Optimale Cache-Nutzung für Realzeitsoftware auf Multiprozessorsystemen

Alexander von Bülow

Dissertation

Lehrstuhl für Realzeit-Computersysteme

Optimale Cache-Nutzung für Realzeitsoftware auf Multiprozessorsystemen

Alexander von Bülow

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr.-Ing. K. Diepold
Prüfer der Dissertation:	
	1. Univ.-Prof. Dr.-Ing. G. Färber
	2. Priv.-Doz. Dr.-Ing. habil. W. Stechele

Die Dissertation wurde am 04. Oktober 2005 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 13. Dezember 2005 angenommen.

Danksagung

Diese Dissertation entstand als Ergebnis meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Realzeit-Computersysteme der Technischen Universität München. Teile der Arbeit wurden von der *Deutschen Forschungsgemeinschaft* (DFG) unter dem Förderkennzeichen Fa 109/15-1 („Real-Time with Commercial Off-the-Shelf Multiprocessor Systems“) gefördert.

Mein besonderer Dank geht an Prof. Färber, der diese Arbeit ermöglicht hat.

Weiterhin danke ich allen Mitarbeitern des Lehrstuhls für die gute Zusammenarbeit und das kollegiale Arbeitsklima. Mein besonderer Dank geht an Jürgen Stohr, der mit vielen wertvollen Diskussionen zum Gelingen dieser Arbeit beigetragen hat. An dieser Stelle möchte ich auch meinem ehemaligen Diplomarbeits-Betreuer am Lehrstuhl, Herrn Dr. Stefan Petters, für seine Hilfe gerade zu Beginn meiner Tätigkeit am Institut danken.

Großen Dank schulde ich meinen Eltern, die mir das Studium ermöglicht und mich jederzeit voll unterstützt haben.

München, im September 2005

Für Anita.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der verwendeten Symbole

AMD	Advanced Micro Devices
APIC	Advanced Programmable Interrupt Controller
BCET	Best-Case Execution Time
BP	Branch Prediction
BTB	Branch Target Buffer
COTS	Commercial off-the-Shelf
DDR	Double Data Rate
DRAM	Dynamic Random Access Memory
E/A	Ein/Ausgabe
GB	Gigabyte ($1024 \cdot 1024 \cdot 1024$ Byte)
GBHC	Global Branch History Counter
GBHR	Global Branch History Register
GPOS	General Purpose Operating System
MB	Megabyte ($1024 \cdot 1024$ Byte)
NUMA	Non-Unified Memory Architecture
PMC	Performance-Monitoring Counter
RAM	Random Access Memory
RTOS	Real-Time Operating System
SDRAM	Syncronous Dynamic Random Access Memory
SMP	Symmetric Multiprocessing
SRAM	Static Random Access Memory
TLB	Translation Lookaside Buffer
TSC	Time-Stamp Counter
WCET	Worst-Case Execution Time

Formelzeichen:

l	Anzahl der Sets im Cache
m	Cacheline-Größe in Bytes
n	Assoziativität des Caches
A	Attribut
S	Größe eines Objekts
D	Datenobjekt

Kurzfassung

In dieser Arbeit wird ein Algorithmus zur Anordnung von Code und Daten im Speicher vorgestellt. Der Algorithmus ermöglicht eine effektive Nutzung der Caches im Hinblick auf einen Einsatz in Realzeitsystemen. Er sorgt dafür, dass Laufzeitverzögerungen durch Cache-Misses besser prädizierbar sind und ermöglicht es, Teile des Codes und der Daten permanent im Cache zu halten. Dies ist für besonders zeitkritische Teilsysteme unerlässlich. Der Einfluss der Caches auf das Laufzeitverhalten von Software wird an praktischen Beispielen untersucht und bewertet.

In diesem Zusammenhang werden auch Techniken zur intelligenten Speicherverwaltung für Realzeitsysteme untersucht. Ein Schwerpunkt ist hier der Datenaustausch zwischen Tasks, auch auf unterschiedlichen Prozessoren. Es werden die Auswirkungen des Cache-Snooping untersucht sowie Eckpunkte für eine vorteilhafte Verteilung von Tasks auf mehrere Prozessoren unter dem Gesichtspunkt der Speicherverwaltung erarbeitet.

