiterhin definieren wir β als

$$\beta = \left\lfloor \frac{1}{U_{max}} \right\rfloor \tag{6.22}$$

β ist eine untere Schranke für die Anzahl von Tasks, die wir auf einem einzelnen Prozessor ausführen können. Wir nehmen an, dass auf jedem Prozessor EDF als lokales *Scheduling*-Verfahren genutzt wird. Lopez et al. zeigten das folgende Theorem:

Theorem 6.7: *Kein Zuordnungsalgorithmus kann eine Auslastungsschranke haben, die größer ist als*

$$U_{B2}(\beta) = \frac{\beta m + 1}{\beta + 1} \tag{6.23}$$

Beweis: Siehe Lopez et al. [356].

Lopez et al. haben auch gezeigt, dass **WF** und **WFI** Gleichung (6.19) als ihre untere Schranke haben, die anderen Algorithmen haben Gleichung (6.23) als ihre untere Schranke. Die Schranke in Gleichung (6.19) nähert sich 1, wenn U_{max} sich der 1 nähert:

$$U_{B1}(1) = 1 \tag{6.24}$$

Wenn U_{max} sich der 1 nähert, so nähert sich β ebenfalls der 1 und für U_{B2} gilt:

$$U_{B2}(1) = \frac{m+1}{2} \tag{6.25}$$

Verglichen mit der Schranke in Gleichung (6.24) erlaubt uns die Schranke in Gleichung (6.25) mehrere Prozessoren effizienter nutzen. Aufgrund dieser Schranken sind daher **WF** und **WFI** schlechter als die anderen sieben Algorithmen. Empirisch wurde gezeigt, dass **FFD** besser zu sein scheint als **FF** oder **FFI** und **BFD** scheint besser zu sein als **BF** und **BFI** [38]. Es gibt auch theoretische Anhaltspunkte, welche diese Beobachtung unterstützen [38].

Die skizzierten neun Algorithmen sind relativ einfache Algorithmen. Wir nehmen davon Abstand, ausgefeiltere Algorithmen für dasselbe Problem zu präsentieren, denn das Problem ist zu stark vereinfacht, um realistische Anwendungen widerzuspiegeln.

- Das *Scheduling*-Problem, wie es in diesem Abschnitt behandelt wurde, ist ein sehr eingeschränktes. Es gibt keine Reihenfolgebeschränkungen, keine Verdrängungen und nur identische Prozessoren.
- Partitioniertes *Scheduling* kann selbst dann, wenn Jobs ausführungsbereit sind, zu unbenutzten Prozessoren führen. Daher ist partitioniertes *Scheduling* nicht arbeitserhaltend (engl. *work conserving*). Folglich ist Optimalität nicht garantiert.

Mithin enthält dieser Abschnitt Basiswissen, aber praktische Probleme benötigen ausgefeiltere Ansätze, wie die in den nachfolgenden Abschnitten präsentierten.

6.3.2 Globales Scheduling mit dynamischen Prioritäten

Globales *Scheduling* kann verhindern, dass trotz vorhandener ausführbarer Jobs einige Prozessoren unbenutzt sind. Beim globalen *Scheduling* ist die Zuordnung von Prozessoren zu Tasks oder Jobs dynamisch. Dies gibt uns mehr Flexibilität, v.a. in Gegenwart von sich verändernden Arbeitslasten oder sich ändernden Verfügbarkeiten von Prozessoren. Aufgrund des Fortfalls von Ausführungsbeschränkungen werden die oberen Schranken der Auslastungen wie in den Gleichungen (6.19) und (6.23) ersetzt durch:

$$U_{sum} \leq m \tag{6.26}$$

Allerdings bringt es diese bessere Auslastung mit sich, dass zusätzlicher *Overhead* für *Scheduling*-Entscheidungen, Verdrängungen und Verlagerungen von Jobs entsteht.

Proportional fair (Pfair) scheduling

Die Grundidee von ***proportional fair (pfair)-Scheduling*** [39] ist es, jede Task mit einer Rate auszuführen, die proportional zu ihrer Auslastung ist[6]. Wenn wir beispielsweise eine 50-prozentige Auslastung (d.h. $u_i = 0,5$) für eine Menge von Tasks haben, dann sollte jede Task etwa zur Hälfte der Zeit ausgeführt werden, unabhängig von der Anzahl der Prozessoren. Für *pfair-Scheduling* setzen wir voraus, dass die Zeit quantisiert ist und mit ganzen Zahlen abgezählt ist. Jedes derart abgezählte Zeitintervall nennen wir einen **Zeitschlitz** (engl. *time slot*). Ebenso nehmen wir an, dass die C_i und T_i-Parameter von ganzen Zahlen repräsentiert werden.

Definition 6.18: Der **Verzug** (engl. *lag*) einer Task τ_i zur Zeit t in Bezug auf ein *Schedule* $\mathcal{S}$, bezeichnet als $lag(\mathcal{S}, \tau_i, t)$, ist die Differenz zwischen der Anzahl von Prozessorzeitschlitzen, die der Task schon zugewiesen wurden, und der Anzahl, welche die Task schon bekommen sollte:

$$lag(\mathcal{S}, \tau_i, t) = u_i * t - \sum_{u=0}^{t-1} alloc(\mathcal{S}, \tau_i, u) \tag{6.27}$$

Der erste Term ist die gewünschte Ausführungszeit der Task τ_i, der zweite Term ist die realisierte Ausführungszeit im *Schedule* $\mathcal{S}$. Ein *Schedule* heißt ***pfair-Schedule***, wenn der Verzug im Intervall $(-1, +1)$ bleibt.

Beispiel 6.10: Abb. 6.19 zeigt die realisierte Ausführungszeit als Funktion der realen Zeit. Die tatsächliche Ausführungszeit sollte die beiden gestrichelten Linien nicht erreichen.

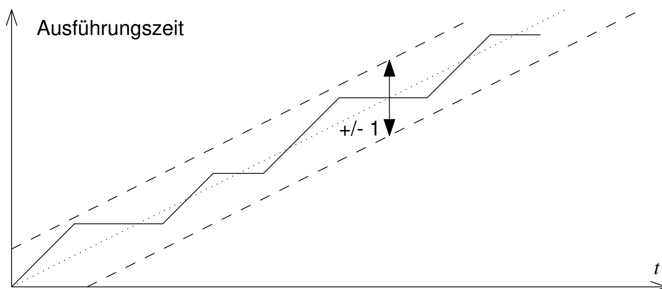


Abb. 6.19 Ausführungszeit als Funktion der Zeit

[6] Die Darstellung von *pfair-Scheduling* basiert auf Folien von I. Puaut [462].

Für das *pfair-Scheduling* teilen wir jede Task τ_i in Subtasks τ_i^j auf, wobei j die Ausführungsintervalle aufzählt. Für jede Subtask definieren wir eine Pseudo-Bereitstellungszeit $r(\tau_i^j)$ und eine Pseudo-*Deadline* $d(\tau_i^j)$:

$$r(\tau_i^j) = \left\lfloor \frac{j-1}{u_i} \right\rfloor \tag{6.28}$$

$$d(\tau_i^j) = \left\lceil \frac{j}{u_i} \right\rceil \tag{6.29}$$

Beispiel 6.11: Betrachte eine Task τ_i mit $C_i = 8, T_i = 11$. Mögliche Intervalle für die Anzahl der realisierten Ausführungszeitschlitze sind für jedes j in Abb. 6.20 gezeigt.

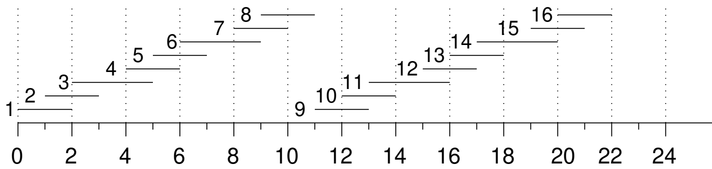


Abb. 6.20 Mögliche Intervalle realisierter Ausführungszeit

Beispielsweise gilt:

$$r(\tau_i^6) = \left\lfloor \frac{6-1}{8/11} \right\rfloor = \left\lfloor \frac{55}{8} \right\rfloor = 6$$

$$d(\tau_i^6) = \left\lceil \frac{6}{8/11} \right\rceil = \left\lceil \frac{66}{8} \right\rceil = 9$$

Daher muss die sechste Subtask von τ_i im Intervall (6:9) ausgeführt werden. ∇

Ein spezieller Ansatz für die Zuordnung von einer korrekten Anzahl von Zeitschlitzen wird in dem Buch von Baruah et al. [38] vorgestellt. Im Allgemeinen gibt es Variationen dieses Schemas: wir können EDF auf Pseudo-*Deadlines* anwenden oder wir können EDF modifizieren, indem wir Regeln definieren, die im Fall eines Gleichstands der *Scheduling*-Kriterien gelten. Es ist möglich, bis zu einer vollen Prozessor-Auslastung, d.h. $U_{sum} \leq m$, *Schedules* zu garantieren.

Potentiell leidet *pfair-Scheduling* unter einer großen Zahl von Verlagerungen hin zu anderen Prozessoren. Aufgrund der Überapproximation der Ausführungszeiten durch ganze Zahlen ist es nicht arbeitserhaltend. Es wurden Varianten vorgeschlagen, welche die Anzahl von Job-Verlagerungen reduzieren. Auch kann die Komplexität bei manchen Varianten reduziert werden. *Pfair-Scheduling* findet viele Anwendungen in Betriebssystemen, beispielsweise beim *Scheduling* in virtuellen Maschinen.

6.3.3 Globales Scheduling für feste Job-Prioritäten

G-EDF-*Scheduling*

Wir können versuchen, das zweidimensionale Problem mit Erweiterungen von *Scheduling*-Algorithmen für Einzelprozessoren zu lösen. Beispielsweise können wir *Global EDF* (G-EDF) benutzen. G-EDF definiert – wie EDF – Job-Prioritäten anhand der Nähe der nächsten *Deadlines*. Wenn m Prozessoren verfügbar sind, werden jene m Jobs ausgeführt, welche die höchsten Prioritäten unter allen verfügbaren Jobs haben. Offensichtlich sind solche Prioritäten abhängig vom Job und nicht nur von der Task. In einer globalen *Scheduling*-Strategie möchten wir *preemptions* von Jobs und Verlagerungen von Jobs zu anderen Prozessoren möglichst selten haben. Für G-EDF hängt die Häufigkeit davon ab, wie wir Tasks oder Jobs den Prozessoren zuordnen [189].

Lemma 6.3: *G-EDF ist nicht optimal.*

Beweis: Der Beweis erfolgt durch Gegenbeispiel, übernommen von Cho et al. [101]. Wir betrachten ein Task-System mit $m = 2$ und $C_1 = 3$, $D_1 = 4$, $C_2 = 2$, $D_2 = 3$, $C_3 = 2$ und $D_3 = 3$. In Abb. 6.21 (links) ist zu sehen, dass G-EDF aufgrund der früheren *Deadline* J_2 und J_3 zuerst einplant. J_1 verpasst die *Deadline*, obwohl ein *Schedule* möglich ist, wie in Abb. 6.21 (rechts) gezeigt.

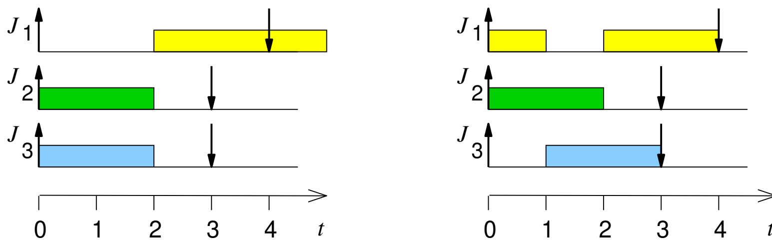


Abb. 6.21 **Links**: G-EDF verpasst *Deadline* bei $t = 4$; **rechts**: mögliches *Schedule*

□

Das Problem rührt offenbar von der Unfähigkeit her, den zweiten Prozessor für $t > 2$ zu nutzen.

Allgemein leidet G-EDF unter Anomalien wie dem sogenannten **Dhall-Effekt** [130]: periodische Task-Mengen, in denen eine Task eine Auslastung nahe Eins hat, können nicht mit G-EDF eingeplant werden.

Beispiel 6.12: Wir betrachten den Fall $n = m + 1$, um den Effekt zu demonstrieren.

$$\forall i \in [1..m] : T_i = 1, C_i = 2\varepsilon, u_i = 2\varepsilon \tag{6.30}$$

$$T_{m+1} = 1 + \varepsilon, C_{m+1} = 1, u_{m+1} = \frac{1}{1+\varepsilon} \tag{6.31}$$

Abb. 6.12 zeigt ein entsprechendes *Schedule*. Anfangs werden nur Tasks $\tau_1, .., \tau_m$ ausgeführt. Die Ausführung von Task τ_{m+1} startet erst, nachdem die ersten m Tasks ihre Ausführung beendet haben. Task τ_{m+1} verpasst ihre *Deadline*. Die Anwesenheit einer einzigen Task τ_{m+1} mit hoher Auslastung ist ausreichend, um eine *Deadline* bei $t = 1 + \varepsilon$ zu verpassen. Dies passiert, obwohl die Auslastung der anderen Tasks sehr klein ist. Tatsächlich kann die Auslastung für die Tasks $\tau_1, ..\tau_m$ beliebig klein sein und wir verpassen immer noch die *Deadline*. ∇

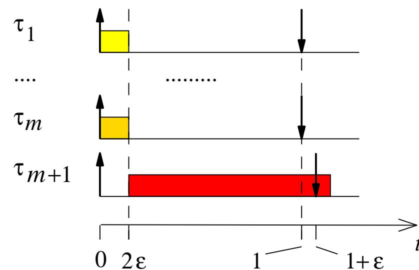


Abb. 6.22 Dhall-Effekt

Dies motiviert uns, Varianten von Algorithmen zu nutzen, die Tasks mit einer hohen Auslastung eine hohe Priorität zuweisen, unabhängig von der *Deadline* oder der Periode.

Algorithmus `fpEDF` ist ein solcher Algorithmus. Wir nehmen an, dass ein sporadisches Task-System $\tau = \{\tau_1, ...\tau_n\}$ mit impliziten *Deadlines* gegeben ist und dass Tasks nach nicht-aufsteigenden Task-Auslastungen u_i sortiert sind. Unser Ziel ist es, einen Ablaufplan (*Schedule*) für die Ausführung dieser Tasks auf m identischen Prozessoren zu entwickeln, wobei der Dhall-Effekt vermieden werden soll. Der `fpEDF`-Algorithmus arbeitet wie folgt [38]:

```
for  (i=1; i ≤ m − 1; i++){
  if (u_i >0.5) die Jobs von τ_i erhalten die höchste Priorität
          /* Bei Gleichstand erfolgt eine beliebige Auswahl */
  else break;
}               /* Verbleibende Jobs erhalten eine Priorität gemäß EDF. */
```

Mithin erhalten $m - 1$ Tasks mit der höchsten Auslastung die höchste Priorität, wenn ihre Auslastung größer ist als 0,5.

Theorem 6.8: *Algorithmus fpEDF hat eine Auslastungsschranke von mindestens* $\frac{m+1}{2}$.

Nach dem folgenden Theorem ist dies die beste Schranke, die wir erwarten können.

Theorem 6.9: *Kein Scheduling-Algorithmus mit festen Job-Prioritäten für m Prozessoren hat eine Auslastungsschranke größer als* $\frac{m+1}{2}$.

Der Beweis beider Theoreme kann bei Baruah [38] nachgelesen werden. Wie im Fall von partitioniertem *Scheduling* sind stärkere Schranken möglich, wenn die größte Auslastung bekannt ist.

Eine ähnliche Idee wird im EDF(k)-*Scheduling*-Algorithmus benutzt. Bei diesem Algorithmus erhalten k Tasks der höchsten Auslastung die höchste Priorität, wobei bei Auslastungsgleichheit beliebig entschieden wird. Alle anderen Tasks werden nach EDF eingeplant.

Theorem 6.10: *Sei τ ein sporadisches Task-System mit impliziten Deadlines. EDF(k) wird τ auf m homogenen Prozessoren einplanen, wobei*

$$m = (k-1) + \left\lceil \frac{U(\tau^{(k+1)})}{1-u_k} \right\rceil \tag{6.32}$$

ist und $U(\tau^{(k+1)})$ ist die Auslastung für die Task-Menge abzüglich der ersten k Tasks.

Auch für dieses Theorem kann der Beweis bei Baruah [38] gefunden werden.

EDZL-*Scheduling*

G-EDF kann *Deadlines* für Task-Mengen verpassen, für die ein *Schedule* möglich wäre. Wir können G-EDF verbessern, indem wir auch den Schlupf betrachten: der EDZL-Algorithmus benutzt G-EDF, sofern der Schlupf größer ist als Null (siehe Baruah et al. [38], Kapitel 20). Wenn der Schlupf eines Jobs allerdings Null wird, dann wird die Priorität dieses Jobs auf die höchste Priorität unter allen Jobs angehoben, einschließlich der gerade ausgeführten Jobs.

Beispiel 6.13: Wir betrachten das Beispiel in Abb. 6.23, welches von I. Puaut übernommen wurde [461]. Die Parameter in diesem Beispiel sind: $n = 3$, $m = 2$, $T_1 = T_2 = T_3 = 3$ und $C_1 = C_2 = C_3 = 2$. Aus der Abb. 6.23 (links) geht hervor, dass G-EDF für diese Parameter die *Deadlines* für τ_3 zu den Zeiten $t = 3n$ mit $n = 1, 2, 3, \ldots$ verpasst. Aus Abb. 6.23 (rechts) kann gesehen werden, dass EDZL

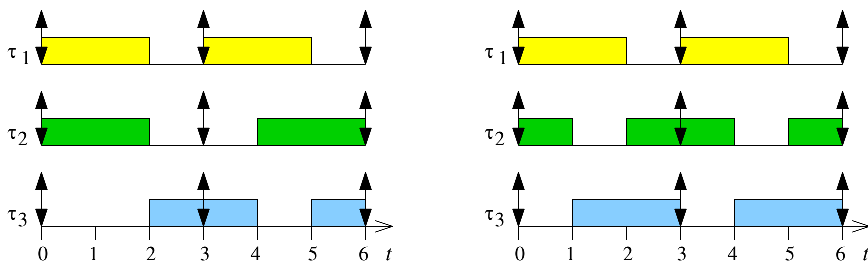


Abb. 6.23 G-EDF: **links**: verpasste *Deadlines*; **rechts**: Verbesserung durch EDZL

dagegen die *Deadlines* einhält. Die Details des Verhaltens hängen dabei etwas von der Prozessor-Zuordnung ab, die EDZL vornimmt. ∇

Choi et al. [102] haben gezeigt, dass EDZL in jedem Fall besser ist als EDF. Informell kann das wie folgt gezeigt werden[7]: Angenommen, $\mathcal{S}$ ist ein EDF-*Schedule* und $\mathcal{S}'$ ist ein EDZL-*Schedule* für dieselbe Task-Menge. Wenn ein Job zur Zeit t in EDZL eingeplant ist, aber nicht in EDF, dann verpasst er die *Deadline* in EDF, aber nicht in EDZL. Das *Schedule* bleibt dasselbe, wenn beide Strategien den Job einplanen. Damit gilt für den ersten Zeitpunkt, an dem $\mathcal{S}$ von $\mathcal{S}'$ verschieden ist, das Folgende:

- entweder EDZL bleibt möglich, aber EDF nicht oder
- EDZL und EDF liefern kein *Schedule*.

Daher ist EDZL in jedem Fall besser als EDF. Piao et al. [452] haben die folgende Auslastungsschranke für EDZL nachgewiesen

$$U_{sum} \leq \frac{m+1}{2} \tag{6.33}$$

6.3.4 Globales Scheduling für feste Task-Prioritäten

Globales *Rate-Monotonic Scheduling*

Ähnlich wie wir EDF zu G-EDF erweitern, können wir auch RMS zu einem Verfahren für das Multiprozessor-Scheduling, genannt G-RM, erweitern. Dabei gibt es für G-RM eine Anomalie bezüglich der Abschwächung von Anforderungen:

Lemma 6.4: *Bei G-RM kann es Fälle geben, in denen ein Schedule für ein bestimmtes Task-System existiert, aber in denen es kein Schedule gibt, wenn wir Perioden verlängern.*

Beweis: Wir beweisen die Existenz dieser Anomalie durch ein Beispiel: Wir betrachten ein Task-System mit den Parametern $m = 2$, $n = 3$, $T_1 = 3$, $C_1 = 2$, $T_2 = 4$, $C_2 = 2$, $T_3 = 12$ und $C_3 = 7$. Abb. 6.24 zeigt das dafür mit G-RM erzeugte *Schedule*.

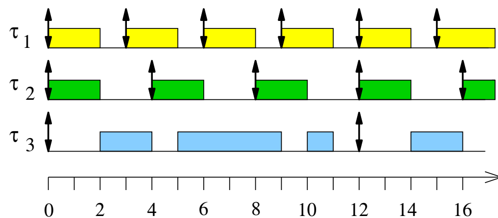


Abb. 6.24 Mit G-RM erzeugtes *Schedule*

[7] Der Hinweis auf diese informelle Erklärung stammt von J.J. Chen, TU Dortmund.

Wir verpassen die *Deadline* für τ_3, wenn wir die Periode von τ_1 auf $T_1 = 4$ verlängern (siehe Abb. 6.25).

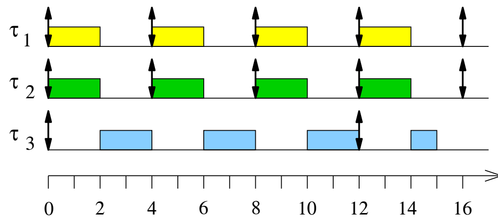


Abb. 6.25 Mit G-RM erzeugtes *Schedule* verpasst *Deadline* bei $t = 12$

Dieses un-intuitive Ergebnis macht Beweise und Beispiele komplizierter als im Einzelprozessor-Fall. □

Das Theorem 6.4 vom kritischen Zeitpunkt für Einzelprozessoren (siehe Seite 341) gilt ebenfalls nicht für Multiprozessoren.
Für G-RM wurde die folgende Auslastungsgrenze bewiesen [94]:

Theorem 6.11: *Ein periodisches oder sporadisches Task-System τ mit impliziten Deadlines mit der Auslastung*

$$U_{sum} \leq \frac{m}{2}(1 - U_{max}(\tau)) + U_{max}(\tau) \tag{6.34}$$

kann mit G-RM erfolgreich auf einem homogenen m-Prozessorsystem (mit Einheitsgeschwindigkeit) eingeplant werden [50].

G-RM leidet ebenfalls unter dem Dhall-Effekt: In Gleichung (6.34) ist zu sehen, dass U_{sum} sich Null annähert, wenn U_{max} gegen Eins geht. Wie G-EDF kann der Algorithmus die Verfügbarkeit mehrerer Prozessoren nicht voll ausschöpfen.

Daher wurde der Algorithmus RM-US(ξ) mit einer Schwelle ξ vorgeschlagen, wobei US für *utilization threshold* steht. Gegeben sei ein sporadisches Task-System mit impliziten *Deadlines*, wobei die Tasks $\tau = \{\tau_1, ...\tau_n\}$ gemäß einer nicht-aufsteigenden Reihenfolge der Auslastungen u_i sortiert seien. Bis zu $(m - 1)$ Tasks mit einer hohen Auslastung sollen auf bis zu $m - 1$ identischen Prozessoren eingeplant werden. Die verbleibenden Prozessoren dienen der Ausführung der verbliebenen Tasks. RM-US(ξ) arbeitet wie folgt:

```
for (i=1; i≤ m−1; i++) {
  if (u_i > ξ) τ_i wird die höchste Priorität zugewiesen
  else break;
}                /* die verbleibenden Tasks werden gemäß G-RM eingeplant*/
```

Theorem 6.12: *Für m Prozessoren mit Einheitsgeschwindigkeit hat RM-US(ξ) eine Auslastungsschranke von mindestens $\frac{m^2}{(3m-2)}$.*

Das Theorem wurde von Andersson et al. [16] bewiesen. Für $3m \gg 2$ nähert sich diese Schranke dem Wert $\frac{m}{3}$. Chen et al. [94] haben eine engere Schranke bewiesen.

RMZL-*Scheduling*

Für bestimmte Task-Mengen kann G-RM *Deadlines* verpassen, obwohl ein *Schedule* existiert. Eine mögliche Verbesserung ist RMZL-*Scheduling*. Beim RMZL-Verfahren nutzen wir (G-) RM-*Scheduling*, solange der Schlupf größer als Null ist. Wir setzen aber die Priorität eines Jobs auf den höchsten Wert, wenn der Schlupf für einen Job Null wird. RMZL-*Scheduling* ist RM-*Scheduling* überlegen, da wir die *Schedules* nur dann ändern, wenn RM-*Scheduling* eine *Deadline* verpasst hätte [38].

Partitioniertes *Scheduling* für explizite *Deadlines*

Partitioniertes *Scheduling* für Tasks mit expliziten *Deadlines* kann ähnlich erfolgen wie partitioniertes *Scheduling* für Tasks mit impliziten *Deadlines*, indem man das Sortieren nach der Auslastung ersetzt durch ein Sortieren nach der Dichte. Allerdings wird dieses Verfahren nicht empfohlen, da die Dichte in manchen Fällen unbeschränkt sein kann. Baruah et al. haben einen besseren Ansatz für partitioniertes *Scheduling* publiziert [38].

6.4 Abhängige Jobs auf homogenen Multiprozessor-Systemen

Die Ergebnisse der vorherigen Abschnitte stellen Basiswissen dar, aber die Beschränkung auf unabhängige Tasks und identische Prozessoren verhindert in vielen Fällen ihre Anwendung. Daher lassen wir jetzt diese Einschränkungen fallen. Zunächst geben wir die Beschränkung auf unabhängige Tasks auf. Wir konzentrieren uns dabei auf einige einfache Algorithmen aus dem Bereich der Entwurfsautomatisierung für elektronische Schaltungen. Sehr populär sind beispielsweise die Algorithmen *As-Soon-As-Possible* (ASAP)-*Scheduling*, *As-Late-As-Possible* (ALAP)-*Scheduling*, *List-Scheduling* (LS) und *Force-Directed-Scheduling* (FDS) im Bereich der automatischen Synthese aus einer algorithmischen Beschreibung, der so genannten *High-Level*-Synthese (HLS) [113].

6.4.1 As-Soon-As-Possible-Scheduling

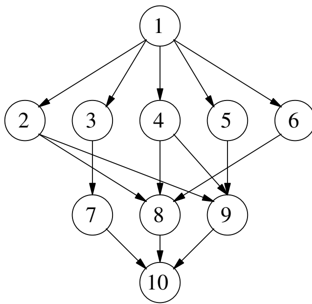
As-Soon-As-Possible-Scheduling (ASAP) versucht, unter Berücksichtigung der Reihenfolgebeschränkungen alle Tasks so früh wie möglich zu starten. In der *High-Level*-Synthese werden üblicherweise nur ganzzahlige Startzeiten ≥ 0 betrachtet. Verdrängungen sind nicht erlaubt. Die Zuordnung zu bestimmten Prozessoren erfolgt erst nach der Bestimmung der Startzeiten. Deshalb bietet ASAP-*Scheduling* auch nur eine Abbildung auf Task-Startzeiten:

$$\mathcal{S} : \tau \rightarrow \mathbb{N}_0 \tag{6.35}$$

Bei diesem Algorithmus setzen wir voraus, dass alle Ausführungszeiten bekannt sind und dass diese unabhängig sind von dem Prozessor, auf dem die Tasks ausgeführt werden, d.h. wir nehmen an, dass die Prozessoren homogen sind. Der Algorithmus berücksichtigt keine Beschränkungen hinsichtlich der Anzahl der Prozessoren und setzt voraus, dass die benötigte Anzahl von Prozessoren zur Verfügung steht. Der Algorithmus arbeitet wie folgt:

```
for (t=0; solange es nicht eingeplante Tasks gibt; t++)   {
   τ'={nicht eingeplante Tasks, deren Vorgänger beendet sind};
   Setze die Startzeit aller Tasks in τ' auf t;
}
```

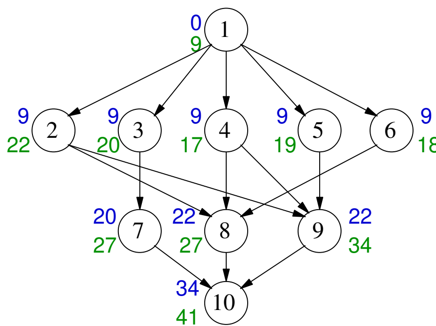
Beispiel 6.14: Wir nehmen an, dass der Task-Graph aus Abb. 6.26 (links) gegeben ist. Jeder mit i bezeichnete Knoten repräsentiert eine Task τ_i. Die rechte Seite der Abb. 6.26 enthält die von uns angenommenen Ausführungszeiten.



Task	C_i
1	9
2	13
3	11
4	8
5	10
6	9
7	7
8	5
9	12
10	7

Abb. 6.26 Links: Task-Graph; **rechts**: Ausführungszeiten

Das ASAP-*Scheduling* wird das in Abb. 6.27 gezeigte *Schedule* erzeugen.



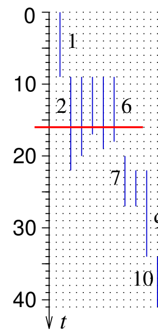


Abb. 6.27 Links: zeitlich eingeplanter Task-Graph; **rechts**: Zeitachse

Blaue Zahlen bedeuten Startzeiten, grüne Zahlen bedeuten das jeweilige Ausführungsende. Tasks τ_2 bis τ_6 starten alle unmittelbar, nachdem Task τ_1 beendet ist. Tasks τ_7 bis τ_9 starten ebenfalls sofort nachdem der letzte ihrer Vorgänger die Ausführung beendet hat. Die rote Linie in Abb. 6.27 (rechts) zeigt, dass im Maximum fünf Prozessoren benötigt werden, da ASAP-*Scheduling* weder eine Grenze für die Anzahl der Prozessoren noch das Ziel einer ausgewogenen Prozessorbelastung berücksichtigt. ∇

ASAP-*Scheduling* minimiert den *Makespan*, da alle Tasks so früh wie möglich ausgeführt werden. Der vorgestellte Algorithmus kann so erweitert werden, dass als Ausführungszeiten auch reelle Zahlen benutzt werden. ASAP-*Scheduling* ist von linearer Komplexität, vorausgesetzt wir benutzen eine intelligente Methode, um τ' zu berechnen. Der Algorithmus kann auch im täglichen Leben angewandt werden: er entspricht der Situation, in der jede Person begierig alle Arbeiten so früh wie möglich startet.

6.4.2 As-Late-As-Possible-Scheduling

As-Late-As-Possible (ALAP)-*Scheduling* ist der zweite einfache Algorithmus. Beim ALAP-*Scheduling* werden alle Tasks so spät wie möglich gestartet. Der Algorithmus funktioniert wie folgt:

```
for (t=0; solange es nicht eingeplante Tasks gibt; t--) {
     τ'={alle Tasks ohne Abhängigkeit zu einer nicht eingeplanten Task};
     Setze die Startzeit aller Tasks in τ' auf (t - ihre Ausführungszeit);
}
Schiebe alle Startzeiten so, dass die erste Task zur Zeit t=0 startet.
```

Der Algorithmus betrachtet zunächst die Tasks, von denen keine weitere Task abhängt. Es wird angenommen, dass diese Tasks zum Zeitpunkt 0 enden. Ihre Startzeit wird dann aus ihrer Ausführungsdauer berechnet. Danach iteriert die Schleife rückwärts über Zeitschritte. Wenn ein Zeitschritt erreicht wird, zu dem eine Task spätestens beendet sein soll, wird die Startzeit dieser Task berechnet und die Task eingeplant. Nach dem Ende der Schleife werden alle Zeiten so angepasst, dass die erste Task zum Zeitpunkt 0 startet. Wir könnten ALAP-*Scheduling* auch als eine Variante des ASAP-*Scheduling* betrachten, die am „anderen" Ende des Graphen beginnt.

Beispiel 6.15: Für den Task-Graphen in Abb. 6.26 würde ALAP-Scheduling das in Abb. 6.28 gezeigte Ergebnis generieren. Die Farbkodierung ist dieselbe wie beim ASAP-Beispiel. Jede Task wird so spät wie möglich beendet. Insbesondere werden Tasks τ_7 bis τ_9 erst zur Zeit 34 beendet. Tasks τ_4 bis τ_6 sind später als beim ASAP-*Schedule* abgeschlossen. Tasks τ_1, τ_2, τ_9 und τ_{10} werden wie beim ASAP-*Schedule* eingeplant, denn diese Tasks bestimmen den *Makespan*. Wir sagen, dass die Tasks, welche den *Makespan* bestimmen, auf dem **kritischen Pfad** liegen. Der roten Linie ist zu entnehmen, dass die Lösung in Abb. 6.28 fünf Prozessoren benötigt.

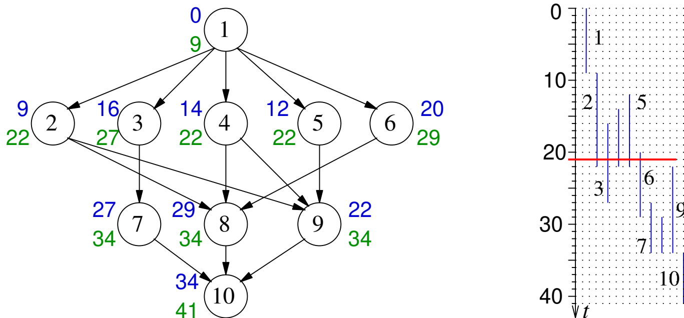


Abb. 6.28 **Links**: Mit ALAP-*Scheduling* eingeplanter Task-Graph; **rechts**: Zeitachse ∇

Auch ALAP-*Scheduling* kann im täglichen Leben eingesetzt werden. Es entspricht einer Situation, in der alle Aufgaben so lange wie möglich aufgeschoben werden. Beim ALAP-*Schedule* werden viele Prozessoren benötigt, wenn der Task-Graph an seinem unteren Ende breit ist. Dementsprechend fällt bei diesem Vorgehen im täglichen Leben zum Schluss viel Arbeit an.

6.4.3 List-Scheduling

Beim *List-Scheduling* (LS) versuchen wir, die geringe Komplexität von ASAP- und ASAP-*Scheduling* beizubehalten, aber auf die Anzahl verfügbarer Prozessoren Rücksicht zu nehmen. Die Prozessoren können verschiedenen Typs sein, aber wir nehmen immer noch an, dass es eine Eins-zu-eins-Beziehung zwischen Tasks und Prozessortypen gibt. Somit können die Prozessoren heterogen sein, aber die kritische Abbildung von Tasks auf Prozessortypen wird nicht durch LS erzeugt.

Wir nehmen an, dass wir eine Menge L von Prozessortypen haben. LS respektiert obere Schranken B_l für die Anzahl von Prozessoren für jeden Typ $l \in L$.

LS setzt voraus, dass eine Prioritätsfunktion definiert ist, welche die **Dringlichkeit** angibt, mit der eine bestimmte Task τ_i eingeplant werden muss. Die folgenden Dringlichkeitsmetriken finden dabei Anwendung [528]:

- Die **Beweglichkeit** (engl. *mobility*) ist die Differenz zwischen den Startzeiten für die ASAP- und ALAP-*Schedules*. Abb. 6.29 (links) zeigt die Mobilität für unser Beispiel in rot. Offenbar ist das *Scheduling* **dringend** für alle Tasks auf dem kritischen Pfad, denn deren Mobilität ist 0.
- Die **Anzahl an Nachfolgern** ist die Anzahl von Knoten im Baum unterhalb des aktuellen Knotens τ (siehe Abb. 6.29 (rechts)).
- Die **Pfadlänge** für einen Knoten τ_i ist definiert als die Länge des Pfades vom Startknoten τ_i bis zum Endknoten des Graphen G. Die Pfadlänge ist typischer-

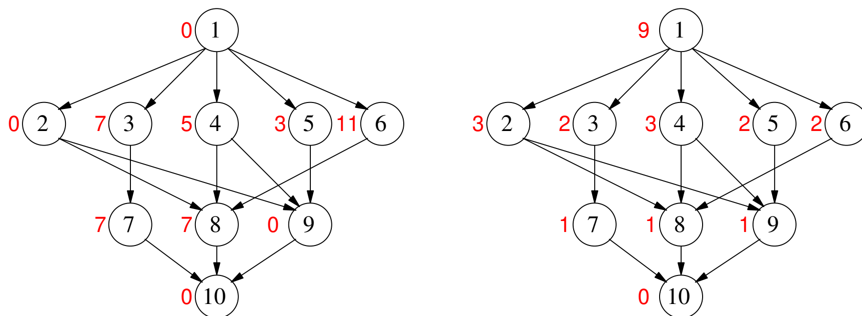


Abb. 6.29 Laufendes Beispiel: **links**: Mobilität; **rechts**: Zahl der Nachfolger;

weise mit der Ausführungszeit der Knoten gewichtet, vorausgesetzt, dass diese Information bekannt ist. In Abb. 6.30 (links) wurde die Pfadlänge eingetragen.

LS verlangt die Kenntnis des einzuplanenden Task-Graphen, einer Abbildung jedes Knotens des Graphen auf den entsprechenden Ressourcentyp $l \in L$, einer Prioritätsfunktion (wie eben erklärt) und der Ausführungszeit für jede Task τ_i in τ. LS versucht dann, Knoten maximaler Priorität jedem der Zeitschritte zuzuordnen, wobei die Randbedingungen nicht verletzt werden [528]:

```
for (t=0; solange nicht eingeplante Tasks vorh.; t++)  /* Zeit-Schleife */
  for (l ∈ L) {                             /* Schleife über Ressourcentypen */
    τ*_{t,l} = Menge der Tasks vom Typ l, die zur Zeit t noch ausgeführt werden;
    τ**_{t,l} = Menge der Tasks vom Typ l, die zur Zeit t starten können;
    Berechne Menge τ'_t ⊆ τ**_{t,l} maximaler Priorität, sodass
    |τ'_t| + |τ*_{t,l}| ≤ B_l .                     /* Anzahl der Tasks ≤ Schranke?/
    Setze Startzeiten aller τ_i ∈ τ'_t auf t: s_i = t;
  }
```

Beispiel 6.16: Abb. 6.30 zeigt das Ergebnis der Anwendung von *List-Scheduling* mit der Pfadlänge als Prioritätsfunktion auf unser Beispiel von Abb. 6.26. Wir nehmen an, dass alle Prozessoren von demselben Typ sind und wir erlauben maximal drei Prozessoren ($B_1 = 3$). Zur Zeit 9 haben Tasks τ_2, τ_4 und τ_5 den längsten Pfad und daher die höchste Priorität. τ_4 beendet die Ausführung zur Zeit 17 und τ_3 wie auch τ_6 haben unter den verbleibenden Tasks die größte Pfadlänge. Wir nehmen an, dass wir τ_3 einplanen. τ_5 beendet die Ausführung zur Zeit 19 und τ_6 kann gestartet werden. Zum Zeitpunkt 28 werden τ_3 und τ_6 die Ausführung beenden, womit Prozessoren für τ_7 und τ_8 frei werden. τ_7 wird zur Zeit 35 fertig gestellt, wodurch die abhängige Task τ_{10} starten und zur Zeit 42 fertig gestellt sein kann. Der Abschluss erfolgt nur wenig später als in den ASAP- und ALAP-*Schedules*, obwohl nur drei Prozessoren zur Verfügung stehen. Für die Zuordnung zu konkreten Prozessoren bestehen noch Wahlmöglichkeiten.

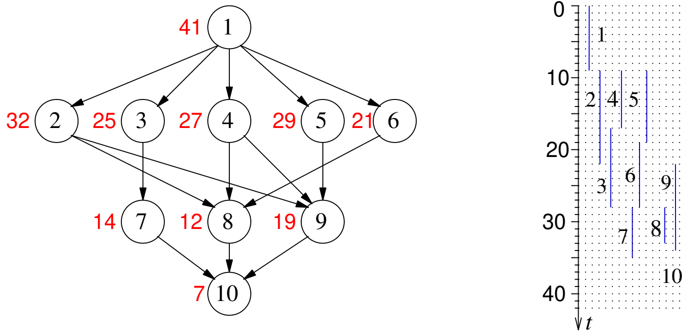


Abb. 6.30 **Links**: Task-Graph mit Pfadlängen; **rechts**: Zeitachse ∇

LS wählt ebenso wie ASAP- und ALAP-*Scheduling* für die Tasks keine Prozessoren aus, aber aufgrund des eingeschränkten Ressourcenmodells muss das auch nicht sein. Eine Erweiterung von LS auf reelle Zahlen als Ausführungszeiten ist möglich. Üblicherweise erzeugt der Algorithmus gute Ergebnisse und er kann an verschiedene Szenarien leicht angepasst werden. Deshalb ist LS ein beliebter *Scheduling*-Algorithmus für Tasks mit Reihenfolgebeschränkungen.

Force-directed scheduling (FDS) ist eine weitere beliebte Heuristik für abhängige Tasks. FDS zielt auf eine möglichst effiziente Nutzung der Prozessoren, indem es versucht, die Nutzung der Prozessoren über der Zeit möglichst gut auszubalancieren. Details finden sich bei Paulin et al. [449].

6.4.4 Optimales Scheduling mit Ganzzahliger Programmierung

Als nächstes beschreiben wir einen Ansatz, um Tasks auf verschiedenen Prozessoren zu verteilen, wobei die Entscheidungen auf einer mehr globalen Basis getroffen werden. Der Ansatz basiert auf der Ganzzahligen Linearen Programmierung (engl. *Integer Linear Programming* (ILP)) (siehe Anhang A). Auf diese Weise werden Randbedingungen und Zielkriterien explizit dargestellt. Wir benutzen dabei Material aus einer Publikation von Coscun et al. [112].

ILP-Modelle bestehen aus einer linearen Kostenfunktion und einer Menge an linearen Randbedingungen. Im Modell werden wir die folgenden Variablen benutzen:

$x_{i,k}$: = 1 wenn Task τ_i auf Prozessor π_k ausgeführt wird und =0 sonst

s_i : Startzeit von Task τ_i

f_i : Fertigstellungszeit von Task τ_i

C_i : Ausführungsdauer von Task τ_i

$b_{i,j}$: = 1 wenn Task τ_i vor τ_j auf demselben Prozessor ausgeführt wird, sonst =0

Wir nehmen an, dass unser Task-Graph $G = (\tau, E)$ einen gemeinsamen Ausgangsknoten τ_{exit} besitzt. Wir fügen einen solchen Knoten künstlich hinzu, wenn es ihn zunächst nicht geben sollte. Die Fertigstellungszeit dieses Knotens entspricht dem *Makespan* MS_{max}. Wir können diese Zeit als unsere zu minimierende Kostenfunktion verwenden. Daher können wir das Zielkriterium wie folgt ausdrücken:

$$Min(f_{\tau_{exit}}) \tag{6.36}$$

Nun sehen wir uns die Randbedingungen an. Erstens müssen wir verlangen, dass jede Task auf einem Prozessor ausgeführt wird:

$$\forall \tau_i \in \tau : \sum_{k \in \{1..m\}} x_{i,k} = 1 \tag{6.37}$$

Zweitens sind die verschiedenen Zeiten über die folgenden Gleichungen miteinander verbunden:

$$\forall \tau_i \in \tau : f_i = s_i + C_i \tag{6.38}$$

Drittens können die folgenden Gleichungen benutzt werden, um die Reihenfolgebedingungen einzuhalten:

$$\forall (\tau_i, \tau_j) \in E : s_j - f_i \geq 0 \tag{6.39}$$

Viertens müssen wir bei einem Einzelprozessor den Code in einer Reihenfolge ausführen, der durch die Variable $b_{i,j}$ bestimmt wird:

$$\forall (\tau_i, \tau_j) : f_i \leq s_j \text{ falls } b_{i,j} = 1 \tag{6.40}$$

Fünftens müssen wir berücksichtigen, dass ein Prozessor zu einer bestimmten Zeit nur eine Task ausführen kann. Dies kann auf die folgende Weise ausgedrückt werden:

$$\forall (\tau_i, \tau_j) : b_{i,j} + b_{j,i} = 1 \text{ falls } \exists\, \pi_k : x_{i,k} = x_{j,k} = 1 \tag{6.41}$$

Die Gleichungen (6.40) und (6.41) können auf die lineare Form gebracht werden, die für ein ILP-Modell erforderlich ist [112].

Das entstehende ILP-Modell kann einem ILP-Lösungsverfahren übergeben werden. ILP-Modelle, wie das vorgestellte, haben den Vorteil einer präzisen Modellierung des Entwurfsproblems und der Zielfunktion. Sie erlauben globale Optimierungen mittels mathematischer Methoden und gehen damit über die imperative Programmierung hinaus.

Das ILP-Problem ist NP-hart. Daher können die Laufzeiten von ILP-Lösungsverfahren groß werden. Allerdings hat es in den letzten Jahren große Fortschritte beim der Konzeption von ILP-Lösungsverfahren gegeben. Daher können moderat große Probleme in akzeptabler Zeit gelöst werden. Aufgrund ihrer Komplexität können sie aber nicht für wirklich große Probleme benutzt werden, weil für diese die

Laufzeiten evtl. unannehmbar groß werden. Für mittelgroße Probleme ist dennoch eine exakte Optimierung möglich und bei größeren Problemen können diese Modelle als Ausgangsbasis für Heuristiken eingesetzt werden.

