Echtzeitfähige Ablaufplanung für simultan mehrfädige Prozessoren

Dissertation
zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs
der Fakultät für Angewandte Informatik der Universität Augsburg

eingereicht von
Dipl.-Inf. Jörg Mische

Erstgutachter:	Prof. Dr. rer. nat. Theo Ungerer
Zweitgutachter:	Prof. Dr.-Ing. Rudi Knorr

Tag der mündlichen Prüfung: 2. Dezember 2011

für Papa

Abstract

This thesis presents the requirements for a Simultaneous Multithreaded (SMT) processor that supports hard real-time capabilities. The microarchitecture of a single threaded superscalar in-order processor is enhanced to support simultaneous multithreading with one completely isolated thread. This thread is executed as if it were the only thread in a single threaded system. Hence established methods for Worst Case Execution Time (WCET) analysis can be applied.

A hardware module, directly connected to the issue stage of the processor pipeline, provides sophisticated scheduling algorithms for a nearly arbitrary number of threads. For hard real-time scheduling, not the popular Earliest Deadline First (EDF) algorithm is used, but another likewise optimal algorithm, based on time slicing. Soft real-time threads are scheduled by controlling the number of executed instructions, given by Instructions Per Cycle (IPC). Furthermore, threads without real-time requirements are scheduled by a round robin algorithm and the response time of aperiodic requests (interrupts) is minimised. For the later, no costly server thread is needed to calculate the remaining time, the time is available automatically.

To evaluate the proposed architecture, a SystemC simulator and a FPGA prototype were developed. This so-called CarCore supports an instruction set that is binary compatible to the Infineon TriCore processor. Although the single threaded execution in CarCore reaches only half the speed of the TriCore, the total throughput with multithreaded execution is comparable to the TriCore throughput. In particular the concurrent execution of hard and soft real-time threads can seriously increase the processor throughput compared to a pure hard real-time system.

Kurzfassung

In dieser Arbeit wird dargestellt, wie ein simultan mehrfädiger (SMT) Prozessor aufgebaut sein muss, um harte Echtzeitanforderungen zu erfüllen. Es werden Veränderungen an der Mikroarchitektur beschrieben, die es erlauben, einen einfädigen Superskalarprozessor mit In-Order-Ausführung zu einem simultan mehrfädigen Prozessor zu erweitern, bei dem ein Programmfaden völlig isoliert von den anderen abläuft. Dadurch verhält sich dieser Programmfaden als würde er allein auf einer einfädigen System ausgeführt und gängige Methoden zur Laufzeitanalyse und Berechnung der Worst Case Execution Time (WCET) können angewendet werden.

Ein Hardware-Modul, das direkt auf der Zuordnungsstufe der Prozessor-Pipeline aufsetzt, erlaubt es, komplexe Scheduling-Algorithmen für eine nahezu beliebige Anzahl von Programmfäden zu ermöglichen. Um harte Echtzeitanforderungen zu erfüllen wird nicht der etablierte Earliest Deadline First (EDF) Algorithmus verwendet, sondern ein auf Zeitschlitzen basierender Algorithmus, der jedoch ebenfalls optimal ist. Bei Programmfäden mit weichen Echtzeitanforderungen wird versucht, eine vorgegebene Ausführungsrate, gemessen in Instruktionen pro Takt (Instructions Per Cycle, IPC) zu erreichen. Weiterhin ist es möglich, Programmfäden ohne Echtzeitanforderungen durch einen Round-Robin-Algorithmus gleichmäßig auszuführen und aperiodische Anfragen (Interrupts) in kurzer Zeit zu beantworten. Für letzteres ist kein aufwändiges Serverprogramm zur Berechnung der nutzbaren Rechenzeit nötig, die Zeit steht vielmehr automatisch zur Verfügung.

Um die vorgeschlagene Architektur zu evaluieren, wurde ein SystemC-Simulator und ein FPGA-Prototyp erstellt, deren Befehlssatz binärkompatibel zum Infineon TriCore 1 sind. Dieser sogenannte CarCore-Prozessor ist bei einfädiger Ausführung zwar nur halb so schnell wie ein einfädiger TriCore, bei mehrfädige Ausführung ist der Gesamtdurchsatz jedoch vergleichbar. Insbesondere durch die gleichzeitige Ausführung von Programmfäden mit harten und weichen Echtzeitanforderungen kann der Prozessordurchsatz im Vergleich zu reinen harten Echtzeitsystemen erheblich gesteigert werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ASIC	Application-Specific Integrated Circuit
CHS	Configurable Hardware Scheduler
CISC	Complex Instruction Set Computer
DDL	Dynamische Datenspeicherlatenz
DDS	Deadline Deferable Server
DHF	Dominanter Hardware-Faden
DISC	Dynamic Instruction Stream Computer
DMAA	Dominant Metathread Access Announcing
DPE	Dynamic Priority Exchange
DSS	Deadline Sporadic Server
DTS	Dominant Time Slicing
DXS	Deadline Exchange Server
EDF	Earliest Deadline First
EDL	Earliest Deadline as Late as possible
EEMBC	Embedded Microprocessor Benchmark Consortium
ELLF	Enhanced Least Laxity First
EPIC	Explicitly Parallel Instruction Computing
FDL	Feste Datenspeicherlatenz
FIFO	First In First Out
FP	Fixed Priority
FPGA	Field Programmable Gate Array
FSP	Falsche Subpipeline
GP	Guaranteed Percentage
INT	Arithmetische Subpipeline des TriCore-Prozessors
IPC	Instructions Per Cycle
IPE	Improved Priority Exchange
ITS	Infinite Time Slice

L/S	Subpipeline für Speicherzugriffe des TriCore-Prozessors
LBF	Leeres Befehlsfenster
LLF	Least Laxity First
LOOP	Schleifenverarbeitende Subpipeline des TriCore-Prozessors
MTB	Mehrtaktbefehl
MTS	Minimal Time Slice
PIQ	Periodic Instruction Quantum
POSIX	Portable Operating System Interface
PRET	Precision Timed Architecture
RISC	Reduced Instruction Set Computer
RM	Rate Monotonic
RMT	Responsive Multithreaded processor
RRS	Round Robin Slack
RTM	Real-time Task manager
RTU	Real-Time Unit
RVMP	Real-time Virtual Multi-Processor
QoS	Quality of Service
SSCoP	Spring Scheduling Co-Processor
SMT	Simultaneous Multithreading
SZB	Sprungzielberechnung
TBS	Total Bandwidth Server
TCB	Thread Control Block
VISA	Virtual Simple Architecture
VLIW	Very Long Instruction Word
WCET	Worst Case Execution Time
WRR	Weighted Round Robin

B	Maximale Breite einer Instruktion in Bit
F	Durchsatz bei der Befehlsbereitstellung
G	Größe des Befehlsfensters
N	Anzahl Programmfäden
P	Anzahl Subpipelines
R	Rundenlänge
S	Anzahl Pipelinestufen
U	Auslastung des Prozessors
Z	Breite eines Speicherzugriffs auf den Instruktionsspeicher
t	Zeitpunkt
t_0	Startzeitpunkt
τ_i	i-ter Programmfaden
σ_k	k-ter Hardware-Faden
π_p	p-te Subpipeline
C_i	WCET des Programmfadens τ_i
D_i	Frist des Programmfadens τ_i
j_i	IPC des Programmfadens τ_i
L_i	Last des Programmfadens τ_i
q_i	Instruktionsquantum des Programmfadens τ_i
Q_i	Taktquantum des Programmfadens τ_i
T_i	Periode des Programmfadens τ_i
C_{ap}	WCET einer aperiodischen Anfrage
D_{ap}	Frist einer aperiodischen Anfrage
n_{regs}	Anzahl Register in einem Kontext
ΔT	Schedulingintervall
Δn	Anzahl ausgeführter Instruktionen
Δt	Dauer einer Ausführung (in Takten)
Δt_{CS}	Dauer eines kompletten Kontextwechsels
Δt_{in}	Dauer des Einlagerns eines Programmfadens in einen Hardwarefaden
Δt_{out}	Dauer des Auslagerns eines Programmfadens in den Speicher
Δt_{slack}	Spielraum – Rundenbruchteil der nicht für DTS-Fäden reserviert ist

L_{fetch}	Latenz beim Lesen aus dem Instruktionsspeicher
L_{mem}	Latenz bei einem Zugriff auf den Datenspeicher
L_{jump}	Latenz bei unbedingten Sprungbefehlen
L_{branch}	Leertakte, die von der Zuordnungstufe nach einem Sprungbefehl eingefügt werden
$L_{F,I}$	Latenz zwischen dem Speicherzugriff und dem Erscheinen des entsprechenden Befehlswortes im Befehlsfenster
$L_{I,E}$	Anzahl Pipelinestufen, die zwischen der Zuordnungs- und der Ausführungsstufe liegen
$L_{I,M}$	Anzahl Pipelinestufen, die zwischen der Zuordnungsstufe und Speichercontroller liegen

1 Über diese Arbeit

1.1 Motivation

Die Auslastung eines Prozessors kann durch die gleichzeitige Ausführung mehrerer Programmfäden erhöht werden. Dies beruht darauf, dass Teile des Prozessors, die von einem Programmfaden gerade nicht belegt werden, von anderen Programmfäden genutzt werden können. Dadurch wird der Gesamtdurchsatz des Prozessors erhöht, jedoch beeinflussen sich die Programmfäden gegenseitig, wodurch die Ausführungszeiten stark variieren, je nachdem welche Programmfäden gleichzeitig ausgeführt werden.

Bei Programmen in Echtzeitsystemen ist es jedoch wichtig, die maximale Ausführungszeit zu kennen, da sonst unvorhersehbare Folgen auftreten. Deshalb wird in dieser Arbeit ein Verfahren vorgestellt, wie trotz mehrerer konkurrierender Programmfäden die Ausführungszeit eines bevorzugten Programmfadens berechenbar ist. Dadurch ist es möglich, mit diesem Programmfaden ein Programm mit harten Echtzeitanforderungen auszuführen und die weiteren Programmfäden zur Ausführung von Programmen mit niedrigeren Echtzeitanforderungen zu verwenden.

Die gleichzeitige Ausführung mehrerer Programmfäden mit harten Echtzeitanforderungen ist prinzipiell nicht möglich (siehe Abschnitt 2.4.3), jedoch können mittels eines Zeitschlitzverfahrens mehrere Programme mit harten Echtzeitanforderungen quasi gleichzeitig ausgeführt werden, wie gezeigt werden wird.

Eine andere Möglichkeit, von der erhöhten Auslastung in mehrfädigen Systemen zu profitieren, ist die gleichzeitige Ausführung von Programmen mit unterschiedlichen Echtzeitanforderungen, sogenannte *gemischtkritische Echtzeitanwendungen* (engl. *mixed criticality applications*). In der Luftfahrtindustrie sind derartige Anwendungen bereits verbreitet, da unterschiedliche Programmteile unterschiedlich hohe Sicherheitsstufen und damit Echtzeitanforderungen erfüllen müssen [Barhorst 2009].

Aber auch in anderen Bereichen gewinnen sie zunehmend an Bedeutung, zum Beispiel in der Automobilindustrie. Dort wird versucht, die Anzahl der Mikrocontroller zu reduzieren, indem nicht wie bisher jede Aufgabe von einem eigenen Mikrocontroller erledigt wird, sondern wenige, dafür mächtigere Mikrocontroller mehrere Aufgaben übernehmen. Dabei sind häufige Wechsel zwischen verschiedenen Programmfäden nötig, die bei einfädiger Ausführung sehr teuer sind, in mehrfädigen Systemen jedoch fast nichts kosten [Hoover 2006].

1.2 Ziele

Primäres Ziel dieser Arbeit ist es, Echtzeitanforderungen mit einer mehrfädigen Prozessorarchitektur zu erfüllen um dadurch die Prozessorauslastung zu erhöhen. Dabei werden die Untersuchungen ausdrücklich auf einen Prozessorkern beschränkt. Das Zusammenspiel mehrerer Prozessorkerne innerhalb eines Prozessors ist nicht Gegenstand der Arbeit, die Ergebnisse könnten aber als Grundlage einer derartigen Untersuchung dienen.

Um eine hohe Auslastung zu erreichen, müssen möglichst viele Programmfäden mit möglichst unterschiedlichen Echtzeitanforderung zur Verfügung stehen, denn durch eine größere Auswahl ist eine optimale Belegung leichter realisierbar. Deshalb unterstützt die hier präsentierte Architektur eine nahezu beliebige Anzahl Programmfäden und unterschiedliche Algorithmen für Programme mit harten, mit weichen und ohne Echtzeitanforderungen.

Die Berechnung und Anpassung der Prozessorbelegung durch verschiedene Programmfäden, die sogenannte *Ablaufplanung* (engl. *scheduling*), wird meist nur auf minimale Weise durch die Prozessorhardware unterstützt, komplexere Algorithmen werden üblicherweise durch Software realisiert. Dies hat jedoch den Nachteil, dass dadurch Prozessorleistung für die Verwaltung verloren geht. Das kann soweit gehen, dass die Erhöhung des Durchsatzes bei einer höheren Anzahl Programmfäden völlig durch den zusätzlichen Organisationsaufwand aufgezehrt wird. Um diese negativen Folgen zu verhindern, wird ein vollständig auf Hardware-Logik basierender Ablaufplaner (engl. *scheduler*) präsentiert, der trotzdem die oben geforderte Flexibilität bietet.

Ein weiterer Punkt ist eine möglichst einfache, flexible und portierbare Schnittstelle zwischen Scheduler und dem Benutzer beziehungsweise Betriebssystem. Dadurch, dass sämtliche Scheduling-Daten im Speicher liegen und das Verwalten von Programmfäden durch einfache Speicherzugriffe erfolgt, also keine speziellen Prozessor-Instruktionen nötig sind, kann der Scheduler auf nahezu beliebige andere Prozessoren portiert werden. Außerdem kann eine Erhöhung der Anzahl der Programmfäden durch eine einfache Vergrößerung des Speichers erreicht werden.

Da Echtzeitsysteme größtenteils in eingebetteten Systemen eingesetzt werden, in denen der Energieverbrauch und die Größe des Prozessor eine entscheidende Rolle spielen, müssen diese Eigenschaften des Systems ebenfalls berücksichtigt werden. Günstig wirkt sich hierbei aus, dass bei einer größeren Anzahl Programmfäden auch mehr potentiell ausführbare Instruktionen zur Verfügung stehen und dadurch die Suche nach passenden Instruktionen für alle vorhandenen Prozessorressourcen vereinfacht wird.

Steht hingegen nur ein Programmfaden zur Verfügung, müssen große Anstrengungen unternommen werden, um die vorhandenen Ressourcen ausreichend mit Instruktionen zu versorgen. Es soll gezeigt werden, dass die dafür nötige Hardware-Logik (zum Beispiel für eine Sprungvorhersage und andere spekulative Techniken) in einem mehrfädigen Prozessor größtenteils eingespart werden, ohne den Gesamtdurchsatz zu verringern.

1.3 Idee

Es gibt mehrere Arten der Mehrfädigkeit (siehe Abschnitt 2.4.1), je flexibler diese sind, desto höher ist die erreichbare Prozessorauslastung. Deshalb wurde die flexibelste Variante, die sogenannte simultane Mehrfädigkeit (engl. *simultaneous multithreading, SMT*) gewählt. Hierbei können mehrere Instruktionen verschiedener Programmfäden gleichzeitig in einem Takt ausgeführt werden.

Bei den verbreiteten SMT-Prozessoren wird üblicherweise die Ausführungsreihenfolge der Instruktionen vom Prozessor verändert (engl. *out-of-order execution*), um die Auslastung zusätzlich zu erhöhen. Da diese Art der Ausführung aber keine harten Echtzeitschranken erlaubt, werden in der hier vorgestellten Architektur die einzelnen Instruktionen in ihrer ursprünglichen Reihenfolge ausgeführt (engl. *in-order execution*). Wie Untersuchungen zeigen (siehe Abschnitt 2.4.2), muss sich das aber nicht negativ auf die Prozessorauslastung auswirken und die nötige Hardware ist sehr viel kleiner und dadurch auch energiesparender.

Die SMT-Prozessor-Pipeline bildet die unterste Ebene des Schedulers. Sie soll möglichst einfach gehalten sein, damit eine hohe Taktfrequenz erreicht werden kann. Hier steht eine feste Anzahl (typischerweise zwei bis acht) Hardware-Programmfäden zur Verfügung. Diese Anzahl ist fest und von der Hardware vorgegeben. Jedem Hardware-Faden ist eine Priorität zugewiesen, anhand derer entschieden wird, von welchem Hardware-Faden die nächsten Instruktionen geholt werden.

Der Hardware-Faden mit der höchsten Priorität wird entsprechend jederzeit ausgeführt, sofern er ausreichend Instruktionen zur Verfügung stellt. Folglich wird er ausgeführt, als wäre er der einzige Programmfaden in einem einfädigen Prozessor. Diese quasi-singuläre Ausführung ermöglicht es, die maximale Ausführungszeit des Programmfadens zu berechnen, eine Voraussetzung für harte Echtzeitumgebungen. Durch verschiedene Effekte kann es jedoch trotzdem passieren, dass niederpriore Hardware-Fäden den höchstprioren Hardware-Faden verzögern. Um dies auszuschließen, sind einige zusätzliche Maßnahmen notwendig, die den höchstprioren Hardware-Faden vollständig von den anderen Hardware-Fäden isolieren.

Über dieser ersten Ebene liegt die zweite Hardware-Ebene, die die Prioritäten für die unterste Ebene steuert und dafür sorgt, dass alle Programmfäden ihre Echtzeitschranken einhalten. Außerdem ist sie dafür verantwortlich, dass jeweils die dringendsten Programmfäden aus dem Speicher geladen werden und in den Hardware-Fäden ausgeführt werden. Sobald ein Hardware-Faden seine Arbeit verrichtet hat, wird er wieder in den Hauptspeicher verschoben und durch einen anderen, dringenderen Programmfaden, der temporär deaktiviert im Speicher wartet, ersetzt.

Die dritte Ebene schließlich befindet sich im Hauptspeicher. Es ist ein spezieller Bereich, der zur Speicherung der Programmfadenkontrollblöcke (engl. *thread control block, TCB*) verwendet wird. In diesen Blöcken sind alle Informationen für die Ablaufplanung und der gesamte Kontext eines Programmfadens zusammengefasst. Um einen Programmfaden, der sich in einem solchen TCB befindet, zu aktivieren, muss sein Kontext, das heißt

im wesentlichen die Werte seiner Prozessorregister, in einen Hardware-Faden geladen und seine Ausführung gemäß der Scheduling-Informationen gesteuert werden. Beides ist Aufgabe der zweiten Ebene des Schedulers.

Das Anlegen eines Programmfadens erfolgt, indem Speicherplatz für einen TCB reserviert wird, die initialen Werte der Register dort vermerkt werden und anschließend der TCB in eine einfach verkettete Liste eingefügt wird. Es gibt mehrere dieser Listen für unterschiedliche Scheduling-Algorithmen. Die zweite Ebene des Schedulers erkennt selbstständig, dass die Scheduling-Daten verändert wurden und reagiert entsprechend darauf. Alle weiteren Operationen zur Verwaltung der Programmfäden (unterbrechen, beenden, schlafen legen, etc.) können ebenfalls durch einfache Schreibzugriffe auf den TCB-Speicherbereich angestoßen werden. Gleichzeitig stehen sämtliche Scheduling-Parameter jederzeit per Lesezugriff zur Verfügung.

1.4 Aufbau

Im nächsten Kapitel werden die Grundlagen, auf die die entwickelte Architektur aufbaut, vorgestellt. Kapitel 3 beschäftigt sich mit der ersten Ebene des Schedulers, dem zugrundeliegenden Prozessor. Dieses Kapitel ist allgemein gehalten, es beschreibt, wie ein beliebiger einfädiger, superskalarer Prozessor erweitert werden muss, um simultane Mehrfädigkeit bei vollständiger Isolation Programmfadens mit der höchtenn Priorität zu garantieren.

Im anschließenden Kapitel 4 wird die zweite Ebene des Schedulers erläutert. Es werden die zur Verfügung stehenden Scheduling-Algorithmen vorgestellt und die Hardware beschrieben, die zum Ein- und Auslagern von Programmfäden nötig ist. Ebenso wird die als dritte Ebene beschriebene Speicherschnittstelle dargestellt.

Die konkrete Realisierung eines Prozessors mit der in den beiden vorangehenden Kapiteln beschriebenen Architektur existiert in Form des CarCore-Prozessors, auf den in Kapitel 5 eingegangen wird. Insbesondere die praktischen Probleme, die bei einer realistischen Erweiterung eines kommerziellen Prozessors entstehen, sind Gegenstand dieses Kapitels.

Die Evaluierung der in den vorangehenden Kapiteln geäußerten Behauptungen anhand des CarCore-Prozessors wurde in Kapitel 6 zusammengefasst, um einen systematischen Überblick über die durchgeführten Experimente zu erlauben. Die vorangehenden Kapitel verweisen jeweils auf den entsprechenden Abschnitt in diesem Kapitel.

In Kapitel 7 werden verwandte Arbeiten mit dieser Arbeit verglichen, bevor im letzten Kapitel die Ergebnisse zusammengefasst werden und ein Ausblick auf weitere Forschungsfelder, die sich durch diese Arbeit erschließen, gegeben wird.

2 Grundlagen

Diese Arbeit verbindet zwei weite Forschungsbereiche der Informatik: Echtzeitsysteme und mehrfädige Programmausführung. Dementsprechend wird in diesem Kapitel auf diese beiden grundlegenden Bereiche eingegangen.

2.1 Echtzeitsysteme

Ein Computerprogramm gilt als korrekt, wenn das Ergebnis, das es liefert, korrekt ist. In vielen technischen Anwendungen spielt es aber nicht nur eine Rolle, ob das Ergebnis richtig ist, sondern auch, wie lange es dauert, bis das Ergebnis zur Verfügung steht [Buttazzo 2004, Seite 1]. Beispielsweise muss ein Routenplaner natürlich einen korrekten Weg vom Start zum Ziel berechnen können, wenn die Berechnung aber länger dauert als die Fahrzeit, ist das Ergebnis unbrauchbar.

Derartige Systeme, in denen das zeitliche Verhalten eine Rolle spielt, nennt man *Echtzeitsysteme*, sie kommen in nahezu allen technischen Bereichen vor, unter anderem in der Unterhaltungsindustrie, Elektroindustrie, Automobilindustrie, Luftfahrt oder dem Maschinenbau.

2.1.1 Harte und weiche Echtzeit

Echtzeitsysteme können in zwei grundsätzlich verschiedene Klassen eingeteilt werden [Manacher 1967], die man mit *hart* beziehungsweise *weich* bezeichnet. Bei harten Echtzeitbedingungen ist ein Ergebnis nur dann korrekt, wenn es auch innerhalb einer vorgegebenen Zeitspanne berechnet wurde. Klassisches Beispiel hierfür ist ein Airbag: das zuständige Programm muss innerhalb von wenigen Millisekunden entscheiden, ob die von den Sensoren gemessenen Werte auf eine Kollision hindeuten und den Airbag auslösen. Braucht das Programm zu lange, schlägt der Kopf des Fahrers bereits auf dem Armaturenbrett auf und das Airbag ist nutzlos.

Erfüllt ein System weiche Echtzeitanforderungen, so hält es die vorgegebenen Zeitschranken im Normalfall ein. Steht ein Ergebnis nach Ablauf der vorgegebenen Zeitspanne jedoch noch nicht zur Verfügung, so sind die Konsequenzen nicht katastrophal, wie bei harten Echtzeitsystemen, vielmehr können sie in einem gewissen Umfang akzeptiert werden. Je öfter die Zeitbegrenzung verletzt wird, desto geringer ist die Dienstgüte (engl. *quality of service, QoS*) des Systems.

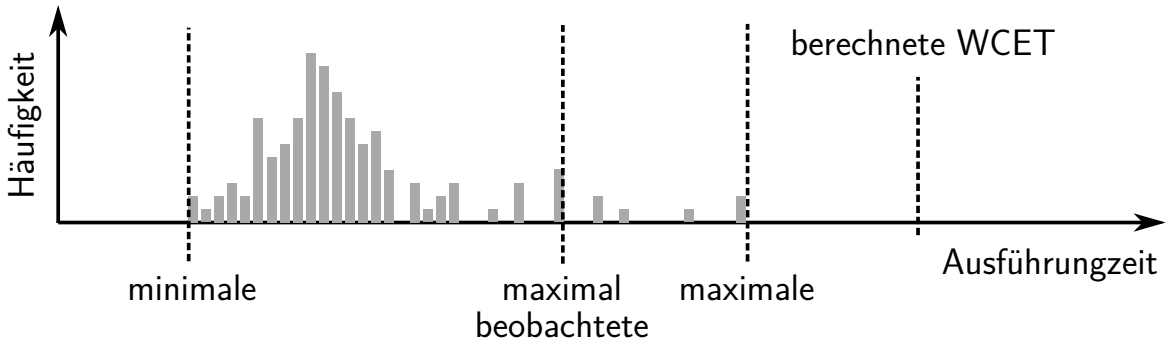

Abb. 2.1: Häufigkeitsverteilung der Ausführungszeit bei unterschiedlichen Pfaden durch einen gegebenem Programmcode

Die Übertragung einer Videokonferenz ist ein typisches Beispiel für weiche Echtzeit. Für ein flüssiges Bild muss 50 mal in der Sekunde ein neues Bild übertragen und berechnet werden. Sollten die Daten eines einzelnen Teilbildes einmal verspätet eintreffen, so wird das nächste Teilbild einfach verspätet angezeigt. Solange eine derartige Verzögerung nur selten auftritt, ist sie kaum zu bemerken und stört nicht. Sollte sie allerdings mehrmals pro Sekunde auftreten, so kann ab einer gewissen Häufigkeit nicht mehr erkannt werden, was am anderen Ende der Leitung passiert.

2.1.2 Laufzeitanalyse

Aufgrund von Schleifen und anderen Verzweigungen im Programmcode kann die Ausführung eines Programms je nach Eingabedaten sehr unterschiedlich lange dauern. Um die maximale Ausführungszeit (WCET) eines Programms zu ermitteln, bedient man sich der sogenannten Laufzeitanalyse. Dabei berechnet man für jeden der möglichen Pfade durch den Programmcode die Ausführungszeit und ermittelt das Maximum.

In der Praxis ist die genaue Ermittlung des Maximums nahezu unmöglich, da nicht alle Situationen und Pfade exakt analysiert werden können. Deshalb unterscheiden wir hier und im weiteren Verlauf der Arbeit zwischen der wirklichen maximalen Ausführungszeit und der (berechneten) WCET. Bei der Berechnung müssen zwei gegenläufige Bedingungen erfüllt werden: zum Einen muss die berechnete WCET sicher sein, das heißt, sie darf auf keinen Fall unter der wirklichen maximalen Ausführungszeit liegen, da sie sonst wertlos ist; zum Anderen sollte sie möglichst dicht an der wirklichen maximalen Ausführungszeit liegen, da sonst Rechenzeit für die Ausführung bereitgestellt wird, die nicht genutzt wird. Abbildung 2.1 verdeutlicht die Relation der verschieden Ausführungszeiten.

Die Abbildung zeigt auch das Problem, dass Messungen eine Laufzeitanalyse nicht ersetzen können, denn egal wie viele verschiedene Programmläufe mit unterschiedlichen Eingabedaten durchgeführt werden, ohne eine theoretische Analyse kann man sich nie sicher sein, ob nicht doch noch ein schlechterer Fall vergessen wurde. Es gibt zwar An-

sätze, durch Messungen die WCET zu bestimmen [Puschner 2003], die Forschung steckt hier allerdings noch in den Kinderschuhen und derart ermittelte WCETs können bislang nur für weiche Echtzeitsysteme verwendet werden.

Eine Laufzeitanalyse bezieht sich immer auf ein konkretes Programm, das auf einer bestimmten Architektur ausgeführt wird. Sollte sich einer der beiden Parameter ändern, so muss die Laufzeitanalyse neu durchgeführt werden, insbesondere ist eine Laufzeitanalyse nicht ohne weiteres von einer Architektur auf eine andere übertragbar, da aufgrund der Komplexität des Systems selbst kleinste Änderungen (im schlechtesten Fall) große Auswirkungen haben können. Mit der Frage, wie Programme mit minimaler WCET aussehen müssen, beschäftigt sich die Softwareentwicklung, die nicht Thema dieser Arbeit ist, stattdessen wird hier eine Architektur vorgestellt, die eine möglichst einfache und genaue Laufzeitanalyse ermöglichen soll.

2.2 Task Modelle

Die Laufzeitanalyse bezieht sich nur auf einen einzigen, sequenziellen Programmfaden, der ohne Interaktion mit anderen Programmfäden ausgeführt wird. Insbesondere in eingebetteten Systemen, die häufig auch Echtzeitanforderungen stellen, bestehen moderne Programme aber meist aus mehreren Dutzend Programmfäden. Es gibt unzählige Bezeichnungen für Programmfäden: Kontrollfaden, Prozess, Task, Thread, und so weiter. Je nach Autor kommen diesen Bezeichnungen unterschiedliche Bedeutungen zu, in dieser Arbeit werden sie jedoch synonym verwendet und bezeichnen einen sequenziellen Programmfaden. Eine Anwendung besteht aus einer zusammengehörenden Menge von mehreren Programmfäden, die man Taskset nennt.

Bereits in den sechziger Jahren des 20. Jahrhunderts waren Echtzeitapplikationen, die aus mehreren Prozessen bestehen, Gegenstand der wissenschaftlichen Forschung [Fineberg 1967]. Es wurden Wege gesucht, diese auf einem einzelnen Prozessor nacheinander auszuführen. Mit dem Aufkommen der Mehrkernprozessoren im letzten Jahrzehnt scheint es inzwischen zwar möglich, jeden Prozess auf einem eigenen Kern auszuführen, jedoch ist die typische Prozessanzahl in kommerziellen Echtzeitsystemen (30-100) weitaus höher als die Anzahl verfügbare Kerne. Daher macht es weiterhin Sinn, die Ausführung mehrerer Prozesse auf *einem* Prozessorkern zu untersuchen.

Nach Brinkley Sprunt [Sprunt 1990] können echtzeitfähige Programmfäden in drei Klassen eingeteilt werden, die im Folgenden genauer erläutert werden.

2.2.1 Periodisches Task Modell

Das in den siebziger Jahren von Liu und Layland [Liu 1973] entwickelte periodische Task Modell (Abbildung 2.2) ist noch immer das Standardmodell für mehrfädige Echtzeitsysteme. Dabei besteht ein Programm aus einer Menge von N Programmfäden τ_1 bis τ_N. Jeder Prozess wird periodisch aufgerufen, wobei die Ausführung während einer Periode

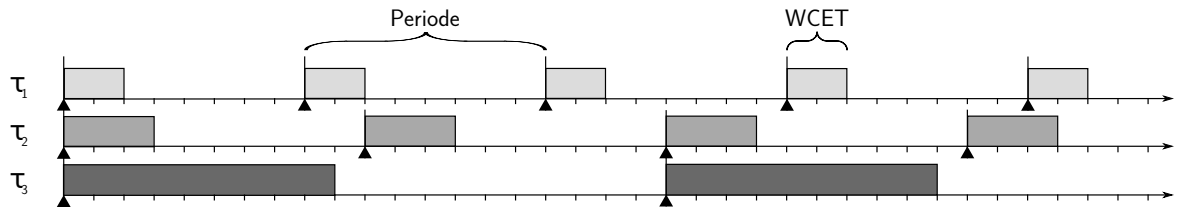

Abb. 2.2: Periodisches Task Modell

Job genannt wird. Ein Prozess besteht folglich aus einer unendlichen Reihe von hintereinander ausgeführten, identischen Jobs. Ein Job und aufgrund der strengen Periodizität ebenso der gesamte Prozess, kann durch zwei Parameter, die Periode T_i und die WCET C_i charakterisiert werden.

Die Periode T_i ist die Zeit, die zwischen dem Starten eines Jobs und dem Start des nächsten Jobs des gleichen Prozesses liegt. Ein Job muss vor dem Beginn des nächsten Jobs fertig sein, das heißt, innerhalb der Periode muss die Ausführung des Jobs beendet sein. Mit der WCET C_i wird die maximale Ausführungszeit eines Jobs auf der betrachteten Architektur bezeichnet.

Die Auslastung U des Prozessors gibt an, welcher Prozentsatz der zur Verfügung stehenden Rechenzeit für ein gegebenes Taskset auf einer gegebenen Prozessorarchitektur benötigt wird. Dazu wird zunächst die Last L_i jedes einzelnen Prozesses berechnet:

$$L_i = \frac{C_i}{T_i} \tag{2.1}$$

Durch Aufsummieren der individuellen Bruchteile L_i erhält man die Auslastung

$$U = \sum_{i=1}^{N} L_i = \sum_{i=1}^{N} \frac{C_i}{T_i} \tag{2.2}$$

Wenn U größer als eins ist, braucht das Taskset mehr Rechenleistung als zur Verfügung steht (Überlast), das heißt, bei einer Ausführung können die Fristen nicht eingehalten werden, das Taskset ist nicht *einplanbar*. Das Planen der Ablaufreihenfolge der Prozesse nennt man *Ablaufplanung* oder (englisch) *Scheduling*.

Die Bedingung, unter der ein Taskset mit einem bestimmten Algorithmus einplanbar ist, nennt man *Einplanbarkeitsbedingung* (engl. *schedulability test*). Einen Scheduling-Algorithmus, bei dem die Einplanbarkeitsbedingung gleich

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1 \tag{2.3}$$

ist, nennt man *optimal*, da er einen brauchbaren Plan erstellen kann, falls er existiert.

2.2.2 Sporadische Prozesse

Sporadische Prozesse nach Aloysius Mok [Mok 1983] erweitern das periodische Modell, indem die Periode nicht mehr als exakter Abstand zwischen dem Start zweier Jobs angenommen wird, sondern als *Mindestabstand*. Das heißt, dass zwischen zwei Jobs noch zusätzliche Zeit vergehen kann. Außerdem wird ein dritter Parameter, die *Frist* (engl. *deadline*) D_i eingeführt. Sie gibt die Zeit an, die vom Start bis zum Ende der Ausführung eines Jobs vergehen darf. Bei periodischen Prozessen ist die Frist gleich der Periode, bei sporadischen kann sie auch kürzer als die Periode sein.

Dadurch, dass die Jobs im Prinzip zu beliebigen Zeitpunkten starten können, ist es möglich, ereignisgesteuerte Interruptroutinen zu modellieren. Der Mindestabstand ist nötig, damit vor dem eigentlichen Start des Systems überprüft werden kann, ob das Taskset einplanbar ist und somit harte Echtzeitanforderungen erfüllt.

2.2.3 Aperiodische Anfragen

Wenn Prozesse in unregelmäßigen Abständen ankommen und keine Aussage über einen Mindestabstand gemacht werden kann, spricht man von *aperiodischen Prozessen*. In eingebetteten Systemen werden sie meist durch sogenannte *Unterbrechungsanforderungen* (engl. *interrupt requests*) ausgelöst. Dies sind Signale von externen Geräten, die den Prozessor auffordern, auf ein äußeres Ereignis (zum Beispiel eine ankommende Nachricht oder einen besonderen Sensormesswert) zu reagieren.

Die erfolgreiche Ausführung dieser Prozesse kann nicht garantiert werden, da bei gleichzeitig oder kurz hintereinander eintreffenden Prozessen die maximale Prozessorkapazität schnell überschritten wird. In solch einem Fall muss die Ausführung einiger Prozesse verzögert werden, bis wieder genügend Kapazität zur Verfügung steht. Da in der Zwischenzeit weitere Prozesse eintreffen können, kann die Last lawinenartig anwachsen; eine Frist, innerhalb der ein Prozess garantiert ausgeführt wird, kann nicht gegeben werden. Deshalb sind aperiodische Prozesse, im Unterschied zu den vorher genannten periodischen und sporadischen Prozessen, nicht echtzeitfähig im harten Sinn.

Um wenigstens weiche Echtzeitanforderungen zu erfüllen, können die ankommenden Prozesse zwischengespeichert werden. Sobald genügend Prozessorressourcen zur Verfügung stehen, werden sie in der Reihenfolge ihrer Ankunft ausgeführt (engl. *first in – first out*). Hierbei spielen weder WCET noch Frist keine Rolle, es wird einfach versucht, die Anfragen so früh wie möglich (engl. *best effort*) zu beantworten.

Alternativ kann an eine aperiodischen Anfrage eine individuelle WCET C_{ap} und Frist D_{ap} angehängt werden. In diesem Fall muss der Scheduler entscheiden, ob die angeforderte Rechenzeit zur Verfügung gestellt werden kann und akzeptiert die Anfrage bzw. lehnt sie ab [Chetto 1989]. Um die beiden Arten der Behandlung von aperiodischen Anfragen zu unterscheiden, spricht man im Falle einer bestmöglichen Ausführung von weichen aperiodischen Prozessen, bei ablehnbaren Anfragen mit Fristen von harten aperiodischen Prozessen.

2.3 Schedulingalgorithmen

Die Ablaufplanung in harten Echtzeitsystemen wird dominiert von zwei Algorithmen, die bereits 1973 in einer bahnbrechenden Arbeit von Liu und Layland [Liu 1973] analysiert wurden, *Rate Monotonic* und *Earliest Deadline First*. Ein alternativer Algorithmus, der etwas in Vergessenheit geraten ist, aber in dem vorzustellenden Scheduler eine große Rolle spielt, ist *Infinite Time Slicing*. Weiterhin bietet es sich für weiche Echtzeitsysteme an, die Ausführungsrate der Instruktionen zu regeln.

2.3.1 Ablaufplanung mit festen Prioritäten

Eine Möglichkeit, um die Ausführungsreihenfolge von Prozessen zu ermitteln, ist, jedem Prozess eine feste Priorität (engl. *fixed priority, FP*) zuzuordnen. Werden die Prioritäten entsprechend den Periodenlängen vergeben (je kürzer die Periode, desto höher die Priorität), so spricht man von einer *Ablaufplanung mit monotonen Raten* (engl. *rate monotonic, RM*).

Der Vorteil dieses Verfahrens ist die einfache Implementierung, da jeder Prozess eine feste Priorität hat und jeweils der Prozess mit der höchsten Priorität ausgeführt wird. Deshalb wird dieser Algorithmus in vielen eingebetteten Systemen eingesetzt. Ein Nachteil des Verfahrens ist hingegen die schlechte Verteilung der Rechenzeit, die zu einer geringen Auslastung der Prozessorressourcen führt. Um Einplanbarkeit zu garantieren, muss für die Auslastung U Folgendes gelten: [Liu 1973]

$$U \leq N \cdot \left(\sqrt[N]{2} - 1 \right) \tag{2.4}$$

Lässt man die Anzahl von Prozesse N gegen unendlich gehen, so ergibt sich

$$U \leq \ln 2 \approx 69,3\% \tag{2.5}$$

Das heißt, über 30% der Prozessorleistung bleiben ungenutzt, wenn man die rechtzeitige Ausführung aller Prozesse garantieren will. Trotzdem ist es bei einer Auslastung von über 69,3% durchaus möglich, dass alle Prozesse ihre Fristen einhalten, es muss aber nicht sein. Diese Abschätzung für den absolut schlechtesten Fall ist sehr pessimistisch, statistische Auswertungen haben gezeigt, dass eine Auslastung von 88% im Normalfall noch einplanbar ist [Lehoczky 1989].

2.3.2 Ablaufplanung mit Fristen

Um eine vollständige Auslastung zu erreichen ist es nötig, die Prioritäten dynamisch zu vergeben, das heißt sie in jedem Takt neu zu berechnen. Ein Algorithmus der dies leistet, ist *Earliest Deadline First (EDF)* [Liu 1973]. Bei diesem Verfahren werden die Prioritäten entsprechend der noch verbleibenden Zeit bis zur jeweiligen Frist des Prozesses vergeben.

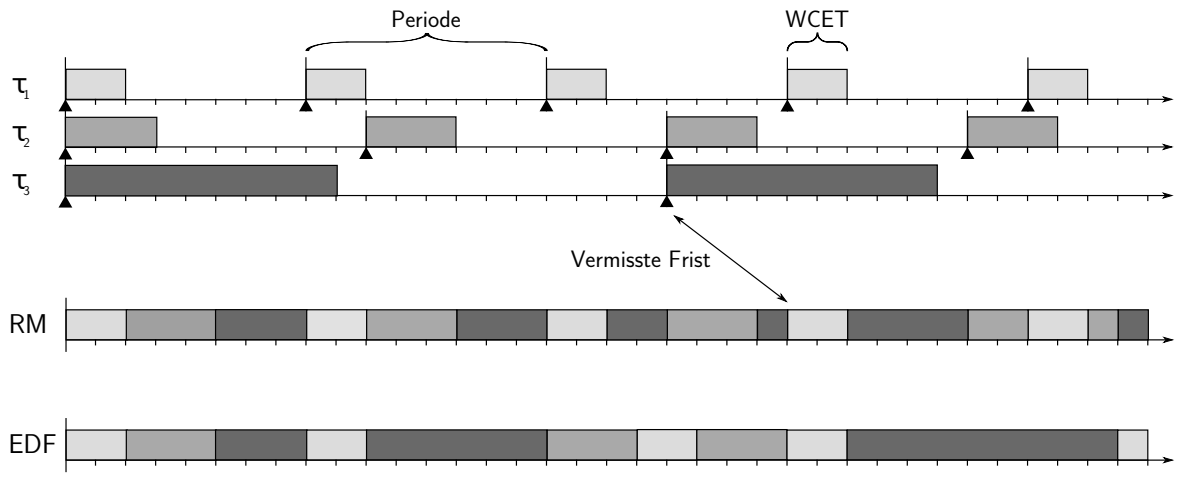

Abb. 2.3: Vergleich von RM- und EDF-Scheduling

Je näher eine Frist rückt, desto höher wird die Priorität. Mit anderen Worten, jeweils der Prozess, der als erstes fertig sein muss, wird ausgeführt.

Dieser Algorithmus ist optimal im Sinne von (2.3), das heißt wenn ein Taskset tatsächlich einplanbar ist, dann kann dies durch einen EDF-Scheduler erreicht werden. Einzige Bedingung ist, dass die Auslastung einen Wert von eins nicht übersteigt. Abbildung 2.3 zeigt ein Taskset, dass mit EDF einplanbar ist, jedoch nicht mit RM.

Im kommerziellen Kontext wird die Ablaufplanung mit Fristen nur selten eingesetzt, da die Ablaufplanung mit Prioritäten angeblich in vielen Bereichen Vorteile bietet. Wie Giorgio Buttazzo [Buttazzo 2005] nachgewiesen hat, trifft das für die meisten Bereiche nicht zu, einzig die Implementierung ist bei der Ablaufplanung mit Fristen etwas aufwendiger. Doch auch hier ist der Unterschied geringer als gemeinhin angenommen, da bei beiden Verfahren aus einer Liste ein aktiver Programmfaden herausgesucht werden muss. Einzig die Schlüssellänge ist bei festen Prioritäten (typischerweise 0 bis 63) kürzer als bei Fristen (mindestens 0 bis 1000).

2.3.3 Ablaufplanung mit Zeitschlitzen

Seit der Veröffentlichung der Arbeit von Liu und Layland konzentriert sich die Forschung hauptsächlich auf Varianten des RM- und des EDF-Algorithmus, ein dritter Algorithmus, der bereits Jahre vorher von Mark Fineberg und Omri Serlin veröffentlicht wurde [Fineberg 1967], wird weitgehend ignoriert. Beim der sogenannten *Ablaufplanung mit infiniten Zeitschlitzen* (engl. *infinite time slicing, ITS*) wird die Rechenzeit in Planungsintervalle ΔT eingeteilt und jeder Prozess erhält in jedem Intervall jeweils soviel Rechenzeit, wie es seiner Last L_i entspricht.

Pro Planungsintervall erhält der Programmfaden τ_i folglich $L_i \Delta T$ Rechenzeit. Damit alle Programmfäden befriedigt werden können, darf die Summe dieser Rechenzeitbruchstücke

die Länge des Intervalls nicht übersteigen:

$$\sum_{i=1}^{N} (L_i \Delta T) = \Delta T \cdot \sum_{i=1}^{N} L_i = \Delta T \cdot U \quad \leq \quad \Delta T \tag{2.6}$$

Gekürzt mit ΔT entspricht das genau der Planbarkeitsbedingung (2.3). Der ITS Algorithmus kann also als direkte Implementierung der Planbarkeitsbedingung verstanden werden, der per Konstruktion optimal ist.

Dass jeder Programmfaden die erforderliche Rechenzeit erhält, kann leicht überprüft werden, indem man die individuelle Rechenzeit pro Programmfaden $L_i \cdot \Delta T$ mit der Anzahl ΔT-Intervalle pro Periode $\frac{T_i}{\Delta T}$ multipliziert:

$$L_i \cdot \Delta T \cdot \frac{T_i}{\Delta T} = L_i \cdot T_i = C_i \tag{2.7}$$

Die zugeteilte Rechenzeit entspricht also genau der gewünschten WCET C_i.

Von praktischer Bedeutung ist, dass bei ITS sehr häufig zwischen den Programmfäden umgeschaltet wird, deshalb darf ein Kontextwechsel keine Zeit kosten. Vor 40 Jahren war dies ein relativ großes Problem, heutzutage stehen aber technische Lösungen zur Verfügung, siehe Abschnitte 2.4 und 4.1. Weitaus bedeutender ist der Umstand, dass das Planungsintervall ΔT infinitesimal klein werden muss, um exakt zu sein. Da ein Programmfaden aber immer mindestens für einen Prozessortakt ausgeführt werden muss, ist dies technisch nicht möglich. Doch auch hier gibt es einen Ausweg, siehe Abschnitt 4.2.1.2.

2.3.4 Aperiodische Server

Aperiodische Anfragen dürfen die Ausführung von harten Echtzeitprozessen (periodischen und sporadischen) nicht beeinträchtigen, deshalb ist die einfachste Methode, um aperiodische Prozesse auszuführen, die Hintergrundausführung. Dabei wird ein aperiodischer Prozess nur dann ausgeführt, wenn gerade kein anderer Prozess ausgeführt werden muss und dadurch den Prozessor belegt. Doch mit dieser Methode ist die Ausführung der aperiodischen Prozesse sehr zufällig und schwankend, weshalb sogenannte *aperiodische Server* eingesetzt werden.

Ein aperiodischer Server analysiert das Laufzeitverhalten der harten Echtzeitprozesse und stellt nicht benötigte Rechenzeit für aperiodische Prozesse zur Verfügung. Dabei muss er bestrebt sein, die von einem aperiodischen Prozess angeforderte Rechenzeit C_i möglichst schnell bereitzustellen. Die durchschnittliche Dauer zwischen der Anfrage und der Beendigung des angeforderten Prozesses (sogenannte *Anwortzeit*) ist ein Maß für die Güte eines aperiodischen Servers.

Eine einfache Möglichkeit einen aperiodischen Server zu realisieren ist der sogenannte *Polling Server* [Sha 1986]. Dies ist ein zusätzlicher Prozess, der eine Priorität entsprechend seiner WCET erhält. Diese WCET (in diesem Fall spricht man von der Kapazität

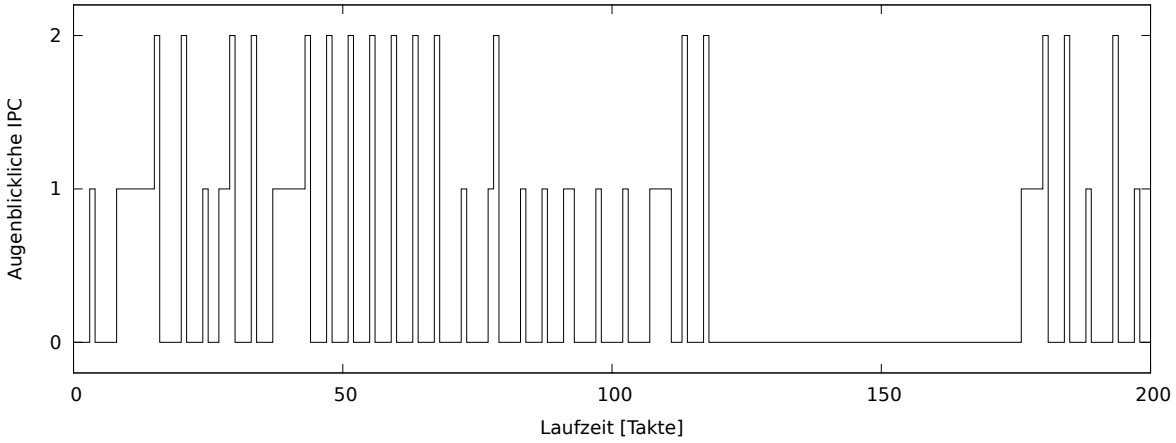

Abb. 2.4: Augenblickliche IPC bei mikroskopischer Betrachtung

des Servers) ist so gewählt, dass alle anderen Prozesse noch einplanbar sind. Dadurch, dass der Polling Server periodisch aufgerufen wird, wird die Antwortzeit gegenüber der Hintergrundausführung deutlich verbessert und es kann ein minimaler Fortschritt aperiodischer Prozesse garantiert werden.

Nachteilig wirkt sich aus, dass ein Polling Server nur sehr schlecht mit mehreren gleichzeitig auftauchenden Anfragen (sogenannten *bursts*) umgehen kann. Falls die Kapazität erschöpft ist, muss die Ausführung aperiodischer Anfragen auf die nächste Periode verschoben werden, was zu sehr langen Antwortzeiten führen kann. Umgekehrt ist die für den Polling Server reservierte Zeit unweigerlich verloren, wenn zu dessen Laufzeit keine aperiodische Anfrage auftritt.

Um diese Probleme in den Griff zu bekommen, wurden zahlreiche andere Verfahren entwickelt, die deutlich kürzere Antwortzeiten erlauben. Eine ausführliche Diskussion dieser Ansätze findet sich in Abschnitt 7.4.

2.3.5 Regelung der Ausführungsrate

Bei weichen Echtzeitanwendungen gibt es häufig keine feste Fristen, zu denen ein Job beendet sein muss, vielmehr soll ein konstanter Mindestdurchsatz garantiert werden. Der Durchsatz bzw. die Ausführungsrate entspricht der Anzahl der ausgeführten Instruktionen Δn geteilt durch die Dauer dieser Ausführung Δt in Takten. Gemessen wird die Ausführungsrate in ausgeführten Instruktionen pro Taktzyklus (engl. *instructions per cycle, IPC*), oft steht die Abkürzung IPC auch für die Ausführungsrate selbst.

Bei mikroskopischer Betrachtung ($\Delta t = 1$ Takt) kann die augenblickliche IPC nur ganzzahlige Werte annehmen, die zwischen null und der maximalen Anzahl gleichzeitig ausführbarer Instruktionen liegen. Da sich Takte, in denen Instruktionen ausgeführt werden und Takte, in denen nichts ausgeführt wird (beispielsweise weil auf den Speicher gewartet wird) ständig abwechseln, schwankt die Ausführungsrate sehr stark zwischen den

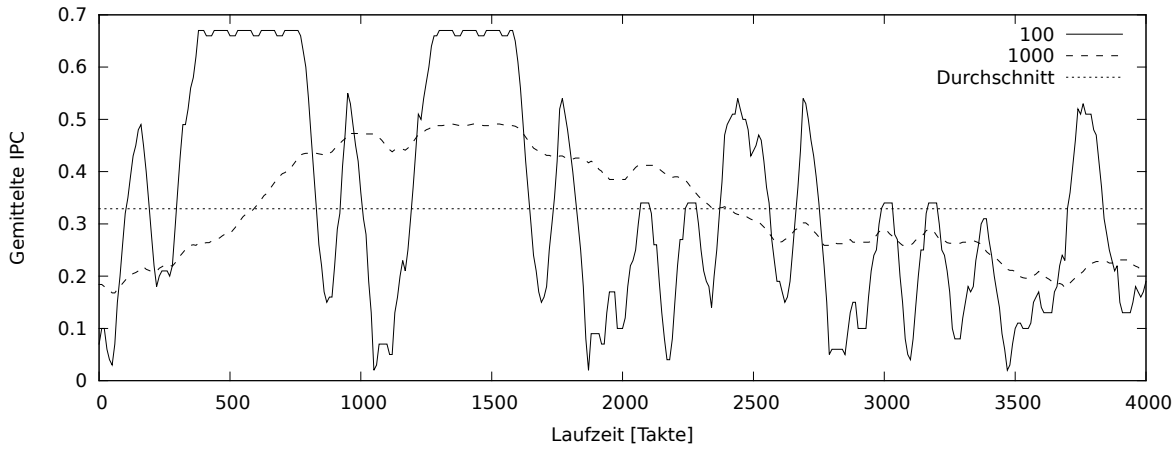

Abb. 2.5: Zeitlicher Verlauf der gemittelten IPC

Extremwerten. Abbildung 2.4 zeigt die augenblickliche IPC bei einem Prozessor, der maximal zwei Befehle gleichzeitig ausführen kann.

Die Kurve kann geglättet werden, indem man einen längeren Zeitraum Δt wählt, jedoch schwankt auch dann die gemittelte Ausführungsrate je nach Position im Programmcode noch stark [Sherwood 1999]. Außerdem hängt die maximale und die minimale Ausführungsrate davon ab, wie groß das Intervall gewählt wird. In Abbildung 2.5 ist der zeitliche Verlauf der IPC in Abhängigkeit vom gewählten Zeitintervall Δt für das Benchmarkprogramm `a2time` dargestellt.

Aufgrund der zeitlichen Schwankungen der Ausführungsrate ist eine Regelung, die einen konstanten Durchsatz garantiert, relativ schwierig. Es existieren einige indirekte Ansätze von anderen Autoren (siehe Abschnitt 7.3), der in dieser Arbeit beschriebene (siehe Abschnitt 4.2.2.5) regelt die Ausführungsrate hingegen direkt.

2.4 Mehrfädige Programmausführung

In frühen Mikroprozessoren dauerte die Ausführung einer Instruktion mehrere Taktzyklen. Erst als Anfang der 1980er Jahre die ersten RISC-Prozessoren [Patterson 1980] aufkamen, wurde es mit Hilfe der sogenannten *Pipelining-Technik* möglich, je Takt eine Instruktion auszuführen. Hierfür wird die Ausführung einer Instruktion in eine feste Anzahl von Stufen eingeteilt und die Ausführung mehrerer Instruktionen überlappt (siehe Abbildung 2.6). Da jede Instruktion nacheinander alle Stufen durchläuft, dauert die eigentliche Ausführung weiterhin mehrere Takte, jedoch belegt eine Instruktion eine Stufe nur einen Takt lang, sodass in jedem Takt eine neue Instruktion am Beginn der Pipeline eingefügt werden kann.

Ein Problem stellen jedoch Datenabhängigkeiten dar. Eine Datenabhängigkeit besteht, wenn eine Instruktion vom Ergebnis einer vorangehenden Instruktion abhängt. Wird das Ergebnis erst in einer späten Pipelinestufe generiert, aber bereits in einer frühen

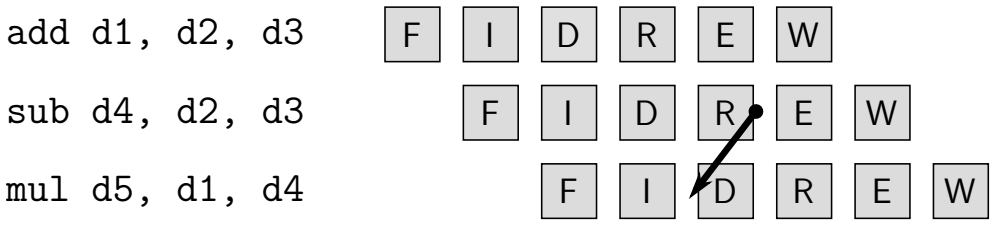

Abb. 2.6: Typische sechsstufige Pipeline mit Datenabhängigkeit

Stufe von der abhängigen Instruktion benötigt, so kann es sein, dass die Erzeugung des Ergebnisses zeitlich nach der Abholung durch die abhängige Instruktion liegt (siehe Abbildung 2.6). Dies kann nur verhindert werden, indem die abhängige Instruktion solange verzögert wird, bis das Ergebnis verfügbar ist.

2.4.1 Entwicklung der Mehrfädigkeit

Völlig vermieden werden können Datenabhängigkeiten durch das sogenannte *Interleaved Multithreading*, das bereits bei einem sehr frühen Computer, dem CDC 6600 eingeführt wurde [Thornton 1965]. Statt nur einen Programmfaden auszuführen, werden mehrere unabhängige Programmfäden ausgeführt, ihre Anzahl entspricht der Anzahl der Pipelinestufen. Von jedem Programmfaden befindet sich in jedem Takt nur eine Instruktion in der Pipeline, erreicht diese das Ende der Pipeline, so wird im nächsten Takt am Beginn der Pipeline die nächste Instruktion des betreffenden Programmfadens eingefügt. Da jede Instruktion in der Pipeline zu einem anderen, unabhängigen Programmfaden gehört, sind Datenabhängigkeiten innerhalb der Pipeline unmöglich. Durch dieses Verfahren wird der Gesamtdurchsatz des Prozessors zwar erhöht, aber es ist sehr unflexibel, da die Anzahl der Programmfäden genau festgelegt ist und die Gesamtausführungszeit eines einzelnen Programmfadens weiterhin mehrere Takte pro Instruktion beträgt.

Feinkörnige Mehrfädigkeit (engl. *fine-grained multithreading*) [Agarwal 1992] erhöht die Flexibilität, indem mehrere Instruktionen des gleichen Programmfadens sich gleichzeitig in der Pipeline befinden können, solange keine Datenabhängigkeiten bestehen. Falls eine Datenabhängigkeit besteht, werden Instruktionen anderer Programmfäden in die Pipeline eingefügt, bis sichergestellt ist, dass zu dem Zeitpunkt, an dem die abhängige Instruktion das betroffene Datum anfordert, dieses bereits berechnet wurde.

Eine weitere Variante stellt die *grobkörniger Mehrfädigkeit* (engl. *coarse-grained multithreading*) [Weber 1989] dar. Bei Verwendung dieser Technik wird nur dann zwischen verschiedenen Programmfäden gewechselt, wenn eine längere Pause von etlichen Takten in der Ausführung auftritt, weil der entsprechende Programmfaden auf etwas warten muss. Dies kann zum Beispiel das Warten auf externe Eingabegeräte oder ein Zugriff auf den Hauptspeicher sein, wenn sich die betreffende Speicherstelle nicht im Cache befindet. Tritt solch ein Ereignis auf, so wird von einem Programmfaden auf einen anderen umgeschaltet. Dieser sogenannte *Kontextwechsel* kann im Unterschied zur feinkörnigen

Mehrfädigkeit mehrere Takte dauern, solange die Kontextwechselzeit klein ist im Verhältnis zur Dauer der Unterbrechung. Bei der feinkörnigen Mehrfädigkeit muss der Kontextwechsel hingegen ohne Verzögerung möglich sein, da bei Unterbrechungen von nur ein, zwei Takten eine Verzögerung nicht akzeptabel ist.

Um den Durchsatz bei einfädigen Prozessoren weiter zu steigern, können in einem Takt mehrere Instruktionen gleichzeitig in die Pipeline eingefügt werden, das heißt jede Pipelinestufe verarbeitet mehrere Instruktionen gleichzeitig. Diese Fähigkeit nennt man *Superskalarität* [Sima 1997] im Gegensatz zur skalaren Ausführung eines Prozessors, der nur eine Instruktion gleichzeitig verarbeiten kann. Hier spielen Datenabhängigkeiten wiederum eine große Rolle, denn Instruktionen können nur dann parallel ausgeführt werden, wenn keine Datenabhängigkeiten bestehen.