Neben den Caches spielen auch Mechanismen wie die out-of-order execution und die Branch Prediction eine wichtige Rolle bei modernen Prozessoren. Diese Mechanismen werden erklärt und im Hinblick auf ihre Auswirkungen auf die Laufzeit von Software untersucht. Im Vergleich zu den Caches spielen diese Faktoren jedoch nur eine geringe Rolle.

Es wird eine Methodik zur präzisen Messung von Ausführungszeiten vorgestellt, die es ermöglicht, Laufzeiten unter verschiedenen Bedingungen zu messen. So können vor der Messung beispielsweise der Instruktionen- oder der Daten-Cache so manipuliert werden, dass sie für den zu messenden Code keine brauchbaren Informationen enthalten. Damit lassen sich gezielt die Einflüsse verschiedener Architekturmerkmale messen oder auch Messungen unter Worst-Case Bedingungen durchführen. Die Messungen haben nur einen minimalen Einfluss auf den gemessenen Code und können mit Hilfe der Performance-Monitoring Counter gut interpretiert werden.

1 Einleitung

Caches spielen bei modernen Prozessoren eine ganz zentrale Rolle für die Ausführungsgeschwindigkeit von Software. In den letzten Jahren ist der Unterschied zwischen den Zugriffsgeschwindigkeiten auf den Hauptspeicher und der Arbeitsgeschwindigkeit der Prozessoren stark gewachsen. Die Caches dienen dazu, diesen Unterschied möglichst zu kompensieren, damit die Prozessoren ihr Leistungspotential voll entfalten können. Leider passt in aller Regel nicht die gesamte Software in die Caches, so dass zur Laufzeit des Systems immer wieder neue Inhalte aus dem Speicher in die Caches geladen werden müssen, und dafür modifizierte Inhalte aus den Caches in den Hauptspeicher zurück geschrieben werden müssen. Durch die stark unterschiedlichen Zugriffszeiten kann es dabei zu größeren Schwankungen und Verzögerungen der Laufzeit von Software kommen.

1.1 Motivation

Die Anforderungen an die Rechenleistung von Realzeitsystemen steigen stetig an. Früher reichte oft ein einfacher Mikroprozessor zur Bewältigung der Aufgaben aus. Diese Prozessoren haben eine einfache, serielle Programmausführung ohne Caches, Pipelines oder Branch Prediction. Sie sind dadurch nicht sehr schnell, aber ihre Ausführungszeiten lassen sich analytisch leicht bestimmen. Diese Eigenschaft macht sie wertvoll für Realzeitsysteme.

In jedem Realzeitsystem müssen die Ausführungszeiten einzelner Codeteile bekannt sein. Von besonderer Bedeutung ist hier die WCET (Worst-Case Execution Time) einer Task. Diese gibt die Ausführungszeit im ungünstigsten Fall an. Da ein Realzeitsystem innerhalb einer bestimmten Zeitspanne (Deadline) ein Ergebnis liefern muss, sind die WCETs von großer Bedeutung.

Verknüpft man beide Anforderungen an ein Realzeitsystem, hohe Rechenleistung und sicher bestimmbare WCETs, kommt man in ein Dilemma. PC-Prozessoren bieten eine sehr hohe Rechenleistung, die durch die rasche Weiterentwicklung dieser Prozessoren weiter zunimmt. Diese hohe Rechenleistung erreichen sie durch Techniken wie Caches, out-of-order execution und Branch Prediction. Diese haben jedoch zur Folge, dass die Ausführungszeiten abhängig vom jeweiligen Zustand dieser Mechanismen stark schwanken können. Vor allem die Caches spielen hier eine große Rolle.

Aber nicht nur die Architektur der Prozessoren ist für die Entstehung von Laufzeiten verantwortlich. Auch die Hardwareanbindung, vor allem die des Hauptspeichers, ist von großer Bedeutung. Blockieren sich mehrere Teilnehmer (Prozessoren, Peripheriegeräte) beim gemeinsamen Zugriff auf den Speicher, können sehr lange und nahezu unvorhersehbare Wartezeiten

entstehen. Dasselbe gilt für direkte Zugriffe auf die umgebende Hardware, die beispielsweise über den PCI-Bus angekoppelt ist. Die zunehmende Komplexität dieser Systeme erschwert eine hinreichend genaue Systemanalyse, mit der man diese Effekte für die Bestimmung der WCET berücksichtigen könnte.