6.5 Abhängige Jobs auf heterogenen Multiprozessoren

6.5.1 Problem-Beschreibung

Nach dem Streichen der Beschränkung auf unabhängige Tasks wollen wir nunmehr auch die Beschränkung auf homogene Prozessoren aufheben. Wir nehmen an, dass die Ausführungszeiten auf den verschiedenen Prozessoren unserer Ausführungsplattform $\pi = \{\pi_1, ..., \pi_m\}$ nicht miteinander in einer Beziehung stehen. Gemäß der Klassifikation von Pinedo betrachten wir damit den Fall $(R_m|r_i, prec, ...|...)$. Dies erlaubt es uns, Plattformen mit einer Mischung von Ausführungseinheiten zu modellieren, einschließlich FPGAs und GPUs.

Die Theorie der resultierenden *Scheduling*-Probleme ist nicht umfangreich untersucht worden. Folglich schreiben Baruah et al. im Kapitel 22 ihres Buchs [38]: „*although unrelated multiprocessors are becoming increasingly more important in real-time systems implementation, the resulting scheduling theoretic study of such systems is, relatively speaking, still in its infancy*." Einige erste Ergebnisse wurden im Buch von Baruah et al. vorgestellt, aber wir ziehen es hier vor, Methoden aus der Entwurfsautomatisierung von Schaltkreisen vorzustellen. Diese Methoden sind in der Lage, realistische Entwurfsaufgaben zu lösen, unter Verzicht auf eine Garantie der Optimalität.

6.5.2 Statisches Scheduling mit lokalen Heuristiken

Nachfolgend werden wir den *Heterogeneous-Earliest-Finish-Time* (HEFT)- und den *Critical-Path-On-a-Processor* (CPOP)-Algorithmus beschreiben. Diese beiden Algorithmen bilden Tasks eines Task-Graphen auf ein heterogenes Multiprozessor-System $\pi = \{\pi_1, ..., \pi_m\}$ ab [545]. Diese beiden Algorithmen sind Standard-Beispiele für schnelle Algorithmen. In gewisser Weise erweitern sie ASAP- und ALAP-*Scheduling* auf heterogene Prozessoren. Wir benutzen die nachfolgend beschriebene Notation:

- Wir nehmen an, dass der Task-Graph einen gemeinsamen Eingangsknoten τ_{entry} besitzt. Sollte ein solcher Knoten anfänglich nicht vorhanden sein, so fügen wir einen Knoten mit einer Ausführungszeit von 0 und ohne Kommunikationsanforderungen künstlich hinzu.
- Wir nehmen an, dass der Task-Graph einen gemeinsamen Ausgangsknoten τ_{exit} besitzt. Sollte ein solcher Knoten anfänglich nicht vorhanden sein, so fügen wir

einen Knoten mit einer Ausführungszeit von 0 und ohne Kommunikationsanforderungen künstlich hinzu.

- Die Matrix $C = (c_{i,k})$ bezeichnet die Ausführungszeit von Task τ_i auf Prozessor π_k.
- Die Matrix $B = (b_{k,l})$ bezeichnet die Kommunikationsbandbreite für die Kommunikation vom Prozessor π_k zum Prozessor π_l.
- Die Matrix $data = (data_{i,j})$ kennzeichnet das Datenvolumen, das von Task τ_i nach Task τ_j übertragen werden muss.
- Der Vektor $\kappa = (\kappa_k)$ enthält die Initialisierungskosten für Kommunikation auf dem Prozessor π_k.
- Die Matrix $H = (h_{i,j,k,l})$ repräsentiert die Kommunikationskosten für die Kommunikation von der Task τ_i zur Task τ_j unter der Annahme, dass die Task τ_i auf den Prozessor π_k abgebildet wird und dass die Task τ_j auf den Prozessor π_l abgebildet wird[8].
  In unserer Notation werden wir generell den Index i für den Ausgangsknoten bei Präzedenzrelationen bzw. Datenabhängigkeiten verwenden und den Index k für den Prozessor, auf den dieser abgebildet wird.
  In ähnlicher Weise steht j für das Ziel bei Präzedenzrelationen und l für den dazugehörigen Prozessor.
- Für eine Abbildung auf Prozessoren π_k und π_l stellt $h_{i,j,k,l}$ die Kommunikationskosten von Task τ_i zu Task τ_j dar:

$$h_{i,j,k,l} = \kappa_k + \frac{data_{i,j}}{b_{k,l}} \text{ falls } k \neq l \tag{6.42}$$

$$= 0 \text{ falls } k = l \tag{6.43}$$

- Die durchschnittlichen Kommunikationskosten sind definiert als

$$\overline{h_{i,j}} = \overline{\kappa} + \frac{data_{i,j}}{\overline{B}} \tag{6.44}$$

  Dabei ist $\overline{\kappa}$ die durchschnittliche Zeit für das Starten der Kommunikation und $\overline{B}$ ist die durchschnittliche Kommunikationsbandbreite.
- Die früheste Startzeit für Task τ_i auf Prozessor π_k für ein partielles *Schedule* ist

$$s_e(\tau_i, \pi_k) \text{ mit } \forall k : s_e(\tau_{entry}, \pi_k) = 0 \tag{6.45}$$

- Die früheste Fertigstellungszeit für Task τ_i auf Prozessor π_k für ein gegebenes partielles *Schedule* heißt

$$f_e(\tau_i, \pi_k) \text{ mit } f_e(\tau_{entry}, \pi_k) = c_{entry,k} \tag{6.46}$$

- Auf der Basis der Entscheidung, Task τ_i auf Prozessor π_k abzubilden, können die tatsächliche Startzeit $s(\tau_i, \pi_k)$ und die tatsächliche Fertigstellungszeit $f(\tau_i, \pi_k)$ berechnet werden.

[8] Die Indices k und l sind in der ursprünglichen Veröffentlichung nicht explizit vorhanden.

$s_e(\tau_j, \pi_l)$ und $f_e(\tau_j, \pi_l)$ können aus einem partiellen *Schedule* iterativ wie folgt berechnet werden:

$$s_e(\tau_j, \pi_l) = max \left\{avail(l), max_{\tau_i \in pred(\tau_j)}(f(\tau_i) + h_{i,j,k,l})\right\} \quad (6.47)$$

$$f_e(\tau_j, \pi_l) = c_{j,l} + s_e(\tau_j, \pi_l) \quad (6.48)$$

wobei $pred(\tau_j)$ die Menge der unmittelbaren Vorgänger-Tasks von Task τ_j ist, k ist der Prozessor, auf den Task τ_i im partiellen *Schedule* abgebildet ist und $avail(l)$ ist die Zeit, zu der Prozessor π_l die Ausführung der letzten Task beendet hat. Der max-Ausdruck im inneren Term bestimmt die Zeit, zu der alle Daten, die von τ_j benötigt werden, beim Prozessor π_l angekommen sind.

- Als Zielkriterium nehmen wir den $Makespan$. Dieser wird aus dem Abschluss der Berechnungen auf dem $exit$-Knoten bestimmt:

$$MS_{max} = f(\tau_{exit}) \quad (6.49)$$

- Die durchschnittliche Ausführungszeit $\overline{c_i}$ ist das Mittel der Ausführungszeiten $c_{i,k}$ über alle Prozessoren k.
- Der Aufwärts-Rang (engl. *upward rank*) $rank_u(\tau_i)$ einer Task τ_i ist die Länge des kritischen Pfades vom $exit$-Knoten bis zum Knoten τ_i (diesen einschließend):

$$rank_u(\tau_{exit}) = \overline{c_{exit}} \quad (6.50)$$

$$rank_u(\tau_i) = \overline{c_i} + \max_{\tau_j \in succ(\tau_i)} (\overline{h_{i,j}} + rank_u(\tau_j)) \quad (6.51)$$

Dabei ist $succ(\tau_i)$ die Menge unmittelbaren Nachfolger von τ_i im Task-Graphen.

- Der Abwärts-Rang (engl. *downward rank*) $rank_d(\tau_j)$ ist die Länge des kritischen Pfades vom $start$-Knoten bis zum Task-Knoten τ_j (ohne τ_j selbst):

$$rank_d(\tau_{entry}) = 0 \quad (6.52)$$

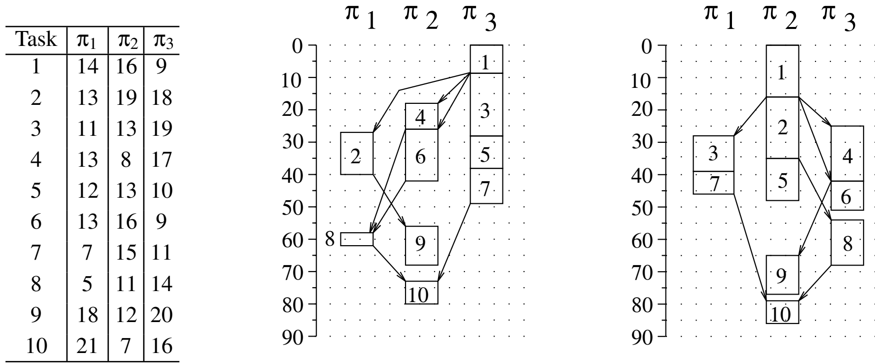
$$rank_d(\tau_j) = \max_{\tau_i \in pred(\tau_j)} (rank_d(\tau_i) + \overline{c_i} + \overline{h_{i,j}}) \quad (6.53)$$

Der HEFT-Algorithmus lässt sich wie folgt beschreiben:

```
Setze die Berechnungs- und die Kommunikationskosten auf ihre Mittelwerte;
Berechne rank_u(τ_i)∀τ_i (Durchlauf nach oben, startend bei τ_exit);
Sortiere Tasks in nicht-aufsteigender Folge von rank_u-Werten;
while es gibt ungeplante Tasks in der Liste do {
Wähle die erste Task τ_i in der Liste für das Scheduling;
  for jeden Prozessor π_k ∈ π {
   Berechne f_e(τ_i, π_k) mit Einfüge-orientiertem Scheduling[9];
    }
  Weise Task τ_i dem Prozessor π_k zu, der f_e(τ_i, π_k) minimiert;
}
```

[9] Der Algorithmus sucht eine ausreichend große Lücke unter den bereits eingeplanten Tasks derart, dass eine Zuweisung zu dieser Lücke die Reihenfolgebeschränkungen einhält.

Beispiel 6.17: Wir nehmen an, dass die Ausführungszeiten durch die Tabelle in Abb. 6.31 (links) gegeben sind. Für jede Task wurden die Ausführungszeiten in Abb. 6.26 (rechts) als Minimum über die drei Prozessoren gewählt.



Task	π_1	π_2	π_3
1	14	16	9
2	13	19	18
3	11	13	19
4	13	8	17
5	12	13	10
6	13	16	9
7	7	15	11
8	5	11	14
9	18	12	20
10	21	7	16

Abb. 6.31 Links: Ausführungszeiten; **Mitte**: Ergebnisse für HEFT; **rechts**: Ergebnisse für CPOP

Abb. 6.31 (Mitte) zeigt das *Schedule*, welches HEFT für den DAG in Abb. 6.26 (links) erzeugt. Es ist kein kurzes *Schedule* wie beim ASAP/ALAP-*Scheduling*, da wir jetzt Reihenfolge- **und Ressourcen**beschränkungen beachten. ∇

Der CPOP-Algorithmus konzentriert sich auf den kritischen Pfad im DAG und er benutzt verschiedene Task-Prioritäten und verschiedene Strategien für die Zuordnung von Prozessoren. Der CPOP-Algorithmus arbeitet wie folgt:

```
Setze die Berechnungs- und die Kommunikationskosten auf ihre Mittelwerte;;
Berechne ∀i : rank_u(τ_i) und rank_d(τ_i);
Berechne ∀i : priority(τ_i) = rank_d(τ_i) + rank_u(τ_i);
|CP| = priority(τ_entry); /* Länge des kritischen Pfades */
SET_CP = {τ_entry}, mit SET_CP: Menge der Tasks auf dem kritischen Pfad;
τ_i = τ_entry;
while τ_i ist nicht der exit-Task {
  Wähle τ_j ∈ succ(τ_i), mit priority(τ_j) == |CP|.
  SET_CP = SET_CP ∪ {τ_j};
  τ_i = τ_j
};
Wähle Prozessor π_CP, der Ausführungszeit auf kritischem Pfad minimiert;
Initialisiere eine Warteschlange mit der entry-Task;
while es gibt eine ungeplante Task in der Warteschlange {
  Wähle die Task der höchsten Priorität τ_i aus der Warteschlange;
  if τ_i ∈ SET_CP {weise τ_i Prozessor π_CP zu}
  else {weise Task τ_i dem Prozessor zu, der f_e(τ_i, π_k) minimiert};
 Aktualisiere Warteschlange mit Nachfolgern von τ_i wenn sie bereit werden;
}
```

Beispiel 6.18: Abb. 6.31 (rechts) zeigt das mit CPOP erzeugte *Schedule* . ∇

HEFT und CPOP sind relativ schnelle und einfache Algorithmen. Diese Algorithmen nutzen verschiedene Näherungen (wie z.B. durchschnittliche Kommunikationskosten) und Heuristiken. HEFT und CPOP wurden für dieses Buch ausgewählt, um Schwierigkeiten mit *Scheduling*-Algorithmen für heterogene Prozessoren zu demonstrieren. Allerdings ist es möglich, im Vergleich zu diesen beiden Algorithmen bessere Ergebnisse zu erzielen. Beispielsweise haben Kim et al. [295] komplexere Algorithmen beschrieben, die bessere Ergebisse liefern. Castrillon et al. [86] haben ein *Scheduling* für KPNs entwickelt, welches ebenfalls den *Makespan* minimieren soll.

6.5.3 Statisches Scheduling mit Ganzzahliger Programmierung

Die Ganzzahlige Lineare Programmierung kann auch auf heterogene Prozessoren angewandt werden. Ein Ansatz hierzu wurde von Maculan et al. [362] publiziert. Sehr wichtig ist, dass dabei prozessorabhängige Ausführungszeiten betrachtet werden. Allerdings bedürfen die gezeigten Gleichungen noch einiger Überarbeitungen, bevor sie vollständig die Form eines Ganzzahligen Optimierungsproblems haben. Weiterhin ist es auch möglich, Techniken der *High-Level-Synthese* [43, 315] zu adaptieren.

In vielen der Publikationen werden Optimierungen für eine einzelne Metrik (ein Zielkriterium) betrachtet. Im allgemeinen sollten mehrere Zielkriterien betrachtet werden. Beispielsweise beschreiben Fard et al. [162] einen Algorithmus, der mehrere Zielkriterien betrachtet.

6.5.4 Statisches Scheduling mit Evolutionären Algorithmen

Verfahren auf der Basis der Ganzzahligen Programmierung leiden unter potentiell großen Rechenzeiten. In vielen Fällen erlauben evolutionäre Algorithmen bei akzeptablen Rechenzeiten eine bessere Optimierung. Wir werden dies am Beispiel der *Distributed Operation Layer* (DOL)-Werkzeuge von der ETH Zürich zeigen [537]. Diese Werkzeuge beinhalten folgende Funktionen:

- **Automatische Selektion von Prozessorarchitekturen:** die Prozessortypen können vollständig heterogen sein. Mögliche Optionen sind Standard-Prozessoren, Mikrocontroller, DSP-Prozessoren, FPGAs usw.
- **Automatische Selektion von Kommunikationstechniken:** verschiedene Verbindungsschemata wie zentrale Busse, hierarchische Busse, Ringe usw. sind realisierbar.
- **Automatische Selektion von *Scheduling* und Arbitrierung:** die in DOL verfügbaren Werkzeuge zur Erkundung des Entwurfsraumes wählen automatisch zwischen *Rate Monotonic Scheduling*, EDF, TDMA- und prioritätsbasierten Schemata.

DOL erwartet eine Menge von Tasks und zugehörigen Anwendungsfällen als Eingabe. Die Ausgabe beschreibt die Ausführungsplattform, die Abbildung von Tasks auf Prozessoren zusammen mit den Task-*Schedules*. Diese Ausgabe soll bestimmte Beschränkungen (wie Speichergröße und Zeitschranken) einhalten und vorgegebene Ziele (wie Größe, Energie usw.) minimieren. Anwendungen werden als sogenannte Problem-Graphen dargestellt, die im Wesentlichen spezielle Task-Graphen sind. In Abb. 6.32 ist ein einfacher DOL-Problemgraph zu sehen. Dieser Graph modelliert Berechnungen (Knoten 1, 2, 3, 4) und Kommunikation (Knoten 5, 6, 7).

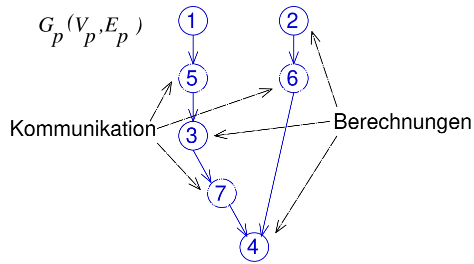
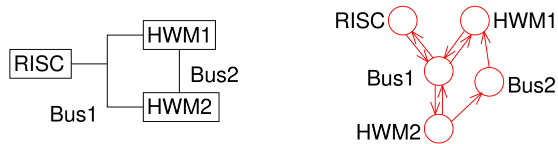


Abb. 6.32 DOL Problem-Graph

Zusätzlich werden mögliche Ausführungsplattformen als sogenannte Architekturgraphen dargestellt. Abb. 6.33 zeigt eine einfache Hardwareplattform und ihren Architekturgraphen. Es gibt einen RISC-Prozessor und zwei Hardwaremodule. Auch hier wird die Kommunikation (mittels Bus1 und Bus2) explizit modelliert.

Abb. 6.33 DOL Architekturgraph



Der Problemgraph und der Architekturgraph werden im **Spezifikationsgraphen** verbunden. In Abb. 6.34 ist ein DOL-Spezifikationsgraph dargestellt. Ein solcher Spezifikationsgraph besteht aus dem Problemgraphen und dem Architekturgraphen. Kanten zwischen den beiden Teilgraphen stellen mögliche Implementierungen dar. Beispielsweise kann die Berechnung 1 nur auf dem RISC-Prozessor, Berechnung 3 auf dem RISC-Prozessor oder auf HWM1 realisiert werden. Kommunikation 5 kann auf dem Bus Bus1 oder – wenn die Berechnungen 1 und 3 beide auf den Prozessor abgebildet werden – lokal im Prozessor stattfinden. Kommunikation 6 kann auf den Bussen Bus1 bzw. Bus2 oder lokal in HWM2 stattfinden.

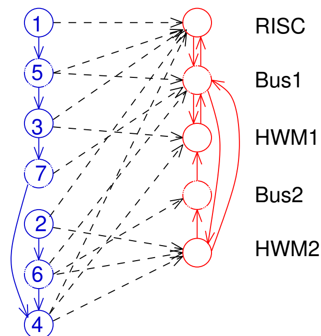


Abb. 6.34 DOL Spezifikationsgraph

Insgesamt werden Implementierungen durch ein Tripel beschrieben:

- **Eine Allokation** A: A ist eine Teilmenge des Architekturgraphen, die für einen bestimmten Entwurf reservierte (ausgewählte) Hardwarekomponenten darstellt.
- **Eine Bindung** b: eine ausgewählte Teilmenge der Kanten zwischen Spezifikation und Architektur kennzeichnet eine Relation zwischen diesen beiden. Die ausgewählten Kannten werden als **Bindung** bezeichnet.
- **Einen Zeitplan** (*Schedule*) $\mathcal{S}$: $\mathcal{S}$ weist jedem Knoten τ_i im Problemgraphen seine Startzeit zu.

Beispiel 6.19: Abb. 6.35 zeigt, wie die Spezifikation aus Abb. 6.34 in eine Implementierung umgesetzt werden kann. HWM2 und der Bus2 werden nicht benutzt und sind nicht in der Menge A enthalten. Eine Teilmenge b der Kanten wurde für die Abbildung ausgewählt. Die Knoten 1, 2, 3, 5 wurden alle auf den RISC-Prozessor abgebildet, wodurch die Kommunikation 5 eine rein lokale Kommunikation wird. Knoten 4 ist auf HWM1 abgebildet und er kommuniziert über den Bus Bus1. Der Ablaufplan $\mathcal{S}$ besagt, dass Berechnung 1 zur Zeit 0 startet, Kommunikation 5 und Berechnung 2 beginnen zur Zeit 1, Berechnung 3 und Kommunikation 6 fangen zur Zeit 21 an, Kommunikation 7 beginnt zur Zeit 29 und Berechnung 4 startet zum Zeitpunkt 30. ∇

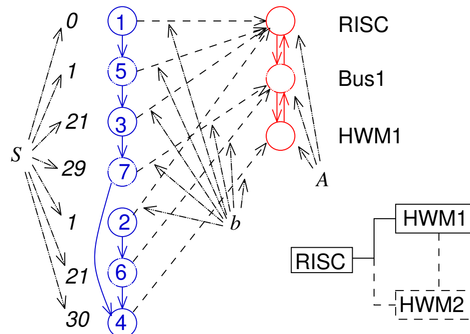


Abb. 6.35 DOL Implementierung

DOL erzeugt Implementierungen mit Hilfe evolutionärer Algorithmen [31, 30, 107]. In solchen Algorithmen werden Lösungen durch Folgen von Werten in den Chromosomen von „Individuen” dargestellt. Mit evolutionären Algorithmen lassen sich neue Lösungsmengen aus existierenden Mengen von Lösungen ableiten. Die Ableitung basiert dabei auf evolutionären Operatoren wie Mutation, Selektion und Rekombination. Die Auswahl neuer Lösungsmengen erfolgt auf der Grundlage von Eignungswerten (engl. *fitness values*). Evolutionäre Algorithmen können komplexe Optimierungsprobleme lösen, bei denen andere Arten von Algorithmen versagen. Es ist nicht leicht, geeignete Kodierungen von Lösungen in Chromosomen zu finden. Einerseits sollte die Dekodierung nicht zu viel Rechenzeit benötigen, andererseits müssen wir die Situation nach den evolutionären Transformationen berücksichtigen. Diese Transformationen könnten nicht realisierbare Lösungen erzeugen, wenn die Kodierungen nicht sorgfältig gewählt werden.

In DOL kodieren die Chromosomen die Auswahl (Allokation) von Hardware und die Bindungen. Zur Berechnung der Fitness einer Lösung müssen Allokationen und Bindungen aus den Individuen dekodiert werden (siehe Abb. 6.36).

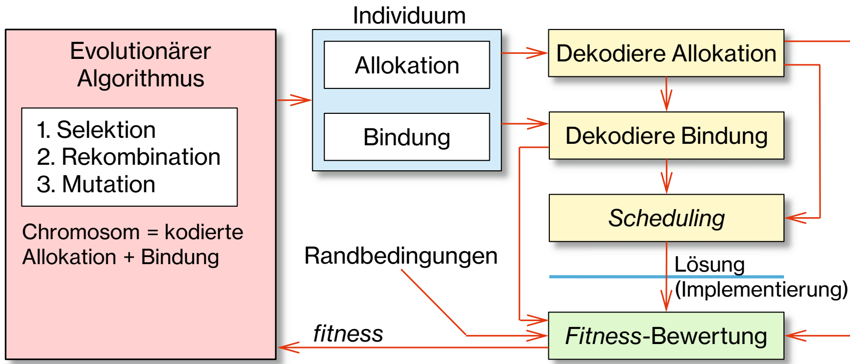


Abb. 6.36 Dekodieren der Lösungen aus den Chromosomen von Individuen

Bei DOL werden Zeitpläne nicht in den Chromosomen kodiert. Vielmehr werden sie aus der Allokation und Bindung abgeleitet. Dadurch wird das Überfrachten der evolutionären Algorithmen mit *Scheduling*-Entscheidungen vermieden. Sobald der Zeitplan berechnet wurde, kann die Eignung der Lösungen evaluiert werden.

Die gesamte Architektur von DOL ist in Abb. 6.37 dargestellt.

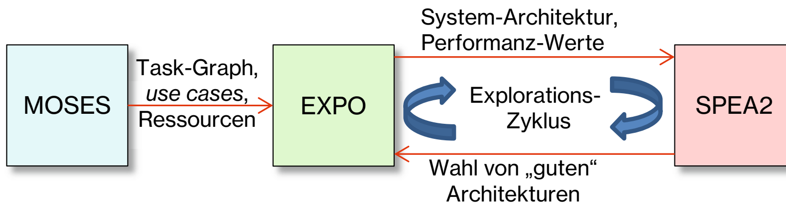


Abb. 6.37 DOL-Werkzeug

Zu Beginn werden der Task-Graph, Anwendungsfälle und die verfügbaren Ressourcen definiert. Dazu dient ein spezieller Editor namens MOSES. Diese Anfangsinformationen werden durch das Evaluations-*Framework* EXPO bewertet. Durch EXPO berechnete Performanzwerte werden dann an SPEA2 weitergeleitet und dort durch evolutionäre Algorithmen verarbeitet. SPEA2 wählt gute Architekturen. Diese werden zur Bewertung an EXPO weitergeleitet. Die Bewertungsergebnisse werden wieder an SPEA2 geschickt, um erneut zu optimieren. Dieses Wechselspiel zwischen EXPO und SPEA2 findet so lange statt, bis gute Lösungen gefunden wurden. Die Auswahl der Lösungen erfolgt nach dem Prinzip der Pareto-Optimalität. Der Entwickler erhält eine Menge Pareto-optimaler Entwürfe und kann dann zwischen den verschiedenen Zielen abwägen.

Beispiel 6.20: In Abb. 6.38 ist die sich ergebende Pareto-Front graphisch dargestellt. Die *Trade-Offs* zwischen der Performanz von zwei Netzwerken und möglichen Einsparungen sind zu erkennen.

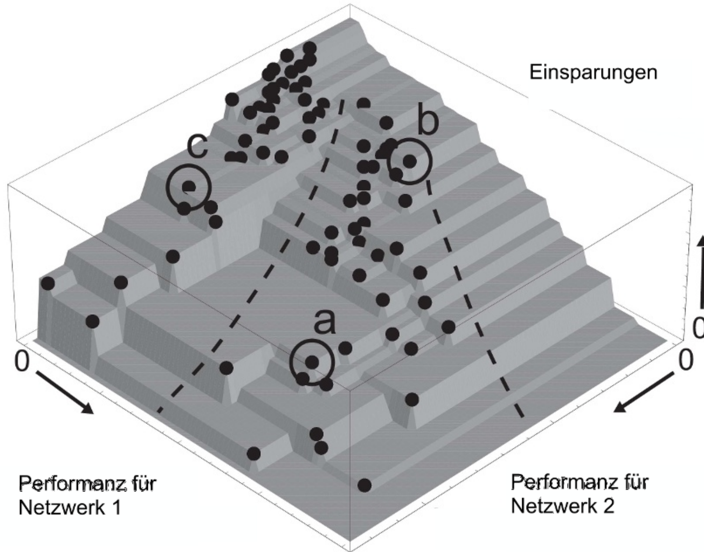


Abb. 6.38 Pareto-Front von Lösungen für ein Entwurfsproblem, ©ETHZ ∇

Holzkamp hat eine Variante von DOL entwickelt, die sich auf die Optimierung der Speicherzuordnung konzentriert [221]. Evolutionäre Algorithmen sind eine Standardtechnik zur Lösung fortgeschrittener *Scheduling*-Probleme (d.h. jenseits der durch HEFT und CPOP gelösten Probleme) geworden.

Die Funktionalität von SystemCodesigner [286] ähnelt der von DOL. Die Systeme unterscheiden sich aber darin, wie Spezifikationen beschrieben werden (hier kann SystemC verwendet werden) und wie die Optimierungen durchgeführt werden. Die Abbildung von Anwendungen wird als ILP-Modell dargestellt. Eine erste Lösung wird mit einem ILP-Optimierer erstellt. Diese Lösung wird dann durch den Einsatz evolutionärer Algorithmen verbessert[10].

Daedalus [423] beinhaltet automatische Parallelisierung. Dazu werden sequenzielle Algorithmen auf Kahn-Prozessnetzwerke abgebildet. Die Entwurfsraumerkundung findet dann unter Verwendung von Kahn-Prozessnetzwerken als Zwischendarstellung statt.

Andere Ansätze beginnen mit einem gegebenen Task-Graphen und bilden auf eine feste Architektur ab. Beispielsweise bildet Ruggiero Anwendungen auf Cell-Prozessoren ab [475]. Das HOPES-System kann auf verschiedene Prozessoren

[10] Eine neuere Version verwendet dafür einen sogenannten *satisfiability solver* (SAT), der Erfüllbarkeitsprobleme der Prädikatenlogik lösen kann.

abbilden [195], dabei verwendet es Berechnungsmodelle, die von den Ptolemy-Werkzeugen unterstützt werden. Einige Werkzeuge berücksichtigen zudem weitere Ziele. So behandelt Xu zum Beispiel die Optimierung der zuverlässigen Lebensdauer des resultierenden Systems [605]. Simunic betrachtet in ihren Arbeiten zusätzlich eine Temperaturanalyse und versucht, die Entstehung zu heißer Stellen auf dem MP-SoC zu vermeiden [492]. Weitere Arbeiten finden sich bei Popovici et al. [457]. Er verwendet verschiedene Ebenen der Modellierung unter Verwendung der Sprachen Simulink und SystemC.

Ansätze zur automatischen Parallelisierung für festgelegte Architekturen finden sich bei Arbeiten an der Universität von Edinburgh [168]. Die MAPS-Werkzeuge [88] verbinden automatische Parallelisierung mit einer eingeschränkten Form der Entwurfsraumerkundung. Cordes [110] hat eine automatische Parallelisierung für Mehrkern-Systeme entwickelt, die ein Kostenmodell auf einer hohen Ebene benutzt. Eine automatische Parallelisierung wurde auch von Neugebauer et al. [417] entwickelt. Sie wurde zur Optimierung eines innovativen Sensors für Bio-Viren, einem cyber-physikalischen System, benutzt.

6.5.5 Dynamisches und Hybrides Scheduling

Beim dynamischen *Scheduling* erfolgt die Prozessorzuordnung zur Laufzeit und nicht zur Entwurfszeit. Dynamisches *Scheduling* hat eine Reihe von Vorteilen [493]:

- **Anpassbarkeit an die verfügbaren Ressourcen**: Dynamisches *Scheduling* kann sich anpassen, wenn sich die Verfügbarkeit von Ressourcen wie Energie, Speicherplatz oder Kommunikationsbandbreite ändert.
- **Anpassbarkeit an sich ändernde Ressourcenanforderungen**: Änderungen bezüglich der Anforderungen der Anwendungen (wie z.B. ein wachsender Speicherbedarf) sind leichter zu integrieren, wenn das *Scheduling* dynamisch ist.
- **Elastizität in Bezug auf Defekte:** Ressourcendefekte können mit einem dynamischen *Scheduling* leichter berücksichtigt werden.
- **Benutzung von nicht echtzeitfähigen Plattformen:** Dynamisches *Scheduling* ist bei nicht echtzeitfähigen Rechnern der Standard. Daher können Techniken dieser Plattformen eingesetzt werden, was den Entwicklungsaufwand reduziert.

Allerdings gibt es auch Nachteile:

- **Fehlende Garantien für das Echtzeitverhalten**: in einem vollständig dynamischem *Schedule* ist es schwer, wenn nicht unmöglich, Garantien für das Einhalten von Echtzeitbedingungen zu geben.
- **Laufzeit-*Overhead***: dynamisches Scheduling erfordert Rechenzeit, um Entscheidungen zu treffen. Daher müssen komplexe *Scheduling*-Techniken vermieden werden.
- **Beschränktes Wissen**: zur Laufzeit ist üblicherweise nur ein sehr begrenztes Wissen über das Task-System und seine Parameter bekannt.

Es gibt zwei Ansätze für ein dynamisches *Scheduling*: ein **vollständig dynamisches *Scheduling*** (engl. *on-the-fly mapping*) und ein **hybrides *Scheduling***, bei dem Ergebnisse der Entwurfsraumexploration ausgenutzt werden.

Einen Überblick über 25 verschiedene Ansätze für ein vollständig dynamisches *Scheduling* haben Singh et al. [493] veröffentlicht. Diese Art des *Schedulings* kommt den Nicht-Echtzeitsystemen am nächsten.

Hybride *Scheduling*-Techniken benutzen Ergebnisse früherer Entwurfsraumexplorationen, um die o.a. Nachteile von dynamischen Techniken auszugleichen. Beispielsweise können wir *Schedules* für wahrscheinliche Laufzeitszenarien vorab berechnen und dann zur Laufzeit aufgrund des aktuellen Szenarios auswählen. Singh et al. unterscheiden Verfahren mit mehreren *Schedules*, die für eine einzige Anwendung vorab berechnet sind, Verfahren mit mehreren *Schedules* für mehrere Anwendungen und Verfahren, welche die Zuverlässigkeit mit einbeziehen[11]. Die Autoren geben eine Übersicht über 21 verschiedene Ansätze für die Abfolge von Analysen zur Entwurfszeit und Entscheidungen zur Laufzeit.

Man könnte noch einen Schritt weiter gehen und *Scheduling* mit der Anwendung integrieren. Beispielsweise hat Kotthaus [308] ein Verfahren zur mathematischen Optimierung entwickelt, bei dem dies der Fall ist. In diesem Ansatz ist die Anzahl der Berechnungen der Zielfunktion nicht fest, sondern hängt von dem Fortschritt der parallelen Berechnungen auf einem Mehrkern-System ab. Eine ähnliche Integration ist auch für andere Anwendungen möglich.

6.6 Aufgaben

Die folgenden Aufgaben sollten entweder zu Hause oder während einer Anwesenheitsphase nach dem *flipped classroom*-Konzept [376] bearbeitet werden:

6.1: Gegeben sei eine Menge von vier Jobs. Ankunftszeiten r_i, *Deadlines* D_i und Rechenzeiten C_i sind wie folgt:

- J_1: r_1=10, D_1=18, C_1=4
- J_2: r_2=0, D_2=28, C_2=12
- J_3: r_3=6, D_3=17, C_3=3
- J_4: r_4=3, D_4=13, C_4=6

Erzeugen sie eine graphische Darstellung der *Schedules* für diese Job-Menge unter Benutzung der *Earliest Deadline First* und der *Least Laxity*-(LL-)*Scheduling*-Algorithmen! Geben Sie beim LL-*Scheduling* für alle Jobs beim Kontextwechsel den Schlupf an! Wird ein Job seine *Deadline* verpassen?

6.2: Gegeben sei eine Menge von sechs Tasks τ_1 bis τ_6. Ihre Ausführungszeiten und ihre *Deadlines* sind die folgenden:

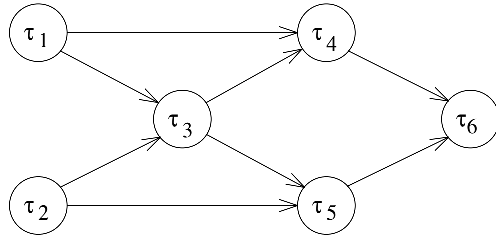
- τ_1: D_1=15, C_1=3

[11] Wir haben hier eine etwas gröbere Unterteilung vorgenommen als Singh et al. selbst.

- τ_2: D_2=13, C_2=5
- τ_3: D_3=14, C_3=4
- τ_4: D_4=16, C_4=2
- τ_5: D_3=20, C_3=4
- τ_6: D_4=22, C_4=3

Abb. 6.39 zeigt Reihenfolgebedingungen für diese Tasks. Die Tasks τ_1 und τ_2 sind sofort ausführungsbereit.

Abb. 6.39 Reihenfolgebedingungen



Erzeugen Sie eine graphische Darstellung eines *Schedules* für diese Task-Menge unter Benutzung des *Latest Deadline First*-Verfahrens.

6.3: Gegeben sei ein Task-System mit zwei Tasks. Task 1 hat eine Periode von 5 und eine Ausführungszeit von 2. Die zweite Task hat eine Periode von 7 und eine Ausführungszeit von 4. Die *Deadlines* seien gleich den Perioden. Nehmen Sie an, dass wir *Rate Monotonic Scheduling* nutzen. Kann eine der beiden Tasks ihre *Deadline* aufgrund einer zu hohen Auslastung verpassen? Berechnen Sie diese Auslastung und vergleichen Sie diese mit einer Schranke, die ein *Schedule* garantiert.

Erzeugen Sie eine graphische Darstellung des resultierenden *Schedules*! Nehmen Sie an, dass Tasks immer bis zum Ende ausgeführt werden, auch wenn sie ihre *Deadline* verpassen.

6.4: Betrachten Sie dieselbe Task-Menge wie in der vorherigen Aufgabe. Benutzen Sie *Earliest Deadline First (EDF)* für das *Scheduling*! Kann eine Task ihre *Deadline* verpassen? Wenn nicht, warum nicht? Erzeugen Sie eine graphische Darstellung des resultierenden *Schedules*! Nehmen Sie an, dass Tasks immer bis zu ihrem Ende ausgeführt werden!

Open Access Dieses Kapitel wird unter der Creative Commons Namensnennung 4.0 International Lizenz (http://creativecommons.org/licenses/by/4.0/deed.de) veröffentlicht, welche die Nutzung, Vervielfältigung, Bearbeitung, Verbreitung und Wiedergabe in jeglichem Medium und Format erlaubt, sofern Sie den/die ursprünglichen Autor(en) und die Quelle ordnungsgemäß nennen, einen Link zur Creative Commons Lizenz beifügen und angeben, ob Änderungen vorgenommen wurden.

Die in diesem Kapitel enthaltenen Bilder und sonstiges Drittmaterial unterliegen ebenfalls der genannten Creative Commons Lizenz, sofern sich aus der Abbildungslegende nichts anderes ergibt. Sofern das betreffende Material nicht unter der genannten Creative Commons Lizenz steht und die betreffende Handlung nicht nach gesetzlichen Vorschriften erlaubt ist, ist für die oben aufgeführten Weiterverwendungen des Materials die Einwilligung des jeweiligen Rechteinhabers einzuholen.

Kapitel 7
Optimierung

Eingebettete Systeme müssen hinsichtlich der in diesem Buch betrachteten Bewertungskriterien effizient sein. Dies gilt insbesondere für mobile Systeme mit begrenzter Ressourcenverfügbarkeit, u.a. für Sensornetzwerke im Internet der Dinge. Es sind schon viele Optimierungen entwickelt worden, um die erforderliche Effizienz zu erreichen. Im Rahmen dieses Buches können wir nur einen Bruchteil davon beschreiben. In diesem Kapitel stellen wir eine Menge an ausgewählten Optimierungen vor. Dieses Kapitel ist wie folgt strukturiert: zunächst präsentieren wir Optimierungstechniken, die typischerweise der Übersetzung mit üblichen Compilern vorangestellt werden können und die früh im Entwurfsablauf benutzt werden können (sogenannte *High-Level*-Optimierungen). Danach werden wir Techniken für die Verwaltung der Nebenläufigkeit von Tasks vorstellen. Abschnitt 7.3 enthält fortgeschrittene Compilertechniken. Der abschließende Abschnitt 7.3.4 beschäftigt sich mit Optimierungen des Stromverbrauchs und des thermischen Verhaltens.

Wie im Entwurfsfluss zu sehen, ergänzen diese Optimierungen die Werkzeuge, die Anwendungen auf die endgültigen Systeme abbilden. Dies wurde in Kapitel 6 beschrieben und ist noch einmal in Abb. 7.1 dargestellt. Die Abbildung von Tasks

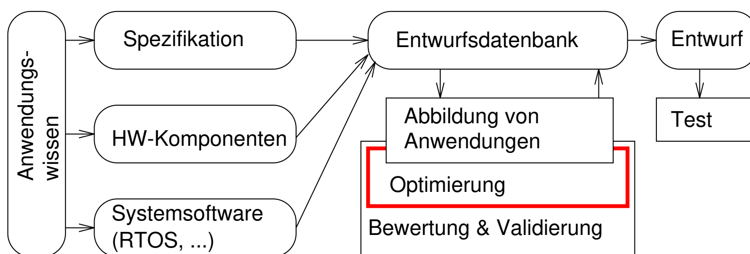


Abb. 7.1 Vereinfachter Entwurfsfluss

und Jobs auf Prozessoren, wie in Kapitel 6 beschrieben, kann Optimierungen enthalten und Optimierungen können die Abbildung von Tasks und Jobs auf Prozessoren

© Der/die Autor(en) 2021

P. Marwedel, *Eingebettete Systeme*, https://doi.org/10.1007/978-3-658-33437-6_7

enthalten. Daher können sich die Themenbereiche zwischen dem Kapitel 6 und dem gegenwärtigen Kapitel überlappen. Der Schwerpunkt des Kapitels 6 liegt bei fundamentalen Techniken der Abbildung auf Ausführungsplattformen. Dagegen liegt der Schwerpunkt des gegenwärtigen Kapitels eher bei (ggf. optionalen) Verbesserungen gegenüber den fundamentalen Techniken.

7.1 *High-Level*-Optimierungen

High-Level-Optimierungen können auf den Quellcode eingebetteter Software angewendet werden. Die Erkennung regulärer Strukturen wie beispielsweise von Zugriffsmustern für Felder (engl. *arrays*) kann auf der Quellcodeebene einfacher sein als auf der Ebene der Maschinensprache. Die Quellcodeebene ist auch deshalb nützlich, weil die Wirkung vieler Optimierungen durch eine Transformation auf dem Quellcode ausgedrückt und so leicht nachvollziehbar gemacht werden kann. In manchen Fällen kann man den Effekt einer Transformation durch Hinweise an den Compiler ausdrücken, was ebenfalls zur Nachvollziehbarkeit beiträgt.

7.1.1 Einfache Schleifentransformationen

Auf Spezifikationen lassen sich eine Reihe von Schleifentransformationen anwenden. Nachfolgend beschreiben wir einige davon:

- **Schleifenpermutation:** Wir betrachten ein zweidimensionales Array. Der C-Standard [290] legt fest, dass zweidimensionale Felder im Speicher wie in Abb. 7.2 beschrieben abgelegt werden. Aufeinanderfolgende Werte des zweiten Indexes werden auf einen zusammenhängenden Speicherbereich abgebildet. Diese Anordnung wird auch ***row-major order*** [406] genannt.

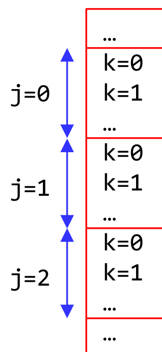


Abb. 7.2 Speicher-Layout für ein 2-dimensionales Array `p[j][k]` in C

Für ein derartiges *Layout* ist es üblicherweise vorteilhaft, Schleifen so zu organisieren, dass der letzte Index der innersten Schleife entspricht. So werden die Zugriffe auf den Speicher möglichst lokal gehalten. Dagegen ist die Anordnung von Feldern in FORTRAN unterschiedlich: hier werden benachbarte Werte des ersten Indexes in einen zusammenhängenden Speicherblock abgebildet (im Englischen ***column-major order*** genannt). Veröffentlichungen, die Optimierungen für FORTRAN beschreiben, können daher leicht verwirren.

Beispiel 7.1: Entsprechend dem jeweiligen Speicherlayout ist die nachfolgende Schleifenpermutation geeignet, die Lokalität der Zugriffe auf den Speicher zu erhöhen:

```
for (k=0; k<m; k++)              for (j=0; j<n; j++)
 for (j=0; j<n; j++)      ⇔       for (k=0; k<m; k++)
  p[j][k]= ...                      p[j][k]= ...
```

Diese Art von Permutationen kann eine positive Auswirkung auf die Wiederverwendung von im Cache befindlichen Feldelementen haben, da die folgende Schleifeniteration auf eine benachbarte Speicherstelle zugreifen wird. ∇

Caches sind üblicherweise so organisiert, dass auf benachbarte Speicherstellen wesentlich schneller zugegriffen werden kann als auf weiter von der vorherigen Stelle entfernte Stellen. Auf diese Weise nutzen Caches eine **räumliche Lokalität** der Speicherzugriffe.

Definition 7.1: Wir betrachten Speicherzugriffe auf Adressen `a` und `b`. Angenommen, es liegt ein Zugriff auf `a` vor. **Räumliche Lokalität** besteht, wenn unter dieser Voraussetzung die Wahrscheinlichkeit eines Zugriffs auf `b` mit einem kleinen Abstand der Adressen `a` und `b` wächst.

- **Schleifen abrollen:** Das Abrollen von Schleifen ist eine Standardtransformation, die mehrere Instanzen des Schleifenkörpers erzeugt.

  Beispiel 7.2: Dieses Beispiel zeigt eine einmal abgerollte Schleife:

```
for (j=0; j<n; j++)            for (j=0; j<n; j+=2)
 p[j]= ... ;            ⇔       {p[j]= ... ;
                                  p[j+1]= ...}
```

∇

Die Anzahl der Kopien der Schleife wird **Abrollfaktor** genannt. Dabei sind Abrollfaktoren größer als zwei möglich. Das Abrollen verringert die Kosten der Schleife (es sind weniger Sprünge pro Ausführung des ursprünglichen Schleifenkörpers erforderlich) und verbessert dadurch normalerweise die Performanz. Im Extremfall können Schleifen vollständig abgerollt werden, was den Kontrollaufwand und Sprünge vollständig beseitigt. Abrollen ermöglicht üblicherweise eine Reihe von Folgetransformationen und kann daher auch in den Fällen vorteilhaft

sein, in denen das Abrollen an sich keine direkten Vorteile bietet. Allerdings erhöht sich die Codegröße durch das Abrollen. Abrollen ist normalerweise auf Schleifen mit einer festen Anzahl an Iterationen beschränkt.

- **Schleifenfusion, Schleifenaufspaltung:** Es kann Fälle geben, in denen zwei getrennte Schleifen zusammengeführt werden können sowie Fälle, in denen eine einzelne Schleife in zwei separate aufgeteilt wird.