Werden die Instruktionen in der Reihenfolge verarbeitet, in der sie im Speicher liegen (sogenannte *In-Order-Ausführung*), stößt man sehr schnell an die Grenzen der superskalaren Ausführung, da selten mehr als zwei, drei direkt aufeinanderfolgende Instruktionen datenunabhängig sind. Einen Ausweg bietet die *Out-of-Order-Ausführung*, bei der Instruktionen, die keine Datenabhängigkeit haben, vorgezogen werden können. Dadurch wird der Befehlsstrom umsortiert, um möglichst viele Instruktionen gleichzeitig ausführen zu können. Verbindet man Superskalarität und Mehrfädigkeit, so spricht man von *simultaner Mehrfädigkeit* (engl. *simultaneous multithreading, SMT*) [Hirata 1992; Tullsen 1995]. Durch einen SMT-Prozessor können folglich in einem Takt von mehreren Programmfäden jeweils mehrere Instruktionen gleichzeitig ausgeführt werden.

2.4.2 Ausführungsreihenfolge

Auch bei SMT Prozessoren kann zwischen In-Order- [Goossens 1996] und Out-of-Order-Ausführung [Tullsen 1995] unterschieden werden, jedoch ist der Vorteil der veränderten Ausführungsreihenfolge nicht so deutlich wie im einfädigen Fall. Das liegt daran, dass bei mehreren Programmfäden auch ohne Änderung der Ausführungsreihenfolge so viele unterschiedliche Instruktionen zur Verfügung stehen, dass es genügend datenunabhängige Instruktionen gibt, um alle Ausführungseinheiten auszulasten [Moon 2003].

Eine Untersuchung von Sébastien Hily [Hily 1999] zeigt, dass In-Order-SMT-Prozessoren 85% des Durchsatzes von Out-of-Order-SMT-Prozessoren erreichen. Da letztere aber typischerweise eine längere Pipeline benötigen und aufgrund der höheren Komplexität mit einer geringeren Taktfrequenz betrieben werden können, geht er von ungefähr vergleichbarer Geschwindigkeit aus und empfiehlt wegen der geringeren Komplexität, die sich auch auf eine kürzere Entwicklungs- und Testphase auswirkt, die Verwendung von In-Order-SMT-Prozessoren.

Zu einem ähnlichen Ergebnis kommt Venkata Krishnan, der für leistungsschwache Systeme In-Order-SMT-Prozessoren empfiehlt, da sie eine stabilere und höhere Leistung bieten als vergleichbare Mehrkernprozessoren mit Out-of-Order-Ausführung. Für Systeme mit hohen Geschwindigkeitsanforderungen empfiehlt er Mehrkernprozessoren mit

In-Order-SMT-Ausführung da diese erheblich kostengünstiger sind als Out-of-Order-Prozessoren mir einem vergleichbaren Durchsatz. Konsequenterweise beschäftigt sich Chengjie Zang mit der Instruktionszuordnung von In-Order-SMT-Prozessoren [Zang 2008] und erreicht eine mit Out-of-Order-SMT-Prozessoren vergleichbare Leistung, wenn die Anzahl der parallelen Funktionseinheiten nicht zu groß wird.

Diese Forschungsergebnisse wurden jedoch lange Zeit ignoriert, die meisten kommerziellen SMT-Prozessoren basieren auf einer Out-of-Order-Zuordnung. Dies könnte daran liegen, dass die kommerziellen Prozessorlinien früh die einfädige Out-of-Order-Ausführung unterstützten, da sie sich im einfädigen Fall deutlich auszahlt, und der weitere Schritt zur mehrfädigen Out-of-Order-Ausführung relativ klein ist. Erst in den letzten Jahren, in denen der Energieverbrauch der Prozessoren immer mehr in den Vordergrund drängt, wird die geringere Komplexität der In-Order-Ausführung wieder attraktiv, da dadurch kleinere Prozessoren mit weniger Leckströmen und einem geringeren Energieverbrauch möglich sind. Ein prominentes Beispiel für Energiesparmöglichkeiten im Zusammenhang mit mehrfädiger In-Order-Ausführung ist der Intel Atom Prozessor [Gerosa 2008].

Die geringere Komplexität von In-Order-SMT-Prozessoren [Moon 2004] wirkt sich nicht nur auf Entwicklungszeit und Energieverbrauch aus, auch die Laufzeitanalyse ist bei einer geringeren Prozessorkomplexität deutlich einfacher [Heckmann 2003]. Durch die feste Ausführungsreihenfolge kann der Zustand des Prozessores genauer vorhergesagt werden, wodurch die WCET näher an der wirklichen maximalen Ausführungszeit liegt als bei Out-of-Order-Prozessoren. Diese bessere Vorhersagbarkeit ohne merklichen Leitungsverlust ist der Grund für die Wahl eines In-Order-SMT-Prozessors als Grundlage dieser Arbeit.

2.4.3 Gegenseitige Beeinflussung von Programmfäden

SMT erhöht den Gesamtdurchsatz des Prozessors, indem die Prozessorressourcen gleichzeitig unter mehreren Programmfäden aufgeteilt werden. Dadurch kommt es jedoch zu Konflikten, wenn zwei Programmfäden die gleiche Ressource beanspruchen. Da nur ein Programmfaden die Ressource belegen kann, wird der andere Programmfaden zwangsläufig verzögert. Wie oft es zu solchen Verzögerungen kommt, hängt davon ab, wie oft Programmfäden gleichzeitig auf dieselbe Ressource zugreifen wollen. Die Ausführungszeit eines Programmfadens hängt folglich davon ab, welche Programmfäden parallel dazu ausgeführt werden [Parekh 2000].

Man kann versuchen, die Auslastung einzelner Prozessorressourcen zu messen, um Programmfäden zu finden, die eine möglichst hohe Gesamtleistung erbringen, das heißt, die sich möglichst wenig gegenseitig beeinflussen. Dies kann man entweder fortlaufend während der Programmausführung durchführen und jeweils den am schlechtest passenden Programmfaden durch einen anderen ersetzen [Parekh 2000], oder man führt zuerst jede Kombination der Programmfäden für jeweils einen kurzen Zeitraum aus und entscheidet dann, welche Kombinationen die günstigsten sind [Snavely 2000].

Beide Verfahren können die gegenseitige Beeinflussung nur minimieren, für Echtzeitanwendung ist es aber notwendig, eine obere Schranke dafür zu finden. Zwar untersucht Rohit Jain [Jain 2002] die gleichzeitige SMT-Ausführung unter Echtzeitbedingungen, jedoch gelingt es auch ihm nur weiche Echtzeitanforderungen zu erfüllen, da extreme Kollisionen nicht völlig ausgeschlossen werden können und sichere obere Schranken kaum angegeben werden können.

Als einzige Möglichkeit, eine statische Laufzeitanalyse zu ermöglichen, bleibt die völlige Vermeidung einer gegenseitigen Beeinflussung durch eine statische Verteilung der zur Verfügung stehenden Ressourcen auf die Programmfäden. Dadurch hat zu jedem Zeitpunkt immer nur ein bestimmter Programmfaden das exklusive Recht auf eine bestimmte Ressource zuzugreifen [Barre 2008a].

Da sich die Zugriffe auf einzelne Ausführungseinheiten eines Out-of-Order-SMT-Prozessors nur sehr schwer direkt kontrollieren lassen, beschränkt man sich in diesem Falle darauf, externe Ressourcen wie die Cache-Größe zu limitieren, um die Ausführungszeit zu regeln. Dadurch können jedoch nur weiche Echtzeitanforderungen erfüllt werden (siehe Abschnitt 7.3).

Um gleichzeitig mehrere parallele Programmfäden mit harten Echtzeitanforderungen auszuführen, müssen alle Prozessorressourcen disjunkt zwischen ihnen aufgeteilt werden, da davon ausgegangen werden muss, dass im schlimmsten Fall jeder Programmfaden alle ihm zur Verfügung stehende Ressource belegt. Gemäß Jonathan Barre [Barre 2008b] gibt es drei Möglichkeiten, die Ressourcen aufzuteilen:

- Zwischenspeicher und Warteschlangen werden statisch partitioniert, das heißt jeder Programmfaden erhält von diesen Ressourcen jeweils eine eigenes Exemplar, dessen Kapazität gegebenenfalls kleiner ist.
- Verarbeitungsstufen, die Instruktionen typischerweise in einem Takt verarbeiten, werden – wenn möglich – vervielfältigt, so dass jedem Programmfaden mindestens eine eigene Einheit zur Verfügung steht.
- Ist letzteres nicht möglich oder zu teuer, so steht noch das Zeitmultiplexverfahren zur Verfügung, bei dem eine Ressource jeweils für einen bestimmte Zeitschlitz einem Programmfaden exklusiv zugeordnet ist. Nach einer festen Reihenfolge erhält jeder Programmfaden periodisch das Zugriffsrecht.

3 Simultan mehrfädige In-Order Ausführung mit Isolation

In diesem Kapitel wird die Hardware-Architektur beschrieben, die es erlaubt, mehrere echtzeitfähige Programmfäden gleichzeitig auszuführen. Ein Superskalarprozessor mit In-Order-Ausführung wird dabei so erweitert, dass mehrere Programmfäden gleichzeitig verarbeitet werden können.

Einer dieser Programmfäden wird derart isoliert, dass seine Ausführungszeit nicht von anderen, gleichzeitig bereit stehenden Programmfäden beeinflusst werden kann. Dennoch soll seine Ausführung nicht verzögert werden, so dass der gleiche Programmcode, auf der zu Grunde liegenden einfädigen Architektur ausgeführt, nur unwesentlich schneller sein darf. Deshalb erhält dieser Programmfaden gleichzeitig die höchste Priorität. Er wird im weiteren Verlauf *Dominanter Hardware-Faden (DHF)* genannt.

Während in diesem Kapitel die allgemeinen Maßnahmen beschrieben werden, die nötig sind, um Mehrfädigkeit mit gleichzeitiger Isolation zu erreichen, werden diese in Kapitel 5 an einer konkreten Prozessorarchitektur verdeutlicht.

3.1 Erweiterung eines Superskalarprozessors

Mit den im Folgenden beschriebenen Techniken kann jeder beliebige einfädige Superskalarprozessor mit In-Order-Ausführung zu einem SMT Prozessor erweitert werden. Auch auf skalare Prozessoren, das heißt Prozessoren, die nur eine Instruktion pro Takt ausführen können, ist das Verfahren anwendbar, jedoch kann der resultierende Prozessor ebenfalls nur eine Instruktion pro Takt ausführen, er ist damit kein SMT Prozessor im eigentlichen Sinn, sondern ein mehrfädiger Skalarprozessor.

Für jeden zusätzlich ausführbaren Programmfaden wird zusätzliche Hardware benötigt, das heißt, die maximale Anzahl paralleler Programmfäden ist durch die Hardware vorgegeben. Typischerweise liegt die Anzahl zwischen zwei und acht. In Kapitel 4 wird gezeigt, wie trotz Beschränkung der physikalischen Anzahl der Programmfäden, eine beliebige Anzahl virtueller Programmfäden ausgeführt werden kann. Zur Unterscheidung werden die physikalischen Programmfäden als Hardware-Fäden bezeichnet, bei den virtuellen Programmfäden spricht man dagegen von Prozessen oder dem Kontext eines Prozesses, der einem Hardware-Faden zugeordnet wird.

Abbildung 3.1 zeigt den prinzipiellen Aufbau einer Superskalar-Pipeline mit sechs Stufen,

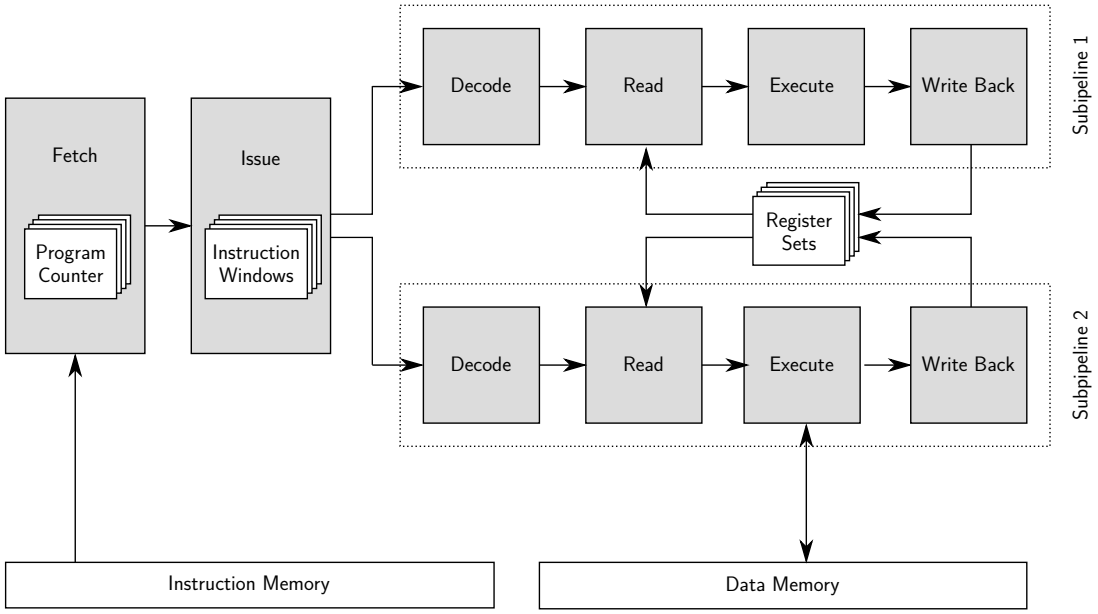

Abb. 3.1: Pipeline eines Zweifach-SMT-Prozessors mit In-Order-Ausführung

deren Funktion in Tabelle 3.1 erläutert wird. Die ersten beiden Stufen existieren nur je einmal, die restlichen Stufen gibt es mehrmals, entsprechend der Skalarität, das heißt, ein zweifach skalarer Prozessor, der maximal zwei Befehle gleichzeitig ausführen kann, spaltet sich in zwei Subpipelines, in denen jeweils eine Decode-, Read-, Execute- und Write-Back-Stufe zusammengeschlossen sind.

Um Mehrfädigkeit zu erlauben, muss der Registersatz vervielfältigt werden, da jeder Hardware-Faden einen eigenen benötigt. Über ein zusätzliches Signal, das sogenannte Tag, wird gesteuert, auf welchen Registersatz Read- und Write-Back-Stufe zugreifen. Dieses Signal wandert gemeinsam mit jeder Instruktion durch die Pipeline. Außerdem erhält jeder Hardware-Faden einen eigenen Programmzähler und ein eigenes Befehlsfenster, in dem die bereits geholten, aber noch nicht zugeordneten Instruktionen zwischengespeichert werden. Weitere Änderungen betreffen die Befehlsholstufe, die Zuordnungsstufe und die Speicherschnittstelle und werden in den folgenden Abschnitten genauer erläutert. Nur die Dekodier- und die Ausführungsstufe bleiben unverändert. Tabelle 3.1 fasst die Änderung zusammen.

Stufe	Aufgabe	Änderungen für SMT
Fetch	Befehle aus dem Speicher holen	mehrere Programmzähler
Issue	Befehle den Subpipelines zuordnen	mehrere Befehlsfenster
Decode	Befehl dekodieren	unverändert
Read	Operandenregister lesen	Tag
Execute	Ausführung des Befehls	unverändert
Write-Back	Zurückschreiben in den Registersatz	Tag

Tab. 3.1: Pipeline-Stufen einer typischen In-Order-Pipeline

3.2 Befehlsbereitstellung

In einem Superskalarprozessor ist die Befehlsholstufe (engl. Fetch-Stage) so dimensioniert, dass jederzeit genügend Befehle für die weitere Verarbeitung in der Pipeline zur Verfügung stehen. Für den DHF des In-Order-SMT-Prozessors soll das gleiche gelten, das heißt, es sollen jederzeit genügend Instruktionen des höchstprioren Programmfadens zur Verfügung stehen, sodass dessen Ausführung nicht aufgrund der Fetch-Stufe verzögert wird.

Im Normalfall werden weit weniger Instruktionen des DHF verarbeitet als im Extremfall, für den die Fetch-Stufe ausgelegt ist. Deshalb stände die Fetch-Stufe häufig still, würde sie nicht für das Holen von Befehlen niederpriorer Hardware-Fäden genutzt. Allein dieses nachrangige Befehlsholen reicht aus, um die niederprioren Hardware-Fäden mit ausreichend Instruktionen zu versorgen (siehe Evaluierung, Abschnitt 6.3.3)

3.2.1 Breite des Speicherzugriffs

Die Zuordnungsstufe verarbeitet im Idealfall so viele Instruktionen, wie Subpipelines vorhanden sind. Mit einer maximalen Instruktionsbreite von B Bit und P Subpipelines ergibt sich ein Durchsatz F von

$$F = B \cdot P \tag{3.1}$$

Bit pro Takt. Dieser gilt für die Issue-Stufe und deshalb in gleicher Weise für die Fetch-Stufe, da letztere diese Bandbreite zur Verfügung stellen muss. Wenn in jedem Takt ein neuer Speicherzugriff durchgeführt werden kann, das heißt die Speicherlatenz L_{fetch} beträgt null, dann entspricht F der Breite des Speicherzugriffs in Bit. Falls zwischen dem Anlegen der Adresse und der Lieferung des gelesenen Datums jedoch $L_{fetch} > 0$ Takte liegen, so muss bei einem Speicherzugriff ein Vielfaches des Durchsatzes angefordert werden:

$$Z = (L_{fetch} + 1) \cdot F \tag{3.2}$$

Neben dieser Fetch-Breite Z gibt es eine weitere Größe, die die Fetch-Stufe charakterisiert und aus vorgegebenen Parametern der Mikroarchitektur errechnet werden kann, die Größe des Befehlsfensters G.

3.2.2 Größe des Befehlsfensters

Das Befehlsfenster speichert die bereits geholten Instruktionen solange, bis die nachfolgende Pipeline-Stufe sie verarbeiten kann. Ist es zu klein, so führt das zu Verzögerungen in der Ausführung des DHF, ist es zu groß, so wird die Fetch-Stufe zu oft vom DHF belegt und die anderen Hardware-Fäden werden nicht mit genügend Instruktionen versorgt. Theoretisch wäre es möglich, dem DHF ein größeres Befehlsfenster zuzuweisen als den niederprioren Hardware-Fäden. Dann wäre der DHF aber auf einen Hardware-Faden festgelegt und ein schneller Kontextwechsel (eine Stärke einer mehrfädigen Architektur,

siehe Abschnitt 4.1) wäre für Programmfäden mit harten Echtzeitanforderungen, die den DHF belegen, nicht möglich, ganz abgesehen von der größeren Komplexität des Hardware-Entwurfs.

Zur Berechnung der optimalen Größe des Befehlsfensters genügt es, nur einen Programmfaden mit maximalem Durchsatz zu betrachten, bei dem in jedem Takt jeder Subpipeline eine neue Instruktion zugeordnet wird. Die Anzahl der Takte $L_{F,I}$, die nötig ist, um die Zeit zwischen der Entscheidung für einen Speicherzugriff bis zur Ankunft des Datums im Befehlsfenster zu überbrücken, multipliziert mit dem Durchsatz F ergibt die Größe des Befehlsfensters G:

$$G = L_{F,I} \cdot F \tag{3.3}$$

Mit optimaler Hardware wird in Takt $t + 0$ in der Zuordnungsstufe die Entscheidung getroffen, welche Instruktionen an die Subpipelines weitergegeben werden. Aufgrund dieser Zuordnung entscheidet die Fetch-Stufe noch im gleichen Takt, ob neue Befehle geholt werden sollen. Ist dies der Fall, so wird während Takt $t + 1$ auf den Speicher zugegriffen und zu Beginn von Takt $t + 2$ stehen die frisch geholten Instruktionen der Zuordnungsstufe zur Verfügung. Folglich müssen sich zu Beginn von Takt $t + 0$ die in diesem Takt zuzuordnenden Instruktionen plus die Instruktionen zur Überbrückung des Speicherzugriffs in Takt $t + 1$ befinden, zusammen $2 \cdot d$ Instruktionen. Das heißt, es müssen zwei Takte überbrückt werden, $L_{F,I} = 2$.

Zwei stellt den minimalen Wert dar, es gibt einige Faktoren, die zu einer Erhöhung von $L_{F,I}$ führen können:

- Falls es möglich ist, dass nicht immer die volle Zahl von P Instruktionen zugeordnet wird (bei nahezu jedem SMT Prozessor ist es tatsächlich möglich), muss $L_{F,I}$ um eins erhöht werden, um einen zusätzlicher Puffer für „angefangene“ Befehlspakete zu schaffen.
- Bei hohen Taktfrequenzen kann es sein, dass die dadurch vorgegebene Verarbeitungszeit pro Pipeline-Stufe zu gering ist, um die Zuordnung der Befehle und die Entscheidung für den nächsten Befehlsspeicherzugriff innerhalb eines Taktes durchzuführen. In diesem Fall müssen beide Operationen auf zwei aufeinander folgende Stufen verteilt werden. Bei sehr hohen Taktfrequenzen oder einer komplizierten Zuordnungslogik ist es sogar möglich, dass die beiden Operationen noch weiter zerlegt und auf mehrere Stufen verteilt werden müssen. Bei mehr als insgesamt einer Pipeline-Stufe muss $L_{F,I}$ entsprechend erhöht werden.
- Falls die Speicherlatenz L_{fetch} des Befehlsspeichers größer als null ist, muss sie zu $L_{F,I}$ addiert werden.

Durch eine genauere Analyse und Vordekodierung der bereits geholten Instruktionen kann unter gewissen Umständen ein erheblicher Teil der Speicherzugriffe auf den Instruktionsspeicher eingespart werden, ohne die Zuordnungsgeschwindigkeit zu verringern. Dies ist jedoch sehr vom genauen Aufbau der Pipeline und dem jeweiligen Befehlssatz abhängig. Zwei mögliche Optimierungen im Rahmen der CarCore-Architektur finden sich in Abschnitt 5.4.2.

3.3 Befehlszuordnung

Die Zuordnungsstufe ist der Dreh- und Angelpunkt der echtzeitfähigen SMT-Pipeline und stellt gleichzeitig die erste Ebene des echtzeitfähigen Hardware-Schedulers dar. Sie liest Befehle aus den Befehlsfenstern und ordnet sie den entsprechenden Subpipelines zu. Dies geschieht anhand der Prioritäten, die von der zweiten Scheduler-Ebene bereitgestellt werden.

3.3.1 Vordekodierung

Die Fetch-Stufe liest feste Bitfolgen und schreibt diese in das Befehlsfenster des jeweiligen Hardware-Fadens. Von dort werden sie durch die Zuordnungsstufe entnommen und vordekodiert. Dies ist nötig, da Befehle durch unterschiedlich lange Bitfolgen kodiert werden können und nur bestimmte Kombinationen von Instruktionen gleichzeitig in verschiedenen Subpipelines ausgeführt werden können. Durch die Vordekodierung wird der Bitstrom analysiert und in die einzelnen Instruktionen aufgeteilt. Anschließend werden Abhängigkeiten zwischen den Instruktionen ermittelt und diejenigen Instruktionen weitergeleitet, die parallel in einem Takt ausgeführt werden können.

Dabei ist zu beachten, dass die Reihenfolge der Instruktionen nicht verändert wird, das heißt, sobald eine Instruktion eine Abhängigkeit von einer anderen, sich noch im Befehlsfenster befindlichen Instruktion hat, ist das Ende der Folge von gleichzeitig zuordenbaren Instruktionen erreicht. Dies vereinfacht das Verfahren und die nötige Hardware erheblich, ohne den Gesamtdurchsatz merklich zu beeinflussen (vgl. Abschnitt 2.4.2).

Durch eine geschickte Kodierung der Instruktionen kann die Ermittlung der parallel ausführbaren Instruktionen stark vereinfacht werden. Beispielsweise kann ein zusätzliches Bit an jede Instruktion angehängt werden, um anzuzeigen, ob Instruktionen zusammengehören und deshalb gemeinsam ausgeführt werden können. Dieses sogenannte Stop-Bit ist gelöscht, wenn noch weitere gleichzeitig ausführbare Instruktionen folgen, nur bei der letzten Instruktion einer gleichzeitig ausführbaren Gruppe ist es gesetzt. Dieses Verfahren wird bei der EPIC-Befehlssatzarchitektur des Intel Itanium Prozessors verwendet [Schlansker 2000].

Eine andere Möglichkeit ist, die Instruktionen so zu kodieren, dass dem Bitmuster entnommen werden kann, in welcher Subpipeline die Instruktion ausgeführt werden soll. Sobald eine Subpipeline zum zweiten Mal auftaucht, ist die gleichzeitig ausführbare Instruktionsgruppe beendet. Noch einfacher ist die Prüfung, wenn die Instruktionen innerhalb der gleichzeitig ausführbaren Gruppe nach aufsteigender Nummer der Subpipeline sortiert werden. Dann ist das Ende einer Gruppe erreicht, sobald eine Nummer nicht größer als ihr Vorgänger ist. Der Befehlssatz des Infineon Tricore, folglich auch der Befehlssatz des noch vorzustellenden CarCore Prototyps, hat diese Eigenschaft, siehe Abschnitt 5.

Ein Extremfall ist ein sogenannter VLIW-Befehlssatz (von engl. *very long instruction*

Algorithmus 1 Pseudocode der Zuordnungsvorschrift

Eingabe: Anzahl Hardware-Fäden N, Anzahl Subpipelines P
Hardware-Faden τ_0 hat die höchste Priorität, τ_{N-1} die niedrigste
Ausgabe: Zuordnung von Instruktionen auf Subpipelines in π_p

```
π_p ← ∅   ∀0 ≤ p < P
for 0 ≤ i < N do
  instr ← nächste Instruktion von τ_i
  for 0 ≤ p < P do
    if (Subpipeline von instr = p) ∧ (π_p = ∅) then
      π_p ← instr
      instr ← nächste Instruktion von τ_i
    end if
  end for
end for
```

word). Hier wird pro Takt für jede Subpipeline genau eine Instruktion fester Länge geholt und zugeordnet. Dadurch entfällt die Vordekodierung, jedoch muss, falls für eine Subpipeline kein passender Befehl zur Verfügung steht, eine leere Instruktion (engl. *no operation, nop*) eingefügt werden. Abgesehen davon, dass diese Befehlssatzvariante sehr verschwenderisch mit dem Instruktionsspeicher umgeht, ist sie für SMT-Prozessoren nicht geeignet. VLIW-Programmcode wird so optimiert, dass möglichst alle Subpipelines immer belegt sind, die Stärke der SMT Programmausführung, freie Ressourcen durch andere Programmfäden zu nutzen, kommt dadurch kaum zum Tragen.

3.3.2 Zuordnungsalgorithmus

Nach der Vordekodierung stellt im Idealfall jeder Hardware-Faden soviele Instruktionen zur Verfügung, wie Subpipelines vorhanden sind. Daraus wird entsprechend der Zuordnungsvorschrift (Algorithmus 1) für jede Subpipeline eine Instruktion ausgewählt. Dies geschieht gemäß den Prioritäten, das heißt, alle Instruktionen des Hardware-Fadens mit der höchsten Priorität werden sofort zugeordnet, dann erst werden die verbleibenden Subpipelines mit Instruktionen aufgefüllt. Auch dann wird streng nach absteigender Priorität verfahren, das heißt, zuerst wird der Hardware-Faden mit der zweithöchsten Priorität betrachtet, dann der mit der dritthöchsten Priorität, und so weiter.

Dabei ist zu beachten, dass die Instruktionen auch weiterhin nur in ihrer ursprünglichen Reihenfolge zugeordnet werden dürfen. Die Instruktionen eines Hardware-Fadens werden in der Reihenfolge, in der sie im Programmcode stehen, nacheinander abgearbeitet. Sobald die zugehörige Subpipeline einer Instruktion bereits belegt ist, kann keine weitere Instruktion des entsprechenden Hardware-Fadens mehr zugeordnet werden. Es wird stattdessen mit dem Hardware-Faden mit nächstniedrigerer Priorität fortgefahren.

Eine wichtige Eigenschaft der hier beschriebenen Zuordnungsstufe ist es, dass die Prioritäten jeweils nur für einen Takt festgelegt sind. Über externe Signale kann die Zuord-

nungsreihenfolge der Hardware-Fäden in jedem Takt beliebig verändert werden. Dadurch wird es möglich, dass durch ein weiteres Hardware-Modul, den sogenannten Scheduler (genauer gesagt die zweite Ebene des Schedulers, die in Kapitel 4 beschrieben wird), die Ausführung sehr präzise beeinflusst werden kann und fortgeschrittene, komplexe Scheduling-Algorithmen realisiert werden können.

3.3.3 Mehrtaktausführung

Eine weitere Aufgabe der Zuordnungsstufe besteht darin, Instruktionen, deren Ausführung mehr als einen Takt dauern, zu verwalten. Mit mehreren Takten ist nicht die Ausführung in mehreren hintereinander liegenden Pipeline-Stufen gemeint, sondern das mehrmalige Belegen der gleichen Pipeline-Stufe. Dies kann nötig sein, wenn die Berechnungen im Rahmen einer Instruktion sehr umfangreich sind oder auf sehr viele Register zugegriffen werden muss.

Dieses Problem stellt sich auch bei einfädigen und skalaren Prozessoren und die Lösung ist die gleiche: die Instruktion wird in eine Folge von einfacheren Instruktionen aufgeteilt, so dass jede innerhalb eines Taktes ausgeführt werden kann. Die Instruktionsfolge, die beim Erkennen der entsprechenden Instruktion automatisch ausgeführt wird, nennt man *Mikrocodesequenz*.

Im Unterschied zu einfädigen Prozessoren muss im mehrfädigen Fall jedoch garantiert werden, dass die Sequenz nach jedem Takt unterbrochen werden kann. Dies ist nötig, da Hardware-Fäden mit niedriger Priorität die Subpipelines oft nur für einzelne Takte belegen dürfen, da sie andernfalls die Ausführung von Hardware-Fäden mit höheren Prioritäten verzögern würden. Realisieren lässt sich das durch zusätzliche Schattenregister, in denen Zwischenergebnisse, die in den einzelnen Schritten einer Mikrocodesequenz erzeugt werden, zwischengespeichert werden. Diese Schattenregister müssen, wie die eigentlichen Architekturregister auch, für jeden Hardware-Faden separat vorhanden sein.

Die Zuordnungsstufe erkennt Instruktionen, die auf Mikrocodesequenzen abgebildet werden und schaltet in diesem Fall auf einen speziellen Programmzähler, der durch die Mikrocodesequenz wandert. Erst wenn das Ende der Sequenz erreicht ist, wird die zugehörige Instruktion aus dem Befehlsfenster entfernt und weitere Instruktionen des Hardware-Fadens können ausgeführt werden.

3.3.4 Sprungbefehle

Auch Sprungbefehle werden in der Zuordnungsstufe verwaltet, da auf einen Sprungbefehl nicht automatisch die nächste Instruktion aus dem Befehlsstrom ausgeführt werden kann. Stattdessen muss abgewartet werden, bis das Sprungziel in einer späteren Pipeline-Stufe berechnet wurde und erst dann kann der dortige Befehl geholt, zugeordnet und ausgeführt werden.

Bei direkten oder indirekten Sprüngen an feste Adressen kann eine feste Sprunglatenz

L_{jump} berechnet werden, die angibt, wie viele Takte ein Hardware-Faden nach dem Erkennen eines Sprungbefehls in der Zuordnungsstufe still stehen muss, bevor die nächste gültige Instruktion im Befehlsfenster steht:

$$L_{jump} = L_{I,E} + 1 + L_{fetch} \tag{3.4}$$

Wobei $L_{I,E}$ die Anzahl der Pipeline-Stufen zwischen Zuordnungsstufe und der Stufe, in der die Zieladresse berechnet wird, ist. Diese Stufe ist normalerweise die Execute-Stufe, im Beispiel aus Abbilding 3.1 gilt folglich $L_{I,E} = 2$. Dabei wird angenommen, dass die berechnete Adresse direkt (das heißt asynchron im gleichen Takt) an die Fetch-Logik weitergeleitet wird und bereits im nächsten Takt der Speicherzugriff erfolgen kann. Andernfalls müssen weitere Takte addiert werden.

Etwas komplizierter wird es bei bedingten Sprüngen, da hier zwei Fälle unterschieden werden müssen, je nachdem ob der Sprung genommen wird (engl. *taken*) oder ob die Ausführung an der aktuellen Stelle fortgeführt wird (engl. *not taken*). Wenn die Entscheidung, ob gesprungen wird oder nicht, vor oder gleichzeitig mit der Zieladressberechnung stattfindet, so unterscheidet sich der erste Fall nicht von einem unbedingten Sprung, beim zweiten Fall entfällt jedoch das Holen von Befehlen, da diese noch im Befehlsfenster vorhanden sind.

Die beiden Fälle und die unbedingten Sprünge lassen sich zusammenführen, indem die Zuordnungsstufe nur solange gesperrt wird, bis die Zieladresse berechnet ist und dann ganz normal weiterarbeitet:

$$L_{branch} = L_{I,E} + 1 \tag{3.5}$$

Bei Sprüngen, deren Bedingung nicht erfüllt ist, führt dies allein zum gewünschten Verhalten, bei genommenen und unbedingten Sprüngen muss in der Pipeline-Stufe, in der die neue Adresse berechnet wird, zusätzlich das Befehlsfenster gelöscht werden, sodass die Zuordnungsstufe im darauf folgenden Takt zwar nicht mehr gesperrt ist, aber aufgrund des leeren Befehlsfensters trotzdem warten muss, bis das Holen des Befehls abgeschlossen ist.

Da bei Sprungbefehlen für mehrere Takte keine Instruktionen ausgeführt werden, ist in den meisten modernen Prozessoren eine Sprungvorhersage [Brinkschulte 2010] eingebaut, die versucht, die Entscheidung der Sprungbedingung vorwegzunehmen und spekulativ einen der beiden Zweige (Sprung genommen oder nicht genommen) ausführt. Dies spart im Normalfall sehr viele ungenutzte Latenztakte ein, da diese Technik inzwischen sehr hoch entwickelt ist und hohe Trefferraten erreicht. Ist die Vorhersage jedoch falsch, so müssen die spekulativ ausgeführten Instruktionen rückabgewickelt werden und die Ausführung beginnt von Neuem beim Alternativzweig.

Bei einfädigen Prozessorarchitekturen ist es belanglos, ob die Prozessor-Pipeline still steht oder ob Instruktionen ausgeführt werden, die später wieder verworfen werden. Bei einer mehrfädigen Architektur macht es aber sehr wohl einen Unterschied aus, da im letzteren Fall die Pipeline nicht für die Ausführung von anderen Programmfäden genutzt werden kann.

Darüber hinaus stellt die Sprungvorhersage ein großes Problem bei der Laufzeitanalyse dar [Heckmann 2003], da nicht die durchschnittliche, sondern die längstmögliche Ausführungszeit gesucht wird. Das kann bedeuten, dass die WCET ohne Sprungvorhersage kleiner ist als mit Sprungvorhersage, falls eine falsche Vorhersage eine zusätzliche Verzögerung aufgrund des Rückabwickelns erfordert.

Wegen dieser beiden Argumente wird in der hier vorgestellten echtzeitfähigen SMT Architektur auf eine Sprungvorhersage völlig verzichtet.

3.4 Speicherzugriffe

In einer klassischen Prozessor-Pipeline gibt es üblicherweise eine eigene Stufe für Speicherzugriffe, die zwischen Ausführungs- und Zurückschreibestufe angesiedelt ist. In dem Takt, der der Speicherstufe zugeordnet ist, wird lesend oder schreibend auf den Speicher zugegriffen, die zugehörige Adresse wird in der vorhergehenden Ausführungsstufe berechnet. Dieses Modell geht von der Annahme aus, dass ein Speicherzugriff innerhalb eines Taktes durchgeführt werden kann. In der Realität ist letzteres aber kaum zu verwirklichen, da die großen Speichermengen aktueller Prozessoren auch längere Zugriffszeiten nach sich ziehen.

In einem einfädigen Prozessor wird bei einem längeren Speicherzugriff einfach die Pipeline angehalten, daß heißt es werden keine Instruktionen mehr von einer zur nächsten Pipeline-Stufe weitergereicht, bis der Speicherzugriff abgeschlossen ist, erst dann läuft sie wieder normal weiter. In einem mehrfädigen Prozessor ist das keine akzeptable Lösung, da dadurch nicht nur der betroffene Hardware-Faden blockiert wird, sondern auch alle anderen.

3.4.1 Mehrere Speicherstufen

Eine Alternative wäre es, die Anzahl der Speicherstufen erhöhen, sodass der Speicherzugriff zum Ende der letzten Speicherstufe abgeschlossen wäre. Hierbei müssen jedoch mögliche Datenkonflikte beachtet werden. Ein Datenkonflikt kann schon bei nur einer Speicherstufe auftreten, dann nämlich, wenn eine direkt auf einen Lesezugriff folgende Instruktion den zu lesenden Wert sofort weiterverarbeiten will. Diese Instruktion erwartet den Wert zu Beginn der Ausführungsstufe, zu diesem Zeitpunkt befindet sich der Lesezugriff aber noch in der Speicherstufe, das heißt, das Datum steht erst einen Takt später zur Verfügung.

Bei einer Speicherlatenz von null, das heißt, wenn eine Speicherstufe für den Zugriff ausreicht, spielt der Datenkonflikt noch keine große Rolle und kann akzeptiert werden, da er bei Schreibzugriffen nicht auftritt und bei Lesezugriffen nur, wenn die direkt folgende Instruktion das Zielregister sofort ausliest. Daher ist es möglich, einen potentiellen Datenkonflikt in der Vordekodierung der Zuweisungsstufe zu erkennen und bei Bedarf die Ausführung um einen Takt zu verzögern.

Sind jedoch mehr Speicherstufen vorhanden, so verlängert sich auch die „Reichweite“ des Datenkonflikts, für jede zusätzliche Speicherstufe muss eine weitere Instruktion im Anschluss an den Lesezugriff auf einen Datenkonflikt hin überprüft werden. Außerdem führen in diesem Fall auch schreibende Speicherzugriffe zu Pausen in der Ausführung, da sich andernfalls die Schreibanforderungen am Speichercontroller anhäufen, wenn jeden Takt eine neue Anforderung kommt, aber nicht in jedem Takt eine abgeschlossen wird. Folglich müsste bei einer Speicherlatenz von L_{mem} Takten die Pipeline $L_{mem} + 1$ Speicherstufen enthalten und der entsprechende Hardware-Faden müsste nach einem lesenden Speicherzugriff für $L_{mem} + 1$ Takte und nach einem schreibenden Speicherzugriff für L_{mem} Takte angehalten werden.

3.4.2 Split Phase Load

Abgesehen von der mangelnden Flexibilität der Mehrstufenlösung (es können nur Speicherzugriffe mit einer festgelegten Latenz verarbeitet werden) sind zusätzliche Pipeline-Stufen sehr teuer und führen keine Funktion aus, sondern warten nur auf den Speichercontroller. Deshalb kann auf sie verzichtet werden, solange die nötigen Verzögerungen eingehalten werden und der Speichercontroller nicht mit konkurrierenden Anfragen belastet wird.

Genau dieser Ansatz wird mit der sogenannten *Split-Phase-Load-Technik* [Boothe 1992] verfolgt: Lesebefehle werden in zwei Phasen geteilt, in der ersten Phase wird die Speicheradresse berechnet, in der zweiten wird das gelesene Datum in den Registersatz zurückgeschrieben. Zwischen den beiden Phasen liegt eine variable Anzahl von Takten, die von der Speicherlatenz und den konkurrierenden Hardware-Fäden abhängt. Bei einer Instruktion, die in den Speicher schreibt, kann die zweite Phase entfallen, da nichts in den Registersatz geschrieben wird.

Diese Aufteilung in zwei Phasen wird auch bei Out-of-Order-Prozessoren angewandt [Raasch 2003], jedoch ist die Hardware-Realisierung völlig anders. In Out-of-Order-Prozessoren ist das Warten auf das Ende der Ausführung einer Instruktion ein inhärentes Architekturmerkmal, wohingegen in einer In-Order-Architektur nur bei bestimmten Instruktionen explizit gewartet werden muss.

Wenn die Zuordnungsstufe eine Instruktion erkennt, die in den Speicher schreibt, so wird diese Instruktion ganz normal zugeordnet und durchläuft die Pipeline, liest die nötigen Registerwerte aus, berechnet in der Ausführungsstufe die Zieladresse und gibt sie anschließend zusammen mit dem zu schreibenden Wert an den Speichercontroller weiter. Die Zuordnungsstufe setzt jedoch sofort nach der Zuordnung des Schreibbefehls die weitere Ausführung von Instruktionen des entsprechenden Hardware-Fadens aus, bis vom Speichercontroller signalisiert wird, dass der Schreibzugriff abgeschlossen wurde. Erst dann wird der betroffene Hardware-Faden bei der Ermittlung der Zuordnung wieder berücksichtigt und die Ausführung dieses Hardware-Fadens wird fortgesetzt. Dadurch wird verhindert, dass mehrere Schreibbefehle eines Hardware-Fadens am Speichercontroller auflaufen.

Andererseits kann dadurch nach einem Speicherzugriff für mehrere Takte keine Instruktion des gleichen Hardware-Fadens ausgeführt werden, da die Zuordnungsstufe erst auf die Bestätigung durch den Speichercontroller warten muss. Selbst wenn letzterer sofort nach Erhalt der Adresse diese bestätigt und erst dann mit dem eigentlichen Speicherzugriff beginnt, kommt es zu einer Verzögerung, da die Adresse die Pipelinestufen von der Zuordnungsstufe bis zum Speichercontroller überwinden musste.

Diese Latenz $L_{I,M}$ ist unvermeidlich und führt dazu, dass, selbst wenn die Speicherzugriffslatenz L_{mem} null beträgt, also in jedem Takt auf eine neue Adresse zugegriffen werden kann, die Ausführung des entsprechenden Hardware-Fadens für $L_{I,M}$ Takte unterbrochen wird. Für die einfädige Ausführungsgeschwindigkeit bringt es daher keinen Vorteil, wenn die Speicherzugriffslatenz L_{mem} kleiner als die beschriebene Totzeit $L_{I,M}$ ist. Bei mehrfädiger Ausführung ist eine kleinere Speicherlatenz jedoch trotzdem von Vorteil, da währenddessen Speicherzugriffe anderer Hardware-Fäden durchgeführt werden können.

Ist die Speicherlatenz L_{mem} größer als der Abstand der Pipelinestufen $L_{I,M}$, so darf das Signal für die Fortsetzung der Programmausführung nicht direkt beim Beginn des Speicherzugriffs stattfinden, sondern erst $L_{I,M}$ Takte vor Abschluss des Zugriffs. Dadurch ist gewährleistet, dass in dem Fall, dass zwei Speicherzugriffe direkt aufeinander folgen, der zweite genau in dem Takt am Speichercontroller ankommt, wenn der erste diesen gerade verlassen hat, es kommt zu keiner Verzögerung.

Bei einer Instruktion, die lesend auf den Speicher zugreift, wird zwar ebenfalls die Instruktion durch die Pipeline geschickt und die Verarbeitung des betreffenden Hardware-Fadens blockiert, bis der Speichercontroller ihn wieder frei gibt, jedoch ist es damit nicht getan, denn das gelesene Datum muss nach erfolgtem Speicherzugriff noch in den Registersatz geschrieben werden.

Dies erfolgt durch die zweite Phase des Ladebefehls, eine weitere Instruktion, die nach Ankunft des Freigabesignals, vor Ausführung der nächsten regulären Instruktion, durch die Pipeline geschickt wird. Sie hat nur die Aufgabe, das gelesene Datum zu empfangen und in der Rückschreibstufe in das passende Register zu schreiben.

3.4.3 Adresspuffer

Die bisherigen Maßnahmen sorgen zwar dafür, dass die Wartezeit auf einen Speicherzugriff die anderen Hardware-Fäden nicht ebenfalls blockiert, jedoch wurde noch nicht auf den Einfluss von Hardware-Fäden untereinander eingegangen. Sind mehrere Hardware-Fäden vorhanden, so können auch mehrere Speicherzugriffe beim Speichercontroller ankommen, da die Verarbeitung eines Zugriffes üblicherweise mehrere Takte dauert.

Man könnte zwar (wie bei den Speicherzugriffen eines Hardware-Fadens) schon in der Zuordnungsstufe dafür sorgen, dass nur jeweils ein Speicherzugriff den Speichercontroller erreicht, jedoch würden dadurch die Speicherzugriffe unnötig verzögert. Hier spielt die oben bereits angesprochene Latenz zwischen Zuordnung und Adressberechnung $L_{I,M}$ eine

große Rolle, da nach einem abgeschlossenen Speicherzugriff die nächste Instruktion mit Speicherzugriff zunächst durch die Pipeline laufen muss und der Speichercontroller erst nach der Adressberechnung in der Ausführungsstufe wieder beschäftigt wird, dazwischen steht er für $L_{I,M}$ Takte still.

Die bessere Lösung besteht in sogenannten *Adresspuffern*. Für jeden Hardware-Faden gibt es einen Adresspuffer, der genau eine Adresse aufnehmen kann. Dadurch ist es möglich, dass alle Hardware-Fäden kurz hintereinander Speicherzugriffe zum Speichercontroller schicken, ohne Rücksicht auf die anderen Hardware-Fäden zu nehmen. In der Zuordnungsstufe ist keine zusätzliche Hardware nötig, da die Hardware-Fäden ja bereits auf die Bestätigung des Speicherzugriffs durch den Speichercontroller warten. Es muss nur sichergestellt werden, dass bei der Bestätigung durch den Speichercontroller auch das Tag des Hardware-Fadens mitgeliefert wird, um bei mehreren wartenden Hardware-Fäden den Richtigen wieder freizugeben.

Zusätzlich kann der Speicherplatz im Adresspuffer zur Zwischenspeicherung nicht abgeholter Lesezugriffe verwendet werden. Da die zweite Phase von Lesebefehlen auf die gleiche Weise zugeordnet wird wie alle anderen Instruktionen, kann es sein, dass das gelesene Datum eines Hardware-Fadens mit niedriger Priorität zwar schon bereitsteht, aber zuerst andere Instruktionen von Hardware-Fäden mit höherer Priorität zugeordnet werden.

Um die Zeit bis zur tatsächlichen Abholung des Datums durch die zweite Phase des Split-Phase-Loads zu überbrücken, muss das Datum zwischengespeichert werden. Dafür ist pro Hardware-Faden ein Puffer in der Größe der maximalen Speicherzugriffsbreite erforderlich. Da sich die Zeiten, in denen dieser Puffer beziehungsweise der Adresspuffer benötigt werden, nicht überschneiden, reicht ein gemeinsamer Puffer aus. Seine Breite richtet sich danach, ob mehr Bits zur Speicherung einer Adresse oder zur Speicherung eines gelesenen Wertes nötig sind.

3.4.4 Arbitrierung

Bisher blieb der Echtzeitaspekt beim Design der Speicherschnittstelle weitgehend unbeachtet, er spielt aber eine äußerst wichtige Rolle. Eine erste Maßnahme ist die Abarbeitung der Adresspuffer nach den gleich Prioritäten, die für die Zuordnungsstufe gelten, das heißt, wenn der Speichercontroller mehr als einen Speicherzugriff zur Auswahl hat, so wählt er denjenigen mit der höchsten Priorität aus.

Durch diese Maßnahme wird der Dominante Hardware-Faden (DHF) schon erheblich bevorzugt, jedoch reicht dies unter Umständen noch nicht aus, da nicht völlig verhindert wird, dass Hardware-Fäden mit niedriger Priorität die Ausführung des DHF verzögern. Wenn ein Speicherzugriff des DHF am Speichercontroller ankommt, kann es sein, dass der Speichercontroller noch nicht bereit ist, da kurz vorher ein Hardware-Faden mit niedrigerer Priorität einen Speicherzugriff begonnen hat, dessen Bearbeitung noch nicht abgeschlossen ist.

Zwar wird durch die priorisierte Abarbeitung der Adresspuffer garantiert, dass der nächste Speicherzugriff der des DHF ist, aber im ungünstigsten Fall (wenn jeweils genau im Takt vor dem DHF Zugriff ein anderer Speicherzugriff beginnt) wird der DHF bei jedem Speicherzugriff zusätzlich um L_{mem} Takte[1] verzögert. Bei Berechnung der WCET muss folglich nicht L_{mem} als Speicherlatenz angenommen werden, sondern der doppelte Wert $2 \times L_{mem}$.

Eine andere Möglichkeit besteht darin, die Speicherzugriffe in einem festen Raster auszuführen, was allerdings nur dann sinnvoll ist, wenn alle Zugriffe gleich lang dauern. Im Raster ausführen bedeutet, dass nur alle $L_{mem}+1$ Takte ein Speicherzugriff durchgeführt wird. Steht zu diesem Zeitpunkt kein Zugriffswunsch in den Adresspuffern, so steht der Speichercontroller für die nächsten $L_{mem} + 1$ Takte still.

Auf den ersten Blick hat dieses Verfahren keinen Vorteil, da im ungünstigsten Fall (wenn der DHF Speicherzugriff jeweils im ersten Takt nach der Überprüfung ankommt), wiederum L_{mem} Takte gewartet werden muss, bis ein neuer Speicherzugriff begonnen wird. Es besteht aber die Möglichkeit, den Code mit dem Raster zu synchronisieren, das heißt, den Code (bei bekanntem L_{mem}) so anzuordnen, dass die Speicherzugriffe genau alle $L_{mem}+1$ Takte ausgeführt werden. Dadurch kann die zusätzliche Verzögerung nur beim ersten Speicherzugriff auftreten, alle weiteren Speicherzugriffe, die im Raster bleiben, werden nicht verzögert, da der DHF die höchste Priorität hat und immer zum genau richtigen Zeitpunkt den Speichercontroller anfordert. Dennoch scheint dieses Verfahren nicht besonders praktikabel, da die Speicherlatenz bei der Kompilierung bekannt sein muss und ein Zusammenspiel von Compiler und Laufzeitanalyse nötig ist, das in derzeitigen Systemen nicht vorgesehen ist.

3.4.5 Zugriffsankündigung

Bei einer dritten Variante werden die Speicherzugriffe des dominanten Hardware-Fadens vorzeitig angekündigt (engl. *Dominant Memory Access Announcing, DMAA*) [Mische 2008]. Hierbei macht man sich zunutze, dass die Zuordnung eines Speicherzugriffes einige Takte vor der Berechnung der Adresse stattfindet, das heißt schon einige Takte bevor die Speicheranfrage beim Speichercontroller eintrifft ist bereits bekannt, zu welchem Zeitpunkt der dominante Speicherzugriff stattfinden wird.

Wenn die Zuordnungsstufe einen Speicherbefehl des DHF zuordnet, teilt sie dies gleichzeitig dem Speichercontroller mit. Da die Instruktion erst einige Takte später beim Speichercontroller ankommt, weiß dieser schon im Voraus, wann ein dominanter Speicherzugriff ankommen wird und beginnt keine neuen Speicherzugriffe, wenn sie nicht rechtzeitig vor dem dominanten Speicherzugriff abgeschlossen sind.

Durch die Ankündigung wird die maximale Latenz, um die ein dominanter Speicher-

[1] L_{mem} gibt die Anzahl der Takte an, die zusätzlich zum ersten Takt für einen Speicherzugriff benötigt werden. Ein Speicherzugriff belegt also für $L_{mem} + 1$ Takte den Speichercontroller und verzögert darum maximal um L_{mem} Takte.

zugriff zusätzlich verzögert werden kann, verringert. Diese Verringerung in Takten entspricht der Anzahl von Stufen zwischen Zuordnung und Speichercontroller $L_{I,M}$. Ist der Abstand zwischen den Stufen größer oder gleich der Speicherlatenz L_{mem}, kann eine Verzögerung durch Hardware-Fäden mit niedrigerer Priorität sogar ganz vermieden werden, der DHF wird dann so schnell ausgeführt, als würde er auf einem einfädigen Prozessor ausgeführt.

Ein Nachteil entsteht aber für die Hardware-Fäden mit niedrigerer Priorität. Da zwischen der Ankündigung und dem Eintreffen des dominanten Speicherzugriffs der Speichercontroller einige Takte still steht, werden entsprechend weniger nicht-dominante Speicherzugriffe ausgeführt und die Ausführungsgeschwindigkeit der Hardware-Fäden mit niedrigen Prioritäten nimmt ab. Eine detaillierte Untersuchung dieses Effekts findet sich in Abschnitt 6.3.2 der Evaluierung.

4 Ablaufplanung

In diesem Kapitel wird die zweite und dritte Ebene des Schedulers beschrieben. Die dritte Ebene besteht aus dem sogenannten TCB-Speicher, er stellt die Programmierschnittstelle des Schedulers dar. In diesem Speicherbereich werden die Kontexte der Programmfäden gesichert und der Scheduler wird durch Zugriffe auf diesen Speicherbereich gesteuert.

Die zweite Ebene verbindet die Speicherschnittstelle mit der Prozessor-Pipeline und bildet damit das Herz des Schedulers. In dieser Ebene werden die abstrakten Scheduling-Informationen ausgewertet und in konkrete Prioritäten für die unterste Scheduler-Ebene übersetzt. Außerdem wird entschieden, welche Programmfäden in physikalisch vorhandenen Hardware-Fäden ausgeführt und welche vorübergehend angehalten und im TCB-Speicher zwischengespeichert werden.

Dieses Kapitel ist folgendermaßen aufgebaut: zuerst wird der spezielle Mechanismus für das Auslagern von Kontexten in mehrfädigen Prozessoren erläutert. Es folgt eine Darstellung der Scheduling-Algorithmen für unterschiedliche Echtzeitanforderungen und anschließend eine detaillierte Beschreibung der Hardware-Implementierung. Abgerundet wird das Kapitel durch Beispiele, die veranschaulichen, wie die einzelnen Komponenten zusammenarbeiten.

4.1 Hardwaregestützter Kontextwechsel

Die Anzahl der von der Hardware zur Verfügung gestellten Programmfäden ist auch bei mehrfädigen Architekturen grundsätzlich beschränkt, da für jeden Programmfaden der gesamte Kontext (das heißt alle Werte aller Register) im Prozessor gespeichert werden muss. Die maximale Anzahl Hardware-Fäden wird spätestens bei der Herstellung des Prozessorchips festgelegt.

Um trotzdem eine beliebig große Anzahl Programmfäden benützen zu können, bedient man sich üblicherweise eines sogenannten Softwareschedulers, der die Kontexte von gerade nicht ausgeführten Programmfäden in den Hauptspeicher auslagert und bei Bedarf wieder in den prozessorinternen Kontext eines Hardware-Fadens einlagert. Durch dieses Verfahren ist die maximale Anzahl Programmfäden nur durch die Größe des Hauptspeichers beschränkt.

Jedoch hat das Ein- und Auslagern per Software den Nachteil, dass sehr viel Rechenzeit dafür verwendet werden muss. Diese steht damit nicht mehr der eigentlichen Ausführung

der Programmfäden zur Verfügung und verringert den Gesamtdurchsatz.

4.1.1 Alternative Techniken

Beim naiven hardwaregestützten Kontextwechsel wird die Prozessor-Pipeline einfach für einige Takte blockiert. Während dieser Zeit wird zuerst der Registersatz des alten Hardware-Fadens in den Speicher geschrieben und anschließend der neue Kontext aus dem Speicher eingelesen. Die Dauer eines Kontextwechsels Δt_{CS} ergibt sich folglich aus dem doppelten Produkt aus Registeranzahl n_{regs} und Speicherzugriffsdauer, die wiederum der Speicherlatenz plus einen Takt für den Zugriff ($L_{mem} + 1$) entspricht:

$$\Delta t_{CS} = \Delta t_{out} + \Delta t_{in} = 2 \cdot n_{regs} \cdot (L_{mem} + 1) \tag{4.1}$$

Es gibt jedoch raffiniertere Methoden, mit denen die Dauer deutlich reduziert werden kann. Sie sind in Abbildung 4.1 skizziert und werden im Folgenden kurz erläutert.

4.1.1.1 Kontext-Cache

Bei Verwendung eines Kontext-Caches [Omondi 1998] werden nicht alle Registerinhalte beim Laden eines Programmfadens automatisch in den Prozessor transferiert, vielmehr wird jedes einzelne Register erst geladen, wenn wirklich darauf zugegriffen wird. Bei Auslagern eines Hardware-Fadens in den Speicher werden nur diejenigen Register zurückgeschrieben, die verändert wurden.

Dadurch können bei gleicher physikalischer Registeranzahl mehr Programmfäden im Prozessor gehalten werden, da jeder Programmfaden nicht die volle Anzahl an Registern belegt. Wird ein neuer Programmfaden in den Prozessor geladen, so werden die Register, die am längsten nicht mehr angesprochen wurden, in den Speicher ausgelagert.

Dieses Verfahren ist sehr flexibel und ermöglicht sehr schnelle Kontextwechsel, jedoch hängt der Zustand des Kontext-Caches von allen aktiven Hardware-Fäden ab, sodass sein Zustand nicht unabhängig von parallelen Programmfäden berechenbar ist. Deshalb ist eine Laufzeitanalyse kaum möglich und die Technik daher nicht hart echtzeitfähig.

4.1.1.2 Balanced Multithreading

Das sogenannte *Balanced Multithreading* [Tune 2004] ist eine Methode, um feinkörnige und grobkörnige Mehrfädigkeit zu verbinden. Dabei werden kurze Latenzen durch mehrere SMT Hardware-Fäden überdeckt und bei langen Latenzen, wie zum Beispiel einem L2-Cache-Miss, wird der auslösende Hardware-Faden ausgelagert und durch einen anderen wartenden Programmfaden ersetzt. Die alternativen Programmfäden liegen in einem sogenannten inaktiven Registerpuffer, auf den nur mit zwei speziellen Lese- und Schreibinstruktionen zugegriffen werden kann.