Die Flexibilität und die hohe Rechenleistung moderner PC-Systeme macht diese interessant für den Einsatz in zeitkritischen Anwendungen. Die Systeme verfügen über eine große Vielfalt an Hardware und können sehr individuell mit Standardkomponenten bestückt werden. Dies macht sie auch im Vergleich zu Speziallösungen wesentlich kostengünstiger. Der schnelle technische Fortschritt bringt zudem in regelmäßigen Abständen noch leistungsfähigere Komponenten auf den Markt. Da diese meist abwärtskompatibel zu ihren Vorgängern sind, kann ein bestehendes System in aller Regel ohne großen Aufwand auf dem Stand der Technik gehalten werden.

1.2 Ziele dieser Arbeit

Der Schwerpunkt dieser Arbeit liegt auf einem Algorithmus zur möglichst optimalen Nutzung der Caches moderner Prozessoren für Realzeitsysteme. Zum einen soll die Anzahl der möglichen Cache-Misses gering gehalten werden; zum anderen sollen die Laufzeitverzögerungen für die verbleibenden Misses besser vorhersagbar sein. Es ist ein großer Unterschied, ob ein Miss mit oder ohne vorheriges Zurückschreiben einer Cacheline erfolgt.

Neben den Caches für Instruktionen und Daten spielen auch die TLBs (Translation Lookaside Buffers) eine Rolle. Diese kleinen Caches speichern Informationen für die Umrechnung einer virtuellen Adresse in eine physikalische. Sind diese Informationen nicht in den TLBs, müssen diese zunächst aus dem Speicher geladen werden. Somit können sich auch Verdrängungen in den TLBs negativ auf die Laufzeiteigenschaften des Systems auswirken. Ziel des Algorithmus zur Anordnung von Code und Daten im Speicher ist es daher, die TLBs in die Optimierung mit einzubeziehen, so dass unnötige Misses vermieden werden können.

Die Ausführungszeiten für Code, der komplett in den Caches läuft, sind besonders kurz und zuverlässig vorhersagbar. Oft gibt es in Realzeitsystemen besonders zeitkritische Anteile, deren Anforderungen nur erfüllt werden können, wenn sich deren Code und Daten in den Caches befinden. Eine Methode, um dies zu realisieren, ist das Cache-Locking. Von manchen Prozessoren wird dies in Hardware unterstützt, PC-Prozessoren haben diese Unterstützung nicht. Der Algorithmus soll daher die Möglichkeit bieten, bestimmte Code- und Datenobjekte permanent im Cache zu halten.

Oft hat man in einem Realzeitsystem mehrere Prozessoren, die jeweils bestimmte Tasks ausführen. Möchte man PC-Systeme als Realzeitsysteme einsetzen, wird meist der Ansatz verfolgt, ein Standard- und ein Realzeitbetriebssystem auf einem Rechner zu betreiben. Dabei ist es von Vorteil, wenn jedes Betriebssystem mindestens einen eigenen Prozessor hat. Damit lassen sich unerwünschte gegenseitige Beeinflussungen weitgehend vermeiden. Dabei hat man jedoch das Problem, dass mehrere Prozessoren dieselben Inhalte in ihren Caches haben können. Um die Daten systemweit konsistent zu halten, müssen diese Inhalte abgeglichen werden, sobald ein

Prozessor diese verändert. Dies wird automatisch vom Cache-Snooping Protokoll erledigt. Je nach Implementierung kann dies zu Laufzeitverzögerungen führen, oder, wie in Abschnitt 5.3.5 gezeigt wird, für einen schnellen Weg zum Datenaustausch zwischen Tasks auf unterschiedlichen Prozessoren genutzt werden. Diese Aspekte sollen von dem Algorithmus berücksichtigt werden, um Verzögerungen durch Cache-Snooping zu vermeiden.

Neben dem Algorithmus zur Anordnung von Code und Daten im Speicher sollen auch die anderen Mechanismen, die zu Laufzeitschwankungen bei der Ausführung von Code beitragen, untersucht werden. Dazu zählen vor allem die out-of-order execution mit Hilfe von Pipelines und die Branch Prediction.