Beispiel 7.3: Dieses Beispiel zeigt zwei Versionen eines Softwarecodes:

```
for (j=0; j<n; j++)                  for (j=0; j<n; j++)
 p[j]= ... ;                          {p[j]= ... ;
for (j=0; j<n; j++)        ⇔            p[j]=p[j]+ ...}
 p[j]=p[j]+ ...
```

Die links dargestellte Version kann Vorteile bringen, wenn der Zielprozessor einen Schleifenbefehl zur Realisierung von Schleifen ohne *Overhead* (engl. *zero-overhead loop instruction*) besitzt, der aber nur für kleine Schleifen genutzt werden kann. Die rechte Version kann zu einem verbesserten Cacheverhalten führen (durch die verbesserte Lokalität der Referenzen auf das Array `p`) und vergrößert zudem das Potential für parallele Berechnungen innerhalb des Schleifenkörpers. Wie bei vielen anderen Transformationen auch, ist es schwierig zu erkennen, welche der Transformationen den besten Code ergibt. ∇

7.1.2 Kachel-/Blockweise Verarbeitung von Schleifen

Da kleine Speicher schneller als große Speicher sind (siehe Seite 186), kann die Verwendung von Speicherhierarchien vorteilhaft sein. Mögliche „kleine" Speicher sind Caches und *Scratchpad*-Speicher (SPMs). Informationen in diesen Speichern sollten einen hohen Wiederverwendungsgrad aufweisen, da die Speicherhierarchie sonst nicht effizient ausgenutzt werden kann.

Beispiel 7.4: Wiederverwendungseffekte lassen sich an einer Analyse des folgenden Beispiels zeigen. Wir betrachten eine Matrixmultiplikation für Felder der Größe `N` × `N` [321]:

```
for (i=0; i<N; i++)
  for(j=0; j<N; j++)    {
      r=0;
      for (k=0; j<N; k++)
         r+=X[i][k]*Y[k][j];
      Z[i][j]=r;
  }
```

Wir betrachten die Zugriffsmuster für diesen Code. Die skalare Variable `r` repräsentiert `Z[i,j]` in allen Iterationen der innersten Schleife. Wir nehmen an, dass die

explizite Benutzung dieser Variablen dem Compiler hilft, hier ein Register zuzuweisen. Wir nehmen an, dass die Elemente des Feldes in *row-major order* angeordnet sind (wie es in C Standard ist). Damit werden Feldelemente mit benachbarten (rechten) Reihenindizes in benachbarten Speicherstellen abgelegt. Also werden die benachbarten Stellen von `X` während der Iterationen der innersten Schleife geladen. Diese Eigenschaft ist von Vorteil, wenn das Speichersystem *Prefetching* verwendet (jedes Mal, wenn ein Wort in den Cache geladen wird, wird auch das Laden des darauf folgenden Wortes gestartet). Abb. 7.3 zeigt Zugriffsmuster für diesen Code. Die Zugriffe auf `Y` besitzen jedoch keine räumliche Lokalität. Wenn der Cache nicht

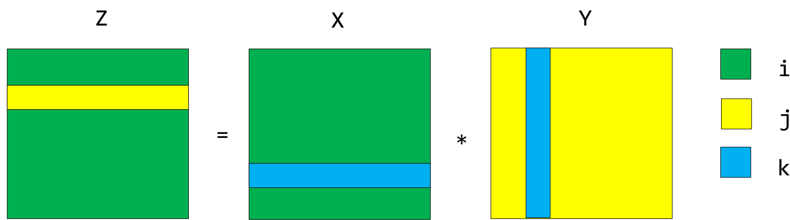


Abb. 7.3 Zugriffsmuster für nicht blockweise Matrixmultiplikation

so groß ist, dass er eine ganze Arrayspalte halten kann, wird jeder Zugriff auf `Y` ein Cachefehler sein. Daher wird es N^3 Zugriffe auf Elemente von `Y` im Hauptspeicher geben.

Arbeiten im Bereich des wissenschaftlichen Rechnens führten zum Entwurf von **geblockten** oder **gekachelten** Algorithmen [321, 606], welche die **Lokalität von Referenzen** verbessern. Eine gekachelte Version des obigen Algorithmus mit einer Blockgröße von `B` sieht wie folgt aus[1]:

```
for (ii=0; kk<N; ii+=B)
 for (jj=0; jj<N; jj+=B)
   for (kk=0; kk<N; kk+=B)
     for (i=ii; i<min(ii+B-1,N); ii++)
       for (j=jj; j<min(jj+B-1,N); jj++) {
           r=0;
           for (k=kk; k<min(kk+B-1,N); k++)
              r+= X[i][k]*Y[k][j];
           Z[i][j]=r;
       }
```

Die innerste Schleife ist nun so begrenzt, dass ein Block der Größe B^2 des Arrays `Y` benutzt wird. Angenommen, ein Block der Größe B^2 passt in den Cache. Dann wird die erste Ausführung der innersten Schleife diesen Block in den Cache laden. Während der Ausführung der zweiten Iteration der Schleife wird dieser Block erneut benutzt. Insgesamt wird es `B-1` Wiederverwendungen der Elemente von `Y` geben. Daher wird die Anzahl der Speicherzugriffe auf $N^3/(B-1)$ reduziert werden. ∇

[1] Dieser Code basiert auf der Quelle http://www.netlib.org/utk/papers/autoblock/node2.html.

Die Optimierung des Wiederverwendungsfaktors war das Thema umfassender Arbeiten. Die ursprünglichen Ansätze konzentrierten sich dabei auf die durch Kachelung erzielbaren Leistungsverbesserungen. Für Matrixmultiplikation wurde eine Leistungsverbesserung um einen Faktor von 3 bis 4,3 von Lam [321] berichtet. Weitere Verbesserungen ergeben sich bei größerem Abstand zwischen Prozessor- und Speichergeschwindigkeiten. Kachelung kann zudem eingesetzt werden, um den Energieverbrauch von Speichersystemen zu senken [103].

7.1.3 Aufteilen von Schleifen

Im Folgenden betrachten wir die Schleifenaufteilung (engl. *loop splitting*) als eine weitere Optimierung, die vor dem Compilieren eines Programms angewendet werden kann. Diese Optimierung könnte möglicherweise auch Bestandteil eines Compilers sein.

Viele Bildverarbeitungsalgorithmen verwenden Filter. Diese Filterung besteht vielfach aus der Betrachtung von Informationen über ein bestimmtes Pixel und einige von dessen Nachparpixeln. Die zugehörigen Berechnungen sind normalerweise recht regulär. Wenn das zu betrachtende Pixel sich aber nahe des Randes eines Bildes befindet, existieren nicht alle benötigten Nachbarpixel und die Berechnungen müssen entsprechend angepasst werden. In einer einfachen Beschreibung des Filteralgorithmus können diese Modifikationen zur Folge haben, dass Tests in der innersten Schleife des Algorithmus durchgeführt werden müssen. Eine effizientere Version des Algorithmus kann erzeugt werden, indem die Schleifen so aufgespalten werden, dass ein Schleifenkörper die regulären Fälle behandelt und ein zweiter Schleifenkörper für die Ausnahmen zuständig ist. Abb. 7.4 ist eine graphische Darstellung dieser Transformation.

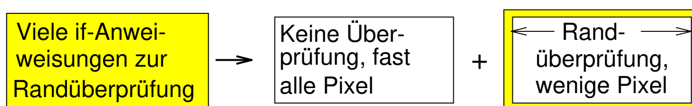


Abb. 7.4 Aufteilung des Bildverarbeitungsbeispiels in reguläre und spezielle Fälle

Die manuelle Aufteilung von Schleifen ist sehr kompliziert und fehleranfällig. Falk et al. haben einen Algorithmus veröffentlicht [159], um diesen Vorgang auch für größere Dimensionen zu automatisieren. Er basiert auf einer ausführlichen Analyse von Zugriffen auf Arrayelementen innerhalb von Schleifen. Optimierte Lösungen werden mit einer Polyederanalyse-Bibliothek [586] und genetischen Algorithmen [341] erzeugt.

Beispiel 7.5: Der folgende Code zeigt eine Schleifenverschachtelung aus dem MPEG-4-Standard, die eine Bewegungsanalyse (engl. *motion estimation*) durchführt:

```
for (z=0; z<20; z++)
 for (x=0; x<36; x++) {x1=4*x;
  for (y=0; y<49; y++) {y1=4*y;
   for (k=0; k<9; k++)  {x2=x1+k-4;
    for (l=0; l<9; l++)  {y2=y1+l-4;
     for (i=0; i<4; i++)  {x3=x1+i; x4=x2+i;
      for (j=0; j<4; j++)  {y3=y1+j; y4=y2+j;
       if (x3<0 || 35<x3 || y3<0 || 48<y3)
        then_block_1; else else_block_1;
       if (x4<0 || 35<x4 || y4<0 || 48<y4)
        then_block_2; else else_block_2;
      }
     }
    }
   }
  }
 }
```

Unter Verwendung von Falk's Algorithmus wird diese Verschachtelung in die folgende transformiert:

```
for (z=0; z<20; z++)
  for (x=0; x<36; x++) {x1=4*x;
    for (y=0; y<49; y++)
      if (x>=10 || y>=14)
        for (; y<49; y++)
          for (k=0; k<9; k++)
            for (l=0; l<9; l++ )
             for (i=0; i<4; i++)
               for (j=0; j<4; j++) {
               then_block_1; then_block_2}
      else {y1=4*y;
       for (k=0; k<9; k++) {x2=x1+k-4;
        for (l=0; l<9; l++) {y2=y1+l-4;
         for (i=0; i<4; i++) {x3=x1+i; x4=x2+i;
          for (j=0; j<4; j++) {y3=y1+j; y4=y2+j;
           if ( 0 || 35 <x3 || 0 || 48 < y3)    /* x3<0, y3<0 never true */
            then_block_1; else else_block_1;
           if (x4 < 0|| 35 < x4 || y4 < 0 || 48 < y4)
            then_block_2; else else_block_2;
          }
         }
        }
       }
      }
  }
```

Anstatt die aufwendigen Tests in der innersten Schleife durchzuführen, besitzt der Code nun eine `if`-Anweisung nach der dritten `for`-Schleifenanweisung. Alle

regulären Fälle werden im **then**-Teil dieser Anweisung behandelt. Der **else**-Teil behandelt die relativ kleine Anzahl von übrigen Fällen. ∇

Eine Schleifenaufspaltung kann Laufzeiten für verschiedene Applikationen und Architekturen reduzieren. Relative Laufzeiten werden in Abb. 7.5 gezeigt. Für die Bewegungsanalyse (engl. *motion estimation*) verringert sich die Zyklenanzahl um bis zu ca. 75% (auf 25% des ursprünglichen Wertes). So sind signifikante Einsparungen möglich, die vielfach größer sind als für die vorher vorgestellten einfachen Transformationen. Die Schleifenaufspaltung kann beispielsweise vor der Übersetzung von Quellcode durch einen Compiler durchgeführt werden.

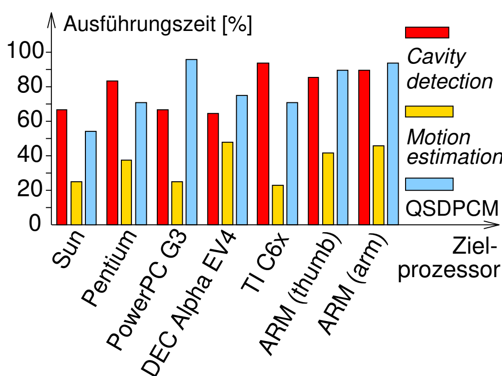


Abb. 7.5 Ergebnisse für die Schleifenaufspaltung

7.1.4 Falten von Feldern

Einige eingebettete Anwendungen, insbesondere im Multimediabereich, verwenden große Felder (engl. *arrays*). Da der Speicherplatz in eingebetteten Systemen begrenzt ist, sollten Möglichkeiten genutzt werden, die Speicheranforderungen von Feldern zu vermindern. Abb. 7.6 stellt die Adressen, die von fünf Feldern verwendet werden, als Funktion über der Zeit dar. Zu jedem einzelnen Zeitpunkt wird nur eine Teilmenge der Feldelemente benötigt. Die maximale Anzahl benötigter Elemente wird das **Adressreferenzfenster** genannt [123]. In Abb. 7.6 wird dieses Maximum durch einen Pfeil mit zwei Spitzen dargestellt.

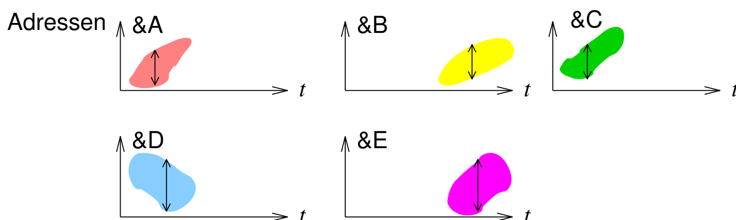


Abb. 7.6 Referenzmuster für Felder

Eine klassische Speicherzuteilung für Felder ist auf der linken Seite von Abb. 7.7 dargestellt. Jedem Feld wird das Maximum an Platz zugewiesen, den es während der gesamten Ausführungszeit benötigt (wenn globale Felder betrachtet werden).

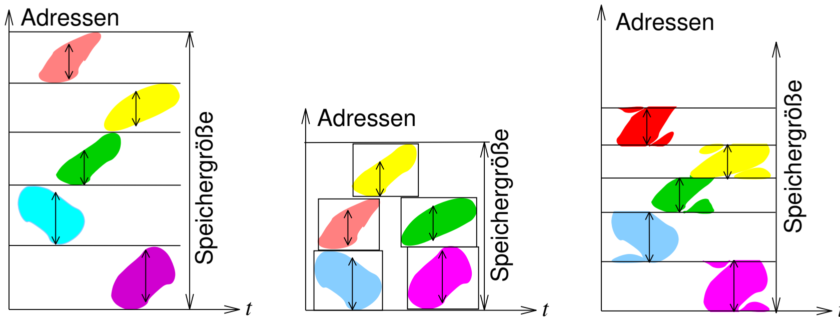


Abb. 7.7 Ungefaltete (**links**), *inter-array* gefaltete (**Mitte**) und *intra-array* gefaltete (**rechts**) Felder

Eine mögliche Verbesserung, ***inter-array folding*** wird in der Mitte von Abb. 7.7 dargestellt. Felder, die nicht in überlappenden Zeitintervallen gleichzeitig benötigt werden, können sich denselben Speicherplatz teilen. Eine weitere Verbesserung, *intra-array folding* [122], ist auf der rechten Seite von Abb. 7.7 zu sehen. Diese nutzt die Eigenschaft aus, dass **innerhalb** eines Feldes nur eine begrenzte Anzahl von Komponenten benötigt wird. Hier kann Speicher auf Kosten von aufwändigeren Adressberechnungen eingespart werden. Diese beiden Arten von Faltungen können zudem auch kombiniert werden.

Andere Arten von Transformationen auf hoher Ebene wurden von Chung, Benini und De Micheli untersucht [103, 524]. Viele weitere verwandte Optimierungen wurden im Bereich des Compilerentwurfs vorgeschlagen.

7.1.5 Wandlung von Gleitkomma- in Festkommadarstellung

Eine oft angewendete Methode der Optimierung ist die Umwandlung von Gleitkomma- in Festkommaberechnungen. Diese Umwandlung wird dadurch motiviert, dass viele Standards für die Signalverarbeitung (wie z.B. MPEG-2 oder MPEG-4) in Form von C-Programmen spezifiziert sind, die Gleitkomma-Datentypen verwenden. Es ist dem Entwickler überlassen, eine effiziente Implementierung für diese Standards zu finden.

In vielen Signalverarbeitungsanwendungen ist es möglich, Gleitkommazahlen durch Festkommazahlen zu ersetzen (siehe Seite 169). Dies kann beträchtliche Vorteile mit sich bringen. So wurde beispielsweise für einen MPEG-2 Videokompressionsalgorithmus eine Verringerung der benötigten Prozessorzyklen um 75% und

des Energieverbrauchs um 76% berichtet [226]. Allerdings bringt die Umwandlung üblicherweise einen gewissen Präzisionsverlust mit sich. Genaugenommen gibt es einen Zielkonflikt zwischen verschiedenen Metriken zur Bewertung eines Entwurfs (siehe Abschnitt 5.3 auf Seite 278), hier z.B. zwischen den Implementierungskosten und der Qualität des Algorithmus (ausgedrückt z.B. in Form des Signal-Rausch-Abstandes (SNR), siehe Seite 156). Bei kleinen Wortlängen kann die Qualität signifikant beeinträchtigt werden. Als Folge können Gleitkomma-Datentypen durch Festkomma-Datentypen ersetzt werden, dabei muss allerdings eine Analyse des Qualitätsverlustes erfolgen. Ursprünglich wurde diese Ersetzung manuell durchgeführt. Dies ist jedoch ein aufwändiger und fehleranfälliger Vorgang. Daher hat man versucht, diese Ersetzung mit Hilfe geeigneter Werkzeuge durchzuführen. Eines dieser Werkzeuge ist FRIDGE (*Fixed-point pRogrammIng DesiGn Environment*) [588, 284]. Die Funktionalität von FRIDGE ist kommerziell als Teil der System Studio-Umgebung der Firma Synopsys verfügbar [518]. SystemC kann dazu verwendet werden, Festkomma-Datentypen zu simulieren und so den Qualitätsverlust zu prüfen.

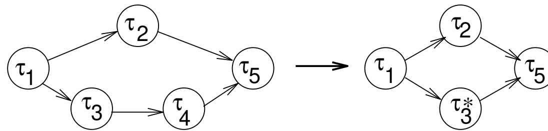
Analysen der Zielkonflikte zwischen hinzugefügtem Rauschen und der benötigten Wortlänge wurden von Shi und Brodersen [486] und auch von Menard et al. [391] vorgeschlagen. Das Thema ist insgesamt weiterhin Gegenstand der Forschung (siehe z.B. Lee et al. [336]). Auch im Bereich des Maschinellen Lernens werden entsprechende Verfahren untersucht [454].

7.2 Nebenläufigkeit von Tasks

Wie bereits auf Seite 41 beschrieben ist die **Granularität** eines Task-Graphen eine der wichtigsten Eigenschaften von Tasks. Auch für hierarchische Task-Graphen kann es nützlich sein, die Granularität der Knoten zu verändern. Die Partitionierung von Spezifikationen in einzelne Tasks zielt dabei nicht notwendigerweise auf die Erreichung maximaler Effizienz der Implementierung ab. In der Spezifikationsphase sind vielmehr eine klare Trennung von Belangen und ein sauberes Softwaremodell wichtiger als Details der Implementierung. Eine klare Trennung der Belange beinhaltet beispielsweise eine klare Trennung der Implementierung abstrakter Datentypen von ihrer Verwendung. Im Laufe des Entwurfs werden Tasks typischerweise zu Objekten des Betriebssystems, also zu Prozessen (siehe Definition 4.1) bzw. *Threads*. Unsere Spezifikation könnte die fließbandartige Verwendung mehrerer Tasks vorsehen, wobei die Verschmelzung einiger dieser Tasks den Aufwand zur Kontextumschaltung zwischen Prozessen verringern könnte. Daher wird nicht notwendigerweise eine Eins-zu-eins-Beziehung zwischen den in der Spezifikation beschriebenen Tasks und den nachher tatsächlich implementierten Prozessen bestehen. Das bedeutet, dass eine Neustrukturierung von Tasks angebracht sein kann. Verschmelzen und Aufteilen einzelner Tasks vor der Abbildung auf Prozesse erlaubt es tatsächlich, eine solche Neustrukturierung zu realisieren.

Task-Graphen können verschmolzen werden, wenn eine Task τ_i unmittelbare Vorgängerin einer Task τ_j ist und τ_j keine andere direkte Vorgängerin besitzt (siehe Abb. 7.8 mit $\tau_i = \tau_3$ und $\tau_j = \tau_4$). Diese Umwandlung kann einen geringeren

Abb. 7.8 Verschmelzen von Tasks

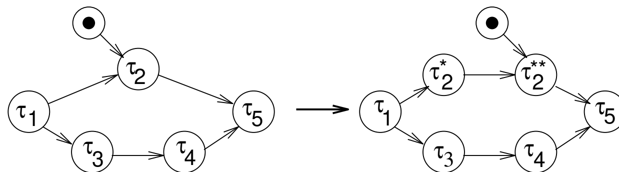


Aufwand für die Kontextumschaltung zwischen Prozessen zur Folge haben, wenn der Knoten in Software implementiert ist. Allgemein kann hierdurch ein größeres Optimierungspotential entstehen.

Weiterhin kann das Aufteilen von Tasks vorteilhaft sein. Beispielsweise können Tasks Ressourcen belegen (wie z.B. große Mengen an Speicher), während sie auf Eingaben warten. Um die Verwendung dieser Ressourcen zu maximieren, kann es sinnvoll sein, die Verwendung der Ressourcen nur in den Zeitintervallen zu erlauben, in denen sie tatsächlich benötigt werden.

Beispiel 7.6: Abb. 7.9 geht davon aus, dass die Task τ_2 eine Eingabe während ihrer Ausführung benötigt. In der ursprünglichen Variante kann die Ausführung von Task

Abb. 7.9 Aufteilen von Tasks



τ_2 nur dann beginnen, wenn diese Eingabe verfügbar ist. Wir können diesen Knoten nun in τ_2^* und τ_2^{**} aufteilen, so dass die Eingabe nur für die Ausführung von τ_2^{**} erforderlich ist. Damit kann nun τ_2^* früher starten, wodurch das System einen größeren Freiraum für *Scheduling*-Entscheidungen erhält. Diese größere Freiheit beim *Scheduling* kann die Auslastung von Ressourcen verbessern und es sogar ermöglichen, eine bestimmte *Deadline* einzuhalten. Zudem kann sie einen Einfluss auf den benötigten Speicherplatz haben, da τ_2^* einen Teil seines Speichers freigeben könnte, bevor die Task sich beendet. Dieser Speicher könnte dann wiederum von anderen Tasks verwendet werden, während τ_2^{**} auf Eingaben wartet. ∇

Man könnte nun argumentieren, dass Tasks solche Ressourcen – wie große Mengen Speicher – sowieso freigeben sollten, bevor sie auf Eingaben warten. Wenn man solche Details der Implementierung aber bereits in einer frühen Spezifikationsphase berücksichtigen würde, könnte die Lesbarkeit der ursprünglichen Spezifikation darunter leiden.

Mit Hilfe einer Petrinetz-basierten Technik, die von Cortadella et al. beschrieben wurde [111], können sehr aufwendige Transformationen der Spezifikationen vorgenommen werden. Diese Vorgehensweise beginnt mit einer Spezifikation einer Menge an Tasks, die in der Sprache *FlowC* beschrieben werden. FlowC erweitert C um Prozessdefinitionen und Kommunikation zwischen Tasks, die durch `READ`- und `WRITE`-Funktionsaufrufe spezifiziert wird.

Beispiel 7.7: Abb. 7.10 zeigt eine Eingabespezifikation in FlowC mit den Eingabeports `IN` und `COEF` sowie dem Ausgabeport `OUT`. Punkt-zu-Punkt-Kommunikation zwischen den einzelnen Prozessen wird durch einen unidirektionalen gepufferten Kanal `DATA` realisiert. Die Task `GetData` liest Umgebungsdaten ein und sendet diese

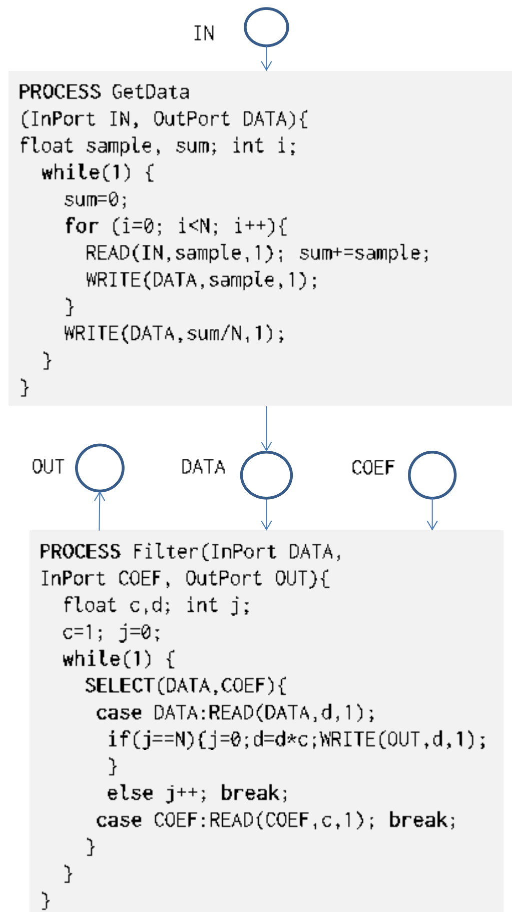


Abb. 7.10 Systemspezifikation

an den Kanal `DATA`. Wenn `N` Datenwerte gesendet wurden, wird auch ihr Durch-

schnittswert auf demselben Kanal übertragen. Die Task `Filter` liest `N` Werte aus dem Kanal (und ignoriert diese) und liest dann den Durchschnittswert, multipliziert diesen Wert mit `c` (`c` kann vom Port `COEF` gelesen werden) und schreibt das Ergebnis auf den Port `OUT`. Der dritte Parameter der `READ`- und `WRITE`-Funktionsaufrufe ist die Anzahl der zu lesenden oder zu schreibenden Werte. `READ`-Aufrufe sind blockierend, `WRITE`-Aufrufe blockieren, wenn die Anzahl an Elementen im Kanal eine bestimmte Obergrenze überschreitet. Die **`SELECT`**-Anweisung hat dieselbe Semantik wie die gleichnamige Anweisung in Ada (siehe Seite 124): die Ausführung der Task wird angehalten, bis eine Eingabe von einem der beiden Ports vorliegt. Dieses Beispiel erfüllt alle Kriterien für die Aufteilung von Tasks, die im Rahmen von Abb. 7.9 aufgeführt wurden. Beide Tasks warten auf Eingaben, während sie bereits Ressourcen belegen. Die Effizienz könnte durch Umstrukturieren der beiden Tasks gesteigert werden. Dabei reicht die einfache Aufteilung von Abb. 7.9 hier aber nicht aus. Bei der von Cortadella et al. vorgeschlagenen Methode werden FlowC-Programme zunächst in (erweiterte) Petrinetze übersetzt. Die Petrinetze für die einzelnen Tasks werden dann zu einem einzigen Petrinetz zusammengefügt. Neue Tasks werden schließlich mit Hilfe von Ergebnissen aus der Petrinetz-Theorie erzeugt. Abb. 7.11 zeigt eine mögliche neue Task-Struktur. In dieser gibt es eine Task, die alle Initialisie-

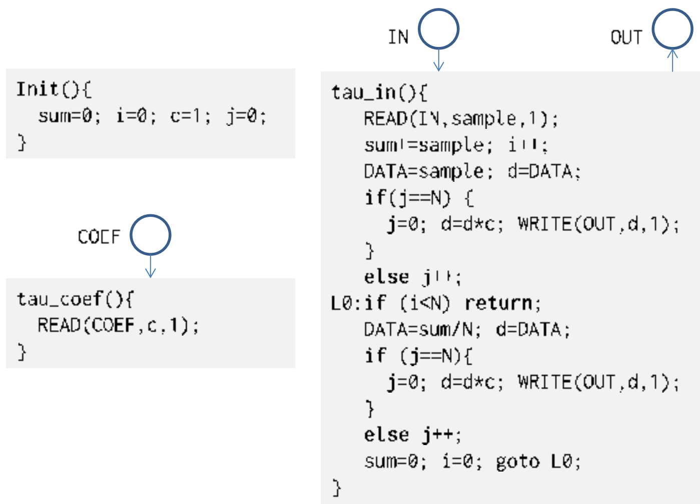


Abb. 7.11 Erzeugte Software-Tasks

rungen ausführt. Zudem gibt es je eine Task für jeden Eingabeport. Eine effiziente Implementierung würde eine Unterbrechung für jede neue Eingabe an einem der Ports erzeugen, wobei pro Port ein eindeutiger Interrupt verfügbar sein sollte. Die Tasks könnten dann direkt durch die jeweiligen Interrupts gestartet werden, ohne dass ein Aufruf des Betriebssystems erforderlich wäre. Kommunikation kann über eine einzelne, gemeinsam benutzte globale Variable implementiert werden (wenn

ein gemeinsamer Adressraum für alle Tasks vorliegt). Der verbleibende Aufwand für das Betriebssystem wäre dadurch sehr klein, wenn ein Betriebssystem überhaupt notwendig wäre.

Der in Abb. 7.11 gezeigte Code für die Task `tau_in` wurde durch die Petrinetz-basierte, taskübergreifende Optimierung der Task-Strukturen erzeugt. Dieser sollte durch Optimierungen innerhalb der Tasks weiter verbessert werden. Eine optimierte Version von `tau_in` sieht wie folgt aus:

```
tau_in () {
    READ(IN,sample,1);
    sum+=sample;  i++;
    DATA=sample;  d=DATA;        /* Zusammenfassen von DATA & d möglich */
L0: if (i<N) return;
    DATA=sum/N; d=DATA;
    d=d*c; WRITE(OUT,d,1);
    sum=0; i=0;
    return;
}
```

Die Gründe für diese Vereinfachung sind: im originalen Code ergibt der Test, der durch die erste **`if`**-Anweisung durchgeführt wird, immer „falsch" (`j` ist in diesem Fall gleich `i-1` und `i` und `j` werden auf `0` zurückgesetzt, sobald `i` den Wert `N` annimmt). In der dritten **`if`**-Anweisung ergibt der Test stets „wahr", da diese Anweisung nur erreicht wird, wenn `i` gleich `N` ist und `i` gleich `j` ist, sobald das Label `L0` erreicht wird. Zudem kann die Anzahl der Variablen verringert werden.

Die optimierte Version von `tau_in` könnte von einem sehr geschickten Compiler erzeugt werden. Allerdings kann kaum einer der heute verfügbaren Compiler eine solche Optimierung vornehmen. Dennoch zeigt dieses Beispiel die Art von Transformationen, die zur Erzeugung „guter" Task-Strukturen notwendig sind. ∇

Mehr Details zur Task-Erzeugung finden sich in Cortadella et al. [111] und in einer Publikation von Meijer et al. [390].

7.3 Compiler für eingebettete Systeme

7.3.1 Einleitung

Für Prozessoren von PCs und Servern sind Optimierungen und Compiler verfügbar. Die Erzeugung von Compilern für häufig verwendete Prozessoren ist ein hinlänglich bekanntes Gebiet. In vielen Fällen werden Standard-Compiler auch für eingebettete Systeme verwendet, da sie üblicherweise kostengünstig oder gar kostenfrei erhältlich sind.

Es gibt jedoch eine Reihe besonderer Gründe, spezielle Optimierungen und Compiler für eingebettete Systeme zu entwerfen:

- Prozessorarchitekturen eingebetteter Systeme verfügen über besondere Eigenschaften (siehe Seite 158). Diese Eigenschaften sollten ausgenutzt werden, um effizienten Code zu erzeugen. Übersetzungstechniken müssen dabei möglicherweise auch die auf den Seiten 163 bis 165 beschriebenen Kompressionstechniken unterstützen.
- Eine hohe Codeeffizienz ist wichtiger als eine hohe Übersetzungsgeschwindigkeit.
- Compiler könnten dabei helfen, Echtzeitbedingungen einzuhalten und zu beweisen. Zunächst wäre es schön, wenn Compiler explizite Zeitmodelle enthielten. Diese könnten für Optimierungen genutzt werden, die das Zeitverhalten wirklich verbessern. Beispielsweise kann es von Vorteil sein, bestimmte Cachezeilen einzufrieren, um zu verhindern, dass häufig ausgeführter Code mehrmals aus dem Cache verdrängt und wieder hineingeladen wird.
- Compiler können dabei helfen, den Energieverbrauch eingebetteter Systeme zu senken. Compiler, die Energieoptimierungen durchführen, sollten verfügbar sein.
- Eingebettete Systeme verfügen über eine größere Vielfalt von Befehlssätzen. Daher gibt es mehr Prozessoren, für die Compiler verfügbar sein sollten. In einigen Fällen gibt es auch die Anforderung, die Optimierung von Befehlssätzen mit **retargierbaren** Compilern zu unterstützen. Bei solchen Compilern kann der Befehlssatz als Eingabe für ein Compiler-Erzeugungssystem spezifiziert werden. Solche Systeme können verwendet werden, um einen Befehlssatz experimentell zu verändern und dann die daraus folgenden Änderungen des resultierenden Maschinencodes zu analysieren. Dies ist ein Sonderfall der **Erkundung des Entwurfsraumes** und wird beispielsweise von Werkzeugen der Firma Tensilica unterstützt [82].

Einige frühe Ansätze für retargierbare Compiler werden im ersten Buch über dieses Thema beschrieben [378]. Optimierungen finden sich in Büchern von Leupers et al. [338, 339]. In diesem Abschnitt stellen wir Beispiele von Übersetzungstechniken für eingebettete Prozessoren vor.

7.3.2 Energiegewahre Übersetzung

Viele eingebettete Systeme sind batteriebetriebene Mobilgeräte. Wahrend die Anforderungen an die Rechenleistung von mobilen Systemen steigen, geht die Suche nach geeigneten künftigen Batterietechnologien weiter [415]. Daher stellt die Verfügbarkeit von Energie einen Flaschenhals für neue Anwendungen dar.

Energieeinsparungen können auf verschiedenen Ebenen stattfinden, wie bei der Technologie des Herstellungsprozesses, der Gerätetechnologie, dem Schaltkreisentwurf, dem Betriebssystem und den Anwendungsalgorithmen. Geeignete Übersetzungen von Algorithmen in Maschinencode können ebenfalls nützlich sein. Optimierungen auf hoher Ebene, wie auf den Seiten 380 bis 387 vorgestellt, können auch zur Verringerung des Energieverbrauchs beitragen. In diesem Abschnitt betrachten wir Compileroptimierungen, die den Energieverbrauch verringern können (oft auch *low power*-Optimierungen genannt). **Energiemodelle** sind wesentliche Bestandteile

aller Energieoptimierungen. Diese wurden in Kapitel 5 vorgestellt. Unter Verwendung solcher Modelle wurden die folgenden Compileroptimierungen eingesetzt, um den Energieverbrauch zu senken:

- **Energiegewahres *Scheduling*:** Die Reihenfolge von einzelnen Befehlen kann geändert werden, solange sich die Bedeutung des Programms dadurch nicht ändert. Die Reihenfolge kann so geändert werden, dass die Anzahl an Bitwechseln auf dem Befehlsbus minimiert wird. Diese Optimierung kann auf der Ausgabe eines Compilers durchgeführt werden und erfordert daher keine Änderung am Compiler selbst.
- **Energiegewahre Befehlsauswahl:** Üblicherweise kann ein Abschnitt eines Quellcodes durch eine Menge an unterschiedlichen Maschinenbefehlsfolgen implementiert werden. Standard-Compiler verwenden die Anzahl an Instruktionen oder Prozessorzyklen als Kriterium (Kostenfunktion) zur Auswahl einer guten Sequenz. Dieses Kriterium kann durch die von dieser Sequenz benötigten Energie ersetzt werden. Steinke et al. haben herausgefunden, dass energiegewahre Befehlsauswahl den Energieverbrauch um einige Prozent senken kann [509].
- **Ersetzen der Kostenfunktion** ist auch bei anderen Standard-Compileroptimierungen möglich, wie z.B. *register pipelining*, *loop invariant code motion* usw. Auch hier liegen die möglichen Verbesserungen in Bereich einiger weniger Prozent.
- **Ausnutzen der Speicherhierarchie:** Wie bereits auf Seite 184 gezeigt, haben kleinere Speicher kürzere Zugriffszeiten und verbrauchen weniger Energie pro Zugriff. Durch Ausnutzung von Speicherhierarchien kann daher eine nennenswerte Menge an Energie eingespart werden. Die durch Verwendung von Speicherhierarchien erzielbaren Energieeinsparungen zählten zu den lohnendsten Optimierungen, die Steinke [512, 510] analysiert hat. Es ist daher vorteilhaft, beispielsweise kleine *Scratchpad*-Speicher (SPM) zusätzlich zu großen Hintergrundspeichern einzusetzen. Alle Zugriffe auf den jeweiligen Speicherbereich werden dadurch weniger Energie benötigen und schneller sein als ein entsprechender Zugriff auf den größeren Speicher. Dabei sollte der Compiler die Zuordnung von Variablen und Befehlen in den SPM übernehmen. Dieser Ansatz erfordert aber, dass häufig verwendete Variablen und Codefolgen identifiziert und in diesen Adressbereich abgebildet werden.

7.3.3 Speicherarchitektur-gewahre Übersetzung

Übersetzungstechniken für *Scratchpads*

Die Vorteile der Verwendung von SPMs wurden bereits deutlich dargelegt [35]. Daher stellt die Ausnutzung von SPMs einen herausragenden Fall der Ausnutzung von Speicherhierarchien dar. Compiler sind üblicherweise dazu in der Lage, Speicherobjekte auf bestimmte Adressbereiche im Speicher abzubilden. Dafür muss der Quellcode üblicherweise annotiert werden.

Beispiel 7.8: Speichersegmente für ARM-Prozessoren können wie folgt durch Einführung von *Pragmas* in den Quellcode festgelegt werden:

```
# pragma arm section rwdata = "foo", rodata = "bar"
```

Die nach diesem *Pragma* deklarierten Variablen würden dem Lese-/Schreibsegment `"foo"` und Konstanten entsprechend dem Nur-Lesesegment `"bar"` zugeordnet werden. Linkerbefehle können dann diese Segmente auf festgelegte Adressbereiche, unter anderem auch in den SPM, abbilden. ∇

Dieser Ansatz wird von Compilern für ARM-Prozessoren verwendet [19]. Dieser Ansatz ist jedoch nicht sehr komfortabel. Es wäre daher schön, wenn Compiler solche Abbildungen für häufig verwendete Objekte automatisch durchführen könnten. Dafür wurden spezielle Optimierungsalgorithmen entwickelt. Ein Teil der Optimierungen wurde in einem separaten Buch vorgestellt [377]. Die verfügbaren SPM-Optimierungen fallen dabei in eine der beiden Kategorien:

- **Nicht-überlagernde** (engl. *non-overlaying*) oder „statische" Speicherzuordnungsstrategien: Bei diesen Strategien bleiben Speicherobjekte im SPM, solange die zugehörige Anwendung ausgeführt wird.
- **Überlagernde** (engl. *overlaying*) oder „dynamische" Speicherzuordnungsstrategien: Bei diesen Strategien werden Objekte zur Laufzeit in den SPM verschoben und wieder aus diesem verdrängt. Dies stellt eine Art „Compiler-kontrollierten Seitenaustauschs" dar. Allerdings findet der Austausch von Objekten zwischen dem SPM und einem langsameren Speicher statt, anstelle der Auslagerung von Speicher auf Festplatten.

Nicht-überlagernde Zuordnung

Die nicht-überlagernde Zuordnung betrachtet die Zuordnung von Funktionen und globalen Variablen zum SPM. Dafür wird jede Funktion und jede globale Variable als Speicherobjekt modelliert. Seien

- S die Größe des SPM,
- sf_i und sv_i die Größen der Funktion i bzw. der Variablen i,
- g die Energiemenge, die pro SPM-Zugriff **eingespart** wird (also die Differenz zwischen den Energien für einen Zugriff auf den langsamen Hauptspeicher und für einen Zugriff auf den SPM),
- nf_i und nv_i die Anzahl der Zugriffe auf Funktion i bzw. Variable i,
- xf_i und xv_i definiert als

$$xf_i = \begin{cases} 1 \text{ wenn Funktion } i \text{ auf den SPM abgebildet ist} \\ 0 \text{ sonst} \end{cases} \tag{7.1}$$

$$xv_i = \begin{cases} 1 \text{ wenn Variable } i \text{ auf den SPM abgebildet ist} \\ 0 \text{ sonst} \end{cases} \tag{7.2}$$

Dann ist das Ziel die Maximierung des Gewinns

$$G = g\left(\sum_i nf_i \cdot xf_i + \sum_i nv_i \cdot xv_i\right) \tag{7.3}$$

unter Berücksichtigung der Größenbeschränkung

$$\sum_i sf_i \cdot xf_i + \sum_i sv_i \cdot xv_i \leq S \tag{7.4}$$

Dieses Problem ist ein **Rucksack-Problem** (engl. *knapsack problem*). Standard-Knapsack-Algorithmen können verwendet werden, um die dem SPM zuzuordnenden Objekte auszuwählen. Die Gleichungen (7.3) und (7.4) haben aber auch die Form eines Ganzzahligen Linearen Programmierungsproblems (ILP) (siehe Anhang A), damit können ILP-*Solver* verwendet werden. Die entsprechende Optimierung kann vor dem üblichen Übersetzungsvorgang durchgeführt werden (siehe Abb. 7.12).

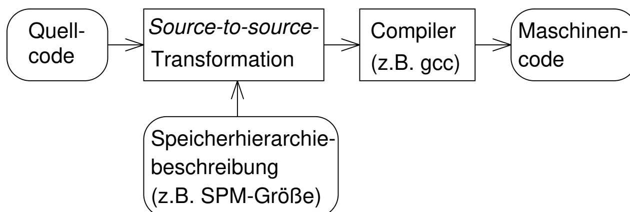


Abb. 7.12 *Source-to-source-*Transformation

Die Optimierung betrifft Adressen von Funktionen und globalen Variablen. Üblicherweise ermöglichen Compiler es, diese Adressen manuell im Quellcode festzulegen. Daher sind keine Änderungen am Compiler selbst notwendig. Der Vorteil solch einer *Source-to-source*-Transformation ist, dass sie mit Compilern für viele verschiedene Zielprozessoren verwendet werden kann. Damit ist es nicht notwendig, eine große Anzahl zielspezifischer Compiler abzuändern.
Dieses Modell lässt sich in verschiedene Richtungen erweitern:

- **Zuordnung von Basisblöcken:** Der gerade beschriebene Ansatz erlaubt es nur, ganze Funktionen oder Variablen im SPM zu allozieren. Folglich kann ein großer Teil des SPM ungenutzt bleiben, wenn Funktionen und Variablen groß sind. Daher versuchen wir, die Granularität der Objekte, die im SPM alloziert werden können, zu reduzieren. Die naheliegende Wahl fällt dabei auf **Basisblöcke** als Speicherobjekte. Zudem betrachten wir auch Mengen benachbarter Basisblöcke, wobei „benachbart" definiert ist als zusammenhängende Teilgraphen im Kontrollflussgraphen. Wir nennen Blöcke, die aus zusammenhängenden Teilgraphen entstehen, auch **Multibasisblöcke** [509]. Abb. 7.13 zeigt drei Multibasisblöcke M12, M23 und M123 für die Basisblöcke BB1, BB2 und BB3.
  Das ILP-Modell kann entsprechend erweitert werden. Seien:

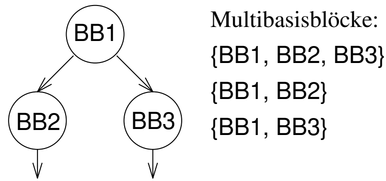


Abb. 7.13 Basisblöcke und Multibasisblöcke

- sb_i and sm_i die Größe des Basisblocks i bzw. des Multibasisblocks i,
- nb_i und nm_i die Anzahl der Zugriffe auf den Basisblock i bzw. den Multibasisblock i,
- xb_i und xm_i definiert als

$$xb_i = \begin{cases} 1 \text{ wenn Basisblock } i \text{ auf SPM abgebildet wird} \\ 0 \text{ sonst} \end{cases} \quad (7.5)$$

$$xm_i = \begin{cases} 1 \text{ wenn Multibasisblock } i \text{ auf SPM abgebildet wird} \\ 0 \text{ sonst} \end{cases} \quad (7.6)$$

Dann ist das Ziel die Maximierung des Gewinns

$$G = g\left(\sum_i nf_i \cdot xf_i + \sum_i nb_i \cdot xb_i + \sum_i nm_i \cdot xm_i + \sum_i nv_i \cdot xv_i\right) \quad (7.7)$$

unter Berücksichtigung der Größenbeschränkung

$$\sum_i sf_i \cdot xf_i + \sum_i sb_i \cdot xb_i + \sum_i sm_i \cdot xm_i + \sum_i sv_i \cdot xv_i \leq S \quad (7.8)$$

$$\forall \text{ Basisblöcke } i : xb_i + xf_{fct(i)} + \sum_{i' \in multibasicblock(i)} xm_{i'} \leq 1 \quad (7.9)$$

Dabei ist $fct(i)$ die Funktion, die den Basisblock i enthält und $multiblock(i)$ die Menge an Multibasisblöcken, die den Basisblock i enthalten. Die zweite Einschränkung (7.9) stellt sicher, dass ein Basisblock nur einmal in den SPM abgebildet wird und nicht potenziell als Teil einer umgebenden Funktion und als Teil eines Multibasisblocks.

Experimente mit diesem Modell wurden von Steinke et al. [512] durchgeführt. Bei bestimmten Benchmarkanwendungen wurden dabei Energieeinsparungen von bis zu 80% erreicht, wobei die Größe des SPM nur einen kleinen Bruchteil der gesamten Größe der Anwendung ausmachte. Ergebnisse für das *bubble sort*-Programm finden sich in Abb. 7.14.

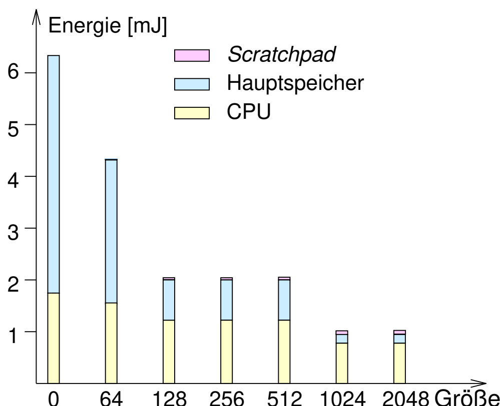


Abb. 7.14 Energiereduktion durch Compiler-basierte *Scratchpad*-Belegung für *bubble sort*

Offenbar haben größere SPMs einen reduzierten Energieverbrauch des Hauptspeichers zur Folge (siehe violette Rechtecke). Die von der CPU benötigte Energie wird auch verringert, da weniger Wartezyklen erforderlich sind. Winzige blaue Rechtecke zeigen den geringen Verbrauch des SPM an. Hierbei wurde die Versorgungsspannung als konstant angenommen, obwohl eine schnellere Ausführung es erlaubt hätte, Taktfrequenzen und Spannungen zu reduzieren, was noch größere Energieeinsparungen zur Folge gehabt hätte.