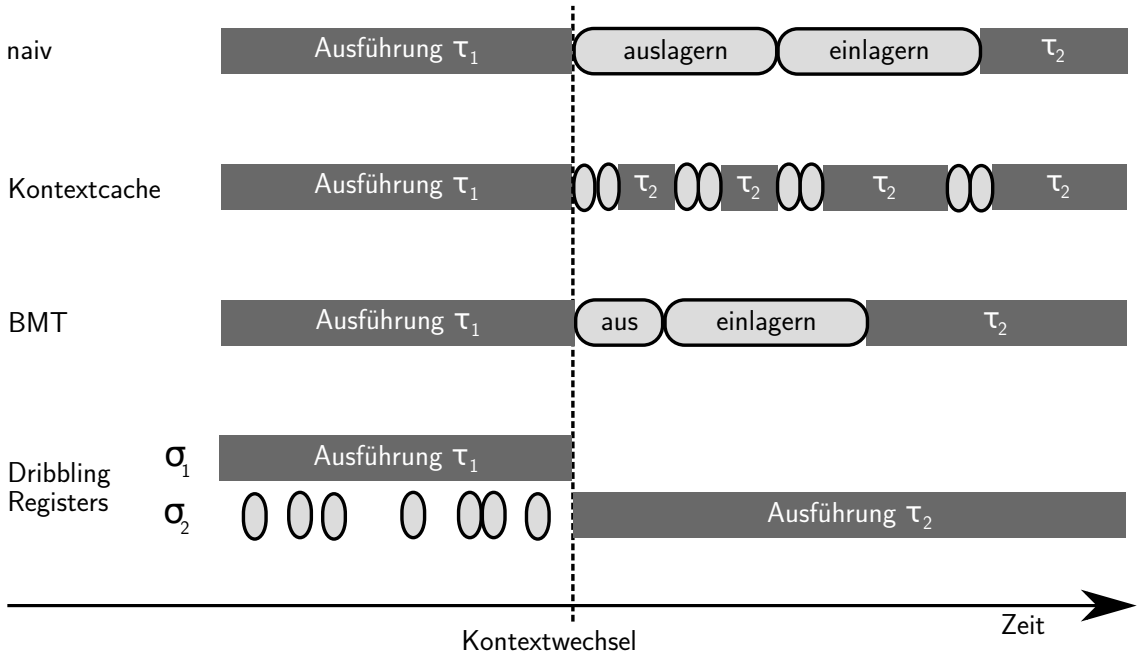

Abb. 4.1: Unterschiedlichen Techniken um Kontextwechsel zu verkürzen

Dadurch muss die eigentliche Prozessorpipeline kaum verändert werden, aber bei einem Kontextwechsel wird für jedes auszulagernde Register ein Schreibbefehl und für jedes einzulagernde Register ein Lesebefehl in der Pipeline ausgeführt. Durch etwas zusätzliche Logik kann die Anzahl erheblich reduziert werden, indem nur die wirklich veränderten Register in den inaktiven Registerpuffer zurückgeschrieben werden.

Dennoch belegt jeder Kontextwechsel für mehrere Takte die Prozessorpipeline und für harte Echtzeitanwendungen muss von der vollen Anzahl auszutauschender Register ausgegangen werden. Außerdem ist die Anzahl inaktiver Programmfäden begrenzt, da sie in einem direkt im Prozessor befindlichen Puffer mit unveränderlicher Größe zwischengespeichert werden und einige spezielle Register (wie Stackzeiger und Prozessorkontrollregister) nicht transferiert, sondern direkt in der Pipeline dupliziert werden.

4.1.1.3 Dribbling Registers

Im Gegensatz zu den beiden bisher vorgestellten Verfahren zum Kontextwechsel wird bei den sogenannten *Dribbling Registers* [Soundararajan 1992] keine Rechenzeit für das Laden oder Sichern der Prozessorregister verschwendet. Stattdessen werden Takte, in denen der Prozessor nicht auf den Speicher zugreift, für das Laden bzw. Sichern der Registerinhalte genutzt. Dadurch kann ein Kontextwechsel zwischen normale Speicherzugriffe geschoben werden, sodass die Programmausführung nicht beeinflusst wird. Einzige Bedingung ist, dass schon deutlich vor dem eigentlichen Kontextwechsel bekannt ist, welcher Programmfaden ausgetauscht werden soll, damit rechtzeitig mit dem Transfer des Kontextes begonnen werden kann.

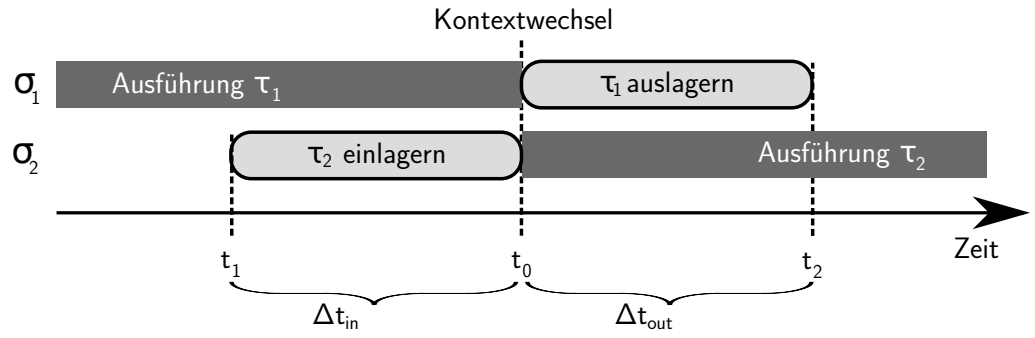

Abb. 4.2: Versteckter Kontextwechsel mit zwei physikalischen Hardware-Fäden σ_1 und σ_2

Das Verfahren ist jedoch nicht hart echtzeitfähig, denn wie viele Speichertakte für Kontextwechsel zur Verfügung stehen hängt davon ab, welche Programmfäden vor dem eigentlichen Kontextwechselzeitpunkt ausgeführt werden. Im schlechtesten Fall muss davon ausgegangen werden, dass alle Speichertakte belegt sind und daher der Kontextwechsel nicht im Hintergrund vorab durchgeführt werden kann. Dann müssen Speichertakte speziell für den Kontextwechsel reserviert werden, was zu einer Blockierung der Prozessor-Pipeline führt. Unter harten Echtzeitgesichtspunkten ist damit auch diese Technik nicht besser als naives Kopieren des Kontextes.

4.1.2 Versteckter Kontextwechsel

Keine der bisher vorgestellten Techniken kann die maximale Dauer eines Kontextwechsels im Vergleich zum naiven Kopieren der Registerinhalte reduzieren. In diesem Sinne ist keiner der Ansätze hart echtzeitfähig. Die Kontextwechselzeit kann aber vollständig versteckt werden, sodass sie auch im echtzeitrelevanten schlechtest denkbaren Fall Null beträgt.

Der Trick besteht darin, einen Kontextwechsel nicht in einem Hardware-Faden durchzuführen, sondern zwischen zwei Hardware-Fäden umzuschalten. Vor dem eigentlichen Kontextwechselzeitpunkt wird in einem deaktivierten Hardware-Faden der nächste Programmfaden schon vorab geladen. Zum eigentlichen Wechselzeitpunkt wird nur der erste Hardware-Fäden stillgelegt und der bisher deaktivierte Hardware-Faden aktiviert. Anschließend wird der Programmfaden im nun deaktivierten Hardware-Faden ausgelagert.

Abbildung 4.2 illustriert den Vorgang: Programmfaden τ_1 wird zunächst in Hardware-Faden σ_1 ausgeführt. Ein Kontextwechsel von Programmfaden τ_1 auf τ_2 beginnt bereits einige Zeit vor dem tatsächlichen Wechselzeitpunkt t_0, genauer gesagt um die Dauer eines Einlagerungsvorgangs Δt_{in} früher, das heißt zum Zeitpunkt

$$t_1 = t_0 - \Delta t_{in} = t_0 - n_{regs} \cdot (L_{mem} + 1) \tag{4.2}$$

Zu diesem Zeitpunkt t_1 wird nicht der eigentlich zu wechselnde Hardware-Faden σ_1 ausgelagert, sondern ein weiterer Hardware-Faden σ_2 wird mit dem Kontext des nächsten

Programmfadens τ_2 geladen. Zum Kontextwechselzeitpunkt t_0 wird σ_1 deaktiviert und stattdessen σ_2 aktiviert. Anschließend wird sofort damit begonnen, den Kontext des Programmfadens τ_1 von σ_1 in den Speicher zu übertragen. Bis Δt_{in} Takte vor dem nächsten Kontextwechsel ist σ_1 unbenutzt, dann beginnt der Kontextwechselvorgang von neuem, aber mit vertauschten Rollen von σ_1 und σ_2.

4.2 Scheduling-Algorithmen

Es stehen drei verschiedene Scheduling-Algorithmen für jeweils unterschiedliche Echtzeitanforderungen zur Verfügung:

Dominant Time Slicing (DTS) garantiert jedem Programmfaden eine festgelegte Ausführungszeit, wodurch harte Echtzeitschranken eingehalten werden können.

Periodic Instruction Quantum (PIQ) kontrolliert die Anzahl der tatsächlich ausgeführten Instruktionen und ermöglicht Programmfäden mit weichen Echtzeitbedingungen.

Round Robin Slack (RSS) verteilt die nach Anwendung der beiden anderen Algorithmen noch verbleibende Rechenzeit gleichmäßig unter beliebig vielen Programmfäden ohne spezielle Echtzeitanforderungen.

4.2.1 Dominant Time Slicing

Durch den Dominant-Time-Slicing-Algorithmus [Mische 2010b] wird jedem harten Echtzeitfaden ein fester Bruchteil der gesamten Rechenzeit zugeteilt. Dazu wird die Ausführungszeit in kurze Runden zerteilt und in jeder bekommen sämtliche DTS-Fäden jeweils den ihnen zustehenden Bruchteil der Runde zugeordnet. Anschaulich werden die DTS-Fäden mit individuell reduzierten Taktfrequenzen ausgeführt, die jeweils genau ausreichen, dass der entsprechende DTS-Faden seine Fristen einhalten kann.

4.2.1.1 Dominanter Metafaden

Die in Kapitel 3 beschriebene Architektur stellt einen ausgezeichneten Hardware-Faden zur Verfügung, der die höchste Priorität hat und von den anderen Hardware-Fäden vollständig isoliert ist. Dadurch kann die Ausführungszeit dieses Hardware-Fadens unabhängig von eventuell gleichzeitig ausgeführten Hardware-Fäden ermittelt werden. Dies ist jedoch nur für einen Hardware-Faden möglich, da bereits für den Hardware-Faden mit der zweithöchsten Priorität keine Garantien mehr gegeben werden können.

Der Grund dafür ist, dass der höchstpriore Hardware-Faden theoretisch alle Prozessorressourcen allein belegen kann. Da er dies möglicherweise für die gesamte Dauer der Periode des sekundären Hardware-Fadens tun kann, erhält der sekundäre Hardware-Faden in

diesem ungünstigsten Fall keine Rechenzeit, eine Mindestrechenzeit kann folglich nicht garantiert werden. (vgl. Abschnitt 2.4.3)

Um dennoch mehrere Programmfäden quasiparallel auszuführen, bleibt nur die Möglichkeit, die Ausführungszeit des höchstprioren Hardware-Fadens aufzuteilen. Zu diesem Zweck wird der sogenannte dominante Metafaden (engl. *dominant meta thread*) verwendet. Der dominante Metafaden ist nicht nur ein Hardware-Faden, sondern besteht aus mehreren Hardware-Fäden, die abwechselnd ausgeführt werden. Jedoch ist zu einem bestimmten Zeitpunkt nur ein Hardware-Faden aktiv, alle anderen sind deaktiviert. Somit ist gewährleistet, dass immer nur ein Hardware-Faden die höchste Priorität hat.

Durch die in Abschnitt 4.1.2 beschriebene Technik kann der dominante Metafaden mit nur zwei Hardware-Fäden realisiert werden. In einem der beiden wird jeweils ein Programmfaden ausgeführt, der andere Hardware-Faden dient zum Ein- beziehungsweise Auslagern des nächsten beziehungsweise letzten Programmfadens.

Dadurch wird die Kontextwechselzeit vollständig versteckt: zu jedem Zeitpunkt wird ein Programmfaden ausgeführt, kein einziger Prozessortakt wird verschwendet. Trotzdem ist die Anzahl der Programmfäden nicht beschränkt, die einzige Bedingung ist, dass zwischen zwei Kontextwechseln genügend Zeit liegt, um einen Programmfaden auszulagern und einen neuen wieder einzulagern. Durch geeignete Wahl des Scheduling-Algorithmus kann diese Bedingung leicht erfüllt werden.

4.2.1.2 Wahl des Scheduling-Algorithmus

Der dominante Metafaden kann als einfädiger Prozessor aufgefasst werden, auf dem klassische Scheduling-Algorithmen (vgl. Abschnitt 2.3) zur Ausführung mehrerer harter Echtzeitprogrammfäden verwendet werden könnten. Da der Prozessor möglichst gut ausgelastet werden soll, kommen nur optimale Scheduling-Algorithmen in Frage, das heißt auf festen Prioritäten basierende Algorithmen scheiden aus.

Der Nachteil von fristengetriebenen Algorithmen wie EDF ist die ziemlich aufwändige Scheduling-Entscheidung, die relativ viel Hardwarelogik verbraucht. Es muss eine nach Fristen sortierte Liste aller Programmfäden vorgehalten werden, in die bei jedem Periodenbeginn eines Programmfadens ein Eintrag eingefügt wird. Die dafür nötige Datenstruktur wird Vorrangwarteschlange (engl. *priority queue*) [Knuth 1998, Kapitel 5.2.3] genannt.

Die Zeitkomplexität einer Einfügeoperation beträgt zwar nur $O(\log N)$, steigt aber dennoch mit steigender Anzahl Programmfäden N. Die obere Schranke für die Einfügedauer hängt folglich von der Anzahl der Programmfäden ab, weshalb bei echtzeitfähigen Schedulern die Anzahl beschränkt werden muss [Kuacharoen 2003]. Erschwerend kommt hinzu, dass eine effiziente Hardwareimplementierung einer Vorrangwarteschlange ebenfalls eine maximale Obergrenze für die Anzahl der Programmfäden erfordert [Moon 2000].

Deshalb fiel die Wahl auf einen Scheduling-Algorithmus, der die Rechenzeit nach Zeitschlitzen aufteilt. Das in Abschnitt 2.3.3 vorgestellte Infinite Time Slicing (ITS) geht

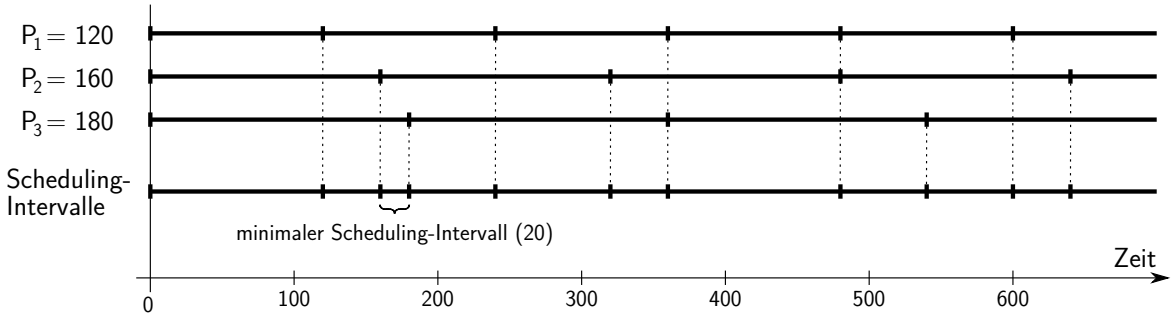

Abb. 4.3: Planungsintervalle bei drei Programmfäden unterschiedlichen Perioden

jedoch von unendlich kleinen Planungsintervallen aus, die es in der Praxis nicht gibt, da der Prozessortakt immer eine untere Schranke darstellt.

Einen Ausweg zeigt Omir Serlin in seiner Veröffentlichung von 1972 [Serlin 1972] auf, den *Minimal Time Slicing (MTS)* Algorithmus. Hier wird das Planungsintervall als Abstand zwischen zwei „Scheduling-Ereignissen“ definiert. Ein Scheduling-Ereignis tritt auf, wenn etwas passiert, das die Ablaufplanung beeinflusst, also der Beginn einer Periode oder das Ablaufen einer Frist. Die Zeit zwischen zwei Scheduling-Ereignissen wird entsprechend den Lastfaktoren L_i der einzelnen Programmfäden verteilt.

Eine Schwierigkeit bei der Implementierung des MTS Algorithmus ist die stark schwankende Länge der Planungsintervalle (siehe Abbildung 4.3). Die Dauer eines Planungsintervalls ist bei gegebenen Perioden der Programmfäden prinzipiell berechenbar, jedoch hängt die Dauer der Berechnung von der Anzahl der Programmfäden ab. Damit ist keine obere Grenze gegeben und das Verfahren nicht echtzeitfähig.

Alternativ wäre eine Vorabberechnung denkbar, die dafür nötige Liste wäre zwar sehr groß (sie muss den Zeitraum des kleinsten gemeinsamen Vielfachen der Perioden abdecken), aber prinzipiell im mehr oder weniger unbegrenzt zur Verfügung stehenden Hauptspeicher ablegbar. Da pro Planungsintervall nur jeweils der nächste Eintrag aus der Liste entnommen wird und am Ende der Liste einfach wieder von vorn begonnen wird, wäre dies auch echtzeitfähig.

Es geht aber auch sehr viel einfacher, indem die Planungsintervalle noch weiter unterteilt werden, bis man ein festes, minimales Zeitintervall, die sogenannte *Runde* erhält. Diese Runde muss ein ganzzahliger Teiler aller möglichen Planungsintervalle sein. Erhält jeder Programmfaden τ_i in jeder Runde einen Zeitschlitz, der seiner Last $L_i = \frac{C_i}{T_i}$ entspricht, so kann jeder Programmfaden seine Zeitschranken einhalten. Dieses Verfahren auf den dominanten Metafaden eines echtzeitfähigen mehrfädigen Prozessors angewendet, ergibt den *Dominant Time Slicing (DTS)* Algorithmus.

Die kleinste Zeiteinheit in einem Prozessor ist der Prozessortakt, deshalb werden alle Zeiten als Vielfaches dieser Einheit ausgedrückt. Die Anzahl Takte, die dem Zeitschlitz eines Programmfadens entspricht, wird deshalb *Taktquantum* Q_i genannt. Es ergibt sich

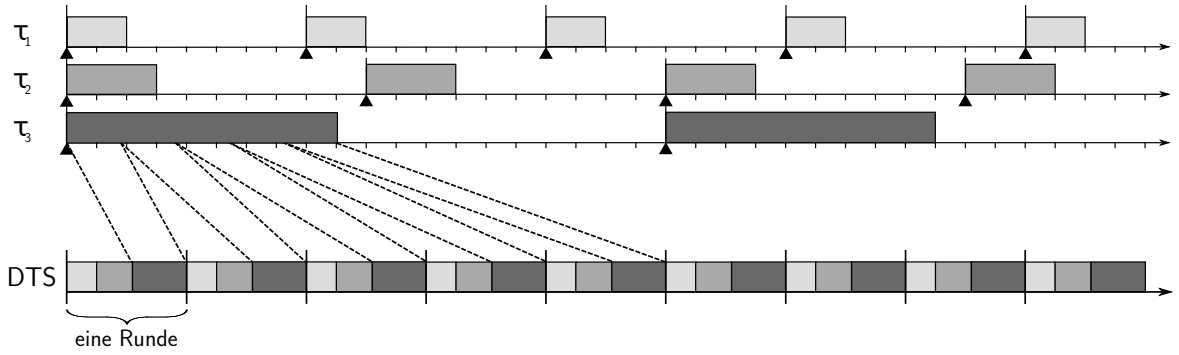

Abb. 4.4: Weitung von Programmfäden beim DTS-Scheduling

aus folgender Formel:

$$Q_i = R \cdot L_i = R \cdot \frac{C_i}{T_i} \tag{4.3}$$

Anders ausgedrückt, wird jeder Programmfaden der Reihe nach für eine bestimmte Anzahl Takte, die seinem Quantum entspricht ausgeführt. Waren alle Programmfäden einmal an der Reihe, wird wieder beim ersten begonnen. So formuliert, entspricht der Algorithmus dem *Weighted Round Robin (WRR)* Algorithmus, der bei der echtzeitfähigen Lastverteilung in Netwerken eingesetzt wird [Liu 2000, S. 61f]. Einziger Unterschied zu WRR ist, dass zwischen dem letzten Programmfaden einer Runde und dem ersten der nächsten Runde noch eine Pause fester Länge liegen kann, in der kein Programmfaden ausgeführt wird. Der *Guaranteed Percentage (GP)* Algorithmus [Kreuzinger 2000] ist ein Spezialfall des DTS-Algorithmus, bei dem die Rundenlänge auf 100 Takte festgelegt ist.

Eine sehr treffende Bezeichnung für dieses Verfahren wurde von Ali Ahmad El-Haj-Mahmoud [El-Haj-Mahmoud 2005] geprägt, mit seinen Worten wird die Ausführung der einzelnen Programmfäden „geweitet“(engl. *dilated*) und gleichmäßig über die gesamte Rechenzeit verteilt. Abbildung 4.4 veranschaulicht das Verfahren grafisch.

Ein Problem von Zeitschlitz-basierenden Algorithmen ist ihre mangelnde Flexibilität. Sie sind zwar perfekt für die Ausführung von periodischen Programmfäden geeignet, bei sporadischen Programmfäden wird jedoch viel zu viel Rechenzeit reserviert, da die maximale Ausführungsrate zwischen dem Beginn eines Jobs bis zur Frist garantiert werden muss (Abbildung 4.5). Als Last ergibt sich deshalb

$$L_i = \frac{C_i}{D_i} \tag{4.4}$$

Diese Last wird nicht nur bis zur Frist reserviert, sondern auch in der Zeit, in der der Programmfaden garantiert nicht aktiviert wird (zwischen Frist und Periodenende) und in der variablen Zeit zwischen Periodenende und dem nächsten Job. In der hier beschriebene Architektur stellt das jedoch kein Problem dar, da die durch einen sporadischen Programmfaden ungenutzte Zeit vollständig von Programmfäden mit niedrigeren Echtzeitanforderungen (PIQ, RSS) genutzt werden kann.

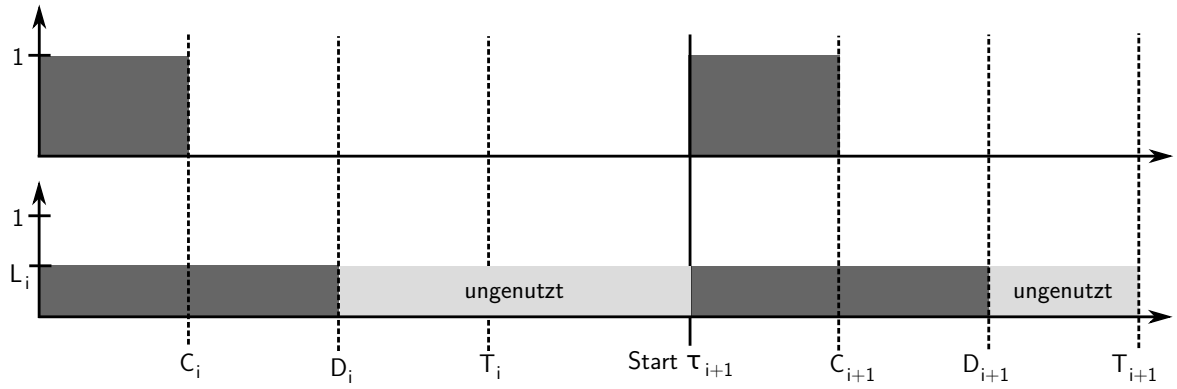

Abb. 4.5: Last L_i die für einen sporadischen Job mit Frist D_i und Periode T_i bei einer Ablaufplanung mit Zeitschlitzen reserviert werden muss

4.2.1.3 Berechnung der Rundenlänge

Wie oben erklärt, entspricht die Rundenlänge dem größten gemeinsame Teiler (ggT) aller möglichen Planungsintervalle. Es ist jedoch nicht nötig, zuerst alle möglichen Planungsintervalllängen aus den Periodenlängen zu berechnen und dann den ggT zu ermitteln, vielmehr erfüllt bereits der ggT der Periodenlängen die Bedingung, dass jedes Scheduling-Intervall ein ganzzahliges Vielfaches dieser Zahl ist.

Denn falls die Rundenlänge R gegeben ist als

$$R = \mathrm{ggT}(P_1, \ldots P_N) \tag{4.5}$$

so ist jede Periode P_i das ganzzahlige Vielfache der Runde R. Damit liegt jeder Periodenbeginn auf einem Vielfachen von R. Da jedes Planungsintervall zwischen dem Beginn von zwei Perioden liegen muss, muss auch jedes Planungsintervall ein ganzzahliges Vielfaches von R sein. Abbildung 4.3 zeigt einen Versuch, diesen Zusammenhang bildlich zu veranschaulichen.

Zu numerischen Problemen kann es kommen, wenn die Rundenlänge extrem klein wird. So kann es leicht vorkommen, dass das Taktquantum $Q_i = \frac{C_i}{T_i} \cdot R$ nicht ganzzahlig ist. In diesem Fall muss aufgerundet werden um sicherzustellen, dass jeder Programmfaden genügend Rechenzeit erhält. Dadurch erhöht sich aber die Gesamtauslastung U, und zwar um maximal einen Takt pro Programmfaden, also höchstens N Takte pro Runde. Daraus ergibt sich eine maximale Gesamtauslastung U' bei der gefahrlos aufgerundet werden kann:

$$U' = 1 - \frac{N}{R} \tag{4.6}$$

Folglich kann, wenn die Gesamtauslastung U ausreichend kleiner als 1 ist (was bei großen Rundenlängen relativ leicht realisierbar ist) das Aufrunden vernachlässigt werden. Bei kleinen Rundenlängen kann es jedoch passieren, dass durch das Aufrunden die Gesamtauslastung so weit erhöht wird, dass sie 1 übersteigt und damit die Einplanbarkeit nicht mehr gegeben ist.

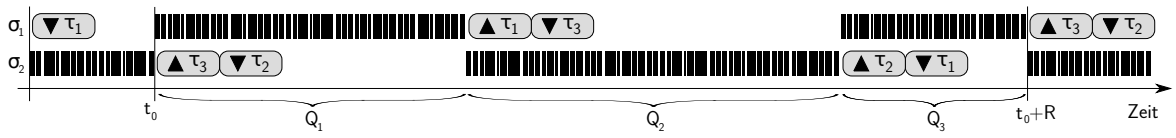

Abb. 4.6: Drei hart echtzeitfähige Programmfäden τ_1, τ_2 und τ_3, die per DTS-Algorithmus in zwei Hardware-Fäden σ_1 und σ_2 ausgeführt werden. Die weißen Zwischenräume repräsentieren freie Ressourcen, die durch weitere Hardware-Fäden genutzt werden könnten (▼ einlagern, ▲ auslagern).

Ein kleines Beispiel soll zeigen, dass der Aufrundungseffekt in realen Systemen nahezu vernachlässigt werden kann: üblicherweise sind die Perioden eines typischen Tasksets in Zehntelmillisekunden angegeben, daraus ergibt sich eine Rundenlänge von $0,1$ ms. Dies entspricht auf einem 100-MHz-Prozessor 10000 Takten und bei 100 Programmfäden einer maximalen Gesamtauslastung von 99%.

4.2.1.4 Technische Realisierung

Abbildung 4.6 illustriert die konkrete technische Realisierung des DTS-Algorithmus mit zwei Hardware-Fäden σ_1 und σ_2 und drei logischen Programmfäden τ_1, τ_2 und τ_3. Zu Beginn der Runde (Zeitpunkt t_0) wurde der DTS-Faden τ_1 bereits in einen Hardware-Faden (hier σ_1) geladen. Deshalb kann seine Ausführung mit Beginn der Runde ebenfalls starten. Parallel zur Ausführung wird der bisher im anderen Hardware-Faden ausgeführte Programmfaden τ_3 ausgelagert, das heißt, sein Kontext wird in den Speicher geschrieben. Anschließend wird der Kontext des zweiten DTS-Fadens τ_2 aus dem Speicher gelesen und in Hardware-Faden σ_2 gespeichert.

Die Dauer dieses Kontextwechsels muss kürzer sein als das Quantum, Q_1, das dem parallel dazu ausgeführten DTS-Faden τ_1 zugeordnet ist. Dadurch ist sichergestellt, dass nach Q_1 Takten (zum Zeitpunkt $t_0 + Q_1$), wenn Hardware-Faden σ_1 und damit DTS-Faden τ_1 deaktiviert wird, der andere Hardware-Faden σ_2 aktiviert werden kann, da dort der Kontext von τ_2 bereits geladen wurde und nur auf das Signal zum Start der Ausführung wartet.

Nach Ablauf des ersten Quantums Q_1, also zum Zeitpunkt $t_0 + Q_1$ läuft demnach der gleiche Vorgang ab wie zu Beginn der Runde, nur mit anderen Hardware- und DTS-Fäden: σ_1 und mit ihm τ_1 wird deaktiviert, σ_2 mit DTS-Faden τ_2 wird dafür aktiviert und anschließend wird der Kontext von σ_1 durch den Kontext von σ_3 ersetzt.

Der gleiche Vorgang wiederholt sich jedes mal, wenn ein Quantum abgelaufen und deshalb ein Kontextwechsel nötig ist. Zu beachten ist, dass, wenn die Anzahl der DTS-Fäden ungerade ist, die logischen Programmfäden in jeder Runde abwechselnd in einen anderen Hardware-Faden geladen werden, bei gerader Anzahl hingegen werden sie jeweils dem gleiche Hardware-Faden zugeordnet.

Falls die DTS-Fäden nicht die gesamte Rechenzeit beanspruchen, mit anderen Wor-

ten, falls die Summe der DTS-Quanten niedriger als die Rundenlänge ist, so wird nach Verstreichen des letzten Quantums zwar der bisher aktive Hardwarefaden deaktiviert, die Aktivierung des anderen Hardware-Fadens erfolgt aber erst zu Beginn der nächsten Runde. Ebenso beginnt das Auslagern des Programmfadens erst beim nächsten Rundenbeginn, das heißt, zwischen dem Ende des letzten Quantums und dem Rundenende stehen beide für das DTS-Scheduling reservierten Hardware-Fäden einfach still.

4.2.2 Periodic Instruction Quantum

Mit Hilfe des Periodic-Instruction-Quantum-Algorithmus [Mische 2009] können die Ausführungszeiten mehrerer Programmfäden mit weichen Echtzeitanforderungen überlappt werden. Ermöglicht wird dies dadurch, dass Instruktionen statt Takte gezählt werden, um den Rechenzeitanteil eines Programmfadens zu bestimmen.

4.2.2.1 Motivation

Der DTS-Algorithmus erlaubt mehrere Programmfäden mit *harten* Echtzeitanforderungen, doch dadurch werden die Möglichkeiten eines SMT-Prozessors nur zu einem kleinen Teil genutzt. Der einzige Vorteil gegenüber einer einfädigen Architektur besteht in der Möglichkeit, Programmfäden ohne Echtzeitanforderungen parallel zu den hart echtzeitfähigen Programmfäden auszuführen, diese selbst profitieren aber kaum von den Vorzügen der SMT-Architektur.

Würde man den DTS-Algorithmus auf einem einfädigen Superskalarprozessor implementieren, würde die gleiche Ausführungsgeschwindigkeit wie bei einem SMT-Prozessor erreicht werden, sieht man von den Kontextwechselzeiten ab. Aber diese Kontextwechselzeiten sind konstant und klein, verglichen mit der Länge einer Runde. Der zusätzliche Zeitbedarf für die Kontextwechsel könnte durch eine geringfügige Erhöhung der Taktfrequenz ausgeglichen werden. Eine solche Frequenzerhöhung ist durchaus realistisch, da ein einfädiger Prozessor einfacher aufgebaut ist als ein SMT-Prozessor und dadurch auch höher getaktet werden kann.

Die parallele Ausführung von Programmfäden durch gemeinsame Nutzung von Prozessorressourcen – der Hauptvorteil eines mehrfädigen Prozessors – ist bei mehr als einem harten Echtzeit-Programmfäden nicht möglich, nur ein harter Echtzeitfaden kann gleichzeitig mit mehreren Programmfäden ohne Echtzeitanforderungen ausgeführt werden. Eine überlappende Ausführung von Echtzeitfäden ist nicht möglich, da der schlimmste Fall betrachtet werden muss und die schlechteste denkbare Konstellation ist gegeben, wenn ein Programmfaden alle verfügbaren Prozessorressourcen belegt und für die Nutzung durch andere Programmfäden keine weiteren übrig lässt.

Wenn alle harten Echtzeitfäden diese hohen Ressourcen-Anforderungen stellen, kann nur jeweils ein Programmfaden zur selben Zeit ausgeführt werden, die Verteilung der Ausführungszeit in Form von Zeitschlitzen bleibt somit die einzig praktikable Möglichkeit, um Programmfäden mit harten Echtzeitanforderungen quasi gleichzeitig auszuführen. Aus

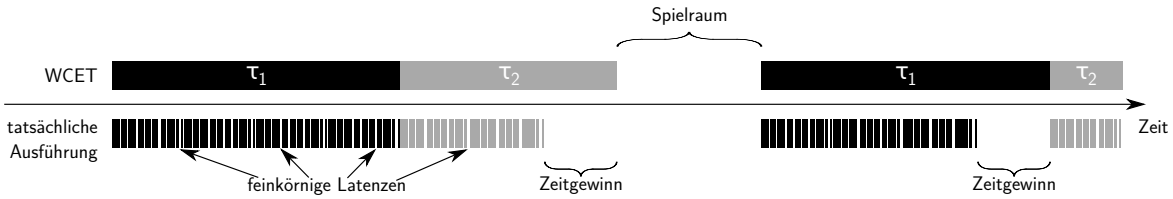

Abb. 4.7: Verschiedene Arten von ungenutzter Ausführungszeit

diesem Grund werden im folgenden Abschnitt Programmfäden mit *weichen* Echtzeitanforderungen behandelt, denn diese sind auch bei überlappender Ausführung erfüllbar.

Durch die zusätzliche Ausführung von Programmfäden mit weichen Echtzeitanforderungen können drei unterschiedliche Arten von andernfalls verschwendeter Ausführungszeit genutzt werden (siehe auch Abbildung 4.7):

Feinkörnige Latenzen (engl. *fine-grained latencies*) Prozessorressourcen, die für einzelne Takte zur Verfügung stehen

Zeitgewinn (engl. *gain time*) Die Zeit, die übrig bleibt, wenn ein harter Echtzeitfaden nicht die volle WCET zur Ausführung benötigt, sondern schon früher fertig wird.

Spielraum (engl. *slack time*) Tritt auf, wenn der Prozessor überdimensioniert ist, das heißt, selbst im schlimmsten Fall, wenn alle Echtzeitfäden die volle WCET benötigen, ist der Prozessor nicht völlig ausgelastet, sondern es bleibt Zeit übrig, da $\sum L_i < 1$.

4.2.2.2 Parallele Ausführung ohne Kontextwechsel

Um die Darstellung des PIQ-Algorithmus zu vereinfachen, wird zunächst angenommen, dass alle Programmfäden entsprechend dem PIQ-Algorithmus geschedult werden und dass genügend Hardware-Fäden für alle Programmfäden vorhanden sind, das heißt, auf Kontextwechsel in den Hauptspeicher kann verzichtet werden.

Das Prinzip, die Gesamtrechenzeit in Runden einzuteilen und Bruchteile dieser Runden an jeden Echtzeitfaden zu verteilen, bleibt auch beim PIQ-Algorithmus erhalten. Aber anstatt der disjunkten Hintereinanderausführung wie bei DTS, werden bei PIQ alle Programmfäden zu Beginn der Runde gleichzeitig aktiviert.

Da jeder Hardware-Faden eine andere Priorität haben muss, gibt es einen PIQ-Faden, der die höchste Priorität hat und daher annähernd so schnell ausgeführt wird, als wäre er der einzige Programmfaden. Nur wenn dieser Hardware-Faden nicht alle Prozessorressourcen vollständig nutzen kann, darf der Hardware-Faden mit der nächsthöheren Priorität sie belegen, dann der nächste und so weiter. Dadurch machen die anderen Hardware-Fäden Fortschritte, obwohl der Hardware-Faden mit der höchsten Priorität mit maximaler Geschwindigkeit ausgeführt wird.

Sobald dieser Hardware-Faden so lange ausgeführt wurde, wie es seinen Bruchteil an der

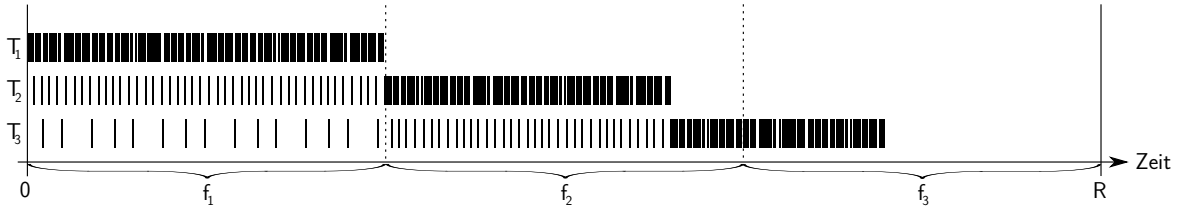

Abb. 4.8: Drei PIQ-Fäden τ_1, τ_2, τ_3, die für jeweils ein Drittel der Runde ausgeführt werden sollen ($C_1 = C_2 = C_3 = \frac{R}{3}$), werden innerhalb einer Runde überlappend ausgeführt.

Runde entspricht, wird er deaktiviert und die Prioritäten der anderen Hardware-Fäden „rutschen nach“, das heißt, ihre Prioritäten werden jeweils um eine Stufe erhöht, da die bisher höchste Stufe nicht mehr benötigt wird. Dadurch läuft ein anderer Hardware-Faden nun mit maximaler Geschwindigkeit, bis sein Bruchteil erreicht ist. Da er trotz der bevorzugten Ausführung des ersten Hardware-Fadens während dieser Zeit bereits einige Befehle ausführen konnte, kann er schneller seinen Rundenbruchteil erreichen, als es ohne überlappende Ausführung möglich wäre. Diese Einsparungen summieren sich, sodass der letzte Hardware-Faden einige Zeit vor dem tatsächlichen Ende der Runde fertig werden kann (siehe Abbildung 4.8).

Wenn die PIQ-Fäden regelmäßig vor dem eigentlichen Ende der Runde fertig werden, kann die verbleibende Rechenzeit entweder für zusätzliche Programmfäden genutzt werden, oder man reduziert die Taktfrequenz, um Energie zu sparen.

4.2.2.3 Reduzierte Anzahl Hardware-Fäden

Auch der PIQ-Algorithmus kann mit weniger Hardware-Fäden als Programmfäden realisiert werden. Dies ist insbesondere deshalb sinnvoll, weil bei mehr als vier oder fünf Hardware-Fäden, der Hardware-Faden mit der niedrigsten Priorität nur noch sehr sporadisch ausgeführt wird (vgl. Evaluierung Abschnitt 6.3.1). Außerdem kann er zusätzlich zum DTS-Scheduler eingesetzt werden.

Wenn weniger Hardware-Fäden als Programmfäden zur Verfügung stehen, so werden zu Beginn einer Runde nacheinander alle verfügbaren Hardware-Fäden durch Kontextwechsel mit PIQ-Programmfäden gefüllt. Verfügbar sind alle Hardware-Fäden, außer den beiden für den dominanten Metafaden reservierten. Sind keine DTS-Programmfäden vorhanden, weil keine harten Echtzeitschranken erforderlich sind, so gibt es natürlich auch keinen dominanten Metafaden nd dessen beiden Hardware-Fäden stehen für das PIQ-Scheduling zur Verfügung. Programmfäden mit noch geringeren Echtzeitanforderungen als die PIQ-Fäden (siehe Abschnitt 4.2.3) werden hingegen verdrängt, das heißt, ihr Kontext wird in den Speicher verschoben und durch den Kontext eines PIQ-Fadens ersetzt (Abbildung 4.9).

Sobald der erste PIQ-Faden seinen Rechenzeitanteil erreicht hat und deaktiviert wird, wird er in den Speicher ausgelagert und per Kontextwechsel durch einen weiteren PIQ-Faden ersetzt. Wurden in einer Runde bereits alle PIQ-Fäden einmal eingelagert und

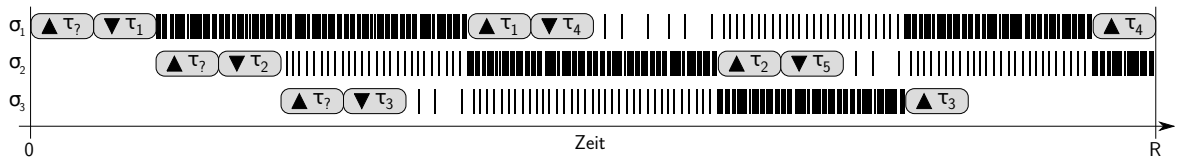

Abb. 4.9: Fünf Programmfäden τ_1 bis $\ldots\tau_5$ werden per PIQ-Algorithmus in drei Hardware-Fäden σ_1, σ_2 und σ_3 ausgeführt. Um die Darstellung zu vereinfachen, wurden keine parallel laufenden DTS-Fäden angenommen. Dichtere Linien bedeuten weniger Latenzen, mehr ausgeführte Instruktionen, eine höhere IPC (▼ einlagern, ▲ auslagern).

ausgeführt, so können die bereits erwähnten Programmfäden ohne Echtzeitanforderungen eingelagert und ausgeführt werden.

4.2.2.4 Messung des Rundenanteils

Bisher wurde nicht darauf eingegangen wie festgestellt wird, ob ein Programmfaden seinen Anteil an der aktuellen Runde erreicht hat. Werden disjunkte Zeitschlitze mit nur jeweils einem gleichzeitig ausgeführten Programmfaden verwendet, wie beim DTS-Algorithmus, so kann die Ausführungszeit durch einfaches Zählen der Takte ermittelt werden. Diese Ausführungszeit geteilt durch die Länge einer Runde ergibt den Anteil eines Programmfadens. Ist der garantierte Bruchteil erreicht, wird der Programmfaden ausgelagert.

Bei überlappender Ausführung ist es jedoch nicht trivial, festzustellen, wie lange ein einzelner Programmfaden bereits ausgeführt wurde, da die Programmfäden sich gegenseitig beeinflussen. Hier erhält jeder Programmfaden eine eigene virtuelle Uhr. In jedem Takt muss entschieden werden, welche Uhren weiterlaufen und welche gestoppt werden. Die Ausführungszeit pro Runde basiert auf der WCET, die ermittelt wird, wenn der Programmfaden alleine, ohne konkurrierende Programmfäden ausgeführt wird. Deshalb muss die virtuelle Uhr die Zeit auf die gleiche Weise verstreichen lassen, wie es bei einfädiger Ausführung des Programmfadens geschähe.

Die Schwierigkeit dieser Entscheidung wird im Folgenden durch einige kurze Beispiele einer Programmausführung dargestellt. In Abbildung 4.10 wird die Programmausführung auf einem zweifach superskalaren SMT-Prozessor dargestellt. Je Block werden horizontal zwei Ressourcen (Ausführungseinheiten) und vertikal vier aufeinanderfolgende Takte der Programmausführung dargestellt. Das Ergebnis der einfädigen Ausführung von Programmfaden τ_1 ist im ganz links stehenden Block dargestellt.

Da er die höchste Priorität hat, belegt er immer die gleichen Ressourcen, sein Schema ändert sich nie. In der oberen Reihe sind die einfädigen Ausführungsschemata von anderen Programmfäden (τ_2 bis τ_5) dargestellt. In der unteren Reihe ist das Ergebnis der gemeinsamen Ausführung der des jeweiligen Programmfadens mit τ_1 dargestellt. Die Zahlen rechts neben den Blöcken geben den jeweiligen Wert der virtuellen Uhr an. Die Ausführung läuft von oben nach unten.

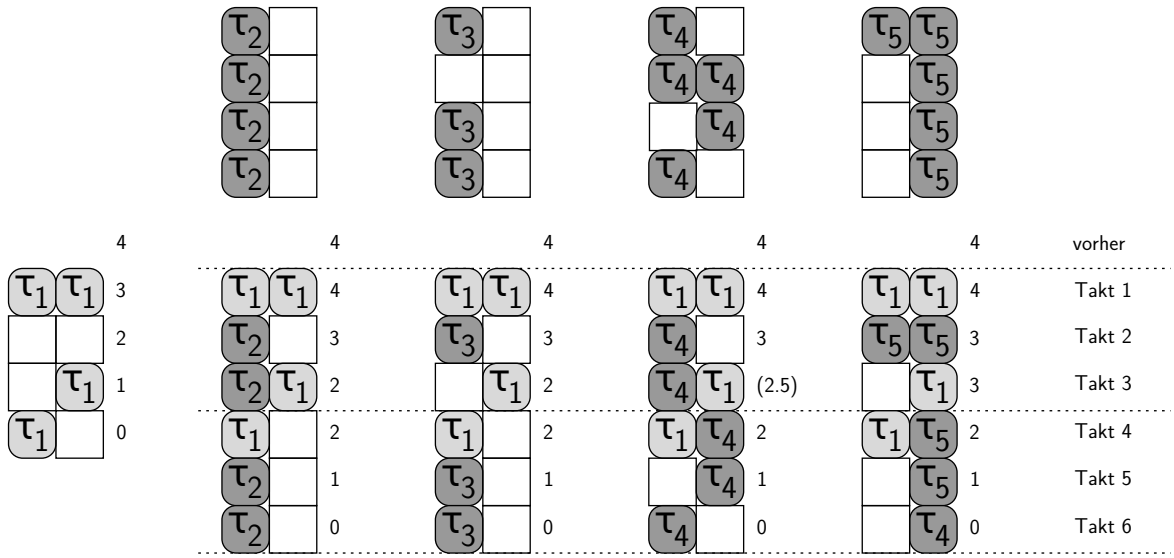

Abb. 4.10: Die Ausführung des dominanten Programmfadens τ_1 wird von weiteren Programmfäden τ_2 bis τ_5 überlappt.

Da die absolute Ausführungszeit für den Scheduler nicht von Bedeutung ist und ein Takt die kleinste Zeiteinheit innerhalb eines Prozessors ist, wird die Zeit in Takten gemessen. Zu Beginn einer Runde wird der Zähler auf das Quantum des entsprechenden Programmfadens gesetzt und wenn die virtuelle Zeit vergeht, wird dieser Zähler erniedrigt, bis er null erreicht und damit der Programmfaden die ihm zugestandene virtuelle Zeit erhalten hat.

Für Programmfaden τ_1 mit der höchsten Priorität ist das Herunterzählen der virtuellen Zeit nicht schwer, da er unabhängig von den anderen Programmfäden abläuft, als wären diese nicht vorhanden. Seine Ausführungszeit entspricht genau der einfädigen Ausführung im ganz linken Block, seine virtuelle Uhr entspricht genau der wirklichen Uhr. Folglich wird sein virtueller Uhrzähler, der zu Beginn der Runde auf vier steht, jeden Takt um eins erniedrigt und erreicht nach vier Takten null.

Für die gleichzeitig ausgeführten Programmfäden mit niedrigerer Priorität ist das Herunterzählen der virtuellen Uhr weitaus komplizierter. Aufgrund der niedrigeren Priorität wird deren Ausführung in etlichen Takten verzögert (zum Beispiel Takte 1 und 4 von τ_2 in Abbildung 4.10). In diesen Takten darf der Zähler von Programmfaden τ_2 nicht verringert werden, die virtuelle Uhr wird angehalten. Eine derartiger Situation tritt auf, wenn der Programmfaden mit der höheren Priorität eine Ressource belegt, die der Programmfaden mit der niedrigeren Priorität im gleichen Takt nutzen könnte. Wenn der Programmfaden mit niedrigerer Priorität mit den von anderen Programmfäden belegten Ressourcen nichts anfangen kann, das heißt anderweitig verzögert wird (zum Beispiel τ_3 in Takt 3), so muss sein Zähler sehr wohl dekrementiert werden, unabhängig davon, ob andere Programmfäden den Takt nützen können oder nicht.

Die bisher beschriebene Berechnung der virtuellen Uhr wäre noch mit vertretbarem

Aufwand realisierbar, der Zähler wird dekrementiert, wenn eine Instruktion ausgeführt wird oder wenn der Programmfaden eine Verzögerung erfährt, die auch im einfädigen Fall aufgetreten wäre. Wird er jedoch aufgrund der Belegung einer Ressource durch einen anderen Programmfaden verzögert, so wird die virtuelle Uhr angehalten. Jedoch reicht diese einfache Verfahren nur aus, wenn nicht gleichzeitig Instruktionen von mehreren Programmfäden ausgeführt werden. Ein derartiger skalar mehrfädiger Prozessoren ist beispielsweise der Komodo-Prozessor, bei dem diese einfache Methode implementiert wurde (siehe Abschnitt 7.2.1).

Bei SMT-Prozessoren kommt ein weiteres Problem hinzu, wenn die Ausführung aufgespalten wird. In Takt 3 belegt τ_1 nur eine Ressource, die andere kann von einem anderen Programmfaden genutzt werden. Programmfaden τ_4 könnte aber beide Ressourcen belegen, würde er allein ausgeführt. Deshalb darf der Zähler der virtuellen Uhr erst dann um einen Takt verringert werden, wenn der gesamte virtuelle Takt (also zwei Instruktionen) verstrichen ist. Bei einer anderen Konstellation (τ_5 in Takt 2) ist es aber durchaus auch möglich, dass beide Ressourcen in einem Takt belegt werden können.

Wie diese Bespiele zeigen, ist die Zeitmessung bei überlappender Ausführung sehr kompliziert und nur möglich, wenn die tatsächliche Zuordnung mit der potentiellen Zuordnung bei alleiniger Ausführung detailliert verglichen wird. Bereits die tatsächliche Zuordnung von mehreren Programmfäden erfordert sehr viel Hardware und zusätzliche Hardware, die für jeden Hardware-Faden die potentielle Zuordnung berechnet und vergleicht, würde den Hardwarebedarf unnötig aufblähen, stattdessen muss eine Alternative gesucht werden.

4.2.2.5 Instruktionszählung

Es bietet sich an, statt einer komplizierten Kombination aus Instruktionen und Latenztakten einfach nur die tatsächlich ausgeführten Instruktionen ohne Latenztakte zu zählen. Dazu erhält jeder Hardware-Faden einen Instruktionszähler, der zu Beginn der Runde, bzw. beim erstmaligen Einlagern eines PIQ-Fadens auf das *Instruktionsquantum* gesetzt wird. Dieser Wert gibt an, wie viele Instruktionen des PIQ-Fadens ausgeführt werden sollen, bevor er wieder ausgelagert wird.

Die erste Stufe des Schedulers, das heißt die Zuordnungsstufe in der Pipeline meldet in jedem Takt, wieviele Instruktionen von welchem Hardware-Faden ausgeführt im jeweiligen Takt zugeordnet wurden. Dementsprechend erniedrigt die zweite Stufe des Schedulers die jeweiligen Instruktionszähler und veranlasst gegebenenfalls einen Kontextwechsel.

Die Ausführungszeit eines Programmteils mit einer festen Anzahl Instruktionen hängt von zwei Faktoren ab:

(i) Der Art der Instruktionen, einige können schnell hintereinander oder gar gleichzeitig ausgeführt werden, andere benötigen zusätzlich noch diverse Latenztakte.

(ii) Den gleichzeitig ausgeführten Hardware-Fäden mit höherer Priorität, insbesondere dem dominanten Meta-Faden mit harten Echtzeitanforderungen, da diese einen

Großteil der Prozessorressourcen belegen und nur die verbleibenden können genutzt werden.

Beide Faktoren sind nur schwer abzuschätzen, da sie ein hochgradig statistisches Verhalten zeigen. Aus diesem Grund schwankt die tatsächliche Ausführungszeit einer festen Anzahl von Instruktionen sehr stark und es ist durchaus möglich, dass das gewünschte Instruktionsquantum innerhalb einer Runde nicht befriedigt werden kann. Da glücklicherweise nur weiche Echtzeitanforderungen erfüllt werden müssen, stellt dies kein verheerendes Problem dar, das verbleibende Quantum wird einfach zum festen Quantum der nächsten Runde addiert.

Über mehrere Runden können damit Schwankungen ausgeglichen werden. Sollte dies nicht der Fall sein, das heißt, die Anzahl der ausgeführten Instruktionen eines Programmfadens bleibt ständig hinter dem vorgesehenen Instruktionsquantum zurück und das von Runde zu Runde übertragene zusätzliche Quantum schwillt unaufhörlich an, so wurde das Instruktionsquantum zu groß gewählt und muss reduziert werden.

Dieser Umstand deutet bereits darauf hin, wie die Ermittlung eines Instruktionsquantums zu erfolgen hat. Durch analytische Methoden kann ein maximales Instruktionsquantum pro Runde kaum ermittelt werden, da zu viele unsichere Faktoren Eingang finden. Zwar bietet die durchschnittliche IPC des einfädig ausgeführten Programmfadens einen gewissen Anhaltspunkt, um das Verhältnis der Laufzeiten von verschiedenen Programmfäden abzuschätzen, jedoch kann der Faktor, mit dem diese Zeit multipliziert werden muss, um den Einfluss von parallel ausgeführten Programmfäden abzuschätzen, nur empirisch ermittelt werden.

Ein solcher Versuch-und-Irrtum-Ansatz hat Vor- und Nachteile. Der Nachteil besteht darin, dass man sich nie sicher sein kann, dass die Quanten nicht zu hoch gewählt wurden und zu einer Überlastung des Systems führen können, der Vorteil liegt darin, dass im Normalfall eine sehr viel höhere Auslastung des Systems erreicht werden kann (vgl. Evaluierung 6.4.2).

4.2.2.6 Weiterführende Nutzung des PIQ-Algorithmus

Der PIQ-Algorithmus kann nicht nur dafür verwendet werden, um durch periodische Programmfäden mit weiche Echtzeitanforderungen die Ressourcen eines Prozessors möglichst effizient zu nutzen, er bietet auch eine einfache Lösung für zwei Probleme, mit denen sich die Echtzeitforschung seit vielen Jahren beschäftigt und teilweise nur sehr komplexe Lösungen anbietet.

Zum Einen kann durch das Verhältnis von Instruktionsquantum und Rundenlänge erreicht werden, dass ein Programmfaden mit einer definierten IPC ausgeführt wird. Da beim Erreichen der gewünschten Instruktionszahl die Ausführung eines Programmfadens sofort abgebrochen wird, wird die IPC sehr genau getroffen, wohingegen andere Arbeiten (siehe Abschnitt 7.3 nur indirekte Verfahren benutzen, um die IPC zu regeln und dadurch gewisse Schwankungen unvermeidbar sind.

Das andere Gebiet betrifft die Unterstützung von Interrupts. In harten Echtzeitsystemen stellen aperiodischen Server (siehe Abschnitt 2.3.4) die von harten Echtzeitfäden ungenutzte Rechenkapazität für die Verarbeitung von Interrupts zur Verfügung. Im hier vorgestellten Scheduling-System wird bei einem aperiodischen Interrupt einfach ein zusätzlicher PIQ-Programmfaden erzeugt, der so schnell wie möglich abgearbeitet wird, ohne die DTS-Fäden mit ihren harten Echtzeitanforderungen zu beeinflussen.

Auch harte aperiodische Anfragen können verarbeitet werden, wenn die Summe der Taktquanten der DTS-Fäden kleiner ist als die Rundenlänge. Dieser sogenannte *Spielraum* Δt_{slack}, der jede Runde zur Verfügung steht, wird auch von anderen aperiodischen Servern zur Ausführung von harten aperiodischen Programmfäden genutzt, jedoch sind dazu teilweise sehr umfangreiche Berechnungen nötig. Bei PIQ dagegen müssen nur die Taktquanten von der Rundelänge abgezogen werden:

$$\Delta t_{slack} = R - \sum_{i=1}^{N} Q_i \tag{4.7}$$

Ist die Ausführungszeit C_{ap} des harten aperiodischen Programmfadens bekannt, so lässt sich die maximale Dauer der Ausführung berechnen:

$$\left\lceil \frac{C_{ap}}{\Delta t_{slack}} \right\rceil \cdot R \quad \stackrel{!}{<} \quad D_{ap} \tag{4.8}$$

Sie muss vor der relativen Frist D_{ap} liegen, damit die Anfrage akzeptiert werden kann. Dies ist zwar eine sehr pessimistische Abschätzung, da die zusätzliche Ausführung aufgrund der gemeinsamen Nutzung von Ressourcen in SMT-Prozessoren nicht berücksichtigt wird, sie ist aber vergleichbar mit den Abschätzungen anderer aperiodischer Server, da diese bisher ebenfalls keine SMT-Eigenschaften einbeziehen.

4.2.3 Round Robin Slack

Ein dritter Scheduling-Algorithmus dient der Ausführung von Programmfäden ohne Echtzeitanforderungen. Derartige Programmfäden können zum Beispiel dafür verwendet werden, statistische Daten zur Ausführung zu erheben oder um die korrekte Funktionsweise einzelner Komponenten zu überprüfen, etc. Bei dieser Klasse von Programmfäden gibt es keine Fristen oder WCETs, stattdessen steht die *faire* Aufteilung der Rechenzeit unter den Programmfäden an erster Stelle.

In diesem Zusammenhang bedeutet Fairness, dass jeder Programmfaden den gleichen Anteil an der Rechenzeit erhält. Die Fairness kann ohne großen zusätzlichen Hardwareaufwand gewährleistet werden, indem der Instruktionszählungs-Mechanismus des PIQ-Algorithmus wiederverwendet wird. Der resultierende Scheduling-Algorithmus wird *Round Robin Slack (RRS)* [Mische 2010b] genannt, da die übriggebliebene Rechenzeit (engl. *slack time*) in einem Round-Robin-Verfahren verteilt wird.

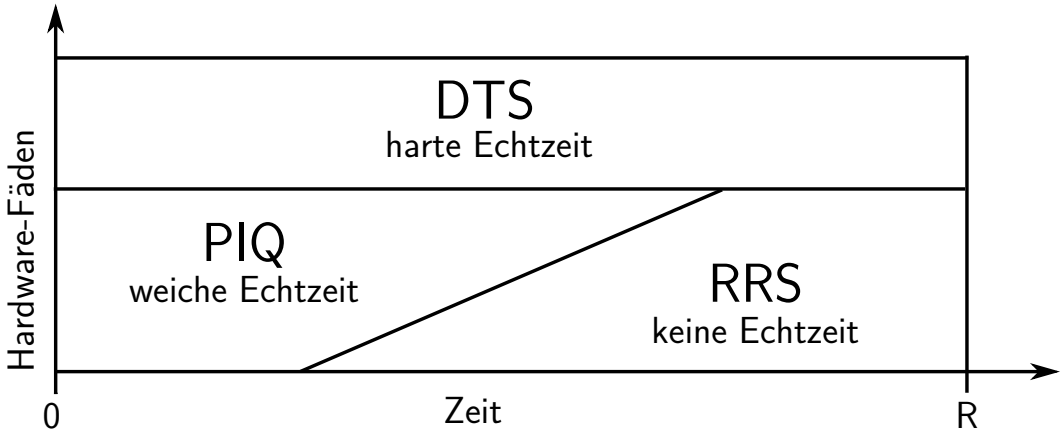

Abb. 4.11: Schematische Belegung der Hardware-Fäden während einer Runde

4.2.3.1 Algorithmus

Die Programmfäden werden der Reihe nach für eine bestimmte Anzahl von Instruktionen ausgeführt. Hat ein Programmfaden sein Quantum erreicht, wird er durch den nächsten RRS-Faden ersetzt. Wurden alle Programmfäden einmal ausgeführt, wird wieder mit dem ersten begonnen.

Der RRS-Algorithmus kann dazu benutzt werden, die nach DTS- und PIQ-Fäden noch verbleibende Rechenzeit sinnvoll für die Ausführung von Programmfäden zu nutzen. Die beiden DTS-Hardware-Fäden sind zwar tabu, aber die von den PIQ-Fäden genutzten Hardware-Fäden können zum Ende der Runde hin von den RRS-Fäden benutzt werden, soweit der PIQ-Scheduler sie nicht mehr benötigt (Abbildung 4.11).

Mit dem Beginn einer Runde werden zunächst die PIQ-Fäden eingelagert und verdrängen dadurch möglicherweise RRS-Fäden. Selbst wenn im weiteren Verlauf PIQ-Fäden ihr Instruktionsquantum erreichen und ausgelagert werden, werden sie durch weitere PIQ-Fäden ersetzt, bis alle PIQ-Fäden eingelagert wurden.