1.3 Gliederung

In Kapitel 2 werden wissenschaftliche Arbeiten auf den Gebieten der WCET-Analyse im Allgemeinen, der Analyse von Realzeitsystemen mit Caches und der Nutzung von PC-Hardware als Realzeitsystem vorgestellt. Dort wird der Stand der Forschung wiedergegeben und zugleich eine Abgrenzung zu dieser Arbeit vorgenommen.

Kapitel 3 gibt einen Überblick über die wichtigsten Grundlagen der PC-Architektur. Es wird sowohl auf die Architektur der Prozessoren eingegangen als auch auf den Aufbau von Multiprozessorsystemen. Weitere Themen sind der Chipsatz und Bussysteme in PCs. Den Schwerpunkt dieses Kapitels bilden der Aufbau und die Funktionsweise der Caches. Das Kapitel wird mit einem Überblick über die Funktionsweise moderner Speichertechnologien abgeschlossen.

Die Ausführungszeit von Software auf modernen Prozessoren ist das Thema von Kapitel 4. Zunächst wird eine Methodik zur präzisen Messung von Ausführungszeiten auf PC-Prozessoren vorgestellt. Danach werden die Prinzipien der Befehlsausführung mit Hilfe von Pipelines und out-of-order execution (Tomasulo-Algorithmus) erklärt und hinsichtlich ihres Einflusses auf die Ausführungszeit von Software bewertet. Analoge Betrachtungen zu den Einflüssen der Branch Prediction, der Caches und der Programmunterbrechung durch Interrupts folgen. Das Kapitel endet mit Betrachtungen zu SMP- und NUMA-Systemen.

In Kapitel 5 wird der Algorithmus zur Anordnung von Code und Daten vorgestellt. Die Anwendung dieser Vorgehensweise wird an einem Beispiel aus der Praxis gezeigt. Im Anschluss wird auf Aspekte der Speicherverwaltung für Realzeitsysteme eingegangen, vor allem im Hinblick auf eine Interprozesskommunikation in Echtzeit und die vollständige Trennung von GPOS und RTOS. Weitere Abschnitte sind Aspekten zur Taskverteilung auf mehreren Realzeitprozessoren und der Durchführung eines Echtzeitnachweises gewidmet.

Kapitel 6 präsentiert die Ergebnisse mehrerer praktischer Untersuchungen zu den Einflüssen der Caches auf das Laufzeitverhalten der Software. Dabei werden die für den Algorithmus wichtigen Ergebnisse herausgearbeitet. Es werden Eckpunkte für eine sinnvolle Nutzung des Caches aus der Sicht eines Anwenders formuliert. Kapitel 7 fasst die Arbeit zusammen.

2 Stand der Technik

Dieses Kapitel gibt einen Überblick über wichtige Arbeiten im Zusammenhang mit Methoden zur Laufzeitbestimmung von Software und der Nutzung von Prozessoren mit Caches in Realzeitsystemen. Bei der Laufzeitbestimmung von Software liegt der Schwerpunkt bei Realzeitsystemen auf der Bestimmung der Worst-Case Execution Time (WCET), die elementarer Bestandteil jedes Realzeitnachweisverfahrens ist. Bei der Bestimmung der WCET spielen die Caches eines Prozessors eine ganz wesentliche Rolle. Viele Forschungsgruppen haben sich daher mit der Bestimmung der WCET für Prozessoren mit unterschiedlichen Cache-Architekturen beschäftigt.

2.1 Bestimmung von Laufzeiten für Realzeitsoftware

Zur Bestimmung von Laufzeiten für Software gibt es zwei Ansätze: Die Berechnung der Laufzeit mit analytischen Methoden und die Messung von Laufzeiten auf dem Zielsystem. Die analytische Laufzeitbestimmung setzt voraus, dass man das Zielsystem hinreichend genau modellieren kann und aus dem Modell heraus den Worst-Case simulieren kann. Aus der Simulation lässt sich dann die WCET bestimmen. Der Ansatz, Laufzeiten zu messen, ist gut geeignet für Systeme, die sich nicht hinreichend genau simulieren lassen.