- **Partitionierte Speicher** [571]: Kleine Speicher sind schneller und benötigen weniger Energie pro Zugriff. Daher ist es sinnvoll, Speicher in mehrere kleinere Speicher zu unterteilen. In diesem Fall unterscheiden wir nicht zwischen den verschiedenen Arten von Speicherobjekten (Funktionen, Basisblöcke, Variablen usw.). Ein Index i stellt ein beliebiges Speicherobjekt dar. Seien

  – S_j die Größe des Speichers j,
  – s_i die Größe des Objekts i (wie zuvor),
  – e_j der Energieverbrauch pro Zugriff auf den Speicher j,
  – n_i die Anzahl von Zugriffen auf Objekt i (wie zuvor),
  – $x_{i,j}$ definiert als

$$x_{i,j} = \begin{cases} 1 & \text{wenn Objekt } i \text{ auf Speicher } j \text{ abgebildet wird} \\ 0 & \text{sonst} \end{cases} \tag{7.10}$$

Anstelle der Maximierung der Energieeinsparung minimieren wir nun den Gesamtenergieverbrauch. Daher ist das Ziel nun die Minimierung von

$$C = \sum_j e_j \sum_i x_{i,j} \cdot n_i \tag{7.11}$$

unter Berücksichtigung der Nebenbedingungen

$$\forall j : \sum_i s_i \cdot x_{i,j} \leq S_j \tag{7.12}$$

$$\forall i : \sum_j x_{i,j} = 1 \tag{7.13}$$

Partitionierte Speicher sind insbesondere für wechselnde Speicheranforderungen vorteilhaft. Häufig verwendete Speicherstellen werden auch die **Arbeitsmenge** (engl. *working set*) einer Anwendung genannt. Anwendungen mit einer kleinen Arbeitsmenge könnten einen sehr kleinen, schnellen Speicher verwenden, wogegen Anwendungen, die eine größere Arbeitsmenge benötigen, einem entsprechend größeren Speicher zugeordnet werden könnten. Ein wesentlicher Vorteil von partitionierten Speichern ist ihre Fähigkeit, sich der Größe des aktuellen Arbeitsmenge anzupassen. Weiterhin können ungenutzte Speicher zur Energieeinsparung abgeschaltet werden. Wir betrachten allerdings nur den „dynamischen" Energieverbrauch, der durch Speicherzugriffe verursacht wird. Zusätzlich kann aber auch Energie benötigt werden, wenn der Speicher nicht verwendet wird; dieser Verbrauch wird hier nicht betrachtet. Daher lassen sich Einsparungen durch das Abschalten von Speichern nicht in den Gleichungen (7.11) und (7.12) finden.

- ***Link*-/Ladezeit-Speicherzuordnung** [421]: Die Optimierung von Code auf eine feste SPM-Größe zur Übersetzungszeit hat einen Nachteil: der Code könnte eine schlechte Leistung aufweisen, wenn er auf Varianten eines Prozessors mit unterschiedlichen SPM-Größen ausgeführt wird. Es sollte vermieden werden, unterschiedliche ausführbare Programme für Varianten eines Prozessors zu erzeugen. Daher sind wir an ausführbaren Programmen interessiert, die von der SPM-Größe unabhängig sind. Dies lässt sich durch Optimierung zur *Link*-Zeit erreichen. Der vorgeschlagene Ansatz berechnet das Verhältnis der Zugriffszahlen dividiert durch die Größe einer Variablen zur Übersetzungszeit und speichert diesen Wert zusammen mit anderen Informationen über die Variable im ausführbaren Programm. Zur Ladezeit wird das Betriebssystem nach der Größe der SPMs gefragt. Daraufhin wird der Code modifiziert, so dass möglichst viele nutzbringende Variablen dem SPM zugeordnet werden.

Überlagernde Belegung

Große Anwendungen können mehrere *hot spots* (Codebereiche mit rechenintensiven Schleifen) haben. Nicht-überlagernde Ansätze liefern in diesem Zusammenhang nicht die bestmöglichen Ergebnisse. Für solche Anwendungen sollte der SPM für jeden der *hot spots* ausgenutzt werden. Dazu ist eine automatische Migration zwischen den einzelnen Ebenen der Speicherhierarchie erforderlich[2]. Diese Migration kann explizit in der Anwendung programmiert werden oder automatisch ausgeführt werden. Bei überlagernder Belegung nehmen wir meist an, dass alle Anwendungen zur Entwurfszeit bekannt sind. Die Algorithmen von Verma [554] und von Udayakumararan et al. [548] sind frühe Beispiele zur Behandlung dieser Situation.

Der Algorithmus von Verma startet mit dem Kontrollflussgraphen (*CFG*) einer einzelnen Anwendung. Für die Kanten dieses Graphen betrachtet Verma die Option, den SPM für lokal genutzte Speicherobjekte freizuräumen.

---

[2] Ein Teil des Stoffes in diesem Unterabschnitt findet sich auch in einem Kapitel desselben Autors in einem Buch desselben Verlages [377].

Beispiel 7.9:

In Abb. 7.15 betrachten wir Basisblöcke B1 bis B10 und eine Verzweigung des Kontrollflusses bei B2. Wir nehmen an, dass das Feld A im linken Pfad definiert, modifiziert und benutzt wird. T3 wird nur im rechten Pfad benutzt. Wir betrachten das potentielle Freiräumen des SPMs, sodass T3 lokal dem SPM zugewiesen werden kann. Dies erfordert Operationen zum Freiräumen und zum Laden in potentiell einzufügenden Blöcken B9 und B10 (rote gepunktete Linien: potentielle Einfügungen). Kosten und Nutzen dieser Operationen werden anschließend in ein globales ILP-Modell übernommen. Eine Lösung dieses Modells stellt eine optimale Menge von Kopieroperationen dar. ∇

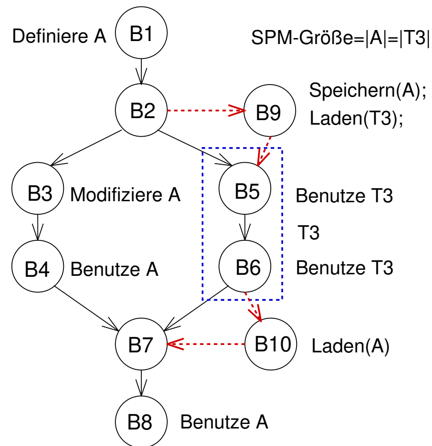


Abb. 7.15 Potentielles Freiräumen des Speichers

Verglichen mit dem nicht-überlagernden Fall wurde für eine Menge an Benchmarks eine durchschnittliche Reduktion des Energieverbrauchs um 34% und der Ausführungszeit von 18% beobachtet.

Der Algorithmus von Udayakumararan ist ähnlich, aber er bewertet Speicherobjekte anhand des Quotienten aus der Anzahl der Speicherzugriffe und der Größe der Objekte.

Schwierigkeiten ergeben sich bei der Zuordnung großer Felder zu SPMs. Tatsächlich kann schon ein einzelnes Feld zu groß sein, um einem SPM zugeordnet zu werden. Eine einzelne Aufspaltung eines größeren Feldes kann mit der Strategie von Verma et al. [160] vorgenommen werden. *Loop tiling* ist eine allgemeinere Technik, die entweder manuell oder automatisch eingesetzt werden kann [343]. Außerdem können Werte von Feldindices im Detail analysiert werden, um so häufig verwendete Elemente von Feldern im SPM zu halten [358].

Unsere Erklärungen betrafen bislang v.a. Code und globale Daten. Der *Stack* und der *Heap* benötigen besondere Aufmerksamkeit. In beiden Fällen können sehr einfache Lösungen möglich sein: In manchen Fällen wird man es vielleicht vorziehen, *Stack* oder *Heap*-Elemente nicht dem SPM zuzuordnen. In anderen Fällen können wir eine Analyse der Größe des *Stacks* [5] oder des *Heaps* ausführen [220], um zu prüfen, ob diese vollständig in den SPM passen und, falls dies der Fall ist, dem SPM zuweisen.

Für den *Heap* haben Dominguez et al. [134] vorgeschlagen, die Lebendigkeit von *Heap*-Objekten zu überprüfen und so nicht benötigte Objekte von der Speicherverwaltung auszuschließen. Wann auch immer ein Objekt potentiell benötigt wird, wird Code erzeugt, der sicherstellt, dass sich dieses Objekt im SPM befindet. Die Objekte befinden sich immer an derselben Adresse im SPM, sodass das Problem der potentiell ungültigen Speicheradressen vermieden wird. McIllroy et al. [385] schlagen eine dy-

namische Speicherverwaltung vor, welche die charakteristischen Eigenschaften von SPMs berücksichtigt. Bai et al. [32] schlagen vor, dass Programmierer alle Zugriffe über globale Zeiger mittels zweier Funktionen `p2s` und `s2p` kapseln. Diese beiden Funktionen konvertieren globale in lokale SPM-Adressen bzw. lokale Adressen in globale Adressen.

Für *Stack*-Variablen haben Udayakumararan et al. [548] vorgeschlagen, zwei *Stacks* zu benutzen: einen für kurze Funktionen, deren *Stack* im Speicher verbleibt und einen für zeitaufwändige Funktionen, deren *Stack* dem SPM zugeordnet wird. Kannan et al. [282] haben vorgeschlagen, die obersten *Stack-Frames* in zyklischer Weise im SPM zu halten. Bei Funktionsaufrufen wird geprüft, ob genug Platz für den *Frame* der gerufenen Funktion im SPM vorhanden ist. Wenn der Platz nicht vorhanden ist, dann werden alte *Stack-Frames* in einen reservierten Bereich des Speichers geschrieben. Bei der Rückkehr von Funktionsaufrufen werden diese *Stack-Frames* zurückkopiert. Verschiedene Optimierungen zielen auf die Minimierung der notwendigen Überprüfungen.

Mehrere *Threads*/Prozesse

Die bisher vorgestellten Ansätze sind noch darauf beschränkt, einen einzelnen Prozess oder *Thread*[3] zu unterstützen. Im Falle mehrerer Prozesse oder *Threads* muss berücksichtigt werden, dass Objekte bei Kontextwechseln in den SPM kopiert und aus diesem verdrängt werden müssen. Verma schlug die drei folgenden Ansätze vor [555] (dabei gelten Aussagen für Prozesse sinngemäß auch für *Threads*):

1. Im ersten Ansatz darf nur ein einzelner Prozess gleichzeitig Speicher im SPM belegen. Bei jedem Kontextwechsel wird die Information des verdrängten Prozesses im SPM gesichert und die Information für den auszuführenden Prozess wird wiederhergestellt. Dieser Ansatz wird als **speichernder/wiederherstellender** (engl. *saving/restoring*) **Ansatz** bezeichnet. Dieser Ansatz funktioniert nicht gut für große SPMs, da der Kopiervorgang selbst eine erhebliche Menge an Zeit und Energie benötigen würde.
2. Beim zweiten Ansatz wird der Speicherplatz des SPMs in Bereiche für die einzelnen Prozesse unterteilt. Die Größe der Partitionen wird durch eine spezielle Optimierung bestimmt. Der SPM wird während der Initialisierung befüllt. Es sind keine weiteren compilergesteuerten Kopiervorgänge erforderlich. Daher wird dieser Ansatz als **nicht-speichernder/wiederherstellender** (engl. *non-saving/restoring*) **Ansatz** bezeichnet. Dieser Ansatz ist nur sinnvoll für SPMs, die groß genug sind, um Speicherbereiche für mehrere Prozesse bereitzuhalten.
3. Der dritte Ansatz ist ein **hybrider** Ansatz: Der SPM wird in einen gemeinsam von allen Prozessen genutzten Bereich und einen zweiten Bereich, in dem Prozesse exklusiven Speicherplatz erhalten können, unterteilt. Die Größe beider Bereiche wird durch eine Optimierung bestimmt.

[3] Siehe Definitionen 4.1 und 4.2.

Die Ansätze von Verma setzen voraus, dass die Prozesse (bzw. *Threads*) zur Übersetzungszeit bekannt ist. Pyka et al. [463] beschreiben ein Verfahren, bei dem Prozesse dynamisch entstehen und beendet werden können sowie eine Laufzeit-SPM-Zuordnung, die durch einen im Betriebssystem integrierten SPM-Manager (SPMM) durchgeführt wird. Damit gelingt es, Speicherplatz für vorcompilierte Bibliotheken im SPM zuzuordnen. Leider benötigt Pykas Algorithmus eine zusätzliche Indirektionsebene. Trotz des Aufwands durch die indirekte Adressierung konnte der Energieverbrauch um 25%-35% im Vergleich zu einem 4-fach mengenassoziativen Cache reduziert werden.

Diese zusätzliche indirekte Adressierung kann vermieden werden, wenn eine Speicherverwaltungseinheit (engl. *Memory Management Unit* (MMU)) verfügbar ist. Egger et al. [149] entwickelten eine Technik, die MMUs ausnutzt: Zur Übersetzungszeit werden Codeabschnitte danach klassifiziert, ob sie von einer Zuordnung in den SPM profitieren oder nicht. Der vom SPM profitierende Code wird in einem bestimmten Bereich im virtuellen Adressraum gespeichert. Zu Anfang ist dieser Bereich nicht auf physikalischen Speicher abgebildet. Damit tritt ein Seitenfehler auf, sobald der Code zum ersten Mal ausgeführt wird. Die Seitenfehlerbehandlung ruft dann den SPMM auf und dieser allokiert (und deallokiert) Speicherplatz im SPM, wobei jeweils die Umsetzungstabellen von virtuellen zu physikalischen Adressen bei Bedarf aktualisiert werden. Dieses Verfahren ist für die Behandlung von Code entworfen worden und ist in der Lage, dynamisch variable Mengen von Anwendungen zu unterstützen. Leider entspricht die Größe von aktuellen SPMs lediglich einigen wenigen Einträgen in aktuellen Seitentabellen, sodass die SPM-Zuordnung relativ grobgranular bleibt.

Unterstützung verschiedener Architekturen und Zielfunktionen

Bislang haben wir lediglich verschiedene Formen der Zuordnung von SPM-Speicherplatz betrachtet. Die verschiedenen Architekturen bilden eine weitere Dimension in der Speicherplatzallokation. Implizit haben wir bislang Systeme mit einem einzelnen Rechenkern, einer einzigen Ebene der Speicherhierarchie und mit einem einzigen SPM betrachtet. Dabei existieren auch andere Architekturen. Beispielsweise können hybride Systeme sowohl Cache- wie auch SPM-Speicher enthalten. Wir können versuchen, Cachefehlerraten zu reduzieren, indem wir im Fall von Cachekonflikten selektiv SPM-Speicher zuordnen [281, 612, 92]. Wir können auch verschiedene Speichertechnologien haben, wie *Flash*-Speicher oder anderen nicht-flüchtigen Speicher [562]. Für *Flash*-Speicher ist es wichtig, eine übermäßige Abnutzung (engl. *wear-out*) einzelner Speicherbereiche durch ein gleichmäßiges Verteilen der Schreibvorgänge (engl. *load balancing*) zu vermeiden.

SPMs können möglicherweise von mehreren Rechenkernen genutzt werden. Auch kann es mehrere Ebenen der Speicherhierarchie geben, von denen eventuell einige von mehreren Kernen gemeinsam genutzt werden. Liu et al. [350] haben einen ILP-basierten Ansatz für eine Optimierung für eine solche Architektur beschrieben.

Die Zielfunktion ist eine weitere Dimension in der Zuordnung von SPM-Speicher. Bislang haben wir die Minimierung der Laufzeit und des durchschnittlichen Energieverbrauchs betrachtet. Es kann auch andere Optimierungsziele geben. Wir könnten auch den maximal möglichen Energieverbrauch (engl. *Worst Case Energy Consumption* (WCEC)) minimieren wollen. Die WCEC ist eine Zielfunktion in der Arbeit von Liu [350]. Für den Entwurf von zuverlässigen Systemen sind Zuverlässigkeit und Lebensdauer wichtige Zielfunktionen, insbesondere bei Systemen mit relevanter Alterung (engl. *aging*) [565]. Auch kann es notwendig sein, das thermische Verhalten zu betrachten.

7.3.4 Zusammenführung von Compilern und Zeitanalyse

Fast keiner der heute verfügbaren Compiler beinhaltet ein Zeitmodell. Daher muss die Entwicklung von Echtzeitsoftware normalerweise einem iterativen Ansatz folgen: Software wird vom einem Compiler übersetzt, der über keinerlei Zeitinformationen verfügt. Der resultierende Code wird dann mit einem *Timing*-Analysator wie aiT [4] analysiert. Wenn die Zeitbedingungen nicht eingehalten werden, müssen einige der Eingabedaten für den Compiler geändert und der Vorgang wiederholt werden. Wie nennen dies den **„Versuch-und-Irrtum-basierten Ansatz der Echtzeitsoftware”**. Dieser Ansatz bringt mehrere Probleme mit sich. Zunächst ist die Anzahl der notwendigen Entwurfsiterationen anfänglich unbekannt. Weiterhin verwendet der eingesetzte Compiler „Optimierungen”, aber eine präzise Auswertung der Zielkriterien mit Ausnahme der Codegröße ist unmöglich. Daher können Compilerentwickler nur **hoffen**, dass ihre „Optimierungen” einen positiven Einfluss auf die Codequalität für die relevanten Ziele hat. Aufgrund des komplexen Zeitverhaltens moderner Prozessoren wird diese Hoffnung allerdings kaum belegt. Schließlich erfordert die „Versuch-und-Irrtum”-basierte Entwicklung von Echtzeitsoftware, dass der Entwickler passende Änderungen der Eingabedaten für den Compiler findet, damit die Echtzeitbedingungen vielleicht irgendwann eingehalten werden.

Diese Probleme können vermieden werden, wenn die Zeitanalyse in den Compiler integriert wird. Dies war das Ziel der Entwicklung des WCET-gewahren Compilers WCC an der TU Dortmund. Dabei wurde der existierende Zeitanalysator aiT in den experimentellen WCC-Compiler für die TriCore-Architektur integriert. Abb. 7.16 zeigt die sich ergebende Gesamtstruktur des WCC.

WCC verwendet die ICD-C Compiler-Infrastruktur [231] zum Lesen und Schreiben von C-Quellcode. Der Quellcode wird dann in eine Zwischendarstellung auf hoher Ebene (engl. *High-Level Intermediate Representation* (HL-IR)) umgewandelt. Die HL-IR ist eine abstrakte Darstellung des Quellcodes, auf die verschiedene Optimierungen angewendet werden können. Die optimierte HL-IR wird dann an den *Code-Selector* übergeben. Der *Code-Selector* bildet Quellcodeoperationen auf Maschinenbefehle ab. Maschinenbefehle werden in der *Low-Level Intermediate Representation* (LLIR) dargestellt. Um die $WCET_{EST}$ zu bestimmen, wird die LLIR in die von aiT verwendete CRL2-Darstellung konvertiert (durch den Konverter LLIR2CRL).

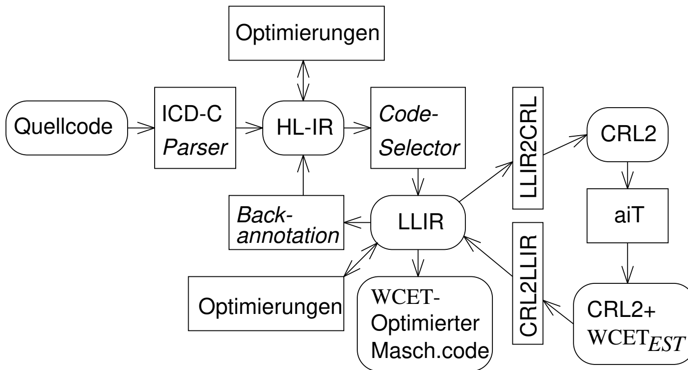


Abb. 7.16 WCET-gewahrer Compiler WCC

aiT kann dann die WCET$_{EST}$ für den angegebenen Maschinencode berechnen. Diese Information wird dann in die LLIR-Darstellung zurück konvertiert (durch den Konverter CRL2LLIR). WCC verwendet diese Information, um die WCET$_{EST}$ als Zielfunktion in der Optimierung berücksichtigen zu können. Dies kann für Optimierungen auf LLIR-Ebene sehr einfach erfolgen. Allerdings finden viele der Optimierungen auf der HL-IR-Ebene statt. WCET$_{EST}$-gesteuerte Optimierungen auf dieser Ebene erfordern den Einsatz von *back annotation* (Rückannotation) von der LLIR-Ebene zur HL-IR-Ebene. WCC beinhaltet diese *back annotation*.

WCC wurde verwendet, um den Einfluss der Optimierung für eine WCET$_{EST}$-Reduktion auf den Compiler zu untersuchen. Die vielfältigen Ergebnisse beinhalten eine Studie des Einflusses des Einsatzes dieses Ziels auf Registerallokation [158]. Wie in Abb. 7.17 zu sehen, zeigen die Ergebnisse eine dramatische Verbesserung auf.

Die WCET$_{EST}$ kann allein durch den Einsatz der WCET-gewahren Registerallokation im WCC im Durchschnitt auf 68,8% der ursprünglichen WCET$_{EST}$ gesenkt werden. Die größte Verbesserung ergibt eine WCET$_{EST}$ von nur 24,1% der ursprünglichen WCET$_{EST}$. Die Auswirkungen der Kombination verschiedener Optimierungen wurde von Lokuciejewski et al. [354] untersucht. Bei den betrachteten Benchmarks konnte Lokuciejewski eine Reduktion auf bis zu 57,1% der ursprünglichen WCET$_{EST}$ feststellen. Lokuciejewski et al. haben auch maschinelles Lernen eingesetzt, um Heuristiken zur Reduktion der WCET zu optimieren [355].

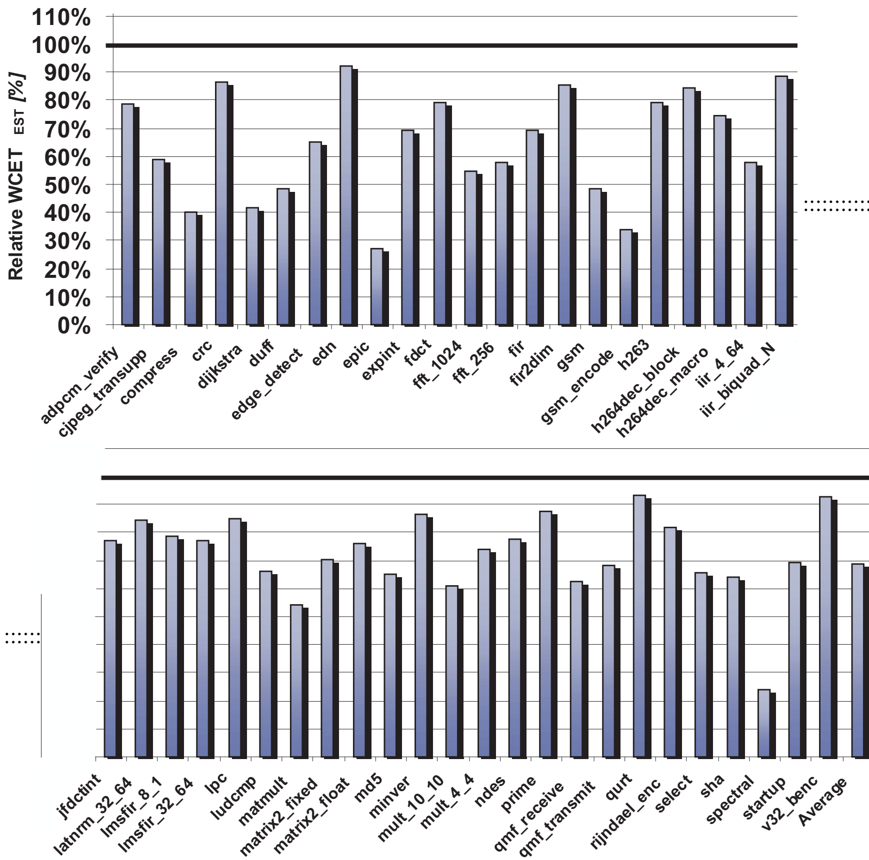


Abb. 7.17 Relative WCET_{EST} nach WCET-gewahrer Registerallokation

7.4 Energieverwaltung und thermisches Management

7.4.1 Dynamische Spannungsskalierung (DVS)

Einige eingebettete Prozessoren unterstützen dynamische Energieverwaltung (siehe Seite 161) und dynamische Spannungsskalierung (siehe Seite 159). Diese Eigenschaften können durch einen zusätzlichen Optimierungsschritt genutzt werden. Üblicherweise folgt ein solcher Schritt nach der Codeerzeugung durch den Compiler. In diesem Schritt vorgenommene Optimierungen erfordern eine globale Sicht auf alle Tasks des Systems zusammen mit ihren Abhängigkeiten, ihrem Schlupf usw.

Beispiel 7.10: Die Möglichkeiten, die dynamische Spannungsskalierung bietet, werden im folgenden Beispiel verdeutlicht [252]. Wir setzen einen Prozessor voraus, der mit den drei unterschiedlichen Spannungen 2,5 V, 4,0 V und 5,0 V betrieben werden

kann. Wenn wir einen Energieverbrauch von 40 nJ pro Zyklus bei 5,0 V zu Grunde legen, kann mit Hilfe von Gleichung (3.14) der Energieverbrauch bei Verwendung der anderen Betriebsspannungen berechnet werden (siehe Tabelle 7.1, hier ist 25 nJ ein gerundeter Wert). Weiterhin nehmen wir an, dass unsere Task innerhalb von 25

Tabelle 7.1 Eigenschaften eines Prozessors mit DVS-Funktionalität

V_{dd} [V]	5,0	4,0	2,5
Energie pro Zyklus [nJ]	40	25	10
f_{max} [MHz]	50	40	25
Zykluszeit [ns]	20	25	40

Sekunden 10^9 Zyklen ausführen muss. Dies ist auf verschiedene Arten machbar, wie in den Abb. 7.18 bis 7.20 dargestellt. Bei Verwendung der maximalen Spannung (Fall a), siehe Abb. 7.18) ist es möglich, den Prozessor während der nicht benötigten Rechenzeit von 5 Sekunden abzuschalten (wir gehen davon aus, dass der Energieverbrauch in diesem Zeitraum Null beträgt).

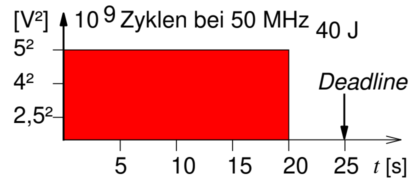


Abb. 7.18 Mögliche Spannungsverteilung (Fall a)

Eine weitere Möglichkeit (Fall b)) ist es, den Prozessor anfänglich mit voller Geschwindigkeit laufen zu lassen und die Spannung zu dem Zeitpunkt zu reduzieren, ab dem die übrigen Zyklen mit der niedrigsten verfügbaren Spannung abgearbeitet werden können (siehe Abb. 7.19).

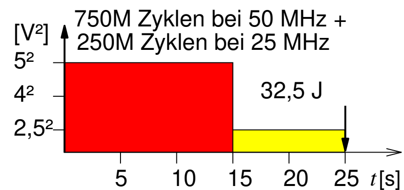


Abb. 7.19 Spannungsverteilung (Fall b)

Schließlich können wir den Prozessor mit einer Taktfrequenz betreiben, die gerade hoch genug ist, um alle Zyklen in der gegebenen Zeit abarbeiten zu können (siehe Abb. 7.20).

Der zugehörige Energieverbrauch lässt sich berechnen zu

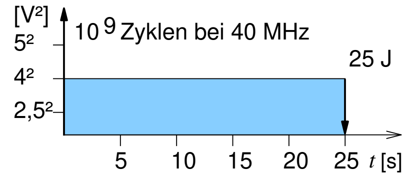


Abb. 7.20 Spannungsverteilung (Fall c)

$$E_a = 10^9 * 40 * 10^{-9}\,\text{J} = 40\,\text{J} \tag{7.14}$$
$$E_b = 750 * 10^6 * 40 * 10^{-9}\,\text{J} + 250 * 10^6 * 10 * 10^{-9}\,\text{J} = 32{,}5\,\text{J} \tag{7.15}$$
$$E_c = 10^9 * 25 * 10^{-9}\,\text{J} = 25\,\text{J} \tag{7.16}$$

Der minimale Energieverbrauch ergibt sich für die ideale Versorgungsspannung von 4 Volt. ∇

Im Folgenden verwenden wir den Begriff **Prozessor mit variabler Spannung** nur für solche Prozessoren, die mit einer **beliebigen** Spannung bis zu einem gegebenen Maximum betrieben werden können. Da es aufwändig ist, wirklich beliebige Spannungen zu unterstützen, verwenden reale Prozessoren nur einige wenige festgelegte Spannungen.

Die Erkenntnisse aus dem obigen Beispiel lassen sich wie folgt verallgemeinern. Beweise für diese Aussagen finden sich in der Veröffentlichung von Ishihara und Yasuura [252].

- Wenn ein Prozessor mit variabler Versorgungsspannung eine Task vor seiner *Deadline* beendet, kann der Energieverbrauch reduziert werden[4].
- Wenn ein Prozessor eine einzige Versorgungsspannung V_s verwendet und eine Task τ genau an der gegebenen *Deadline* beendet, dann ist V_s die Versorgungsspannung, die den Energieverbrauch von τ minimiert.

Wenn ein Prozessor nur eine bestimmte Anzahl diskreter Spannungsstufen verwenden kann, kann ein Spannungs-*Schedule* verwendet werden, das zwischen den beiden Spannungsstufen, die V_{ideal} am nächsten liegen, umschaltet. Die Verwendung dieser beiden Spannungen ergibt den minimalen Energieverbrauch. Es kann lediglich vorkommen, dass die Verwendung einer ganzzahligen Anzahl von Zyklen in einer kleinen Abweichung von diesem Minimum resultiert[5].

Mit den oben gegebenen Aussagen lassen sich Spannungen zu Tasks zuordnen. Nun betrachten wir die Zuordnung von Spannungen zu einer Menge von Tasks. Dabei verwenden wir die folgende Notation:

[4] Dies ist eine explizite Formulierung einer impliziten Annahme in Lemma 1 der Veröffentlichung von Ishihara and Yasuura.

[5] Diese Anforderung wird in der ursprünglichen Veröffentlichung nicht berücksichtigt.

n : Anzahl der Tasks,
EC_j : Anzahl ausgeführter Zyklen von Task j,
L : Anzahl von Spannungsstufen des Zielprozessors,
V_i : die i-te Spannungsstufe, $1 \leq i \leq L$,
f_i : die Taktfrequenz für Versorgungsspannung V_i,
d : die globale *Deadline*, an der alle Tasks beendet sein müssen,
SC_j : die durchschnittliche Schalthäufigkeit während der Ausführung von Task j (SC_i besteht aus der tatsächlichen Kapazität C_L und der Schalthäufigkeit α (siehe Gleichung (3.14) auf Seite 159)).

Damit lässt sich das Spannungsskalierungs-Problem als ein Ganzzahliges Lineares Programmierproblem (ILP)-Problem beschreiben (siehe Seite 425). Hierzu führen wir Variablen $X_{i,j}$ ein, welche die Anzahl von Zyklen beschreiben, die bei einer bestimmten Spannung ausgeführt werden:
$X_{i,j}$: Anzahl der Taktzyklen, die Task j bei Spannung V_i ausführt

Das ILP-Modell macht die folgenden vereinfachenden Annahmen:

- Es gibt nur einen Zielprozessor, der mit einer begrenzten Anzahl von diskreten Spannungen betrieben werden kann.
- Der Zeitaufwand für Spannungs- und Frequenzumschaltungen ist vernachlässigbar.
- Die größtmögliche Zyklenanzahl für jede Task ist bekannt.

Unter diesen Annahmen kann das ILP-Problem wie folgt beschrieben werden: Minimiere

$$E = \sum_{j=1}^{n} \sum_{i=1}^{L} SC_j * X_{i,j} * V_i^2 \tag{7.17}$$

unter den Randbedingungen

$$\forall j : \sum_{i=1}^{L} X_{i,j} = EC_j \tag{7.18}$$

und

$$\sum_{j=1}^{n} \sum_{i=1}^{L} \frac{X_{i,j}}{f_i} \leq d \tag{7.19}$$

Das Ziel ist es, die Anzahl $X_{i,j}$ an Zyklen zu finden, die eine Task j bei Spannung V_j ausgeführt wird. Wir haben weiter oben bereits festgestellt, dass keine Task jemals mehr als zwei Spannungen benötigen wird. Mit diesem Modell zeigen Ishihara und Yasuura, dass die Effizienz üblicherweise gesteigert werden kann, wenn Tasks aus einer größeren Anzahl von Spannungsstufen wählen können. Wenn ein großer

Schlupf verfügbar ist, erleichtern viele Spannungsstufen das Finden von solchen Spannungsstufen, die dem Optimum nahe sind. In der Praxis liefern aber bereits vier Spannungsstufen häufig gute Ergebnisse.

In vielen Fällen laufen Tasks tatsächlich schneller ab, als ihre vorhergesagte $WCET_{EST}$ angibt. Diese Eigenschaft kann vom obigem Algorithmus nicht ausgenutzt werden. Diese Einschränkung lässt sich beseitigen, indem *Checkpoints* eingeführt werden, zu denen die tatsächliche und *worst case*-Ausführungszeiten verglichen werden. Auf Basis dieser Information kann die Spannung potenziell reduziert werden [29]. Weiterhin wurde Spannungsskalierung in Multiraten-Taskgraphen vorgeschlagen [479]. DVS kann mit anderen Optimierungen wie *body biasing* [370] kombiniert werden. Die *body biasing*-Technik dient zur Reduktion von Leckströmen.

7.4.2 Dynamische Leistungsverwaltung (DPM)

Um den Energieverbrauch zu reduzieren, können wir auch Stromsparzustände, wie auf Seite 161 beschrieben, ausnutzen. Die grundlegende Frage bei der Verwendung von DPM ist es, wann ein System in den energiesparenden Zustand wechseln soll. Einfache Ansätze verwenden nur einen einfachen Zeitgeber, um in einen energiesparenden Zustand zu wechseln. Aufwändigere Ansätze modellieren die Ruhezeiten durch stochastische Prozesse und setzen diese dann dazu ein, um die Verwendung von Subsystemen mit größerer Genauigkeit vorherzusagen. Auf Exponentialverteilungen basierende Modelle haben sich dabei als unpräzise herausgestellt. Ausreichend genaue Modelle beruhen z.B. auf der *renewal theory* [490].

Es existieren einige Veröffentlichungen, die das Thema Energieverwaltung umfassend behandeln (z.B. [46, 357]). Zudem gibt es weiterentwickelte Algorithmen, die DVS und DPM zu einem einzigen Optimierungsansatz zur Einsparung von Energie zusammenfassen [491]. Die Zuordnung von Spannungen und die Berechnung von Übergangszeitpunkten für DPM könnten die beiden letzten Schritte der Optimierung eingebetteter Software darstellen.

Energieverwaltung ist eng mit dem Temperaturmanagement verknüpft.

7.4.3 Verwaltung des thermischen Verhaltens

Die Temperaturverwaltung basiert auf zur Laufzeit verfügbaren Temperaturinformationen. Diese Informationen werden verwendet, um die Erzeugung zusätzlicher Wärme zu steuern, zusätzlich hat sie einen Einfluss auf die Kühlmechanismen. Die Steuerung von Lüftern kann als ein sehr einfacher Fall von Temperaturverwaltung angesehen werden. Weiterhin könnten Systeme sich komplett oder teilweise abschalten, wenn die Temperaturen einen maximalen Grenzwert überschreiten. Es können so wieder abgeschaltete Teile von Silizium-*Chips* entstehen, die man *dark silicon* nennt. Fortschrittlichere Systeme könnten zudem Taktfrequenzen und Spannungen

absenken. Bei Multiprozessor-Systemen könnten Tasks automatisch zwischen verschiedenen Prozessoren migriert werden. In diesen Fällen wird das Ziel „Temperatur" zur Laufzeit ausgewertet und dazu verwendet, einen Einfluss zur Laufzeit zu erzeugen. Die Vermeidung von Überhitzung ist das Ziel von Arbeiten von Merkel et al. [392] und Donald et al. [135].

Wenn Temperatursensoren benutzt werden, um das thermische Verhalten von Systemen zu beeinflussen, dann entstehen Regelkreise. Grundsätzlich können solche Regelkreise instabil werden und schwingen. Atienza et al. haben verschiedene Strategien für solche Regelkreise verglichen und sind zu dem Schluss gekommen, dass eine bestimmte fortschrittliche Regelstrategie die besten Ergebnisse erzielt, im Vergleich zu Standardverfahren mit größerer Performanz bei niedrigeren Temperaturen [610]. Details einer solchen Strategie gehen über Basistechniken eingebetteter Systeme hinaus und werden daher hier nicht behandelt.

7.5 Aufgaben

Die folgenden Aufgaben sollten entweder zu Hause oder während einer Anwesenheitsphase nach dem *flipped classroom*-Konzept [376] bearbeitet werden:

7.1: Das Abrollen von Schleifen ist eine der nützlichen Optimierungen. Nennen Sie zwei mögliche Vorteile und zwei mögliche Nachteile des Abrollens!

7.2: Nehmen Sie an, dass Ihr Rechner einen Hauptspeicher und einen SPM enthält und dass wir auf Variablen wie in Tabelle 7.2 (links) beschrieben zugreifen. Die Größen und die pro Zugriff benötigte Energie sind in der Tabelle 7.2 (rechts) aufgeführt.

Variable	Größe [Bytes]	Anzahl Zugriffe
a	1024	16
b	2048	1024
c	512	2048
d	256	512
e	128	256
f	1024	512
g	512	64
h	256	512

Speicher	Größe [Bytes]	Energie je Zugriff
Scratchpad	4096 (4k)	1,3 nJ
Hauptspeicher	262.144 (256 k)	31 nJ

Tabelle 7.2 SPM mapping: **links**: Zugriffe auf Variablen **rechts**: Speicher-Charakteristiken

Welche der Variablen sollten dem SPM zugeordnet werden, wenn wir eine statische, nicht-überlagernde Zuordnung der Variablen voraussetzen? Benutzen Sie ein ILP-Modell, um die Variablen auszuwählen! Ihr Ergebnis sollte das ILP-Modell wie auch die ausgewählten Variablen enthalten. Zur Lösung des ILP-Modells können Sie beispielsweise das Programm lp_solve nutzen [17].

7.3: Angenommen, Sie möchten *loop tiling* benutzen. Wie können Sie diese Schleifentransformation an die vorhandenen Speicher anpassen?

7.4: Für welche Architekturen erwarten Sie den größten Nutzen, wenn Sie Gleitkommarechnungen durch Festkommarechnungen ersetzen?

7.5: Geben Sie eine Übersicht über Techniken zur Nutzung von *Scratchpad*-Speichern!

7.6: Betrachten Sie das nachfolgende Programm:

```
1  #include <stdio.h>
2  #define DATALEN 15
3  #define FILTERTAPS 5
4  double x[DATALEN] = { 128.0, 130.0, 180.0, 140.0, 120.0,
5                       110.0, 107.0, 103.5, 102.0,  90.0,
6                        84.0,  70.0,  30.0,  77.3,  95.7 };
7  const double h[FILTERTAPS]={0.125,-0.25,0.5,-0.25,0.125};
8  double y[DATALEN]; // result;
9  int main(void) {
10   int i,n;
11   for(i=0;i<DATALEN;++i) {
12     y[i] = 0;
13     for(n=0; n < FILTERTAPS; ++n)
14       if ((i-n) > = 0) y[i] += h[n]*x[i-n];
15   }
16   for(i = 0; i < DATALEN; ++i) printf("%.2f ",y[i]);
17   return 0;
18 }
```

Nehmen Sie mindestens die folgenden Optimierungen vor:

- Entfernung des `if` der innersten Schleife (Zeile 14)
- Abrollen der Schleife (Zeile 13)
- Umwandlung von Gleitkomma- in Festkommaformat
- Vermeidung aller Feldzugriffe

Erstellen Sie die optimierte Version des Programms nach jeder der Transformationen und prüfen Sie die Ergebnisse auf Konsistenz!

Open Access Dieses Kapitel wird unter der Creative Commons Namensnennung 4.0 International Lizenz (http://creativecommons.org/licenses/by/4.0/deed.de) veröffentlicht, welche die Nutzung, Vervielfältigung, Bearbeitung, Verbreitung und Wiedergabe in jeglichem Medium und Format erlaubt, sofern Sie den/die ursprünglichen Autor(en) und die Quelle ordnungsgemäß nennen, einen Link zur Creative Commons Lizenz beifügen und angeben, ob Änderungen vorgenommen wurden.

Die in diesem Kapitel enthaltenen Bilder und sonstiges Drittmaterial unterliegen ebenfalls der genannten Creative Commons Lizenz, sofern sich aus der Abbildungslegende nichts anderes ergibt. Sofern das betreffende Material nicht unter der genannten Creative Commons Lizenz steht und die betreffende Handlung nicht nach gesetzlichen Vorschriften erlaubt ist, ist für die oben aufgeführten Weiterverwendungen des Materials die Einwilligung des jeweiligen Rechteinhabers einzuholen.

Kapitel 8
Test

Leider können wir nicht davon ausgehen, dass ein entworfenes und ggf. produziertes System wie vorgesehen funktioniert. Es kann im laufenden Betrieb defekt geworden sein oder bereits Herstellungs- bzw. versteckte Entwurfsfehler aufweisen. Der Zweck des Testens ist es, zu überprüfen, ob ein eingebettetes/cyber-physikalisches System wie vorgesehen funktioniert. In diesem Kapitel gehen wir auf einige Grundbegriffe und Techniken dazu ein. Dazu geben wir eine kurze Einführung in die Ziele der Testmustererzeugung und deren Anwendung beim zu testenden System. In diesem Rahmen erklären wir die Begriffe der Fehlermodell, Fehlerüberdeckung, Fehlersimulation und Fehlerinjektion. Schließlich stellen wir schaltungstechnische Maßnahmen vor, mit denen das Testen unterstützt werden kann. Hierzu gehören insbesondere die Erzeugung von Pseudo-Zufallszahlen und die Signaturanalyse. Hilfreich kann es sein, Fragen der Testbarkeit bereits beim Entwurf zu berücksichtigen. Bei fehlertoleranten Systemen ist es unverzichtbar, die Fehlertoleranz detailliert zu überprüfen.

8.1 Anwendungsbereich

Tests können während oder nach der Produktion (sogenannte Produktionstests) und auch beim Kunden (sogenannte Feldtests) stattfinden. Das Testen eingebetteter Systeme, die in cyber-physikalischen oder IoT-Systemen enthalten sind, benötigt besondere Sorgfalt:

- Eingebettete Systeme, die in eine physische Umgebung integriert sind, können sicherheitskritisch sein. Daher kann ihre Fehlfunktion viel gefährlicher sein als beispielsweise die von Bürogeräten. Daher sind die Anforderungen an die Produktqualität höher als bei nicht sicherheitskritischen Systemen.
- Das Testen zeitkritischer Systeme muss das korrekte Zeitverhalten überprüfen. Das bedeutet, dass das Testen des funktionalen Verhaltens nicht ausreicht.

© Der/die Autor(en) 2021

P. Marwedel, *Eingebettete Systeme*, https://doi.org/10.1007/978-3-658-33437-6_8

- Der Test eingebetteter Systeme in ihrer echten Umgebung kann gefährlich sein. So kann der Test von Steuerungssoftware eines Kernkraftwerks eine Quelle ernsthafter, weitreichender Probleme sein.

Vorbereitungen für Tests sollten nicht später als zum Ende der Entwurfsphase erfolgen. Vorzugsweise sollte die notwendige Unterstützung für das Testen bereits früher berücksichtigt werden, verknüpft mit dem Entwurfsprozess und unter Verwendung der Testbarkeit als ein Zielkriterium bei der Bewertung von Entwürfen. Um Kapitel 5 nicht zu überfrachten, haben wir dennoch alle mit dem Testen zusammenhängenden Aspekte in diesem Kapitel zusammengefasst. Wir stellen den Test hier als Schritt am Ende des Entwurfsprozesses dar (siehe Abb. 8.1), obwohl es ratsam wäre, ihn früher während eines realen Entwurfs in Betracht zu ziehen. Eine frühe Berücksichtigung von Tests ist allerdings nicht immer gängige Praxis, daher könnte Abb. 8.1 auch einem tatsächlichen Entwurfsprozess entsprechen.

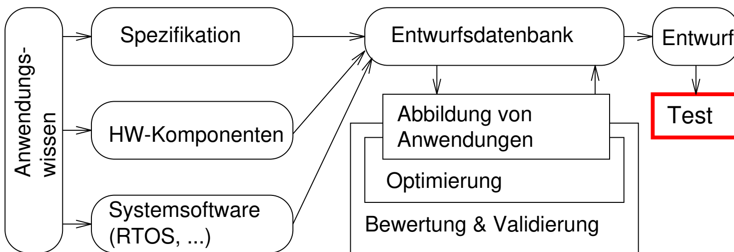


Abb. 8.1 Entwurfsfluss mit abschließendem Test

Bei Tests bezeichnen wir das im Entwurf befindliche System als zu testendes System oder Gerät im Test (engl. *Device Under Test* (DUT)). Das DUT wird einer Menge speziell ausgewählter Eingabemuster, sogenannter Testmuster, als Eingabe unterworfen, sein Verhalten wird dann analysiert und mit dem erwarteten Verhalten verglichen. Testmuster werden meist auf das reale, bereits gefertigte System angewendet. Testen beinhaltet eine Anzahl unterschiedlicher Aktionen:

1. **Testmustererzeugung**,
2. **Testmusteranwendung**,
3. **Überwachung der Antwort** und
4. **Vergleich der Ergebnisse**

8.2 Testverfahren

8.2.1 Testmustererzeugung für Modelle auf Gatterebene

Durch die Erzeugung von Testmustern versuchen wir, eine Menge an Testmustern zu identifizieren, die es ermöglichen, ein korrekt funktionierendes von einem nicht korrekt funktionierenden System zu unterscheiden. Die Testmustererzeugung basiert normalerweise auf **Fehlermodellen**. Solche Fehlermodelle sind Modelle möglicher Fehler. Die Testmustererzeugung versucht, Tests für alle nach einem bestimmten Fehlermodell möglicherweise auftretenden Fehler zu erzeugen.