Erst wenn gegen Ende der Runde ein PIQ-Faden fertig wird und kein weiterer PIQ-Faden zur Verfügung steht (da alle bereits einmal in dieser Runde ausgeführt wurden), wird der erste RRS-Faden eingelagert und sein Instruktionszähler auf sein Quantum gesetzt. Analog zu den PIQ-Fäden wird sein Zähler bei jeder im zugeordneten ausgeführten Instruktion dekrementiert und beim Erreichen der Null wird der RRS-Faden per Kontextwechsel durch den nächsten RRS-Faden ersetzt. Werden weitere PIQ-Fäden fertig, so können auch mehrere RRS-Fäden gleichzeitig ausgeführt werden.

Die Prioritäten der Hardware-Fäden, die RRS-Fäden ausführen, sind grundsätzlich niedriger als die der DTS- und PIQ-Fäden und werden in Reihenfolge der Einlagerung vergeben, das heißt, der erste RRS-Faden hat die höchste Priorität, später dazukommende RRS-Fäden eine niedrigere. Dadurch soll erreicht werden, dass die RRS möglichst in der Reihenfolge fertig werden, in der sie auch gestartet wurden, um die Round-Robin-Reihenfolge nicht durcheinander zu bringen und die Fairness zu beeinträchtigen

4.2.3.2 Rundengrenze

Die RRS-Fäden unterliegen nicht der Beschränkung, dass sie nur einmal pro Runde ausgeführt werden dürfen (wie DTS- und PIQ-Fäden). Falls in einer Runde sehr viel Rechenzeit übrig ist oder nur wenige RRS-Fäden vorhanden sind, kann ein RSS-Faden beliebig oft in einer Runde ausgeführt werden, Bedingung ist nur, dass alle anderen RSS-Fäden in der Zwischenzeit einmal ausgeführt wurden.

Die Rundengrenze hat trotzdem eine Bedeutung für die RRS-Fäden, da mit dem Beginn einer neuen Runde die PIQ-Fäden wieder ausgeführt werden müssen und letztere aufgrund ihrer Echtzeitgarantien bevorzugt werden. Zu Beginn einer Runde werden deshalb die RRS-Fäden nach und nach durch PIQ-Fäden ersetzt. Da jeweils nur ein Kontextwechsel gleichzeitig stattfinden kann, bleiben die Kontexte einiger RRS-Fäden noch für die Dauer von ein, zwei Kontextwechseln in ihren Hardware-Fäden und werden auch noch ausgeführt, bis sie schließlich ersetzt werden.

Wenn in der neuen Runde alle PIQ-Fäden abgearbeitet wurden und wieder ein RRS-Faden eingelagert werden soll, so wird mit demjenigen begonnen, der vor der Auslagerung der RRS-Fäden als nächstes eingelagert worden wäre. Durch dieses Verfahren werden diejenigen RSS-Fäden benachteiligt, die noch kurz vor Ende der vorangegangenen Runde eingelagert und möglicherweise nur für wenige Instruktionen ausgeführt wurden. Um diesen Effekt auszugleichen, wird bei einer Zwangsauslagerung zu Beginn einer neuen Runde das verbleibende Instruktionsquantum gespeichert und bei der nächsten Einlagerung des gleichen RRS-Fadens zu seinem standardmäßigen Quantum zugeschlagen.

4.2.3.3 Gewichtung durch unterschiedliche Quanten

Eine offensichtliche Erweiterung des RRS-Algorithmus wäre es, durch unterschiedliche Quanten die Ausführung der RRS-Fäden zu gewichten. Ein höheres Quantum führt zwar meist zu einer häufigeren Ausführung des Programmfadens, wie die Evaluierung in Abschnitt 6.4.4 jedoch zeigt, wird die Rechenzeit nicht proportional zu den jeweiligen Quanten auf die Programmfäden verteilt. Dadurch ist eine genaue Regelung der Rechenzeitverteilung nicht möglich. Der Grund dafür ist, dass Programmfäden mit einem kurzem Quantum unter gewissen Umständen häufiger ein- und ausgelagert werden als Programmfäden mit längerem Quantum. Das folgende Beispiel verdeutlicht diesen Umstand:

Wenn ein RRS-Faden τ_1 ein deutlich größeres Instruktionsquantum hat als RRS-Faden τ_2 und letzterer kurz nach τ_2 eingelagert wird, so kann es sein, dass τ_2 trotz der niedrigeren Priorität vor τ_1 sein Quantum erreicht und ausgelagert wird. Falls nun die anderen RRS-Fäden ebenfalls relativ kurze Quanten haben, so ist es möglich, dass der Scheduler beim Durchgehen der RRS-Fäden wieder bei τ_1 ankommt und deshalb ihn als nächsten Programmfaden einlagern will.

Da der gleiche Programmfaden nicht in zwei Hardware-Fäden ausgeführt werden kann, bleiben vier Möglichkeiten:

(i) τ_1 wird übersprungen und die nachfolgenden RSS-Fäden werden nacheinander erneut eingelagert. In diesem Fall wird die Fairness verletzt, da τ_1 seltener ausgeführt wird als die anderen Programmfäden.

(ii) Zum Ausgleich könnte der aktuelle Instruktionszähler von τ_1 um sein Quantum erhöht werden. Dies führt ebenfalls nicht zur gewünschten Verteilung der Ausführungszeit, da eine Erhöhung des Quantums nicht zwangsläufig zu einer erhöhten Ausführung führt. Vielmehr gibt es eine obere Schranke für das Verhältnis von maximalem zu minimalem Quantum, ein höheres Quantum hat keinen Effekt (siehe Abschnitt 6.4.4).

(iii) Der RRS-Scheduler wartet zunächst, bis τ_1 sein altes Quantum erreicht hat. Dann erhält τ_1 ein neues Quantum und der Scheduler fährt ordnungsgemäß mit dem Einlagern des nächsten RRS-Fadens fort. Dadurch bleibt die Fairness erhalten, jedoch kann die Auslastung des Prozessors leiden, weil das Starten weiterer RRS-Fäden verzögert wird.

(iv) Wie bei (iii) wartet der RRS-Scheduler, bis τ_1 sein Quantum erreicht hat, nur werden die anderen RRS-Fäden nicht angehalten, wenn sie ihr Quantum erreichen. Dadurch wird die Rechenzeit weiterhin effektiv genutzt. Um dennoch Fairness zu gewährleisten, werden die Instruktionen, die vom Ende des Quantums bis zum wirklichen Auslagern ausgeführt wurden mitgezählt. Dieses überschüssige Instruktionsquantum wird beim Auslagern als negatives zusätzliches Quantum vermerkt und reduziert dadurch beim nächsten Einlagern das Quantum des betreffenden Programmfadens.

Wegen der größten Fairness ist die letzte Verhaltensweise standardmäßig aktiviert, durch Setzen des `SF_DONT_STALL`-Bits in den Scheduling-Attributen (siehe Tabelle 4.3) kann sie deaktiviert werden.

4.3 Implementierung

Nachdem die Scheduling-Algorithmen erklärt wurden, folgt in diesem Abschnitt eine Beschreibung der konkreten Implementierung. Hierbei wird unterschieden zwischen der internen Organisation der Hardwaremodule und der nach außen sichtbaren Programmierschnittstelle. Die Hardwaremodule bilden die zweite logische Ebene des Schedulers, die die erste Ebene der Prozessor-Pipeline mit der dritten Ebene, der Programmierschnittstelle verbindet. Für beide gelten unterschiedliche Entwicklungsziele: während die Programmierschnittstelle möglichst komfortabel sein sollte, spielt bei der Hardware-Entwicklung die Minimierung der Chipfläche die wichtigste Rolle.

Als gemeinsame Ziele können die Einfachheit und die Portabilität identifiziert werden. Eine geringe Komplexität erleichtert die Wartung und Erweiterung des Entwurfs und ein von der restlichen Architektur möglichst unabhängiges Konzept erlaubt den Einsatz in anderen (eventuell zukünftigen) Prozessoren und Echtzeitsystemen.

Feld	Beschreibung
`flags`	Scheduling-Algorithmus und zusätzliche Attribute
`next`	Zeiger auf den nächsten TCB mit gleicher Echtzeitklasse
`quantum`	Takt- (DTS) oder Instruktions- (PIQ, RRS) Quantum
`overhead`	Korrektur für nächstes Instruktionsquantum (nur RRS)

Tab. 4.1: Scheduling-Parameter eines Thread Control Blocks

Feld	Beschreibung
`dts_head`	Zeiger auf den ersten TCB der DTS-Liste
`piq_head`	Zeiger auf den ersten TCB der PIQ-Liste
`rrs_head`	Zeiger auf den ersten TCB der RRS-Liste
`roundlen`	Länge einer Runde in Takten

Tab. 4.2: Globale Scheduling-Daten in TCB 0

4.3.1 Programmierschnittstelle

Wie in Abschnitt 4.1 bereits erwähnt, besteht der Kontext eines Programmfadens aus den Werten aller Register, die zusammen den Zustand des Programmfadens beschreiben. Da nur eine begrenzte Anzahl von Hardware-Programmfäden zur Verfügung steht, können nicht alle Kontexte aller Programmfäden gleichzeitig in den Prozessor geladen und dort aktiv ausgeführt werden. Die Kontexte der inaktiven Programmfäden werden in den Speicher ausgelagert, wo sie darauf warten, wieder eingelagert und ausgeführt zu werden.

Die Speicherung erfolgt in sogenannten Kontrollblöcken (engl. *Thread Control Blocks, TCB*), die nicht nur den Kontext des Programmfadens enthalten, sondern auch Informationen darüber, wie oft er ausgeführt werden soll, die sogenannten *Scheduling-Parameter* (Tabelle 4.1). Die TCBs haben eine feste Länge und liegen aneinandergereiht in einem speziellen Adressbereich des Hauptspeichers. Sie werden mit null beginnend durchnummeriert, wobei der erste TCB mit der Nummer 0 eine spezielle Bedeutung hat. Er enthält nicht den Kontext eines Programmfadens, sondern *globale Scheduling-Daten*, siehe Tabelle 4.2.

Die TCBs werden getrennt nach dem zugehörigen Scheduling-Algorithmus verwaltet. Für jede der drei Scheduling-Klassen (DTS, PIQ, RRS) gibt es je eine einfach verkettete lineare Liste [Wirth 2000, Kapitel 4.3]. Während die Köpfe der Listen in den globalen Scheduling-Daten vermerkt sind, dient das `next`-Feld in den TCBs jeweils als Zeiger auf den nächsten TCB der gleichen Klasse. Mit null wird die Liste beendet.

In der DTS- und der RRS-Liste spielt die Reihenfolge der TCBs keine Rolle, da immer die ganze Liste abgearbeitet und erst dann wieder am Anfang begonnen wird. Daher ist es am einfachsten, neue TCBs am Beginn der Liste einzufügen. Bei der PIQ-Liste hat die Reihenfolge jedoch eine Bedeutung, da die TCBs am Ende der Liste möglicherweise nicht so oft ausgeführt werden, wie die TCBs am Anfang der Liste, falls die DTS-Fäden während einer Runde zu viele Ressourcen belegen.

Bit	Beschreibung
SF_DTS	DTS-Algorithmus (harte Echtzeitanforderungen) verwenden
SF_PIQ	PIQ-Algorithmus (weiche Echtzeitanforderungen) verwenden
SF_RRS	RRS-Algorithmus (keine Echtzeitanforderungen) verwenden
SF_DMAA	Zugriffsankündigung beim Programmfaden mit höchster Priorität
SF_DONT_STALL	Vermeide Blockierungen beim RSS-Algorithmus
SF_RESIDENT	Programmfaden darf nicht in den Speicher ausgelagert werden
SF_SUSPENDED	Ausführung des Programmfaden wird vorübergehend unterbunden
SF_TERMINATE	Programmfaden wurde mit sofortiger Wirkung beendet

Tab. 4.3: Scheduling-Attribute eines Programmfadens

4.3.1.1 Scheduling-Attribute

Durch das Feld `flags` eines TCBs wird das Verhalten des Programmfadens charakterisiert, Tabelle 4.3 listet die Bits auf, die gesetzt werden können. In erster Linie wird das Verhalten den Programmfadens durch den Scheduling-Algorithmus festgelegt, dieser wird durch das Setzen eines der Bits `SF_DTS`, `SF_PIQ` oder `SF_RRS` ausgewählt.

Je nach Scheduling-Algorithmus kann das Verhalten des Programmfadens variiert werden. Das Bit `SF_DMAA` aktiviert die Speicherzugriffsankündigung (DMAA, siehe Abschnitt 3.4.5), wenn der Programmfaden die höchste Priorität hat. Dadurch kann für einen DTS-Faden eine Verzögerung durch Speicherzugriffe konkurrierende Programmfäden verhindert werden. In Verbindung mit den anderen Scheduling-Algorithmen macht die Verwendung weniger Sinn, ist aber dennoch erlaubt.

Das `SF_DONT_STALL`-Bit hat nur in Verbindung mit dem RSS-Algorithmus eine Bedeutung, es verhindert, dass der Scheduler beim Durchgehen der RSS-Liste bei einem TCB, dessen Programmfaden gerade ausgeführt wird, solange stehen bleibt, bis er ausgelagert wird, siehe Abschnitt 4.2.3.3.

Durch das `SF_RESIDENT`-Bit wird verhindert, dass ein Programmfaden, der gerade ausgeführt wird, in den Speicher ausgelagert wird. Er läuft also weiter, auch wenn sein Quantum bereits verbraucht ist. Den genau entgegengesetzten Effekt hat das Setzen der Bits `SF_SUSPENDED` bzw. `SF_TERMINATE`. Sie dienen dazu, die Ausführung sofort zu beenden und den Kontext des Programmfadens in den Speicher zu schreiben, unabhängig vom verbleibenden Quantum. Der Unterschied zwischen den beiden Bits besteht darin, dass durch `SF_SUSPENDED` die Ausführung nur vorübergehend unterbrochen wird, wohingegen bei `SF_TERMINATE` die Ausführung unwiederbringlich beendet wird.

Durch das Setzen eines der beiden Bits wird die weitere Ausführung zwar verhindert, doch der entsprechende TCB steht weiterhin in einer der Scheduling-Listen. Da der TCB bei jedem Durchgang der Liste erneut gelesen wird, sollte der TCB deshalb so schnell wie möglich aus seiner Liste entfernt werden, um unnötige Speicherzugriffe zu vermeiden. Dies gilt uneingeschränkt nach einer Beendigung mit `SF_TERMINATE`, da ein so markierter TCB nicht wieder aktiviert werden kann, bei einer Unterbrechung der Ausführung durch

SF_SUSPENDED kann es jedoch Sinn machen, den TCB in der Liste zu belassen, um eine möglichst schnelle Reaktivierung (durch das Löschen des Bits) zu ermöglichen.

4.3.1.2 Konsistenz der Scheduling-Daten

Für die Verwaltung der TCBs werden einfach verkettete Listen verwendet, da Einfügungen und Entfernungen so durchgeführt werden können, dass die Liste stets konsistent ist[1], spezielle atomare Instruktionen sind dafür nicht notwendig. Die ständige Konsistenz ist wichtig, da der Hardware-Scheduler unabhängig von der Pipeline auf die Scheduling-Daten zugreift und deshalb in keinen inkonsistenten Zustand geraten darf.

Durch die einfach verketteten Listen wird zwar die Konsistenz zwischen Hardware-Scheduler und Software-Schnittstelle gewahrt, jedoch können gleichzeitige Veränderungen durch zwei unterschiedliche Programmfäden widersprüchliche Scheduling-Daten erzeugen. Deshalb ist das Ändern der Scheduling-Daten ein sogenannter *kritischer Abschnitt*, der einen *gegenseitigen Ausschluss* erfordert, das heißt, nur jeweils ein Programmfaden darf an den Scheduling-Daten Veränderungen vornehmen, andere Programmfäden müssen warten bis dessen Bearbeitungen abgeschlossen sind. Implementiert werden kann der gegenseitige Ausschluss eines kritischen Bereichs durch ein *Mutex* (vgl. [Ungerer 1997, 54ff]).

Ein weiteres Problem entsteht durch die Möglichkeit des Auslagerns von Programmfäden. Wird nämlich ein Programmfaden ausgelagert, während er gerade die Scheduling-Daten verändert, so kann abermals ein inkonsistenter Zustand erzeugt werden. Folglich reicht es nicht aus, den kritischen Abschnitt mit der Modifikation der Scheduling-Daten nur durch ein Mutex zu schützen, zusätzlich muss das Auslagern des Programmfadens durch das Setzen des SF_RESIDENT-Bits in den Scheduling-Attributen verhindert werden.

Doch selbst dann kann ein undefinierter Zustand eintreten, falls ein Programmfaden versucht, sich selbst vorübergehend zu deaktivieren. In diesem Fall wird er zuerst sein SF_RESIDENT-Bit setzen, um ein Auslagern zu verhindern, dann den kritischen Bereich mit Hilfe des Mutex betreten, den eigenen TCB aus der passenden TCB-Liste entfernen, den kritischen Bereich per Mutex verlassen und das SF_RESIDENT-Bit wieder löschen und das SF_SUSPENDED-Bit setzen, um sich selbst zu deaktivieren.

Ein Problem tritt auf, wenn zwischen dem Löschen des SF_RESIDENT-Bits und dem Setzen des SF_SUSPENDED-Bits der Programmfaden in den Speicher ausgelagert wird. Zunächst ist das Verhalten noch korrekt, die Ausführung wird erwartungsgemäß unterbrochen. Doch sobald ein anderer Programmfaden den TCB wieder in eine Liste einfügt und der Hardware-Scheduler schließlich den Kontext einlagert und ausführt, ist der erste Befehl das Setzen des SF_SUSPENDED-Bits, der Programmfaden wird also sofort wieder deaktiviert, anstatt weiterzulaufen.

[1] Um Y zwischen X und Z einzufügen, wird zunächst Z als Nachfolger von Y eingetragen, anschließend Y als Nachfolger von X. Für das Entfernen ist sogar nur eine Instruktion nötig, die den Nachfolger des Vorgängers neu setzt.

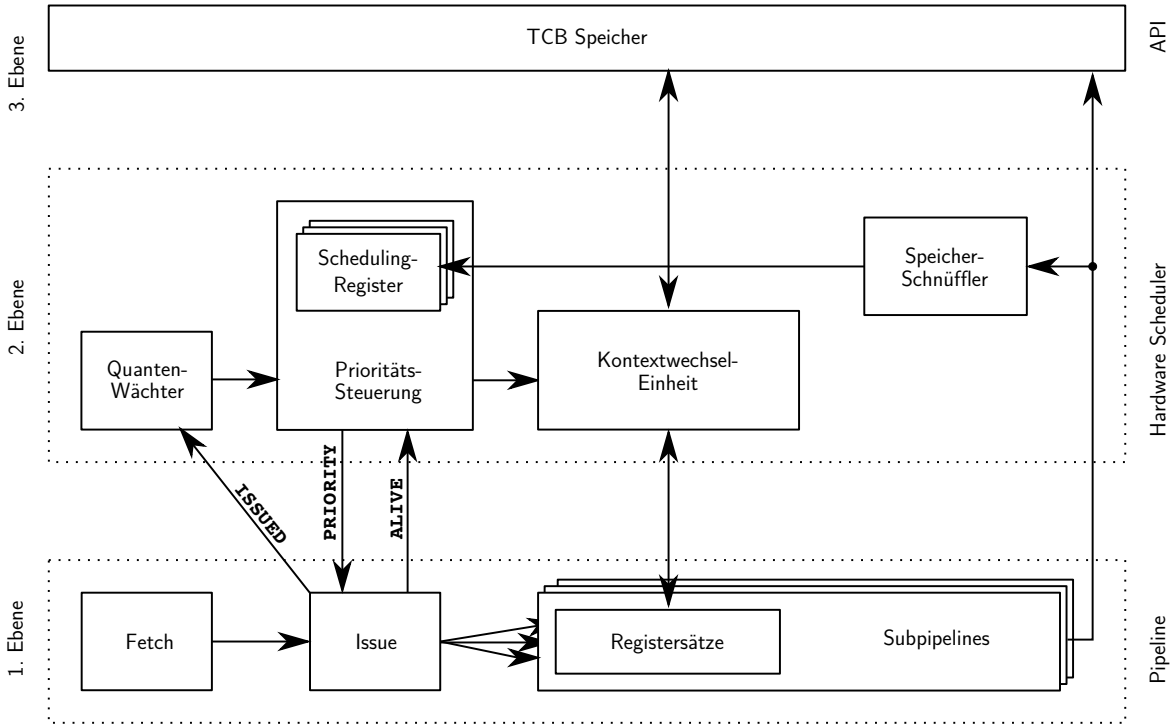

Abb. 4.12: Blockdiagramm des Hardware Schedulers

Durch früheres Setzen des `SF_SUSPENDED`-Bits kann das Problem nicht vermieden werden, da der Programmfaden sofort nach dem Setzen angehalten wird und keine weiteren Befehle mehr ausführt, der kritische Bereich folglich nie verlassen wird.

Einziger Ausweg sind zwei spezielle Instruktionen, `tie` und `untie`, die das `SF_RESIDENT`-Bit des aktuellen Programmfadens setzen bzw. löschen, aber gleichzeitig eine weitere Veränderung der Scheduling-Attribute verhindern. In einem von `tie` und `untie` begrenzten Abschnitt können die Scheduling-Attribute zwar verändert werden, diese Veränderung wird jedoch erst nach der Ausführung des `untie`-Befehls aktiv.

4.3.2 Hardware Realisierung

Die Funktionalität der zweiten Ebene des Schedulers könnte theoretisch auch durch Software implementiert werden, da die elementaren Operationen zur Ablaufplanung bereits durch die unterste Ebene (die isolierte In-Order-SMT-Ausführung) garantiert werden, jedoch wäre dies mit einem erheblichen Bedarf an Rechenleistung verbunden, der dann nicht mehr für die eigentliche Applikation zur Verfügung stehen würde.

Um die zweite Ebene des Hardware-Schedulers dennoch möglichst architekturunabhängig und portabel zu gestalten, wurde darauf geachtet, dass sie möglichst wenige, einfache Schnittstellen zu den benachbarten Ebenen hat. Das schematische Blockdiagramm in Abbildung 4.12 offenbart die Verbindungen.

Die Kontextwechseleinheit ist über breite bidirektionale Leitungen mit dem Registersatz und dem TCB-Speicher verbunden, da hier relativ große Datenmengen übertragen werden müssen. Der Informationsaustausch zwischen der Zuordnungsstufe und der zweiten Schicht beschränkt sich auf drei unidirektionale Signale und der Speicherschnüffler hört die ohnehin existierende Leitung von der Pipeline zum Speicher nur ab.

4.3.2.1 Kontextwechseleinheit

Die höchste Hardware-Komplexität weisen zweifellos die beiden Verbindungen der Kontextwechseleinheit (engl. *Context Switch Unit*) mit dem TCB-Speicher und dem Registersatz auf. Der Aufwand ist so hoch, da in beiden Fällen sowohl Schreib- als auch Lesezugriffe möglich sein müssen und diese jeweils parallel zu anderen Zugriffen auf die Registersätze bzw. den TCB-Speicher stattfinden können. Diese hohen Hardware-Kosten werden damit gerechtfertigt, dass dadurch Kontextwechsel in den Speicher vollständig parallel zur normalen Programmausführung durchgeführt werden können und keinerlei gegenseitige Beeinflussung auftritt.

Positiv auf den Platzverbrauch der Hardware wirkt es sich aus, dass die Kontextwechseleinheit zwar grundsätzlich auf jedes Register Zugriff haben muss, es aber nie zu einer Kollision mit einem Registerzugriff der Prozessor-Pipeline kommen kann, da die Kontextwechseleinheit nur auf inaktive, die Prozessor-Pipeline hingegen nur auf aktive Hardware-Fäden zugreift.

Um die Verwendung von teurem Speicher mit zwei gleichzeitigen Zugriffskanälen (engl. *dual-ported memory*) zu vermeiden, können Speicherzugriffe der Pipeline so lange verzögert werden, bis ein gleichzeitig stattfindender Kontextwechsel abgeschlossen ist. Dadurch werden die Speicherzugriffszeiten auf den TCB-Speicher zwar enorm verlängert (insbesondere die WCET), dies spielt jedoch nur eine untergeordnete Rolle, da die Zugriffe auf den TCB-Speicher meist in die Initialisierungsphase fallen und nicht in zeitkritischen Programmteilen vorkommen.

Beim Auslagern eines Hardware-Fadens werden nicht nur die Registerwerte, sondern auch die Scheduling-Parameter (Tabelle 4.1) in den TCB-Speicher geschrieben, analog werden sie beim Einlagern zusätzlich zum Kontext gelesen. Diese Parameter werden aber nicht an die Pipeline weitergegeben, sondern bereits innerhalb der zweiten Schicht in den Scheduling-Registern zwischengespeichert. Gesteuert wird die Kontextwechseleinheit durch die Prioritätssteuerung (siehe unten), die zu bestimmten Zeitpunkten das Ein- bzw. Auslagern eines bestimmten Hardware-Fadens anweist.

4.3.2.2 Speicherschnüffler

Der Hardware-Scheduler greift niemals direkt auf den TCB-Speicher zu, da dies einen zusätzlichen teuren Zugriffskanal kosten würde. Stattdessen beobachtet er alle Speicherzugriffe der Prozessor-Pipeline und filtert die den TCB-Speicher betreffenden Zugriffe heraus. Dieses Verfahren wird „ schnüffeln“(engl. *snooping*) genannt, da der Datenstrom

nur passiv abgehört wird, nicht aber selbst gesteuert oder verändert wird.

Nicht einmal alle TCB-Zugriffe sind relevant, nur bei Zugriffen auf die Scheduling-Parameter der TCBs, deren Kontext gerade in einem Hardware-Faden ausgeführt wird, werden die Scheduling-Register der Prioritätssteuerung aktualisiert. Außerdem werden Zugriffe auf die globalen Scheduling-Daten in TCB 0 erkannt und entsprechend in die internen Datenstrukturen übernommen.

4.3.2.3 Quantenwächter

Jedem Hardware-Faden ist ein Zählregister zugeordnet, das in einer geeigneten Weise dekrementiert wird. Beim Einlagern eines Programmfadens wird der Zähler auf das entsprechende Quantum des Programmfadens gesetzt und dann solange dekrementiert, bis er null erreicht. Das Erreichen der Null wird der Prioritätssteuerung gemeldet, die die Ausführung des entsprechenden Hardware-Fadens beendet und den Programmfaden gegebenenfalls auslagert.

Bei DTS-Fäden wird der Zähler in jedem Takt um eins erniedrigt, bei PIQ- und RRS-Fäden hängt die Veränderung jedoch vom `ISSUED`-Signal ab, das von der Zuordnungsstufe kommt. Durch dieses Signal teilt die Zuordnungsstufe dem Quantenwächter mit, wie viele Instruktionen von welchem Hardware-Faden im letzten Takt ausgeführt wurden. Entsprechend werden die Zähler erniedrigt.

Durch einen zusätzlichen Zähler wird ein periodisches Signal erzeugt, das der Prioritätsteuerung den Beginn einer neuen Runde anzeigt.

4.3.2.4 Prioritätssteuerung

Die zentralen Entscheidungen der Ablaufplanung werden in der Prioritätssteuerung der zweiten Scheduler-Ebene getroffen. Hier laufen die Informationen aus den anderen Teilen des Hardware-Schedulers zusammen, werden bewertet und in Prioritäten für die Zuordnungsstufe umgerechnet. Eine weitere Aufgabe ist das Anstoßen von Kontextwechseln, die dann selbstständig von der Kontextwechseleinheit durchgeführt werden.

Sobald der Quantenwächter das Ende eines Quantums anzeigt, wird die Ausführung des entsprechenden Hardware-Fadens gestoppt, indem die Priorität einen speziellen, ungültigen Wert erhält. Falls ein DTS-Faden gestoppt wurde, wird die Priorität des Hardware-Fadens mit dem anderen DTS-Faden auf das Maximum gesetzt, ohne einen Takt der Ausführungszeit zu vergeuden.

Mit dem Auslagern des Hardware-Fadens kann jedoch nicht sofort begonnen werden, da der Hardware-Faden sich möglicherweise mitten in der Ausführung einer mehrtaktigen Instruktion befinden. In diesem Fall muss die Ausführung der Instruktion erst abgeschlossen werden, erst dann signalisiert die Zuordnungsstufe durch Löschen des `ALIVE`- Signals des entsprechenden Hardware-Fadens, dass der Kontext nun in den Speicher geschrieben werden kann. Auch ohne eine vorherige Aufforderung durch die Prioritätssteuerung

kann die Zuordnungsstufe das `ALIVE`-Signal löschen, wenn der Hardware-Faden durch eine spezielle Instruktion (`yield`) selbst eine Auslagerung verlangt.

Der eigentliche Auslagerungsvorgang wird durch die Kontextwechseleinheit durchgeführt, die Prioritätssteuerung gibt nur das Startsignal weiter, wenn das `ALIVE`-Signal gelöscht wird. Während der Kontext in den Speicher kopiert wird, hat die Prioritätssteuerung Zeit, zu entscheiden, welcher derzeit ausgelagerte TCB den bisherigen Programmfaden ersetzen soll. Die einzelnen Scheduling-Klassen werden nacheinander geprüft:

(i) Falls ein DTS-Faden beendet wurde, so wird er durch den übernächsten DTS-Faden ersetzt (der nächste war bereits geladen worden und wird schon ausgeführt). Wenn die DTS-Liste nicht mehr genügend Einträge enthält, wird wieder der erste DTS-Faden geladen, aber erst zu Beginn der nächsten Runde aktiviert.

(ii) Sofern noch nicht das Ende der PIQ-Liste erreicht wurde, es also noch PIQ-Fäden gibt, die in der aktuellen Runde noch nicht ausgeführt wurden, so wird der nächste PIQ-Faden geladen.

(iii) Andernfalls wird der nächste RRS-Faden geladen, am Ende der RRS-Liste wird einfach wieder von Anfang an begonnen, die Runden spielen hier keine Rolle.

Am Beginn einer Runde ist das Verhalten der Prioritätssteuerung leicht abweichend. Zuerst bekommt der Hardware-Faden, der den ersten DTS-Faden enthält, die höchste Priorität und der andere DTS-Hardware-Faden wird deaktiviert. Dann wird die Kontextwechseleinheit angewiesen, den alten DTS-Faden auszulagern und durch den zweiten DTS-Faden zu ersetzen. Anschließend ermittelt die Prioritätssteuerung einen Hardware-Faden, der einen RSS-Faden ausführt. Dieser wird dann durch den ersten PIQ-Faden ersetzt. Nach diesem Kontextwechsel wird ein weiterer RSS-Faden durch den nächsten PIQ-Faden in der PIQ-Liste ersetzt, solange bis nur noch DTS- und PIQ-Fäden die Hardware-Fäden belegen, oder das Ende der PIQ-Liste erreicht wurde.

Wenn die Ausführung eines RRS-Fadens unterbrochen und er zurück in den Speicher geschrieben wird, wird sein verbleibendes Instruktionsquantum ebenfalls in den TCB zurückgeschrieben, das Feld `quantum` (Tabelle 4.1) ist dafür vorgesehen. Wenn der TCB das nächste Mal eingelagert wird, so wird dieser Wert zu seinem normalen Quantum addiert, um den vorzeitigen Abbruch auszugleichen und eine gleichmäßige Ausführung der RRS-Fäden zu erreichen.

Da ein Kontextwechsel mehrere Takte dauert, kann es passieren, dass der Quantenwächter das Ende eines Quantums signalisiert, während die Kontextwechseleinheit noch beschäftigt ist. In diesem Fall setzt die Prioritätssteuerung die Priorität des betreffenden Hardware-Fadens auf den niedrigsten möglichen Wert und beginnt den Kontextwechsel im Anschluss an den derzeitigen Kontextwechsel. Auf diese Weise wird die Ausführung der anderen Hardware-Fäden nicht beeinträchtigt, etwaige freie Ressourcen liegen andererseits aber nicht brach, sondern können für die Ausführung des wartenden Programmfadens genutzt werden.

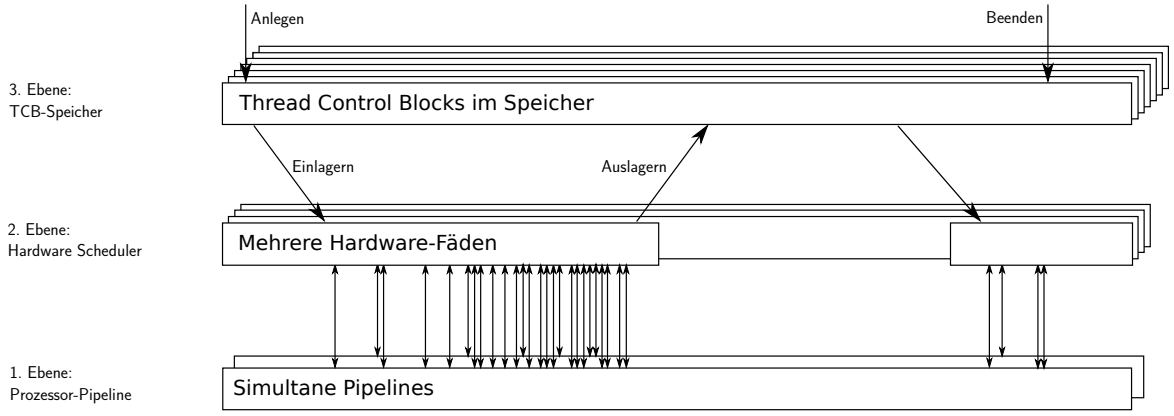

Abb. 4.13: Lebenszyklus eines Programmfadens von der Erstellung bis zur Beendigung

4.4 Beispiele

Zwei Beispiele sollen die Funktionsweise des Schedulers verdeutlichen. Zuerst wird der „Lebenszyklus“ eines Threads, von der Erstellung bis zur Beendigung dargestellt, im weiteren ein ganzes Taskset während zwei aufeinanderfolgenden Runden.

4.4.1 Lebenszyklus eines Programmfadens

Ein Programmfaden wird erstellt, indem ein anderer Programmfaden einen TCB reserviert und dort Startwerte für die Register einträgt. Die meisten Startwerte spielen im Normalfall keine Rolle, wichtig sind nur spezielle Register wie der Stackpointer, der auf einen freien Speicherbereich zeigen sollte und der Programmzähler, der auf die Startadresse des Programmfadens gesetzt werden muss.

Außerdem muss durch das Setzen der Scheduling-Attribute der Scheduling-Algorithmus festgelegt werden und das gewünschte Takt- oder Instruktionsquantum in den TCB eingetragen werden. Anschließend wird der TCB durch einfache Speicherzugriffe in eine der drei Scheduling-Listen eingehängt. Da der Speicherschnüffler nur Teile des TCB-Speichers überwacht, ist es nicht unwahrscheinlich, dass der Hardware-Scheduler den neu hinzugekommenen Programmfaden nicht direkt bemerkt. Durch das regelmäßige Durchgehen der Scheduling-Listen erkennt er ihn aber spätestens dann, wenn er zum ersten Mal eingelagert werden muss. Bei DTS- und PIQ-Fäden ist das zwangsläufig in der folgenden Runde der Fall, bei RSS-Fäden kann es je nach Auslastung des Prozessors auch länger dauern.

Sobald der Scheduler entschieden hat, dass der Programmfaden ausgeführt werden soll, da ein Hardware-Faden bereit steht, beginnt der Einlagerungsprozess: Der Kontext des Programmfadens wird vom TCB-Speicher in den Registersatz des entsprechenden Hardware-Fadens kopiert, die Scheduling-Attribute werden gelesen und der Zähler des Hardware-Fadens wird auf den Wert des Scheduling-Quantums aus dem TCB gesetzt.

Der Zeitpunkt, wann ein bestimmter Programmfaden eingelagert wird, hängt vom Scheduling-Algorithmus ab. Bei DTS-Fäden ist es immer der gleiche Zeitpunkt innerhalb einer Runde, außer ein neuer DTS-Faden kommt hinzu, dann verschieben sich die Zeitpunkte einmalig und bleiben dann für die folgenden Runden gleich. Der genaue Einlagerungszeitpunkt von PIQ-Fäden hängt dagegen von der Auslastung des Prozessors ab, jeder Programmfaden wird aber genau einmal pro Runde eingelagert und die Reihenfolge der Einlagerungen bleibt gleich. Bei RSS-Fäden bleibt die Reihenfolge ebenfalls gleich (außer dies wird durch das `SF_DONT_STALL`-Attribut explizit verhindert), jedoch sind die Einlagerungen nicht an Rundengrenzen gebunden, das heißt, es können mehrere pro Runde erfolgen, aber auch viele Runden zwischen zweien liegen.

Wenn die Einlagerung abgeschlossen ist, bekommt der Hardware-Faden eine Priorität zugewiesen, die an die Zuordnungsstufe der Prozessor-Pipeline weitergereicht wird. Dadurch wird der Hardware-Faden aktiviert und in die Pipeline-Verarbeitung einbezogen, das heißt, es werden Instruktionen geholt und durch die Pipeline geschickt. Bei DTS-Fäden kann eine gewisse Zeit zwischen dem Ende der Einlagerung und der Aktivierung liegen, da sie erst mit Ablauf des Quantums des vorhergehenden DTS-Fadens aktiviert werden.

Die Priorität hängt wiederum vom Scheduling-Algorithmus ab. Es ist nur jeweils ein DTS-Faden zu einem bestimmten Zeitpunkt aktiv, der jeweils die höchste Priorität hat, damit er von keinen anderen Programmfäden beeinflusst wird. Anschließend folgen die Prioritäten der PIQ-Fäden, wobei schon früher eingelagerte PIQ-Fäden höhere Prioritäten haben, das heißt, ein neu hinzukommender PIQ-Faden erhält die niedrigste Priorität aller PIQ-Fäden. RSS-Fäden haben grundsätzlich niedrigere Prioritäten als PIQ-Fäden und sie sind ebenfalls nach absteigendem Alter verteilt.

Die Ausführung eines Programmfadens wird unterbrochen, sobald sein Quantum abgelaufen ist. Bei DTS-Fäden ist das nach einer festen Anzahl Takte der Fall, bei PIQ- und RRS-Fäden nach einer festen Anzahl ausgeführter Instruktionen – oder schon früher, falls das Ende der Runde erreicht wurde. Bis zur Unterbrechung werden DTS-Fäden gleichmäßig mit höchster Priorität ausgeführt, als wären sie der einzige auszuführende Hardware-Faden.

PIQ-Fäden beginnen jedoch mit einer sehr niedrigen Priorität, diese steigt aber, indem ältere PIQ-Fäden mit höheren Prioritäten nach und nach ausgelagert werden und die Priorität des betrachteten PIQ-Fadens entsprechend „nachrutscht“. Er wird kurz nach dem Einlagern nur sehr sporadisch ausgeführt, weshalb sein Instruktionszähler sich nur langsam verringert. Nach und nach wird die Ausführung aufgrund der höheren Priorität häufiger und dadurch der Zähler schneller erniedrigt.

Zum Ende des Quantums hin kann er sogar die höchste Priorität aller Hardware-Fäden erreichen, soweit kein konkurrierender DTS-Faden vorhanden ist (entweder weil keine DTS-Fäden im Taskset enthalten sind, oder weil sie nicht die gesamte Rundenlänge beanspruchen). Das gleiche gilt für RRS-Fäden, nur dass hier erst dann höhere Prioritäten erreicht werden, wenn die Zähler aller PIQ-Fäden bereits abgelaufen sind.

		Parameter	Algorithmus	Quantum
τ_1	HRT	$C_1 = 240$, $T_1 = 1500$	DTS	$Q_1 = 120$ Takte
τ_2	HRT	$C_2 = 180$, $T_2 = 750$	DTS	$Q_2 = 180$ Takte
τ_3	HRT	$C_3 = 900$, $T_3 = 2250$	DTS	$Q_3 = 300$ Takte
τ_4	SRT	$j_4 = 0,2$	PIQ	$q_4 = 150$ Instruktionen
τ_5	SRT	$j_5 = 0,333$	PIQ	$q_5 = 250$ Instruktionen
τ_6	NRT		RRS	

Tab. 4.4: Ein Beispiel-Taskset mit sechs Programmfäden.

Wenn der Zähler des Programmfadens null erreicht, wird der entsprechende Hardware-Faden gestoppt und es beginnt die Auslagerung. Dazu werden die Registerwerte in den TCB-Speicher geschrieben. Die Scheduling-Daten müssen nicht zurückgeschrieben werden, da sie durch den Hardware-Scheduler nicht verändert wurden. Sie können nur durch Schreibzugriffe von der Pipeline in den TCB-Speicher verändert werden und in diesem Fall steht selbstverständlich anschließend der richtige Wert im Speicher.

Durch das Auslagern wird ein Hardware-Faden frei, der durch einen anderen Programmfaden belegt werden kann. Der Hardware-Scheduler entscheidet, wann ein Programmfaden wieder vom TCB-Speicher in einen Hardware-Faden geladen und ausgeführt wird. Dadurch wechseln die Programmfäden ständig zwischen TCB-Speicher und Hardware-Fäden hin und her, bis sie das Ende ihrer Ausführung erreichen.

Um einen Programmfaden zu beenden, muss sein TCB aus der entsprechenden Liste entfernt werden, damit er beim nächsten Durchlauf durch die Scheduling-Listen nicht mehr vom Hardware-Scheduler erkannt wird. Um den Programmfaden auch zu beenden, wenn er gerade in einem Hardware-Faden ausgeführt wird (dies ist insbesondere immer der Fall, falls ein Programmfaden sich selbst beendet), wird in seinen Scheduling-Attribute das `SF_TERMINATE`-Bit gesetzt, was die sofortige Aussetzung der Ausführung nach sich zieht.

4.4.2 Komplettes Taskset

Gegeben sei ein Taskset mit sechs Programmfäden (Tabelle 4.4), das auf einen Prozessor mit vier Hardware-Fäden ausgeführt wird. Ein Kontextwechsel soll $\Delta t_{CS} = 70$ Takte dauern, wobei $\Delta t_{out} = 34$ Takte auf das Auslagern und $\Delta t_{in} = 36$ Takte auf das Einlagern entfallen.

Für die Programmfäden mit harten Echtzeitanforderungen ist jeweils die WCET C_i und die Periode T_i gegeben, für die weichen Echtzeitfäden ist eine Ziel-IPC j_i vorgegeben und beim Programmfaden ohne Echtzeitbedingungen sind keine weiteren Parameter nötig. Aus den Echtzeitanforderungen ergibt sich direkt der zu wählende Algorithmus und das jeweilige Quantum kann aus den gegebenen Parametern berechnet werden.

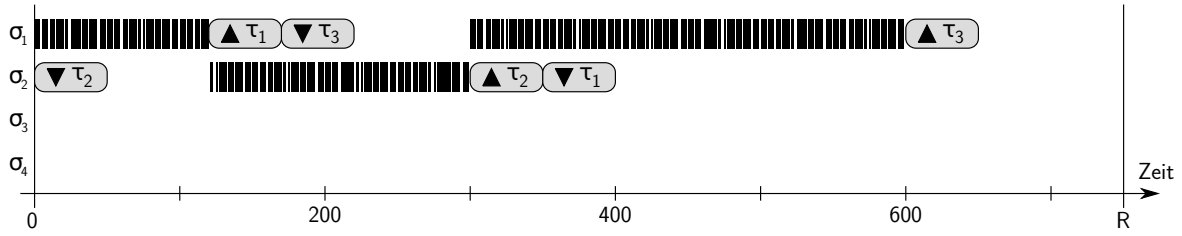

Abb. 4.14: Eine Scheduling-Runde (nur DTS-Fäden)

4.4.2.1 Vorabberechnungen

Gemäß (4.5) entspricht die Rundenlänge R dem größten gemeinsamen Teiler der Perioden

$$R = \text{ggT}(1500, 750, 2250) = 750 \tag{4.9}$$

Die Taktquanten ergeben sich aus (4.3)

$$Q_1 = 750 \cdot \frac{240}{1500} = 120 \tag{4.10}$$

$$Q_2 = 750 \cdot \frac{180}{750} = 180 \tag{4.11}$$

$$Q_3 = 750 \cdot \frac{900}{2250} = 300 \tag{4.12}$$

und für die Instruktionsquanten muss nur die gewünschte IPC mit der Rundenlänge multipliziert werden

$$q_4 = 750 \cdot 0,2 = 150 \tag{4.13}$$

$$q_5 = 750 \cdot 0,333 = 250 \tag{4.14}$$

Da es nur einen RRS-Faden gibt, spielt sein Quantum keine Rolle. Da alle Quanten größer als die Kontextwechselzeit $\Delta t_{CS} = 70$ sind, müssen keine weiteren Anpassungen vorgenommen werden (vgl. Abschnitt 4.2.1.3).

4.4.2.2 DTS-Fäden

Abbildung 4.14 zeigt beispielhaft den Ablauf einer Runde mit dem gegebenen Taskset. Auf der waagrechten Achse ist die Zeit in Takten angegeben, auf der senkrechten Achse sind die vier Hardware-Fäden σ_1 bis σ_4 aufgetragen. Da die DTS-Fäden völlig unabhängig von den anderen Programmfäden ausgeführt werden, werden zunächst nur diese betrachtet. Dadurch reduziert sich die Betrachtung der Hardware-Fäden auf σ_1 und σ_2, da DTS-Fäden keine weiteren Hardware-Fäden belegen.

Bereits vor Beginn der Runde wurde τ_1 in σ_1 geladen und wird deshalb dort ab Beginn der Runde mit höchster Priorität ausgeführt. Ebenfalls am Anfang der Runde beginnt das Einlagern des zweiten DTS-Fadens τ_2 in den Hardware-Faden σ_2.

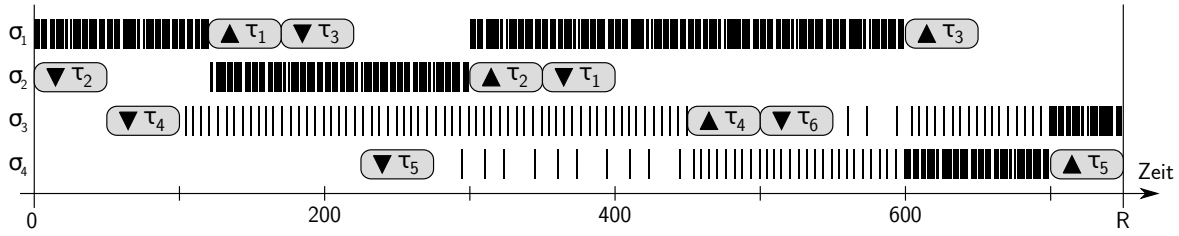

Abb. 4.15: Eine Scheduling-Runde (alle Programmfäden sind dargestellt)

In Takt 120 ist das Quantum von τ_1 ($Q_1 = 120$) abgelaufen, deshalb wird der entsprechende Hardware-Faden σ_1 deaktiviert und sofort mit dem Auslagern von τ_1 begonnen. Gleichzeitig wird τ_2 in σ_2 aktiviert und mit höchster Priorität ausgeführt. Nachdem σ_1 ausgelagert wurde, wird der dritte DTS-Faden τ_3 geladen und ersetzt τ_1 in Hardware-Faden σ_1. Nach $\Delta t_{CS} = 70$ Takten, also in Takt 190 ist dies abgeschlossen.

τ_2 wird noch bis zu Takt 300 mit höchster Priorität in σ_2 ausgeführt, dann ist auch sein Zeitschlitz abgelaufen und σ_2 wird deaktiviert, gleichzeitig mit der erneuten Aktivierung von Hardware-Fadenσ_1, der jetzt aber Programmfaden τ_3 ausführt. Analog zum vorherigen Kontextwechsel wird der gerade deaktivierte Programmfaden τ_2 ausgelagert und durch den nächsten DTS-Faden ersetzt. Da es keinen weiteren DTS-Faden gibt, wird wieder der erste DTS-Faden τ_1 geladen, damit er zu Beginn der nächsten Runde zur Verfügung steht.

Der Zeitschlitz von τ_3 endet $Q_3 = 300$ Takte später in Takt 600. Zu diesem Zeitpunkt wird σ_1 deaktiviert und τ_3 ausgelagert. Da keine weiteren DTS-Fäden zur Verfügung stehen, sind von Takt 600 bis zum Ende der Runde bei Takt 750 die beiden, für das DTS-Scheduling reservierten Hardware-Fäden deaktiviert. Der Zeitraum von Takt 600 bis 750 entspricht dem Spielraum Δt_{slack} (vgl. Abschnitt 4.2.2.1).

4.4.2.3 Weitere Programmfäden

Um die Auslastung des Prozessors zu erhöhen, können die weiteren Programmfäden des Tasksets zusätzlich ausgeführt werden. Da sie grundsätzlich mit niedrigeren Prioritäten als die DTS-Fäden ausgeführt werden, ändert sich an der Ausführung der DTS-Fäden in den Hardware-Fäden σ_1 und σ_2 nichts, die zusätzliche Programmausführung findet nur in den restlichen Hardware-Fäden statt, wie Abbildung 4.15 zeigt.

Sobald der Scheduler nicht mit den DTS-Fäden beschäftigt ist, kümmert er sich darum, die Auslastung zu erhöhen. Dies ist zum ersten Mal der Fall, nachdem τ_2 in σ_2 geladen wurde. Direkt anschließend (ab Takt 34) wird der erste PIQ-Faden τ_4 in Hardware-Faden σ_3 eingelagert und schnellstmöglich aktiviert. Da er mit niedrigerer Priorität als die DTS-Fäden ausgeführt wird, dauert es relativ lang, bis $q_4 = 150$ Instruktionen ausgeführt wurden und er wieder ausgelagert werden kann.

Da noch ein weiterer Hardware-Faden zur Verfügung steht, kann ein weiterer PIQ-Faden geladen werden, sobald die Kontextwechseleinheit nicht belegt ist. In dem Beispiel ist

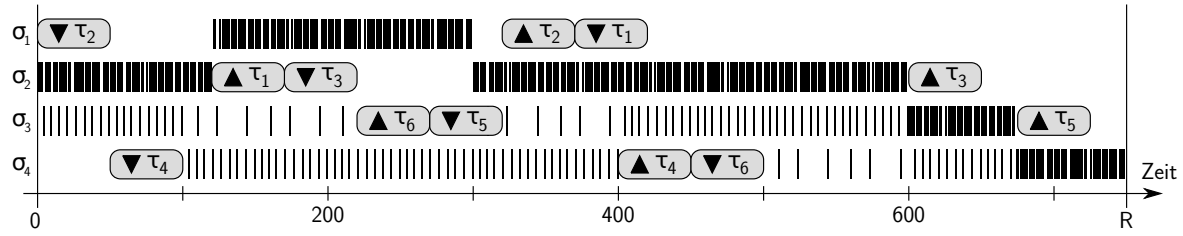

Abb. 4.16: Weitere Scheduling-Runde mit dem gleichen Taskset

dies nach dem Kontextwechsel von τ_1 auf τ_3 in Hardware-Faden σ_1 in Takt 190 der Fall. Zu diesem Zeitpunkt wird der PIQ-Faden τ_5 in den Hardware-Faden σ_4 geladen. Bei dessen Ausführung sieht man sehr schön, wie die Ausführungswahrscheinlichkeit mit der Zeit höher wird (dichtere Linien), da die Hardware-Fäden mit höheren Prioritäten nach und nach deaktiviert werden. Sobald das Instruktionsquantum erreicht ist (im Beispiel kurz vor Ende der Runde), wird τ_5 wieder ausgelagert.

In Takt 484 ist das Auslagern von PIQ-Faden τ_4 abgeschlossen, es steht folglich wieder ein Hardware-Faden zur Verfügung, da alle PIQ-Fäden bereits einmal eingelagert wurden, kann nun ein RRS-Faden gestartet werden. Da der daraufhin eingelagerte RRS-Faden τ_6 erst sehr spät innerhalb der Runde aktiviert wurde, erreicht er sein Quantum nicht vor Ende der Runde und bleibt deshalb über das Rundenende hinweg in σ_4 geladen und wird auch weiterhin ausgeführt.

4.4.2.4 Nächste Runde

Die darauffolgende Runde verläuft, wie alle Runden, sehr ähnlich, es gibt aber einige kleine Unterschiede, wie Abbildung 4.16 zeigt. Da die Anzahl der DTS-Fäden ungerade ist, wird τ_1 am Ende der ersten Runde in σ_2, nicht in σ_1 geladen und wird dementsprechend dort in der zweiten Runde ausgeführt. Die Hardware-Fäden σ_1 und σ_2 tauschen ihre Rollen, an den Kontextwechselzeitpunkten ändert sich jedoch nichts.

Mehr Veränderungen sieht man in den Hardware-Fäden σ_3 und σ_4, die Nicht-DTS-Fäden ausführen. Da RRS-Fäden eine niedrigere Priorität als PIQ-Fäden haben, könnte τ_6 in σ_3 durch τ_4 ersetzt werden. Da zum Beginn der Runde aber σ_4 ungenutzt ist, ist es zur Steigerung der Gesamtauslastung günstiger, zuerst τ_3 in σ_4 zu laden und erst dann den RSS-Faden τ_6 durch den zweiten PIQ-Faden τ_5 zu ersetzen. Dadurch wird die Zeit, in der σ_4 sonst ungenutzt wäre, noch zur Ausführung des RSS-Faden genutzt.

Die Abbildung zeigt weiterhin, dass der Abstand zwischen Einlagerung und Auslagerung variieren kann, da er nicht durch eine feste Taktanzahl, sondern durch eine feste Instruktionsanzahl gegeben ist, die aufgrund unterschiedlicher Ressourcennutzung schneller oder langsamer erreicht werden kann.

5 Der CarCore-Prozessor

Um die in den vorhergehenden Kapiteln beschriebene Architektur zu evaluieren, wurde ein Prototyp auf Basis des Infineon-TriCore-Prozessors geschaffen. Der TriCore wurde als Basisarchitektur ausgewählt, da er eine superskalare In-Order-Ausführung unterstützt und in Echtzeitsystemen der Automobilindustrie oft eingesetzt wird.

Mit dem TriCore 2 hat Infineon zwar bereits einen mehrfädigen Prozessor auf Basis der TriCore-Architektur vorgestellt, dieser unterstützt aber nur grobkörnige Mehrfädigkeit. Obwohl er bereits 2002 vorgestellt wurde, ist kein Einsatz dieses Prozessors in einem kommerziellen System bekannt. Selbst in den Marketing-Prospekten von Infineon spielt er keine Rolle mehr, stattdessen wird die TriCore-1-Linie weiterentwickelt.

Der in diesem Kapitel beschriebene *CarCore*-Prozessor implementiert eine Untermenge des TriCore-1-Befehlssatzes, die es erlaubt, nahezu jedes für den TriCore kompilierte Programm auszuführen. Die Pipeline wurde Kapitel 3 entsprechend modifiziert, um feinkörnige mehrfädige Programmausführung zu erlauben und mit einen Hardware-Scheduler wie in Kapitel 4 verbunden.

5.1 Der Infineon-TriCore-Prozessor

Der Name *TriCore* hat nichts mit einem Dual-Core-Prozessor zu tun, das heißt, er rührt nicht daher, dass der Prozessor aus drei Prozessorkernen zusammengesetzt ist, sondern steht dafür, dass der Prozessor drei Prozessorarchitekturen zu vereinen sucht: Er verbindet die Merkmale eines echtzeitfähigen Mikrocontrollers mit der Rechenstärke eines Signalprozessors und der schnellen, kostengünstigen Ausführung eines superskalaren RISC-Prozessors [Schultes 1999].

Es existieren mehrere Versionen der TriCore-Architektur, wovon jedoch nur die minimal unterschiedlichen Versionen 1.3 und 1.3.1 in öffentlich zugänglichen Dokumenten [Infineon 2008a] beschrieben werden. Informationen zu mindestens zwei weiteren Architekturvarianten, der Version 2.0 des nicht mehr weiterentwickelten Tricore 2, sowie der Version 1.6 der Anfang 2011 veröffentlichten Infineon-AUDO-MAX-Reihe, beschränken sich meist auf Marketing-Prospekte [Infineon 2011]. Aus diesem Grund beziehen sich alle weiteren Aussagen über die TriCore-Archtektur ausschließlich auf die Version 1.3.

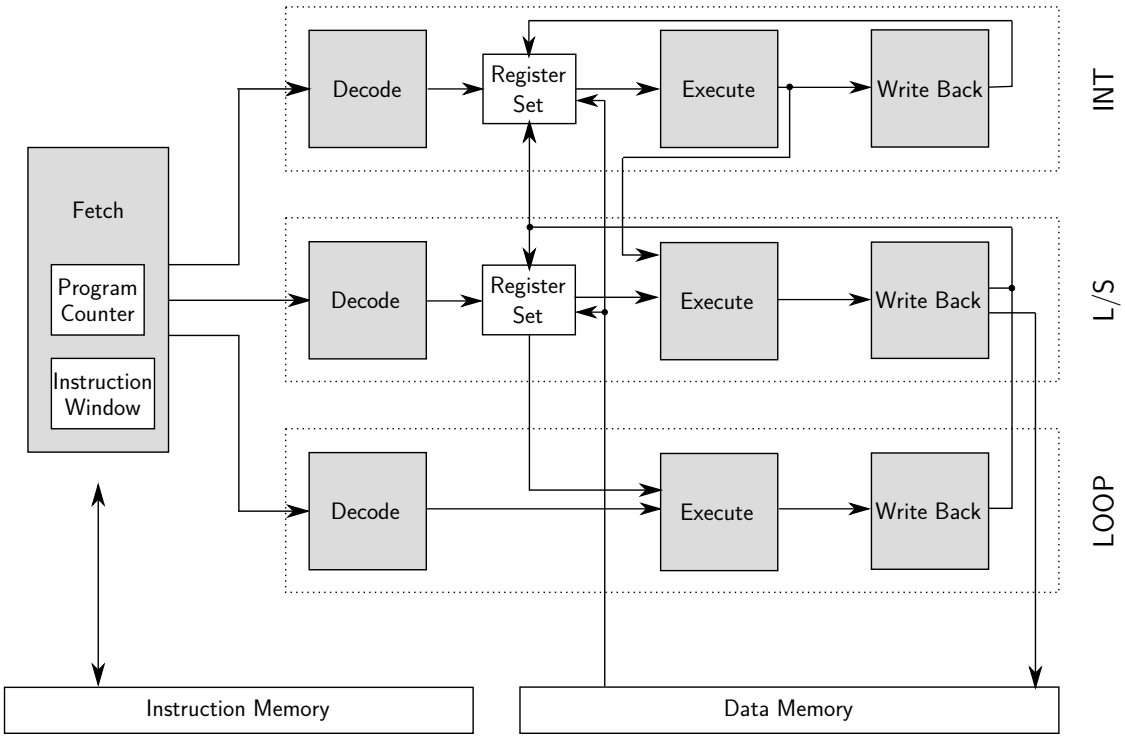

Abb. 5.1: Blockschaltbild der TriCore-Pipeline

5.1.1 Mikroarchitektur

Die Prozessor-Pipeline des TriCore-Prozessors (Abbildung 5.1) besteht aus vier Stufen – Fetch, Decode, Execute und Writeback – wobei die letzten drei Stufen jeweils dreimal vorhanden sind und jeweils eine Subpipeline bilden. Die Subpipelines sind jeweils für die Ausführung einer disjunkten Untermenge des Befehlssatzes verantwortlich und tragen folgende Namen:

INT Ausführung von arithmetisch-logischen Befehlen auf ganzen Zahlen

L/S Adressberechnung und Speicherzugriffe

LOOP Spezielle Schleifen ohne Sprung-Overhead

Da pro Prozessortakt jeweils ein Befehl jeder Untermenge ausgeführt werden kann, können bis zu drei Instruktionen gleichzeitig ausgeführt werden. Dazu müssen die Befehle aber genau in der Reihenfolge INT - L/S - LOOP vorliegen. Es kann jederzeit eine Untermenge ausgelassen werden, sobald die Reihenfolge jedoch verletzt wird, wird die Ausführung auf den nächsten Takt verschoben.