Analytische Methoden

Mit der Modellierung und Simulation moderner Prozessorarchitekturen beschäftigt sich die Forschungsgruppe um R. Wilhelm der Universität des Saarlandes. In [64] wird ein Ansatz vorgestellt, die Pipeline eines modernen Mikroprozessors zu modellieren, um daraus die Ausführungszeit einzelner Befehle berechnen zu können. Eine Methode zur Separierung einer Cache- und Pfad-Analyse zur Bestimmung der WCET wird in [77] vorgestellt. Die Anwendung der Analysemethoden für Cache, Pipelines und der Programmpfade wird in [26] gezeigt. Diese Toolkette zur WCET-Bestimmung wird in [35] auf einen SuperSPARC, einen Motorola ColdFire 5307 und einen Motorola PowerPC 755 angewandt und die Ergebnisse präsentiert. Die Arbeit in [78] gibt einen Überblick über Designtrends für Hard- und Software Design und zeigt Methoden auf, diese so zu konzipieren, dass das Zeitverhalten zukünftiger Systeme besser abgeschätzt werden kann.

Ein wichtiger Teilaspekt bei der Bestimmung maximaler Laufzeiten auf modernen Prozessoren ist die out-of-order execution und die dynamische Sprungvorhersage. Die out-of-order execu-

tion ermöglicht es dem Prozessor, die Reihenfolge der abzuarbeitenden Befehle umzustellen. Die Sprungvorhersage versucht bei der Dekodierung eines Sprungbefehls bereits das Verhalten des Befehls und den nächsten auszuführenden Befehl vorherzusagen. Dieser wird dann bereits geladen und spekulativ ausgeführt (Details dazu in Abschnitt 4.2).

Mit den Einflüssen dieser Techniken, insbesondere der Branch Prediction, befassen sich Colin und Puaut in [23]. Sie schlagen eine Methodik vor, mit der die Anzahl der Verzögerungen im Programmablauf aufgrund fehlerhafter Sprungvorhersagen angegeben werden kann. Die Methodik basiert auf einer statischen Code-Analyse und einer Modellierung des Branch Target Buffers. Die Arbeit von Bate und Reutemann [7] basiert auf diesem Ansatz und erweitert ihn, indem die Semantik der Sprunganweisungen im Source Code mit herangezogen wird, um die Befehle als „easy-to-predict" oder „hard-to-predict" zu klassifizieren. Mit dieser Klassifizierung wird eine statische Analyse-Methodik vorgestellt, die auf Vorhersagemechanismen mit bimodalen Sprungzählern und einer globalen Sprunghistorie angewandt werden kann. Die Arbeit von Engblom [25] untersucht die Auswirkungen der Sprungvorhersage an Fallstudien mit verschiedenen Prozessortypen, wie beispielsweise UltraSparcII, Pentium III oder AMD-Athlon. Li, Roychoudhury und Mitra modellieren in [43] einen Prozessor mit out-of-order execution und einem Instruktionen-Cache und beschreiben eine Methodik, die Ausführungszeiten für dieses Modell zu berechnen. Sie berücksichtigen dabei die Abhängigkeit der Ausführungszeit einzelner Basic Blocks von vorangegangenen Befehlen. Das Modell ihrer Pipeline ist jedoch relativ einfach und außerdem wird die Branch Prediction außer Acht gelassen.

Einen Graphen-basierten Ansatz zur Berechnung von Ausführungszeiten von Software präsentieren Puschner und Schedl in [63]. Die Berechnung der WCET wird auf ein Graphentheoretisches Problem abgebildet. Programme werden durch „T-Graphs" (timing graphs) repräsentiert, die die Struktur und das zeitliche Verhalten der Software widerspiegeln. Diese können auch Benutzer-Informationen über nicht ausführbare Pfade enthalten. Der Graph kann in ein ILP-Problem (Integer Linear Programming) abgebildet werden, dessen Lösung die gesuchten WCETs der einzelnen Pfade liefert.