Häufig wird das Haftfehler- (engl. *stuck-at-*) Modell verwendet. Dieses basiert auf der Annahme, dass eine interne Leitung einer elektronischen Schaltung entweder permanent mit `'0'` oder mit `'1'` verbunden ist. Beobachtungen zeigen, dass viele Fehler sich tatsächlich auf diese Art manifestieren.

Beispiel 8.1: Wir betrachten die Schaltung in Abb. 8.2.[1]. Wir möchten nun testen, ob ein *stuck-at*-1-Fehler für das Signal f vorliegt. Dazu versuchen wir, f auf `'0'`

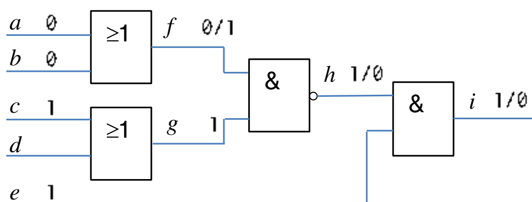


Abb. 8.2 Testmuster auf Gatterebene

zu setzen, indem wir $a = b =$`'0'` setzen. Als Ergebnis sollte f den Wert `'1'` besitzen, wenn der Fehler auftritt und ansonsten den Wert `'0'`. Um diesen Unterschied beobachten zu können, müssen wir diesen zum Ausgangssignal i weiterleiten. Dafür müssen wir e auf `'1'` und entweder c oder d auf `'1'` setzen. h und i werden den Wert `'1'` annehmen, wenn kein Fehler aufgetreten ist, ansonsten `'0'`. Das Testmuster umfasst alle Kombinationen von Eingabewerten für a bis e. Zur Erzeugung dieses Testmusters kann der D-Algorithmus verwendet werden [319]. ∇

Das *stuck-at*-Fehlermodell ist die Basis für viele Techniken zur Testmustererzeugung. CMOS-Technologien benötigen jedoch umfassendere Fehlermodelle. Bei CMOS-Schaltungen können Fehler aus einer kombinatorischen Schaltung eine Schaltung mit internen Zuständen machen. Dieses Problem kann auftreten, wenn Leitungen unterbrochen sind (dieser Fall ist als *stuck-open*-Fehler bekannt). Als Folge können Verbindungen zu den *Gates* von Transistoren unterbrochen werden. Solche Transistoren verhalten sich dann entweder leitend oder nicht leitend, abhängig von der

[1] In Übereinstimmung mit dem Standard ANSI/IEEE 91 kennzeichnen die Symbole ≥1 und & ODER- bzw. UND-Gatter.

vor der Unterbrechung im *Gate* vorhandenen Ladung. Auf diese Weise „erinnert" sich das *Gate* an sein Eingangssignal. Zudem können transiente Fehler und Verzögerungsfehler (Fehler, welche die Laufzeit einer Schaltung verändern) vorkommen. Verzögerungsfehler können die Folge von Übersprechen zwischen benachbarten Leitungen sein. Fehlermodelle, die solche Hardwarefehler betrachten, sind verfügbar [312].

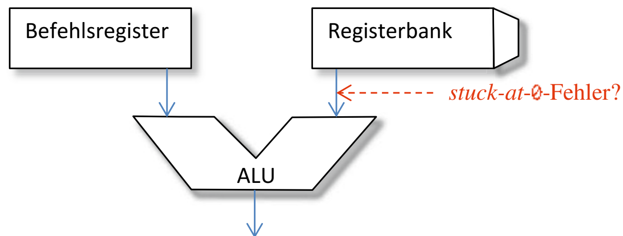
Während gute Fehlermodelle für Hardwaretests existieren, ist dies bei Softwaretests nicht der Fall.

8.2.2 Selbsttestprogramme

Eines der Hauptprobleme beim Test moderner integrierter Schaltungen ist deren begrenzte Anzahl an Kontakten. Dies erschwert immer mehr den Zugriff auf interne Komponenten. Zudem ist es mittlerweile sehr schwer, die Schaltungen bei voller Geschwindigkeit zu testen, da die Tester mindestens so schnell wie die Schaltung selbst sein müssen. Ein Ausweg aus diesem Dilemma bietet die Tatsache, dass viele eingebettete Systeme auf Prozessoren basieren: Prozessoren sind dazu in der Lage, Testprogramme oder **Diagnoseroutinen** auszuführen. Solche Diagnoseroutinen kommen bei Großrechnern schon seit Jahrzehnten zum Einsatz.

Beispiel 8.2: Abb. 8.3 zeigt einige Komponenten, die in einem Prozessor enthalten sein können.

Abb. 8.3 Einige Komponenten der Prozessorhardware



Um nun auf *stuck-at*-Fehler am Eingang der ALU zu testen, können wir ein kleines Testprogramm ausführen:

```
Speichere Testmuster bestehend aus '1'-Bits im Registersatz;
Führe XOR-Operation zwischen der Konstanten "0000...00" und dem Register aus,
Teste, ob das Ergebnis ein '0'-Bit enthält,
Wenn ja, melde Fehler;
ansonsten starte Test auf nächsten Fehler
```
∇

Ähnliche kleine Programme können für andere Fehler generiert werden. Unglücklicherweise ist die Erzeugung von Diagnoseroutinen für Großrechner bisher hauptsächlich manuell erfolgt. Es wurde vorgeschlagen, Diagnoseroutinen automatisch zu generieren [64, 314, 53, 313, 309, 48].

8.3 Auswertung von Testmustermengen und Systemrobustheit

8.3.1 Fehlerüberdeckung

Die Qualität von Testmustermengen kann mittels **Fehlerüberdeckung** als Metrik getestet werden.

Definition 8.1: Die Fehlerüberdeckung ist der Prozentsatz an möglichen Fehlern, der mit einer gegebenen Menge an Testmustern gefunden werden kann:

$$\text{Fehlerüberdeckung} = \frac{\text{Anzahl erkennbarer Fehler für gegebene Testmuster-Menge}}{\text{Anzahl von im Fehlermodell möglichen Fehlern}}$$

In der Praxis ist eine Fehlerüberdeckung im Bereich von mehr als 98% bis 99% zur Erzielung einer guten Produktqualität erforderlich. Die Anforderungen können für bestimmte Systeme noch höher liegen. Für manche Hardware, wie z.B. Batterien, können zudem spezielle Fehlermodelle erforderlich sein.

Zusätzlich zur Erzielung einer hohen Fehlerüberdeckung muss auch eine hohe sogenannte *correctness coverage* erzielt werden. Dies bedeutet, dass fehlerfreie Systeme auch als solche erkannt werden müssen. Ansonsten wäre es möglich, eine 100%-ige Fehlerüberdeckung zu erreichen, indem man alle Systeme als fehlerhaft klassifiziert.

Um die Anzahl an Möglichkeiten zur Validierung von Systemen zu erhöhen, wurde vorgeschlagen, Testmethoden bereits in der Entwurfsphase anzuwenden. Beispielsweise können Fehlermustermengen auf Softwaremodelle von Systemen angewendet werden, um zu überprüfen, ob sich zwei Softwaremodelle identisch verhalten. Zeitaufwendigere formale Methoden müssen nur in den Fällen zum Einsatz kommen, in denen das System diese testbasierte Gleichheitsprüfung bestanden hat.

8.3.2 Fehlersimulation

Es ist momentan nicht möglich (und dies wird sich voraussichtlich auch nicht ändern), das Verhalten von Systemen in Gegenwart von Fehlern vollständig vorherzusagen oder die Fehlerüberdeckung analytisch zu berechnen. Daher wird das Verhalten von Systemen unter Fehlereinfluss üblicherweise simuliert. Diese Art von Simulation wird **Fehlersimulation** genannt. Bei der Fehlersimulation werden Systemmodelle verändert, um das Verhalten des Systems bei Vorhandensein eines bestimmten Fehlers wiederzugeben. Die Fehlersimulation zielt darauf

- den Effekt eines Fehlers auf Systemebene herauszufinden (d.h. zu wissen, ob ein Fehler redundant ist),
- zu wissen, ob Mechanismen zur Verbesserung der Fehlertoleranz tatsächlich eine Verbesserung bringen.

Definition 8.2: Fehler werden **redundant** genannt, wenn sie das beobachtbare Verhalten des Systems nicht beeinflussen.

Die Fehlersimulation erfordert die Simulation des Systems für alle im Fehlermodell möglichen Fehler und für eine möglicherweise große Menge an unterschiedlichen Eingabemustern. Dementsprechend ist Fehlersimulation ein extrem zeitaufwendiger Vorgang. Zur Beschleunigung wurden daher verschiedene Verfahren vorgeschlagen.

Ein solches Verfahren bezieht sich auf Fehlersimulation auf Gatterebene. In diesem Fall sind die internen Signale einzelne Bits. Dies ermöglicht die Abbildung eines Signals auf ein einzelnes Bit eines Maschinenworts eines Simulationsrechners. UND- und ODER-Maschinenbefehle können damit verwendet werden, um Boolesche Netzwerke zu simulieren. Damit würde aber nur ein einzelnes Bit pro Maschinenwort verwendet werden. Die Effizienz wird durch parallele Fehlersimulation verbessert.

Definition 8.3: Fehlersimulation heißt **parallele Fehlersimulation**, wenn $n > 1$ verschiedene Testmuster gleichzeitig simuliert werden, wobei n die Länge eines Bitvektors ist, der als Datentyp von der simulierenden Maschine unterstützt wird.

Die Werte jedes der n Testmuster werden auf eine unterschiedliche Bitposition im Maschinenwort abgebildet. Die Ausführung derselben Menge an UND- und ODER-Befehlen simuliert dann das Verhalten des Booleschen Netzwerks für n Testmuster anstelle von nur einem Muster. AVX-Befehle (siehe Seite 170) sind für diese Simulation sehr hilfreich.

8.3.3 Fehlerinjektion

Für reale Systeme kann Fehlersimulation zu zeitaufwendig sein. Wenn echte Systeme verfügbar sind, kann stattdessen Fehlerinjektion zum Einsatz kommen. Bei der Fehlerinjektion werden real existierende Systeme modifiziert und der Gesamteinfluss auf das Systemverhalten überprüft. Fehlerinjektion beruht nicht auf Fehlermodellen (auch wenn sie zum Einsatz kommen können). Daher hat die Fehlerinjektion das Potential, Fehler zu erzeugen, die von einem Fehlermodell nicht vorhergesagt worden wären. Wir unterscheiden zwischen zwei Arten der Fehlerinjektion:

- lokale Fehler innerhalb des Systems und
- Fehler in der Umgebung (Verhalten, das nicht der Spezifikation entspricht: beispielsweise können wir die Reaktion des Systems testen, wenn es außerhalb der vorgesehenen Temperatur- oder Strahlungsbereiche betrieben wird).

Für die Fehlerinjektion kommen verschiedene Methoden zum Einsatz:

- Fehlerinjektion auf Hardwareebene: Beispiele sind Manipulation von Signalpegeln an Kontakten, elektromagnetische und nukleare Strahlung und

- Fehlerinjektion auf Softwareebene: Beispiele sind hier das Kippen von Bits im Speicher.

Der Testvorgang selbst kann einen Einfluss auf das Verhalten des Systems haben, z.B. durch eine zusätzliche Strombelastung oder ein geöffnetes Gehäuse. Dieser Einfluss sollte so gering wie möglich, im besten Fall vernachlässigbar, sein.

Experimente von Kopetz [304] zeigen, dass Software-basierte Fehlerinjektion in der Tat genauso effektiv wie Hardware-basierte Fehlerinjektion ist. Die einzige nennenswerte Ausnahme stellt hier nukleare Strahlung dar. Diese erzeugte Fehler, die mit anderen Methoden nicht erzeugt wurden.

8.4 Entwurf für Testbarkeit

8.4.1 Motivation

Ansätze für die Testmustererzeugung für Boolesche Schaltungen wurden in Unterabschnitt 8.2.1 vorgestellt. Für Schaltungen, die endliche Automaten implementieren, ist die Testmustererzeugung schwieriger. Der Vergleich, ob zwei endliche Automaten äquivalent sind, kann komplexere Eingabefolgen erfordern [302].

Beispiel 8.3: Wir betrachten das Zustandsdiagramm in Abb. 2.25, das hier der Übersichtlichkeit halber noch einmal in Abb. 8.4 dargestellt ist: Angenommen, wir möch-

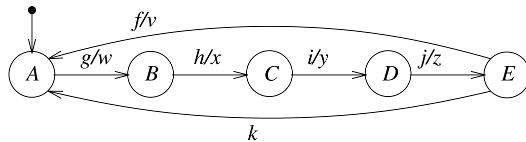


Abb. 8.4 Zu testender endlicher Automat

ten den Übergang von Zustand C zu Zustand D testen. Dies erfordert, dass wir zuerst Zustand C durch eine passende Folge von Eingabemustern erreichen. Daraufhin müssen wir das Eingabeereignis i erzeugen und überprüfen, ob die Ausgabe y erzeugt wird. Außerdem müssen wir überprüfen, ob wir Zustand D erreicht haben. Dieser Vorgang ist recht kompliziert, braucht eine Menge Zeit und ist anfällig für die Beeinflussung durch andere auftretende Fehler[2]. ∇

Dieses Beispiel zeigt: wenn Testen nur im Nachhinein stattfindet, kann es sehr schwierig sein, ein System zu testen. Um Tests zu vereinfachen, kann spezielle Hardware hinzugefügt werden. Der Vorgang, für bessere Testbarkeit zu entwerfen, nennt

[2] Der gesamte Test dieses speziellen Automaten wird dadurch vereinfacht, dass der Automat eine lineare Kette an Transitionen enthält (siehe auch die Übungsaufgaben zu diesem Kapitel).

sich **Entwurf für Testbarkeit** (engl. *Design for Testability* (DfT)). Ein bedeutendes Beispiel hierfür ist die Verwendung spezieller Hardware zum Testen endlicher Automaten.

8.4.2 Scanpfad-Entwurf

Das Erreichen bestimmter Zustände und die Überwachung von Zuständen, die aus der Anwendung von Eingabemustern resultieren, wird mit dem *Scan*pfad-Entwurf (engl. *scan design*) wesentlich erleichtert. Beim *Scan*pfad-Entwurf werden alle Flipflops, die Zustände speichern, verbunden und bilden serielle Schieberegister (siehe Abb. 8.5). Die Schaltung enthält drei D-Flipflops DFF und einen Multiplexer an

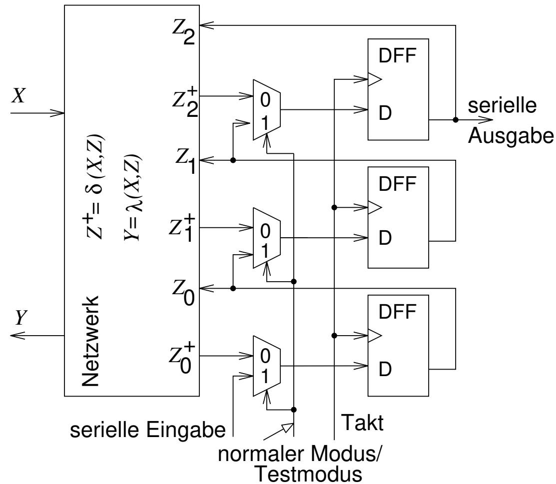


Abb. 8.5 *Scan*pfad-Entwurf

jedem Flipflop-Eingang. Unter Verwendung des Kontrolleingangs der Multiplexer (unter den Multiplexereingängen dargestellt) können wir die Flipflops entweder an das Schaltnetz anschließen, das aus aktuellem Zustand und aktuellen Eingabedaten den Folgezustand berechnet oder wir verbinden die Flipflops, sodass sie eine serielle Kette bilden. Wenn wir die Multiplexer in den *Scan*-Modus versetzen, können wir nacheinander Zustandbits in die *Scan*-Kette laden (ein Bit pro Taktzyklus). Auf diese Weise können wir seriell jeden beliebigen Zustand in die drei Flipflops laden. In einer zweiten Phase können wir ein Eingabemuster am endlichen Automaten anlegen, während sich die Multiplexer im normalen Modus befinden. Nach dem nächsten Taktzyklus befindet sich der Automat dann in einem neuen Zustand. Dieser neue Zustand kann nun in der dritten und letzten Phase wieder seriell hinausgeschoben werden (wieder ein Bit pro Taktzyklus). Im Endergebnis müssen wir uns also beim Erzeugen von Tests für den endlichen Automaten keine Gedanken darüber machen,

wie bestimmte Zustände erreicht werden können und wie überwacht werden kann, ob die Boolesche Funktion δ, die den Folgezustand berechnet, korrekt implementiert wurde. Tatsächlich hat die Tatsache, dass wir nun ein zustandsbasiertes System betrachten, nur einen Einfluss auf die beiden (einfachen) Schiebephasen, damit kann die Testmustererzeugung für (zustandslose) Boolesche Netzwerke verwendet werden, um auf korrekte Ausgaben zu prüfen. Damit reicht es also aus, Testmustererzeugungsverfahren für Boolesche Funktionen (zustandslose Netzwerke) zu verwenden und man muss sich keine Gedanken über komplexe Eingabesequenzen und ähnliches machen.

Für einzelne *Chips* funktioniert *Scan-Design* sehr gut. Wenn mehrere Schaltungen auf einer Platine integriert werden, benötigt man eine Technik, um *Scan*-Ketten mehrerer *Chips* miteinander zu verbinden. **JTAG** ist ein Standard, der dies ermöglicht. Dieser Standard definiert Register als Grenzen aller *Chips* und eine Anzahl von Testpins und Steuerbefehlen, um alle *Chips* miteinander in *Scan*-Ketten zu verbinden. JTAG wird auch als *boundary scan* bezeichnet [447].

8.4.3 Signaturanalyse

Um zu verhindern, dass auch die Antwort des zu testenden Systems hinausgeschoben wird, können Antworten komprimiert werden. Dazu kann eine Konfiguration wie in Abb. 8.6 verwendet werden. Erzeugte Testmuster werden als Eingabe (auch Stimuli

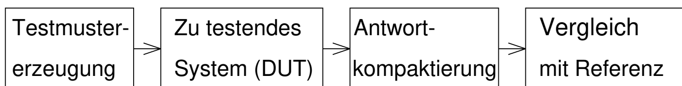


Abb. 8.6 Testen eines Systems

genannt) für das DUT verwendet. Die Antwort des DUT wird dann zu einer **Signatur** komprimiert, welche die Antwort beschreibt. Diese Antwort wird dann später mit der erwarteten Antwort verglichen. Die erwartete Antwort selbst kann mit Hilfe von Simulation berechnet werden.

Normalerweise wird die Komprimierung mit Hilfe linearer rückgekoppelter Schieberegister (engl. *Linear Feedback Shift Register* (LFSR)) durchgeführt, die über eine Rückkopplung mittels XOR verfügen.

Beispiel 8.4: Abb. 8.7 zeigt ein 4-Bit LFSR (links) und das zugehörige Zustandsdiagramm (rechts) [319]. Blaue gestrichelte Linien zeigen die Eingabe einer `'1'` an, rote durchgezogene Linien die Eingabe einer `'0'`. Die ausgewählte Rückkopplung erzeugt alle möglichen Signaturen. Während des Tests wird die Antwort des getesteten Systems an die Eingänge des LFSR angelegt. Das LFSR erzeugt daraufhin eine

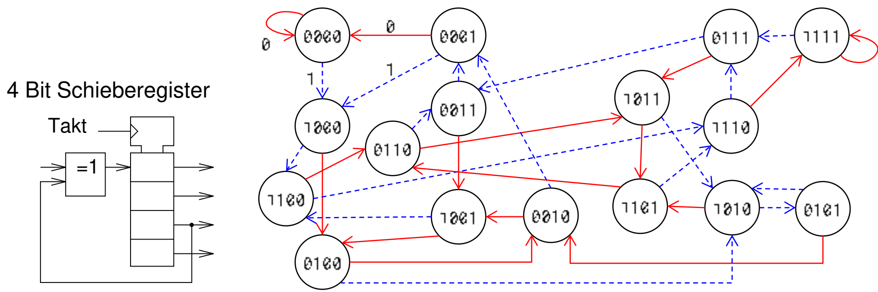


Abb. 8.7 Linear rückgekoppeltes Schieberegister zur Kompaktierung von Antworten: **links**: Schaltplan; **rechts**: Zustandsdiagramm

Signatur, welche die Antwort beschreibt. Die Signatur ist wesentlich kompakter als der komplette Strom von Eingabedaten. ∇

Da nicht die gesamte Antwort, sondern nur eine Signatur gespeichert wird, können unterschiedliche Antwortmuster auf dieselbe Signatur abgebildet werden. Wie hoch ist nun die Wahrscheinlichkeit dafür, eine korrekte Signatur aus einer inkorrekten Antwort zu erhalten?

Im Allgemeinen erzeugt ein n-Bit-Signaturgenerator 2^n Signaturen. Bei einer Antwort des DUT von m Bits können wir im besten Fall $2^{(m-n)}$ Antworten gleichmäßig auf eine Signatur abbilden. Wir erwarten nun, dass eine bestimmte Signatur für die korrekte Antwort des Systems erzeugt wird. Damit würden aber auch $2^{(m-n)} - 1$ inkorrekte Antworten auf dieselbe Signatur abgebildet werden. Bei m Bit langen Antworten gibt es insgesamt $2^m - 1$ inkorrekte Antworten. Daher beträgt die Wahrscheinlichkeit, dass eine inkorrekte Antwort auf die korrekte Signatur abgebildet wird (unter der Voraussetzung, dass die gegebenen Muster gleichmäßig auf die Signaturen verteilt sind):

$$P = Pr\left(\frac{\text{andere, auf dieselbe Signatur abgebildete Muster}}{\text{gesamte Anzahl anderer Muster}}\right) \tag{8.1}$$

$$= \frac{2^{(m-n)} - 1}{2^m - 1} \tag{8.2}$$

$$\approx \frac{2^{(m-n)}}{2^m} \text{ für } m \gg n \tag{8.3}$$

$$\approx \frac{1}{2^n} \text{ für } m \gg n \tag{8.4}$$

Damit ist die Wahrscheinlichkeit, dass eine korrekte Signatur aus einer inkorrekten Testantwort erzeugt wird, sehr gering, wenn das Schieberegister lang ist. Beispielsweise können 32-Bit Register zum Einsatz kommen. Trotzdem ist es möglich, dass falsche Werte am Eingang zu einer korrekten Signatur führen. Diesen Effekt nennt man ***Aliasing***. Die komprimierten Antworten verhalten sich sehr ähnlich wie CRC-

Zeichen, die zur Absicherung bei der Datenübertragung benutzt werden und die ebenfalls mit LFSR-Registern berechnet werden. Bei sicherheitskritischen Systemen ist eine sorgfältige Analyse des *Aliasings* ratsam.

8.4.4 Erzeugung von Pseudozufalls-Testmustern

Bei *Chips*, die eine große Anzahl an Flipflops enthalten, kann das Hineinschieben der Testmuster einige Zeit dauern. Der Vorgang der Erzeugung von Mustern auf dem *Chip* kann durch die Integration von Hardware zur Testmustererzeugung auf dem *Chip* beschleunigt werden. So können zum Beispiel Pseudozufallsmuster (die auch von den LFSRs erzeugt werden) als Testmuster zum Einsatz kommen.

Beispiel 8.5: Wir können die Schaltung aus Abb. 8.7 wie in Abb. 8.8 gezeigt abändern. Diese Schaltung erzeugt alle möglichen Testmuster mit Ausnahme des Musters, das ausschließlich aus Nullen besteht.

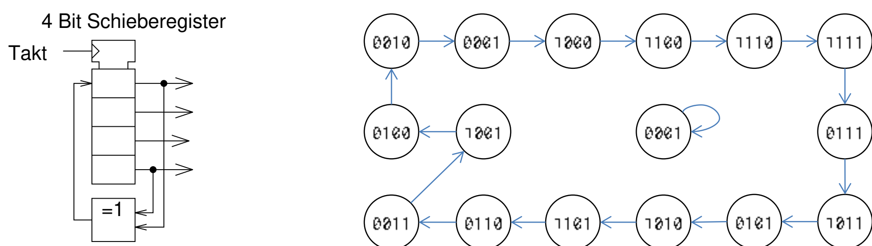


Abb. 8.8 Linear rückgekoppeltes Schieberegister zur Testmustererzeugung ∇

Ein Muster aus lauter Nullen muss vermieden werden, da ansonsten der Generator anhält, wenn er das Muster erreicht. Die so erzeugten Muster untersuchen zu testende Systeme meist sehr viel besser als einfache Zähler.

8.5 Aufgaben

Die folgenden Aufgaben sollten entweder zu Hause oder während einer Anwesenheitsphase nach dem *flipped classroom*-Konzept [376] bearbeitet werden:

8.1: Betrachten Sie die Schaltung in Abb. 8.2 und erzeugen Sie ein Testmuster für einen *stuck-at*-0-Fehler beim Signal h.

8.2: Welches Zustandsdiagramm entspricht dem LFSR aus Abb. 8.9?

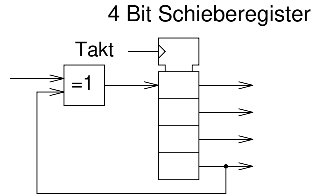


Abb. 8.9 LFSR

8.3: Geben Sie für den endlichen Automaten aus Abb. 8.4 Testmuster und erwartete Antworten an. Die Testmuster müssen als eine Folge von Paaren (Testmuster, erwartete Antwort) angegeben werden. In Abb. 8.4 gezeigte Ereignisse können als Testmuster verwendet werden. Wir gehen davon aus, dass sich der Automat nach dem Einschalten im Ausgangszustand befindet. Erstellen Sie einen umfassenden Test für alle möglichen Übergänge! Denken Sie dabei daran, dass die besondere Kettenstruktur des Automaten das Testen vereinfacht.

Open Access Dieses Kapitel wird unter der Creative Commons Namensnennung 4.0 International Lizenz (http://creativecommons.org/licenses/by/4.0/deed.de) veröffentlicht, welche die Nutzung, Vervielfältigung, Bearbeitung, Verbreitung und Wiedergabe in jeglichem Medium und Format erlaubt, sofern Sie den/die ursprünglichen Autor(en) und die Quelle ordnungsgemäß nennen, einen Link zur Creative Commons Lizenz beifügen und angeben, ob Änderungen vorgenommen wurden.

Die in diesem Kapitel enthaltenen Bilder und sonstiges Drittmaterial unterliegen ebenfalls der genannten Creative Commons Lizenz, sofern sich aus der Abbildungslegende nichts anderes ergibt. Sofern das betreffende Material nicht unter der genannten Creative Commons Lizenz steht und die betreffende Handlung nicht nach gesetzlichen Vorschriften erlaubt ist, ist für die oben aufgeführten Weiterverwendungen des Materials die Einwilligung des jeweiligen Rechteinhabers einzuholen.

Anhang A
Ganzzahlige Lineare Programmierung

In diesem Buch wird der Anhang benutzt, um grundlegende Kenntnisse zu vermitteln, die für das Verständnis der vorangegangenen Kapitel erforderlich sind, die man aber nicht unbedingt voraussetzen kann. Dabei werden drei Themenkreise behandelt. Im aktuellen Anhang wird die Ganzzahlige Lineare Programmierung vorgestellt. Die Ganzzahlige Lineare Programmierung (engl. *Integer Linear Programming* (ILP)) ist eine mathematische Optimierungstechnik, die auf eine große Zahl von Optimierungsproblemen anwendbar ist. ILP-Modelle sind ein allgemeiner Ansatz, um Optimierungsprobleme zu lösen.

ILP-Modelle bestehen aus zwei Teilen: einer Kostenfunktion und einer Menge an Nebenbedingungen. Beide Teile referenzieren eine Menge $X = \{x_i\}$ ganzzahliger Variablen. Die Kostenfunktionen müssen lineare Funktionen dieser Variablen sein. Damit haben sie die allgemeine Form

$$C = \sum_i a_i x_i, \text{ with } a_i \in \mathbb{R}, x_i \in \mathbb{N}_0 \tag{A.1}$$

Die Menge der Nebenbedingungen J muss ebenfalls aus linearen Funktionen über ganzzahligen Variablen bestehen. Diese müssen folgende Form haben:

$$\forall j \in J : \sum_i b_{i,j} x_i \geq c_j \text{ with } b_{i,j}, c_j \in \mathbb{R} \tag{A.2}$$

Definition A.1: Die **Ganzzahlige Lineare Programmierung (ILP)** beschreibt die Aufgabe, die Kostenfunktion von Gleichung (A.1) unter den gegebenen Nebenbedingungen von Gleichung (A.2) zu minimieren. Wenn alle Variablen entweder den Wert 0 oder 1 annehmen müssen, dann wird das entsprechende Modell ein **0/1-ILP-Modell** genannt. In diesem Fall werden die Variablen auch als **(binäre) Entscheidungsvariablen** bezeichnet.

Hierbei kann in Gleichung (A.2) $\geq$ durch $\leq$ ersetzt werden, wenn die Konstanten $b_{i,j}$ entsprechend angepasst werden. Der Fall von negativen Variablen x_i entsteht, wenn man zulässt, dass x_i einen beliebigen ganzzahligen Wert annehmen kann. Er kann in den Fall nicht-negativer Variablen gewandelt werden, indem man die

© Der/die Autor(en) 2021

P. Marwedel, *Eingebettete Systeme*, https://doi.org/10.1007/978-3-658-33437-6

Konstanten mit -1 multipliziert. Anwendungen, welche die **Maximierung** einer **Gewinnfunktion** C' erfordern, können in die obige Form gebracht werden, indem $C = -C'$ gesetzt wird.

Beispiel A.1: Unter der Annahme, dass x_1, x_2 und x_3 ganzzahlig sein müssen, stellt das folgende Gleichungssystem ein 0/1-ILP-Modell dar:

$$C = 5x_1 + 6x_2 + 4x_3 \tag{A.3}$$

$$x_1 + x_2 + x_3 \geq 2 \tag{A.4}$$

$$x_1 \leq 1 \tag{A.5}$$

$$x_2 \leq 1 \tag{A.6}$$

$$x_3 \leq 1 \tag{A.7}$$

Durch die Nebenbedingungen nehmen alle Variablen entweder den Wert 0 oder 1 an. Es gibt vier mögliche Lösungen, diese sind in Tabelle A.1 aufgeführt. Die Lösung mit Kosten von 9 ist optimal.

Tabelle A.1 Mögliche Lösungen des vorgestellten ILP-Problems

x_1	x_2	x_3	C
0	1	1	10
1	0	1	9
1	1	0	11
1	1	1	15

∇

ILP ist eine besondere Variante der Linearen Programmierung (LP). Bei Linearer Programmierung können Variablen beliebige reelle Werte annehmen. ILP- und LP-Modelle sind optimal durch Anwendung mathematischer Programmiertechniken lösbar. Leider ist ILP NP-vollständig (LP hingegen nicht) und die Ausführung einer ILP-Optimierung kann sehr viel Zeit erfordern.

Dennoch sind ILP-Modelle nützlich, um Optimierungsprobleme zu modellieren, solange die Modelle nicht sehr groß werden. Die Modellierung von Optimierungsproblemen als ILP-Probleme ist trotz der Komplexität des Problems sinnvoll: viele Probleme lassen sich in akzeptabler Zeit lösen. Wenn dies einmal nicht der Fall ist, liefert das ILP-Modell einen guten Ausgangspunkt zur Entwicklung von Heuristiken. Die Ausführungszeit hängt von der Anzahl der Variablen und der Anzahl und Beschaffenheit der Nebenbedingungen ab. Gute ILP-Löser (wie lp_solve [17] oder CPLEX) können gut strukturierte Probleme mit einigen tausend Variablen in akzeptabler Zeit, meist im Bereich von Minuten, lösen. Mehr Informationen zum Thema ILP und LP findet man in Fachliteratur zu diesen Themen, z.B. bei Wolsey [594].

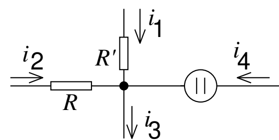
Anhang B
Kirchhoffsche Gesetze und Operationsverstärker

Unsere Beschreibung von D/A-Wandlern ab Seite 197 setzt einige Grundkenntnisse über Operationsverstärker voraus. Informatikstudenten verfügen häufig nicht über dieses Wissen, daher stellen wir die notwendigen Grundlagen in diesem Anhang vor. Diese Grundlagen bauen auf dem Verständnis der Kirchhoffschen Regeln auf, die hier der Vollständigkeit halber ebenfalls vorgestellt werden.

Kirchhoffsche Regeln

Die Kirchhoffschen Regeln sind ein Mittel, um elektrische Schaltungen zu analysieren. Die erste Regel ist die Knotenregel. Diese gilt für Knoten wie den in Abb. B.1 gezeigten.

Abb. B.1 Ein Knoten in einer elektrischen Schaltung



Theorem B.1 (Kirchhoffsche Knotenregel [274]): *In jedem Punkt einer elektrischen Schaltung ist die Summe der zu diesem Punkt hinfließenden Ströme gleich der Summe der von diesem Punkt wegfließenden Ströme. Formal gilt also für jeden Knoten einer Schaltung:*

$$\sum_k i_k = 0 \tag{B.1}$$

Wenn diese Regel in der Form von Gleichung (B.1) verwendet wird, dann müssen Ströme, die durch einen von einem Knoten wegführenden Pfeil gekennzeichnet sind,

© Der/die Autor(en) 2021

P. Marwedel, *Eingebettete Systeme*, https://doi.org/10.1007/978-3-658-33437-6

als negativ betrachtet werden, wobei diese Zählung von der tatsächlichen Richtung des Stromflusses unabhängig ist.

Beispiel B.1: Für die Ströme aus Abb. B.1 gilt

$$i_1 + i_2 - i_3 + i_4 = 0 \tag{B.2}$$

$$i_1 + i_2 + i_4 = i_3 \tag{B.3}$$

Diese Invarianz ist eine Folge der Erhaltung der elektrischen Ladung. Ohne diese Regel würde die Gesamtladung nicht konstant bleiben und die Spannung ansteigen.

Die zweite Kirchhoffsche Regel beschreibt Maschen in einer Schaltung und ist daher auch als Maschenregel bekannt. Ein Beispiel ist in Abb. B.4 zu sehen.

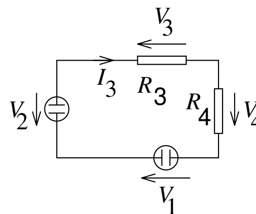


Abb. B.2 Eine Masche in einer elektrischen Schaltung

Theorem B.2 (Kirchhoffsche Maschenregel [274]): *Die Summe der Potentialunterschiede über alle Elemente einer Masche muss Null ergeben . Für jede Masche in einer Schaltung gilt also die Formel:*

$$\sum_k V_k = 0 \tag{B.4}$$

Wenn Spannungen entgegen der Pfeilrichtung betrachtet werden, müssen diese als negativ angesehen werden.

Beispiel B.2: Für die Schaltung in Abb. B.2 gilt

$$V_1 - V_2 - V_3 + V_4 = 0 \tag{B.5}$$

Die zugrunde liegende Ursache hierfür ist die Energieerhaltung. Ohne diese Regel könnte eine Ladung in der Masche beschleunigt werden und die Ladung würde an Energie gewinnen, ohne dass an einer anderen Stelle Energie „verbraucht" werden würde.

Im allgemeinen ist es unwichtig, in welche Richtung die Elektronen in einer Schaltung tatsächlich fließen und welcher Anschluss das höhere Potential im Vergleich zu einem anderen Anschluss besitzt. Die Richtung der Pfeile kann also beliebig gewählt werden. Es muss nur sichergestellt sein, dass die Richtung der Pfeile bei der Anwendung der Kirchhoffschen Regeln berücksichtigt wird. Wenn Pfeile für

Spannungen und Ströme über Komponenten in unterschiedliche Richtungen zeigen, muss die Gleichung für diese Komponente dies berücksichtigen.

Beispiel B.3: Daher liest sich beispielsweise das Ohmsche Gesetz für den Widerstand R_3 in Abb. B.2 wie folgt :

$$I_3 = -\frac{V_3}{R_3} \tag{B.6}$$

Natürlich wird man meist versuchen, die Richtung von Spannungen und Strömen so zu wählen, dass die Anzahl der erforderlichen Minuszeichen minimiert wird.

Operationsverstärker

In elektronischen Schaltungen muss oft ein Signal $x(t)$ verstärkt werden, um ein verstärktes Signal $y(t) = a \cdot x(t)$ mit $a > 1$ zu erhalten. a wird der **Verstärkungsfaktor** genannt. Es wäre sehr aufwendig, unterschiedliche Schaltungen für jeden möglichen Verstärkungsfaktor zu entwerfen. Deshalb wird oft ein allgemeiner Verstärker verwendet, der einfach auf den benötigten Verstärkungsfaktor eingestellt werden kann. Solch ein allgemeiner Verstärker heißt **Operationsverstärker** (engl. *Operational Amplifier*, kurz Op-Amp). Op-Amps sind auf einen sehr großen maximalen Verstärkungsfaktor ausgelegt. Die tatsächlich benötigte Verstärkung kann durch die passende Auswahl einiger Hardwarekomponenten in der zugehörigen Schaltung eingestellt werden.

Genauer betrachtet, ist ein Operationsverstärker eine Komponente mit zwei Signaleingängen und einem Signalausgang. Zusätzlich hat ein Operationsverstärker noch mindestens zwei Eingänge für die Spannungsversorgung (siehe Abb. B.3).

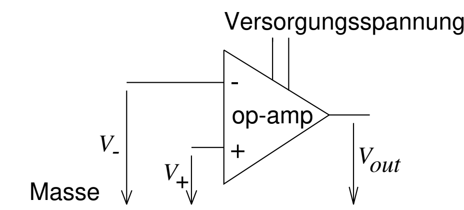


Abb. B.3 Operationsverstärker

Operationsverstärker verstärken die Differenz zwischen den Spannungen an den beiden Signaleingängen bezogen auf das Massepotential um einen Faktor g:

$$V_{out} = g * (V_+ - V_-) \tag{B.7}$$

g wird die **Leerlaufverstärkung** genannt. Diese ist meist sehr groß ($10^4 < g < 10^6$), bei einem idealen Op-Amp würde sie unendlich erreichen. Weiter verfügen Op-Amps normalerweise über eine sehr hohe Eingangsimpedanz ($> 1M\Omega$). Daher können wir

bei der Betrachtung dieser Bauteile Eingangsströme für gewöhnlich vernachlässigen. Bei einem idealen Operationsverstärker wäre die Eingangsimpedanz unendlich und damit die Eingangsströme gleich Null.

Operationsverstärker sind seit vielen Jahrzehnten kommerziell verfügbar, sowohl als integrierte Schaltungen wie auch als Teile anderer Schaltungen. Die verfügbaren Modelle unterscheiden sich in Geschwindigkeit, Spannungsbereichen, maximalem Ausgangsstrom und weiteren Eigenschaften. Der tatsächliche Verstärkungsfaktor der Schaltung wird mit externen Widerständen eingestellt. Abb. B.4 zeigt, wie dies erfolgen muss.

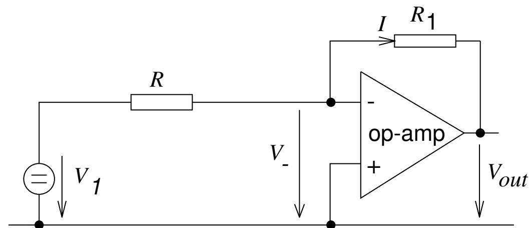


Abb. B.4 Operationsverstärker mit Rückkopplung

Jede noch so kleine Spannungsdifferenz zwischen den beiden Signaleingängen wird um einen großen Faktor verstärkt. Die entstehende Ausgangsspannung wird über den Widerstand R_1 zurückgekoppelt. Diese Rückkopplung geht an den invertierenden Eingang, damit ergibt jede positive Spannung V_- eine negative Spannung V_{out} und umgekehrt. Damit arbeitet die Rückkopplung durch die große Verstärkung stark gegen die Eingangsspannung an. Die Rückkopplung senkt also die Spannung am Eingangskontakt. Die Frage ist, um wie viel? Wir können die Kirchhoffschen Regeln anwenden, um die sich ergebende Spannung V_- zu berechnen (siehe Abb. B.5).

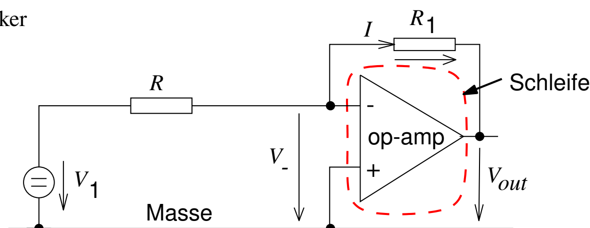


Abb. B.5 Operationsverstärker mit Rückkopplung (Masche hervorgehoben)

Wegen der Eigenschaften von Operationsverstärkern gilt

$$V_{out} = -g * V_- \tag{B.8}$$

Die Maschenregel für die mit einer gestrichelten Linie markierte Masche in Abb. B.5 ergibt

$$I * R_1 + V_{out} - V_- = 0 \tag{B.9}$$

Wir verwenden hier ein Minuszeichen für V_-, da wir einen Abschnitt der Masche entgegen der Pfeilrichtung betrachten. Aus den Gleichungen (B.8) und (B.9) ergibt sich dann

$$I * R_1 + (-g) * V_- - V_- = 0 \tag{B.10}$$

$$(1 + g) * V_- = I * R_1 \tag{B.11}$$

$$V_- = \frac{I * R_1}{1 + g} \tag{B.12}$$

$$V_{-,ideal} = \lim_{g \to \infty} \frac{I * R_1}{1 + g} \tag{B.13}$$

$$= 0 \tag{B.14}$$

Damit ist für einen idealen Operationsverstärker der Wert von V_- gleich 0. Daher wird der invertierende Signaleingang auch **virtuelle Masse** genannt. Dennoch darf dieser Anschluss nicht mit der Masse verbunden werden, da dies die Ströme verändern würde.

Die Berechnung der tatsächlichen Verstärkung in Abb. B.4 ist eine Übungsaufgabe zu Kapitel 3.

Anhang C
Seitenadressierung und Speicherverwaltungseinheiten

In diesem Anhang präsentieren wir eine Basistechnik zur Speicherverwaltung, die eine Grundlage des Verständnisses moderner Ausführungsplattformen ist. In sehr einfachen Rechensystemen werden die physikalischen Speicher tatsächlich mit den Adressen adressiert, die vom (Assembler-) Programmierer gesehen werden. Aus reiner Hardwaresicht ist dieser Ansatz einfach zu realisieren. Die Nutzung leidet allerdings unter einer Reihe von Nachteilen: so ist beispielsweise die Zuordnung von Objekten zum Speicher sehr statisch und während der Ausführung kaum zu ändern. Daher muss die Größe der Objekte vor der Zuordnung von Speicher bekannt sein oder zumindest relativ gut abgeschätzt werden können.

Mehr Flexibilität wird erreicht, wenn wir unterscheiden zwischen den Adressen, welche der (Assembler-) Programmierer sieht, und den Adressen, mit denen wirklich der Speicher adressiert wird. Die Adressen, welche der Programmierer sieht, heißen **virtuelle Adressen**. Die Adressen, mit denen der Speicher adressiert wird, heißen **reale Adressen**.

Bei der **Seitenadressierung** (engl. *paging*) partitionieren wir den Raum der virtuellen Adressen in Blöcke gleicher Größe, genannt **Seiten**. Die Größe dieser Seiten ist eine Zweierpotenz, z.B. 2 kBytes oder 4 kBytes. Daher bestehen virtuelle Adressen aus den Bits, die eine bestimmte Seite adressieren, sowie aus den Bits, die ein bestimmtes Wort oder Byte innerhalb der Seite adressieren. Die erste Menge der Bits heißt **Seitennummer**, die zweite *Offset*. Der physikalische Speicher wird in Blöcke gleicher Größe partitioniert, die Seitenrahmen (engl. *page frames*) genannt werden.

Mit Hilfe einer sogenannten Seitentabelle (engl. *page table*) kann man nun den Seitennummern Anfangsadressen des korrespondierenden Blocks im physikalischen Speicher zuordnen. Der *Offset* ist für virtuelle und reale Adressen identisch (siehe Abb. C.1 (links)). Dies erlaubt eine dynamische Zuordnung von Objekten zum Speicher. Zusammenhängende Bereiche im virtuellen Adressraum müssen nicht mehr zusammenhängenden Bereichen im realen Adressraum zugeordnet werden (siehe Abb. C.1 (rechts)).

Es kann mehr als einen virtuellen Adressraum geben, beispielsweise je einen Adressraum für jeden Prozess, der dem Betriebssystem bekannt ist. In diesem Fall

© Der/die Autor(en) 2021

P. Marwedel, *Eingebettete Systeme*, https://doi.org/10.1007/978-3-658-33437-6

virtuelle Adresse
Seite | Offset
Seitentabelle
frame | Offset
reale Adresse

virtueller Adressraum
physikalischer Adressraum
......
......