Die LOOP-Subpipeline verarbeitet nur zwei Befehle – einen Sprungbefehl, der ein Register herunterzählt und einen unbedingten Sprungbefehl. Die Zieladressen der letzten beiden LOOP-Sprungbefehle werden zwischengespeichert, so dass in speziellen Schleifen der Schleifensprung parallel zur letzten Instruktion ausgeführt werden kann und dadurch keine Verzögerung erzeugt.

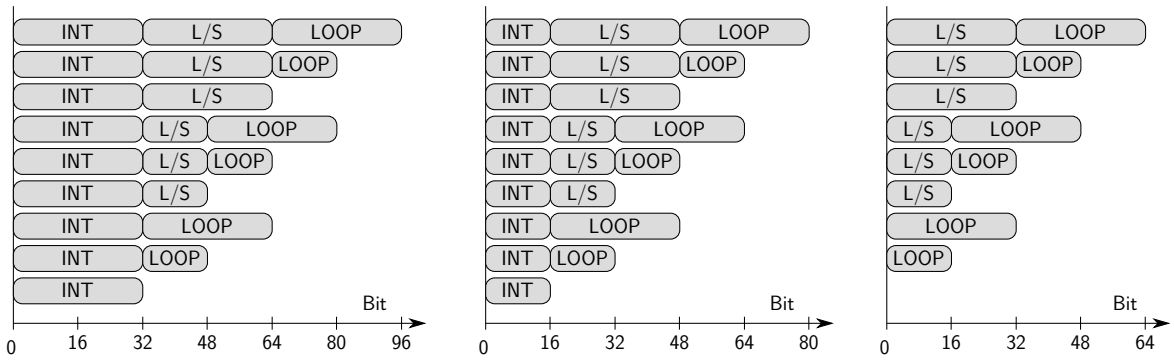

Abb. 5.2: Die 26 verschiedenen Kombinationen von TriCore-Befehlen

Die beiden anderen Subpipelines unterstützen hingegen sehr viele unterschiedliche Instruktionen. Sie sind sehr ähnlich aufgebaut, jede hat einen eigenen Registersatz von 16 32-Bit-Registern und kann pro Takt eine Instruktion ausführen. Die Integer-Befehle arbeiten nur auf den internen Registern der INT-Subpipeline, auf den Speicher kann nur durch spezielle Lade- und Speicher-Befehle der L/S-Subpipeline zugegriffen werden.

Neben den klassischen Adressierungsarten (absolut, Register-indirekt, prä-inkrement, post-inkrement) stehen zwei weitere, sehr spezielle Adressierungsarten zur Verfügung. Dies sind die sogenannte Bit-Reverse-Addressierung zur Beschleunigung von schnellen Fourier-Transformationen und die Circular-Adressierung für Ringpuffer.

Instruktionen können 16 oder 32 Bit lang sein. Die Länge ist im niederwertigsten Bit des Befehlswortes kodiert, 0 steht für eine kurze 16-Bit-Instruktion, 1 für eine lange 32-Bit-Instruktion. Auf die gleiche Weise ist die Subpipeline-Zugehörigkeit im zweitniedrigsten Bit kodiert, 0 steht für die L/S-Subpipeline, 1 für die INT-Subpipeline. Die beiden LOOP-Instruktionen werden durch 252 (hexadezimal FC_{16}) für die 16-Bit-Version bzw. 253 (FD_{16}) für die 32-Bit-Version im niederwertigen Byte markiert.

Da die Befehle In-Order ausgeführt werden und die Untermengen in einer festen Reihenfolge aufeinander folgen müssen, kann man die Befehlskodierung auch als eine sehr kompakte Kodierung eines VLIW-Befehlswortes von 3×32 Bit interpretieren. In Abbildung 5.2 sind die 26 möglichen Kodierungen aufgelistet.

Die meisten Befehle werden in einem Takt ausgeführt, es gibt jedoch einige Instruktionen, die mehrere Takte dauern.

5.1.2 Befehlssatz

Der Befehlssatz des TriCore [Infineon 2008b] ist sehr umfangreich. Es gibt insgesamt 312 Instruktionen und 39 Opcode-Formate, was zusammen 766 Varianten ergibt. Die Befehle können grob in zehn Gruppen eingeteilt werden, siehe Abbildung 5.3. Jede dieser Gruppen kann genau einer Subpipeline zugeordnet werden, nur der Gruppe der Sprungbefehle gehören Befehle aller drei Subpipelines an.

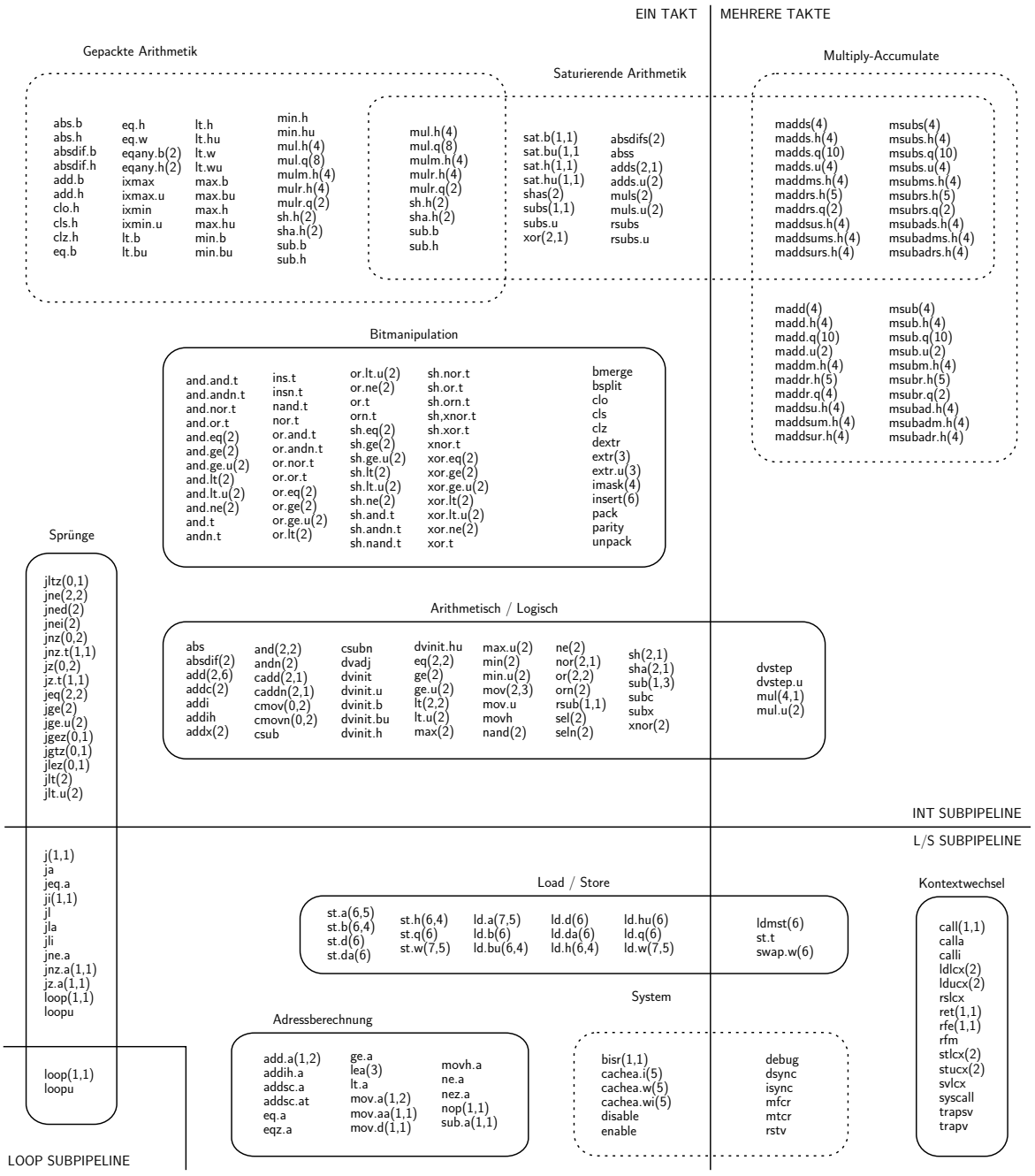

Abb. 5.3: Die Befehle des TriCore 1. Falls es mehr als eine 32-Bit-Variante des Befehls gibt, so ist diese Anzahl in Klammern angegeben. Wenn es außerdem 16-Bit-Varianten gibt, steht die zweite Zahl für deren Anzahl. Die gestrichelt umrandeten Befehlsgruppen wurden im CarCore nicht implementiert.

Neben den klassischen arithmetisch-logischen Funktionen bietet der TriCore einen umfangreichen Satz an Befehlen zur Bitmanipulation, für saturierende Arithmetik (bei einem Überlaufs wird der größtmögliche Wert zurückgeliefert) und für die Manipulation von gepackten Datentypen (ein 32-Bit-Wort wird als zwei 16-Bit-Werte oder als vier 8-Bit-Werte interpretiert). All diese Instruktionen werden in der INT-Subpipeline innerhalb eines Taktes ausgeführt, nur die Multiply-Accumulate-Befehle benötigen zwei Takte, die Divisionsbefehle sogar vier.

In der L/S-Subpipeline stehen abgesehen von den Sprung- und den Load-/Store-Befehlen auch einfache arithmetische Instruktionen zur Verfügung, die zur Adressberechnung verwendet werden können. Außerdem gibt es Befehle zur Verwaltung des Mikrocontroller-Systems (Cache-Synchronisation, Interrupt-Handling, etc.) und für schnelle Kontextwechsel (nächster Abschnitt).

5.1.3 Kontextwechsel

Eine Besonderheit des TriCore stellen die Befehle zum schnellen Kontextwechsel dar. Zu diesem Zweck ist der Registersatz des TriCore in zwei Teile geteilt, den oberen und den unteren Kontext. Bei einem Funktionsaufruf (`call`-Befehl) wird der obere Kontext automatisch in den Speicher geschrieben und beim Verlassen der Funktion wieder aus dem Speicher gelesen. Dadurch können die Register des oberen Kontextes innerhalb einer Funktion beliebig verändert werden, die Register des unteren Kontextes dienen zur Parameterübergabe und Funktionswertrückgabe.

Auch bei der Unterbrechung des normalen Programmablaufs durch Interrupts, Traps oder Exceptions wird der obere Kontext automatisch gesichert. Falls die Unterbrechungsroutine den gesamte Registersatz benützen möchte, kann durch spezielle Befehle auch der untere Kontext gesichert werden.

Ein Kontext umfasst 16 Register, also 64 Bytes, die in einer verketteten Liste in einem speziellen Speicherbereich (der sogenannten *Context Save Area*) abgelegt werden. Die Verwaltung dieser Liste liegt vollständig in der Hand der Hardware, die Liste muss nur einmal am Beginn der Programmausführung von der Software initialisiert werden.

Da das Speichern beziehungsweise Lesen von 64 Byte relativ viel Zeit in Anspruch nehmen kann, ist der obere Kontext beim TriCore zweimal vorhanden, sodass bei einem Unterprogrammaufruf nur zwischen den beiden Kontexten gewechselt werden muss. Erst wenn ein weiterer geschachtelter Unterprogrammaufruf folgt, muss wirklich in den Speicher geschrieben werden. Laut Infineon können durch diesen Mechanismus 70-80% der Unterprogrammaufrufe beschleunigt werden [Infineon 2003, Seite 23]. In den anderen Fällen sorgt eine spezielle 128-Bit-Speicherschnittstelle trotzdem für eine relativ geringe Verzögerung.

5.2 Hohe Komplexität der TriCore-Architektur

Da der Entwurf des TriCore-Befehlsatzes maßgeblich von einem Compiler-Hersteller beeinflusst wurde [Hoogenboom 1997], stand bei der Entwicklung des TriCore eine kompakte Instruktionskodierung mit großer Funktionsvielfalt im Vordergrund. Dieser Optimierung der Software-Bedürfnisse wurde eine Optimierung aus Hardware-Sicht untergeordnet.

Der Entwurf hat einige Eigenschaften, die nur mit großem Hardware-Aufwand realisiert werden können und die maximale Taktfrequenz des Prozessors stark eingrenzen. Ein weiterer Nachteil der hohen Komplexität, der im Rahmen dieser Arbeit eine Rolle spielte, ist der hohe zeitliche Aufwand für den Nachbau der Architektur. Entgegen der Marketing-Aussagen über den TriCore [Schultes 1999] ist er eindeutig ein CISC-Prozessor [Patterson 1980].

Schon die komfortable Kontextsicherung bei Unterprogrammaufrufen deutet auf das Designziel hin: der Aufwand des Programmierers und des Compilers wird auf ein Minimum reduziert, da keine Register explizit gesichert werden müssen. Das wird erkauft durch relativ langsame Unterprogrammaufrufen und einen 50% größeren Registersatz, der diesen Nachteil einigermaßen auffangen soll.

Es gibt Instruktionen für sehr komplexe Aufgaben, beispielsweise kann ein Bitmuster beliebiger Länge an einer beliebigen Stelle innerhalb eines 32-Bit-Wertes eingefügt werden. Diese Operation erfordert vier Eingabewerte, das heißt, der Registersatz muss vier unabhängige Lesezugriffe innerhalb eines Taktes erlauben. Es ist zwar kein Problem einen derartigen Registersatz zu konstruieren, er verbraucht aber sehr viel Chipfläche.

Eine weitere Eigenschaft, die nicht nur die Chipgröße weiter aufbläht, sondern vor allem die Taktfrequenz stark herabsetzt, ist die direkte Weiterverarbeitung des Ergebnisses eines INT-Befehl im direkt darauffolgenden L/S-Befehl, der im gleichen Takt ausgeführt wird. Die Ausführung einer derart abhängigen L/S-Instruktion kann deshalb erst beginnen, wenn die Berechnungen der INT-Instruktion abgeschlossen sind und beides muss innerhalb der engen Grenzen eines Taktes geschehen.

Die weitaus größte Schwäche des TriCores ist aber seine komplizierte Instruktions-Dekodierung. Sie hat drei Gründe:

(i) Variable Instruktionslänge.

(ii) 39 Opcode-Formate mit unterschiedlich positioniertem sekundären Opcodes.

(iii) Inkonsistenzen im Befehlssatz.

In jedem Takt werden drei Instruktionen geholt, wobei jede Instruktion eine von drei Längen haben kann: 0, 16 oder 32 Bit. Allein dadurch wird die Dekodierung bereits sehr komplex (siehe Abbildung 5.2). Es kommt jedoch hinzu, dass zwar durch ein Bit unterschieden werden kann, ob die Instruktion 16 oder 32 Bit lang ist, wenn sie jedoch die Länge 0 hat, also ausgelassen wurde, so kann das nur daran erkannt werden, dass anstatt der erwarteten Instruktionsuntermenge die Instruktion einer anderen Untermenge folgt.

Beispielsweise muss der Dekoder erkennen, dass kein L/S-Befehl ausgeführt werden soll, wenn ein LOOP-Befehl direkt auf einen INT-Befehl folgt. Folgen zwei Befehle der gleichen Subpipeline direkt aufeinander, bedeutet dies, dass in den anderen Subpipelines keine weiteren Befehle gleichzeitig mit dem ersten ausgeführt werden, und so weiter.

Ist bereits die Trennung der einzelnen Befehlsuntermengen ein zeitaufwendiges Problem, so wird es bei der Dekodierung der einzelnen Befehle nicht besser. Die hohe Zahl der Opcode-Formate erfordert es, dass zu Beginn der Dekodierung anhand der acht niederwertigsten Bit entschieden wird, welches Opcode-Format gültig ist. Erst in einem zweiten Schritt können die zu lesenden Register bestimmt und der sekundäre Opcode ausgewertet werden.

Weiter erschwert wird die Dekodierung durch Inkonsistenzen im Befehlssatz. So können viele Instruktionen mit einer Bedingung wie „größer gleich“ oder „kleiner“ verknüpft werden. Die Bitmuster, mit denen die Bedingungen kodiert werden unterscheiden sich aber von Befehl zu Befehl. Es gibt immer wieder Spezialfälle in denen ein einzelner Befehl nicht in das Schema der anderen passt. Diese unregelmäßige Anordnung der Instruktionen bläht die für die Dekodierung nötige Logik unnötig auf.

5.3 Vereinfachungen im CarCore

Um den zeitlichen Aufwand für die Entwicklung des CarCore-Prototypen in Grenzen zu halten, wurde versucht, die Komplexität der TriCore-Architektur auf das Nötigste zu reduzieren, ohne die Funktionsfähigkeit einzuschränken. Dadurch ist es möglich, die TriCore-Toolchain für den CarCore zu verwenden. Jedes in ANSI C geschriebene Programm, das mit dem *HighTec GNU C/C++ TriCore Compiler 3.3.7.9*[1] oder dem *Altrium TASKING TriCore C Compiler 3.0*[2] kompiliert wurde, kann auf dem CarCore ausgeführt werden.

Bei den Vereinfachungen wurde berücksichtigt, dass

(i) die Echtzeitfähigkeit nicht beeinträchtigt wird,

(ii) die Ausführungsgeschwindigkeit nur minimal veringert wird und

(iii) eine mehrfädige Programmausführung möglich ist.

Die Veränderung sind im einzelnen [Mische 2010a]:

Weniger Adressierungsarten Die beiden speziellen Adressierungsarten Bit-Reverse und Circular werden von keinem der beiden Compiler verwendet, weshalb sie im CarCore ersatzlos gestrichen werden.

Reduzierter Befehlssatz Es werden nur Instruktionen unterstützt, die die Compiler auch wirklich generieren. Keiner der beiden Compiler verwendet saturierende Be-

[1] HighTec EDV-Systeme GmbH, http://www.hightec-rt.com/, abgerufen am 1. August 2011

[2] TASKING Embedded software development tools from Altium, http://www.tasking.com/, abgerufen am 1. August 2011

fehle, gepackte Datentypen oder Multiply-Add-Instruktionen, das heißt, diese Befehlsgruppen fallen heraus. Mit Ausnahme von `mfcr` und `mtcr` werden auch keine Systembefehle unterstützt. Sogar die Bitmanipulationsbefehle könnten entfallen, wenn sie nicht durch die handoptimierten Fließkommabibliotheken des TASKING Compilers verwendet werden würden. Dadurch (und durch die Reduzierung der Adressierungsarten) wird die Anzahl der Befehlsvarianten auf 433 verringert.

Keine Befehlssatzerweiterungen Einige TriCore-Modelle unterstützen zusätzliche Befehle, zum Beispiel zur Ansteuerung einer Speicherverwaltungseinheit (engl. *memory management unit, MMU*) oder für Fließkommaberechnungen mit einfacher Genauigkeit. Außerdem wurde der Basis-Befehlssatz beim TriCore 2 um spezielle Befehle für schnelle Unterprogrammaufrufe und 128-Bit-Speicherzugriffe erweitert. Keine der Befehlssatzerweiterungen wird vom CarCore unterstützt.

Keine Doppel-Pipeline-Befehle Instruktionen, die sowohl auf Register der INT- als auch der L/S-Subpipeline zugreifen, belegen beim TriCore beiden Subpipelines gleichzeitig. Beim CarCore hingegen werden sie nur in der L/S-Subpipeline ausgeführt, die INT-Subpipeline bleibt frei. Dadurch ist zwar ein zusätzlicher Zugriffspfad von der L/S-Subpipeline zum INT-Registersatz nötig, andererseits spart man sich Logik bei der Zuordnung der Instruktionen ein. Da durch die mehrfädige Erweiterung die Zuordnungsstufe stark erweitert wird, ist es sinnvoll, sie so einfach wie möglich zu halten.

Spätere Adressberechnung Im CarCore werden die Adressen von Sprungzielen und Speicherzugriffen erst in der Ausführungsstufe berechnet, eine Stufe später als im TriCore. Dadurch wird die Latenz von Sprungbefehlen und Speicherzugriffen um eins erhöht, andererseits wird dadurch die Dekodierstufe vereinfacht. Da die Dekodierstufe die Pipelinestufe mit der längsten Ausführungszeit ist, kann bei einer Beschleunigung dieser Stufe der ganze Prozessor höher getaktet werden.

Keine Sprungvorhersage Eine Sprungvorhersage reduziert die durchschnittliche Ausführungszeit, im ungünstigsten Fall muss jedoch immer eine falsche Voraussage angenommen werden. Deshalb wird durch eine Sprungvorhersage die WCET nicht reduziert, sie kann sogar erhöht werden, da ein falsch vorhergesagter Sprung häufig länger dauert als ein nicht vorhergesagter Sprung. Aus diesem Grund ist eine Sprungvorhersage für einen Programmfaden mit harten Echtzeitanforderungen nicht von Vorteil. Programmfäden mit niedrigeren Prioritäten profitieren ebenso wenig, da die Sprunglatenz durch die Ausführung von Programmfäden mit höherer Priorität überdeckt werden. Durch den Verzicht auf eine Sprungvorhersage entsteht folglich kein Nachteil, vielmehr werden weniger Prozessorressourcen durch falsch vorhergesagte Befehle blockiert, wodurch der Gesamtdurchsatz des Prozessors erhöht wird.

Schnelle Division Im TriCore 1 dauert die Ausführung eines Divisionsbefehls fünf Takte [Infineon 2003, Seite 38], im CarCore hingegen nur einen. Da ein Divisionsbefehl (`dvstep`) nur durch eine 8-Bit-Zahl dividiert, sind für eine vollständige 32-Bit-Division vier Instruktionen nötig, das heißt, eine Division dauert auf dem CarCore

vier statt 20 Takte.

Keine LOOP-Subpipeline Die LOOP-Subpipeline des TriCores beschleunigt spezielle Schleifen mit einer festen Anzahl Schleifendurchläufe. Die beiden unterstützten Compiler erkennen jedoch nur sehr selten Fälle, in denen sie eingesetzt werden können. Um die Zuordnungslogik zu vereinfachen, werden im CarCore deshalb LOOP-Befehle in der L/S-Subpipeline wie normale Sprungbefehle ausgeführt. Dadurch erhöht sich zwar die Ausführungszeit (auch die WCET) der Programmfäden, der Gesamtdurchsatz wird aber kaum verringert, da während der Sprunglatenzen andere Programmfäden die Prozessorressourcen nutzen können.

Langsame Unterprogrammaufrufe Wie in Abschnitt 5.1.3 erläutert, wird beim TriCore mit hohem technischen Aufwand versucht, Unterprogrammaufrufe möglichst zu beschleunigen. Diese Techniken werden beim CarCore nicht eingesetzt, wodurch Unterprogrammaufrufe etwa um den Faktor zehn langsamer werden (siehe Evaluierung Abschnitt 6.2.2).

5.4 CarCore Architektur

Abbildung 5.4 zeigt den prinzipiellen Aufbau der CarCore-Pipeline. Im Vergleich zur TriCore-Pipeline (Abbildung 5.1) hat sie eine Subpipeline weniger, da die LOOP-Befehle in der L/S-Subpipeline ausgeführt werden. Hinzugekommen ist die separate Zuordnungsstufe, da durch mehrere Programmfäden die Zuordnung so komplex wird, dass sie nicht mehr mit der Dekodierung in einer Stufe kombiniert werden kann, wie es beim TriCore der Fall ist.

5.4.1 Befehlsbereitstellung

In der Fetch-Stufe können pro Takt 64 Bit aus dem Instruktionsspeicher gelesen werden. Da eine Instruktion maximal 32 Bit lang ist und nur zwei Subpipelines versorgt werden müssen, reicht dieser Wert gemäß Gleichung (3.2) aus, um beide Subpipelines ständig mit Instruktionen beliefern. Zu Verzögerungen aufgrund der Befehlsbereitstellung kommt es nur bei Sprüngen.

Da die Latenz des Instruktionsspeichers null beträgt und die Pipelinestufen optimal angeordnet sind, wäre nach Gleichung (3.3) eine Befehlsfenstergröße von 2×64 Bit ausreichend. Da die Instruktionen gemäß der TriCore-Definition nicht an vollen 64-Bit-Adressen ausgerichtet werden müssen, der CarCore-Speichercontroller aber nur auf volle 64-Bit-Adressen zugreifen kann, muss das Befehlsfenster jedes Hardware-Fadens 3 Einträge à 64 Bit umfassen.

Im einfachsten Fall werden solange Befehle für den Hardware-Faden mit der höchsten Priorität geholt, bis dessen Befehlsfenster voll ist. Erst dann für denjenigen mit der zweithöchsten Priorität, dann für den dritten, und so weiter. Sobald in einem Befehlsfenster

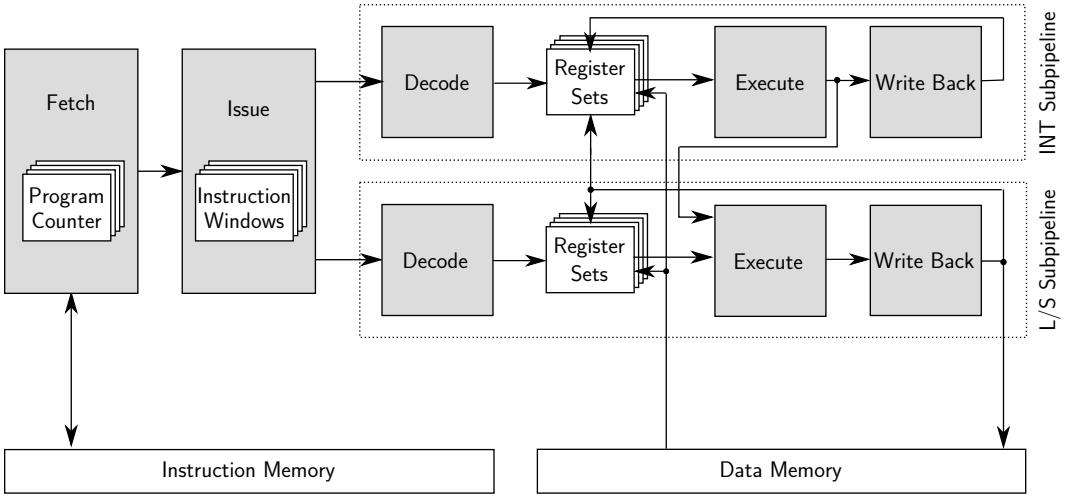

Abb. 5.4: Blockschaltbild der CarCore-Pipeline

eines Hardware-Fadens mit einer hohen Priorität wieder ein Eintrag frei wird, wird dieser bevorzugt gefüllt. Dies garantiert die maximale Auslastung des harten Echtzeitfadens mit der höchsten Priorität.

5.4.2 Optimierung der Befehlsbereitstellung

Durch eine genaue Analyse der geholten Instruktionen kann die Anzahl der Speicherzugriffe jedoch reduziert werden, ohne die Ausführungsgeschwindigkeit des Hardware-Faden mit der höchsten Priorität zu beeinträchtigen. Dies ist möglich, da häufig das Befehlsfenster mit Befehlen gefüllt wird, die durch einen Sprung wieder verworfen werden. Im CarCore wurden zwei Verfahren implementiert [Mische 2010a]:

Bei der ENOUGH-Technik werden bereits in der Fetch-Stufe die Längen der enthaltenen Instruktionen ermittelt. Wie in Abschnitt 3.2.2 erläutert, muss das Befehlsfenster genügend Befehle für $L_{F,I}$ Takte enthalten. Beim CarCore gilt $L_{F,I} = 2$, es reichen also vier Befehle aus. Falls diese in einer ungünstigen Reihenfolge vorkommen, reichen sogar schon weniger Instruktionen aus (Zum Beispiel bei zwei direkt aufeinander folgenden L/S-Befehle. In diesem Fall werden keine INT-Befehle ausgeführt, da sie jeweils vor dem L/S-Befehl stehen müssten). Und da die Instruktionen unterschiedlich lang sein können, reichen eventuell bereits 64-Bit aus, um die Zuordnungsstufe zwei Takte lang zu beschäftigen.

Die zweite Technik, AHEAD genannt, erkennt Sprungbefehle so früh wie möglich und verhindert jedes weitere Befehlsholen, bis der Sprungbefehl zugeordnet wird. Da von der Zuordnung bis zur Sprungentscheidung in der Ausführungsstufe noch genügend Zeit vergeht, kann noch ein Befehl geholt werden, so dass es auch bei einem nicht genommenen Sprung zu keiner Verzögerung kommt.

Abbildung 5.5 zeigt in der ersten Zeile das nicht optimierte Fetch-Verhalten für einen

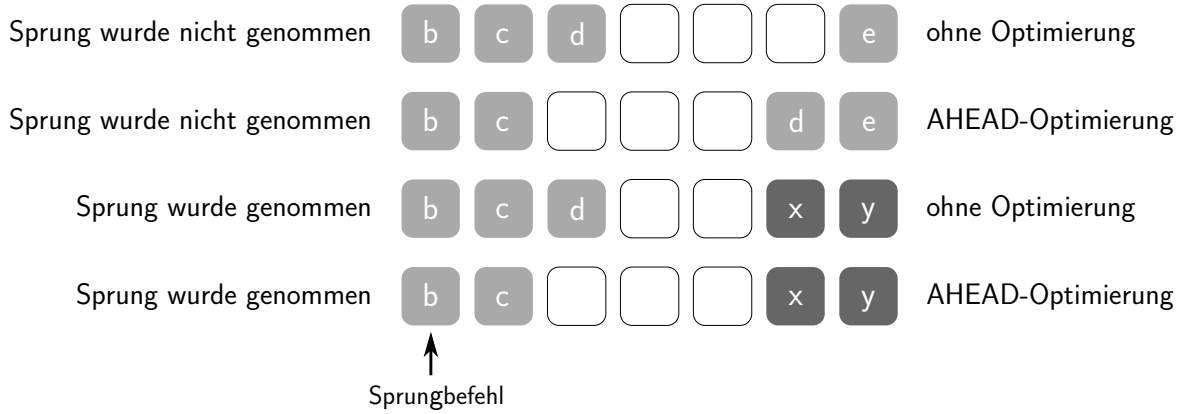

Abb. 5.5: Optimierung der Befehlsbereitstellung mit der AHEAD-Technik

nicht genommenen Sprung. Nach dem Holen des Sprungbefehls b werden noch zwei weitere Wörter geholt, bis das Befehlsfenster voll ist und erst nach der Sprunglatenz wird wieder weiter geholt. Der erste Speicherzugriff nach dem Sprungbefehl c kann mit der AHEAD-Optimierung nicht verhindert werden, da er in Auftrag gegeben wurde, bevor das Speicherwort mit dem Sprungbefehl zurück kam. Der darauf folgende d aber wird so lange verzögert, bis er wirklich gebraucht wird (zweite Zeile in Abbildung 5.5). Dadurch wird zwar noch kein Zugriff eingespart, aber die untergeordneten Hardware-Fäden können früher Befehle holen, was bereits eine Leistungssteigerung ergeben kann. Richtig zum Tragen kommt die AHEAD-Optimierung erst bei genommenen Sprüngen. Obwohl c wiederum nicht verhindert werden kann, wird d komplett eingespart (dritte und vierte Zeile).

Die Evaluierung (Abschnitt 6.3.4) zeigt, dass beide Optimierungen die Ausführungszeit des Hardwarefadens mit der höchsten Priorität nicht beeinflussen und die Anzahl der Speicherzugriffe reduziert wird. Überraschend ist es, dass sich bei der Kombination der beiden Verfahren die Einsparungen nicht nur völlig unabhängig voneinander ergänzen, sondern dass sogar weitere Zugriffe vermieden werden. Der Grund für diesen positiven Effekt wird in Abbildung 5.6 dargestellt:

Wie bereits oben bemerkt, lässt sich mit der AHEAD-Optimierung allein die direkt auf den Sprung folgende Instruktion c nicht verhindern. Falls jedoch durch die ENOUGH-Analyse die Zugriffe für die folgenden Instruktionen c und d nach hinten geschoben werden, kann zu diesem späteren Zeitpunkt erkannt werden, dass c und d aufgrund eines vorangehenden Sprungbefehls überflüssig sind.

5.4.3 Zuordnung

Durch den Verzicht auf die LOOP-Subpipeline gibt es beim CarCore-Prozessor nur noch acht Kombinationen von Befehlen (Abbildung 5.7). Trotzdem ist die Zerlegung des Befehlsstroms in die einzelnen Instruktionen weiterhin ein komplizierter Vorgang, der durch

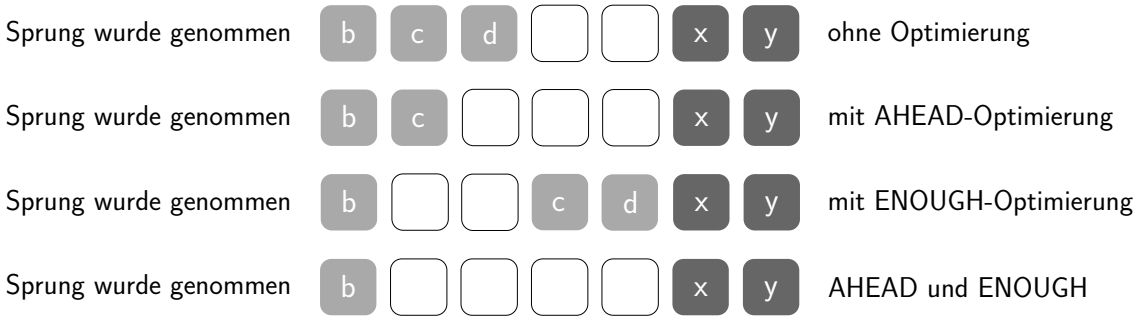

Abb. 5.6: Positive Verstärkung der beiden Optimierungen

ein separates Hardware-Modul erfolgt. Dieser Vordekodierer existiert einmal für jeden Hardwarefaden. Er analysiert die ersten Bytes des jeweiligen Befehlsfensters und zerlegt sie in je einen 31-Bit-Befehl. Das jeweils niederwertigste Bit dient nur zur Unterscheidung der Subpipeline und wird durch die Vordekodierung überflüssig und 16-Bit-Befehle werden mit Nullen im höherwertigen Bereich aufgefüllt.

Der Zuordnungsstufe stehen somit jeweils zwei Instruktionen jedes Hardware-Fadens zur Verfügung. Es wird mit dem INT-Befehl des Hardware-Fadens mit der höchsten Priorität begonnen. Falls er eine Instruktion enthält, wird diese an die Dekodierstufe der INT-Subpipeline weitergegeben. Falls nicht, wird der INT-Befehl des Hardware-Fadens mit der nächsthöheren Priorität untersucht, und so weiter. Falls kein INT-Befehl zur Verfügung steht, wird eine leere Instruktion ohne Funktion an die INT-Subpipeline weitergegeben.

Erst nachdem klar ist, von welchem Hardware-Faden der INT-Befehl zugeordnet wird, kann mit der Suche des L/S-Befehls begonnen werden, denn ein L/S-Befehl kann erst zugeordnet werden, wenn alle vorangegangenen Instruktionen des Hardware-Fadens bereits zugeordnet wurden, das heißt, der INT-Befehl des Hardware-Fadens muss entweder leer sein oder in diesem Takt zugeordnet worden sein.

Es gibt noch weitere Bedingungen, die vor der Zuordnung beachtet werden müssen. Nach einem Sprungbefehl in der INT-Subpipeline, darf der nächste Befehl nicht zugeordnet werden, selbst wenn es ein L/S-Befehl wäre. Zusätzlich darf nach Sprungebefehlen (egal von welcher Subpipeline) drei Takte lang kein Befehl des gleichen Hardware-Fadens zugeordnet werden, da erst dann die Adresse der nächsten Instruktion feststeht (bei nicht-genommenen Sprüngen stehen dann die nächsten Befehle schon im Befehlsfenster, bei genommenen Sprüngen ist das Befehlsfenster leer und muss erst mit Instruktionen von der neuen Adresse gefüllt werden).

Auch nach Speicherzugriffsbefehlen (die ausschließlich der L/S-Subpipeline angehören) dürfen drei Takte lang keine weiteren Befehle zugeordnet werden, da es solange dauert, bis der Befehl den Speicher-Controller erreicht. Ab diesem Zeitpunkt kann die Zuordnung weiter unterbunden werden, falls der Speicher-Controller den Speicherzugriff noch nicht durchführen kann und er dies der Zuordnungsstufe durch das Setzen eines Signals mitteilt. Erst wenn das Signal des jeweiligen Hardware-Fadens wieder gelöscht wurde,

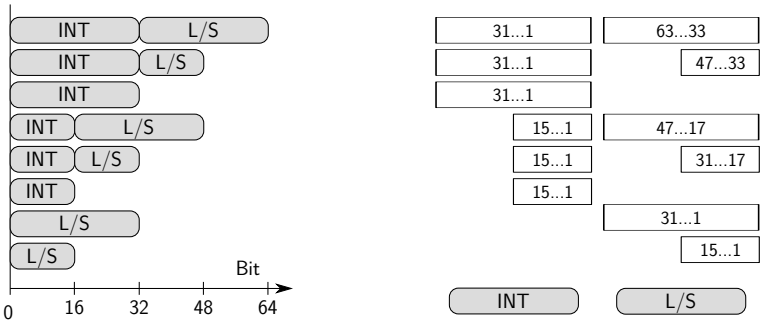

Abb. 5.7: Die acht verbleibenden Kombinationen von CarCore-Befehlen (links) und ihre Zerlegung durch die Vordekodierung (rechts).

kann mit der Zuordnung von Instruktionen fortgefahren werden.

Zu guter Letzt ist die Zuordnungsstufe auch für die Verarbeitung von Befehlen mit mehr als einem Takt Dauer verantwortlich, die durch Mikrocodesequenzen realisiert werden. Nur L/S-Befehle können mehr als einen Takt dauern, weshalb die Mikrocodeverarbeitung ausschließlich in dieser Subpipeline stattfindet. Sobald die Zuordnungsstufe einen Mikrocodebefehl erkennt, verhindert sie die Erhöhung des Programmzählers und zählt stattdessen einen internen Mikrocodezähler hoch, der in Kombination mit dem Befehlswort einen neuen Befehl für die L/S-Subpipeline ergibt. Am Ende der Mikrocodesequenz wird der Programmzähler weiter gezählt und der Mikrocodezähler wieder auf null gesetzt. Auch die zweiphasigen Speicherladebefehle werden wie eine Mikrocodesequenz der Länge zwei behandelt.

5.4.4 Dekodierung und Registerbereitstellung

Ab der Dekodierstufe sind alle Pipelinestufen doppelt vorhanden, für jede Subpipeline eine. Der Aufbau der beiden Dekodierstufen ist jedoch sehr ähnlich. Anhand der niederwertigsten sieben Bits (das niederwertigste Bit mit der Subpipeline-Zugehörigkeit wurde bereits abgeschnitten) wird das Opcode-Format des Befehls ermittelt. Entsprechend diesem Opcode-Format werden bestimmte Register ausgelesen. Bei INT-Instruktionen können das bis zu vier Register sein, bei L/S-Instruktionen bis zu zwei.

Da das Auslesen der Register relativ viel Zeit in Anspruch nimmt und zur Dekodierungszeit addiert werden muss, ist die Dekodierstufe die langsamste Pipelinestufe, die die maximale Ausführungszeit aller Pipelinestufen und damit die maximale Taktfrequenz begrenzt. Aus diesem Grund wird die Adressberechnung für Speicherzugriffe und Sprünge nicht auch noch in dieser Stufe durchgeführt wie beim TriCore, sondern erst eine Stufe später in der Ausführungsstufe.

Parallel zum Auslesen der Register wird anhand des Opcode-Formates ermittelt, an welcher Stelle sich der zweite Opcode befindet. Die Kombination aus primärem und sekundärem Opcode wird dann in einen Funktionscode für die Ausführungsstufe und

einige zusätzliche Signale umgewandelt.

5.4.5 Ausführung und Zurückschreiben

Die beiden Ausführungsstufen unterscheiden sich erheblich, da in der INT-Subpipeline viele verschiedene, teilweise sehr komplexe Operationen (zum Beispiel eine Division mit 8 Bit Breite) mit Operanden in bis zu vier verschiedenen Registern durchgeführt werden müssen. In der L/S-Subpipeline dagegen sind nur einfache Operationen erlaubt, meist wird eine Addition durchgeführt.

Der einzige Teil, der sehr ähnlich ist, ist die Verarbeitung von Sprüngen. Dabei wird jeweils das relative Sprungziel aus dem Befehlswort zum aktuellen Programmzähler addiert. Ob diese Zieladresse jedoch an die Fetch-Stufe weitergegeben wird, wird durch die parallele Prüfung der Bedingung ermittelt. Diese Bedingung kann der Vergleich zweier Werte sein oder das Prüfen eines bestimmten Bits eines Registers. Wenn die Bedingung zutrifft, wird durch ein Signal an die Fetch-Stufe das Befehlsfenster des entsprechenden Hardware-Fadens gelöscht und der Programmzähler auf die neu berechneten Adresse gesetzt.

Das Ergebnis der Ausführungsstufe wird im Normalfall an die letzte Pipelinestufe weitergereicht, die den Wert in das entsprechende Register zurückschreibt. Im Falle der L/S-Subpipeline gibt es außerdem die Speicherzugriffsbefehle, bei denen die berechnete Adresse an die Speicherschnittstelle weitergegeben wird. Falls der Zugriff nicht sofort ausgeführt werden kann, wird die Adresse und die Breite des Zugriffs zwischengespeichert und der Zuordnungsstufe signalisiert, dass sich die Verarbeitung verzögert, damit diese weiterhin die Zuordnung von Instruktionen des betroffenen Hardware-Fadens verhindert.

6 Evaluierungsergebnisse

In diesem Kapitel wird untersucht, wie unterschiedliche Programmfäden sich gegenseitig beeinflussen. Bei Programmfäden mit weichen oder ohne Echtzeitanforderungen macht es Sinn, den Durchsatz zu messen und zu ermitteln wie dieser gesteigert werden kann. Nicht jedoch bei Programmfäden mit harten Echtzeitanforderungen. Hier spielt es keine Rolle, wie schnell ein Programm im Durchschnitt ausgeführt wird, allein entscheidend ist, dass die WCET nicht überschritten wird. Deshalb macht die Messung der Ausführungszeiten ohne eine zugehörige WCET keinen Sinn.

Eine derartige Laufzeituntersuchung, bei der für verschiedene Programme eine WCET berechnet und mit der tatsächlichen Ausführungszeit auf dem CarCore-Prozessor verglichen wird, wurde von Cédric Landet im Rahmen seiner Dissertation an der Universität von Toulouse [Landet 2009] durchgeführt. Sie ergab, dass beim CarCore die WCET durchschnittlich nur um 4 Prozent überschätzt wird.

Hier hingegen wird untersucht, wie viel Rechenzeit zusätzlich zur Ausführung der harten Echtzeitfäden noch zur Verfügung und wie sie genutzt werden kann.

6.1 Evaluierungsumgebung

Um den in Kapitel 5 beschriebenen CarCore zu untersuchen, wurden mehrere Simulatoren des Prozessors entwickelt, die jeweils unterschiedliche Aspekte beleuchten. Getestet wurden sie mit unterschiedlichen Benchmark-Programmen für echtzeitfähige eingebettete Systeme.

6.1.1 Prozessormodelle

Die folgenden drei Prozessormodelle wurden zur Analyse des CarCore-Prozessors erstellt:

(i) Ein leicht veränderbares SystemC-Modell, um eine umfangreiche Evaluierung zu ermöglichen.

(ii) Ein kaum veränderbarer FPGA-Prototyp, um den Hardware-Bedarf zu ermitteln.

(iii) Ein Trace-Simulator zum Vergleich mit der original TriCore-Archtektur.

Das umfangreichste Modell wurde in der Hardware-Beschreibungssprache SystemC [IEEE 2005] entwickelt. Eigentlich wird diese Sprache für eine eher abstrakte Modellierung und Simulation verwendet, in diesem Fall wurde jedoch auf höchste Detailgenauigkeit

Tab. 6.1: Die verwendeten Prozessormodelle

		SystemC	VDHL	Trace
Hardwarefäden		1-21	4	1
Pipelinestufen		5	5	5
Instruktionsspeicher		beliebig	64 KiByte[1]	beliebig
Datenspeicher		beliebig	64 MiByte	beliebig
Fetch-Breite	Z	64 Bit	64 Bit	64 Bit
Befehlsfenster	G	24-128 Byte	24 Byte	24 Byte
Fetch-Latenz	L_{fetch}	0-7	0	0-7
Speicherlatenz	L_{mem}	0-7	2	2-7
Sprung-Latenz	L_{jump}	$2 + L_{fetch}$	2	$2 + L_{fetch}$
Leertakte bei Sprung	L_{branch}	2	2	2
	$L_{F,I}$	$2 + L_{fetch}$	2	$2 + L_{fetch}$
	$L_{I,E}$	1	1	1
	$L_{I,M}$	$\max(2, L_{mem})$	2	$\max(2, L_{mem})$
Maximale Anzahl TCBs		128	32	0
Einlagerung	Δt_{in}	21 Takte	21 Takte	21 Takte
Auslagerung	Δt_{out}	19 Takte	19 Takte	19 Takte
Kontextwechsel	Δt_{CS}	40 Takte	40 Takte	40 Takte

geachtet und jedes einzelne Signal modelliert, wie es auch in einem fertigen integrierten Schaltkreis nötig wäre.

Dadurch sollte die anschließende Übersetzung in VHDL [Reichardt 2009], eine andere Hardware-Beschreibungssprache, erleichtert werden. Diese Übersetzung ist nötig, da SystemC nicht synthetisierbar ist, das heißt, aus dem Quellcode kann kein lauffähiger integrierter Schaltkreis synthetisiert werden, nur Simulationen sind möglich. Trotz der angestrebten Ähnlichkeit zwischen SystemC- und VHDL-Quellcode tauchten bei der Übersetzung viele Probleme auf, die im SystemC-Modell nicht erkannt wurden. Deshalb erscheint es im Nachhinein günstiger, zuerst ein synthetisierbares und damit lauffähiges VHDL-Modell zu erstellen und erst anschließend ein abstrakteres SystemC-Modell abzuleiten, in dem die Rahmenbedingungen der Architektur leichter variiert werden können als im VHDL-Modell.

Da dieser Sachverhalt vor Beginn der Arbeiten jedoch nicht bekannt war, wurde zuerst das detaillierte und gut parametrisierbare SystemC-Modell erstellt und anschließend ein VHDL-Modell abgeleitet, mit dem schließlich ein FPGA programmiert wurde. In einem FPGA können beliebige logische Schaltungen nachgebildet werden, ohne jedes Mal einen neuen Silizium-Chip herstellen zu müssen. Die jederzeitige Neuprogrammierung wird dadurch erkauft, dass die Schaltung nur mit einer deutlich geringeren Taktfrequenz betrieben werden kann.

[1] Gemäß einer Empfehlung des für das SI-Einheitensystem zuständigen internationalen Büros für Maß und Gewicht werden für Datenmengen die binären Präfixe der DIN EN 80000-13:2009-01 verwendet

Das letzte Modell schließlich ist ein Trace-gesteuerter Simulator, der kompilierte Binärdatei mit Maschinenbefehlen nicht direkt ausführen kann. Stattdessen ist er auf eine Aufzeichnung des Programmablaufs, das sogenannte *Trace* angewiesen, das in diesem Fall mit Hilfe des SystemC-Modells erzeugt wurde. Dieses Modell unterstützt nur die einfädige Programmausführung und dient dem Vergleich mit dem original TriCore und zur Untersuchung tiefgreifender Veränderungen der Pipeline, die in den anderen beiden Modellen nur mit äußerst hohem Zeitaufwand realisiert hätten werden können.

Alle drei Modelle haben bei den gleichen Parametern ein identisches Zeitverhalten, jedoch können jeweils nur bestimmte Parameter verändert werden. Der FPGA-Prototyp ist am meisten eingeschränkt, das SystemC-Modell hat am meisten Möglichkeiten und mit dem Trace-Simulator können spezielle Änderungen der Pipeline im einfädigen Fall untersucht werden. Tabelle 6.1 fasst die veränderbaren Parameter der einzelnen Modelle zusammen.

6.1.2 Benchmark-Programme

Zur Leistungsbewertung (engl. *benchmarking*) des Prozessors wurden 23 Testprogramme ausgewählt, die aus zwei bekannten Benchmark-Sammlungen stammen. Sie sind in Tabelle 6.2 aufgelistet. Bei den mit einem großen Buchstaben abgekürzten Programmen handelt es sich um 13 Programme des AutoBench 1.1 [EEMBC 2000], einer Benchmark-Sammlung des *Embedded Microprocessor Benchmark Consortium (EEMBC)*[2] um die Prozessorleistung im Hinblick auf Anwendungen der Industrie, insbesondere der Automobilindustrie zu bewerten. Drei Programme dieser Sammlung (`aifftr`, `aiifft`, `matrix` wurden nicht verwendet, da sie im Vergleich zu den anderen Programmen extrem lange Laufzeiten haben.

Sie wurden ersetzt durch `fft1` und `mm`, die ebenfalls eine schnelle Fourier-Transformation und eine Matrixmultiplikation durchführen, aber mit Laufzeiten, die besser zu den anderen Benchmark-Programmen passen. Auf eine inverse schnelle Fouriert-Transformation wurde ganz verzichtet, da `aifftr` und `aiifft` sich nur dadurch unterscheiden, dass bei der inversen Tranformation (`aiifft`) an einer Stelle ein Zwischenergebnis negiert wird.

`fft1` und `mm` gehören zu den sieben, mit einem kleinen Buchstaben abgekürzten Programmen, die aus der WCET-Benchmark-Sammlung des *Mälardalen Real-Time Research Center*[3] [Gustafsson 2010] stammen. Diese Programme, die vormals auch als C-Lab-Benchmarks bekannt waren, werden häufig von Forschergruppen benützt, die sich mit statischen Laufzeitanalysen und der Berechnung einer WCET beschäftigen.

Kompiliert wurden die Benchmark-Programme mit dem *HighTec GNU C/C++ TriCore Compiler 3.3.7.9* der *HighTec EDV-Systeme GmbH*[4] mit den in Tabelle 6.3 aufgeführten Compilerschaltern. Besonders wichtig ist die Ausrichtung von Funktionen auf volle

[2] http://www.eembc.org/, abgerufen am 1. August 2011
[3] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html, abgerufen am 1. August 2011
[4] http://www.hightec-rt.com/, abgerufen am 1. August 2011

Tab. 6.2: Übersicht der Benchmark-Programme. Neben der durchschnittlichen IPC ist die Verteilung der Instruktionen auf die Pipelines (nur INT, nur L/S oder beide) und der prozentuale Anteil von lesenden (Le) und schreibenden (Sc) Speicherzugriffen, Sprungbefehlen (Sp) und Mikrocodesequenzen (MC) angegeben. Die letzte Spalte gibt die Dauer einer Iteration an.

			Subpipelines			Befehlsart				
Abk.	Name	IPC	INT [%]	L/S [%]	beide [%]	Le [%]	Sc [%]	Sp [%]	MC [%]	Dauer [Takte]
a	adpcm	0,247	29,2	56,0	14,8	21,7	9,2	9,0	3,4	14748
c	crc	0,524	55,0	34,8	10,2	12,0	2,5	12,7	0,2	3267
f	fft1	0,296	65,3	26,4	8,4	4,5	5,8	17,1	3,0	5190
l	lms	0,231	56,6	35,4	8,0	7,5	6,6	18,3	4,6	2364001
m	mm	0,456	1,3	65,4	33,3	23,8	8,3	8,3	0,0	221363
n	cnt	0,273	0,0	83,7	16,3	46,4	16,2	16,3	0,0	189347
s	srt	0,443	40,2	39,5	20,3	15,2	8,1	23,9	0,0	35437
A	a2time	0,329	72,2	22,1	5,7	6,9	3,9	15,9	2,3	11839
B	basefp	0,211	45,8	45,7	8,4	9,4	5,7	14,6	5,5	98497
C	canrdr	0,264	11,1	64,5	24,4	14,4	17,4	18,1	2,7	639
D	idctrn	0,397	11,5	69,8	18,8	15,2	9,1	9,6	1,1	127321
F	aifirf	0,338	11,0	64,2	24,8	27,3	9,1	16,7	0,6	7309
H	cacheb	0,142	9,9	71,8	18,4	25,1	13,5	4,9	9,0	1570
I	iirflt	0,374	72,6	20,1	7,3	6,3	3,3	11,4	1,9	15308
M	bitmnp	0,312	49,8	35,1	15,1	10,6	10,5	25,3	1,7	111679
O	tblook	0,235	54,5	37,4	8,1	9,0	5,6	16,9	4,5	61656
P	pntrch	0,345	37,7	56,0	6,3	28,2	3,8	27,4	0,1	78312
R	rspeed	0,205	33,9	47,1	19,0	20,3	11,2	6,3	5,2	935
T	ttsprk	0,301	52,3	35,3	12,4	17,7	2,0	20,2	2,1	12809
U	puwmod	0,147	25,6	54,8	19,6	21,0	11,2	6,0	8,9	2139

8-Byte Adressen durch den Compilerschalter `-align-functions=8`. Andernfalls hängt der Compiler (genauer gesagt der Linker) den Code der einzelnen Programmfäden eines Tasksets einfach hintereinander, ohne es am Raster, das durch das Holen der Befehlsworte von vollen 8-Byte-Adressen vorgegeben ist, zu beachten.

Dadurch können, je nachdem welcher Code für die anderen Programmfäden generiert wurde, die Instruktionen innerhalb des Rasters verschoben sein. Ohne diese Option kann die gleiche Instruktion in einer Version mitten innerhalb eines 8-Byte-Wortes liegen und in einer anderen Version mit anderen Programmfäden liegt sie so, dass die zweite Hälfte der Instruktion im nächsten 8-Byte-Wort liegt und zwei Speicherzugriffe nötig sind.

Durch den Compilerschalter `-align-functions=8` wird nur der Beginn jeder Funktion ausgerichtet, innerhalb der Funktion können die Instruktionen und insbesondere Sprungziele weiterhin ungünstig liegen. Der zusätzliche Code wird dadurch auf einige Bytes pro Funktion begrenzt.

Tab. 6.3: Verwendete Compiler-Schalter und ihre Bedeutung

Schalter	Bedeutung
`-mtc13`	Maschinencode für den TriCore 1.3
`-O3`	Maximale Codeoptimierung
`-fno-expensive-optimizations`	Vermeidet einen Fehler im Compiler bei die Kompilierung nicht beendet wird
`-align-functions=8`	Funktionen an durch 8 teilbaren Adressen ausrichten

Die AutoBench-Programme bestehen jeweils aus drei Teilen: einer Initialisierung, einem Hauptteil, der für eine festgelegte Anzahl von Iterationen wiederholt wird und einem Abschluss. Alle Messungen beziehen sich jeweils nur auf den Hauptteil. Das die Programme in ihrer ursprünglichen Form globale Variablen benutzten und dadurch nicht threadsicher waren, mussten sie leicht modifiziert werden. Ein Programm gilt als threadsicher, wenn selbst bei gleichzeitiger Ausführung mehrerer Instanzen des Programms, jede Instanz korrekt terminiert. Die Mälardalen-Benchmark-Programme entsprechen nicht dem beschriebenen dreigeteilten Aufbau, sie wurden dementsprechend angepasst.

In Tabelle 6.2 sind neben dem Benchmark-Namen und seiner hier verwendeten Abkürzung die durchschnittliche IPC und die prozentuale Verteilung der Instruktionen auf die Pipelines (nur INT, nur L/S oder beide) aufgeführt. Zur weiteren Charakterisierung der Programme werden auch die prozentualen Anteile von lesenden (Le) und schreibenden (Sc) Speicherzugriffen, Sprungbefehlen (Sp) und Mikrocodesequenzen (MC) an der Gesamtzahl der ausgeführten Instruktionen angegeben. In der letzte Spalte ist die durchschnittliche Dauer einer Iteration angegeben (gemittelt über die ersten 1000 Iterationen).

6.2 Einfädige Programmausführung

Um die Leistungsfähigkeit der CarCore-Implementierung zu beurteilen, wird in diesem Abschnitt die einfädige Ausführung mit dem original TriCore verglichen, soweit dies aus veröffentlichten Daten über das Timing des TriCore-Prozessors möglich ist. Weiterhin werden die Einflüsse der Entwurfsentscheidungen aus Abschnitt 5.3 bewertet.

Gemessen wird die Änderung der Ausführungszeit, die eine bestimmte Modifikation bewirkt. Hierfür wird für jedes der 20 Benchmark-Programme die Ausführungszeit des Hauptteils mit Hilfe des Trace-Simulators gemessen und daraus die absolute IPC berechnet. Da der FPGA-Prototyp als Basis dient, teilt man anschließend diese absolute IPC durch die IPC, die das gleiche Programm bei Ausführung auf dem FPGA-Modell hätte (siehe Tabelle 6.2) und erhält dadurch die relative IPC, die in Prozent angegeben wird. Um den Einfluss einer Architekturänderung auf die Ausführungszeit zu bewerten, wird der Durchschnitt der relativen IPCs der 20 Benchmark-Programme herangezogen.