Puschner und Bernat stellen in [62] einen Ansatz vor, mit dem die WCET für wiederverwendbaren Code berechnet werden kann. Im Allgemeinen stützt sich eine WCET-Analyse auf Informationen über den Programmablauf und die Zielhardware, auf der die Software ausgeführt werden soll. Im Falle von portierbarem und wiederverwendbarem Code sind nicht alle Parameter von vornherein bekannt. Die Analyse wird daher in zwei Schritte untergliedert: Im ersten Schritt wird die Berechnung der WCET soweit abstrahiert, dass sie nur von den noch nicht bekannten Parametern abhängt. Somit bleibt auch die WCET Berechnung portierbar. Im zweiten Schritt wird diese abstrakte WCET Darstellung verwendet, um mit Hilfe konkreter Parameter des Zielsystems die WCET zu berechnen.

Lundqvist und Stenström stellen in [48] eine Methodik vor, mit der sich die WCET für Software auf modernen Prozessoren mit Pipelines und Caches mit nur sehr geringer Überabschätzung berechnen lässt. Das Verfahren basiert auf einer Simulation des Befehlssatzes des Prozessors. Dadurch ist es prinzipiell für alle Architekturen anwendbar. Die Simulationstechnik wird dahingehend erweitert, dass Eingabewerte berücksichtigt werden können, und somit nicht ausführbare Pfade und maximale Schleifendurchlaufzahlen berechnet werden können.

Alle diese analytischen Methoden haben den Vorteil, dass sie die WCET für ein prinzipiell beliebiges Programm berechnen können und sich auf unterschiedliche Rechnerarchitekturen anwenden lassen. Das Problem in der Praxis ist dabei die Simulation der Prozessorarchitekturen, die für komplexere Rechner sehr aufwändig und fehleranfällig wird. Auch werden die Interna vieler moderner Prozessoren nicht alle durch die Hersteller dokumentiert und beschrieben, so dass eine hinreichend genaue Simulation gar nicht möglich ist. Die Analyse aller Programmpfade in Abhängigkeit der Eingangsdaten liefert auch schon für kleine Programme riesige Datenmengen, die bearbeitet werden müssen. Um diese Datenmengen zu reduzieren, werden oft Annotationen der Nutzer herangezogen. Für sehr umfangreiche Programme ist diese Methodik kaum anwendbar. Die Interaktion mehrerer Programme bezüglich der Mechanismen der Prozessorarchitektur lassen sich durch analytische Methoden auch nur sehr schwer erfassen, vor allem, wenn es sich um ereignisgesteuerte Systeme handelt. Auch hängt die WCET eines Programms nicht nur von der Prozessorarchitektur ab, sondern auch von der Speicheranbindung des Prozessors. Umfangreichere Programme passen meist nicht mehr vollständig in den Cache, so dass Speicherzugriffe während der Laufzeit unvermeidlich sind. Da die Speicherzugriffszeiten stark variieren können, müssen diese für eine WCET-Analyse unbedingt mit berücksichtigt werden.

Messung von Laufzeiten

Ein anderer Ansatz zur Bestimmung von maximalen Laufzeiten besteht darin, diese direkt auf der Zielhardware zu messen. Einen Überblick über verschiedene Ansätze zur Messung von Laufzeiten gibt Petters in [57]. Grundsätzlich unterscheidet man zwischen „Software Monitoring" und „Hardware Monitoring". Unter „Software Monitoring" versteht man Methoden, die für die Messung Code einfügen. Man unterscheidet zwischen „light weight" und „heavy weight Software Monitoring". Methoden der ersten Kategorie fügen Code ein, der ausschließlich zur Messung dient. Die Methoden der letztgenannten Kategorie fügen nicht nur Code zur Messung ein, sondern manipulieren zusätzlich die Beschleunigungsmechanismen des Prozessors, um für die nachfolgende Messung den Worst-Case zu erzwingen. Die in dieser Arbeit vorgestellte Methode zur Messung von Laufzeiten (siehe Kapitel 4) kann sowohl als „light weight", als auch als „heavy weight Software Monitor" eingesetzt werden. Auch beim „Hardware Monitoring" wird Code zur Messung eingefügt, die Daten werden dann auf einen externen Port des Prozessors gesendet. Solche Portzugriffe sind in der Regel recht zeitintensiv und meist auch nur bei Mikrocontrollern anwendbar.