Abb. C.1 Seitenadressierung: **links**: Adressberechnung; **rechts**: Abbildung der Adressräume

muss beim Prozesswechsel jeweils dafür gesorgt werden, dass ein Verweis auf die relevante Seitentabelle gesetzt wird. Die Abbildung von virtuellen Adressen auf reale Adressen wird typischerweise durch eine spezielle Speicherverwaltungseinheit (engl. *Memory Management Unit* (MMU)) unterstützt. Die MMU befindet sich zwischen den Prozessoren und dem Speicher. Sie übersetzt virtuelle Adressen in reale Adressen. Für ihre Arbeit benötigt die MMU die Kenntnis der jeweils relevanten Seitentabelle. Die Seitentabellen können groß werden und daher können spezialisierte schnelle und kleine Puffer für diese Tabellen erforderlich werden. Solche Puffer nennt man ***Translation Look-aside Buffers*** (TLB) oder ***Address Translation Memories*** (ATM). Für häufig benutzte Seitennummern sollen TLBs die benötigten Adressumrechnungs-Informationen enthalten.

Für PCs wird der beschriebene Ansatz häufig mit ***demand paging*** kombiniert. Hier müssen sich nicht alle benötigten Seiten auch im Hauptspeicher befinden. Wird auf eine Seite zugegriffen, die sich nicht im Hauptspeicher befindet, spricht man von einem **Seitenfehler** (engl. *page fault*). In einem solchen Fall muss die benötigte Information von einem langsameren Hintergrundspeicher nachgeladen werden. Der Begriff *paging* wird häufig mit *demand paging* gleichgesetzt. Allerdings ist es für eingebettete Systeme wichtig, zwischen der reinen Partitionierung von Adressbereichen in Seiten und Seitenrahmen sowie dem automatischen Nachladen von einem Hintergrundspeicher zu unterscheiden. *Paging* ist auch für den **Speicherschutz** nützlich. Seitentabellen enthalten häufig auch Informationen über Zugriffsrechte für den jeweiligen Eintrag. So können wir zwischen Lese-, Schreib- und Ausführungsberechtigungen unterscheiden. Damit ist es möglich, die Adressbereiche zu schützen, beispielsweise indem einem Prozess nur Leserechte eingeräumt werden. So können fehlerhafte, kompromittierende und/oder sicherheitskritische Zugriffe vermieden werden. Der zweite Aspekt wird beim Internet der Dinge zunehmend wichtig. Weitere Informationen zur Speicherverwaltung können in Büchern zur Rechnerarchitektur (siehe z.B. Hennessy et al. [212]) nachgelesen werden

Literaturverzeichnis

1. Aamodt, T., Chow, P.: Embedded ISA support for enhanced floating-point to fixed-point ANSI C compilation. Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES) pp. 128–137 (2000)
2. Abella, J., Hardy, D., Puaut, I., Quiñones, E., Cazorla, F.J.: On the comparison of deterministic and probabilistic WCET estimation techniques. In: Euromicro Conference on Real-Time Systems (ECRTS), pp. 266–275 (2014). DOI 10.1109/ECRTS.2014.16
3. Abeni, L., Buttazzo, G.: Integrating multimedia applications in hard real-time systems. In: Proceedings of the Real-Time Systems Symposium (RTSS), pp. 4–13 (1998)
4. Absint: The industry standard for static timing analysis. http://www.absint.com/ait (2021)
5. AbsInt Angewandte Informatik GmbH: Stack overflow is a thing of the past. https://www.absint.com/stackanalyzer/index.htm (2016)
6. acatech: Cyber-Physical Systems. Innovationsmotor für Mobilität, Gesundheit, Energie und Produktion. https://www.acatech.de/publikation/cyber-physical-systems/ (2011)
7. Accellera Systems Initiative™: Core SystemC Language and Examples. http://accellera.org/downloads/standards/systemc/files (2014)
8. Accellera Systems Initiative™: SystemC Synthesizable Subset – Version 1.4.7. http://accellera.org/images/downloads/standards/systemc/SystemC_Synthesis_Subset_1_4_7-Apache.pdf (2016)
9. ACM SIGBED: Home page. http://www.sigbed.org (2020)
10. ACM/IEEE: Computer science curricula 2013: Curriculum guidelines for undergraduate degree programs in computer science. The Joint Task Force on Computing Curricula Association for Computing Machinery (ACM) IEEE Computer Society, https://computingcurricula.com/files/CS2013.pdf (Dec. 2013)
11. Ahmad, I., Ranka, S.: Handbook of Energy-Aware and Green Computing – Two Volume Set. CRC Press (2016)
12. aicas: Realtime Specification for Java 2.0. https://www.aicas.com/cms/en/rtsj (2016)
13. Ambler, S.: Introduction to the diagrams of UML 2.X. http://www.agilemodeling.com/essays/umlDiagrams.htm (2020)
14. Analog Devices Inc. Eng.: ADSP-2100 Family User's Manual. Out of print (1995)
15. Analog Devices Inc. Eng.: Data Conversion Handbook (Analog Devices). Newnes (2004)
16. Andersson, B., Baruah, S., Jonsson, J.: Static-priority scheduling on multiprocessors. In: Proceedings of the Real-Time Systems Symposium (RTSS), pp. 193–202 (2001)
17. Anonymous: Introduction to lp_solve 5.5.2.5. http://lpsolve.sourceforge.net (2021)
18. ANSYS: Embedded software - embedded systems and software development. https://www.ansys.com/products/embedded-software (2021)
19. ARM Ltd.: #pragma arm section [section_type_list]. http://www.keil.com/support/man/docs/armcc/armcc_chr1359124985290.htm (2019)

© Der/die Autor(en) 2021

P. Marwedel, *Eingebettete Systeme*, https://doi.org/10.1007/978-3-658-33437-6

20. ARM Ltd.: AMBA specifications. http://www.arm.com/products/system-ip/amba-specifications.php (2021)
21. ARM Ltd.: big.LITTLE Technology. http://www.arm.com/products/processors/technologies/biglittleprocessing.php (2021)
22. ARM Ltd.: Mali-T860 and Mali-T880 GPUs. https://developer.arm.com/ip-products/graphics-and-multimedia/mali-gpus/mali-t860-and-mali-t880-gpus (2021)
23. Arnaud, F., Colquhoun, S., Mareau, A., Kohler, S., Jeannot, S., Hasbani, F., Paulin, R., Cremer, S., Charbuillet, C., Druais, G., Scheer, P.: Technology-Circuit Convergence for Full-SOC Platform in 28 nm and Beyond. International Electron Devices Meeting (2011)
24. Artist Consortium: Home page. http://www.artist-embedded.org (2009)
25. Atienza, D., Baloukas, C., Papadopoulos, L., Poucet, C., Mamagkakis, S., Hidalgo, J.I., Catthoor, F., Soudris, D., Lanchares, J.: Optimization of dynamic data structures in multimedia embedded systems using evolutionary computation. In: Proceedings of the International Workshop on Software and Compilers for Embedded Systems (SCOPES), pp. 31–40 (2007). DOI http://doi.acm.org/10.1145/1269843.1269849
26. Atmel: 32-Bit AVR Microcontroller. http://www.atmel.com/Images/doc32058.pdf (2012)
27. AUTOSAR: The standardized software framework for intelligent mobility. http://www.autosar.org (2021)
28. Avižienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing **1**(1), 11–33 (2004)
29. Azevedo, A., Issenin, I., Cornea, R., Gupta, R., Dutt, N., Veidenbaum, A., Nicolau, A.: Profile-based dynamic voltage scheduling using program checkpoints. Proceedings of Design, Automation and Test in Europe (DATE) pp. 168–175 (2002)
30. Bäck, T., Fogel, D., Michalewicz, Z.: Handbook of Evolutionary Computation. Oxford Univ. Press (1997)
31. Bäck, T., Schwefel, H.P.: An overview of evolutionary algorithms for parameter optimization. Evolutionary computation pp. 1–23 (1993)
32. Bai, K., Shrivastava, A.: Heap data management for limited local memory (LLM) multi-core processors. In: Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp. 317–325 (2010)
33. Baker, T.: Rate monotone scheduling. http://www.cs.fsu.edu/~baker/realtime/restricted/notes/rmscheduling.html (2008)
34. Baker, T.P.: Stack-based scheduling of real-time processes. Journal of Real-Time Systems **3** (1991)
35. Banakar, R., Steinke, S., Lee, B.S., Balakrishnan, M., Marwedel, P.: Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. Proceedings of the International Symposium on Hardware-Software Codesign (CODES) pp. 73–78 (2002). DOI http://doi.acm.org/10.1145/774789.774805
36. Barney, B.: POSIX threads programming. https://computing.llnl.gov/tutorials/pthreads (2021)
37. Barrett, S., Pack, D.: Embedded Systems – Design and Applications with the 68HC12 and HCS12. Prentice Hall (2005)
38. Baruah, S., Bertogna, M., Buttazzo, G.: Multiprocessor Scheduling for Real-Time Systems. Springer (2015)
39. Baruah, S.K., Cohen, N.K., Plaxton, C.G., Varvel, D.A.: Proportionate progress: A notion of fairness in resource allocation. Algorithmica **15**(6), 600–625 (1996). DOI 10.1007/BF01940883. URL http://dx.doi.org/10.1007/BF01940883
40. Baruah, S.K., Mok, A.K., Rosier, L.E.: Preemptively scheduling hard-real-time sporadic tasks on one processor. In: Proceedings of the Real-Time Systems Symposium (RTSS), pp. 182–190 (1990). DOI 10.1109/REAL.1990.128746
41. Basten, T.: Opening remarks, 2nd Artist workshop on models of computation and communication. Eindhoven, http://www.es.ele.tue.nl/~tbasten/mocc2008/presentations/mocc.pdf (2008)

42. Benavides, T., Treon, J., Hulbert, J., Chang, W.: The enabling of an execute-in-place architecture to reduce the embedded system memory footprint and boot time. JCP **3**(1), 79–89 (2008). DOI 10.4304/jcp.3.1.79-89. URL http://dx.doi.org/10.4304/jcp.3.1.79-89
43. Bender, A.: MILP based task mapping for heterogeneous multiprocessor systems. In: Design Automation Conference, 1996, with EURO-VHDL '96 and Exhibition, Proceedings EURO-DAC '96, European, pp. 190–197 (1996). DOI 10.1109/EURDAC.1996.558204
44. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. In: J. Desel, W. Reisig and G. Rozenberg (eds.): ACPN 2003, Springer LNCS **3098**, 87–124 (2004)
45. Benini, L., Bogliolo, A., De Micheli, G.: A survey of design techniques for system-level dynamic power management. IEEE Transactions on VLSI Systems **8**(3), 299–316 (2000)
46. Benini, L., De Micheli, G.: Dynamic Power Management – Design Techniques and CAD Tools. Kluwer Academic Publishers (1998)
47. van Berkel, C.H.K.: Multi-core for mobile phones. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE '09, pp. 1260–1265. European Design and Automation Association, 3001 Leuven, Belgium, Belgium (2009). URL http://dl.acm.org/citation.cfm?id=1874620.1874924
48. Bernardi, P., Rebaudengo, M. Reorda, S.: Using infrastructure IPs to support SW-based self-test of processor cores. Workshop on Fibres and Optical Passive Components pp. 22–27 (2005)
49. Bernstein, J.B., Gurfinkel, M., Li, X., Walters, J., Shapira, Y., Talmor, M.: Electronic circuit reliability modeling. Microelectronics Reliability **46**(12), 1957 – 1979 (2006). DOI http://dx.doi.org/10.1016/j.microrel.2005.12.004. URL http://www.sciencedirect.com/science/article/pii/S0026271406000023
50. Bertogna, M., Cirinei, M., Lipari, G.: Improved schedulability analysis of EDF on multiprocessor platforms. In: Euromicro Conference on Real-Time Systems (ECRTS), pp. 209–218 (2005). DOI 10.1109/ECRTS.2005.18
51. Bertolotti, I.C.: Real-time embedded operating systems: Standards and perspectives. In: R. Zurawski (ed.): Embedded Systems Handbook, CRC Press (2006)
52. Beszedes, A.: Survey of code size reduction methods. ACM Computing Surveys pp. 223–267 (2003)
53. Bieker, U., Marwedel, P.: Retargetable self-test program generation using constraint logic programming. Proceedings of the Design Automation Conference (DAC) pp. 605–611 (1995)
54. Bini, E., Buttazzo, G., Buttazzo, G.: A hyperbolic bound for the rate monotonic algorithm. Euromicro Conference on Real-Time Systems (ECRTS) pp. 59–73 (2001)
55. Black, J.R.: Electromigration failure modes in aluminum metallization for semiconductor devices. Proceedings of the IEEE **57**(9), 1587–1594 (1969). DOI 10.1109/PROC.1969.7340
56. Boldt, M., Traulsen, C., von Hanxleden, R.: Compilation and Worst-case Reaction Time Analysis for Multithreaded Esterel Processing. EURASIP J. Embedded Syst. **2008**, 4:1–4:21 (2008). DOI 10.1155/2008/594129. URL http://dx.doi.org/10.1155/2008/594129
57. Bonfietti, A., Benini, L., Lombardi, M., Milano, M.: An efficient and complete approach for throughput-maximal SDF allocation and scheduling on multi-core platforms. In: Proceedings of Design, Automation and Test in Europe (DATE), pp. 897–902. European Design and Automation Association, 3001 Leuven, Belgium, Belgium (2010). URL http://dl.acm.org/citation.cfm?id=1870926.1871143
58. Bonny, T., Henkel, J.: Huffman-based code compression techniques for embedded processors. ACM Trans. Des. Autom. Electron. Syst. **15**(4), 31:1–31:37 (2010). DOI 10.1145/1835420.1835424. URL http://doi.acm.org/10.1145/1835420.1835424
59. Bordoloi, U.: Scheduling with shared resources. Ursprüngliche Adresse: http://www.ida.liu.se/~unmbo/RTS_CUGS_files/Lecture3.pdf, z.Zt. (2021) verfügbar unter https://tams.informatik.uni-hamburg.de/lehre/2016ss/vorlesung/es/doc/ida.liu.se-rts-Lecture3.pdf (2020)
60. Boulanger, J.L.: CENELEC 50128 and IEC 62279 Standards. John Wiley & Sons (2015)
61. Boussinot, F., de Simone, R.: The Esterel language. Proc. of the IEEE, Vol. 79, No. 9 pp. 1293–1304 (1991)

62. Bouwmeester, D., Ekert, A., (eds.), A.Z.: The Physics of Quantum Information: Quantum Cryptography, Quantum Teleportation, Quantum Computation. Springer (2000)
63. Bouyssounouse, B., Sifakis, J. (eds.): Embedded Systems Design, The ARTIST Roadmap for Research and Development. Lecture Notes in Computer Science, Vol. 3436, Springer (2005)
64. Brahme, D., Abraham, J.A.: Functional testing of microprocessors. IEEE Trans. on Computers pp. 475–485 (1984)
65. Brand, D., Bergamaschi, R.A., Stok, L.: Don't cares in synthesis: theoretical pitfalls and practical solutions. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **17**(4), 285–304 (1998). DOI 10.1109/43.703819
66. Braun, A., Bringmann, O., Lettnin, D., Rosenstiel, W.: Simulation-based verification of the MOST netinterface specification revision 3.0. Proceedings of Design, Automation and Test in Europe (DATE) (2010)
67. Bremaud, P.: Markov Chains. Springer (1999)
68. Brettel, M., Friederichsen, N., Keller, M., Rosenberg, M.: How virtualization, decentralization and network building change the manufacturing landscape: An industry 4.0 perspective. International Journal of Mechanical, Industrial Science and Engineering **8**(1), 37–44 (2014)
69. Bril, R.J.: Real-time scheduling for media processing using conditionally guaranteed budgets. PhD thesis, TU Eindhoven (2004)
70. Brooks, D., Tiwari, V., Martonosi, M.: Wattch: a framework for architectural-level power analysis and optimizations. 27th Int. Symp. on Computer Architecture (ISCA) pp. 83–94 (2000)
71. Bruno, E., Bollella, G.: Real-Time Java Programming: With Java RTS. Prentice Hall (2009)
72. Bryant, R.: A switch-level model and simulator for MOS digital circuits. IEEE Trans. on Computers, Vol. 33 (1984)
73. Buck, J.T.: Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D. thesis, University of California at Berkeley (1993)
74. Budkowski, S., Dembinski, P.: An introduction to Estelle: A specification language for distributed systems. Computer Networks and ISDN Systems **14**(1), 3 – 23 (1987). DOI DOI:10.1016/0169-7552(87)90084-5. URL https://www.sciencedirect.com/science/article/pii/0169755287900845?via%3Dihub
75. Bundesamt für Sicherheit in der Informationstechnik: IT-Grundschutz – Glossar. https://www.bsi.bund.de/DE/Themen/ITGrundschutz/ITGrundschutzKompendium/vorkapitel/Glossar_.html (2021)
76. Burd, T., Brodersen, R.: Design issues for dynamic voltage scaling. Int. Symp. on Low Power Electronics and Design (ISLPED) pp. 9–14 (2000)
77. Burd, T., Brodersen, R.W.: Energy efficient microprocessor design. Kluwer Academic Publishers (2003)
78. Burks, A., Goldstine, H., von Neumann, J.: Preliminary discussion of the logical design of an electronic computing element. Report to U.S. Army Ordnance Department, reprinted at https://www.cs.princeton.edu/courses/archive/fall10/cos375/Burks.pdf (1946)
79. Burns, A., Wellings, A.: Real-Time Systems and Programming Languages. Addison-Wesley (1990)
80. Burns, A., Wellings, A.: Real-Time Systems and Programming Languages (Fourth Edition). Addison Wesley (2009)
81. Buttazzo, G.: Hard Real-time Computing Systems. Springer, 4th printing (2011)
82. Cadence Design Systems Inc.: Tensilica Customizable Processor IP. http://ip.cadence.com/ipportfolio/tensilica-ip (2021)
83. Cai, L., Gajski, D.: Transaction level modeling: An overview. In: Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp. 19–24. ACM, New York, NY, USA (2003). DOI 10.1145/944645.944651. URL http://doi.acm.org/10.1145/944645.944651
84. Carroll, A., Heiser, G.: An analysis of power consumption in a smartphone. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC'10, pp. 21–21. USENIX Association, Berkeley, CA, USA (2010). URL http://dl.acm.org/citation.cfm?id=1855840.1855861

85. Caspi, P., Sangiovanni-Vincentelli, A., Almeida, L., et al: Guidelines for a graduate curriculum on embedded software and systems. ACM Transactions on Embedded Computing Systems pp. 587–611 (2005)
86. Castrillon, J., Tretter, A., Leupers, R., Ascheid, G.: Communication-aware mapping of KPN applications onto heterogeneous MPSoCs. In: Proceedings of the Design Automation Conference (DAC), pp. 1266–1271. ACM, New York, NY, USA (2012). DOI 10.1145/2228360.2228597. URL http://doi.acm.org/10.1145/2228360.2228597
87. Cederqvist, P.: The CVS manual - version management with CVS. Network Theory Ltd. (2006)
88. Ceng, J., Castrillón, J., Sheng, W., Scharwächter, H., Leupers, R., Ascheid, G., Meyr, H., Isshiki, T., Kunieda, H.: MAPS: an integrated framework for MPSoC application parallelization. Proceedings of the Design Automation Conference (DAC) pp. 754–759 (2008)
89. Chamberlain, R., Taha, W., Törngren, M. (eds.): Cyber Physical Systems. Model-Based Design – 2019 8th International Workshop, CyPhy 2018, and 14th International Workshop, WESE 2018, Revised Selected Papers. Lecture Notes in Computer Science (2019)
90. Chandrakasan, A.P., Sheng, S., Brodersen, R.W.: Low-power CMOS digital design. IEEE Journal of Solid-State Circuits **27**(4), 119–123 (1992)
91. Chanet, D., Sutter, B.D., Bus, B.D., Put, L.V., Bosschere, K.D.: Automated reduction of the memory footprint of the Linux kernel. ACM Trans. Embed. Comput. Syst. **6**(4), 23 (2007). DOI http://doi.acm.org/10.1145/1274858.1274861
92. Chang, D.W., Lin, I.C., Chien, Y.S., Lin, C.L., Su, A.W.Y., Young, C.P.: CASA: Contention-Aware Scratchpad Memory Allocation for Online Hybrid On-Chip Memory Management. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **33**(12), 1806–1817 (2014). DOI 10.1109/TCAD.2014.2363385
93. Chao, C., Saeta, B.: HotChips 2019 Tutorial – Cloud TPU: Codesigning Architecture and Infrastructure. https://www.hotchips.org/hc31/HC31_T3_Cloud_TPU_Codesign.pdf (2019)
94. Chen, J., Huang, W., Liu, C.: k2Q: A Quadratic-Form Response Time and Schedulability Analysis Framework for Utilization-Based Analysis. CoRR, http://arxiv.org/abs/1505.03883 (2015)
95. Chen, K., Sztipanovits, J., Neema, S.: Compositional specification of behavioral semantics. Design, Automation and Test in Europe (DATE) pp. 906–911 (2007)
96. Chen, M., Rincón-Mora, G.: Accurate electrical battery model capable of predicting runtime and i-v performance. IEEE Trans. on Energy Conversion pp. 504–511 (2006)
97. Chen, X., Dick, R., Shang, L.: Properties of and improvements to time-domain dynamic thermal analysis algorithms. Proceedings of Design, Automation and Test in Europe (DATE) (2010)
98. Chetto, H., Silly, M., Bouchentouf, T.: Dynamic scheduling of real-time tasks under precedence constraints. Journal of Real-Time Systems, 2 (1990)
99. Cheung, E., Hsieh, H., Balarin, F.: Automatic buffer sizing for rate-constrained KPN applications on multiprocessor system-on-chip. In: High Level Design Validation and Test Workshop, 2007. HLVDT 2007. IEEE International, pp. 37–44 (2007). DOI 10.1109/HLDVT.2007.4392782
100. Chiu, Y.: Folding and Interpolating ADC. University of Texas at Dallas, EECT 7327, http://www.utdallas.edu/~yxc101000/courses/7327/slides/intp%20folding%20adc.pptx (2014)
101. Cho, H., Ravindran, B., Jensen, E.: An optimal real-time scheduling algorithm for multiprocessors. Proceedings of the Real-Time Systems Symposium (RTSS) (2006)
102. Cho, S., Lee, S.K., Han, A., Lin, K.J.: Efficient real-time scheduling algorithms for multiprocessor systems. IEICE Trans. Communications pp. 2859–2867 (2002)
103. Chung, E.Y., Benini, L., De Micheli, G.: Source code transformation based on software cost analysis. Proceedings of the International Symposium on System Synthesis (ISSS) pp. 153–158 (2001)
104. Clarke, E.M., Grumberg, O., Hiraishi, H., Jha, S., Long, D.E., McMillan, K.L., Ness, L.A.: Verification of the Futurebus+ cache coherence protocol. Formal Methods in System Design **6**(2), 217–232 (2005)

105. Clouard, A., Jain, K., Ghenassia, F., Maillet-Contoz, L., Strassen, J.: Using transactional models in SoC design flow. In: [408] pp. 29–64 (2003)
106. Coelho, D.R.: The VHDL handbook. Kluwer Academic Publishers (1989)
107. Coello, C.A.C., Lamont, G.B., Veldhuizen, D.A.v.: Evolutionary Algorithms for Solving Multi-Objective Problems. Springer (2007)
108. Collins-Sussman, B., Fitzpatrick, B., Pilato, C.: Version control with subversion – for subversion 1.5. http://svnbook.red-bean.com/en/1.5/svn-book.pdf (2008)
109. Cooling, J.: Software Engineering for Real-Time Systems. Addison Wesley (2003)
110. Cordes, D.A.: Automatic parallelization for embedded multi-core systems using high-level cost models. Ph.D. thesis, TU Dortmund, Department of Computer Science (2013)
111. Cortadella, J., Kondratyev, A., Lavagno, L., Massot, M., Moral, S., Passerone, C., Watanabe, Y., Sangiovanni-Vincentelli, A.: Task generation and compile-time scheduling for mixed data-control embedded software. Proceedings of the Design Automation Conference (DAC) pp. 489–494 (2000)
112. Coskun, A.K., Rosing, T.S., Whisnant, K.A., Gross, K.C.: Temperature-aware MPSoC scheduling for reducing hot spots and gradients. In: Proceedings of the 2008 Asia and South Pacific Design Automation Conference, ASP-DAC '08, pp. 49–54. IEEE Computer Society Press, Los Alamitos, CA, USA (2008). URL http://dl.acm.org/citation.cfm?id=1356802.1356815
113. Coussy, P., Morawiec, A.: High-Level Synthesis – From Algorithm to Digital Circuit. Springer (2008)
114. Craig, I.D.: Virtual Machines. Springer (2006)
115. Cyber-Physical Systems Virtual Organization: Home page. https://cps-vo.org/ (1920)
116. Cyber-Physical Systems Virtual Organization: Cyber-physical systems (CPS) – program solicitation. https://cps-vo.org/node/59030 (2019)
117. D. L. Oliveira L. A. Faria, H.A.D., Garcia, K.: An architecture for globally-synchronous locally-asynchronous systems on FPGAs. IEEE XXIII International Congress on Electronics, Electrical Engineering and Computing (INTERCON) pp. 1–6 (2016)
118. Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. Formal Methods in System Design (2001)
119. Dasgupta, S.: The organization of microprogram stores. ACM Computing Surveys, Vol. 11 pp. 39–65 (1979)
120. Davis, J., Hylands, C., Janneck, J., Lee, E.A., Liu, J., Liu, X., Neuendorffer, S., Sachs, S., Stewart, M., Vissers, K., Whitaker, P., Xiong, Y.: Overview of the Ptolemy project. Technical Memorandum UCB/ERL M01/11; http://ptolemy.eecs.berkeley.edu (2001)
121. Davis, R.I., Burns, A.: A survey of hard real-time scheduling for multiprocessor systems. ACM Comput. Surv. **43**(4), 35:1–35:44 (2011). DOI 10.1145/1978802.1978814. URL http://doi.acm.org/10.1145/1978802.1978814
122. De Greef, E., Catthoor, F., Man, H.: Memory size reduction through storage order optimization for embedded parallel multimedia applications. Proc. Workshop on Parallel Processing and Multimedia pp. 84–98 (1997)
123. De Greef, E., Catthoor, F., Man, H.D.: Array placement for storage size reduction in embedded multimedia systems. IEEE Int. Conf. on Application-Specific Systems, Architectures and Processors (ASAP) pp. 66–75 (1997)
124. De Micheli, G., Ernst, R., Wolf, W.: Readings in Hardware/Software Co-Design. Academic Press (2002)
125. Derin, O.: Self-adaptivity of applications on network on chip multiprocessors – the case of fault-tolerant Kahn Process Networks. Ph.D. thesis, Università della Svizzera Italiana de Lugano (2013)
126. Derler, P., Lee, E.A., Vincentelli, A.S.: Modeling cyber-physical systems. Proceedings of the IEEE **100**(1), 13–28 (2012). DOI 10.1109/JPROC.2011.2160929
127. Deutsches Institut für Normung: DIN 66253, Programmiersprache PEARL, Teil 2 PEARL 90. Beuth-Verlag; http://www.din.de (1997)
128. Devi, U.M.C., Anderson, J.H.: Tardiness bounds under global EDF scheduling on a multiprocessor. In: Proceedings of the Real-Time Systems Symposium (RTSS), pp. 12 pp.–341 (2005). DOI 10.1109/RTSS.2005.39

129. Devillers, R.R., Goossens, J.: Liu and Layland's schedulability test revisited. Inf. Process. Letters pp. 157–161 (2000)
130. Dhall, K., Liu, C.: On a real-time scheduling problem. Operations Research **26**(1), 127–140 (1978). DOI 10.1287/opre.26.1.127. URL http://dx.doi.org/10.1287/opre.26.1.127
131. Dibble, P.C.: Real-Time Java Platform Programming: Second Edition. BookSurge Publishing (2008)
132. Diederichs, C., Margull, U., Slomka, F., Wirrer, G.: An application-based EDF scheduler for OSEK/VDX. Proceedings of Design, Automation and Test in Europe (DATE) pp. 1045–1050 (2008)
133. Dill, D., Alur, R.: A theory of timed automata. Theoretical Computer Science pp. 183–235 (1994)
134. Dominguez, A., Udayakumaran, S., Barua, R.: Heap Data Allocation to Scratch-Pad Memory in Embedded Systems. Journal of Embedded Computing **1**(4), 521–540 (2005)
135. Donald, J., Martonosi, M.: Techniques for multicore thermal management: Classification and new exploration. SIGARCH Comput. Archit. News **34**(2), 78–88 (2006). DOI http://doi.acm.org/10.1145/1150019.1136493
136. Dósa, G.: The Tight Bound of First Fit Decreasing Bin-Packing Algorithm Is FFD(I) ? 11/9OPT(I) + 6/9, pp. 1–11. Springer, Berlin, Heidelberg (2007). DOI 10.1007/978-3-540-74450-4_1. URL http://dx.doi.org/10.1007/978-3-540-74450-4_1
137. Douglass, B.P.: Real-Time UML, 3rd edition. Addison Wesley (2004)
138. Dozio, L., Mantegazza, P.: Real time distributed control systems using RTAI. In: Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (2003)
139. Drepper, U.: What every programmer should know about memory. http://www.akkadia.org/drepper/cpumemory.pdf (2007)
140. Dressler, F.: Cyber physical social systems: Towards deeply integrated hybridized systems. International Conference on Computing, Networking and Communications (ICNC) (2018)
141. Drusinsky, D., Harel, D.: Using statecharts for hardware description and synthesis. IEEE Trans. on Computer Design pp. 798–807 (1989)
142. Dunkels, A.: Design and Implementation of the lwIP TCP/IP Stack. Swedish Institute of Computer Science **2**, 77 (2001)
143. Dunn, W.: Practical Design of Safety-Critical Computer Systems. Reliability Press (2002)
144. Dusza, B., Marwedel, P., Spinczyk, O., Wietfeld, C.: A context-aware battery lifetime model for carrier aggregation enabled LTE-A systems. IEEE Consumer Communications and Networking Conference (CCNC) (2014)
145. Ecker, W., Müller, W., Dömer, R.: Hardware-dependent software - Principles and practice. Springer (2009)
146. Edwards, S.: Dataflow languages. http://www.cs.columbia.edu/~sedwards/classes/2001/w4995-02/presentations/dataflow.ppt (2001)
147. Edwards, S.: Languages for embedded systems. In: R. Zurawski (ed.): Embedded Systems Handbook, CRC Press (2006)
148. efunda: Materials home. http://www.efunda.com/materials/elements/element_info.cfm?Element_ID=Si (2021)
149. Egger, B., Lee, J., Shin, H.: Scratchpad memory management for portable systems with a memory management unit. Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES) pp. 321–330 (2006)
150. Eggermont, L.: Embedded systems roadmap. STW, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.119.6407&rep=rep1&type=pdf (2002)
151. Elsevier B.V.: Sensors and Actuators A: Physical. An International Journal (2021)
152. Elsevier B.V.: Sensors and Actuators B: Chemical. An International Journal (2021)
153. Esmaeilzadeh, H., Blem, E., Amant, R.S., Sankaralingam, K., Burger, D.: Dark silicon and the end of multicore scaling. In: 2011 38th Annual International Symposium on Computer Architecture (ISCA), pp. 365–376 (2011)
154. Esterel Technologies SA: Homepage. http://www.esterel-technologies.com (2021)

155. European Commission: Topic: Smart cyber-physical systems. http://ec.europa.eu/research/participants/portal/desktop/en/opportunities/h2020/topics/ict-01-2014.html (2013)
156. European Commission: Computing technologies and engineering methods for cyber-physical systems of systems. https://ec.europa.eu/info/funding-tenders/opportunities/portal/screen/opportunities/topic-details/ict-01-2019 (2019)
157. Evidence: Erika enterprise. http://erika.tuxfamily.org/drupal/ (2019)
158. Falk, H.: WCET-aware register allocation based on graph coloring. Proceedings of the Design Automation Conference (DAC) pp. 726–731 (2009)
159. Falk, H., Marwedel, P.: Control flow driven splitting of loop nests at the source code level. Proceedings of Design, Automation and Test in Europe (DATE) pp. 410–415 (2003)
160. Falk, H., Verma, M.: Combined Data Partitioning and Loop Nest Splitting for Energy Consumption Minimization. In: Proceedings of the International Workshop on Software and Compilers for Embedded Systems (SCOPES), pp. 137–151. Amsterdam/The Netherlands (2004)
161. Falkenberg, R., Sliwa, B., Piatkowski, N., Wietfeld, C.: Machine Learning Based Uplink Transmission Power Prediction for LTE and Upcoming 5G Networks Using Passive Downlink Indicators. 2018 IEEE 88th Vehicular Technology Conference (VTC-Fall) pp. 1–7 (2018)
162. Fard, H.M., Prodan, R., Barrionuevo, J.J.D., Fahringer, T.: A multi-objective approach for workflow scheduling in heterogeneous environments. In: Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012), CCGRID ’12, pp. 300–309. IEEE Computer Society, Washington, DC, USA (2012). DOI 10.1109/CCGrid.2012.114. URL http://dx.doi.org/10.1109/CCGrid.2012.114
163. Ferrell, T.K., Ferrell, U.D.: RTCA DO-178-B/EUROCAE ED-12B. in: C. Spitzer (ed.): The Avionics Handbook (Electrical Engineering Handbook), Chapter 27, CRC Press (2001)
164. Fiorin, L., Palermo, G., Lukovic, S., Silvano, C.: A data protection unit for NoC-based architectures. In: Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp. 167–172 (2007)
165. Flautner, K.: Heterogeneity to the rescue. https://www.bscmsrc.eu/sites/default/files/media/arm-heterogenous-mp-november-2011.pdf (2011)
166. Fowler, M., Scott, K.: UML Distilled - Applying the Standard Object Modeling Language. Addison-Wesley (1998)
167. Franke, B.: Fast cycle-approximate instruction set simulation. In: Proceedings of the International Workshop on Software and Compilers for Embedded Systems (SCOPES), pp. 69–78 (2008)
168. Franke, B., O’Boyle, M.F.: A complete compiler approach to auto-parallelizing C programs for multi-DSP systems. IEEE Transactions on Parallel and Distributed Systems **16**, 234–245 (2005)
169. Franke, H., et al.: Lexikon der Physik, Stichwort Wärmeleitvermögen. Deutscher Taschenbuch Verlag (1971)
170. Freescale Semiconductor/NXP: ColdFire® Family Programmer’s Reference Manual. http://www.nxp.com/assets/documents/data/en/reference-manuals/CFPRM.pdf (2005)
171. Gajski, D., Kuhn, R.: New VLSI tools. IEEE Computer pp. 11–14 (1983)
172. Gajski, D., Vahid, F., Narayan, S., Gong, J.: Specification and Design of Embedded Systems. Prentice Hall (1994)
173. Gajski, D., Zhu, J., Dömer, R., Gerstlauer, A., Zhao, S.: SpecC: Specification Language Methodology. Kluwer Academic Publishers (2000)
174. Gajski, D.D., Abdi, S., Gerstlauer, A., Schirner, G.: Embedded System Design. Springer, Heidelberg (2009)
175. Ganssle, J. (ed.): Embedded Systems (World Class Designs). Newnes (2008)
176. Ganssle, J.G.: The Art of Designing Embedded Systems. Newnes (2000)
177. Ganssle, J.G., Noergaard, T., Eady, F., Edwards, L., Katz, D.J., Gentile, R., Arnold, K., Hyder, K., Perrin, B.: Embedded Hardware - Know it all. Newnes (2008)
178. Garey, M.R., Johnson, D.S.: Computers and Intractability. Bell Labaratories, Murray Hill, New Jersey (1979)

179. Garg, R., Khatri, S.: Analysis and Design of Resilient VLSI Circuits. Springer (2009)
180. Gebotys, C.: Security in Embedded Devices. Springer (2010)
181. Geffroy, J.C., Motet, G.: Design of Dependable Computing Systems. Kluwer Academic Publishers (2002)
182. Gerum, P.: Xenomai – implementing a RTOS emulation framework on GNU Linux. Xenomai White Paper (2004)
183. Gierlichs, B., Poschmann, A.: Int. workshop on cryptographic hardware and embedded systems (CHES). https://ches.iacr.org/2018/ (2016)
184. Girard, A., Pappas, G.J.: Approximation metrics for discrete and continuous systems. IEEE Transactions on Automatic Control **52**(5), 782–798 (2007). DOI 10.1109/TAC.2007.895849
185. Giusto, D., Iera, A., Morabito, G., Atzori, L. (eds.): The Internet of Things, 20th Tyrrhenian Workshop on Digital Communications. Springer (2010)
186. Goldberg, D.: What every computer scientist should know about floating point arithmetic. ACM Computing Surveys **23**, 5–48 (1991)
187. Gomez, L., Fernandes, J.: Behavioral Modeling for Embedded Systems and Technologies. IGI Global (2010)
188. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016)
189. Goossens, J., Funk, S., Baruah, S.: EDF scheduling on multiprocessor platforms: some (perhaps) counterintuitive observations. In: RealTime Computing Systems and Applications Symposium (2002)
190. Graham, R., Lawler, E., Lenstra, J.K., Kan, A.H.G.: Optimization and approximation in deterministic sequencing and scheduling: A survey. Annals of Discrete Mathematics pp. 287–326 (1979)
191. Grötker, T., Liao, S., Martin, G.: System design with SystemC. Springer (2002)
192. Gubbia, J., Buyyab, R., Marusica, S., Palaniswamia, M.: Internet of Things (IoT): A vision, architectural elements, and future directions. Future Generation Computer Systems **29**, 1645–1660 (2013)
193. Guerrieri, A., Loscri, V., Rovella, A., Fortino, G.: Management of Cyber Physical Objects in the Future Internet of Things: Methods, Architectures and Applications. Springer (2016)
194. Gupta, R.: Tasks and task management. Course ICS 212, Winter 2002, UC Irvine, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.7.8704&rep=rep1&type=pdf (2002)
195. Ha, S.: Model-based programming environment of embedded software for MPSoC. Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC) pp. 330–335 (2007). DOI http://dx.doi.org/10.1109/ASPDAC.2007.358007
196. de Haan, L., Ferreira, A.: Extreme Value Theory – An Introduction. Springer (2006)
197. Hahn, S., Reineke, J., Wilhelm, R.: Toward compact abstractions for processor pipelines. Correct System Design pp. 205–220 (2015)
198. Halbwachs, N.: Synchronous programming of reactive systems, a tutorial and commented bibliography. Tenth International Conference on Computer-Aided Verification, CAV'98, LNCS 1427, Springer Verlag; see also: https://link.springer.com/chapter/10.1007/BFb0028726 (1998)
199. Halbwachs, N.: Personal communication. South American Artist School on Embedded Systems, Florianopolis (2008)
200. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow language LUSTRE. Proc. of the IEEE Trans. on Software Engineering **79**, 1305–1320 (1991)
201. Hamada, T., Benkrid, K., Nitadori, K., Taiji, M.: A Comparative Study on ASIC, FPGAs, GPUs and General Purpose Processors in the O(N^2) Gravitational N-body Simulation. In: Proceedings of the 2009 NASA/ESA Conference on Adaptive Hardware and Systems, AHS '09, pp. 447–452. IEEE Computer Society, Washington, DC, USA (2009). DOI 10.1109/AHS.2009.55. URL http://dx.doi.org/10.1109/AHS.2009.55
202. Harbour, M.G.: RT-POSIX: An overview. http://www.ctr.unican.es/publications/mgh-1993a.pdf (1993)
203. Hardkernel co, Ltd.: Odroid-XU3. https://www.hardkernel.com/shop/odroid-xu3/ (2019)
204. Harel, D.: StateCharts: A visual formalism for complex systems. Science of Computer Programming pp. 231–274 (1987)

205. Hastie, T., Tibshirani, R., Friedman, R.: The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer (2008)
206. Hattori, T.: MPSoC approaches for low-power embedded SoC's. https://mpsoc-forum.org/archive/2007/slides/Hattori.pdf (2007)
207. Haubelt, C., Teich, J.: Digitale Hardware/Software-Systeme: Spezifikation und Verifikation. eXamen.press. Springer Berlin Heidelberg (2010). URL https://books.google.de/books?id=eNwfBAAAQBAJ
208. Haugen, O., Moller-Pedersen, B.: Introduction to UML and the modeling of embedded systems. In: R. Zurawski (ed.): Embedded Systems Handbook, CRC Press (2006)
209. Hayes, J.: A unified switching theory with applications to VLSI design. Proceedings of the IEEE, Vol.70 pp. 1140–1151 (1982)
210. Heemels, W., Johansson, K., Tabuada, P.: An introduction to event-triggered and self-triggered control. In: Decision and Control (CDC), 2012 IEEE 51st Annual Conference on, pp. 3270–3285 (2012). DOI 10.1109/CDC.2012.6425820
211. Henia, R., Hamann, A., Jersak, M., Racu, R., Richter, K., Ernst, R.: System level performance analysis - the SymTA/S approach. IEEE Computers and Digital Techniques pp. 148–166 (2005)
212. Hennessy, J.L., Patterson, D.A.: Computer Architecture – A Quantitative Approach, 5 edn. Morgan Kaufmann Publishers Inc. (2011)
213. Hennessy, J.L., Patterson, D.A.: Computer Organization – The Hardware/Software Interface. Morgan Kaufmann Publishers Inc. (2013)
214. Henzinger, T., Sifakis, J.: The embedded systems design challenge. FM 2006: Formal Methods, Lecture Notes in Computer Science, Vol. 4085 pp. 1–15 (2006)
215. Herken, R.: The Universal Turing Machine: A half-century survey. Springer (1995)
216. Herrera, F., Fernández, V., Sánchez, P., Villar, E.: Embedded software generation from SystemC for platform based design. In: [408] pp. 247–272 (2003)
217. Herrera, F., Posadas, H., Sánchez, P., Villar, E.: Systemic embedded software generation from SystemC. Proceedings of Design, Automation and Test in Europe (DATE) pp. 10,142–10,149 (2003)
218. Hoare, C.: Communicating Sequential Processes. Prentice Hall International Series in Computer Science (1985)
219. Hofer, W., Lohmann, D., Scheler, F., Schröder-Preikschat, W.: Sloth: Threads as interrupts. Proceedings of the Real-Time Systems Symposium (RTSS) pp. 2004–2013 (2009)
220. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: Proceedings of the Symposium on Principles of Programming Languages (POPL), pp. 185–197. ACM, New York, NY, USA (2003). DOI 10.1145/604131.604148
221. Holzkamp, O.: Memory-aware mapping strategies for heterogeneous MPSoC systems. Ph.D. thesis, TU Dortmund (2017). URL https://eldorado.tu-dortmund.de/handle/2003/35958
222. Hopcroft, J., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison Wesley (2006)
223. Horn, W.: Some simple scheduling algorithms. Naval Research Logistics Quarterly, Vol. 21 pp. 177–185 (1974)
224. Huang, L., Xu, Q.: AgeSim: A simulation framework for evaluating the lifetime reliability of processor-based SoCs. Proceedings of Design, Automation and Test in Europe (DATE) (2010)
225. Huerlimann, D.: Opentrack home page. http://www.opentrack.ch (2020)
226. Hüls, T.: Energieoptimierung einer MPEG-Applikation. Master thesis, CS Dept., Univ. Dortmund, https://ls12-www.cs.tu-dortmund.de/daes/en/research/publications/completed-theses.html (2002)
227. Hunt, V.D., Puglia, A., Puglia, M.: RFID: a guide to radio frequency identification. Wiley (2007)
228. IBM: What's new in Rational Rhapsody 7.5.1. http://www.ibm.com/developerworks/rational/library/09/whatsnewinrationalrhapsody-7-5-1 (2009)
229. IBM: Rational DOORS. http://www-01.ibm.com/software/awdtools/doors/ (2016)

230. IBM: IBM Rational Statemate 4.6. https://www.ibm.com/support/pages/ibm-rational-statemate-46 (2018)
231. ICD Staff: ICD-C compiler framework. http://www.icd.de/en/embedded-systems/compiler-tool-development/icd-c (2016)
232. IEC: IEC 60848 – GRAFCET specification language for sequential function charts. https://webstore.iec.ch/publication/3684 (2002)
233. IEC: IEC 61508-1:2010 functional safety of electrical/ electronic/ programmable electronic safety-related systems – part 1: General requirements. https://webstore.iec.ch/publication/5515 (2010)
234. IEC: IEC 61508-2:2010 functional safety of electrical/electronic/programmable electronic safety-related systems - part 2: Requirements for electrical/electronic/programmable electronic safety-related systems. https://webstore.iec.ch/publication/5516 (2010)
235. IEC: IEC 61508-3:2010 functional safety of electrical/electronic/programmable electronic safety-related systems - part 3: Software requirements. https://webstore.iec.ch/publication/5517 (2010)
236. IEC: IEC 61513:2011 – nuclear power plants – instrumentation and control important to safety – general requirements for systems. https://webstore.iec.ch/publication/5532 (2011)
237. IEC: IEC 61511:2016 SER functional safety – safety instrumented systems for the process industry sector – all parts. https://webstore.iec.ch/publication/5527 (2016)
238. IEEE: 1076-1987 – IEEE Standard VHDL Language Reference Manual. http://standards.ieee.org/findstds/standard/1076-1987.html (1987)
239. IEEE: IEEE Graphic Symbols for Logic Functions Std 91a-1991. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=27895 (1991)
240. IEEE: 1076-1993 – IEEE Standard VHDL Language Reference Manual. http://standards.ieee.org/findstds/standard/1076-1993.html (1993)
241. IEEE: 1076-2000 – IEEE Standard VHDL Language Reference Manual. http://standards.ieee.org/findstds/standard/1076-2000.html (2000)
242. IEEE: 1076-2002 – IEEE Standard VHDL Language Reference Manual. http://standards.ieee.org/findstds/standard/1076-2002.html (2002)
243. IEEE: 1076-2008 – IEEE Standard VHDL Language Reference Manual. http://standards.ieee.org/findstds/standard/1076-2008.html pp. c1–626 (2009)
244. IEEE: 1666-2011 - IEEE Standard for Standard SystemC Language Reference Manual. https://ieeexplore.ieee.org/document/6134619 (2011)
245. IEEE: 1800-2012 - IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language. https://standards.ieee.org/findstds/standard/1800-2012.html (2012)
246. IEEE: P1076.1-2017 – Standard VHDL Analog and Mixed-Signal Extensions. https://standards.ieee.org/standard/1076_1-2017.html (2017)
247. Intel: Enhanced Intel® SpeedStep® Technology for the Intel® Pentium® M Processor – White paper. http://download.intel.com/design/network/papers/30117401.pdf (2004)
248. Intel: Motion estimation with Intel® streaming SIMD extensions 4 (Intel® SSE4). http://software.intel.com/en-us/articles/motion-estimation-with-intel-streaming-simd-extensions-4-intel-sse4 (2008)
249. Intel: Intel® AVX. http://software.intel.com/en-us/avx (2010)
250. Intel: Intel Itanium processors. https://www.intel.com/content/www/us/en/products/processors/itanium.html (2021)
251. International Electrotechnical Commission (IEC): Functional safety. https://www.iec.ch/functionalsafety/explained/ (2020)
252. Ishihara, T., Yasuura, H.: Voltage scheduling problem for dynamically variable voltage processors. Intern. Symp. on Low Power Electronics and Design (ISLPED) pp. 197–202 (1998)
253. ISO: Road vehicles – functional safety – part 3: Concept phase. https://www.iso.org/obp/ui/#iso:std:iso:26262:-3:ed-1:v1:en (2011)
254. ISO: Road vehicles – FlexRay communications system – Part 1: General information and use case definition. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=59804 (2013)