Tab. 6.4: Einfluss der Sprungbefehlsverarbeitung auf die Ausführungsgeschwindigkeit

Bezeichnung	Vorher-sage	Address-berechnung	LOOP-Subpipeline	⌀ IPC [%]	Min. [%]	Max. [%]
	keine	Execute	nein	100,00		
ST	statisch	Execute	nein	103,39	100,00	110,65
LO	keine	Execute	ja	104,50	100,00	126,27
ID	keine	Decode	nein	105,19	100,70	114,10
ST+LO	statisch	Execute	ja	106,92	100,00	128,85
ID+LO	keine	Decode	nein	108,55	100,70	132,25
ST+ID	statisch	Decode	nein	109,21	100,70	128,18
ST+ID+LO	statisch	Decode	ja	111,34	100,70	138,88
PE	perfekt	irrelevant	nein	111,36	101,42	132,84
PE+LO	perfekt	irrelevant	ja	114,01	101,42	145,22

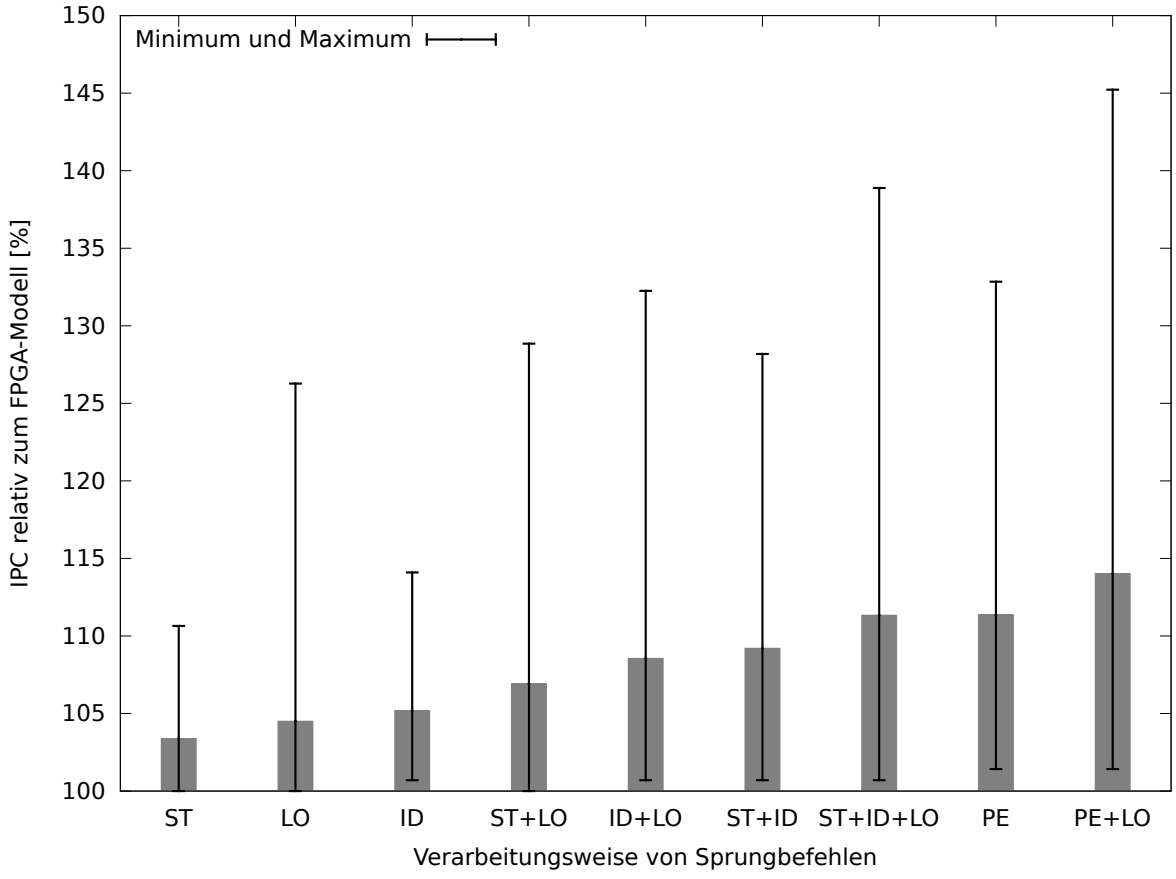

Abb. 6.1: Einfluss der Sprungbefehlsverarbeitung auf die Ausführungsgeschwindigkeit

Tab. 6.5: TriCore-Timing-Modell mit Quellen

	Eigenschaft	Quelle
Befehlsbereitstellung	64 Bit pro Takt an 16-Bit-Grenzen ausgerichtet	[Infineon 2003, S.39]
Zuordnung	3 Befehle in der festen Reihenfolge: INT, L/S, LOOP	[Infineon 2003, S.31f]
Sprungvorhersage	16 Bit: *taken* 32 Bit: *backward taken, forward not*	[Infineon 2004, S.10]
Sprunglatenzen	0 *not taken* korrekt vorhergesagt 1 *taken* korrekt vorhergesagt 2 falsche Vorhersage	[Infineon 2004, S.10]
Dual-Pipeline-Instruktionen	`addsc.a`, `addsc.at`, `mov.a`, `mov.d`, `eq.a`, `eqz.a`, `ge.a`, `lt.a`, `nez.a`, alle bedingten Sprünge der INT-Subpipeline	[Infineon 2003, S.32]
`dvstep`	Ausführungszeit: 5 Takte	[Infineon 2003, S.38]
`madd`, `mul`, `msub`	Ausführungszeit: 2 Takte	[Infineon 2003, S.38]
`call`,`ret`	jeweils 4 Speicherzugriffe à 128 Bit	[Infineon 2003, S.23]
`ldmst`, `st.t`, `swap.w`	jeweils 2 Speicherzugriffe à 32 Bit	keine

6.2.1 Einfluss von Verzögerungen bei Sprungbefehlen

Drei der Veränderungen am CarCore gegenüber dem TriCore betreffen die Verarbeitung von Sprungbefehlen. Ihr Einfluss auf die Ausführungsgeschwindigkeit ist in Tabelle 6.4 aufgeführt. Es wurde die relative IPC im Vergleich zur IPC bei Ausführung auf dem FPGA-Prototyp berechnet. Die Tabelle enthält den Mittelwert über alle 20 Benchmark-Programme, sowie den kleinsten als auch den größten gemessenen Wert der IPC bei einem einzelnen Programm. Abbildung 6.1 veranschaulicht die gleichen Daten grafisch.

Folgende drei Besonderheiten des TriCores wirken sich positiv auf seine Ausführungsgeschwindigkeit im Vergleich zum CarCore aus:

ST Eine statische Sprungvorhersage (statt keiner beim TriCore).

LO Eine zusätzliche Subpipeline zur Verarbeitungen von speziellen Schleifenbefehlen.

ID Die vorzeitige Berechnung der Zieladresse eines Sprungs in der Dekodierstufe.

Alle drei Faktoren wirken sich in ungefähr gleichem Maße auf die Ausführungsgeschwindigkeit aus und durch die Kombination der Faktoren lässt sich die IPC weiter steigern. Es fällt auf, dass durch die Kombination aller drei Faktoren (ST+ID+LO) die Ausführungsgeschwindigkeit einer perfekten Sprungvorhersage (PE) erreicht werden kann, allerdings ohne eine zusätzliche separate Subpipeline für spezielle Schleifen, die die Ausführungszeit noch einmal erhöhen würde (PE+LO).

6.2.2 Weitere Unterschiede zum TriCore

Um die Ausführungszeiten des CarCore mit dem TriCore zu vergleichen, wurde der Trace-Simulator um ein Modell des TriCore 1.3 erweitert. Soweit möglich, wurden die dafür nötigen Informationen den TriCore-Handbüchern [Infineon 2003; Infineon 2004; Infineon 2008a; Infineon 2008b] und Zeitschriftenartikel [Schultes 1999] entnommen. Tabelle 6.5 listet die wesentlichen Informationen auf. Das ernüchternde Ergebnis zeigt Abbildung 6.2: der TriCore ist mehr als doppelt so schnell.

Um den Grund für die deutliche Überlegenheit der TriCore-Prozessors zu ermitteln, wurden deshalb einzelne Architekturmerkmale, die dem CarCore fehlen, im Trace-Simulator nachgebildet und die entsprechenden Laufzeiten verglichen. Als Metrik dient wiederum die durchschnittliche relative IPC im Verhältnis zum FPGA-Prototyp. Ergänzend zum Diagramm enthält Tabelle 6.6 die exakten Durchschnitts-, Minimal- und Maximalwerte.

Nur zwei der in Abschnitt 5.3 aufgeführten Unterschiede zwischen dem TriCore und dem CarCore wirken sich positiv auf die Ausführungsgeschwindigkeit des CarCores aus. Werden sie entfernt (um dem TriCore möglichst nahe zu kommen), so liegt die relative IPC unter 100%. Doch der Einfluss ist sehr gering, da die beiden Modifikationen nur wenige, sehr spezielle Instruktionen betreffen: Divisionen werden in einem statt fünf Takten durchgeführt (Kürzel DIV in Tabelle und Diagramm) und Instruktionen, die beim TriCore die INT- und die L/S-Subpipeline gleichzeitig belegen, belegen beim CarCore nur die L/S-Subpipeline (Kürzel DP).

Bei der Divisionsbeschleunigung liegt das daran, dass die Division nur selten benutzt wird, nur drei der Programme enthalten überhaupt einen Divisionsbefehl. Überdies ist die Beschleunigung bei `A-a2time` und `R-rspeed` mit unter zwei Prozent vergleichsweise gering, nur beim IIR-Filter `I-iirflt` gibt es eine erhebliche Beschleunigung von fast 13%.

Von der Möglichkeit, einen INT-Befehl gleichzeitig mit einem Dual-Pipeline-Befehl auszuführen, profitieren zwar alle Programme, die Steigerung der IPC ist jedoch äußerst gering und beträgt bei der überwiegenden Zahl der Programme weniger als 1,3%. Nur bei `F-aifirf` und `s-srt` beträgt die Beschleunigung 3% und bei der ganzzahligen Matrixmultiplikation `mm` werden sogar 7% erreicht.

Die Auswirkungen von Architekturmerkmalen, die nur im TriCore vorhanden sind und dort die Ausführung beschleunigen, ist wesentlich höher. Könnte man auf den zusätzlichen Takt, den jeder Ladebefehl durch die zweite Phase des Split Phase Loads benötigt, verzichten, so würde sich die IPC um 4% erhöhen (Kürzel KS). In der gleichen Größenordnung bewegen sich die drei Verfahren zur Beschleunigung von Sprungbefehlen, die bereits im vorhergehenden Abschnitt diskutiert wurden (Kürzel SV). Zusammengenommen könnten sie die IPC um 11% erhöhen.

Sehr viel größeren Einfluss hat die hochoptimierte Ausführung von Kontextwechseln bei Unterprogrammaufrufen im TriCore (Kürzel UA). Durch die Verwendung von zwei Registerbänken und einem speziellen Datenbus zum Sichern beziehungsweise Wiederher-

Tab. 6.6: Auswirkung von Architekturveränderungen auf die Ausführungsgeschwindigkeit

Kürzel	Beschreibung	⌀ IPC [%]	Min. [%]	Max. [%]
FPGA	Basismodell	100,00	100,00	100,00
DIV	Langsame Division	99,34	87,32	100,00
DP	Dual-Pipeline-Befehle	99,01	92,94	99,96
KS	Kein Split Phase Load	104,40	101,29	112,18
SV	Sprungbefehlsoptimierungen (ST+ID+LO)	111,34	100,70	138,88
UA	Schnelle Unterprogrammaufrufe	176,50	100,04	337,53
SV+UA		200,35	111,78	354,27
SV+UA+KS		216,45	127,21	393,30
TriCore	Modellierter TriCore gemäß Tabelle 6.5	221,35	115,96	441,13

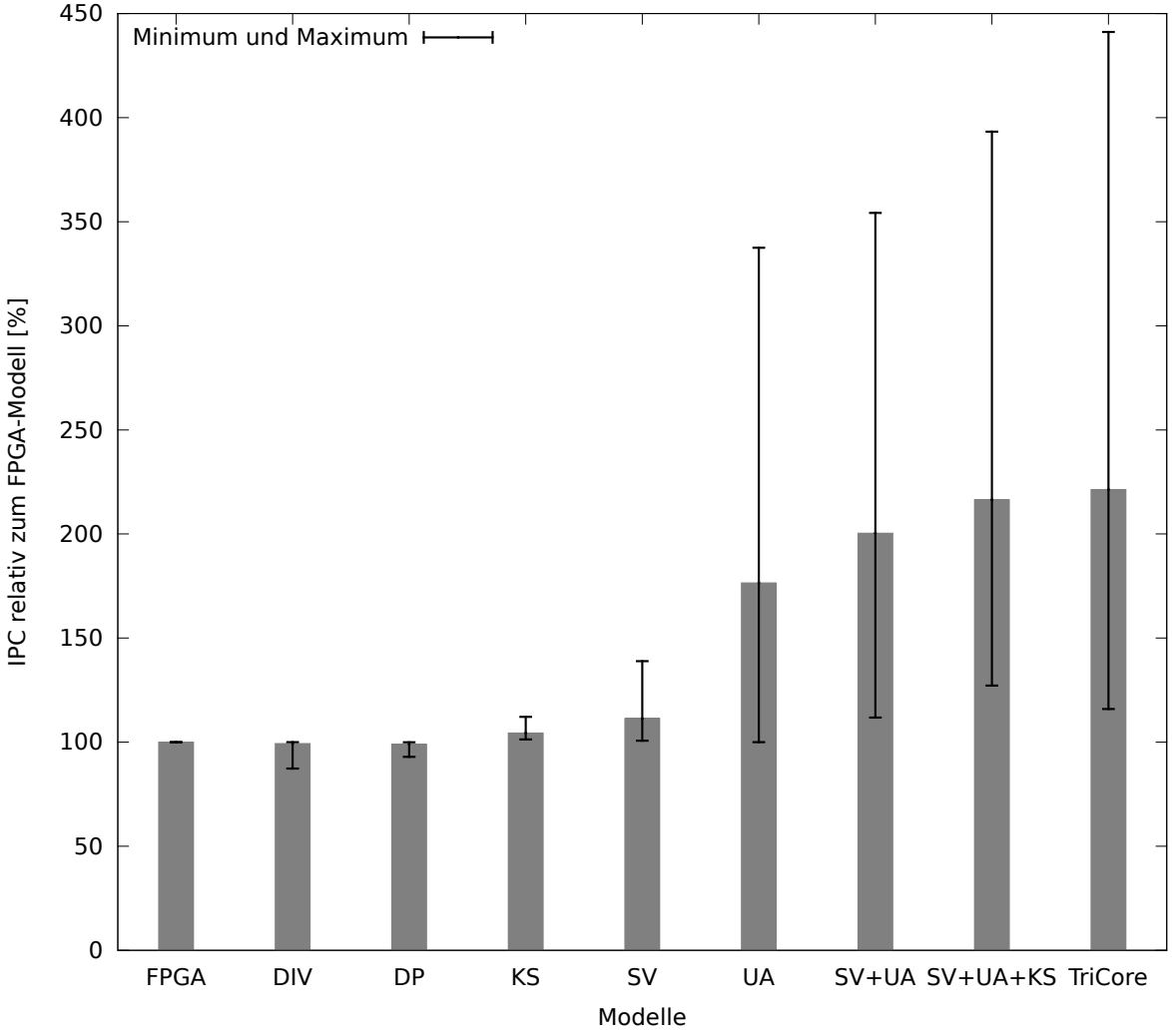

Abb. 6.2: Auswirkung von Veränderungen am CarCore auf die IPC

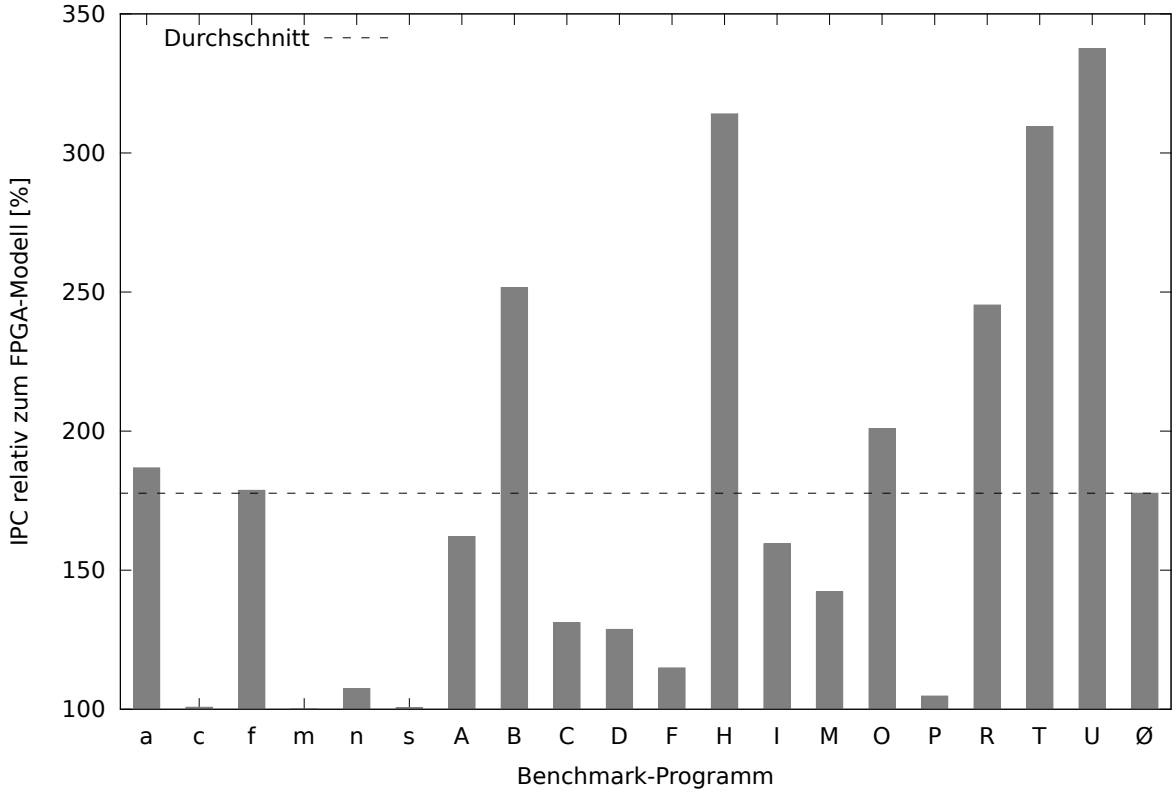

Abb. 6.3: Einfluss der Optimierung der Unterprogrammaufrufe auf die IPC

stellen der Register wird die Dauer von Funktionsaufrufen erheblich reduziert. Da der Einfluss sehr stark von der Anzahl der Unterprogrammaufrufe und damit vom einzelnen Programmcode abhängt, sind in Abbildung 6.3 die IPC-Steigerungen für jedes einzelne Benchmark-Programm aufgeführt. Die Beschleunigung ist erheblich, im Durchschnitt über 75%.

Stattet man den CarCore sowohl mit einem beschleunigten Unterprogrammaufruf als auch mit den im vorherigen Abschnitt erwähnten Sprungverarbeitungsoptimierungen aus (Kürzel SV+UA), so wird die relative IPC weiter erhöht. Interessanterweise auf einen höheren Wert, als theoretisch durch einfache Multiplikation der beiden Faktoren ($111,34\% \times 176,50\% = 196,52\%$) zu erwarten wäre. Es gibt demnach positive Synergieeffekte, ganz im Unterschied zur Kombination der einzelnen Sprungoptimierungen untereinander, wo die Gesamtbeschleunigung bei Kombinationen etwas zurückgeht.

Doch selbst wenn man auch noch die Split Phase Load Verzögerung herausrechnet (Kürzel SV+UA+KS), so erreicht man mit 216% noch immer nicht den TriCore-Wert von 221%. Zurückführen lässt sich dies auf die unterschiedlichen Mechanismen zum Bereitstellen von Befehlen. Beim TriCore können von jeder durch zwei teilbaren Adresse auf einmal 64 Bit geholt werden, wohingegen beim CarCore die Adresse durch acht teilbar sein muss und daher bei einer ungünstigen Anordnung der Befehle zwei Speicherzugriffe für eine Instruktion nötig sind.

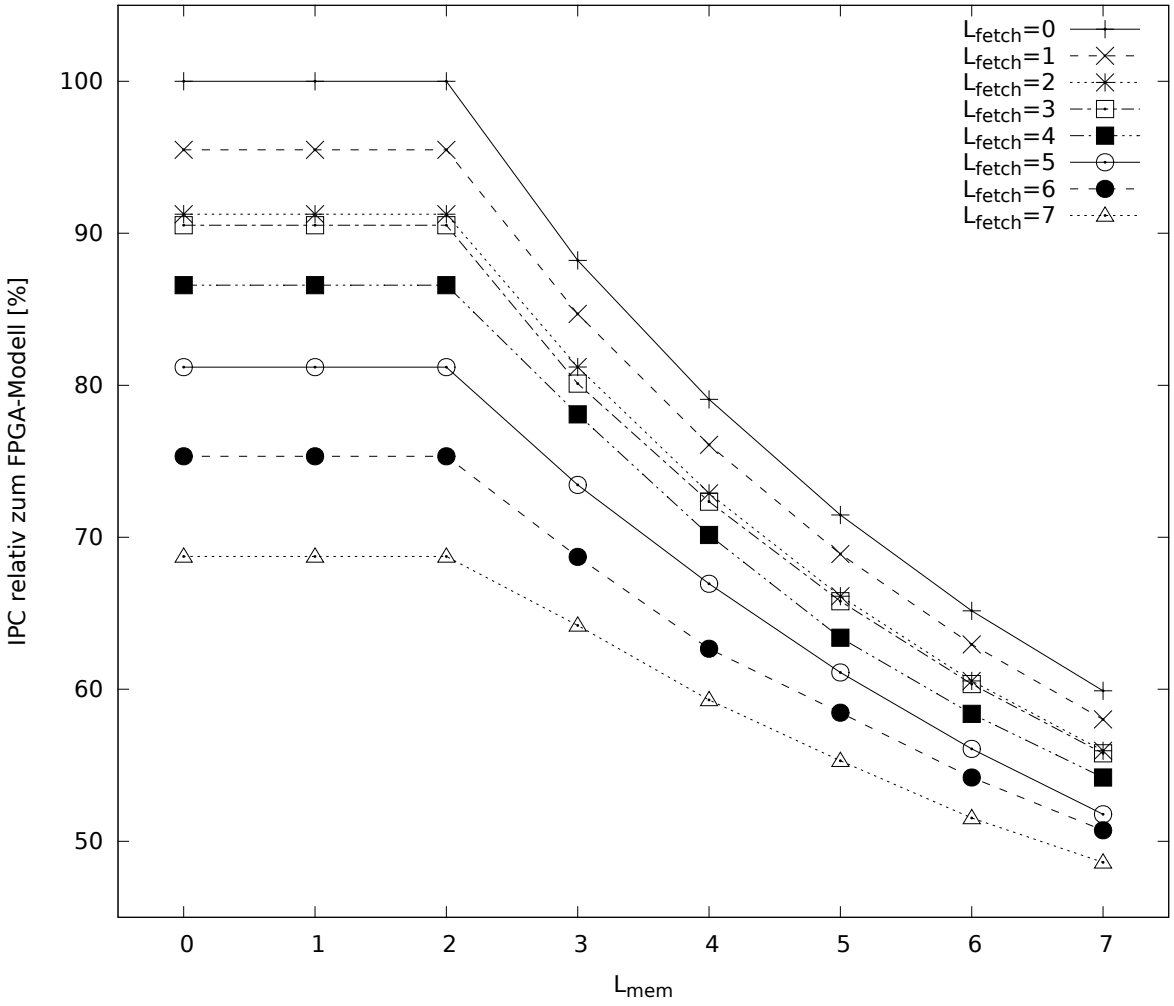

Abb. 6.4: Einfluss der Speicherlatenzen auf die Ausführungsgeschwindigkeit

6.2.3 Einfluss von Speicherlatenzen

Wie Abbildung 6.4 zeigt, hängt die Ausführungsgeschwindigkeit, gemessen als IPC relativ zum FPGA-Prototyp ($L_{fetch} = 0$, $L_{mem} = 2$), wesentlich stärker von der Latenz des Datenspeichers ab als von der Latenz des Befehlsspeichers. Dies liegt daran, dass das Holen der Befehle durch die Befehlsfenster gepuffert wird, wodurch die Speicherzugriffe teilweise von der Pipeline entkoppelt werden, die Zugriffe passieren teilweise im Hintergrund. Die Zugriffe auf den Datenspeicher hingegen blockieren stets die Ausführung des entsprechenden Programmfadens.

Durch die Aufsplittung des lesenden Speicherzugriffs in zwei Phasen gibt es eine Mindestverzögerung durch Speicherzugriffe, leicht erkennbar durch die waagrechten Plateaus bei den Datenspeicherlatenzen von 0 bis 2 in Abbildung 6.4. Der Grund für diese Mindestverzögerung wird in Abschnitt 3.4.2 genauer erläutert.

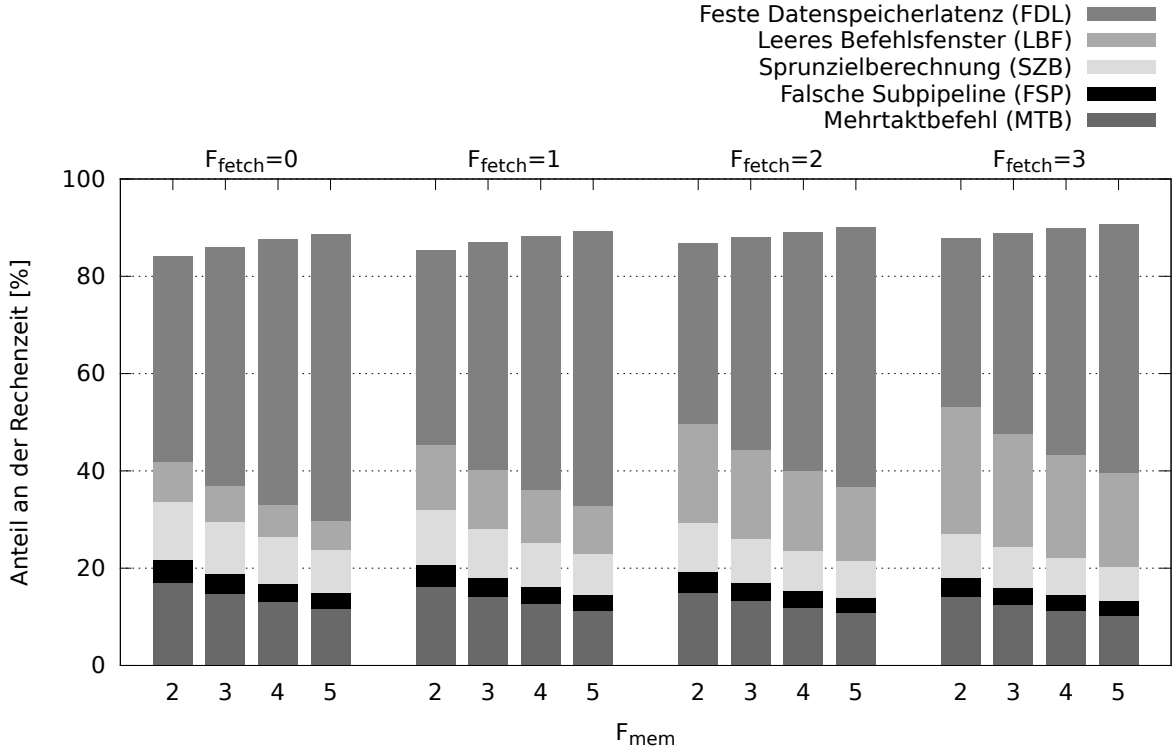

Abb. 6.5: Gründe für die Nichtzuordnung von Befehlen bei einfädiger Ausführung

6.2.4 Gründe für den Leerlauf von Subpipelines

Es gibt sechs Gründe, warum in einem bestimmten Takt einer Subpipeline keine Instruktion eines bestimmten Hardware-Fadens zugeordnet werden kann:

Feste Datenspeicherlatenz (FDL) Mindestverzögerung bei einem Speicherzugriff, da der am Ende der Pipeline befindliche Speichercontroller jeden Speicherzugriff bestätigen muss (drei Takte).

Dynamische Datenspeicherlatenz (DDL) Verzögerung, weil der Speicherzugriff eines anderen Hardware-Faden erst beendet werden muss, bevor ein neuer gestartet werden kann. Tritt bei einfädiger Ausführung nicht auf.

Leeres Befehlsfenster (LBF) Das Befehlsfenster ist leer, da noch keine Instruktionen bereitgestellt wurden.

Sprungzielberechnung (SZB) Bei einem Sprungbefehl muss zwei Takte auf die Berechnung der Zieladresse gewartet werden.

Falsche Subpipeline (FSP) Der nächste auszuführende Befehl des Hardware-Fadens kann in der betreffenden Subpipeline nicht ausgeführt werden.

Mehrtaktbefehl (MTB) Es wird gerade eine mehrtaktige Instruktion ausgeführt, die die Subpipeline für mehrere Takte belegt. Dadurch ist die Subpipeline zwar belegt, es wird aber keine neue Instruktion ausgeführt.

Abbildung 6.5 zeigt den prozentualen Anteil der Gründe an der Gesamtlaufzeit. Der zu 100 Prozent fehlende Teil der Balken entspricht dem Anteil der Zeit, in dem eine Instruktion in einer bestimmten Subpipeline für genau einen Takt ausgeführt wird. Da der CarCore zwei Subpipelines hat, entspricht diese Zahl mit zwei multipliziert der IPC. Abgesehen von diesem Bruchteil wird nur noch bei Mehrtaktbefehlen wirklich ein Befehl ausgeführt, in allen anderen Fällen bleiben die Subpipelines leer. Dies zeigt, wie gering die Auslastung der Ausführungseinheiten ist und warum sich die gleichzeitige Nutzung durch mehrere Programmfäden bei der simultan mehrfädigen Ausführung lohnt.

Weiterhin zeigt das Diagramm die starke Abhängigkeit von den Latenzen bei Speicherzugriffen, wobei die Abhängigkeit von Zugriffen auf den Datenspeicher (FDL) noch stärker ist als die Abhängigkeit von Zugriffen auf den Befehlsspeicher (LBF). Wie oben bereits erwähnt, kann die Datenspeicherlatenz aber aufgrund des Split Phase Load nicht unter zwei reduziert werden.

6.3 Mehrfädige Programmausführung

In diesem Abschnitt wird der Einfluss von Hardware-Fäden auf die Ausführungszeit der anderen Hardware-Fäden untersucht. Diese Untersuchung ist unabhängig vom bestimmten Scheduling-Algorithmen, jeder Hardware-Faden wird mit einer festen, absteigenden Priorität versehen.

Aus den 20 Benchmark-Programmen werden 400 verschiedene Tasksets erzeugt, indem jedes Programm mit jedem anderem und sich selbst kombiniert wird und die restlichen Hardware-Fäden durch eine zufällige Auswahl aufgefüllt werden. Die Tasksets werden mit dem System-C-Simulator ausgeführt, wobei zunächst nur der Initialisierungsteil des jeweiligen Benchmark-Programms ausgeführt wird. Sind alle Programmfäden initialisiert, so wird bei allen Hardware-Fäden gleichzeitig die Ausführung des Hauptteils begonnen. Jeder Hardwarefaden wiederholt so lange die Ausführung des Hauptteils, bis eine Million Takte vergangen sind und die Simulation abgebrochen wird.

Die im folgenden präsentierten Werte sind jeweils die Mittelwerte der 400 Tasksets.

6.3.1 Anzahl Hardware-Fäden

Da jeder zusätzlicher Hardware-Faden teuer ist (vgl. Abschnitt 6.5), lohnt er sich nur, wenn der zusätzliche Geschwindigkeitsgewinn angemessen groß ist. Um dies zu bewerten, wurde die Ausführungshäufigkeit jedes einzelnen Hardware-Fadens in Abhängigkeit von der Anzahl der verfügbaren Hardware-Fäden ermittelt. Die Ausführungshäufigkeit wurde in Form der durchschnittlichen IPC des Hardware-Fadens gemessen. Tabelle 6.7 zeigt das Ergebnis in Verbindung mit der akkumulierten IPC in der untersten Zeile, die ein Maß für den Gesamtdurchsatz darstellt.

Dieser Gesamtdurchsatz wird durch die mehrfädige Ausführung erheblich gesteigert,

Tab. 6.7: IPC-Verteilung in Abhängigkeit von der Anzahl der Hardware-Fäden

	Anzahl Hardware-Fäden							
Priorität	1	2	3	4	5	6	7	8
0	0,3144	0,2920	0,2818	0,2776	0,2757	0,2748	0,2743	0,2741
1		0,2462	0,2375	0,2334	0,2321	0,2315	0,2312	0,2311
2			0,1306	0,1308	0,1310	0,1321	0,1260	0,1253
3				0,0585	0,0608	0,0575	0,0593	0,0574
4					0,0251	0,0243	0,0247	0,0254
5						0,0108	0,0112	0,0108
6							0,0054	0,0054
7								0,0029
	0,3144	0,5382	0,6499	0,7003	0,7247	0,7310	0,7321	0,7324
	Summe							

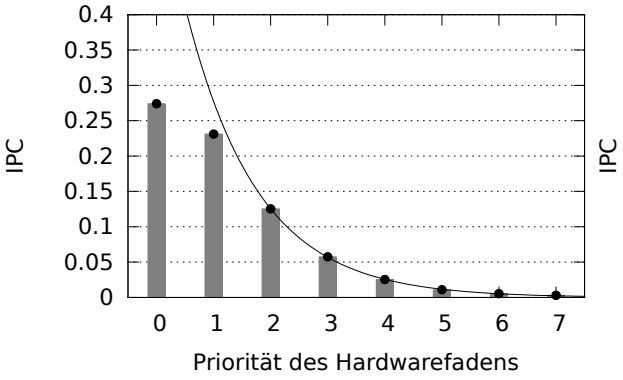

Abb. 6.6: Exponentieller Abfall der Ausführungshäufigkeit

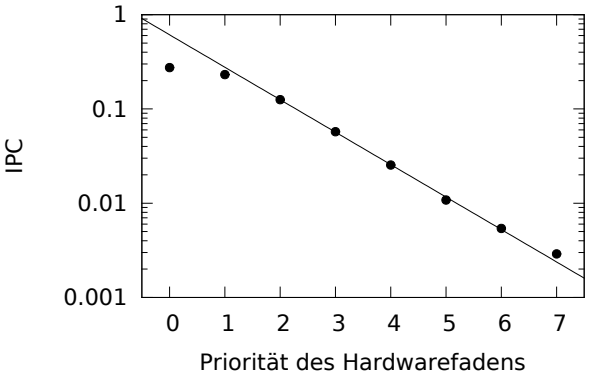

Abb. 6.7: Logarithmische Darstellung des Ausführungshäufigkeitsabfalls

mit drei Hardware-Fäden hat er sich schon mehr als verdoppelt und mit acht Hardware-Fäden wird die Gesamtleistung auf 233% gesteigert.

Es zeigt sich ein exponentieller Abfall der durchschnittlichen IPC mit abnehmender Priorität des Hardware-Fadens (Abbildung 6.6), jedoch erst ab dem dritten Hardware-Faden. Die logarithmische Auftragung (Abbildung 6.7) zeigt dies deutlicher. Deshalb sind drei Hardware-Fäden durchaus sinnvoll und selbst der vierte Hardware-Fäden zeigen noch eine merkliche Programmausführung von über 18% der einfädigen IPC. Weitere Hardware-Fäden kommen jedoch zu selten zur Ausführung, um den zusätzlichen Hardware-Aufwand zu rechtfertigen. So wird der Gesamtdurchsatz bei einer Erhöhung der Anzahl Hardware-Fäden von vier auf acht nur um 10 Prozentpunkte von 223% auf 233% erhöht.

In der Tabelle ist außerdem ein wichtiger Effekt zu beobachten, der die harte Echtzeitfähigkeit der mehrfädigen Ausführung verhindert: Die IPC des Programmfadens mit der höchsten Priorität sinkt, je mehr weitere Hardware-Fäden hinzukommen, das heißt seine

Tab. 6.8: IPC-Verteilung bei Isolierung des DHF per Zugriffsankündigung

	Anzahl Hardware-Fäden							
Priorität	1	2	3	4	5	6	7	8
0	0,3144	0,3144	0,3144	0,3144	0,3144	0,3144	0,3144	0,3144
1		0,1514	0,1448	0,1425	0,1417	0,1414	0,1411	0,1410
2			0,0818	0,0853	0,0873	0,0844	0,0827	0,0879
3				0,0370	0,0360	0,0368	0,0366	0,0359
4					0,0152	0,0144	0,0145	0,0151
5						0,0063	0,0067	0,0067
6							0,0033	0,0034
7								0,0020
	0,3144	0,4658	0,5410	0,5792	0,5946	0,5977	0,5993	0,6064
	Summe							

Ausführungszeit wird von anderen Programmfäden beeinflusst, eine Laufzeitanalyse ist damit fast unmöglich. Der Grund dafür sind die Speicherzugriffe der Hardware-Fäden mit niedrigeren Prioritäten, die für mehrere Takte den Speichercontroller belegen. Im folgenden Abschnitt wird untersucht, wie dies verhindert werden kann.

6.3.2 Zugriffsankündigung

Um den dominanten Hardware-Faden (DHF) vollständig zu isolieren, muss verhindert werden, dass Speicherzugriffe der anderen Hardware-Fäden den DHF verzögern. Durch das in Abschnitt 3.4.5 beschriebene DMAA-Verfahren zur Ankündigung von Speicherzugriffen ist eine derartige Isolation möglich. Doch das Verfahren hat Auswirkungen auf die Ausführungszeiten der anderen Hardware-Fäden, siehe Tabelle 6.8.

Wie erwartet, bleibt die IPC des DHF konstant auf dem Wert, den der Hardware-Faden auch bei alleiniger Ausführung ohne konkurrierende Programmfäden erreicht. Dies wird jedoch durch eine deutlich geringere IPC der anderen Hardware-Fäden erkauft, wie Abbildung 6.8 zeigt. Ohne Zugriffsankündigung erreicht der zweite Hardware-Faden 73% seiner einfädigen IPC, mit Zugriffsankündigung nur knapp 45%. Auch die IPC der weiteren Hardware-Fäden wird durch die Zugriffsankündigung ungefähr um ein Drittel reduziert. Dadurch ist es noch unrentabler, mehr als vier Hardware-Fäden zu benutzen. Das Diagramm zeigt auch, dass, wenn wirklich nur vier Hardware-Fäden genutzt werden, die IPCs wieder ein wenig ansteigen, der Effekt ist aber äußerst gering.

Da die Takte zwischen der Ankündigung des Speicherzugriffs und dem wirklichen Zugriff nur genutzt werden, wenn der Speichercontroller bereits vorher mit einem Speicherzugriff begonnen hat, verstreichen sie oft ungenutzt und reduzieren damit den Gesamtdurchsatz des Prozessors. Bei vier Hardware-Fäden wird deshalb mit Zugriffsankündigung eine akkumulierte IPC von 0,5792 erreicht, das entspricht 184% der einfädigen Durchschnitts IPC und ist deutlich weniger als die 223%, die ohne Zugriffsankündigung erreicht werden.

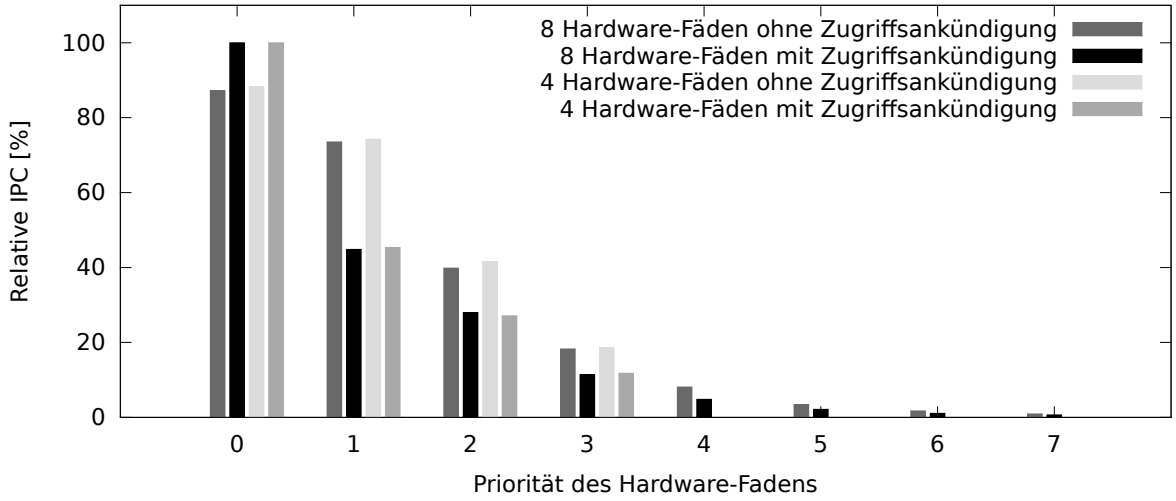

Abb. 6.8: Relative IPCs der Hardware-Fäden mit und ohne Zugriffsankündigung

6.3.3 Verzögerungsgründe niederpriorer Hardware-Fäden

In Abbildung 6.9 werden die Gründe dargestellt, warum ein Hardware-Faden bei mehrfädiger Ausführung nicht ausgeführt wird. Waagerecht sind die Prioritäten der Hardware-Fäden angetragen (0 ist die höchste), senkrecht die prozentuale Verteilung der Gründe. Zu beachten ist hierbei, dass 100% sich jeweils auf die dem Hardware-Faden zur Verfügung stehende Zeit bezieht, das heißt, zu den 100% zählt nur die Zeit, in der die Subpipelines nicht bereits durch andere Hardwarefäden mit höheren Prioritäten belegt sind.

Wie bereits bei der einfädigen Ausführung beobachtet (Abschnitt 6.2.4), dominieren die Speicherlatenzen bei den Gründen, warum ein Hardware-Faden nicht ausgeführt wird. Im Unterschied dazu gibt es jedoch zwei verschiedene Speicherlatenzen, die festen, die durch die Anzahl der zwischen Zuordnungsstufe und Speichercontroller liegenden Pipelinestufen gegeben sind und die dynamischen, die auftreten, wenn der Speichercontroller durch Speicherzugriffe anderer Hardware-Fäden blockiert ist. Bei den ersten Hardware-Fäden ist der feste Anteil noch größer, aber mit sinkender Priorität dominiert die dynamische Datenspeicherlatenz alle anderen Gründe.

Im übrigen zeigt sich deutlich der Einfluss der Zugriffsankündigung: Während der Balken von Hardware-Faden 0 mit Zugriffsankündigung genau dem Balken in Abbildung 6.5 mit den entsprechenden Latenzen 0 und 2 entspricht, gibt es ohne Zugriffsankündigung einen zusätzlichen Anteil durch dynamische Datenspeicherlatenzen. Ebenso deutlich ist der starke Einfluss von den dynamischen Datenspeicherlatenzen auf den zweiten Hardware-Faden, der mit Zugriffsankündigung fast die Hälfte der Beeinträchtigung ausmacht, wohingegen ohne Zugriffsankündigung die ersten beiden Hardware-Fäden eine vergleichbare Verteilung der Gründe aufweisen. (Man beachte, dass die Ausführung des zweiten Hardware-Fadens trotzdem deutlich geringer ist, da die 100% sich nur auf denjenigen Anteil beziehen, der von ersten Hardware-Faden nicht genutzt werden konnte.)

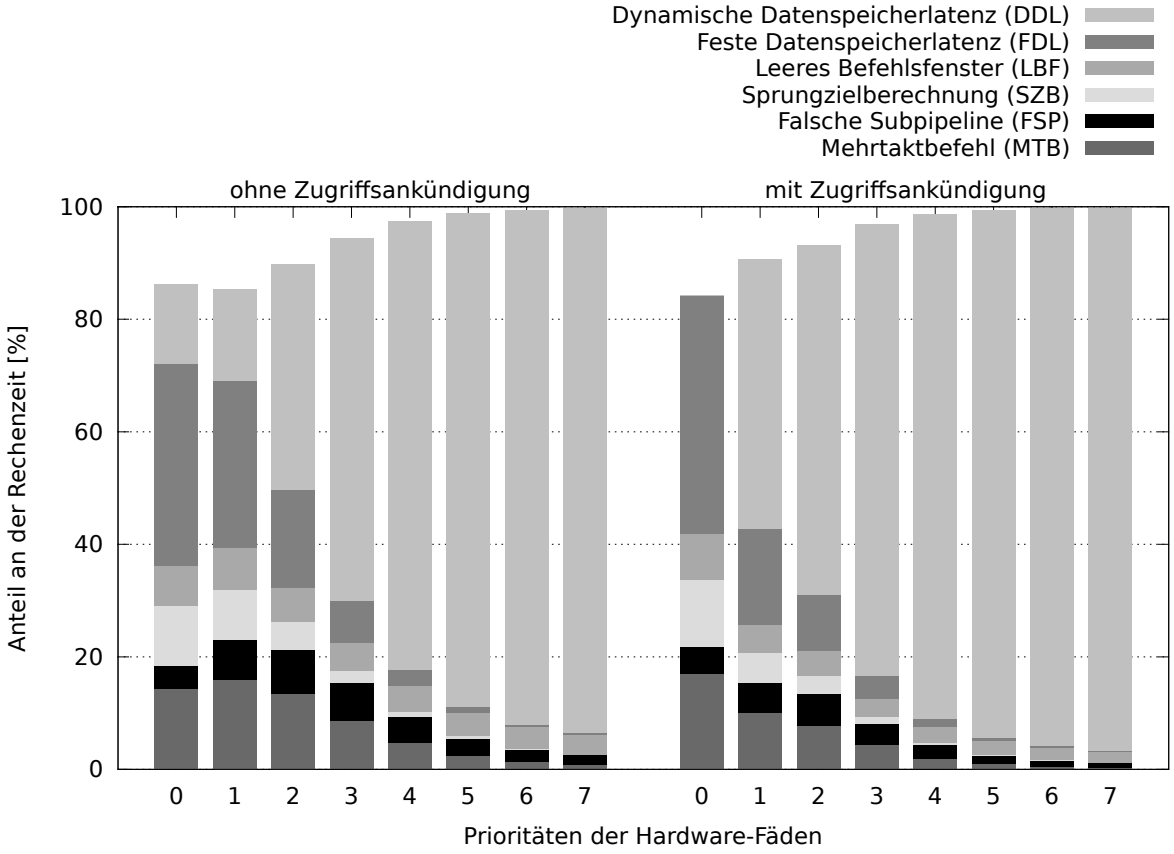

Abb. 6.9: Verzögerungsgründe in Abhängigkeit von den Priorität

6.3.4 Optimierung der Befehlsbereitstellung

In Abschnitt 5.4.2 wurden zwei Techniken vorgestellt, mit deren Hilfe die Anzahl der Speicherzugriffe auf den Befehlsspeicher reduziert werden können. Ihre Auswirkungen auf die Anzahl der durchgeführten Befehlspeicherzugriffe hängt stark vom jeweiligen Benchmark ab (Abbildung 6.10). Je nach Benchmark reduziert eine der beiden Optimierungen die Anzahl der Zugriffe stärker.

Im Durchschnitt jedoch reduziert die ENOUGH-Optimierung die Zugriffe um 9,3%, die AHEAD-Optimierung ist deutlich stärker mit 14,3% und beide zusammen erreichen eine Verringerung um 23,2%. Dieser Wert ist etwas besser als der theoretischen Wert von 22,2%, den man erhält, wenn man annimmt, dass beide Optimierungen unabhängig voneinander sind. Die Erklärung für dieses positive Zusammenwirken wird in Abschnitt 5.4.2 gegeben.

Trotz dieser erheblichen Reduzierung, wirken sich die Optimierungen in so geringem Maße auf die durchschnittlichen IPCs aus, dass sie geringer sind als die statistischen Schwankungen. Das liegt daran, dass das Warten auf einen Befehlsspeicherzugriff in weniger als 8% der Fälle der Grund für die Nichtausführung eines Programmfadens ist (siehe LBF in Abbildung 6.9).

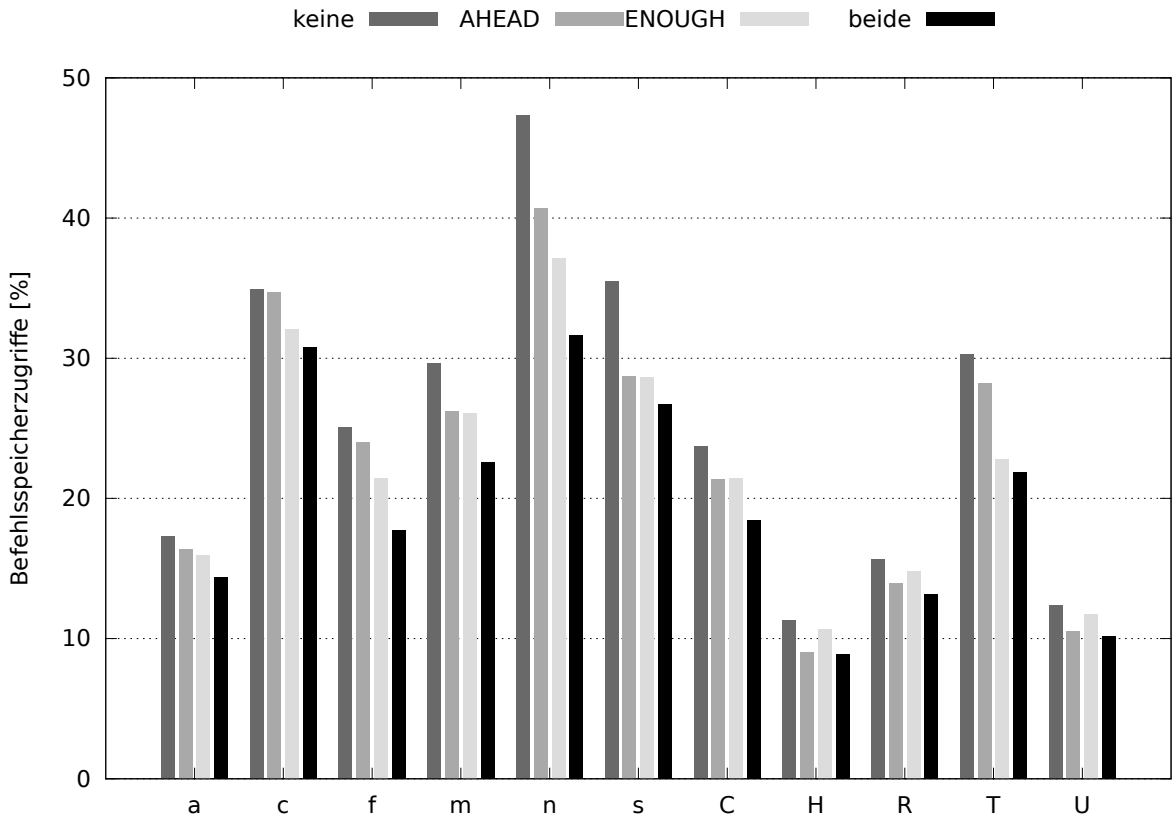

Abb. 6.10: Prozentualer Anteil der Befehlsspeicherzugriffe an der Gesamtlaufzeit

Insofern ist es fraglich, ob sich die Optimierungen lohnen, wenngleich die Hardware-Kosten relativ gering sind. Nur die AHEAD-Optimierung allein belegt 18 zusätzliche ALUTs pro Hardware-Faden, die ENOUGH-Optimierung benötigt 44 und beide zusammen könenn mit 60 ALUTs pro Hardware-Faden realisiert werden. Dies ist etwa ein Prozent der Gesamtgröße eines Hardware-Fadens und damit in einer vertretbaren Größenordnung. Für Einzelheiten zu der Bedeutung der ALUTs und der Größe der Hardware-Fäden siehe Abschnitt 6.5.

6.4 Ablaufplanung mit weichen Echtzeitanforderungen

Die herausragendste Eigenschaft von SMT-Prozessoren ist das Überlappen der Ausführungszeiten verschiedener Programmfäden, um die Auslastung der Prozessor-Ressourcen zu erhöhen. Da zu einem bestimmten Zeitpunkt nur maximal ein hart echzeitfähiger Programmfaden aktiviert sein darf und dieser stets die höchste Priorität haben muss, bleibt für andere, gleichzeitig ausgeführte Programmfäden nur ein exponentiell abfallende Rechenzeitanteil, wie er in Abschnitt 6.3.1 ermittelt wurde. Anders sieht es aus, wenn mehrere gleichberechtigte Programmfäden mit weichen Echtzeitanforderungen zur selben Zeit ausgeführt werden.

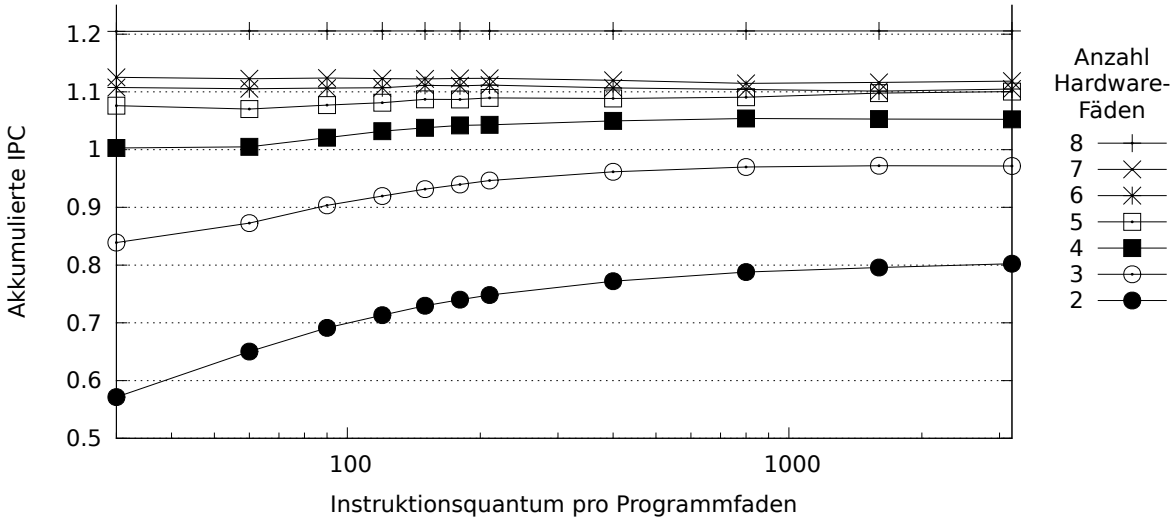

Abb. 6.11: Durchschnittliche Gesamtrechenleistung eines Tasksets von 8 Programmfäden in Abhängigkeit vom Instruktionsquantum und der Anzahl Hardware-Fäden

6.4.1 Minimales Instruktionsquantum

Kontextwechsel in den Speicher kosten Zeit. Während dieser Zeit ist ein Hardware-Faden belegt, ohne für die Ausführung zur Verfügung zu stehen. Beim DTS-Scheduling für harte Echtzeitprogramme werden stets zwei Hardware-Fäden reserviert, um diese Kontextwechselzeit zu verstecken. Trotzdem dauert ein Kontextwechsel eine gewisse Zeit ($\Delta t_{out} + \Delta t_{in} = 40$ Takte beim CarCore), die die Länge des Taktquantums eines Programmfadens nach unten hin begrenzt.

Beim PIQ- und RSS-Scheduling werden die Kontextwechsel zwar nicht versteckt, jedoch setzt die Programmausführung für die Dauer des Kontextwechsels aus, wodurch der Gesamtdurchsatz sinkt. Bei zu häufigem Kontextwechseln kann die Leistungsminderung sehr hoch werden. Dadurch ergibt sich zwar kein scharfes Mindestquantum wie bei DTS, jedoch gibt es ein ungefähre Untergrenze für das Instruktionsquantum, dass die Beeinträchtigung durch die Kontextwechsel auf ein unbedeutendes Maß reduziert.

Um diese Untergrenze zu ermitteln, wurden 400 zufällig erzeugte Tasksets von jeweils acht Programmfäden per RSS-Algorithmus ausgeführt. Jeder Programmfaden erhält dabei das gleiche Instruktionsquantum. In Abbildung 6.11 ist die akkumulierte IPC über alle Hardware-Fäden gegen die Länge des Instruktionsquantums aufgetragen (logarithmische Skala).

Es zeigt sich, dass nur bei einer kleinen Zahl von Hardware-Fäden ein merklicher Effekt gemessen werden kann, für fünf oder mehr Hardware-Fäden ist der Unterschied vernachlässigbar. Das liegt daran, dass sobald ein Hardware-Faden aufgrund eine Kontextwechsels deaktiviert wird, alle Hardware-Fäden mit niedrigerer Priorität häufiger ausgeführt werden, so als hätten sie eine um eine Stufe höhere Priorität. Dadurch wirkt sich das Ausschalten eines Hardware-Fadens immer nur so stark aus, wie der Anteil des

Hardware-Fadens mit der niedrigsten Priorität an der Gesamtrechenleistung wäre. Dieser IPC-Anteil kann Tabelle 6.7 entnommen werden und entspricht einem Bruchteil von $0,0251$ der Gesamt-IPC $0,7247$, also weniger als 4%. Durch eine weitere Erhöhung der Anzahl von Hardware-Fäden sinkt dieser Anteil exponentiell.

Doch selbst bei weniger als fünf Hardware-Fäden wird die Verringerung der Rechenleistung so gering, dass spätestens ab einem Quantum von 400 Instruktionen von einer vernachlässigbaren Beeinträchtigung gesprochen werden kann.

6.4.2 Überlappung von zwei Programmfäden

Um die Vorteile der überlappenden Programmausführung zu untersuchen, wurde das PIQ-Scheduling für weiche Echtzeitanforderungen dem hart echtzeitfähigen DTS-Scheduling gegenübergestellt. Beide Algorithmen verwenden eine Rundenlänge von 1000 Takten, die gleichmäßig auf die beiden Programmfäden verteilt wird. Bei DTS geschieht dies durch Zuweisung von jeweils 500 Takten, bei PIQ bekommt jeder Programmfaden ein individuelles Instruktionsquantum, das der Anzahl Instruktionen, die durchschnittlich in 500 Takten ausgeführt werden, entspricht ($= 500 \times IPC$).

Bevor die weiteren Messungen durchgeführt wurden, wurde jedes Benchmark-Programm einmal mit PIQ und einmal mit DTS ausgeführt, um zu vergleichen, ob mit der Wahl der beiden Quanten wirklich der gleiche Rechenzeitanteil erreicht wird. Dazu wurde für jedes Programm eine Iterationszahl gewählt, die zu einer Laufzeit des Hauptteils von mindestens 10 Millionen Takten führte. Bei keinem der Benchmark-Programme unterschied sich die Ausführungszeit bei PIQ und DTS um mehr als 5000 Takte, also 0,05 Prozent. Dieser geringe Unterschied ist auf Rundungsfehler zurückzuführen, da das Instruktionsquantum ganzzahlig sein muss.

Um die Einsparungsmöglichkeiten durch das Überlappen zu untersuchen, wurde ein Taskset mit dem PIQ-Algorithmus ausgeführt und für jede der ersten 1000 Runden ermittelt, nach wie vielen Takten beide Programmfäden ihr Instruktionsquantum erreicht haben. Diese Dauer wird im folgenden als *effektiver Rundenlänge* R_{eff} bezeichnet, da ab diesem Zeitpunkt keine weiteren Instruktionen innerhalb der Runde ausgeführt werden.

Aus diesen 1000 Werten wurde die minimale, die maximale und die durchschnittliche effektive Rundenlänge ermittelt und in Abbildung 6.12 dargestellt. Die Tasksets wurden von oben nach unten aufgetragen, wobei der Buchstabe an der Y-Achse angibt, welches Benchmark-Programm vom ersten Programmfaden ausgeführt wurde. Der Name des zweiten Bechmark-Programms kann aufgrund der räumlichen Begrenzung nicht angegeben werden, die Tasksets sind aber in der gleichen Reihenfolge sortiert wie die des ersten Programmfadens.

Die effektive Rundenlänge beträgt bei keiner Runde eines Tasksets mehr als 824 Takte. Bei den Tasksets ohne Beteiligung von Benchmark `c-crc` reichen sogar 627 Takte aus. 18 beziehungsweise 37 Prozent der Runde werden also niemals genutzt, so dass die Taktfrequenz um diesen Prozentsatz reduziert werden könnte, um Energie zu sparen.