In den Arbeiten [55], [56] und [59] stellt Petters eine Methodik vor, Ausführungszeiten auf Prozessoren mit Caches, Pipelines und Branch Prediction durch Messung zu ermitteln. Er nutzt dazu den CFG (Control Flow Graph), den ein Compiler zur Optimierung der Software erstellt. Die einzelnen Pfade im Programm werden mit Code zur Messung instrumentiert. Zusätzliche Informationen wie maximale Schleifendurchlaufzahlen müssen vom Anwender in Form von Annotationen im Quelltext angegeben werden. Das Prinzip der Messung beruht auf dem Time-Stamp Counter der verwendeten Prozessoren (Intel-Pentium III, AMD-Athlon). Die Messroutine in dieser Arbeit (siehe Kapitel 4) basiert auf jener von Petters, daher soll an dieser Stelle nicht näher darauf eingegangen werden.

Colin und Petters untersuchen in [22] die Auswirkungen einzelner Beschleunigungsmechanismen moderner Prozessoren auf die Laufzeit von Software. Diese Untersuchungen werden anhand einiger Benchmarks und praktischer Programmbeispiele durchgeführt. Die Autoren kommen zu dem Schluss, dass der entscheidende Faktor für die Laufzeit die Caches sind (sowohl Instruktionen- als auch Daten-Cache). Die anderen Faktoren wie die out-of-order execution und die Branch Prediction können dagegen vernachlässigt werden. Sie werden durch den grundsätzlich vorhandenen Pessimismus der Methodik mit einbezogen.

Lundqvist und Stenström untersuchen in [49] das Laufzeitverhalten von Prozessoren mit dynamischer Befehlsausführung. Alle moderneren, schnellen Prozessoren verfügen über eine dynamische Befehlsausführung. Sie stellen dabei fest, dass unter bestimmten Bedingungen beispielsweise ein Cache-Miss eine kürzere Ausführungszeit zur Folge haben kann als ein Cache-Hit. Diese Ergebnisse sind wichtig für alle Ansätze zur Bestimmung der WCET mit Hilfe einer Messung. Diese versuchen alle, vor der Messung den Worst-Case herzustellen und daraus die WCET zu bestimmen. Diese Arbeit zeigt, in welchen Situationen solche Anomalien entstehen können und wie man sie vermeiden kann.

Wenzel, Rieder, Kirner und Puschner präsentieren in [86] und [87] einen neuen Ansatz, eine vollständige WCET-Analyse mit Hilfe von Messungen durchzuführen. Die Methodik basiert auf dem CFG eines Programms, der in mehrere Teilgraphen unterteilt wird, um die Anzahl der notwendigen Messungen auf ein zu bewältigendes Niveau zu reduzieren. Die gemessenen Zeiten werden dann dazu verwendet, die WCET zu berechnen.

Einen ganz anderen Ansatz zur Bestimmung der WCET mit Hilfe von Messungen verfolgen Burns und Edgar in [12]. Da man nie sicher sagen kann, ob die WCET tatsächlich während einer Messung aufgetreten ist, versuchen die Autoren, die Messergebnisse für eine Extremwertstatistik zu verwenden. Dabei werden die maximalen gemessenen Werte mittels einer Extremwertfunktion angenähert, um über den Verlauf dieser Funktion eine sichere Abschätzung der tatsächlichen WCET vornehmen zu können.

Dieser Ansatz wird von Bernat, Colin und Petters in [8] aufgegriffen. Sie führen sogenannte „probabilistic hard real-time systems" ein. Diese Systeme müssen auch ihre Deadlines einhalten, aber es genügt eine hinreichend hohe Wahrscheinlichkeit dafür als Realzeitnachweis. Um einen entsprechenden statistischen Nachweis führen zu können, werden Methoden der Messung und der analytischen Bestimmung von Ausführungszeiten kombiniert. Dabei werden die während einer Messung beobachteten Zeiten eines Basic Blocks mit statistischen Methoden zur Modellierung der WCET eines Programmpfades herangezogen.

Einen Überblick über Methoden zur Bestimmung von Ausführungszeiten im Worst-Case geben Puschner und Burns in [61]. Sie fassen die wichtigsten Ansätze aus zehn Jahren Forschung auf diesem Gebiet zusammen (etwa 1990-2000). Sie definieren klar die Ziele und die Problemstellungen der WCET-Analyse. Die Arbeiten auf diesem Gebiet werden nach Forschungsgruppen geordnet vorgestellt.