255. ISO: ISO 9001:2015(en)-Quality management systems – Requirements. https://www.iso.org/obp/ui/#iso:std:iso:9001:ed-5:v1:en (2015)
256. ISO: ISO/IEC 27000:2018(en) Information technology — Security techniques — Information security management systems — Overview and vocabulary. https://www.iso.org/obp/ui/#iso:std:iso-iec:27000:ed-5:v1:en (2018)
257. ISO/IEC: ISO/IEC 15437:2001 - Information technology – Enhancements to LOTOS (E-LOTOS). http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=27680 (2001)
258. ISO/IEC: ISO/IEC 25024:2015 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Measurement of data quality. https://www.iso.org/standard/35749.html (2015)
259. ISO/IEC: ISO/IEC 25022:2016 Systems and software engineering – Systems and software quality requirements and evaluation (SQuaRE) – Measurement of quality in use. https://www.iso.org/standard/35746.html (2016)
260. ISO/IEC: ISO/IEC 25023:2016 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Measurement of system and software product quality. https://www.iso.org/standard/35747.html (2016)
261. Israr, A., Huss, S.: Specification and design considerations for reliable embedded systems. Proceedings of Design, Automation and Test in Europe (DATE) pp. 1111–1116 (2008)
262. ITRS Organization: International Technology Roadmap for Semiconductors – 2013 Edition – Executive Summary. http://www.itrs2.net/2013-itrs.html (2013)
263. Iyer, A., Marculescu, D.: Power and performance evaluation of globally asynchronous locally synchronous processors. Int. Symp. on Computer Architecture (ISCA) pp. 158–168 (2002)
264. Jackson, J.: Scheduling a production line to minimize maximum tardiness. Management Science Research Project 43, University of California, Los Angeles (1955)
265. Jacobs, M., Hahn, S., Hack, S.: WCET analysis for multi-core processors with shared buses and event-driven bus arbitration. In: Proceedings of the 23rd International Conference on Real Time and Networks Systems, RTNS ’15, pp. 193–202. ACM, New York, NY, USA (2015). DOI 10.1145/2834848.2834872. URL http://doi.acm.org/10.1145/2834848.2834872
266. Jacobs, M., Hahn, S., Hack, S.: A framework for the derivation of WCET analyses for multi-core processors. In: 28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016, pp. 141–151. IEEE Computer Society (2016). DOI 10.1109/ECRTS.2016.19. URL http://dx.doi.org/10.1109/ECRTS.2016.19
267. Jain, M., Balakrishnan, M., Kumar, A.: ASIP design methodologies: Survey and issues. 14th Int. Conf. on VLSI Design pp. 76–81 (2001)
268. Janka, R.: Specification and Design Methodology for Real-Time Embedded Systems. Kluwer Academic Publishers (2002)
269. Jantsch, A.: Modeling Embedded Systems and SoC’s: Concurrency and Time in Models of Computation. Morgan Kaufmann (2004)
270. Jantsch, A.: Models of embedded computation. In: R. Zurawski (ed.): Embedded Systems Handbook, CRC Press (2006)
271. Java Community Process: JSR-1 – Real-time Specification for Java. http://www.jcp.org/en/jsr/detail?id=1 (2019)
272. Jayaseelan, R., Mitra, T., Li, X.: Estimating the worst-case energy consumption of embedded software. In: 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS’06), pp. 81–90. IEEE (2006)
273. Jensen, K.: Coloured Petri nets: basic concepts, analysis methods and practical use, vol. 1. Springer (2013)
274. Jewett, J.W., Serway, R.A.: Physics for scientists and engineers with modern physics. Thomson Higher Education (2007)
275. Jha, P., Dutt, N.: Rapid estimation for parameterized components in high-level synthesis. IEEE Transactions on VLSI Systems pp. 296–303 (1993)
276. Jones, M.: What really happened on Mars Rover Pathfinder. http://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html (1997)

277. Jones, N.D.: An introduction to partial evaluation. ACM Comput. Surv. **28**(3), 480–503 (1996). DOI http://doi.acm.org/10.1145/243439.243447
278. Jouppi, N., Young, C., Patil, N., Patterson, D.: A Domain-Specific Architecture for Deep Neural Networks. Commun. ACM pp. 50–59 (2018). DOI 10.1145/3154484
279. Kahn, G.: The semantics of a simple language for parallel programming. Proc. of the Int. Federation for Information Processing (IFIP) pp. 471–475 (1974)
280. Kamal, R.: Embedded systems: architecture, programming and design. Tata McGraw-Hill (2003)
281. Kang, S., Dean, A.G.: Leveraging Both Data Cache and Scratchpad Memory Through Synergetic Data Allocation. In: Proceedings of the Real Time and Embedded Technology and Applications Symposium (RTAS), pp. 119–128. IEEE Computer Society, Washington, DC, USA (2012). DOI 10.1109/RTAS.2012.22
282. Kannan, A., Shrivastava, A., Pabalkar, A., Lee, J.E.: A Software Solution for Dynamic Stack Management on Scratch Pad Memory. In: Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC), pp. 612–617 (2009)
283. Karp, R.M., Miller, R.E.: Properties of a model for parallel computations: Determinancy, termination, queueing. SIAM Journal of Applied Mathematics **14**, 1390–1411 (1966)
284. Keding, H., Willems, M., Coors, M., Meyr, H.: FRIDGE: A fixed-point design and simulation environment. Design, Automation and Test in Europe (DATE) pp. 429–435 (1998)
285. Keim, M., Drechsler, R., Becker, B., Martin, M., Molitor, P.: Polynomial formal verification of multipliers. Formal Methods in System Design **22**, 39–58 (2003)
286. Keinert, J., Streubühr, M., Schlichter, T., Falk, J., Gladigau, J., Haubelt, C., Teich, J., Meredith, M.: SystemCodesigner – an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. ACM Transactions on Design Automation of Electronic Systems **14**, 1–23 (2009)
287. Kelter, T., Marwedel, P.: Parallelism analysis: Precise WCET values for complex multi-core systems. Sci. Comput. Program. **133**, 175–193 (2017). DOI 10.1016/j.scico.2016.01.007. URL http://dx.doi.org/10.1016/j.scico.2016.01.007
288. Kempe, M.: Ada 95 reference manual, ISO/IEC standard 8652:1995. (HTML-version), http://www.adahome.com/rm95/ (1995)
289. Kempe Software Capital Enterprises (KSCE): Ada home: The web site for Ada. http://www.adahome.com (2021)
290. Kernighan, B.W., Ritchie, D.M.: The C Programming Language. Prentice Hall (1988)
291. Kerrison, S., Eder, K.: Energy modeling of software for a hardware multithreaded embedded microprocessor. ACM Trans. Embed. Comput. Syst. **14**(3), 56:1–56:25 (2015). DOI 10.1145/2700104. URL http://doi.acm.org/10.1145/2700104
292. Khorramabadi, H.: ADC Converters – Pipelined ADCs. UC Berkeley, EECS 247, Lecture 22, http://www-inst.eecs.berkeley.edu/~ee247/fa05/lectures/L22_f05.pdf (2005)
293. Khorramabadi, H.: Oversampled ADCs. UC Berkeley, EECS 247, Lecture 23, http://www-inst.eecs.berkeley.edu/~ee247/fa05/lectures/L22_f05.pdf (2009)
294. Kienhuis, B., Rijjpkema, E., Deprettere, E.: Compaan: Deriving process networks from Matlab for embedded signal processing architectures. Proceedings of the International Symposium on Hardware-Software Codesign (CODES) pp. 29–40 (2000)
295. Kim, J., Lee, S., Shin, H.: Effective task scheduling for embedded systems using iterative cluster slack optimization. Circuits and Systems **4**, 479–488 (2013)
296. Klaiber, A.: The technology behind Crusoe™ processors. http://web.archive.org/web/20010602205826/www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf (2000)
297. Kleidermacher, D., Kleidermacher, M.: Embedded Systems Security – Practical Methods for Safe and Secure Software and Systems Development. Newnes (2012)
298. Klumpp, M., Clausen, U., ten Hompel, M.: Logistics Research and the Logistics World of 2050, pp. 1–6. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). DOI 10.1007/978-3-642-32838-1_1. URL http://dx.doi.org/10.1007/978-3-642-32838-1_1
299. Ko, M., Koo, I.: An overview of interactive video on demand system. http://www.ece.ubc.ca/~irenek/techpaps/vod/vod.html (1996)

300. Kobryn, C.: UML 2001: A standardization Odyssey. Communications of the ACM (CACM), available at http://www.omg.org/attachments/pdf/UML_2001_CACM_Oct99_p29-Kobryn.pdf pp. 29–36 (2001)
301. Kocher, P., Lee, R., McGraw, G., Raghunathan, A.: Security as a new dimension in embedded system design. In: Proceedings of the 41st Annual Design Automation Conference, DAC '04, pp. 753–760. ACM, New York, NY, USA (2004). DOI 10.1145/996566.996771. URL http://doi.acm.org/10.1145/996566.996771. Moderator-Ravi, Srivaths
302. Kohavi, Z., Jha, N.K.: Switching and Finite Automata Theory, 3 edn. Cambridge University Press (2010)
303. Koopman, P.J., Upender, B.P.: Time division multiple access without a bus master. United Technologies Research Center, UTRC Technical Report RR-9500470, http://www.ece.cmu.edu/~koopman/jtdma/jtdma.html (1995)
304. Kopetz, H.: Real-Time Systems – Design Principles for Distributed Embedded Applications –. Springer (2011)
305. Kopetz, H., Grunsteidl, G.: TTP – a protocol for fault-tolerant real-time systems. IEEE Computer **27**, 14–23 (1994)
306. Korf, R.E.: A new algorithm for optimal bin packing. American Association for Artificial Intelligence Proceedings, www.aaai.org pp. 731–736 (2002)
307. Korte, B., Vygen, J.: Kombinatorische Optimierung, Kapitel 18. Springer (2012)
308. Kotthaus, H.: Methods for efficient resource utilization in statistical machine learning algorithms. Ph.D. thesis, TU Dortmund University, Dortmund (2018). URL http://dx.doi.org/10.17877/DE290R-18928
309. Kranitis, N., Paschalis, A., Gizopoulos, D., Zorian, Y.: Instruction-based self-testing of processor cores. Journal of Electronic Testing **19**, 103–112 (2003)
310. Krhovjak, J., Matyas, V.: Secure hardware – pv018. http://www.fi.muni.cz/~xkrhovj/lectures/2006_PV018_Secure_Hardware_slides.pdf (2006)
311. Krishna, C., Shin, K.G.: Real-Time Systems. McGraw-Hill, Computer Science Series (1997)
312. Krstić, A., Cheng, K.: Delay fault testing of VLSI circuits. Kluwer Academic Publishers (1998)
313. Krstić, A., Dey, S.: Embedded software-based self-test for programmable core-based designs. IEEE Design & Test pp. 18–27 (2002)
314. Krüger, G.: Automatic generation of self-test programs: A new feature of the MIMOLA design system. Proceedings of the Design Automation Conference (DAC) pp. 378–384 (1986)
315. Kuang, S.R., Chen, C.Y., Liao, R.Z.: Partitioning and pipelined scheduling of embedded system using integer linear programming. In: 11th International Conference on Parallel and Distributed Systems (ICPADS'05), vol. 2, pp. 37–41 (2005). DOI 10.1109/ICPADS.2005.219
316. Kühn, R.: Analyse und Evaluation von Gütekriterien für approximative Source-to-Source Transformationen. Bachelorarbeit, Fakultät für Informatik, TU Dortmund (2016)
317. Kumar, R., Farkas, K.I., Jouppi, N.P., Ranganathan, P., Tullsen, D.M.: Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 36, pp. 81–. IEEE Computer Society, Washington, DC, USA (2003). URL http://dl.acm.org/citation.cfm?id=956417.956569
318. Labrosse, J.: Embedded Systems Building Blocks - Complete and Ready-to-use Modules in C. Elsevier (2000)
319. Lala, P.: Fault Tolerant and Fault Testable Hardware Design. Prentice Hall (1985)
320. Lam, K.Y., Kuo, T.W. (eds.): Real-Time Database Systems: Architecture and Techniques. Kluwer Academic Publishers, Norwell, MA, USA (2001)
321. Lam, M.S., Rothberg, E.E., Wolf, M.E.: The cache performance and optimizations of blocked algorithms. Architectural Support for Programming Languages and Operating Systems (ASPLOS) pp. 63–74 (1991)
322. Lang, F.: Analog-Digital-Umsetzer für die hochbitratige Datenübertragung. Ph.D. thesis, Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart (2014)
323. Laplante, P.: Real-Time Systems: Design and Analysis - An Engineer's Handbook. IEEE Press (1997)

324. Laprie, J.C. (ed.): Dependability: basic concepts and terminology in English, French, German, Italian and Japanese. IFIP WG 10.4, Dependable Computing and Fault Tolerance, In: volume 5 of Dependable computing and fault tolerant systems, Springer Verlag (1992)
325. Latendresse, M.: The code compression bibliography. http://www.iro.umontreal.ca/~latendre/compactBib (2004)
326. Law, A.M.: Simulation Modeling & Analysis. McGraw-Hill (2006)
327. Lawler, E.L.: Optimal sequencing of a single machine subject to precedence constraints. Managements Science, Vol. 19 pp. 544–546 (1973)
328. Le Boudec, J., Thiran, P.: Network Calculus. Springer, LNCS # 2050 (2001)
329. Lee, E.A.: Embedded software – an agenda for research. Tech. rep., UCB ERL Memorandum M99/63 (1999)
330. Lee, E.A.: The future of embedded software. ARTEMIS Conference, Graz, http://ptolemy.eecs.berkeley.edu/presentations/06/FutureOfEmbeddedSoftware_Lee_Graz.ppt (2006)
331. Lee, E.A.: Computing foundations and practice for cyber-physical systems: A preliminary report. Tech. Rep. UCB/EECS-2007-72, EECS Department, University of California, Berkeley (2007). URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-72.html
332. Lee, E.A.: Leveraging synchronized clocks in cyber-physical systems. Workshop on Synchronization in Telecommunications Systems, http://www.atis.org/wsts/papers/3-3-1_UCBerkeley_Lee_LeveragingClocks.pdf (2014)
333. Lee, E.A.: Absolutely positively on time. IEEE Computer (July, 2005)
334. Lee, E.A., Messerschmitt, D.: Synchronous data flow. Proc. of the IEEE, vol. 75 pp. 1235–1245 (1987)
335. Lee, E.A., Seshia, S.A.: Introduction to Embedded Systems, A Cyber-Physical Systems Approach, 2 edn. MIT Press, http://LeeSeshia.org (2017)
336. Lee, S., Gerstlauer, A.: Fine grain word length optimization for dynamic precision scaling in DSP systems. In: VLSI-SoC, pp. 266–271 (2013)
337. Lelli, J., Scordino, C., Abeni, L., Faggioli, D.: Deadline scheduling in the Linux kernel. Software: Practice and Experience **46**(6), 821–839 (2016). DOI 10.1002/spe.2335. URL http://dx.doi.org/10.1002/spe.2335. Spe.2335
338. Leupers, R.: Retargetable Code Generation for Digital Signal Processors. Kluwer Academic Publishers (1997)
339. Leupers, R.: Code Optimization Techniques for Embedded Processors - Methods, Algorithms, and Tools. Kluwer Academic Publishers (2000)
340. Leveson, N.: Safeware, System Safety and Computers. Addison Wesley (1995)
341. Levine, D.: Users guide to the PGAPack parallel genetic algorithm library. Tech. Rep. ANL-95/18, Tech. Rep. Argonne National Lab. (1996)
342. Lewis, J., Rashba, E., Brophy, D.: VHDL-2006-D3.0 Tutorial. Tutorial at Design, Automation, and Test in Europe (DATE), http://www.amos.eguru-il.com/vhdl_info/date_vhdl_tutorial.pdf (2007)
343. Li, L., Wu, H., Feng, H., Xue, J.: Towards Data Tiling for Whole Programs in Scratchpad Memory Allocation. In: Proceedings of the Asia-Pacific Conference on Advances in Computer Systems Architecture (ACSAC), pp. 63–74. Springer, Berlin, Heidelberg (2007). DOI 10.1007/978-3-540-74309-5_8
344. Li, Y.T., Malik, S.: Performance analysis of embedded software using implicit path enumeration. ACM SIGPLAN Notices **30** (1995)
345. Li, Z.R. (ed.): Organic light-emitting materials and devices. CRC Press (2015)
346. Liebisch, D.C., Jain, A.: Jessi common framework design management: the means to configuration and execution of the design process. In: Conf. on European Design Automation (EURO-DAC), pp. 552–557. IEEE Computer Society Press (1992)
347. LIN Consortium: LIN Specification Package - Revision 2.2A. https://www.cs-group.de/wp-content/uploads/2016/11/LIN_Specification_Package_2.2A.pdf (2010)
348. Liu, C.L., Layland, J.W.: Scheduling algorithms for multi-programming in a hard real-time environment. Journal of the Association for Computing Machinery (JACM) pp. 40–61 (1973)
349. Liu, J.W.: Real-Time Systems. Prentice Hall (2000)

350. Liu, Y., Zhang, W.: Scratchpad Memory Architectures and Allocation Algorithms for Hard Real-Time Multicore Processors. Journal of Computing Science and Engineering (2015)
351. Lohmann, D., Hofer, W., Schröder-Preikschat, W., Spinczyk, O.: CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In: USENIX Annual Technical Conference (2009)
352. Lohmann, D., Scheler, F., Schröder-Preikschat, W., Spinczyk, O.: PURE Embedded Operating Systems – CiAO. Proc. International Workshop on Operating System Platforms for Embedded Real-Time Applications, (OSPERT) (2006)
353. Lohmann, D., Streicher, J., Hofer, W., Spinczyk, O., Schröder-Preikschat, W.: Configurable memory protection by aspects. In: Proceedings of the 4th Workshop on Programming Languages and Operating Systems (PLOS '07). ACM Press, New York, NY, USA (2007)
354. Lokuciejewski, P., Marwedel, P.: WCET-aware Source Code and Assembly Level Optimization Techniques for Real-Time Systems. Springer (2010)
355. Lokuciejewski, P., Stolpe, M., Morik, K., Marwedel, P.: Automatic selection of machine learning models for WCET-aware compiler heuristic generation. In: Proceedings of the 4th Workshop on Statistical and Machine Learning Approaches to ARchitecture and compilaTion (SMART) (2010). URL http://ctuning.org/dissemination/smart10-01.pdf
356. López, J.M., Díaz, J.L., García, D.F.: Utilization bounds for EDF scheduling on real-time multiprocessor systems. Real-Time Syst. **28**(1), 39–68 (2004). DOI 10.1023/B:TIME.0000033378.56741.14. URL http://dx.doi.org/10.1023/B:TIME.0000033378.56741.14
357. Lu, Y.H., Chung, E.Y., Šimunic, T., Benini, L., De Micheli, G.: Quantitative comparison of power management algorithms. Proceedings of Design, Automation and Test in Europe (DATE) pp. 20–26 (2000)
358. Luican, I.I., Zhu, H., Balasa, F.: Formal Model of Data Reuse Analysis for Hierarchical Memory Organizations. In: Proceedings of the International Conference on Computer-Aided Design (ICCAD), pp. 595–600. ACM, New York, NY, USA (2006). DOI 10.1145/1233501.1233623
359. Machanik, P.: Approaches to addressing the memory wall. Technical Report, November, Univ. Brisbane (2002)
360. Macii, A., Benini, L., Poncino, M.: Memory Design Techniques for Low Energy Embedded Systems. Kluwer Academic Publishers (2002)
361. Macii, E. (ed.): Ultra low-power electronics and design. Springer (2004)
362. Maculan, N., Porto, S.C.S., Ribeiro, C., de Souza, C.C.: A new formulation for scheduling unrelated processors under precedence constraints. Revue Francaise d'Automatique, d'informatique et de recherche Operationelle (RAIRO) pp. 87–92 (1999)
363. Magnusson, P.S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A full system simulation platform. Computer **35**(2), 50–58 (2002). DOI 10.1109/2.982916
364. Man, H.D.: From the heaven of software to the hell of nanoscale physics: an industry in transition. Keynote, HiPEAC ACACES Summer School, L'Aquila (2007)
365. Manwell, J.F., McGowan, J.G.: Lead acid battery storage model for hybrid energy systems. Solar Energy **50**, 399–405 (1993)
366. Marian, N., Ma, Y.: Translation of Simulink models to component-based software models. 8th Int. Workshop on Research and Education in Mechatronics REM, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.86.1490&rep=rep1&type=pdf pp. 262–267 (2007)
367. Marongiu, A., Benini, L.: Efficient OpenMP support and extensions for MPSoCs with explicitly managed memory hierarchy. Proceedings of Design, Automation and Test in Europe (DATE) pp. 809–814 (2009)
368. Martello, S., Toth, P.: Knapsack Problems – Algorithms and Computer Implementations, chapter 6. John Wiley & Sons, http://www.or.deis.unibo.it/kp/Chapter6.pdf (1990)
369. Martin, G., Müller, W. (eds.): UML™for SOC Design. Springer (2010)
370. Martin, S.M., Flautner, K., Mudge, T., Blaauw, D.: Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design pp. 721–725 (2002). DOI http://doi.acm.org/10.1145/774572.774678

371. Marwedel, P.: Ein Software-System zur Synthese von Rechnerstrukturen und zur Erzeugung von Microcode. Tech. rep., Habilitationsschrift, Univ. Kiel; Nachdruck: Bericht Nr. 356, FB Informatik, Univ. Dortmund (1985)
372. Marwedel, P.: Embedded System Design. Kluwer Academic Publishers (2003)
373. Marwedel, P.: Towards laying common grounds for embedded system design education. ACM SIGBED Review pp. 25–28 (2005)
374. Marwedel, P.: MIMOLA – a fully synthesizable language. in: Prabhat Mishra, Nikil Dutt (Ed.): Processor Description Languages – Applications and Methodologies, Morgan Kaufmann pp. 35–63 (2008)
375. Marwedel, P., Engel, M.: Plea for a holistic analysis of the relationship between information technology and carbon-dioxide emissions. In: Workshop on Energy-aware Systems and Methods (GI-ITG). Hanover, Germany (2010)
376. Marwedel, P., Engel, M.: Flipped Classroom Teaching for a Cyber-Physical System Course – an Adequate Presence-Based Learning Approach in the Internet Age. Proceedings of the Tenth European Workshop on Microelectronics Education (EWME) (2014)
377. Marwedel, P., Falk, H., Neugebauer, O.: Memory-aware optimization of embedded software for multiple objectives. In: Soonhoi Ha, J. Teich (eds.): Handbook of Hardware/Software CoDesign (2017)
378. Marwedel, P., Goossens, G. (eds.): Code Generation for Embedded Processors. Kluwer Academic Publishers (1995)
379. Marwedel, P., Mitra, T., Grimheden, M., Andrade, H.A.: Survey on Education for Cyber-Physical Systems. IEEE Design & Test (2020)
380. Marwedel, P., Schenk, W.: Cooperation of synthesis, retargetable code generation and test generation in the MSS. Proceedings of the European Design Automation Conference (Euro-DAC) pp. 63–69 (1993)
381. Marzario, L., Lipari, G., Balbastre, P., Crespo, A.: IRIS: a new reclaiming algorithm for server-based real-time systems. Proceedings of the Real Time and Embedded Technology and Applications Symposium (RTAS) (2004)
382. Massa, A.J.: Embedded Software Development with eCos. Prentice Hall (2002)
383. MathWorks, T.: Stateflow. https://www.mathworks.com/products/stateflow.html (2021)
384. McGregor, I.: The relationship between simulation and emulation. Winter Simulation Conference pp. 1683–1688 (2002)
385. McIlroy, R., Dickman, P., Sventek, J.: Efficient Dynamic Heap Allocation of Scratch-pad Memory. In: Proceedings of the International Symposium on Memory Management, pp. 31–40 (2008)
386. Mckusick, M.K., Karels, M.J.: A New Virtual Memory Implementation for Berkeley UNIX. In: EUUG Conference Proceedings (Autumn), pp. 451–458 (1986)
387. McLaughlin, M., Moore, A.: Real-Time Extensions to UML. http://www.ddj.com/184410749 (1998)
388. McNamee, D., Walpole, J., Pu, C., Cowan, C., Krasic, C., Goel, A., Wagle, P., Consel, C., Muller, G., Marlet, R.: Specialization tools and techniques for systematic optimization of system software. ACM Trans. Comput. Syst. **19**(2), 217–251 (2001). DOI http://doi.acm.org/10.1145/377769.377778
389. Meena, J.S., Sze, S.M., Chand, U., Tseng, T.Y.: Overview of emerging nonvolatile memory technologies. Nanoscale Research Letters **9**(1), 526 (2014). DOI 10.1186/1556-276X-9-526. URL http://dx.doi.org/10.1186/1556-276X-9-526
390. Meijer, S., Nikolov, H., Stefanov, T.: Throughput modeling to evaluate process merging transformations in polyhedral process networks. Proceedings of Design, Automation and Test in Europe (DATE) (2010)
391. Menard, D., Sentieys, O.: Automatic evaluation of the accuracy of fixed-point algorithms. Proceedings of Design, Automation and Test in Europe (DATE) pp. 529–535 (2002)
392. Merkel, A., Bellosa, F.: Event-driven thermal management in SMP systems. Proceedings of the Second Workshop on Temperature-Aware Computer Systems (TACS'05) (2005)

393. Mermet, J., Marwedel, P., Ramming, F.J., Newton, C., Borrione, D., Lefaou, C.: Three decades of hardware description languages in Europe. Journal of Electrical Engineering and Information Science **3**, 106pp (1998)
394. Merriam-Webster Inc.: Dictionary, Eintrag für „Task“. https://www.merriam-webster.com/dictionary/task (2021)
395. Mesa-Martinez, F.J., Ardestani, E.K., Renau, J.: Characterizing processor thermal behavior. ASPLOS ’10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems, http://doi.acm.org/10.1145/1736020.1736043 pp. 193–204 (2010)
396. Message Passing Interface Forum: MPI: A message-passing interface standard - version 3.1. http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf (2015)
397. MISRA C Working Group: MISRA compliance:2016 – achieving compliance with MISRA coding guidelines. http://www.misra.org.uk/LinkClick.aspx?fileticket=w_Syhpkf7xA%3d&tabid=57 (2016)
398. Mittal, S.: A survey of techniques for approximate computing. ACM Comput. Surv. **48**(4), 62:1–62:33 (2016). DOI 10.1145/2893356. URL http://doi.acm.org/10.1145/2893356
399. Mittal, S., Vetter, J.S.: A survey of methods for analyzing and improving gpu energy efficiency. ACM Comput. Surv. **47**(2), 19:1–19:23 (2014). DOI 10.1145/2636342. URL http://doi.acm.org/10.1145/2636342
400. Modelica Association: Modelica® – A Unified Object-Oriented Language for Systems Modeling – Language Specification – Version 3.3. https://www.modelica.org/documents/ModelicaSpec33.pdf (2012)
401. Möller, S., Raake, A.: Quality of Experience – Advanced Concepts, Applications and Methods. Springer (2014)
402. Montoreano, M.: Transaction level modeling using OSCI TLM 2.0. http://accellera.org/images/downloads/standards/systemc/systemc-2.3.1.tgz (2007)
403. MOST Cooperation: MOST Worldwide. http://www.mostcooperation.com/ (2021)
404. Mosterman, P.J.: Hybrid Dynamic Systems: Modeling and Execution. in: Paul A. Fishwick (ed.): Handbook of Dynamic System Modeling, CRC Press (2007)
405. Moynihan, T.: CMOS is winning the camera sensor battle, and here’s why. http://www.techhive.com/article/246931/cmos_is_winning_the_camera_sensor_battle_and_heres_why.html?page=0 (2011)
406. Muchnick, S.S.: Advanced compiler design and implementation. Morgan Kaufmann Publishers, Inc. (1997)
407. Mukherjee, S.: Architecture Design for Soft Errors. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2008)
408. Müller, W., Rosenstiel, W., Ruf, J.: SystemC – Methodologies and Applications. Kluwer Academic Publishers (2003)
409. Muralimanohar, N., Balasubramonian, R., Jouppi, N.P.: CACTI 6.0: A Tool to Model Large Caches. International Symposium on Microarchitecture (2007), http://www.hpl.hp.com/techreports/2009/HPL-2009-85.pdf (2009)
410. National Academies of Sciences, Engineering, and Medicine: A 21st Century Cyber-Physical Systems Education. Washington, DC: The National Academies Press. https://doi.org/10.17226/23686 (2016)
411. National Research Council: Embedded, Everywhere. National Academies Press (2001)
412. National Research Council: Interim Report on the 21st Century Cyber-Physical Systems Education. The National Academies Press (2015)
413. National Science Foundation: Cyber-physical systems (CPS). http://www.nsf.gov/pubs/2013/nsf13502/nsf13502.htm (2013)
414. National Space-Based Positioning, Navigation, and Timing Coordination Office: Global positioning system. http://www.gps.gov (2019)
415. Nationale Plattform Elektromobilität (NPE): Roadmap integrierte Zell- und Batterieproduktion Deutschland. Gemeinsame Geschäftsstelle Elektromobilität der Bundesregierung (GGEMO), http://nationale-plattform-elektromobilitaet.de/fileadmin/user_upload/Redaktion/NPE_AG2_Roadmap_Zellfertigung_final_bf.pdf (2016)

416. Navet, N., Simonot-Lion, F.: Automotive Embedded Systems Handbook. CRC Press (2009)
417. Neugebauer, O., Engel, M., Marwedel, P.: A parallelization approach for resource-restricted embedded heterogeneous MPSoCs inspired by OpenMP. Journal of Systems and Software (2016)
418. Neugebauer, O., Libuschewski, P., Engel, M., Mueller, H., Marwedel, P.: Plasmon-based virus detection on heterogeneous embedded systems. In: Proceedings of the International Workshop on Software and Compilers for Embedded Systems (SCOPES) (2015)
419. Neumann, P.G.: Computer Related Risks. Addison Wesley (1995)
420. Neumann, P.G. (ed.): The risks digest, forum on the risks to the public in computers and related Systems. http://catless.ncl.ac.uk/risks (2021)
421. Nguyen, N., Dominguez, A., Barua, R.: Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES) pp. 115–125 (2005)
422. Nicolescu, G., Mosterman, P.J.: Model-Based Design for Embedded System. CRC Press (2010)
423. Nikolov, H., Thompson, M., Stefanov, T., Pimentel, A., Polstra, S., Bose, R., Zissulescu, C., Deprettere, E.: Daedalus: toward composable multimedia MP-SoC design. In: Proceedings of the Design Automation Conference (DAC), pp. 574–579 (2008)
424. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Lecture Notes in Computer Science. Springer Berlin Heidelberg (2002). URL https://books.google.de/books?id=R6ul20M6nTIC
425. N.N.: First workshop on CPS education. Philadelphia, PA (part of CPSWeek 2013), http://cpsvo.org/group/edu/workshop (2013)
426. Noergard, T.: Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers. Newnes (2012)
427. Northeast Sustainable Energy Association: Buildingenergy. http://www.nesea.org/ (2021)
428. Novosel, D.: Timing the power grid. http://www.pserc.wisc.edu/documents/general_information/presentations/smartr_grid_executive_forum/ (2009)
429. Object Management Group (OMG): Real-time CORBA specification, version 1.2, jan. 2005. http://www.omg.org/cgi-bin/doc?formal/05-01-04.ps (2005)
430. Object Management Group (OMG): UML™ profile for schedulability, performance, and time specification, version 1.1. http://www.omg.org/cgi-bin/doc?formal/05-01-02.pdf (2005)
431. Object Management Group (OMG): A UML™ profile for MARTE: Modeling and analysis of real-time embedded systems - 1.0. http://www.omg.org/spec/MARTE/1.0/PDF (2009)
432. Object Management Group (OMG): Systems modeling language (SysML™). https://www.omg.org/technology/readingroom/System-Modeling-Language.htm (2020)
433. Object Management Group (OMG): CORBA® basics. https://www.corba.org/faq.htm (2021)
434. Object Management Group (OMG): OMG® Specifications. http://www.omg.org/spec/ (2021)
435. Object Management Group (OMG): Unified modeling language™ resource page. http://www.uml.org (2021)
436. Occam user group: Transputer and occam bibliography. http://www.transputer.net/obooks/oug/oug-bib.pdf (1990)
437. O'Neill, A.: Analog to digital types. IEEE TV, https://ieeetv.ieee.org/conference-highlights/analog-to-digital-types? (2006)
438. Open Connectivity Foundation: About UPnP. https://openconnectivity.org/developer/specifications/upnp-resources/upnp/ (2021)
439. OpenMP Architecture Review Board: OpenMP application program interface. http://www.openmp.org/mp-documents/spec30.pdf (2008)
440. Oppenheim, A.V., Schafer, R., Buck, J.R.: Digital Signal Processing. Pearson Higher Education (2009)
441. OSEK Group: OSEK/VDX – communication (version 3.0.3). http://portal.osek-vdx.org/files/pdf/specs/osekcom303.pdf (2004)
442. Otter, M., Winkler, D.: Modelica Overview. https://www.modelica.org/education/educational-material/lecture-material/english/ModelicaOverview.ppt (2013)

443. Pallister, J., Kerrison, S., Morse, J., Eder, K.: Data dependent energy modelling for worst case energy consumption analysis. arXiv preprint arXiv:1505.03374 (2015)
444. Palumbo, M.: The ERTMS/ETCS signalling system – an overview on the Standard European Interoperable signalling and train control system. http://www.railwaysignalling.eu/wp-content/uploads/2016/09/ERTMS_ETCS_signalling_system_revF.pdf (2016)
445. Pan, S., Hu, Y., Li, X.: IVF: Characterizing the vulnerability of microprocessor structures to intermittent faults. Proceedings of Design, Automation and Test in Europe (DATE) (2010)
446. Pappas, G.J.: Science of Cyber-Physical Systems-Bridging CS and Control. http://cps-vo.org/node/5876 (2012)
447. Parker, K.P.: The Boundary Scan Handbook. Kluwer Academic Publishers (1992)
448. Patterson, D.: Domain-Specific Architectures for Deep Neural Networks. https://inst.eecs.berkeley.edu//~cs152/sp19/lectures/L20-DSA.pdf (2019)
449. Paulin, P., Knight, J.: Force-directed scheduling in automatic data path synthesis. Proceedings of the Design Automation Conference (DAC) (1987)
450. Petri, C.A.: Kommunikation mit Automaten. Schriften des Rheinisch-Westfälischen Institutes für instrumentelle Mathematik an der Universität Bonn (1962)
451. Peukert, W.: Über die Abhängigkeit der Kapacität von der Entladestromstärcke bei Bleiakkumulatoren. Elektrotechnische Zeitschrift 20 (1897)
452. Piao, X., Han, S., Kim, H., Park, M., Cho, Y., Cho, S.: Predictability of earliest deadline zero laxity algorithm for multiprocessor real-time systems. Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06) pp. 6 pp.– (2006). DOI 10.1109/ISORC.2006.64
453. Piatkowski, N.: Exponential families on resource-constrained systems. Ph.D. thesis, TU Dortmund University, Dortmund (2018). URL https://eldorado.tu-dortmund.de/handle/2003/36877
454. Piatkowski, N., Lee, S., Morik, K.: Integer undirected graphical models for resource-constrained systems. Neurocomputing **173**(1), 9–23 (2016). URL http://www.sciencedirect.com/science/article/pii/S0925231215010449
455. Pinedo, M.L.: Scheduling – Theory, Algorithms, and Systems, 5th edn. Springer (2016)
456. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering. Springer, ISBN-10: 3540289011 (2005)
457. Popovici, K., Rousseau, F., Jerraya, A.A., Wolf, M.: Embedded Software Design and Programming of Multiprocessor System-on-Chip. Springer (2010)
458. Potop-Butucaru, D., de Simone, R., Talpin, J.P.: The synchronous hypothesis and synchronous languages. In: R. Zurawski (ed.): Embedded Systems Handbook, CRC Press (2006)
459. Press, D.: Guidelines for Failure Mode and Effects Analysis for Automotive, Aerospace and General Manufacturing Industries. CRC Press (2003)
460. Ptolemaeus, C. (ed.): System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org (2014). URL http://ptolemy.org/books/Systems
461. Puaut, I.: Multi-core real-time scheduling. http://www.irisa.fr/alf/downloads/puaut/STR/STRmulticore.pdf (2012-2013)
462. Puaut, I.: Real-time systems – slides of a master course. https://team.inria.fr/pacap/members/isabelle-puaut/#teaching (2017-2018)
463. Pyka, R., Faßbach, C., Verma, M., Falk, H., Marwedel, P.: Operating system integrated energy aware scratchpad allocation strategies for multi-process applications. Proceedings of the International Workshop on Software and Compilers for Embedded Systems (SCOPES) pp. 41–50 (2007)
464. Radetzki, M. (ed.): Languages for Embedded Systems and their Applications. Springer (2009)
465. Rajkumar, R.: Real-time synchronization protocols for shared memory multiprocessors. Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS) pp. 116–123 (1990)
466. Rajkumar, R., Sha, L., Lehoczky, J.: Real-time synchronization protocols for multiprocessors. Proceedings of the Real-Time Systems Symposium (RTSS) pp. 259–269 (1988)
467. Rao, R., Vrudhula, S., Rakhmatov, D.: Battery modeling for energy-aware system design. IEEE Computer pp. 77–87 (2003)

468. Reisig, W.: Petri nets. Springer (1985)
469. Riccobene, E., Scandurra, P., Rosti, A., Bocchio, S.: A UML™ 2.0 profile for SystemC: toward high-level SoC design. In: Proceedings of the International Conference on Embedded Software (EMSOFT), pp. 138–141 (2005). DOI http://doi.acm.org/10.1145/1086228.1086254
470. Ritchie, D.M., Thompson, K.: The UNIX Time-sharing System. Commun. ACM **17**(7), 365–375 (1974). DOI 10.1145/361011.361061. URL http://doi.acm.org/10.1145/361011.361061
471. Rixner, S., Dally, W.J., Khailany, B.J., Mattson, P.J., Kapasi, U.J.: Register organization for media processing. 6th High-Performance Computer Architecture (HPCA-6) pp. 375–386 (2000)
472. Roitzsch, M.: Kurs Betriebsysteme und Sicherheit, Vorlesung 3. https://os.inf.tu-dresden.de/Studium/Bs/WS2019/V03-Prozesse.pdf (2019)
473. Rosenblum, M., Ousterhout, J.K.: The design and implementation of a log-structured file system. ACM Trans. Comput. Syst. **10**(1), 26–52 (1992). DOI 10.1145/146941.146943. URL http://doi.acm.org/10.1145/146941.146943
474. RTCA: DO-178B, Software Considerations in Airborne Systems and Equipment Certification. RTCA Inc. (1992)
475. Ruggiero, M., Benini, L.: Mapping task graphs to the CELL BE processor. http://www.artist-embedded.org/docs/Events/2008/Map2MPSoC/Map2mpsoc-08-ruggiero.pdf (2008)
476. Sangiovanni-Vincentelli, A.: The context for platform-based design. IEEE Design & Test of Computers p. 120 (2002)
477. Sangiovanni-Vincentelli, A., Zeng, H., Natale, M.D., Marwedel, P. (eds.): Embedded Systems Development – From Functional Methods to Implementations. Springer (2013). ISBN 978-1-4616-3878-6
478. Sarajlic, E., Yamahata, C., Cordero, M., Fujita, H.: Three-phase electrostatic rotary stepper micromotor with a flexural pivot bearing. Journal of Microelectromechanical Systems pp. 38 – 349 (2010)
479. Schmitz, M., Al-Hashimi, B., Eles, P.: Energy-efficient mapping and scheduling for DVS enabled distributed embedded systems. Proceedings of Design, Automation and Test in Europe (DATE) pp. 514–521 (2002)
480. Schneider, K.: The synchronous programming language Quartz. Tech. rep., Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany (2009)
481. Science X: IMEC reports breakthrough power efficiency and performance of coarse-grain processor. https://phys.org/news/2005-11-imec-breakthrough-power-efficiency-coarse-grain.html (2005)
482. SDL Forum Society: List of commercial tools. http://www.sdl-forum.org/Tools/Commercial.htm (2016)
483. SDL Forum Society: Home page. http://www.sdl-forum.org (2019)
484. SGI: OpenGL software development kit. https://www.opengl.org/sdk/libs/ (2016)
485. Sha, L., Rajkumar, R., Lehoczky, J.: Priority inheritance protocols: An approach to real-time synchronisation. IEEE Trans. on Computers pp. 1175–1185 (1990)
486. Shi, C., Brodersen, R.: An automated floating-point to fixed-point conversion methodology. Int. Conf. on Audio Speed and Signal Processing (ICASSP) pp. 529–532 (2003)
487. Siciliano, B., Oussama, O.: Handbook of Robotics. Springer (2016)
488. Siemens: SIMATIC STEP 7 Programming Software. http://www.automation.siemens.com/simatic/industriesoftware/html_76/products/step7.htm (2016)
489. Sifakis, J.: A notion of expressiveness for component-based design. Workshop on Foundations and Applications of Component-based Design, ES-Week, http://www.artist-embedded.org/docs/Events/2008/Components/SLIDES/12-JosephSifakis-WFCD-ArtistDesign-Oct192008.pdf (2008)
490. Simunic, T., Benini, L., Acquaviva, A., Glynn, P., De Micheli, G.: Energy efficient design of portable wireless devices. Intern. Symp. on Low Power Electronics and Design (ISLPED) pp. 49–54 (2000)

491. Simunic, T., Benini, L., Acquaviva, A., Glynn, P., De Micheli, G.: Dynamic voltage scaling and power management for portable systems. Proceedings of the Design Automation Conference (DAC) pp. 524–529 (2001)
492. Simunic-Rosing, T., Coskun, A.K., Whisnant, K.: Temperature aware task scheduling in MPSoCs. Proceedings of Design, Automation and Test in Europe (DATE) pp. 1659–1664 (2007)
493. Singh, A.K., Shafique, M., Kumar, A., Henkel, J.: Mapping on multi/many-core systems: Survey of current and emerging trends. In: Proceedings of the Design Automation Conference (DAC), pp. 1:1–1:10. ACM, New York, NY, USA (2013). DOI 10.1145/2463209.2488734. URL http://doi.acm.org/10.1145/2463209.2488734
494. Sipser, M.: Introduction to the Theory of Computation. Thomson Course Technology, Parts One and Two (2006)
495. Sirocic, B., Marwedel, P.: Levi Flexray® simulation software. https://ls12-www.cs.tu-dortmund.de/daes/media/documents/teaching/downloads/levi/download/leviFRP.zip (2007)
496. Sirocic, B., Marwedel, P.: Levi KPN simulation software. https://ls12-www.cs.tu-dortmund.de/daes/media/documents/teaching/downloads/levi/download/leviKPN.zip (2007)
497. Sirocic, B., Marwedel, P.: Levi RTS simulation software. https://ls12-www.cs.tu-dortmund.de/daes/media/documents/teaching/downloads/levi/download/leviRTS.zip (2007)
498. Sirocic, B., Marwedel, P.: Levi TDD simulation software. https://ls12-www.cs.tu-dortmund.de/daes/media/documents/teaching/downloads/levi/download/leviTDD.zip (2007)
499. Skadron, K., Stan, M.R., Ribando, R.J., Gurumurthi, S., Huang, W., Sankaranarayanan, K., Tarjan, D., Burr, J., Ghosh, S., Velusamy, S., Link, G.: Hotspot 6.0. http://lava.cs.virginia.edu/HotSpot/index.htm (2019)
500. Skadron, K., Stan, M.R., Sankaranarayanan, K., Huang, W., Velusamy, S., Tarjan, D.: Temperature-aware microarchitecture: Modeling and implementation. ACM Trans. Archit. Code Optim. **1**(1), 94–125 (2004). DOI 10.1145/980152.980157. URL http://doi.acm.org/10.1145/980152.980157
501. Skjellum, A., Kanevsky, A., Dandass, Y.S., Watts, J., Paavola, S., Cottel, D., Henley, G., Hebert, L.S., Cui, Z., Rounbehler, A.: The Real-Time Message Passing Interface Standard (MPI/RT-1.1). Concurrency and Computation: Practice and Experience **16**(S1), Si–S322 (2004). DOI 10.1002/cpe.744. URL http://dx.doi.org/10.1002/cpe.744
502. Smith, J.J., Nair, R.: Virtual Machines: Versatile Platforms For Systems And Processes. Morgan Kaufmann Publishers (2005)
503. Society for Display Technology: Home page. http://www.sid.org (2003)
504. Spivey, M.: The Z Notation: A Reference Manual, 2nd edn. Prentice Hall International Series in Computer Science (1992)
505. Sprint Consortium: Open SoC design platform for reuse and integration of IPs. http://ecsi.org/sprint (2008)
506. Sridhar, A., Vincenzi, A., Atienza, D., Brunschwiler, T.: 3D-ICE: a Compact Thermal Model for Early-Stage Design of Liquid-Cooled ICs. IEEE Transactions of Computers **63** (2014)
507. Stallings, W.: Operating Systems: Internals and Design Principles, 8 edn. Prentice Hall (2015)
508. Stankovic, J., Ramamritham, K.: The Spring kernel: a new paradigm for real-time systems. IEEE Software **8**, 62–72 (1991)
509. Steinke, S.: Untersuchung des Energieeinsparungspotenzials in eingebetteten Systemen durch energieoptimierende Compilertechnik. PhD thesis, TU Dortmund, http://hdl.handle.net/2003/2769 (2003)
510. Steinke, S., Grunwald, N., Wehmeyer, L., Banakar, R., Balakrishnan, M., Marwedel, P.: Reducing energy consumption by dynamic copying of instructions onto onchip memory. Proceedings of the International Symposium on System Synthesis (ISSS) pp. 213–218 (2002)
511. Steinke, S., Knauer, M., Wehmeyer, L., Marwedel, P.: An accurate and fine grain instruction-level energy model supporting software optimizations. Int. Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS) (2001)
512. Steinke, S., L.Wehmeyer, Lee, B.S., Marwedel, P.: Assigning program and data objects to scratchpad for energy reduction. Proceedings of Design, Automation and Test in Europe (DATE) pp. 409–417 (2002)