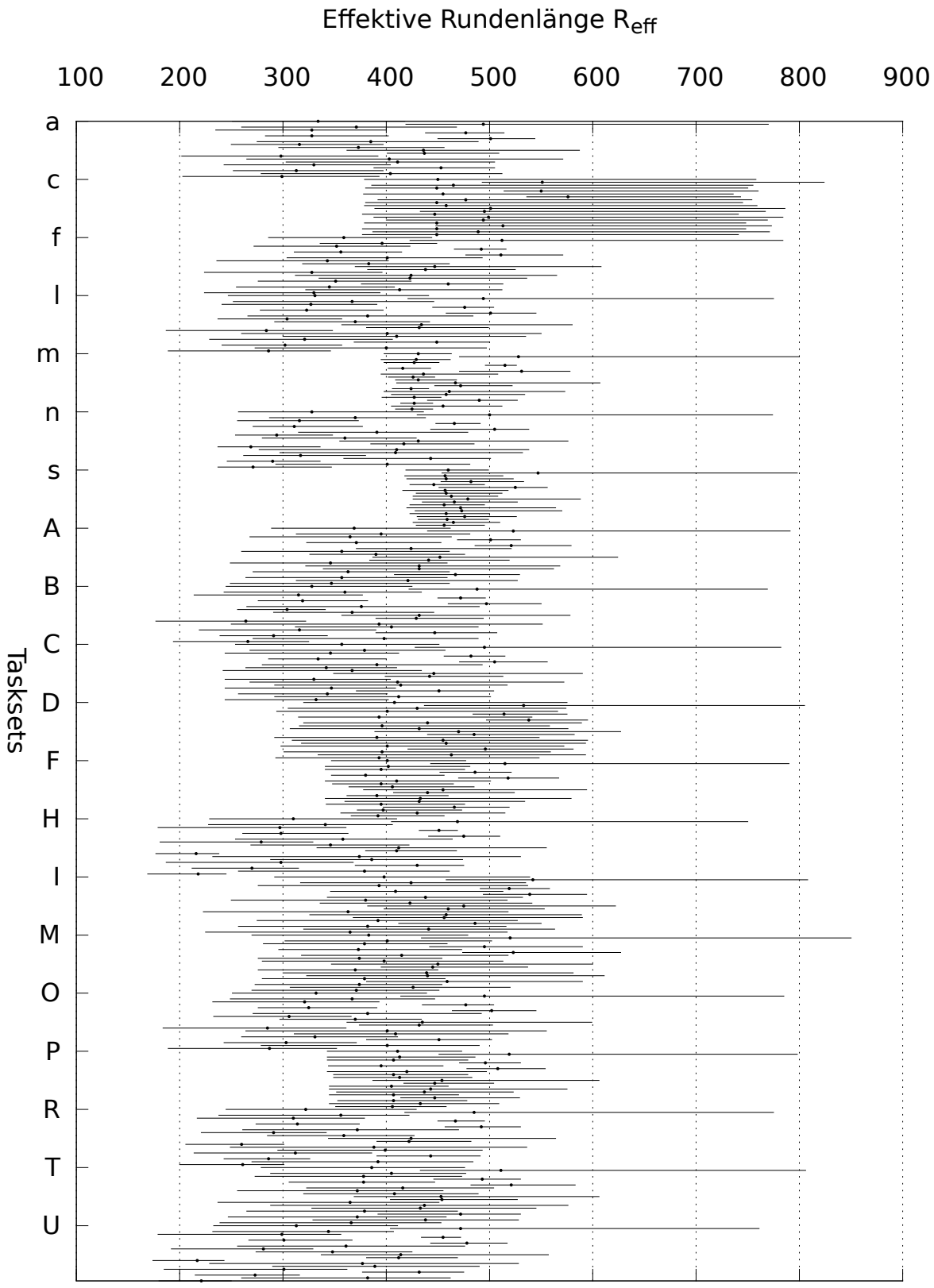

Abb. 6.12: Effektive Rundenlänge für zwei PIQ-Programmfäden

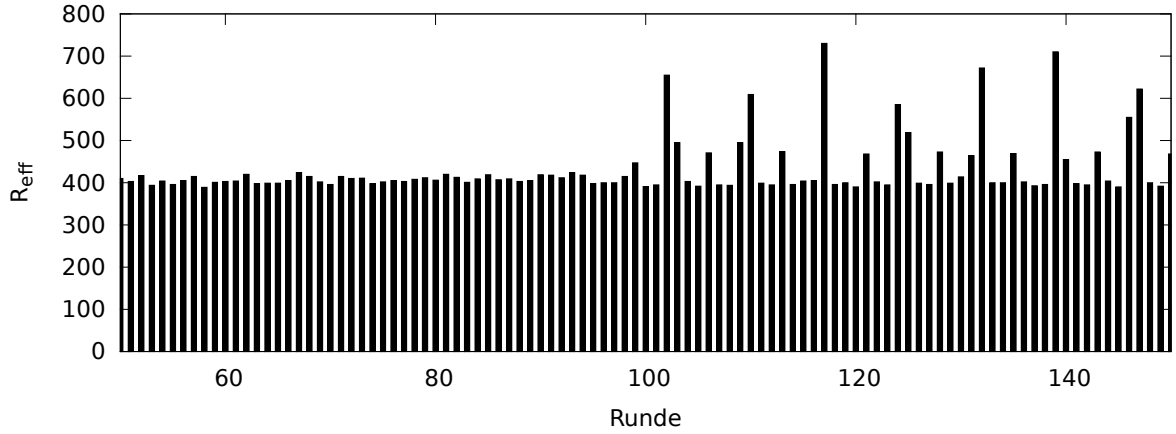

Abb. 6.13: Effektive Rundenlängen von Taskset `cc`

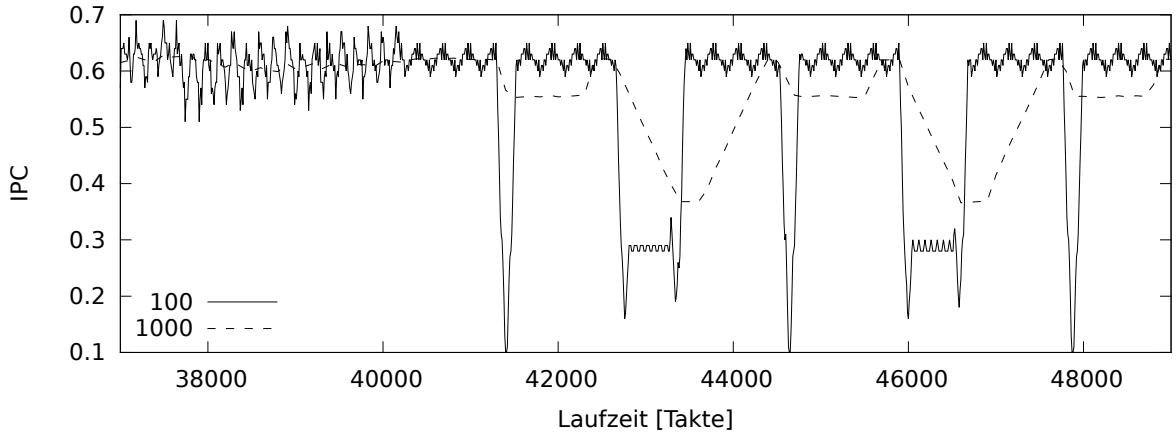

Abb. 6.14: IPC des Benchmarkprogramms `c-crc`

Der Grund, für das auffällige Verhalten von `c-crc` ist aus Abbildung 6.13 ersichtlich: während der ersten 100 Runden liegt die effektive Rundenlänge ganz gleichmäßig bei circa 400 Takten, doch dann ist ungefähr jede siebte Runde fast doppelt so lange. Sieben Runden mit je 500 Takten entsprechen ungefähr der durchschnittlichen Iterationslänge von Programmfaden `c-crc` (3267, siehe Tabelle 6.2). Folglich wird einmal pro Iteration ein problematisches Codestück mit extrem niedriger IPC ausgeführt. Diese Verhalten beginnt jedoch erst nach ungefähr 40000 Takten, wie aus dem IPC-Diagramm (Abbildung 6.14) ersichtlich ist.

Eine andere Möglichkeit, die Leerlaufzeit des Prozessors zu nutzen, funktioniert selbst mit Ausreißern wie bei Benchmark `c-crc` noch sehr gut: der Prozessor wird nach Erreichen der effektiven Rundenlänge in einen Schlafmodus versetzt und erst zu Beginn der nächsten Runde wieder aufgeweckt. Nimmt man 10 Takte für das Umschalten in den Schlafmodus und wiederum 10 Takte für das Aufwecken an, würde der Prozessor im Durchschnitt 57% der Zeit schlafen (mindestens 40,4% bei Taskset `cs` und maximal 76,4% bei Taskset `HH`).

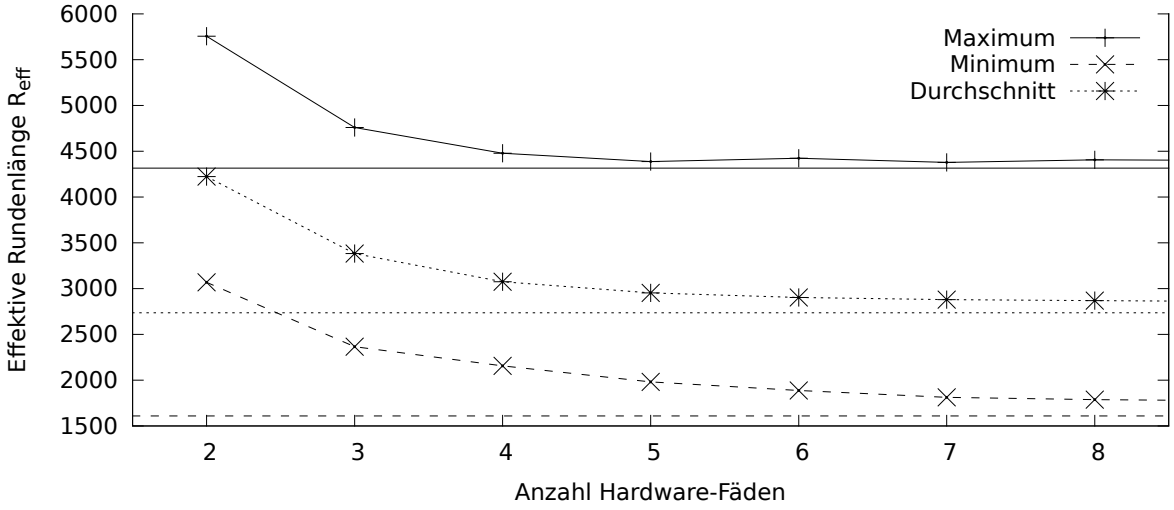

Abb. 6.15: Effektive Rundenlänge in Abhängigkeit von der Anzahl Hardware-Fäden

6.4.3 Überlappung von vielen Programmfäden

Um das volle Potential der Überlappung von Ausführungszeiten zu ermitteln, wurde ein weiterer Versuch mit Tasksets von 20 Programmfäden durchgeführt. Wiederum wurde die Rundenlänge von 10000 Takten gleichmäßig auf alle Programmfäden aufgeteilt und die resultierenden 500 Takte mit der IPC des jeweiligen Benchmark-Programms multipliziert, um das Instruktionsquantum zu erhalten. Es wurden 400 Taskset mit einer zufälligen Kombination von 20 Programmfäden generiert und für jeweils 10000 Runden ausgeführt. Der Name eines Tasksets ergibt sich aus der Aneinanderreihung der Abkürzungsbuchstaben der Benchmark-Programme.

Während die Ausführung von DTS-Fäden gleichmäßig über die ganze Runde verteilt wird, werden beim PIQ die Programmfäden am Beginn der Runde „zusammengezogen“ , wodurch am Ende die Differenz zwischen effektiver Rundenlänge und wirklicher Rundenlänge übrig bleibt. Die effektive Rundenlänge hängt stark von den Benchmark-Programmen ab, meistens (bei 389 der 400 untersuchten Tasksets) ergibt sich jedoch eine Verteilung in Form einer gaußschen Glockenkurve (Abbildungen 6.16 und 6.17). Es gibt jedoch auch unregelmäßige Verteilungen wie in Abbildung 6.18 und 6.19.

Abbildung 6.15 zeigt die minimale, durchschnittliche und maximale effektive Rundenlänge in Abhängigkeit von der Anzahl der zur Verfügung stehenden Hardware-Fäden. Mit steigender Anzahl von Hardware-Fäden nähern sich die drei Graphen jeweils dem Idealwert an, der durch die waagrechten Linien gegeben ist. Er entspricht den Werten der effektiven Rundenlänge, wenn keine Kontextwechsel nötig sind, also 20 Hardware-Fäden für die 20 Programmfäden zur Verfügung stehen. Bei vier Hardware-Fäden entspricht die durchschnittliche effektive Rundenlänge einem Drittel der wirklichen Rundenlänge, bei keiner einzigen Runde eines Tasksets übersteigt die effektive Rundenlänge 45% der wirklichen Rundenlänge. Über die Hälfte der Rechenzeit kann somit für andere Programmfäden oder zum Einsparen von Energie verwendet werden.

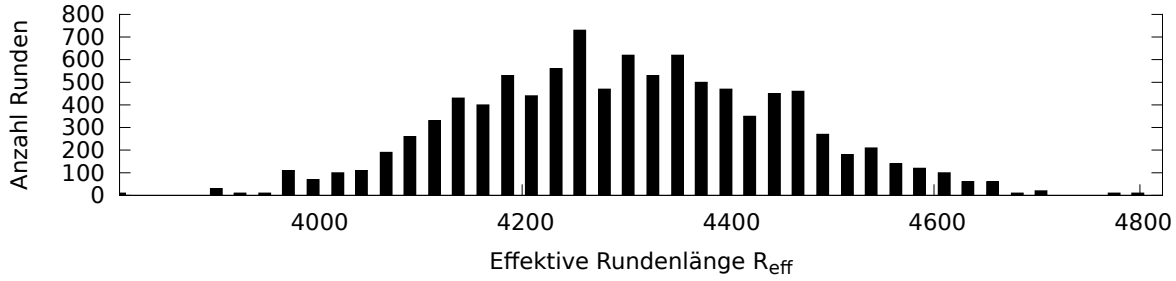

Abb. 6.16: Effektive Rundenlänge von `lfmFCHIUMDFfOHDPmMsT` bei 2 Hardware-Fäden

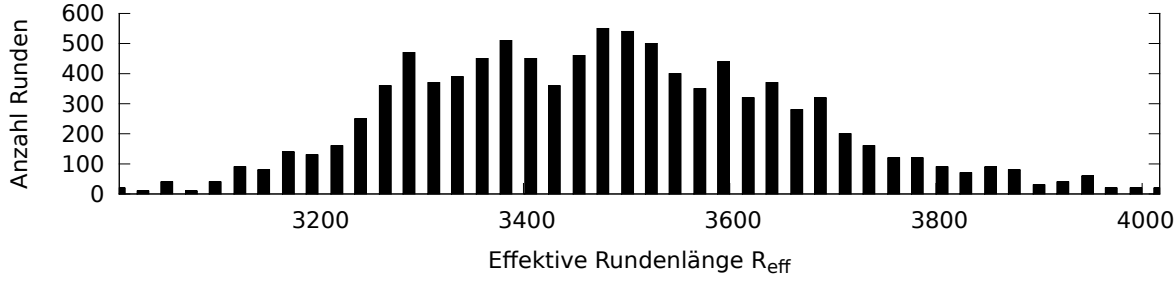

Abb. 6.17: Effektive Rundenlänge von `nFAMCHMmAnMPOHITTclc` bei 3 Hardware-Fäden

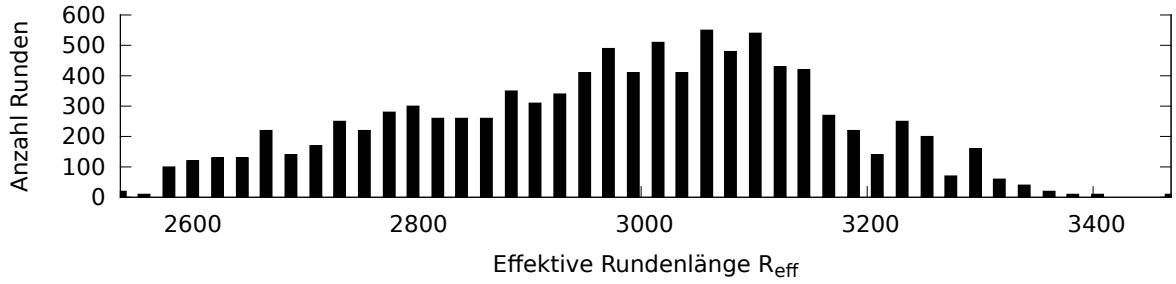

Abb. 6.18: Effektive Rundenlänge von `mAsFHAsUBAmDOmABBPUH` bei 4 Hardware-Fäden

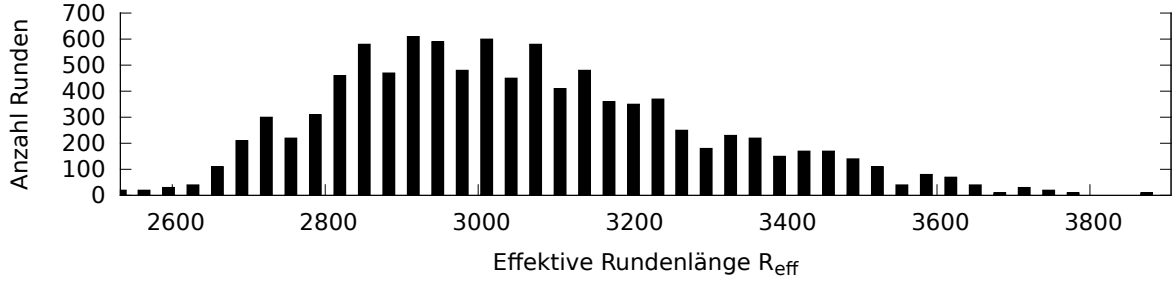

Abb. 6.19: Effektive Rundenlänge von `cMaDTCDsnRcMARDcPHCM` bei 5 Hardware-Fäden

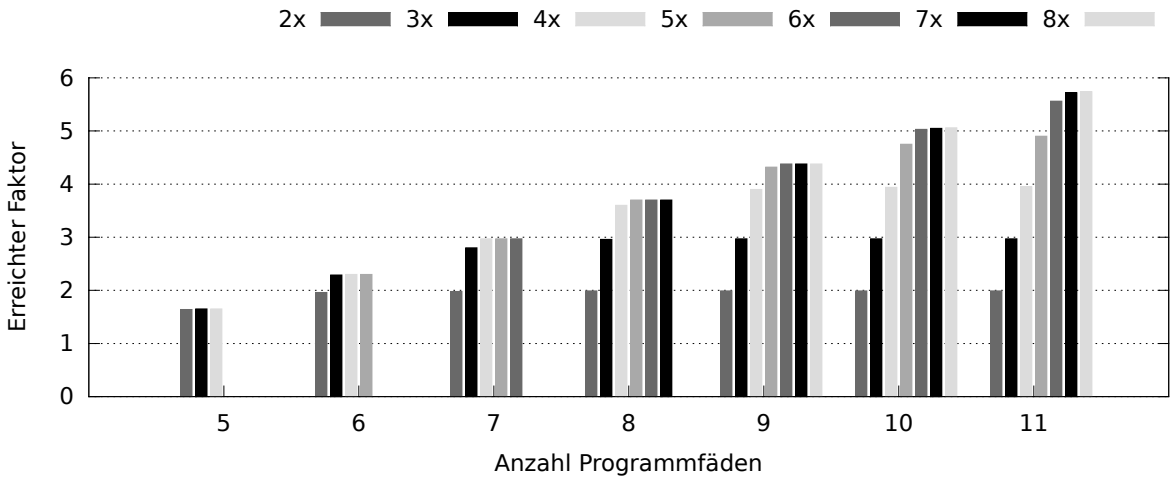

Abb. 6.20: Erreichte Rechenzeit des RRS-Fadens mit erhöhten Quantum im Verhältnis zum durchschnittlichen Rechenzeitanteil der anderen RRS-Fäden

Weiterhin zeigt das Diagramm, dass mit mehr als fünf Hardware-Fäden keine Verbesserung der Auslastung erreicht wird, es aber trotzdem einen konstanten Overhead durch das Ein- und Auslagern gibt, da die jeweiligen Kurven nicht ganz den waagerecht eingezeichneten Idealwert ohne Kontextwechsel errreichen.

6.4.4 Unterschiedliche Quanten bei RRS

Werden mehreren RRS-Fäden unterschiedliche Quanten zugeordnet, so wird die zur Verfügung stehende Rechenzeit nicht genau im Verhältnis der Quanten auf die jeweiligen Programmfäden verteilt. Wie in Abschnitt 4.2.3.3 bereits erläutert, liegt das daran, dass es sein kann, dass der Scheduler beim wiederholten Durchgehen der RRS-Liste bei einem Programmfaden ankommt, dessen Ausführung noch andauert. Dies passiert, wenn weniger RRS-Fäden als Hardware-Fäden vorhanden sind oder wenn einige der RRS-Fäden ein so kurzes Quantum haben, dass sie wieder ausgelagert werden, bevor der vor ihnen eingelagerte RRS-Faden mit einem großen Quantum letzteres verbraucht hat.

Die einfachste Reaktion auf das beschriebene Problem ist es, den entsprechenden Programmfaden einfach zu überspringen. Doch dadurch kommt es zu einer ungleichen Verteilung der Rechenzeit, wie Abbildung 6.20 zeigt. Für dieses Diagramm wurden Tasksets mit einer unterschiedlichen Anzahl von Programmfäden ($5 - 11$) per RRS-Scheduling auf einem CarCore mit vier Hardware-Fäden ausgeführt. Jeder Programmfaden hatte das gleiche Quantum von 200 Instruktionen, nur das Quantum des ersten RRS-Fadens wurde mit einem Faktor multipliziert. Jede Messung wurde mit 400 verschiedenen Tasksets durchgeführt und darüber gemittelt. Jeder Balken entspricht einem anderen Faktor.

Es ist zu erkennen, dass der maximal erreichbare Faktor davon abhängt, wie viele Programmfäden insgesamt zur Verfügung stehen. Erst ab 6 Programmfäden kann ein RRS-

Faden das doppelte Quantum erhalten, für das dreifache sind bereits 8 RRS-Fäden nötig. Höhere Faktoren steigern zwar den Rechenzeitanteil, ab einem gewissen Faktor nimmt der Zuwachs des Rechenzeitanteils aber im Verhältnis zur Steigerung des Faktors stark ab. Dadurch ergibt sich eine obere Schranke für das Verhältnis von maximalem zu minimalem Quantum. In Abbildung 6.20 ist zu erkennen, dass diese obere Schranke linear von der Anzahl der RRS-Fäden abhängt.

Dieses Verhalten erklärt sich daraus, dass der maximale Faktor erreicht ist, wenn alle anderen RRS-Fäden zusammen genommen schneller ausgeführt werden als der Programmfaden mit dem hohen Quantum. Denn dann erreicht der Hardware-Scheduler beim Durchgehen der RRS-Liste den Programmfaden mit dem hohen Quantum noch bevor er sein Quantum verbraucht hat und überspringt ihn. Damit ist verständlich, dass durch eine größeren Anzahl von Programmfäden die Dauer, bis es zu einer Überholung kommt erhöht wird und dadurch dem Programmfaden ein höherer effektiver Faktor zugestanden werden kann, bevor er übersprungen wird.

6.5 Hardwareverbrauch des CarCore FPGA Prototyps

In Tabelle 6.9 sowie der zugehörigen Abbildung 6.21 sind die Eckdaten den FPGA-Prototyps in Abhängigkeit von der maximalen Anzahl von Hardware-Fäden aufgeführt. Für die Synthese wurde ein Altera Stratix II EP2S180F1020C3 FPGA-Chip verwendet. Die Größe des Prototyps ist in Adaptive Lookup Tables (ALUTs) angegeben, dies sind die elementaren Logikbausteine dieses FPGA-Chips.

Die Anzahl der ALUTs und der Register-Bits steigt nahezu linear mit der Anzahl der Hardware-Fäden, es gibt nur eine kleine Delle bei vier Hardware-Fäden da diese Konfiguration besonders optimiert wurde. Demnach benötigt jeder zusätzliche Hardware-Faden etwa 6000 ALUTs plus 9000 ALUTs für den Rest des Prozessors. Der Scheduler zeigt ebenfalls eine lineare Abhängigkeit von der Anzahl der Hardware-Fäden.

Tab. 6.9: Hardwareverbrauch und maximale Taktfrequenz des FPGA-Prototyps

Anzahl	Gesamtgröße		Scheduler		Anteil	Taktfrequenz
Hardware-Fäden	[ALUTs]	[Bits]	[ALUTs]	[Bits]	(ALUTs)	[MHz]
1	14808	3775	1012	492	6,8%	27,17
2	21129	5886	1621	624	7,7%	26,10
3	27519	7979	2372	753	8,6%	24,38
4	31603	10043	3171	892	10,0%	17,55
5	39325	12113	4345	1103	11,0%	11,10
6	45400	14189	5303	1246	11,7%	8,52
7	49082	16296	6135	1338	12,5%	7,00

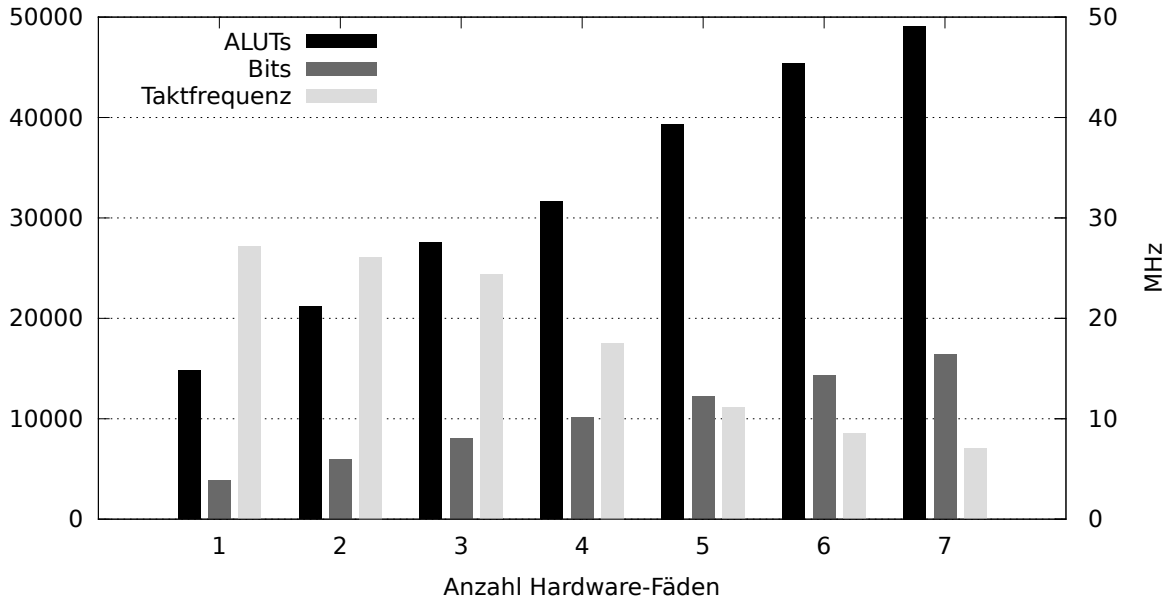

Abb. 6.21: Hardwareverbrauch und maximale Taktfrequenz des FPGA-Prototyps

Durch weitere Arbeiten wäre eine Erhöhung der Taktfrequenz durchaus möglich. Sie wird durch den kritischen Pfad begrenzt, den Teil der Schaltung, der am längsten für die Ausführung benötigt. Momentan liegt dieser kritische Pfad im Scheduler, der nicht speziell optimiert wurde. Es wäre ohne Verlängerung der Kontextwechselzeit möglich, kritische Entscheidungen des Schedulers auf mehrere Takte zu verteilen, um damit den kritischen Pfad zu verkürzen.

Ohne den Scheduler liegt der kritische Pfad bei der Division. Dieser könnte ebenso umgangen werden, wenn die Division auf vier Takte aufgeteilt werden würde wie beim TriCore. Doch die für beide Optimierungen nötigen Arbeiten sind sehr aufwendig und ihr wissenschaftlicher Wert ist fraglich, weshalb darauf verzichtet wurde.

6.6 Zusammenfassung

Die einfädige Ausführung von Programme dauert auf dem CarCore ungefähr doppelt so lange wie auf dem originalen TriCore. Das liegt nur zu einem geringen Prozentsatz an den Maßnahmen, die nötig sind, um einen hart echtzeitfähigen Programmfaden von den konkurrierenden Programmfäden zu isolieren. Hauptgrund ist die spezielle Hardware-Beschleunigung von Unterprogrammaufrufen im TriCore, die nur schlecht auf ein mehrfädiges System übertragen werden kann. Ausgeglichen wird dies jedoch durch die Überlappende Ausführung mehrerer Programmfäden, die zu einer Verdoppelung des Prozessordurchsatzes führt.

Die Ausführungszeiten reagieren sehr empfindlich auf Erhöhungen der Speicherlatenzen, sowohl bei Zugriffen auf den Befehls- als auch auf den Datenspeicher. Aufgrund der sehr

speziellen Befehlskodierung des TriCore-Befehlssatzes kann durch geschickte Analyse der bereitgestellten Instruktionen die Anzahl der Speicherzugriffe auf den Befehlsspeicher reduziert werden, wodurch die Ausführung von Programmfäden mit niedrigeren Prioritäten etwas beschleunigt wird.

Die optimale Anzahl von Hardware-Fäden ist drei, bei zusätzlichen Hardware-Fäden wird die Gesamtauslastung nur noch in geringem Maße erhöht. Da bei Benutzung des DTS-Algorithmus jedoch immer ein Hardware-Faden still steht, sind in diesem Fall vier Hardware-Fäden optimal. Während das minimale Quantum beim DTS-Scheduling der Kontextwechselzeit (40 Takten) entspricht, liegt das minimale Quantum beim PIQ-Scheduling bei ungefähr 400 Instruktionen. Ab diesem Instruktionsquantum wird die Ausführung nicht mehr durch zu häufige Kontextwechsel beeinträchtigt.

Wenn auf harte Echtzeitanforderungen verzichtet werden kann, kann durch die Überlappung der Programmausführung über die Hälfte der Rechenzeit eingespart werden. Egal ob nur zwei oder 20 Programmfäden, durchschnittlich sind die per PIQ ausgeführten Programmfäden bereits zur Hälfte der Runde fertig.

Der Hardware-Bedarf des CarCore-Prozessors hält sich in Grenzen, er vergrößert sich mit jedem zusätzlichem Hardware-Faden um circa 6000 ALUTs, also ungefähr 40% der einfädigen Version.

7 Verwandte Arbeiten

Durch die Verbindung der drei Gebiete Echtzeitfähigkeit, simultane Mehrfädigkeit und Hardware-Scheduler ergeben sich zahlreiche Schnittpunkte mit anderen Forschungsarbeiten und kommerziellen Produkten.

In der folgenden Darstellung wurden sie so angeordnet, dass zuerst auf hart echtzeitfähige Architekturen eingegangen wird, die eine mehrfädige Programmausführung bieten, wobei nur eine der Arbeiten auch die simultan mehrfädige Ausführung unterstützt. Im anschließenden Abschnitt liegt der Schwerpunkt hingegen auf der simultanen Mehrfädigkeit, die Echtzeitanforderungen sind geringer, weshalb nur weiche Echtzeitbedingungen erfüllt werden.

Der dritte Abschnitt beschäftigt sich mit der Verarbeitung aperiodischer Anfragen in harten Echtzeitsystemen. In diesem Bereich gibt es im Unterschied zu den beiden vorangehenden bisher keine nennenswerten kommerziellen Implementierungen. Den letzten Teil bilden Hardware-Komponenten, die die Ablaufplanung übernehmen können.

Aus der großen Menge von verwandten Arbeiten stechen drei Projekte besonders hervor:

XCore (7.1.2) Einziger wirklich hart echtzeitfähiger kommerzieller SMT-Prozessor.

RVMP (7.2.2) Wissenschaftliche Untersuchung von harter Echtzeit in Kombination mit SMT-Prozessoren.

RMT-Prozessor (7.3) Bietet zwar nur weiche Echtzeit, wurde aber wirklich hergestellt und hat eine ähnliche Organisation und Aufteilung der Architektur wie der CarCore.

7.1 Harte Echtzeit durch Interleaved Multithreading

Eine sehr gebräuchliche Methode, um harte Echtzeitanforderungen in Verbindung mit einer mehrfädigen Programmausführung zu erfüllen, ist das *Interleaved Multithreading* (Abschnitt 2.4.1). Dabei befindet sich von jedem Programmfaden nur jeweils eine Instruktion gleichzeitig in der Pipeline, weshalb Wechselwirkungen zwischen den Programmfäden relativ einfach unterbunden werden könenn. Allerdings wird dadurch die Ausführungsgeschwindigkeit der einzelnen Programmfäden stark herabgesetzt, da bei einer Pipelinelänge von S Stufen ein Programmfaden nur alle S Takte ausgeführt wird. Außerdem kann nur jeweils eine Instruktion pro Takt ausgeführt werden, nicht mehrere wie bei der simultan mehrfädigen Ausführung des CarCore-Prozessors. Interleaved Multihreading eignet sich daher eher für die Ausführung vieler kurzer Programmfäden.

7.1.1 Precision Timed Architecture

Die einfachste Version des *interleaved multithreading* wurde in der *Precision Timed (PRET) Architecture* [Lickly 2008] realisiert. Es stehen genau sechs Programmfäden zur Verfügung, die in einer festgelegten Reihenfolge jeweils eine Instruktion durch die Pipeline schicken. Da die Pipeline genau sechs Stufen hat, ist zu jedem Zeitpunkt nur eine Instruktion eines Programmfadens in der Pipeline und es kann zu keinerlei Abhängigkeiten kommen.

Auch die anderen Teile der Architektur sind so ausgelegt, dass ihr zeitliches Verhalten möglichst genau voraussagbar ist. Jeder Programmfaden hat einen schnellen, privaten Daten- und Code-Speicher, wodurch hier keine Kollisionen auftreten können. Der Zugriff auf den gemeinsamen Speicher wird über ein sogenanntes Speicherrad realisiert, das heißt, innerhalb einer Umdrehung des Rades ($6 \times 13 = 78$ Takte) kann jeder Programmfaden zu einem festen Zeitpunkt einen 13 Takte dauernden externen Speicherzugriff durchführen. Im ungünstigsten Fall muss also 77 Takte auf das nächste Fenster im Speicherrad gewartet werden. Zusammen mit der Zugriffsdauer ergibt sich somit die maximale Speicherlatenz von 90 Takten.

Da bei der Entwicklung der PRET Architektur sehr viel Wert auf exakte zeitliche Abläufe gelegt wurde, ist eine Laufzeitanalyse sehr einfach durchzuführen, jedoch ist die Architektur sehr unflexibel, die Anzahl der Programmfäden ist fest mit der Anzahl der Pipeline-Stufen verknüpft, wodurch eine Änderung nahezu unmöglich ist. Noch dazu erhält jeder Programmfaden genau ein Sechstel der Prozessorleistung, weshalb bei der Software-Entwicklung sehr genau auf die Gleichverteilung der Programmfäden geachtet werden muss.

7.1.2 XMOS XCore

Beim kommerziellen *XCore*-Prozessor [May 2009] der britischen Firma XMOS[1] gestaltet sich die Ablaufplanung etwas weniger restriktiv, da es keine Untergrenze für die Anzahl der Programmfäden gibt. Falls die Anzahl der Programmfäden N kleiner als die Anzahl der Pipeline-Stufen S ist, so werden regelmäßig leere Instruktionen durch die Pipeline geschickt, sodass weiterhin garantiert ist, dass sich nur jeweils eine Instruktion eines Programmfadens in der Pipeline befindet. In diesem Fall erhält jeder Programmfaden $\frac{1}{S}$ der Rechenzeit. Ist die Anzahl der Programmfäden größer, so erhält jeder Programmfaden $\frac{1}{N}$ der Rechenzeit.

Nach oben hin ist die Anzahl der Programmfäden auf acht begrenzt, da es keine, durch Hardware unterstützte Möglichkeit gibt, Programmfäden in den Speicher auszulagern. Eine gegenseitige Beeinflussung der Programmfäden durch Speicherzugriffe wird verhindert, indem die Speicherlatenz null beträgt, also in jedem Takt auf den Speicher zugegriffen werden kann. Die hohe Geschwindigkeit wird erkauft durch eine relativ kleine Größe von 64 KiBytes für den gemeinsamen Speicher für Code und Daten.

[1] http://www.xmos.com/, abgerufen am 1. August 2011

Die einfache RISC-Architektur mit 12 Registern ist auch als Mehrkernprozessor mit bis zu vier Kernen erhältlich und kann mit bis zu 500 MHz betrieben werden.

7.1.3 Dynamic Instruction Stream Computer

Eine andere Möglichkeit, die Anzahl der Programmfäden bei der Interleaved-Ausführung zu variieren, wurde 1991 von Daniel Nemirovsky vorgeschlagen [Nemirovsky 1991]. Sein *Dynamic Instruction Stream Computer (DISC)* wurde speziell für Echtzeitsysteme mit aperiodischen Unterbrechungen entwickelt. Anstatt die Programmfäden in der immer gleichen Reihenfolge auszuführen, bestimmt eine Scheduling-Tabelle, in welcher Reihenfolge die Programmfäden Instruktionen durch die Pipeline schicken dürfen.

Jeder Eintrag in der Scheduling-Tabelle steht für einen Takt und enthält die Nummer eines Programmfadens. Im jeweiligen Takt wird der entsprechende Programmfaden ausgeführt. Programmfäden, die öfter in der Tabelle auftauchen, werden dementsprechend häufiger ausgeführt. Nach 16 Einträgen ist das Ende der Tabelle erreicht und es wird wieder am Tabellenanfang begonnen.

Da in der Tabelle der gleiche Programmfaden auch in zwei aufeinander folgenden Einträgen vorkommen darf, ist nicht mehr garantiert, dass zwischen den Instruktionen keine Abhängigkeiten bestehen. Wird eine Abhängigkeiten erkannt, so wird die abhängige Instruktion nicht ausgeführt, stattdessen wird eine Instruktion des Programmfadens, der im nächsten Takt an der Reihe wäre, ausgeführt. Falls dies der gleiche Programmfaden ist oder er ebenfalls in Abhängigkeit zu einer in der Pipeline befindlichen Instruktion steht, wird der nächste Tabelleneintrag verwendet, solange bis eine nicht-abhängige Instruktion gefunden wird [Donalson 1993].

Wichtig für die harte Echtzeitfähigkeit ist es, dass durch das Vorziehen einer Instruktion die dazwischenliegenden Befehle nicht übersprungen werden, sondern an der ursprünglichen Stelle fortgefahren wird. Mit anderen Worten, die Scheduling-Tabelle wird niemals verschoben, es werden nur Instruktionen durch zusätzliche ersetzt. Dadurch wird sichergestellt, dass jedem Programmfaden mindestens die durch die Tabelle garantierten Takte zugewiesen werden und eine statische Laufzeitanalyse möglich ist.

Ein Problem stellen Speicherzugriffe über den externen Datenbus dar, da diese mehrere Takte in Anspruch nehmen können. Falls der Bus bereits belegt ist, wird ein weiterer zugreifender Programmfaden in einen Wartezustand versetzt und die verbleibenden Programmfäden werden entsprechend dem oben beschriebenen Verfahren häufiger ausgeführt. Falls mehr als ein Programmfaden auf den Bus wartet, wird die Reihenfolge gespeichert und der Bus später entsprechend vergeben. Dies ist wiederum wichtig für die harte Echtzeitfähigkeit, da durch dieses Verfahren garantiert werden kann, dass jeder der vier Programmfäden von jedem anderen Programmfaden nur maximal einmal, also insgesamt maximal dreimal, verzögert wird, bevor er seinen Zugriff durchführen kann.

7.1.4 Ubicom IP3023

Der Netzwerkprozessor *IP3023* [Ubicom 2003] von Ubicom[2] verwendet ebenfalls eine Scheduling-Tabelle wie der DISC. Der IP3023 ist bereits acht Jahre alt und wurde zwischenzeitlich durch mehrere Nachfolge-Prozessorfamilien (IP8100, IP8500) ersetzt, über deren Scheduling-Algorithmus jedoch kaum etwas bekannt ist. Da nicht anzunehmen ist, dass die Ablaufplanung verändert wurde, sind die folgenden Bemerkungen vermutlich auch für die aktuelle Produktlinie von Ubicom gültig.

Die Länge der Scheduling-Tabelle ist einstellbar, jedoch ist durch die Hardware eine Obergrenze von 64 Einträgen und 8 Programmfäden (12 beim IP8500) vorgegeben [Ubicom 2004]. Zusätzlich zu den Programmfäden für harte Echtzeit, die in der Scheduling-Tabelle vermerkt sind, gibt es noch weiche Echtzeitfäden, die nur dann ausgeführt werden, wenn eine Abhängigkeit bei einem Befehl eines harten Echtzeitfadens erkannt wird.

Im Unterschied zum DISC werden somit keine harten Echtzeit-Instruktionen vorgezogen. Dieses Verhalten stellt keinen Nachteil dar, da harte Echtzeitfäden nur ihre Frist einhalten müssen, wie weit der Abstand zur Frist letztlich ist, spielt keine Rolle.

Der Vorteil dieses Verfahrens liegt darin, dass die Rechenzeit sehr feinkörnig verteilt werden kann, wodurch die teilweise Überbrückung von Latenzen möglich ist (wenn beispielsweise ein Programmfaden nur jeden zweiten Takt ausgeführt wird, können die Latenzen halbiert werden). Insgesamt ist die Technik jedoch sehr statisch und die Aufteilung der Rechenzeit durch die kurze Runde relativ grob und unflexibel (alle Rechenzeitanteils müssen ein Vielfaches von $\frac{1}{64} \approx 1,6\%$ sein).

7.1.5 MIPS32 34K

Die amerikanische Firma MIPS Technologies[3] bietet ebenfalls eine echtzeitfähige mehrfädige Prozessorlinie an, die *MIPS32 34K* [MIPS 2010b] genannt wird. Sie implementiert eine sehr mächtige Mehrfädigkeits-Schnittstelle die sogenannte *MIPS Multithreading Application-Specific Extension (MIPS MT ASE)* [MIPS 2010a]. Darin ist festgelegt, dass der sogenannte *Policy Manager* jedem Programmfaden eine Priorität zuordnet. Der Programmfaden mit der höchsten Priorität wird ausgewählt und seine nächste Instruktion in der Pipeline ausgeführt. Diese Technik entspricht der Prioritätssteuerung und der Zuordnungsstufe des CarCore-Prozessors. Falls mehrere Programmfäden die gleiche Priorität haben, werden sie abwechselnd ausgeführt (engl. *round robin*).

Von MIPS werden drei Berechnungsmethoden für die Prioritäten angeboten: *Equal Priority* wechselt die Programmfäden der Reihe nach ab, *Fixed Priority* vergibt feste Prioritäten und bei *Weighted Round Robin* gibt es vier Gruppen, die jeweils $\frac{1}{15}$, $\frac{2}{15}$, $\frac{4}{15}$ und $\frac{8}{15}$ der Rechenzeit erhalten. Insbesondere die Equal-Priority-Methode, die *interleaved multithreading* bereitstellt, erfüllt auf den ersten Blick harte Echtzeitanforderungen.

[2] http://www.ubicom.com/, abgerufen am 1. August 2011
[3] http://www.mips.com/, abgerufen am 1. August 2011

Problematisch ist jedoch das Verhalten, falls einer der Programmfäden aufgrund von Abhängigkeiten nicht ausgeführt werden kann. In diesem Fall wird er einfach übersprungen und der nächste Programmfaden mit gleicher Priorität wird ausgeführt. Dadurch wird die Ausführungsreihenfolge der Programmfäden verändert und es kann zu zeitlichen Anomalien kommen, die dazu führen, dass die Rechenzeit weder gleichmäßig noch vorhersehbar unter den Programmfäden verteilt wird.

Dies ist zum Beispiel der Fall, wenn ein Programmfaden so optimiert wurde, dass keine Abhängigkeiten bestehen, wenn jeweils vier Takte zwischen der Ausführung zweier Instruktionen liegen. Er läuft optimal, wenn vier andere Programmfäden per *interleaved multithreading* zwischenzeitlich ausgeführt werden. Fällt ein dazwischenliegender Programmfaden aus, so besteht ein Abhängigkeit zwischen den Instruktionen des optimierten Programmfadens und er kann erst eine Runde später ausgeführt werden. Dadurch liegen jeweils sechs Takte zwischen der Ausführung zweier Instruktionen, der Programmfaden wird also erheblich langsamer ausgeführt, obwohl ihm eigentlich mehr Takte zugeordnet werden.

Beim DISC und beim Ubicom IP3023 kann diese Zeitanomalie nicht auftreten, da der Programmfaden immer an der gleichen Stelle innerhalb der festen Reihenfolge ausgeführt wird.

7.2 Innovative mehrfädige Architekturen für harte Echtzeit

Während kommerzielle Prozessoren ausschließlich die Interleaved-Ausführung zur mehrfädigen Ausführung harter Echtzeitprogramme verwenden, gibt es einige Forschungsarbeiten, die sich mit einer weitergehenden Verbindung von harter Echtzeit und Mehrfädigkeit befassen.

7.2.1 Komodo

Der *Komodo* Javaprozessor [Kreuzinger 2000] ist ein mehrfädiger Prozessor, der Java Bytecode [Lindholm 1999] direkt ausführen kann. Neben EDF- und FP-Scheduling bietet er den sogenannten *Guaranteed Percentage (GP)* Scheduling-Algorithmus [Kreuzinger 2000], bei dem die Rechenzeit in Intervalle von jeweils genau 100 Takten aufgeteilt wird. Jeder echtzeitfähige Programmfaden erhält einen bestimmten Prozentsatz der zur Verfügung stehenden Rechenzeit. Dieser Prozentsatz entspricht einer Anzahl Takte pro Intervall, in denen der jeweilige Programmfaden die höchste Priorität erhält.

Falls der Programmfaden mit der im entsprechenden Takt höchsten Priorität wegen Abhängigkeiten oder Latenzen blockiert ist, wird ein anderer Programmfaden ausgeführt. Dadurch ist es möglich, dass die Programmfäden mehr Rechenzeit als garantiert bekommen, bzw. Hintergrund-Programmfäden ohne garantierten Prozentsatz trotzdem

ausgeführt werden.

Der Komodo kann als Vorläufer des CarCore angesehen werden. Drei Einschränkungen des Komodo wurden im CarCore behoben:

(i) Durch den Kontextwechselmechanismus ist die Anzahl der Programmfäden nicht mehr auf die durch die Hardware vorgegebene Zahl beschränkt.

(ii) Anstatt nur einer Instruktion pro Takt können beim CarCore zwei Instruktionen simultan ausgeführt werden (simultane Mehrfädigkeit, SMT).

(iii) Die Rundenlänge ist nicht auf 100 festgelegt.

7.2.2 Real-Time Virtual Multiprocessor

Die Doktorarbeit *„Hard-Real-Time Multihreading: A Combined Microarchitectural and Scheduling Approach“* von Ali El-Haj-Mahmoud [El-Haj-Mahmoud 2006] ist nicht nur aufgrund des Titels sehr stark mit der vorliegenden Arbeit verwandt. Sie beschäftigt sich ebenfalls mit der Ablaufplanung in harten Echtzeitsystemen unter Verwendung eines SMT-Prozessors und empfiehlt dazu Veränderungen an der Mikroarchitektur eines superskalaren In-Order-Prozessors (Alpha 21164 [Edmondson 1995]). Im Unterschied zur vorliegenden Arbeit wird jedoch die Nutzung brachliegender Ressourcen durch Programmfäden mit geringeren Echtzeitanforderungen nicht unterstützt und es wurden nur Simulationen der Architektur durchgeführt, auf einen lauffähigen Prototypen wurde verzichtet.

Der in der Dissertation vorgestellte *Real-time Virtual Multiprocessor (RVMP)* [El-Haj-Mahmoud 2005] verwendet ein ähnliches Verfahren wie DTS oder WRR (vgl. Abschnitt 4.2.1.2) um die Rechenzeit zwischen den einzelnen Programmfäden innerhalb einer kurzen Runde zu verteilen. Zusätzlich können auch die Subpipelines zwischen den Programmfäden aufgeteilt werden, sodass zwei hart echtzeitfähige Programmfäden wirklich parallel in einem Takt ausgeführt werden. Jedem parallelen Programmfaden steht jedoch nur eine festgelegte Untermenge der Subpipelines zur Verfügung. Diese Untermengen dürfen sich nicht überschneiden.

Die Aufteilung der Subpipelines funktioniert gut, wenn identische Subpipelines vorhanden sind. Falls jedoch bestimmte Instruktionen ausschließlich in einer Subpipeline ausgeführt werden, so muss der Zugriff wiederum zeitlich abgewechselt werden. In diesem Fall kommt es somit zu einer Doppelung der Zeitschlitze oder Zeitschlitzen innerhalb von Zeitschlitzen.

Es ist fraglich, ob die feinkörnige Aufteilung der Subpipelines Sinn macht, denn identische Subpipelines sind nicht sehr häufig (bei TriCore gibt es beispielsweise keine) und der Hardware-Aufwand für zusätzliche Subpipelines ist aus Sicht der Echtzeitfähigkeit besser in einem separaten, isoliert analysierbaren Prozessorkern aufgehoben. Die feinkörnige zeitliche Aufteilung der Subpipelines innerhalb eines Rundenbruchteils wiederum verbraucht ebenfalls zusätzliche Hardware, die eingespart werden kann, wenn die Run-

denbruchteile einfach halbiert werden und dafür jeder Programmfaden exklusiven Zugriff hat.

Es gibt drei Gründe, die den Nutzen der feinkörnigen Aufteilung der Subpipelines in Frage stellen:

(i) Identische Subpipelines sind nur selten vorhanden(bei TriCore gibt es beispielsweise keine) und der Hardware-Aufwand für zusätzliche Subpipelines ist mit Blick auf die Echtzeitfähigkeit besser in einem separaten, isoliert analysierbaren Prozessorkern aufgehoben.

(ii) Die feinkörnige zeitliche Aufteilung der Subpipelines innerhalb eines Rundenbruchteils verbraucht zusätzliche Hardware, die eingespart werden kann, wenn die Rundenbruchteile einfach halbiert werden und dafür jeder Programmfaden exklusiven Zugriff hat.

(iii) Die Laufzeitanalyse eines Programmfadens hängt von den zur Verfügung stehenden Subpipelines ab. Deshalb muss die Laufzeitanalyse parametrierbar sein und nach jeder Veränderung der Konfiguration neu durchgeführt werden.

7.2.3 Virtual Simple Architecture

Einen völlig anderen Ansatz stellt die *Virtual Simple Architecture (VISA)* von Aravindh Anantaraman [Anantaraman 2003] dar. Ein hochperformanter Prozessor mit spekulativen Eigenschaften, wie dynamischer Sprungvorhersage, mehrstufigem Cache und Out-of-Order-Ausführung wird so modifiziert, dass diese Eigenschaften deaktiviert werden können. Nach der Abschaltung verhält sich der Prozessor wie ein einfädiger superskalarer Prozessor mit In-Order-Ausführung, für den eine Laufzeitanalyse durchführbar ist. Prinzipiell lässt sich dieses Verfahren auch auf die simultane Mehrfädigkeit anwenden, die Autoren empfehlen dies auch, da sie darin ein großes Potential sehen, es wurde von ihnen jedoch weder evaluiert noch verwirklicht.

Das Taskset wird auf diesem einfachen Modell basierend analysiert und die Fristen und Perioden dementsprechend berechnet. Ausgeführt wird das Taskset jedoch auf dem spekulativen Prozessor, ohne Abschaltungen. Im Normalfall ist dadurch die Ausführung deutlich schneller und die gesparte Zeit kann für zusätzliche Programmfäden oder die Senkung der Taktfrequenz verwendet werden.

Unter ungünstigen Umständen kann es jedoch zu einer längeren Ausführungszeit kommen, sodass die Fristen nicht eingehalten werden. Um ein derartiges Verhalten möglichst früh zu erkennen, werden zusätzliche Checkpoints im Programm eingefügt, an denen überprüft wird, ob dass Programm schnell genug ausgeführt wird, um die Fristen einzuhalten. Sobald die Ausführung einen Checkpoint zu spät erreicht, wird der Prozessor umkonfiguriert und alle spekulativen Einheiten abgeschalten. Dadurch ist die Ausführungszeit besser vorhersehbar und die WCET geringer.

Die Checkpoints und Fristen müssen so gewählt werden, dass beim Verpassen eines

Checkpoints noch genügend Zeit für die Ausführung des restlichen Programms ohne spekulative Elemente bleibt. Dieser Punkt ist die große Schwäche des Ansatzes, denn da man an einem Checkpoint nicht weiß, wie viel eines Programms bereits ausgeführt wurde, muss davon ausgegangen werden, dass der gesamte Code zwischen den letzten beiden Checkpoints noch einmal im nicht-spekulativen Modus ausgeführt werden muss. Das bedeutet, dass zum Zeitpunkt des ersten Checkpoints noch einmal die gesamte nicht-spekulative WCET zur Verfügung stehen muss, bevor die Frist erreicht wird. Infolgedessen müssen trotz einer im Durchschnitt schnelleren Ausführung die Fristen verlängert werden.

7.3 SMT Architekturen für weiche Echtzeit

Als Anfang der Nutzung von SMT-Prozessoren für die Echtzeitausführung können Arbeiten angesehen werden, bei denen versucht wurde, die Ausführungsgeschwindigkeit eines Vordergrund-Programmfadens möglichst wenig durch die gleichzeitige Ausführung eines Hintergrund-Programmfadens zu stören. Durch Bevorzugung des Vordergrundfadens beim Instruktionsholen kann die Verlangsamung des Vordergrundfadens auf 6% beschränkt werden, bei einer gleichzeitigen Steigerung des Gesamtdurchsatzes um 25% [Raasch 1999]. Werden auch alle anderen Ressourcen geteilt, kann die Verlangsamung sogar auf 3% im Vergleich zur alleinigen Ausführung gedrückt werden, der Hintergrundfaden erreicht dann sogar 60% [Dorai 2002].

Genauer kann die Ausführung des Vordergrundfadens kontrolliert werden, wenn die IPC (vgl. Abschnitt 2.3.5) des Programmfades gemessen und geregelt wird. Dadurch ist es möglich, von außen eine gewisse Anforderung (engl. *quality of service, QoS*) an den Programmfaden zu stellen. Da diese aber nicht zwangsläufig erfüllt werden handelt es sich dabei um eine weiche Echtzeitanforderung.

Beim *Predictable Performance (PP)* Verfahren von Francisco J. Cazorla [Cazorla 2004] werden in einer anfänglichen Sample-Phase die IPC ermittelt und in der folgenden Tune-Phase ein bestimmter Prozentsatz der ermittelten IPC eingeregelt. Um Schwankung der IPC zu erkennen, werden die längeren Tune-Phasen immer wieder durch kurze Sample-Phasen unterbrochen.

Im zu Grunde liegenden Out-of-Order SMT-Prozessor wird von allen Ressourcen ein gewisser Anteil exklusiv für den zu regelnden Programmfaden reserviert. Durch dynamische Veränderung dieses Anteils (insbesondere an der Fetch- und Issue-Bandbreite) kann die IPC beeinflusst werden. Der Regelalgorithmus ist jedoch relativ einfach, je nachdem, ob die Abweichung positiv oder negativ ist, wird der Ressourcenanteil um einen konstanten Wert verringert oder erhöht, weshalb die geregelte IPC relativ stark um den Zielwert oszilliert.

Einen besseren Regelalgorithmus, der schnell und stabil den gewünschten IPC-Wert einstellt, wird beim Ansatz von Uwe Brinkschulte [Brinkschulte 2005] verwendet. Er baut auf dem Guaranteed-Percentage-Scheduling des Komodo-Prozessors (Abschnitt 7.2.1)

auf und benötigt keine Sample-Phase. Der Prozentwert, der den Rechenzeitanteil des Programmfadens angibt, ist die Stellgröße, die durch einen PID-Regler [Föllinger 1994, S. 237] beeinflusst wird, um eine bestimmte IPC zu erreichen. Da zusätzlich zu dem geregelten Programmfaden noch weitere mit dem hart echtzeitfähigen GP-Scheduling ausgeführt werden können, unterstützt diese Variante des Komodo-Prozessors sogar gemischte Echtzeitanforderungen.

Die gleichzeitige Regelung der IPC mehrerer Programmfäden bietet der *Responsive Multithreaded (RMT) Processor* [Yamasaki 2007]. Er arbeitet nach dem gleichen Prinzip wie der PP-Mechanismus. Durch Beschränkung der Ressourcennutzung eines Programmfadens wird die IPC geregelt. Dies ist jedoch für jeden Programmfaden unabhängig möglich. Ein weiterer Unterschied ist, dass der RMT Prozessor nicht nur simuliert wurde, sondern tatsächlich hergestellt wurde.

Die Architektur des RMT Prozessors ist ähnlich aufgeteilt wie beim CarCore: die acht Hardware-Fäden des Out-of-Order-SMT-Prozessor werden durch einen speziellen Kontext-Cache erweitert, in dem die Kontexte von weiteren 32 Programmfäden Platz finden, die extrem schnell (4 Takte) in einen Hardware-Faden geladen werden können. Die Befehlsbereitstellung und die Zuordnung der Befehle wird wie beim CarCore durch Prioritäten der Hardware-Fäden geregelt, die durch eine zweite Schicht, die IPC-Steuerung verändert werden.

Mit Out-of-Order-SMT-Prozessoren stellt selbst die Messung der IPC ein Problem dar. Beim sogenannten *Per-Thread Cycle Accounting* [Eyerman 2010] wird versucht, durch zahlreiche Zähler innerhalb des SMT-Prozessorkerns die Ausführungszeit eines jeden Programmfadens einzeln zu ermitteln und durch Beeinflussung der Befehlsbereitstellung die Ausführung zu steuern.

Dazu wird die Gesamtausführungszeit in drei Komponenten geteilt, die eigentliche Ausführungszeit, in der Prozessor mit dem Programmfaden beschäftigt ist, die Fehlzeit, die durch Latenzen des Programmfadens verursacht wird und die Wartezeit, wenn ein Programmfaden nicht ausgeführt wird, weil ein anderer Programmfaden die benötigte Ressource belegt. Aufgrund der Out-of-Order-Ausführung kann die IPC jedoch selbst mit solch hohem Aufwand nicht genau ermittelt werden, die berechneten Werte weichen um circa 7% von den wirklichen Werten ab.

Alle vorgestellten Verfahren zur Kontrolle der IPC von Programmfäden in SMT-Prozessoren verbindet, dass sie die IPC nur indirekt beeinflussen, sie wird regelmäßig gemessen und bei Abweichungen wird die Ausführung des betroffenen Programmfadens entsprechend geregelt. Dadurch wird die IPC nur selten genau getroffen, sondern pendelt um den Zielwert. Im Gegensatz dazu wird beim CarCore eine IPC exakt erreicht, da die Instruktionen genau mitgezählt werden und beim Erreichen der IPC die Ausführung sofort eingestellt wird. Als weiterer positiver Aspekt, neben der größeren Genauigkeit, ist die Einsparung der aufwändigen Regelungslogik beim CarCore zu erwähnen.

7.4 Aperiodische Server

In Abschnitt 2.3.4 wurde bereits der Polling Server zur Verarbeitung aperiodischer Anfragen vorgestellt, in diesem Abschnitt werden verfeinerte Verfahren dargestellt, die dessen Probleme, die langen Antwortzeiten und die Verschwendung von Rechenzeit, wenn keine Anfrage anliegt, reduzieren. Trotz des erheblichen Aufwandes, der bei einigen der Verfahren betrieben wird, kann keines mehr Rechenzeit für die Verarbeitung aperiodischer Anfragen zur Verfügung stellen als die Verbindung von DTS- und PIQ-Scheduler, die in Abschnitt 4.2.2.6 beschrieben wurde.