513. Storey, N.: Safety-critical Computer Systems. Addison Wesley (1996)
514. Stritter, E., Gunter, T.: Microprocessor architecture for a changing world: The Motorola 68000. IEEE Computer **12**, 43–52 (1979)
515. Stuijk, S.: Predictable mapping of streaming applications on multiprocessors. Ph.D. thesis, TU Eindhoven (2007)
516. Sundmaeker, H., Guillemin, P., Friess, P., Woelfflé, S.: Vision and Challenges for Realising the Internet of Things. Cluster of European Research Projects on the Internet of Things, European Commision (2010)
517. Sutherland, S.: An overview of SystemVerilog 3.1. EEdesign, May, available at http://www.eetimes.com/news/design/features/showArticle.jhtml?articleID=16501063 (2003)
518. Synopsys: System studio. https://www.synopsys.com/verification/virtual-prototyping.html (2020)
519. Synopsys® Inc.: HSPICE®: The gold standard for accurate circuit simulation. https://www.synopsys.com/Tools/Verification/AMSVerification/CircuitSimulation/HSPICE/Documents/hspice_ds.pdf (2019)
520. SYSGO AG: PikeOS RTOS and Virtualization Concept. http://www.sysgo.com (2019)
521. SystemC: Home page. http://www.SystemC.org (2020)
522. Taha, W., Cartwright, R.: Some challenges for model-based simulation. In: The 4th Analytic Virtual Integration of Cyber-Physical Systems Workshop, Vancouver, Canada, December 3, 2013, pp. 1–4. Linköping University Electronic Press (2013)
523. Takada, H.: Real-time operating system for embedded systems. In: M. Imai and N. Yoshida (eds.): Tutorial 2 – Software Development Methods for Embedded Systems, Asia South-Pacific Design Automation Conference (ASP-DAC) (2001)
524. Tan, T.K., Raghunathan, A., Jha, N.K.: Software architectural transformations: A new approach to low energy embedded software. Proceedings of Design, Automation and Test in Europe (DATE) pp. 11,046–11,051 (2003)
525. Tanenbaum, A.: Moderne Betriebssysteme. Pearson (2016)
526. Tartler, R., Lohmann, D., Sincero, J., Schröder-Preikschat, W.: Feature consistency in compile-time-configurable system software: facing the Linux 10,000 feature problem. Sixth Conference on Computer Systems (EuroSys) (2011)
527. Tehrani, S.N.: Functional safety and IEC 61508 – a basic guide. http://www.ida.liu.se/~simna73/teaching/SCRTS/IEC61508_Guide.pdf (2004)
528. Teich, J.: Digitale Hardware/Software-Systeme. Springer (1997)
529. Tewari, A.: Modern Control Design with MATLAB and SIMULINK. John Wiley and Sons Ltd. (2001)
530. Texas Instruments Inc.: 6000 multicore DSP + ARM SoC. http://www.ti.com/lsds/ti/processors/dsp/c6000_dsp-arm/66ak2x/overview.page (2020)
531. Thayer, D., Miller, K.: Four UNIX programs in four UNIX collections: seeking consistency in an Open Source icon. In: Proceedings of the 37th Midwest Instruction and Computing Symposium (2004)
532. The Dobelle Institute: Home page. http://www.dobelle.com (no longer accessible) (2003)
533. The MathWorks Inc.: Simulink – simulation and model-based design. http://www.mathworks.com/products/simulink (2021)
534. Thesing, S.: Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models. Pirrot Verlag (2004)
535. Thiele, L.: Design space exploration of embedded systems. Artist Spring School on Embedded Systems, Xi'an, http://www.artist-embedded.org/docs/Events/2006/ChinaSchool/4_DesignSpaceExploration.pdf (2006)
536. Thiele, L.: Performance analysis of distributed embedded systems. In: R. Zurawski (Hrg.): Embedded Systems Handbook, CRC Press, 2006 (2006)
537. Thiele, L. et al.: SHAPES TIK. http://www.tik.ee.ethz.ch/~shapes/dol.html (2009)
538. Thoen, F., Catthoor, F.: Modelling, Verification and Exploration of Task-Level Concurrency in Real-Time Embedded Systems. Kluwer Academic Publishers (2000)
539. Thomas, D.E., Moorby, P.: The Verilog Hardware Description Language. Kluwer Academic Publishers (1991)

540. Tian, T., Shih, C.P.: Software techniques for shared-cache multi-core systems. https://software.intel.com/en-us/articles/software-techniques-for-shared-cache-multi-core-systems (2012)
541. Tiller, M.M.: Modelica by Example. http://book.xogeny.com/ (2016)
542. Tiwari, A., Ballal, P., Lewis, F.L.: Energy-efficient wireless sensor network design and implementation for condition-based maintenance. ACM Trans. Sen. Netw. **3**(1) (2007). DOI 10.1145/1210669.1210670. URL http://doi.acm.org/10.1145/1210669.1210670
543. Tiwari, V., Malik, S., Wolfe, A.: Power analysis of embedded software: A first step towards software power minimization. IEEE Trans. On VLSI Systems pp. 437–445 (1994)
544. Tomasulo, R.M.: An efficient algorithm for exploiting multiple arithmetic units. IBM Journal of Research and Development **11**(1), 25–33 (1967). DOI 10.1147/rd.111.0025
545. Topcuoglu, H., Hariri, S., Wu, M.Y.: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Transactions on Parallel and Distributed Systems **13**(3), 260–274 (2002). DOI 10.1109/71.993206
546. TriQuint Semiconductor Inc.: FAQ 11: What is the MTBF for gallium arsenide devices? http://www.triquint.com/about-us/quality/reliability-faq (2016)
547. Tsai, J., Yang, S.J.H.: Monitoring and Debugging of Distributed Real-Time Systems. IEEE Computer Society Press (1995)
548. Udayakumararan, S., Dominguez, A., Barua, R.: Dynamic allocation for scratch-pad memory using compile-time decisions. ACM Transactions on Embedded Computing Systems (TECS) pp. 472–511 (2006)
549. US Department of Transportation – Federal Aviation Administration: AC 25.1309-1A – System Design and Analysis. https://www.faa.gov/regulations_policies/advisory_circulars/index.cfm/go/document.information/documentID/22680 (1988)
550. V-Modell XT Authors: V-Modell® XT. ftp://ftp.heise.de/pub/ix/projektmanagement/vmodell/V-Modell-XT-Gesamt-Englisch-V1.3.pdf (2006)
551. Vahid, F.: Procedure exlining. Proceedings of the International Symposium on System Synthesis (ISSS) pp. 84–89 (1995)
552. Vahid, F.: Embedded System Design. John Wiley& Sons (2002)
553. Verachtert, W.: Introduction to parallelism. Tutorial at Design, Automation, and Test in Europe (DATE) (2008)
554. Verma, M., Marwedel, P.: Overlay Techniques for Scratchpad Memories in Low Power Embedded Processors. IEEE Transactions on Very Large Scale Integrated Systems **14**(8) (2006)
555. Verma, M., Petzold, K., Wehmeyer, L., Falk, H., Marwedel, P.: Scratchpad sharing strategies for multiprocess embedded systems: A first approach. In: IEEE 3rd Workshop on Embedded System for Real-Time Multimedia (ESTIMedia), pp. 115–120 (2005)
556. Vladimirescu, A.: SPICE user's guide. Northwest Laboratory for Integrated Systems, Seattle (1987)
557. Vogels, M., Gielen, G.: Architectural Selection of A/D Converters. Proceedings of the Design Automation Conference (DAC) pp. 974–977 (2003). DOI 10.1145/775832.776076. URL http://doi.acm.org/10.1145/775832.776076
558. Wägemann, P., Distler, T., Hönig, T., Janker, H., Kapitza, R., Schröder-Preikschat, W.: Worst-case energy consumption analysis for energy-constrained embedded systems. In: 2015 27th Euromicro Conference on Real-Time Systems, pp. 105–114. IEEE (2015)
559. Wainwright, M.J., Jordan, M.I.: Graphical models, exponential families, and variational inference. Foundations and Trends in Machine Learning **1**(1–2), 1–305 (2008). URL http://dx.doi.org/10.1561/2200000001
560. Wandeler, E., Thiele, L.: Real-Time Calculus (RTC) Toolbox. http://www.mpa.ethz.ch/Rtctoolbox (2015)
561. Wang, C., Li, X., Zhang, J., Zhou, X., Nie, X.: MP-Tomasulo: A Dependency-Aware Automatic Parallel Execution Engine for Sequential Programs. ACM Trans. Archit. Code Optim. **10**(2), 9:1–9:26 (2013). DOI 10.1145/2459316.2459320. URL http://doi.acm.org/10.1145/2459316.2459320

562. Wang, P., Sun, G., Wang, T., Xie, Y., Cong, J.: Designing scratchpad memory architecture with emerging STT-RAM memory technologies. In: Proceedings of the International Symposium on Circuits and Systems (ISCAS), pp. 1244–1247 (2013). DOI 10.1109/ISCAS.2013.6572078
563. Wang, Z., Bovik, A.C.: A universal image quality index. IEEE Signal Processing Letters **3**, 81–84 (2002)
564. Wang, Z., Bovik, A.C., Sheikh, H.R., Simoncelli, E.P.: Image quality assessment: From error visibility to structural similarity. IEEE Transactions on Image Processing **13**, 600–612 (2004)
565. Wang, Z., Gu, Z., Yao, M., Shao, Z.: Endurance-Aware Allocation of Data Variables on NVM-Based Scratchpad Memory in Real-Time Embedded Systems. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **34**(10), 1600–1612 (2015). DOI 10.1109/TCAD.2015.2422846
566. Waxman, R., Bergé, J.M., Levia, O., Rouillard, J.: High-Level System Modeling. Springer (1996)
567. Weast, R. (ed.): CRC Handbook of Chemistry and Physics. CRC Press (1977)
568. Wedde, H., Lind, J.: Integration of task scheduling and file services in the safety-critical system MELODY. EUROMICRO ’98 Workshop on Real-Time Systems, IEEE Computer Society Press pp. 18–25 (1998)
569. Weddell, A.S., Magno, M., Merrett, G.V., Brunelli, D., Al-Hashimi, B.M., Benini, L.: A survey of multi-source energy harvesting systems. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE ’13, pp. 905–908. EDA Consortium, San Jose, CA, USA (2013). URL http://dl.acm.org/citation.cfm?id=2485288.2485505
570. Wegener, I.: Branching programs and binary decision diagrams – Theory and Applications. SIAM Monographs on Discrete Mathematics and Applications (2000)
571. Wehmeyer, L., Marwedel, P.: Fast, Efficient and Predictable Memory Accesses. Springer (2006)
572. Weiser, M.: The computer for the 21st century. Mobile Computing and Communications Review **3.3**, 3–11 (1999)
573. Wellings, A.: Concurrent and Real-Time Programming in Java. Wiley (2004)
574. Wikipedia: Betriebssicherheit. https://de.wikipedia.org/wiki/Betriebssicherheit (2021)
575. Wikipedia: Energy harvesting. https://en.wikipedia.org/wiki/Energy_harvesting (2021)
576. Wikipedia: Heat capacity. https://en.wikipedia.org/wiki/Heat_capacity (2021)
577. Wikipedia: Netzdiagramm. https://de.wikipedia.org/wiki/Netzdiagramm (2021)
578. Wikipedia: OSEK. https://en.wikipedia.org/wiki/OSEK (2021)
579. Wikipedia: Radio clock. https://en.wikipedia.org/wiki/Radio_clock#List_of_radio_time_signal_stations (2021)
580. Wikipedia: Roll-Nick-Gier-Winkel. https://de.wikipedia.org/wiki/Roll-Nick-Gier-Winkel (2021)
581. Wikipedia: Safety integrity level. https://en.wikipedia.org/wiki/Safety_integrity_level (2021)
582. Wikipedia: Structured systems analysis and design method. http://en.wikipedia.org/wiki/Structured_Systems_Analysis_and_Design_Methodology (2021)
583. Wikipedia: Thermal conductivity. https://en.wikipedia.org/wiki/Thermal_conductivity (2021)
584. Wikipedia: Thermal design power. https://en.wikipedia.org/wiki/Thermal_design_power (2021)
585. Wikipedia: Zeno’s paradoxes. https://en.wikipedia.org/wiki/Zeno’s_paradoxes#Achilles_and_the_tortoise (2021)
586. Wilde, D.K.: A library for doing polyhedral operations. Tech. Rep. 785, Tech. Rep., IRISA, Rennes (1993)
587. Wilhelm, R.: Determining bounds on execution times. In: Zurawski, R. (Ed.): Embedded Systems Handbook, CRC Press, 2006 (2006)
588. Willems, M., Bürsgens, V., Keding, H., Grötker, T., Meyr, H.: System level fixed-point design based on an interpolative approach. Proceedings of the Design Automation Conference (DAC) pp. 293–298 (1997)
589. Wilton, S., Jouppi, N.: CACTI: An enhanced access and cycle time model. Int. Journal on Solid State Circuits **31**(5), 677–688 (1996)

590. Wind River: VxWorks. http://www.windriver.com/products/vxworks (2020)
591. Wind River: Web pages. http://www.windriver.com (2020)
592. Winkler, J.: The CHILL homepage. http://psc.informatik.uni-jena.de/languages/chill/chill.htm (2019)
593. Wolf, W.: Computers as Components. Morgan Kaufmann Publishers (2001)
594. Wolsey, L.: Integer Programming. Jon Wiley & Sons (1998)
595. Wong, C., Marchal, P., Yang, P., Prayati, A., Catthoor, F., Lauwereins, R., Verkest, D., Man, H.D.: Task concurrency management methodology to schedule the MPEG4 IM1 player on a highly parallel processor platform. Proceedings of the International Symposium on Hardware-Software Codesign (CODES) pp. 170–177 (2001)
596. Woodhouse, D.: Jffs: The journalling flash file system. Red Hat, Inc. White Paper (2001)
597. ws4d: Web services for devices. http://www.ws4d.org (2019)
598. Xilinx: MicroBlaze processor reference guide. http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf (2008)
599. Xilinx: Device reliability report – first half 2019. http://www.xilinx.com/support/documentation/user_guides/ug116.pdf (2019)
600. Xilinx Inc.: UltraScale+ FPGAs – Product Tables and Product Selection Guide. http://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf (2020)
601. Xilinx Inc.: UltraScale Architecture Configurable Logic Block. http://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf (Feb. 2017)
602. Xilinx Inc.: UltraScale Architecture and Product Overview. http://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf (June 2019)
603. XMOS Ltd.: Home page. http://www.xmos.com/ (2019)
604. Xu, J., Parnas, D.L.: On satisfying timing constraints in hard real-time systems. IEEE Transactions on Software Engineering **19**, 70–84 (1993)
605. Xu, Q., Huang, L., Yuan, F.: Lifetime reliability-aware task allocation and scheduling for MPSoC platforms. Proceedings of Design, Automation and Test in Europe (DATE) pp. 51–56 (2009)
606. Xue, J.: Loop tiling for parallelism. Kluwer Academic Publishers (2000)
607. Yodaiken, V.: Adding real-time support to general purpose operating systems. US Patent 5,995,745 (1997)
608. Yodaiken, V.: Real-time applications in real-time linux. In: USENIX'99 Tutorial 3: Linux on the Edge (1999)
609. Young, S.: Real Time Languages – Design and Development –. Ellis Horwood (1982)
610. Zanini, F., Atienza, D., Jones, C.N., Benini, L., De Micheli, G.: Online thermal control methods for multiprocessor systems. ACM Trans. Des. Autom. Electron. Syst. **18**(1), 6:1–6:26 (2013). DOI 10.1145/2390191.2390197. URL http://doi.acm.org/10.1145/2390191.2390197
611. Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R.P., Mao, Z.M., Yang, L.: Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In: Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, pp. 105–114. ACM (2010)
612. Zhang, W., Ding, Y.: Hybrid SPM-cache architectures to achieve high time predictability and performance. In: Proceedings of the Conference on Application-Specific Systems, Architectures and Processors (ASAP), pp. 297–304 (2013). DOI 10.1109/ASAP.2013.6567593
613. Zhuo, C., Sylvester, D., Blaauw, D.: Process variation and temperature-aware reliability management. Proceedings of Design, Automation and Test in Europe (DATE) (2010)
614. Zurawski, R. (ed.): Embedded Systems Handbook. CRC Press (2006)

Sachverzeichnis

5G, 196

A/D-Wandler, 150
  Flash - ~, 151
  Delta-Sigma ~, 155
  faltender ~, 155
  integrierende ~, 154
  mit sukzessiver Approximation, 153
Abbildung von Anwendungen, 321
Abhängigkeitsgraph, 40
Abschwächer, 101
Abstraktion, 32
Abtastkriterium, 149, 200
Abtastrate, 149
Abtasttheorem, 149, 200
Abtastung, 149
  adaptive ~, 14
AC-AMC 25.1309, 304
Acatech Report, 3
accuracy, 282
ACID-Eigenschaft, 257
Ada, 43, 44, 78, 109, 123, 132
Address Translation Memories (ATM), 434
Adresse
  reale ~, 433
  virtuelle ~, 433
Advanced Vector Extensions (AVX), 170, 418
Aktor, 75, 79, 83, 130
Aktuator, 18, 194, 195, 206
ALAP, 360
Algorithmus
  Critical-Path-On-a-Processor (CPOP) ~, 368
  Heterogeneous-Earliest-Finish-Time (HEFT) ~, 365
  fpEDF ~, 354
Aliasing, 149, 200, 422
AMBA-Bus, 194
ambient intelligence, 1
Analysierbarkeit, 130
Anforderungen, 31
Anti-Aliasing, 149
Anwendungsgebiet, 23, 35
API, 225, 228
Application Programming Interface (API), 253, 255, 256
Application-Specific Integrated Circuit (ASIC), 157, 212
approximate computing, 278
arbitrary deadline task, 327, 344
Arithmetik
  Festkomma- ~, 387
  Gleitkomma- ~, 387
  Sättigungs- ~, 168
ARM®, 174, 183, 194
array folding, 386
ARTIST Richtlinien, viii
ASAP, 358
Auflösung, 152
Ausdrucksstärke, 130, 131
Ausführbarkeit, 35
Ausfall, 306
Ausfallrate, 307
Auslastung, 339, 341, 343, 344, 348–350, 352–358
  Task- ~, 339
Ausnahme, 34, 56, 58
AUTOSAR, 19, 323

back annotation, 404
Badewannenkurve, 308
Ball
  springender ~, 12, 50
Basisblock, 271, 272, 396, 397

© Der/die Autor(en) 2021

P. Marwedel, *Eingebettete Systeme*, https://doi.org/10.1007/978-3-658-33437-6

Batterie, 208
bedingte Ausführung, 164, 172
Bedingungs/Ereignis-Netz, 86
Befehlssatz-Architektur, 128
Befehlssatz-Ebene, 128
Benutzerschnittstelle, 19
Best Case Execution Time (BCET), 268, 270, 275
best effort, 20
Betriebssicherheit, 10, 303
Betriebssystem
  eingebettetes ~, 222
Betriebssystemkern, 228
bin packing, 347
Binary Decision Diagram (BDD), 317, 318
Bindung, 371
Bit Error Ratio (BER), 281
Bitfehlerverhältnis, 281
Bluetooth, 196
body biasing, 409
bouncing ball, 12, 50
boundary scan, 421
branch delay penalty, 172
Broadcast, 65, 67, 68

Cache, 169, 185, 187, 189, 267, 269, 381–383, 393
CACTI, 184, 189, 285
CENELEC 50128, 304
Charge-Coupled Device (CCD), 144
Chill, 127
code size, 16
code size efficiency, 16, 162
Codegröße, 16, 397
ColdFire, 163
Coloured Petri Net (CPN), 93
column-major order, 381
Commercial Off-The-Shelf (COTS), 158
Communicating Finite State Machine (CFSM), 43, 53
Communicating Sequential Processes (CSP), 122, 135
Compiler, 384, 392, 394
  energiegewahrer ~, 393
  retargierbarer ~, 393
Computer
  unsichtbarer ~, 1
Configurable Logic Block (CLB), 182
constrained deadline task, 327, 344
CORBA®, 254
Cortex®, 174
critical instant theorem, 341, 357
Critical-Path-On-a-Processor (CPOP) algorithm, 365
CSA-Theorie, 98
CSMA/CA, 194
CSMA/CD, 194
CSP, 122
Curriculum, 22
Cyber-Physical System (CPS), vii, 2, 4, 9, 50
cyber-physikalisches System, vii, 3, 32, 43, 142, 191, 253, 266, 302, 314
CyPhy-Interface, 141, 143, 146, 196

D/A-Wandler, 153, 197, 198
DAG, 336
dark silicon, 15, 179, 294, 297, 409
Datenanalyse, 8, 17
Datenfluss, 75
  -modellierung, 75
  Cyclo-Static Data Flow (CSDF), 82
  Homogeneous Synchronous Data Flow (HSDF), 82, 131
  Synchronous Data Flow (SDF), 78
  data flow, 75
  synchroner ~, 78, 131
Deadline, 131, 227, 339, 340, 343, 389, 407
deadlock, 38, 136, 235
Deep Neural Network (DNN), 180
delta cycle, 118
demand paging, 434
Demand-Bound-Function (DBF), 344
density, 344
Deployment-Diagramm, 133
design flow, 25
Design for Testability (DfT), 419
design framework, 24
Design Space Exploration (DSE), 323, 374
determinate, 64
deterministisch, 14, 53, 64, 82, 118, 122, 124, 226
  nicht-~, 86, 89
Device Under Test (DUT), 414
Devices Profile for Web Services (DPWS), 256
Dezibel, 157, 279
Dhall-Effekt, 354, 357
Diagnosefähigkeit, 191
Diagramme
  UML-, 134
Dichte, 344, 358
Dienst, 306
Differentialgleichung, 43, 50
Digital European Cordless Telecommunications (DECT), 196
Digital-Analog-Wandler, 143, 198
Digital-to-Analog Converter (DAC), 143, 198
dining philosophers' problem, 93
Diskretes Ereignis, 44, 96, 135

Dispatcher, 327
display, 196
Distributed Operation Layer (DOL), 369, 372
DO-178B, 304
DO-254, 304
DSP, 165, 167
dynamic scheduling (of instrutions), 45, 174
Dynamische Leistungsverwaltung (DPM), 409
Dynamische Spannungsskalierung (DVS), 162, 405, 406

Earliest Deadline First (EDF), 238, 329, 332, 343, 375
Earliest Due Date (EDD), 331
Echtzeit
  -CORBA, 255
  -POSIX, 255
Echtzeitbetriebssystem, 28, 224, 226, 228, 391
Echtzeitdatenbanken, 256
Echtzeitfähigkeit, 169
Echtzeitkern, 228
Echtzeitverhalten, 190
eCos, 224
ECU, 323
Effizienz, 36, 190
  Codegrößen-~, 162
  Laufzeit-~, 16, 165, 187
Eingabe, 34, 36, 40, 41, 56, 66, 70, 110, 122
eingebettetes System, 2, 9, 20, 28, 32, 60, 131, 134, 141, 223, 321, 379, 413
Elektromagnetische Kompatibilität (EMC), 314
Emulation, 315
Endlicher Automat, 39, 44, 53, 55, 58, 69, 135, 317
Energie, 14, 15, 157, 163, 170, 206, 283, 286, 294, 323, 370, 384, 393, 399, 409
Energieeffizienz, 158, 186, 211
Energieerzeugung, 207
Energiemodell, 283, 393
Energiequelle, 143
Energiespeicher, 143
energy, 15, 19, 318
energy awareness, 211
energy efficiency, 158, 186, 211
energy harvesting, 207
entirely time triggered (TT systems), 327
Entwurf für Testbarkeit, 419
Entwurfsfluss, 23, 142, 262, 322
Entwurfsphasen
  frühe ~, 45
Entwurfsziel, 323
EPIC, 170
Ereignis, 34, 84, 86, 116
  diskretes ~, 96
  sporadisches ~, 346
Erfüllbarkeit, 373
ERIKA, 240
error, 306
Estelle, 74
Esterel, 67
European Installation Bus (EIB), 195
European Train Control System (ETCS), 5
Evaluierung, 261
evolutionärer Algorithmus, 371, 373
extreme value theory (EVT), 273
Extremwerttheorie, 273

F-measure, 283
F1 score, 283
failure, 306
Failure Mode and Effect Analysis (FMEA), 313
Failure unIT (FIT), 308
False Negative (FN), 282
fault, 306
fault tree, 312
Fehler, 306
  -überdeckung, 417
  -injektion, 418
  -modell, 415
  -simulation, 417
  -toleranz, 191
Fehlerbaumanalyse, 312
Feldtest, 413
Fertigungsanlagen, 5
Festkommaarithmetik, 169
Field Programmable Gate Array (FPGA), 181, 369
field test, 413
FIFO, 74, 76
  in SDL, 70
finishing time, 325
Finite State Machine (FSM), 68
FIT, 308
flash memory, 248
Flash-Speicher, 402
FlexRay™, 218
Fließband-A/D-Wandler, 154
flipped classroom, 21, 29, 135, 216, 318, 375, 410, 423
Folien, 21
formale Verifikation, 316
Fourier-Approximation, 146
Fouriersches Gesetz, 295
FPGA, 316, 369

G-EDF, 353
G-RM, 356

Gajskis Y-Diagramm, 26
Ganzzahlige Lineare Programmierung, 396, 408, 425
Gebäude
intelligentes ~, 6
gegenseitiger Ausschluss, 41, 84, 231
gem5, 315
Genauigkeit, 282
Gesetz von Peukert, 209
Gesundheitswesen, 7
Gewicht, 16
Gibbs-Phenomen, 147
Gleichung von Black, 294, 309
Gleitkommarechnungen, 168
Global Positioning System (GPS), 18
globale Erwärmung, 7, 15, 17
GPGPU, 176
GPS, 227
Granularität, 41
Graphics Processing Unit (GPU), 178, 183, 252
Graphikprozessor, 176
GSM, 194

hard core, 183
hard real-time, 12
Hardware, 141
sichere ~, 214
Hardware in der Schleife, 142
Hardware-Abstraktionsschicht, 253
hazard, 312
heat capacity, 297
Herausforderungen, 9
Heterogeneous-Earliest-Finish-Time (HEFT) algorithm, 365
Hierarchie, 32
-Blätter, 57, 72
in SDL, 71
in StateCharts, 55
High-Level Optimierungen
High-Level-Optimierungen, 380
HOPES, 373
HSDF, 131
HSPA, 196
Hyperperiode, 324, 343

I^{2}C, 195
IEC 62279, 304
IEEE 802.11, 196
IEEE 1076, 109
IEEE 1164, 119, 138
IEEE 1364, 120
IEEE 1394, 196
IEEE 802.11, 196

IKT, ix, 3, 6, 8, 9
ILP, 373, 396
implicit deadline task, 327
Implicit Path Enumeration Technique (IPET), 273
Industrie 4.0, 4, 5
Inertial Measurement Units (IMUs), 143
inertiale Messeinheit, 143
Informations- und Kommunikationstechnologie (IKT), vii
Informationssicherheit, 9, 214, 302
Instruction Set Architecture (ISA), 174
Integer Linear Programming (ILP), 363, 400, 408, 425
Intellectual Property (IP), 177, 252, 253
Internet der Dinge, vii, 4, 8, 9, 145, 190, 206, 213, 214, 243, 253
Internet of Things (IoT), vii, 49, 143, 315
interrupt, 225, 226
ISO 26262, 304
ISO 9001, 303
ITRON, 228

Java, 126
Job, 231, 324
unabhängiger ~, 330
JTAG, 421

künstliche Intelligenz (KI), vii
Kahn-Prozessnetzwerk, 122, 130, 135, 373
Kanal, 70
kausale Abhängigkeit, 39
kinetic battery model, 210
Kirchhoff's laws, 428
Kirchhoffsche Regeln, 198, 427, 429
Kiviat diagram, 264
Kiviat-Diagramm, 264
Knapsack
0/1-Multiples-~Problem (MKP), 347
Kommunikation, 32, 189
blockierende ~, 122
Kommunikationsbibliotheken, 125
komponentenbasierter Entwurf, 32
Komprimierung, 163
Wörterbuch-basierte ~, 165
Konfiguration
Link-Zeit-~, 228
Konfigurierbarkeit, 223
Korrektklassifikationsrate, 282
Kosten, 16
-modell für Energie, 286
-modell für Ganzzahlige Lineare Programmierung, 425
der Verdrahtung, 195

der Kommunikation, 190
eines zweiten Befehlssatzes, 165
für Gleitkomma-Arithmetik, 168
von ASICs, 157
von Schäden, 312
KPN, 122, 130, 373
Kriterien
Optimierungs-~, 265
kritischer Abschnitt, 42, 231
Kühlkörper, 143
Künstliche Intelligenz, 8
kurze Vektorbefehle, 170

Ladungstransportspeicher, 144
Laufzeit, 16
laxity, 324, 334
Least Slack Time First (LST), 334
Lehrplan, 20
Leistung, 283, 393
Leistungsmodelle, 283
Lesbarkeit, 35
Leserkreise, x
Levi-Simulation, 48, 77, 136, 195, 233, 235, 258
life cycle assessment (LCA), 16
LIN (Local Interconnect Network), 195
Linear Feedback Shift Register (LFSR), 421
Lineare Programmierung (LP), 426
lineare Transformation, 148
Linux Betriebssystem, 243
Liu-and-Layland task, 327
Logik
Aussagen-~, 316
mehrwertige ~, 97
Prädikaten-~ höherer Stufe, 317
rekonfigurierbare ~, 181
Logistik, 5
Lokalität, 383
räumliche ~, 381
Lokalität von Speicherzugriffen, 381
Long-Term Evolution (LTE), 196
loop fission, 382
loop fusion, 382
loop splitting, 384
loop tiling, 400
LOTOS, 133
lp_solve, 426
Lustre, 67

M-aktiviert, 89
Makespan, 328, 330, 360, 364, 367, 369
Mali™, 177, 178, 183
manufacturing test, 413
MAP, 195
MAPS, 374
Maschenregel, 431
Maschine
virtuelle ~, 230, 352
Maschinelles Lernen, 17, 21, 145, 180, 285, 289, 404
mass density, 298
MATLAB, 82, 127
maximale Verspätung, 328, 331, 332, 335, 336, 346
maximum lateness, 328
may-Analyse, 319
Mean-Absolute Error (MAE), 279
Mean-Squared Error (MSE), 278
Media Oriented Systems Transport (MOST), 196
Mehrkern-Prozessor
homogener ~, 173
Meltdown, 214
Memory Management Unit (MMU), 178, 250, 402, 434
Memory Protection Unit (MPU), 225
memory wall, 185
message passing, 43, 44
Message Passing Interface (MPI), 125
Message Sequence Charts (MSC), 47
middleware, 253
MIMOLA, 109
Model of Computation (MoC), 39, 44, 136
Modelica, 44
Modell, 31
auf Layout-Ebene, 129
auf Schalter-Ebene, 129
Berechnungs-~, 36
diskretes Ereignis-~, 43
modellbasierter Entwurf, 45
Modi, 82
Modular Performance Analysis, 274
Module Charts, 66
Mooresches Gesetz, 244
MPSoC, 178, 374
MSC, 47
MTBF, 310
MTTF, 309
MTTR, 310
multi-core processor, 248
Multi-Processor System on a Chip (MPSoC), 178, 179, 183, 217
multi-project wafer (MPW), 157
multikriterielle Optimierungstechniken, 262
Multimediaprozessor, 169
Multiply-Accumulate-Befehle, 167
must-Analyse, 319
Mutex, 231

Nachfolger, 40
Nachrichtenaustausch, 107
Nanoprogrammierung, 165
Nebenläufigkeit, 32, 38
Nebenläufigkeit von Tasks, 388
Netz
  einfaches ~, 87
nicht arbeitserhaltend, 336, 350, 352
Niedrigenergiehaus, 7
NP-hart, 317, 330, 336
Nullenergiehaus, 7
Nyquist-Frequenz, 149

Objektorientierung, 35
Observer, 37, 81, 119
Occam, 122
OMNET++, 315
op-amp, 429
Open-Collector-Schaltung, 99
Openmodelica, 13
OpenMP, 126
Operationsverstärker, 198, 427, 429
Optimierung, 379, 384, 393, 394, 425, 426
OSEK, 228, 254
out-of-order scheduling, 76

page table, 433
Paging, 433
Parallelisierung, 374
Pareto-Front, 263
Pareto-Optimalität, 263, 264, 318, 330, 373
Peak-Signal-to-Noise Ratio (PSNR), 279
Pearl, 127
Performanz, 265
Periode, 324, 343, 376
periodisches *Scheduling*, 41
Personal Computer (PC), 1, 2, 38
pervasive computing, 1
Petrinetz, 84, 390
Pfadlänge, 361
Pinedo-Notation, 325
plattformbasierter Entwurf, 141, 322
plug-and-play, 39
Portierbarkeit, 35
POSIX Threads, 255
power wall, 184
Prädikat/Transitions-Netz, 92
Prädikatenlogik erster Stufe, 317
Präzedenzrelation, 40, 326, 336, 366
precedence constraint, 326, 367
precision, 282
predicated execution, 164, 172
preemption, 238, 326
prefetching, 383

Prioritätsencoder, 152
Prioritätsumkehr, 231, 232
Prioritätsvererbung, 233–235
Priority Ceiling Protocol (PCP), 236
Priority Inheritance Protocol (PIP), 233
Prozedur-Exlining, 165
Prozess, 36, 222, 223
Prozessnetz, 39
Prozessor, 158, 393
  Cell ~, 373
  digitaler Signal- ~, 165
  heterogener ~, 365
  Mehrkern- ~, 172, 173, 325
Prozessoren
  heterogene ~, 326
  homogene ~, 325
  uniforme ~, 325
Ptolemy, 44, 135, 374
Pulsbreitenmodulation (PWM), 143, 204, 205

Qualitätsmetriken, 278
Quality of Experience (QoE), 283
Quality of Service (QoS), 283
Quantisierungsrauschen, 156

radio frequency identification (RFID), 5, 145
rapid prototyping, 315
reactive system, 18
Reaktionszeit, 326
Real-Time Calculus (RTC), 265, 274
Real-Time Operating System (RTOS), 28, 226
Realzeitkalkül, 274
reasonable allocation (RA) algorithm, 348
reasonable allocation decreasing (RAD) algorithm, 348
recall, 282
Rechnen
  näherungsweises ~, 278
Rechner
  verschwindender ~, 19
Regelschleife, 13, 142
Register-Transfer-Ebene, 128
Reihenfolgebeschränkung, 40, 336, 346, 365
relative *Deadline*, 324
release time, 324, 326
reliability, 305, 307
Rendez-Vous, 122, 123
repairability, 309
resilience, 305
Ressourcen-gewahr, 14
Ressourcenallokation, 229
RFID, 145
richtig positiv, 281
Risiko, 302

robotics, 6, 28
Robotik, 6, 28
Robustheit, 190, 191
Root-Mean-Squared Error (RMSE), 279
row-major order, 380, 383
RT-Linux, 229
Rucksackproblem, 396
Rückkopplung, 14, 154, 430

Sättigungsarithmetik, 169
safety, 10, 313
sample-and-hold circuit, 145
SAT, 373
Scade, 67
scan design, 419
scan path, 420
*Scan*pfad-Entwurf, 420
Schaden, 312
schedulability tests, 330
schedulable, 330
Scheduling
  ~für abhängige Jobs, 365
  ~mit Ganzzahliger Programmierung, 369
  ~mit Reihenfolgebeschränkung, 336
  ~mit expliziter *Deadline*, 344
  ~ohne Verdrängung, 335
  ~von Maschinenbefehlen, 394
  ~, 227, 321, 329, 330
  As-Late-As-Possible (ALAP) Scheduling, 360
  As-Soon-As-Possible (ASAP) Scheduling, 358
  Deadline Monotonic (DMS) Scheduling, 345
  EDF(k) Scheduling, 355
  Earliest Deadline First (EDF) Scheduling, 343, 369
  Earliest Deadline First Zero Laxity (EDZL) Scheduling, 355
  G-EDF Scheduling, 353
  Global Rate Monotonic (G-RM) Scheduling, 356
  Latest Deadline First (LDF) Scheduling, 337
  List Scheduling, 361
  Multiprozessor- Scheduling, 346
  RMZL Scheduling, 358
  Rate Monotonic Scheduling, 339, 369, 376
  dynamisches Scheduling, 374
  global Scheduling, 350
  hybrides Scheduling, 374
  optimales Scheduling, 338
  partitioniertes Scheduling, 347
  periodisches Scheduling, 338
  pfair Scheduling, 351
  statisches Scheduling, 328
Schleife
  abrollen, 381
  blockweise Verarbeitung, 382
  kachelweise Verarbeitung, 382
Schleifenaufspaltung, 382, 384
Schleifenfusion, 382
Schleifeninstruktion ohne Mehraufwand, 382
Schleifentransformation, 380
Schlupf, 334, 355, 405
Schutz, 225
scratchpad allocation, 400
Scratchpad-Speicher (SPM), 185, 188, 394, 396, 398, 399, 401
SDF, 78, 133, 135
SDL, 68, 74, 132
Secure Socket Layer (SSL), 216
security, 9, 214, 302
Seitenadressierung, 433
Seitenfehler, 434
Selbsttestprogramm, 416
select-Anweisung, 119, 124
Semantik
  StateMate ~, 61
  von SDL, 70
Semaphore, 231, 236
Sensitivität, 282
Sensor, 18, 142
  Beschleunigungs ~, 143
  Bild ~, 144
  biometrischer ~, 144
sensor network, 143
Sensoren, 143
Sensornetzwerk, 213
Sequenzdiagramm, 47
shared memory, 222
Sicherheit, 9, 191, 253, 254
Signal
  assymmetrisches ~, 192
Signal-Rausch-Abstand (SNR), 156, 388
Signalübertragung
  differentielle ~, 192
  symmetrische ~, 192
Signalzuweisung, 97, 111, 112
Signaturanalyse, 421
SimpleScalar, 315
Simulation, 313
  bitgenaue ~, 128
  zyklengenaue ~, 128
Simulink, 82, 374
Single Instruction, Multiple Data (SIMD), 170
Sinussignal, 147
slack, 324, 334, 356, 358
slides, 21
smart grid, 6

SNR, 156
SoC, 163
soft core, 183
SpecC, 25, 105
specific heat, 297
specificity, 283
Spectre, 214
Speicher, 184
  -hierarchie, 185
  flüchtiger ~, 181
  Flash ~, 240
  hierarchie, 394
  persistenter ~, 181, 184
Speicherschutz, 434
Spezifikationssprachen, 31
Spezifizität, 283
Spinnennetzdiagramm, 264
SPM, 394–396, 398, 399
Sporadic Task Server, 346
Sprache, 31
  aktorbasierte ~, 83
  synchrone ~, 66
Störung, 306
stack resource policy (SRP), 238
state diagram, 34
StateCharts, 55, 136
StateMate, 61
Stellen/Transitions-Netze, 87
STEP 7, 67, 127
Steuerspeicher mit zwei Ebenen, 165
Stoß
  teilelastischer ~, 12, 50
Stoßzahl, 12, 51
Streaming SIMD Extensions (SSE), 170
stuck-at-error, 415
sukzessive Approximation, 153
Symbole
  für Gatter, 63
  für Speicher, 163
SymTA/S, 278
Synchronisation, 32
Synchronous Data Flow (SDF), 131
Synthese, 27
System
  dediziertes ~, 19
  hybrides ~, 12
  reaktives ~, 135
System on a Chip (SoC), 16, 163
SystemC, 107, 127, 374
SystemCodesigner, 373
Systemebene, 127
Systemsoftware, 221
SystemVerilog, 121

TAI, 227, 257
Task, 36, 323
  aperiodische ~, 324
  periodische ~, 324, 326, 338, 339, 343, 346
  sporadische ~, 324, 346
task set, 324
Task-Descriptor List (TDL), 327
Task-Graph, 39
  Aufteilen von Knoten, 389
Task-Menge, 324
TDMA, 194, 369
Temperaturmanagement, 409
Tensor Processing Unit (TPU), 180
Terminierung, 36
Test, 413
  notwendiger ~, 330
Testbarkeit, 419
Testmuster, 414
Testmustererzeugung, 415
  pseudo-zufällige ~, 423
thermal capacitance, 297
thermal capacity, 297
thermal conductance, 296
Thermal Design Power (TDP), 297
Thermische Modelle, 294
thermischer Widerstand, 296
Thread, 36, 38, 222, 223
THUMB, 164
Tiefpassfilter, 150, 203
Time Division Multiple Access (TDMA), 193
Time-Triggered Protocol (TTP), 195
Timer, 33, 60, 72
timing analysis, 268
TPG, 415
Transaction-Level Modeling (TLM), xi, 103
Transaktionsebene
  Modellierung auf ~, 128
Translation Look-Aside Buffer (TLB), 434
Triplet-Notation, 330
True Positive (TP), 281

ubiquitous computing, 1
UML, 74, 132
UML-Profil, 134
Unified Modeling Language (UML), 46, 94, 133
Universal Plug-and-Play (UPnP), 256
Universal Serial Bus (USB), 193
Unterbrechung, 225
UPPAAL, 55, 318
use case, 46
UTC, 227, 257
utilization, 354, 357

V-Modell, 25
Validierung, 261, 262
Verarbeitungseinheiten, 157
Verarbeitungsleistung, 265
Verarmungstransistor, 101
Verdrängung, 232, 234, 236–239, 241, 242, 326, 329, 331, 332, 335, 336, 338, 340–343, 346, 350
Verfügbarkeit, 11, 310
Verhalten
  deterministisches ~, 63, 64, 67, 108
  Echtzeit-~, 193
  nicht-funktionales ~, 36
Verilog, 120
Verlässlichkeit, 9, 36, 265, 302
Verstärkung, 429
Vertraulichkeit, 10, 214, 302
Very Long Instruction Word (VLIW)-Prozessor, 170, 215
VHDL, 64, 109
  VHDL
    Architecture, 110
  VHDL
    Entity, 110
  VHDL
    Port Map, 112
VHDL-AMS, 127
virtuelle Masse, 431
VLIW, 170
volumetric heat capacity, 298
Voraussetzungen, 21
Vorgänger, 40
Vorhersagbarkeit, 254, 267, 346
Vorladen von Leitungen, 102
VxWorks, 224, 228

Wärmekapazität, 297
Wärmeleitfähigkeit, 295
Wärmeleitvermögen, 296
Wörterbuch, 165
wait, 115
Wartbarkeit, 11, 191, 309
Wattch, 287
WCET, 267–270, 273
WCET-gewahrer Compiler WCC, 404
wear-out, 402
weiterführende Lehrveranstaltungen, 22
Wind River Systems, 230
Wireless Local Area Network (WLAN), 196
work conserving, 336, 350
working set, 399
Worst Case Energy Consumption (WCEC), 294
Worst Case Execution Time (WCET), 16, 169, 267, 269–271, 275, 324
Worst Case Power Consumption (WCPC), 294

Y-Diagramm, 26

Z-Sprache, 133
Zeit, 40, 48, 127
  -Information, 40
  -schranken, 33
  -verhalten, 32
Zeitgeber, 33, 60, 72, 73, 327
zeitgesteuerte Automaten, 54
Zeitpunkt
  kritischer ~, 341, 357
Zeitschranke
  harte ~, 12, 326
Zeno-Effekt, 12, 13
zero-overhead loop instructions, 167
Zielfunktion, 328, 329, 402, 403
Zustand
  Basis-~, 56
  History-~, 57
  ODER-Super-~, 56
  Standard-~, 57
  Super-~, 56
  UND-Super-~, 58
Zustandsdiagramm, 53
Zuverlässigkeit, 9, 306, 307, 312, 313