Das liegt daran, dass die Hauptaufgabe eines Programmfadens, der als aperiodischer Servers dient, darin besteht, einen bestimmten Bruchteil der Rechenzeit zu reservieren. Im Unterschied zu anderen Programmfäden wird während der für den Server-Faden reservierten Rechenzeit nicht immer etwas ausgeführt, sondern nur dann, wenn eine aperiodische Anfrage vorliegt. Bei RM oder EDF-Schedulern ist diese Reservierung, ohne die anderen Programmfäden zu beeinflussen, nicht trivial. Beim DTS-Scheduling hingegen ist es sehr einfach möglich, da in jeder Runde die Zeit zwischen dem Ende des letzten DTS-Fadens und dem Beginn der nächsten Runde automatisch für die PIQ-Fäden und damit die aperiodischen Programmfäden zur Verfügung steht.

Die Antwortzeit einer aperiodischen Anfrage bei Verwendung von DTS und PIQ kann im Vergleich zu den folgenden Servern relativ hoch sein, da nur jeweils gegen Ende einer Runde Zeit für die exklusive Ausführung aperiodischer Programmfäden zur Verfügung steht. Dieser Nachteil wird jedoch durch die SMT-Ausführung ausgeglichen, da ab dem Zeitpunkt der aperiodischen Anfrage sofort mit der Ausführung im Hintergrund begonnen werden kann, wenngleich mit niedrigerer Priorität und geringerer Ausführungsgeschwindigkeit.

7.4.1 Bandbreitenerhaltende Server

Weit besser als ein Polling Server nutzen *bandbreitenerhaltende Server* die ihnen zugestandene Kapazität, indem sie die Ausführung von periodischen Prozessen vorziehen, wenn während der für den Server reservierten Zeit keine aperiodische Anfrage auftritt. Diese Zeit steht dem Server dann später innerhalb der Periode zur Verfügung. Dadurch wird die Antwortzeit reduziert und die Ausführung ist gleichmäßiger.

Eine mögliche Implementierung stellt der *Priority Exchange Server* [Lehoczky 1987; Sprunt 1988] dar, er gibt seine Priorität an einen Prozess mit niedrigerer Priorität weiter, solange keine aperiodische Anfrage auftritt. Beim *Deferrable Server* [Strosnider 1995] wird der Server erst aktiviert, wenn eine aperiodische Anfrage auftritt. In diesem Fall wird der entsprechende aperiodische Prozess solange ausgeführt, bis er terminiert oder die Kapazität des Servers verbraucht ist. Zu Beginn der nächsten Periode wird die Kapazität wieder aufgefüllt und ein unterbrochener aperiodischer Prozess kann weitergeführt werden oder eine neue Anfrage verarbeitet werden.

Das Verhalten des *Sporadic Server* [Sprunt 1989] unterscheidet sich nur durch den Zeitpunkt, zu dem die Kapazität wieder aufgefüllt wird. Hier wird nicht zu Beginn der nächsten Periode, sondern genau eine Periodenlänge nach dem Verbrauch der Kapazität wieder aufgefüllt, wodurch die Ausführung gleichmäßiger wird und Prozesse mit niedriger Priorität weniger beeinflusst werden.

Zwar unterscheiden sich die Vorgehensweisen der Algorithmen, im Ergebnis sind die Unterschiede aber so minimal, dass keiner als besser oder schlechter klassifiziert werden kann [Bernat 1999]. Eine kleine Ausnahme stellt der *Extended Priority Exchange* Algorithmus [Sprunt 1988] dar, da er – im Unterschied zu den anderen drei – den Zeitgewinn nutzen kann, der sich ergibt, wenn ein periodischer oder sporadischer Prozess nicht die volle WCET benötigt (vgl. Abschnitt 4.2.2.1).

7.4.2 Slack Stealing Server

Einen optimalen Algorithmus für Systeme mit RM-Scheduler stellt der *Slack Stealing* Algorithmus [Lehoczky 1992] dar: sobald Rechenzeit zur Verfügung steht, die nicht unbedingt benötigt wird, um die Fristen der harten Echtzeitprozesse einzuhalten, wird diese für die Ausführung aperiodischer Prozesse verwendet. Um diese Zeiten zu erkennen, wird vor Beginn der Ausführung die Hyperperiode der Prozesse (das kleinste gemeinsame Vielfache aller Perioden) analysiert und in einer Tabelle vermerkt, wann wie viel Zeit zur Verfügung steht.

Negativ ist anzumerken, dass weder der Zeitgewinn beim vorzeitigen Beenden eines periodischen Prozesses vor der WCET, noch ein über den Mindestabstand hinausgehender Abstand von sporadischen Prozessen für aperiodische Prozesse genutzt werden kann und die Größe der Tabelle schon ab 10 Prozessen problematisch werden kann [Davis 1993]. Diese Probleme werden durch den *Dynamic Slack Stealing* Algorithmus [Davis 1993], behoben, indem die freien Zeiten dynamisch berechnet werden, jedoch ist diese Berechnung für realistische Systeme inakzeptabel komplex.

7.4.3 Dynamische bandbreitenerhaltende Server

Die bisher aufgeführten Server bauen alle auf einen Scheduler mit festen Prioritäten auf. Mmit Blick auf eine möglichst hohe Auslastung durch periodische Prozesse ist es jedoch wünschenswert, einen EDF-Scheduler zu verwenden. Eine Möglichkeit besteht darin, die bekannten Verfahren für EDF-Scheduler zu adaptieren. Die Berechnung der Serverkapazität ist aufgrund der Optimalität sehr einfach: die dem Server zur Verfügung stehende Last L_S ergibt sich aus der Differenz der maximalen Auslastung von 1 und der Summe der periodischen Lasten. Multipliziert mit einer frei wählbaren Serverperiode T_S ergibt sich die Serverkapazität C_S wie folgt:

$$C_S = T_S \cdot L_S = T_S \cdot \left(1 - \sum_{i=1}^{N} L_i\right) = T_S \cdot \left(1 - \sum_{i=1}^{N} \frac{C_i}{T_i}\right) \tag{7.1}$$

Der *Deadline Deferable Server (DDS)* [Ghazalie 1995] imitiert das Verhalten eines Deferable Servers, indem er statt der höchsten Priorität eine Frist erhält, die seiner Periode entspricht. Tritt während der Periode eine aperiodische Anfrage auf, so wird die Kapazität des Server entsprechend der Laufzeit des Antwortprozesses reduziert. Zu Beginn einer neuen Periode wird die Kapazität des Servers jeweils wieder aufgefüllt.

Wird die Kapazität jeweils genau eine Periodenlänge nach ihrem Verbrauch wieder aufgefüllt, so spricht man vom *Deadline Sporadic Server (DSS)* [Ghazalie 1995]. Da dessen Implementierung relativ komplex ist, wurde der *Deadline Exchange Server (DXS)* [Ghazalie 1995] entwickelt, eine Vereinfachung bei der die Kapazität nach Befriedigung einer Anfrage auf null gesetzt wird (selbst wenn noch Kapazität übrig wäre) und ein Zeitpunkt berechnet wird, zu dem die Kapazität dann wieder auf den vollen Wert gesetzt werden kann.

Wie schon im Fall mit festen Prioritäten unterscheiden sich die drei dynamischen bandbreitenerhaltenden Varianten kaum, DDS hat leichte Vorteile bei kurzen Perioden, DSS und DXS bei längeren Perioden, wobei DXS deutlich geringerem Aufwand implementiert werden kann. [Ghazalie 1995]

Beim *Dynamic Priority Exchange (DPE)* [Spuri 1996] erhält jeder Prozess, nicht nur der Server, eine dynamische Kapazität, die zu Beginn null beträgt. Die Anfangskapazität des Server entspricht der Rechenzeit, die bei voller Auslastung durch die periodischen Prozesse während seiner Periode übrig bleibt. Solange keine aperiodische Anfrage anliegt, aber der Server an der Reihe wäre, wird ein anderer Prozess vorgezogen. Im Gegenzug zu dessen Ausführung wird seine Kapazität um die entsprechende Ausführungszeit erhöht (während die Kapazität des Servers weiter sinkt). Dadurch muss er die Zeit, um die er vorgezogen wurde, später wieder zur Ausführung von aperiodischen Prozessen zur Verfügung stellen.

Tritt nun eine aperiodische Anfrage auf und es gibt einen Prozess, dessen Kapazität größer als null ist, so wird diese Kapazität verbraucht, bis der aperiodische Prozess beendet ist oder alle Kapazitäten bei null sind. Der Vorteil dieses Verfahrens liegt darin, dass ein Prozess, der vor seiner eigentlichen WCET endet, die verbleibende Zeit zu seiner Kapazität addieren kann, um sie so für aperiodische Anfragen zu nutzen.

Das Verfahren kann weiter verbessert werden, indem die Kapazität des Servers nicht nur bei jeder neuen Runde neu aufgefüllt wird, sondern eine Periodenlänge nachdem sie verbraucht wurde, wie bei DXS und DSS. Der resultierende Algorithmus wird *Improved Priority Exchange (IPE)* [Spuri 1996] genannt und erreicht fast optimale Antwortzeiten. Die Zeitpunkte und Mengen der Kapazitätsauffüllung müssen offline, das heißt vor der eigentlichen Ausführung des Systems berechnet werden.

7.4.4 Total Bandwidth Server

Optimale Antwortzeiten können durch den *Earliest Deadline as Late as possible (EDL)* Algorithmus [Chetto 1989] erreicht werden. Wenn aperiodische Anfragen anliegen, wer-

den die periodischen Prozesse so lange wie möglich verzögert, erst im letzten Moment, bevor sie ihre Frist verletzen würden, werden sie gestartet. Liegen keine aperiodischen Anfragen vor, so werden die periodischen Prozesse früher gestartet, um später mehr Zeit für aperiodische Anfragen zu haben. Das Verfahren ist zwar optimal, die Berechnungen sind aber zu komplex, um wirklich implementiert zu werden.

Zwar erreicht IPE fast die Optimalität von EDL, die Implementierung ist jedoch nur unwesentlich einfacher. Die dynamischen bandbreitenerhaltenden Server (DDS, DSS, DXS, DPE) hingegen sind zwar realisierbar, haben jedoch deutlich höhere Antwortzeiten, vor allem bei langen Perioden. Einen Ausweg bietet ein drittes Verfahren, dessen Implementierung einfacher als die der dynamisch bandbreitenerhaltenden Server ist und welches trotzdem fast optimale Antwortzeiten erreicht, der sogenannte *Total Bandwidth Server (TBS)* [Spuri 1996].

Wie bei den bisherigen dynamischen Servern wird dem Server eine gewisse Last L_S garantiert, jedoch gibt es weder eine feste Periode noch eine Kapazität, stattdessen wird bei jeder aperiodischen Anfrage eine individuelle Frist berechnet. Sie ergibt sich aus der WCET des aperiodischen Prozesses multipliziert mit der Serverlast L_S. Der aperiodische Prozess erhält damit eine minimale Frist, belegt aber auch die komplette Serverbrandbreite bis zur Frist. Falls bis zu dieser Frist eine weitere aperiodische Anfrage auftritt, muss deren Bearbeitung nach hinten verschoben werden, das heißt, zu ihrer Frist muss noch die Dauer bis zum Ende der vorherigen Frist addiert werden.

7.5 Hardware Scheduler

Bisher wurden Architekturen vorgestellt, die aufgrund ihres Aufbaus eine echtzeitfähige Ablaufplanung erlauben. Die innerhalb dieser Prozessoren implementierten Scheduling-Algorithmen sind allerdings relativ einfach, sodass sie auf einen zusätzlichen Software-Scheduler angewiesen sind. Sie entsprechen in etwa der ersten Stufe des CarCore Schedulers. In der zweiten Ebene des Schedulers werden komplexere Scheduling-Algorithmen per Hardware bereitgestellt. Hierzu gibt es ebenfalls zahlreiche verwandte Arbeiten, die jedoch auf einfachen einfädigen Prozessoren aufbauen, keiner der folgenden Ansätze unterstützt Mehrfädigkeit.

7.5.1 Koprozessoren

Eine Möglichkeit, den Scheduling-Overhead des Betriebssystems zu reduzieren, ist es, die rechenintensive Ermittlung des nächsten auszuführenden Programmfadens in einen Koprozessor auszulagern. Allerdings ist die maximale Anzahl der Programmfäden dann durch die Hardware vorgegeben.

Ein derartiger Koprozessor wurde von Jens Hildebrandt vorgestellt [Hildebrandt 1999]. Er verwendet den sogenannten *Enhanced Least Laxity First (ELLF)* Algorithmus, eine Weiterentwicklung des *Least Laxity First (LLF)* Algorithmus [Mok 1983, S. 38ff], der das

	ELLF	RTM	CHS	RTU	hthreads	Spring	Saez
Algorithmen	ELLF	FP	FP, RM, EDF	FP	FP, RR, FIFO	FP, EDF	EDF
Prioritätsstufen	2^{16}	256		8	128	16	
Multiprozessor	nein	nein	nein	ja	nein	ja	nein
Programmfäden	32	64	16	64	256(BRAM)	8	
Prototyp	FPGA	nein	FPGA	ASIC	FPGA	ASIC	nein

Tab. 7.1: Eigenschaften von Koprozessoren für die Ablaufplanung

ständige Ein- und Auslagern von Programmfäden reduziert und trotzdem eine optimale Auslastung von 100% wie EDF erreicht.

Der *Real-Time Task Manager (RTM)* [Kohout 2003] unterstützt zwar nur eine Ablaufplanung mit festen Prioritäten, kann dafür aber die Ausführungszeit des jeweils aktiven Programmfadens kontrollieren und einzelne Programmfäden schlafen legen, wenn sie auf eine Ressource oder ein anderes Ereignis warten. Werden die herkömmlichen Software-Routinen in einem Echtzeitbetriebssystem durch Aufrufe des RTM ersetzt, so wird der Overhead durch das Betriebssystem um 90% reduziert [Kohout 2003].

Zusätzlich zum FP-Scheduling unterstützt der *Configurable Hardware Scheduler* von Pramote Kuacharoen [Kuacharoen 2003] auch RM- und EDF-Scheduling. Ein weiterer Vorteil des CHS ist es, dass die Rahmendaten des Prozessors nicht fest vorgegeben sind, vielmehr kann die Anzahl der Programmfäden, die Anzahl externer Interrupts und die Zeitauflösung konfiguriert werden. Aus diesen Angaben wird dann Programmcode generiert, mit dem ein FPGA programmiert werden kann. Diese Parameter können jedoch nicht zur Laufzeit verändert werden, nach der Generierung stehen sie fest.

Die *Real Time Unit (RTU)* von Lennart Lindh [Adomat 1996] ist ein Koprozessor, der in seiner Grundversion nur drei elementare Operationen unterstützt: einen Prozess anlegen, ihn beenden und ihn für eine bestimmte Zeit schlafen legen. Zusammen mit einem einfachen RISC-Prozessor ergibt sich das *FASTCHART*-Projekt [Lindh 1991], ein Prozessorsystem mit streng deterministischem Zeitverhalten. Aufgrund der damaligen (1991) Ressourcen-Begrenzung unterstützte der FPGA-Prototyp nur acht Prozesse und zwei Prioritätsstufen.

Im nachfolgenden *FASTHARD*-Projekt [Lindh 1992] wurden Scheduling-Koprozessor und Hauptprozessor entkoppelt, das heißt, die RTU kann mit einem beliebigen Prozessor verbunden werden und bietet 64 Prozesse und acht Prioritätsstufen. Zusätzlich zu den elementaren drei Operationen können Interrupts verarbeitet, Prozesse periodisch gestartet und synchronisiert werden. In der letzten Version kann die RTU mit bis zu drei Prozessoren verbunden werden und kommuniziert über externe Register mit ihnen [Adomat 1996].

Der Spring Kernel [Stankovic 1989] ist ein echtzeitfähiges Betriebssystem für Multipro-

zessorsysteme mit verteilten Ressourcen. Für derartige Systeme gibt es keinen optimalen Scheduling-Algorithmus, weshalb im Spring Kernel heuristische Scheduling-Algorithmen verwendet werden. Aus Performance-Gründen werden die Heuristiken durch einen speziellen Koprozessor, den *Spring Scheduling Co-Processor (SSCoP)* [Burleson 1999] berechnet. Er bildet zusammen mit einem oder mehreren Anwendungsprozessoren und optional weiteren Ressourcen einen Spring-Knoten, mehrere dieser Knoten bilden ein vollständiges System.

Der SSCoP hat Informationen über alle möglichen Prozesse und versucht daraus einen gültigen Ablaufplan zu erstellen, ohne einen Prozess unterbrechen zu können. Falls er einen gültigen Ablaufplan findet, so kann jedem Prozess garantiert werden, dass er seine zeitlichen Anforderungen erfüllt. Neue Prozesse werden nur hinzugefügt, falls wieder ein gültiger Ablaufplan gefunden werden kann.

Die Ablaufplanung des Spring Kernels ist speziell auf dessen hardwarel und die heuristische Ablaufplanung in Multiprozessorsystemen ausgelegt und daher nur begrenzt mit klassischen Echtzeitsystemen vergleichbar, jedoch kann die Hardware des SSCoP auch so programmiert werden, dass sie das EDF-Scheduling für einen einzeln angeschlossenen Anwendungsprozessor übernimmt.

7.5.2 Andere Ansätze

Im *HybridThreads (hthreads)* Projekt können Programmfäden sowohl als Software, als auch als programmierte Logik in einem FPGA realisiert werden. Durch eine einheitliche Schnittstelle merkt der Programmierer nicht, auf welche Weise bestimmte Funktionen realisiert wurden. An Scheduling-Algorithmen stehen ein einfacher FIFO-Algorithmus, der die Programmfäden in der Reihenfolge ihrer Erzeugung ausführt, ein Round-Robin-Algorithmus und ein echtzeitfähiger Algorithmus mit festen Prioritäten zur Auswahl. Die Verwaltung der Programmfäden wird vollständig durch Hardware-Komponenten erledigt, die durch eine POSIX-kompatible Schnittstelle [POSIX 2008] angesprochen werden.

Besonders interessant ist die Speicherung der Scheduling-Daten der Programmfäden, da nicht ein systolisches Array [Moon 2000], wie bei den meisten anderen Ansätzen verwendet wird, sondern ein hierarchisches Verfahren, das auf normalen Speicherblöcken basiert. Dadurch ist es relativ einfach und mit geringem Hardware-Aufwand möglich, die Anzahl der Programmfäden über die im Prototyp verwendete Anzahl von 256 hinaus zu erhöhen.

Bisher wurde in keinem kommerziellen Prozessor ein Mechanismus integriert, der es erlaubt aperiodische Anfragen neben hart echtzeitfähigen Programmfäden auszuführen. Auch im akademischen Bereich findet sich nur eine einzige Arbeit, die sich mit der konkreten Implementierung eines aperiodischen Servers beschäftigt. Sergi Sáez hat bereits im Jahre 1999 eine Architektur vorgestellt [Sáez 1999], die einen Earliest-Deadline-First-Scheduler (vgl. Abschnitt 2.3.2) mit einem Dynamic-Slack-Stealer (vgl. Abschnitt 7.4.2)

kombiniert. Allerdings wird die Architektur nur detailliert beschrieben und erklärt, wie eine Hardware-Realisierung aussehen könnte, es wurden weder ein Simulator noch ein lauffähiger Prototyp entwickelt.

8 Schlussfolgerungen

Zum Abschluss der Arbeit werden die Ergebnisse noch einmal zusammengefasst, Schwächen analysiert und zukünftige Weiterentwicklungen vorgeschlagen.

8.1 Zusammenfassung

Es ist möglich, einen simultan mehrfädigen Prozessor so zu erweitern, dass ein einzelner Programmfaden von den anderen, gleichzeitig ausgeführten Programmfäden unter keinen Umständen beeinflusst wird. Dadurch ist eine statische Laufzeitanalyse möglich, mit deren Hilfe eine WCET berechnet werden kann, wodurch wiederum harte Echtzeitanforderungen erfüllt werden können. Auch mehrere hart echtzeitfähige Programmfäden sind möglich, sie können jedoch nicht gleichzeitig ausgeführt werden, sondern müssen nacheinander ausgeführt werden, wie bei der klassischen Ablaufplanung für einfädige Prozessoren.

Prinzipiell kann für diese Ablaufplanung jeder beliebige Uniprozessor-Scheduling-Algorithmus verwendet werden. Das hier vorgestellte DTS-Scheduling auf Basis von Zeitschlitzen hat jedoch eine geringere Hardware-Komplexität als etablierte Algorithmen, obwohl es eine optimale Prozessorauslastung garantiert. Ein weiterer Vorteil ist die Möglichkeit, das DTS-Scheduling mit einem hardwarebasierenden Kontextwechselmechanismus zu kombinieren, wodurch die Anzahl der quasi gleichzeitig ausgeführten Programmfäden nur noch durch den zur Verfügung stehenden Speicher begrenzt wird.

Um jedoch in vollem Maße von der erhöhten Prozessorauslastung durch die mehrfädige Programmausführung zu profitieren, müssen weitere Programmfäden mit geringeren Echtzeitanforderungen gleichzeitig mit den hart echtzeitfähigen Programmfäden ausgeführt werden, was gemischtkritische Ausführung (engl. *mixed criticality*) genannt wird. Weiche Echtzeitanforderungen werden durch den PIQ-Algorithmus erfüllt, der pro Zeitintervall die Ausführung einer bestimmten Anzahl von Instruktionen garantiert. Es gibt zwar zahlreiche Veröffentlichung zu derartigen Mechanismen zur Regelung der IPC, die hier vorgeschlagene unterscheidet sich jedoch durch ihre Einfachheit und das exakte Erreichen der Ziel-IPC, da auf eine indirekte Regelung verzichtet wird.

Die verbleibende Rechenzeit, die nicht durch echtzeitfähige Programmfäden genutzt werden kann, wird entweder gleichmäßig auf eine beliebige Anzahl von Best-Effort-Programmfäden verteilt oder für die Verarbeitung von aperiodischen Anfragen (Interrupts) verwendet. Bei letzterem ist sowohl eine möglichst schnelle Ausführung, um die Antwortzeit zu minimieren, möglich, als auch eine Abschätzung der zur Verfügung ste-

henden Rechenzeit, um Anfragen gegebenenfalls ablehnen zu können.

Anhand des CarCore-Prototyps konnte gezeigt werden, dass die mehrfädigen Erweiterungen und der Hardware-Scheduler mit einem vertretbaren Hardware-Aufwand realisiert werden können. Zwar wird die einfädige Programmausführung durch die Maßnahmen zur Isolation verlangsamt, dieser Nachteil wird jedoch durch einen erhöhten Gesamtdurchsatz des Prozessors mehr als aufgewogen. Insbesondere die überlappende Ausführung von mehreren Programmfäden mit weichen Echtzeitanforderungen stellt eine gute Möglichkeit dar, den Prozessor sehr viel höher als bei einfädiger Ausführung auszulasten.

Insgesamt besticht die beschriebene Architektur durch ihre Flexibilität: sie ist auf jeden beliebigen einfädigen In-Order-Prozessor anwendbar, die Anzahl der Programmfäden ist unbegrenzt und es wird ein weites Spektrum an Echtzeitanforderungen durch passende Algorithmen unterstützt.

8.2 Schwächen

Ein Schwachpunkt des CarCore-Prototyps ist die deutlich geringere Ausführungsgeschwindigkeit im Vergleich zum TriCore, welche zum größten Teil auf die äußerst langsame Implementierung der Unterprogrammaufrufe zurückzuführen ist. Die im TriCore realisierte Hardware-Beschleunigung lässt sich nur schwer auf ein echtzeitfähiges, mehrfädiges System übertragen. Eine Möglichkeit wäre es aber, den Kontextwechsel bei Unterprogrammaufrufen und den Kontextwechsel bei der Auslagerung von Programmfäden zusammenzulegen und in einem separaten Hardware-Modul zu erledigen, das der Echtzeitfähigkeit Rechnung trägt. Dabei sind jedoch sehr viele Aspekte zu beachten, die weit über den Rahmen dieser Arbeit hinausgehen.

Unbefriedigend ist ebenfalls die Tatsache, dass hart echtzeitfähige Programmfäden aus prinzipiellen Gründen nicht überlappend ausgeführt werden können. Die hier vorgeschlagene Lösung mit Zeitschlitzen löst zwar das Problem der gleichzeitigen Ausführung, der Hardware-Aufwand ist jedoch relativ hoch. Viel einfacher ist die Verhinderung gegenseitiger Einflussnahme von hart echtzeitfähigen Programmfäden, wenn sie auf unterschiedliche Rechenkerne eines Mehrkernprozessoren verteilt werden. Dadurch sind sie auf natürliche Weise isoliert und es kommt nur noch in der gemeinsamen Speicherschnittstelle zu Konflikten und gegenseitiger Beeinflussung.

8.3 Ausblick

Da die Anzahl von Rechenkernen pro Prozessor ohne Zweifel weiter steigen wird, scheint die Verteilung von hart echtzeitfähigen Programmfäden auf mehrere Rechenkerne sinnvoller als das Scheduling mehrere Programmfäden auf einem Kern [Schoeberl 2009]. Genau diesen Ansatz verfolgte das MERASA-Projekt [Ungerer 2010], in dem ebenfalls ein auf dem TriCore-Befehlssatz basierender Prozessor für Echtzeitanwendungen entwickelt

wurde. Im Unterschied zum CarCore wurde jedoch nicht nur ein Rechenkern mit einem komplexer Scheduler entwickelt, sondern ein Mehrkernprozessor, der pro Rechenkern nur einen hart echtzeitfähigen Programmfaden ausführen kann.

Allerdings müssen bei der Mehrkernlösung mindestens so viele Rechenkerne wie Programmfäden zur Verfügung stehen. Anwendungen mit einer hohen Zahl von Programmfäden mit einer kurzen Ausführungszeit sind dabei nicht möglich. Als Alternative bietet sich Interleaved Multithreading an, bei dem der Rechenzeitanteil pro Programmfaden zwar stark begrenzt ist, der Prozessor aber sehr intensiv ausgelastet werden kann. Ein Mehrkernprozessor, bei dem kurze Programmfäden mit harten Echtzeitanforderungen auf einem oder mehreren Rechenkernen mit Hilfe von Interleaved Multithreading gebündelt werden und die restlichen Rechenkerne nur jeweils einen Programmfaden mit harten Echtzeitanforderungen ausführen, ist vermutlich die beste Lösung im Hinblick auf maximalen Durchsatz von hart echtzeitfähigen Programmfäden.

Die Stärken der simultan mehrfädige Ausführung kommt jedoch dann zur Geltung, sobald Programmfäden mit unterschiedlichen Echtzeitanforderungen auf dem gleichen Prozessor ausgeführt werden sollen. Denn keine andere Mikroarchitektur ist in der Lage, die zur Verfügung stehenden Prozessorressourcen effektiver zu nutzen. Bei Verwendung einer In-Order-Zuordnung kann der zusätzliche Hardware-Aufwand für die Ablaufplanung relativ klein gehalten werden, so dass das Verhältnis von Leistung zu Chipfläche optimiert wird. Da im allgemeinen eine kleinere Chipfläche auch mit einem geringeren Energieverbrauch einhergeht, ist dies ein viel versprechender Ansatz um Leistungsaufnahme und Wärmeentwicklung des Chips zu minimieren.

Von Seiten der Industrie wird die Bündelung einer größeren Zahl von Programmfäden auf einem Prozessor nicht nur aus Gründen des Energiesparens verfolgt, auch andere Kosten für die Herstellung und Wartung lassen sich dadurch reduzieren. Deshalb wird die gleichzeitige Ausführung von unterschiedlich kritischen Programmfäden auch als ein grundlegendes Konzept für zukünftige eingebettete Systeme gesehen, die noch enger mit ihren Umwelt in Kontakt stehen und diese beeinflussen, den sogenannten *Cyber Physical Systems* [Baruah 2010].

In der industriellen Praxis muss in den wenigsten Fällen wirklich eine teure statische Laufzeitanalyse durchgeführt werden. Meist ist sie nur nötig, wenn es um extrem sicherheitskritische Aspekte geht, die eine Zertifizierung erfordern. In vielen Fällen begnügen sich die Ingenieure im industriellen Umfeld damit, die maximal beobachtete Laufzeit bei einer großen Zahl von Testläufen zu ermitteln und dieser Zeit noch ein Sicherheitspolster zuzuschlagen. Ist diese Vorgehensweise zulässig, so kann der PIQ- statt dem DTS-Algorithmus verwendet werden, wodurch ein Überlappen der Ausführungszeiten möglich ist und der Durchsatz des Prozessors erheblich gesteigert werden kann.

Literaturverzeichnis

[Adomat 1996] Joakim Adomat, Johan Furunäs, Lennart Lindh und Johan Starner. „Real-Time Kernel in Hardware RTU: A Step Towards Deterministic and High-Performance Real-Time Systems“. In: *Proceedings of the 8th Euromicro Conference on Real-Time Systems (ECRTS '96)*. 1996, Seite 0164. ISBN: 0-8186-7496-2. DOI: 10.1109/EMWRTS.1996.557849.

[Agarwal 1992] Anant Agarwal. „Performance Tradeoffs in Multithreaded Processors“. In: *IEEE Transactions on Parallel and Distributed Systems* 3.5 (Sep. 1992), Seiten 525 –539. DOI: 10.1109/71.159037.

[Anantaraman 2003] Aravindh Anantaraman, Kiran Seth, Kaustubh Patil, Eric Rotenberg und Frank Mueller. „Virtual Simple Architecture (VISA): Exceeding the Complexity Limit in Safe Real-Time Systems“. In: *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA '03)*. 2003, Seiten 350 –361. ISBN: 0-7695-1945-8. DOI: 10.1145/859618.859659.

[Barre 2008a] Jonathan Barre, Christine Rochange und Pascal Sainrat. „A Predictable Simultaneous Multithreading Scheme for Hard Real-Time“. In: *Proceedings of the 21st International Conference on Architecture of Computing Systems (ARCS '08)*. 2008, Seiten 161 –172. DOI: 10.1007/978-3-540-78153-0_13.

[Barre 2008b] Jonathan Barre, Christine Rochange und Pascal Sainrat. „An Architecture for the Simultaneous Execution of Hard Real-Time Threads“. In: *Proceedings of the Embedded Computer Systems: Architectures, Modeling, and Simulation, 2008. SAMOS 2008. International Conference on*. Juli 2008, Seiten 18 –24. DOI: 10.1109/ICSAMOS.2008.4664842.

[Barhorst 2009] James Barhorst, Todd Belote, Pam Binns, Jon Hoffman, James Paunicka, Prakash Sarathy, John Scoredos, Peter Stanfill, Douglas Stuart und Russell Urzi. *A Research Agenda for Mixed-Criticality Systems*. White Paper. Cyber Physical Systems Week 2009 Workshop on Mixed Criticality. San Francisco, CA, USA, Apr. 2009. URL: http://www.cse.wustl.edu/~cdgill/CPSWEEK09_MCAR/.

[Baruah 2010] Sanjoy Baruah, Haohan Li und Leen Stougie. „Towards the Design of Certifiable Mixed-criticality Systems“. In: *Proceedings of the 16th Real-Time and Embedded Technology and Applications Symposium (RTAS '10)*. 2010, Seiten 13 –22. ISBN: 978-0-7695-4001-6. DOI: 10.1109/RTAS.2010.10.

[Bernat 1999] Guillem Bernat und Alan Burns. „New Results on Fixed Priority Aperiodic Servers“. In: *Proceedings of the 20th Real-Time Systems Symposium (RTSS '99)*. 1999, Seite 68. ISBN: 0-7695-0475-2.

[Boothe 1992] Bob Boothe und Abhiram Ranade. „Improved multithreading techniques for hiding communication latency in multiprocessors“. In: *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA '92)*. 1992, Seiten 214 –223. ISBN: 0-89791-509-7. DOI: 10.1145/139669.139729.

[Brinkschulte 2005] Uwe Brinkschulte und Mathias Pacher. „Implementing Control Algorithms Within a Multithreaded Java Microcontroller“. In: *Proceedings of the 18th International Conference on Architecture of Computing Systems (ARCS '05)*. 2005, Seiten 33 –49. DOI: 10.1007/978-3-540-31967-2_3.

[Brinkschulte 2010] Uwe Brinkschulte und Theo Ungerer. *Mikrocontroller und Mikroprozessoren*. 3. Auflage. 2010. ISBN: 978-3-642-05397-9. DOI: 10.1007/978-3-642-05398-6.

[Burleson 1999] Wayne Burleson, Jason Ko, Douglas Niehaus, Krithi Ramamritham, John A. Stankovic, Gary Wallace und Charles Weems. „The spring scheduling coprocessor: a scheduling accelerator“. In: *IEEE Transactions on Very Large Scale Integrated Systems* 7.1 (März 1999), Seiten 38 –47. DOI: 10.1109/92.748199.

[Buttazzo 2004] Giorgio Buttazzo. *Hard Real-Time Computing Systems*. 2. Auflage. 2004. ISBN: 978-0-387-23137-2.

[Buttazzo 2005] Giorgio Buttazzo. „Rate Monotonic vs. EDF: Judgment Day“. In: *Real-Time Systems* 29.1 (2005), Seiten 5 –26. DOI: 10.1023/B:TIME.0000048932.30002.d9.

[Cazorla 2004] Francisco J. Cazorla, Peter M.W. Knijnenburg, Rizos Sakellariou, Enrique Fernández, Alex Ramirez und Mateo Valero. „Predictable Performance in SMT Processors“. In: *Proceedings of the 1st Conference on Computing Frontiers*. 2004, Seiten 433 –443. ISBN: 1-58113-741-9. DOI: 10.1145/977091.977152.

[Chetto 1989] Houssine Chetto und Maryline Chetto. „Some Results of the Earliest Deadline Scheduling Algorithm“. In: *IEEE Transactions on Software Engineering* 15.10 (1989), Seiten 1261 –1269. DOI: 10.1109/TSE.1989.559777.

[Davis 1993] Robert I. Davis, Ken W. Tindell und Alan Burns. „Scheduling Slack Time in Fixed Priority Pre-emptive Systems“. In: *Proceedings of the 14th Real-Time Systems Symposium (RTSS '93)*. 1993, Seiten 222 – 231.

[Donalson 1993] Douglas Donalson, Mauricio Serrano, Roger Wood und Mario Nemirovsky. „DISC: dynamic instruction stream computer-an evaluation of performance“. In: *Proceedings of the 26th Hawaii International Conference on System Sciences*. Jan. 1993, 448 –456 vol.1. DOI: 10.1109/HICSS.1993.270620.

[Dorai 2002] Gautham K. Dorai und Donald Yeung. „Transparent Threads: Resource Sharing in SMT Processors for High Single-thread Performance“. In: *Proceedings of the 11th Annual International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Sep. 2002, Seiten 30 –41. ISBN: 0-7695-1620-3. DOI: 10.1109/PACT.2002.1105971.

[Edmondson 1995] John H. Edmondson, Paul Rubinfeld, Ronald Preston und Vidya Rajagopalan. „Superscalar Instruction Execution in the 21164 Alpha Microprocessor“. In: *IEEE Micro* 15.2 (Apr. 1995), Seiten 33 –43. DOI: 10.1109/40.372349.

[EEMBC 2000] *AutoBench 1.1 Software Benchmark Data Book*. Embedded Microprocessor Benchmark Consortium (EEMBC). URL: http://www.eembc.com/techLit/datasheets/autobench_db.pdf.

[El-Haj-Mahmoud 2005] Ali El-Haj-Mahmoud, Ahmed S. AL-Zawawi, Aravindh Anantaraman und Eric Rotenberg. „Virtual Multiprocessor: An Analyzable, High-Performance Architecture for Real-Time Computing“. In: *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '05)*. 2005, Seiten 213 –224. ISBN: 1-59593-149-X. DOI: 10.1145/1086297.1086326.

[El-Haj-Mahmoud 2006] Ali Ahmad El-Haj-Mahmoud. „Hard-Real-Time Multihreading: A Combined Microarchitectural and Scheduling Approach“. Dissertation. North Carolina State University, 2006.

[Eyerman 2010] Stijn Eyerman und Lieven Eeckhout. „Per-Thread Cycle Accounting“. In: *IEEE Micro* 30.1 (Jan. 2010), Seiten 71 –80. DOI: 10.1109/MM.2010.23.

[Fineberg 1967] Mark S. Fineberg und Omri Serlin. „Multiprogramming for hybrid computation“. In: *Proceedings of the AFIPS Fall Joint Computer Conference*. Nov. 1967, Seiten 1 –13. DOI: 10.1145/1465611.1465613.

[Föllinger 1994] Otto Föllinger. *Regelungstechnik, Einführung in die Methoden und ihre Anwendung*. 8. Auflage. 1994. ISBN: 3-7785-2336-8.

[Gerosa 2008] Gianfranco Gerosa, Steve Curtis, Mike D'Addeo, Bo Jiang, Belliappa Kuttanna, Feroze Merchant, Binta Patel, Mohammed Taufique und Haytham Samarchi. „A Sub-1W to 2W Low-Power IA Processor for Mobile Internet Devices and Ultra-Mobile PCs in 45nm Hi-M Metal Gate CMOS". In: *Proceedings of the International Solid-State Circuits Conference (ISSCC 2008)*. Feb. 2008, Seiten 256 –611. DOI: 10.1109/ISSCC.2008.4523154.

[Ghazalie 1995] T. M. Ghazalie und Theodore P. Baker. „Aperiodic servers in a deadline scheduling environment". In: *Real-Time Systems* 9.1 (Juli 1995), Seiten 31 –67. DOI: 10.1007/BF01094172.

[Goossens 1996] Bernard Goossens und Duc Thang Vu. „On-Chip Multiprocessing". In: *Proceedings of the 2nd International Euro-Par Conference on Parallel Processing (Euro-Par '96)*. 1996, Seiten 789 –796. DOI: 10.1007/BFb0024778.

[Gustafsson 2010] Jan Gustafsson, Adam Betts, Andreas Ermedahl und Björn Lisper. „The Mälardalen WCET Benchmarks: Past, Present And Future". In: *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Juli 2010, Seiten 136 –146. DOI: 10.4230/OASIcs.WCET.2010.136.

[Heckmann 2003] Reinhold Heckmann, Marc Langenbach, Stephan Thesing und Reinhard Wilhelm. „The influence of processor architecture on the design and the results of WCET tools". In: *Proceedings of the IEEE* 91.7 (Juli 2003), Seiten 1038 –1054. ISSN: 0018-9219. DOI: 10.1109/JPROC.2003.814618.

[Hildebrandt 1999] Jens Hildebrandt, Frank Golatowski und Dirk Timmermann. „Scheduling Coprocessor for Enhanced Least-Laxity-First Scheduling in Hard Real-Time Systems". In: *Proceedings of the 11th Euromicro Conference on Real-Time Systems (ECRTS '99)*. Juni 1999, Seiten 208 –215. ISBN: 0-7695-0240-7. DOI: 10.1109/EMRTS.1999.777467.

[Hily 1999] Sébastien Hily und André Seznec. „Out-of-Order Execution may not be Cost-Effective on Processors Featuring Simultaneous Multithreading". In: *Proceedings of the 5th International Symposium on High Performance Computer Architecture (HPCA '99)*. Jan. 1999, Seiten 64 –67. ISBN: 0-7695-0004-8. DOI: 10.1109/HPCA.1999.744331.

[Hirata 1992] Hiroaki Hirata, Kozo Kimura, Satoshi Nagamine, Yoshiyuki Mochizuki, Akio Nishimura, Yoshimori Nakase und Teiji Nishizawa. „An elementary processor architecture with simultaneous instruction issuing from multiple threads". In: *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA '92)*. 1992, Seiten 136 –145. ISBN: 0-89791-509-7. DOI: 10.1145/139669.139710.

[Hoover 2006] Greg Hoover, Forrest Brewer und Timothy Sherwood. „A case study of multi-threading in the embedded space“. In: *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '06)*. 2006, Seiten 357 –367. ISBN: 1-59593-543-6. DOI: 10.1145/1176760.1176803.

[Hoogenboom 1997] Peter Hoogenboom. *TASKING helps Siemens with 32-bit TriCore architecture design*. White Paper. TASKING Inc. Dez. 1997. URL: www.tasking.com/resources/tricore-architecture.pdf.

[IEEE 2005] *IEEE Standard SystemC Language Reference Manual (IEEE Std 1666-2005)*. IEEE Computer Society. 2006.

[Infineon 2003] *TriCore Compiler Writer's Guide*. 1.4. Infineon Technologies AG. Dez. 2003.

[Infineon 2004] *Application Note AP32071: TriCore 1 Pipeline Behaviour & Instruction Timing*. 1.1. Infineon Technologies AG. Juni 2004.

[Infineon 2008a] *TriCore 1 Architecture Volume 1: Core Architecture V1.3 & V1.3.1*. 1.3.8. Infineon Technologies AG. Jan. 2008.

[Infineon 2008b] *TriCore 1 Architecture Volume 2: Instruction Set V1.3 & V1.3.1*. 1.3.8. Infineon Technologies AG. Jan. 2008.

[Infineon 2011] *Highly Integrated and Performance Optimized 32-bit Microcontrollers for Automotive and Industrial Applications*. Infineon Technologies AG. Feb. 2011.

[Jain 2002] Rohit Jain, Christopher J. Hughes und Sarita V. Adve. „Soft Real-Time Scheduling on Simultaneous Multithreaded Processors“. In: *Proceedings of the 23rd Real-Time Systems Symposium (RTSS '02)*. 2002, Seiten 134 –. ISBN: 0-7695-1851-6.

[Knuth 1998] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. 2. Auflage. 1998. ISBN: 0-201-89685-0.

[Kohout 2003] Paul Kohout, Brinda Ganesh und Bruce Jacob. „Hardware support for real-time operating systems“. In: *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '03)*. 2003, Seiten 45 –51. ISBN: 1-58113-742-7. DOI: 10.1145/944645.944656.

[Kreuzinger 2000] Jochen Kreuzinger, Alexander Schulz, Matthias Pfeffer, Theo Ungerer, Uwe Brinkschulte und Christian Krakowski. „Real-time Scheduling on Multithreaded Processors“. In: *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications (RTCSA '00)*. Dez. 2000, Seiten 155 –159. ISBN: 0-7695-0930-4.

[Kuacharoen 2003] Pramote Kuacharoen, Mohamed A. Shalan und Vincent J. Mooney III. „A Configurable Hardware Scheduler for Real-Time Systems“. In: *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '03)*. Juni 2003, Seiten 96 –101.

[Landet 2009] Cédric Landet. „Modélisation d'un processeur à exécution simultanée de flots pour le temps réel strict“. Dissertation. Université de Toulouse III - Paul Sabatier, Dez. 2009.

[Lehoczky 1987] John P. Lehoczky, Lui Sha und Jay K. Strosnider. „Enhanced Aperiodic Responsiveness in Hard Real-Time Environments“. In: *Proceedings of the 8th Real-Time Systems Symposium (RTSS '87)*. 1987, Seiten 261 –270.

[Lehoczky 1989] John Lehoczky, Lui Sha und Ye Ding. „The rate monotonic scheduling algorithm: exact characterization and average case behavior“. In: *Proceedings of the 10th Real-Time Systems Symposium (RTSS '89)*. Dez. 1989, Seiten 166 –171. DOI: 10.1109/REAL.1989.63567.

[Lehoczky 1992] John P. Lehoczky und Sandra Ramos-Thuel. „An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems“. In: *Proceedings of the 13th Real-Time Systems Symposium (RTSS '92)*. Dez. 1992, Seiten 110 –123. DOI: 10.1109/REAL.1992.242671.

[Lickly 2008] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards und Edward A. Lee. „Predictable programming on a precision timed architecture“. In: *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '08)*. 2008, Seiten 137 –146. ISBN: 978-1-60558-469-0. DOI: 10.1145/1450095.1450117.

[Lindh 1991] Lennart Lindh und Frank Stanischewski. „FASTCHART-Idea and Implementation“. In: *Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer and Processors (ICCD '91)*. 1991, Seiten 401 –404. ISBN: 0-8186-2270-9.

[Lindh 1992] Lennart Lindh und Frank Stanischweski. „FASTHARD - A Fast Time Deterministic HARDware Based Real-Time-Kernel“. In: *Proceedings of the 4th Euromicro Workshop on Real-Time Systems (ECRTS '92)*. 1992, Seiten 21 –25. DOI: 10.1109/EMWRT.1992.637466.

[Lindholm 1999] Tim Lindholm und Frank Yellin. *Java Virtual Machine Specification*. 2. Auflage. 1999. ISBN: 0201432943.

[Liu 2000] Jane W. S. Liu. *Real-Time Systems*. 1. Auflage. 2000. ISBN: 0-13-099651-3.

[Liu 1973] C. L. Liu und James W. Layland. „Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment“. In: *Journal of the ACM* 20.1 (Jan. 1973), Seiten 46 –61.

[Manacher 1967] Glenn K. Manacher. „Production and Stabilization of Real-Time Task Schedules“. In: *Journal of the ACM* 14.3 (Juli 1967), Seiten 439 –465. DOI: 10.1145/321406.321408.

[May 2009] David May. *The XMOS XS1 Architecture*. XMOS Ltd. Okt. 2009.

[MIPS 2010a] *MIPS Architecture for Programmers Volume IV-f: The MIPS MT Application-Specific Extension to the MIPS32 Architecture*. Revision 1.06. MIPS Technologies, Inc. Apr. 2010.

[MIPS 2010b] *Programming the MIPS32 34KTM Core Family*. Revision 1.64. MIPS Technologies, Inc. Nov. 2010.

[Mische 2008] Jörg Mische, Sascha Uhrig, Florian Kluge und Theo Ungerer. „Exploiting Spare Resources of In-order SMT Processors Executing Hard Real-time Threads“. In: *Proceedings of the 26th International Conference on Computer Design (ICCD '08)*. Okt. 2008, Seiten 371 –376. DOI: 10.1109/ICCD.2008.4751887.

[Mische 2009] Jörg Mische, Sascha Uhrig, Florian Kluge und Theo Ungerer. „IPC Control for Multiple Real-Time Threads on an In-order SMT Processor“. In: *Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC '09)*. Jan. 2009, Seiten 125 –139. DOI: 10.1007/978-3-540-92990-1_11.

[Mische 2010a] Jörg Mische, Irakli Guliashvili, Sascha Uhrig und Theo Ungerer. „How to Enhance a Superscalar Processor to Provide Hard Real-Time Capable In-Order SMT“. In: *Proceedings of the 23rd International Conference on Architecture of Computing Systems (ARCS '10)*. Feb. 2010, Seiten 2 –14. DOI: 10.1007/978-3-642-11950-7_2.

[Mische 2010b] Jörg Mische, Sascha Uhrig, Florian Kluge und Theo Ungerer. „Using SMT to Hide Context Switch Times of Large Real-Time Tasksets“. In: *Proceedings of the 16th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '10)*. Aug. 2010, Seiten 255 –264. DOI: 10.1109/RTCSA.2010.33.

[Mok 1983] Aloysius K. Mok. „Fundamental Design Problems of Distributed Systems fpr the Hard Real-Time Environment“. Dissertation. Massachusetts Institute of Technology. Deptartment of Electrical Engineering and Computer Science, Mai 1983.

[Moon 2000] Sung-Whan Moon, Kang G. Shin und Jennifer Rexford. „Scalable Hardware Priority Queue Architectures for High-Speed Packet Switches“. In: *IEEE Transactions on Computers* 49.11 (2000), Seiten 1215 –1227. DOI: 10.1109/12.895938.

[Moon 2003] Byung In Moon, Moon Gyung Kim, In Pyo Hong, Ki Chang Kim und Yong Surk Lee. „Study of an In-order SMT Architecture and Grouping Schemes“. In: *International Journal of Control, Automation, and Systems* 1.3 (2003), Seiten 339 –350.

[Moon 2004] Byung In Moon, Hongil Yoon, Ilgun Yun und Sungho Kang. „An In-Order SMT Architecture with Static Resource Partitioning for Consumer Applications“. In: *Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*. 2004, Seiten 539 –544.

[Nemirovsky 1991] Mario Daniel Nemirovsky, Forrest Brewer und Roger C. Wood. „DISC: dynamic instruction stream computer“. In: *Proceedings of the 24th Annual International Symposium on Microarchitecture (MICRO 24)*. 1991, Seiten 163 –171. ISBN: 0-89791-460-0. DOI: 10.1145/123465.123498.

[Omondi 1998] Amos R. Omondi und Michael Horne. „Performance of a context cache for a mulithreaded pipeline“. In: *Journal of Systems Architecture: the EUROMICRO Journal* 45.4 (1998), Seiten 305 –322. DOI: 10.1016/S1383-7621(97)00084-2.

[Parekh 2000] Sujay S. Parekh, Susan J. Eggers, Henry M. Levy und Jack L. Lo. *Thread-Sensitive Scheduling for SMT Processors*. Technischer Bericht 2000-04-02. Department of Computer Science and Engineering, University of Washington, 2000.

[Patterson 1980] David A. Patterson und David R. Ditzel. „The case for the reduced instruction set computer“. In: *SIGARCH Computer Architecture News* 8.6 (Okt. 1980), Seiten 25 –33. DOI: 10.1145/641914.641917.

[POSIX 2008] *POSIX.1-2008 IEEE Std 1003.1-2008*. The Open Group Technical Standard Base Specifications, Issue 7. 2008.

[Puschner 2003] Peter Puschner. „Hard Real-Time Programming is Different“. In: *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS '03)*. 2003, Seite 116.2. ISBN: 0-7695-1926-1.

[Raasch 2003] Steven E. Raasch und Steven K. Reinhardt. „The Impact of Resource Partitioning on SMT Processors“. In: *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*. 2003, Seiten 15 –26. ISBN: 0-7695-2021-9.

[Raasch 1999] Steven E. Raasch und Steven K. Reinhardt. „Applications of Thread Prioritization in SMT Processors“. In: *Proceedings of the 1999 Workshop on Multithreaded Execution, Architecture, and Compilation*. Jan. 1999.

[Reichardt 2009] Jürgen Reichardt und Bernd Schwarz. *VHDL-Synthese, Entwurf digitaler Schaltungen und Systeme*. 5. Auflage. 2009. ISBN: 978-3-486-58987-0.

[Schlansker 2000] Michael S. Schlansker und B. Ramakrishna Rau. „EPIC: Explicitly Parallel Instruction Computing“. In: *IEEE Computer* 33.2 (Feb. 2000), Seiten 37 –45. DOI: 10.1109/2.820037.

[Schoeberl 2009] Martin Schoeberl und Peter Puschner. „Is chip-multiprocessing the end of real-time scheduling?“ In: *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET '09)*. Juli 2009.

[Schultes 1999] Renate Schultes. „TriCore – 32-Bit Power für das nächste Jahrtausend“. In: *Proceedings of the PCNEWS-64A*. Sep. 1999. URL: http://www.pcnews.at/d/_pdf/n64a0063.pdf.

[Serlin 1972] Omri Serlin. „Scheduling of time critical processes“. In: *Proceedings of the AFIPS Spring Joint Computer Conference*. Mai 1972, Seiten 925 –932. DOI: 10.1145/1478873.1478995.

[Sha 1986] Lui Sha, John P. Lehoczky und Ragunathan Rajkumar. „Solutions for some practical problems in prioritized preemptive scheduling“. In: *Proceedings of the 7th Real-Time Systems Symposium (RTSS '86)*. 1986, Seiten 181 –191.

[Sherwood 1999] Timothy Sherwood und Brad Calder. *Time varying Behavior of Programs*. Technischer Bericht UCSD-CS99-630. Department of Computer Science and Engineering, University of California at San Diego, Aug. 1999.

[Sima 1997] Dezsõ Sima. „Superscalar Instruction Issue“. In: *IEEE Micro* 17.5 (Sep. 1997), Seiten 28 –39. DOI: 10.1109/40.621211.

[Snavely 2000] Allan Snavely und Dean M. Tullsen. „Symbiotic Jobscheduling for a Simultaneous Multithreading Processor“. In: *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2000, Seiten 234 –244. DOI: 10.1145/356989.357011.

[Soundararajan 1992] Vijayaraghavan Soundararajan und Anant Agarwal. *Dribbling Registers: A Mechanism for Reducing Context Switch Latency in Large-Scale Multiprocessors*. Technischer Bericht MIT/LCS/TM-474. Massachusetts Institute of Technology, Laboratory for Computer Science, Juni 1992.

[Sprunt 1988] Brinkley Sprunt, John Lehoczky und Lui Sha. „Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm“. In: *Proceedings of the 9th Real-Time Systems Symposium (RTSS '88)*. Dez. 1988, Seiten 251 –258. DOI: 10.1109/REAL.1988.51120.

[Sprunt 1989] Brinkley Sprunt, Lui Sha und John Lehoczky. „Aperiodic Task Scheduling for Hard-Real-Time Systems“. In: *Real-Time Systems* 1.1 (1989), Seiten 27 –60. DOI: 10.1007/BF02341920.

[Sprunt 1990] Brinkley Sprunt. „Aperiodic task scheduling for real-time systems“. Dissertation. Carnegie Mellon University, Department of Electrical and Computer Engineering, Aug. 1990.

[Spuri 1996] Marco Spuri und Giorgio Buttazzo. „Scheduling aperiodic tasks in dynamic priority systems“. In: *Real-Time Systems* 10.2 (1996), Seiten 179 –210. DOI: 10.1007/BF00360340.

[Stankovic 1989] John A. Stankovic und Krithi Ramamritham. „The Spring kernel: a new paradigm for real-time operating systems“. In: *SIGOPS Operating Systems Review* 23.3 (1989), Seiten 54 –71. DOI: 10.1145/71021.71024.

[Strosnider 1995] Jay K. Strosnider, John P. Lehoczky und Lui Sha. „The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments“. In: *IEEE Transactions on Computers* 44.1 (1995), Seiten 73 –91. DOI: 10.1109/12.368008.

[Sáez 1999] Sergio Sáez, Joan Vila, Alfons Crespo und Angel Garcia. „A hardware scheduler for complex real-time systems“. In: *Proceedings of the IEEE International Symposion on Industrial Electronics (ISIE '99)*. Jan. 1999, Seiten 43 –48. DOI: 10.1109/ISIE.1999.801754.

[Thornton 1965] James E. Thornton. „Parallel operation in the control data 6600“. In: *Proceedings of the AFIPS Fall Joint Computer Conference*. Okt. 1965, Seiten 33 –40. DOI: 10.1145/1464039.1464045.

[Tullsen 1995] Dean M. Tullsen, Susan J. Eggers und Henry M. Levy. „Simultaneous multithreading: maximizing on-chip parallelism“. In: *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*. 1995, Seiten 392 –403. ISBN: 0-89791-698-0. DOI: 10.1145/223982.224449.

[Tune 2004] Eric Tune, Rakesh Kumar, Dean M. Tullsen und Brad Calder. „Balanced Multithreading: Increasing Throughput via a Low Cost Multithreading Hierarchy“. In: *Proceedings of the 37th Annual International Symposium on Microarchitecture (MICRO 37)*. 2004, Seiten 183 –194. ISBN: 0-7695-2126-6. DOI: 10.1109/MICRO.2004.8.

[Ubicom 2003] *The Ubicom IP3023 Wireless Network Processor*. White Paper. Ubicom, Inc. Mountain View, CA, USA, Apr. 2003.

[Ubicom 2004] *Programmer's Reference Manual IP3000 Wireless Network Processor Family*. Ubicom, Inc. Apr. 2004.

[Ungerer 2010] Theo Ungerer, Francisco J. Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quinones, Mike Gerdes, Marco Paolieri, Julian Wolf, Hugues Casse, Sascha Uhrig, Irakli Guliashvili, Michael Houston, Florian Kluge, Stefan Metzlaff und Jörg Mische. „MERASA: Multicore Execution of Hard Real-Time Applications Supporting Analyzability“. In: *IEEE Micro Special Issue on*

European Multicore Processing Projects 30 (Sep. 2010), Seiten 66 –75. DOI: 10.1109/MM.2010.78.

[Ungerer 1997] Theo Ungerer. *Parallelrechner und parallele Programmierung*. 1. Auflage. 1997. ISBN: 3-8274-0231.

[Weber 1989] Wolf-Dietrich Weber und Anoop Gupta. „Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: preliminary results“. In: *Proceedings of the 16th Annual International Symposium on Computer Architecture (ISCA '89)*. 1989, Seiten 273 –280. ISBN: 0-89791-319-1. DOI: 10.1145/74925.74956.

[Wirth 2000] Niklaus Wirth. *Algorithmen und Datenstrukturen - Pascal Version*. 5. Auflage. 2000. ISBN: 3-519-22250-7.

[Yamasaki 2007] Nobuyuki Yamasaki, Ikuo Magaki und Tsutomu Itou. „Prioritized SMT Architecture with IPC Control Method for Real-Time Processing“. In: *Proceedings of the 13th Real-Time and Embedded Technology and Applications Symposium (RTAS '07)*. 2007, Seiten 12 –21. DOI: 10.1109/RTAS.2007.28.

[Zang 2008] Chengjie Zang, Shigeki Imai, Steven Frank und Shinji Kimura. „Issue Mechanism for Embedded Simultaneous Multithreading Processor“. In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E91-A.4 (2008), Seiten 1092 –1100. DOI: 10.1093/ietfec/e91-a.4.1092.

Danksagung

Ich danke Prof. Dr. Theo Ungerer dafür, dass er mir einerseits sehr viele Freiheiten lies, andererseits aber auch zum Abschluss der Arbeiten drängte, wenn dies nötig war. Außerdem danke ich Prof. Dr. Rudi Knorr für die Zweitkorrektur und Prof. Dr. Bernhard Bauer und Prof. Dr. Robert Lorenz für die mündliche Prüfung.

Besonders zu Bedanken habe ich mich bei Prof. Dr. Sascha Uhrig, denn mit niemandem kann man sich besser über Fragen der Prozessorarchitektur streiten als mit ihm. Auch wenn mein ehemaliger Kollege Irakli Guliashvili es beständig abstreiten wird, so hat er doch großen Anteil an dieser Dissertation, da er mir durch die Erstellung des FPGA-Modells viel Arbeit erspart hat und etliche Unzulänglichkeiten aufdeckte. Weiterhin möchte ich mich bei Dr. Florian Kluge, Stefan Metzlaff und den anderen Mitgliedern des Lehrstuhls für die Zusammenarbeit und Inspiration bedanken.

Lebenslauf

Persönliche Daten

Name:	Jörg Alfred Mische
Geburtsdatum:	10. November 1977
Geburtsort:	Augsburg

Schulische Ausbildung

Sep. 1984 – Aug. 1988	Grundschule Hochzoll-Nord in Augsburg
Sep. 1988 – Jun. 1997	Rudolf-Diesel-Gymnasium in Augsburg

Zivildienst

Sep. 1997 – Sep. 1998	Kolping Familienferienwerk Augsburg

Hochschulausbildung

Okt. 1998 – Sep. 2001	Studium Diplom-Physik an der Universität Augsburg
Okt. 2001 – Sep. 2002	Studium Diplom-Physik an der Ludwig-Maximilians-Universität München
Okt. 2002 – Apr. 2006	Studium Diplom-Informatik an der Universität Augsburg

Wissenschaftliche Tätigkeit

seit Mai 2006	Wissenschaftlicher Angestellter am Lehrstuhl für systemnahe Informatik und Kommunikationssysteme an der Universität Augsburg