Kirner und Puschner geben in [41] einen Überblick über verschiedene Aspekte der WCET-Analyse. Sie definieren grundlegende Begriffe und klassifizieren verschiedene Tools zur Bestimmung der WCET nach deren Eignung für bestimmte Anforderungen an die WCET-Analyse.

Sie geben dabei einen Überblick über die prinzipielle Vorgehensweise einer WCET-Analyse, die sich in den Entwurfsfluss moderner Software integrieren lässt.

2.2 Verwendung von Caches in Realzeitsystemen

Die Caches sind ein wesentliches Architekturmerkmal aller modernen und schnellen Prozessoren. Sie sorgen dafür, dass die sehr schnell arbeitenden Prozessoren möglichst ohne Unterbrechung mit neuen Instruktionen und Daten versorgt werden. Die vergleichsweise langsamen Hauptspeicher können dies nicht leisten. Je größer der Geschwindigkeitsunterschied zwischen Speicher und Prozessor wird, desto wichtiger werden die Caches. Allerdings sind die Caches auch für starke Schwankungen in der Ausführungszeit von Programmen verantwortlich, die auftreten, weil manche Zugriffe in den Cache gehen und manche in den Hauptspeicher, da in der Regel nicht alle benötigten Instruktionen und Daten in den Caches vorrätig gehalten werden können.

Viele Forschungsgruppen haben sich mit den Einflüssen der Caches auf die Laufzeit von Software beschäftigt. Gerade im Hinblick auf Realzeitsysteme ist es sehr wichtig, maximale Ausführungszeiten sicher bestimmen zu können. Die Caches haben an der Entstehung der WCET einen großen Anteil. Die folgenden Abschnitte geben einen Überblick über wichtige Arbeiten auf diesem Gebiet.

Laufzeitanalyse mit Caches

Viele Arbeiten beschäftigen sich mit einer statischen Cache-Analyse. Sie untersuchen den Code auf Speicherzugriffe und versuchen, diese eindeutig als Cache-Hit oder als Cache-Miss zu klassifizieren. Diese Ansätze beschränken sich jedoch häufig auf sehr einfache Cache-Modelle und berücksichtigen nur einen Instruktionen- oder nur einen Daten-Cache. Weitergehende Ansätze verbinden die Analyse der Speicherzugriffe mit einer Modellierung der ganzen Prozessorhardware, speziell der Pipelines.

Li, Malik und Wolfe stellen in [44] eine Methodik vor, eine Programmpfadanalyse mit einer Modellbildung der Hardware zu verbinden, um daraus die WCET zu bestimmen. Der Ansatz geht von einem direct-mapped Cache und einem einfachen Modell einer Pipeline aus. Aus dem CFG des Programms wird ein CCG (Cache Conflict Graph) für jede Cacheline generiert, mit dessen Hilfe Speicherzugriffe als Hits oder Misses klassifiziert werden können. Diese Informationen werden in ein ILP-Problem transformiert, dessen Lösung die Laufzeit des Programms ergibt. Dieser Ansatz wird in [45] für Set-assoziative Caches erweitert. Es werden auch Daten- und Unified-Caches in die Analyse mit einbezogen.

Einen ähnlichen Ansatz verfolgen Hergenhan und Rosenstiel in [37] für eine statische Laufzeitanalyse für Embedded Software. Sie präsentieren eine ganz ähnliche Methodik zur Laufzeitanalyse, die ebenfalls auf der Modellierung der Caches und der Pipeline beruht. Diese Vorgehensweise wird am Beispiel eines PPC403 und MPC750 demonstriert.

Mueller *et al.* beschäftigen sich in [34] mit der Laufzeitanalyse für Prozessoren mit einer einfachen Pipeline und einem Instruktionen-Cache. Diese Methodik wird anhand eines MicroSPARC-I-Prozessors erläutert. Sie führen vier Zustände für Instruktionen ein, die ihr Verhalten in Bezug auf den Cache widerspiegeln. Diese Zustände (Always-Hit, First-Hit, First-Miss, Always-Miss) haben eine unterschiedliche Bedeutung, je nach dem, ob eine Best-Case oder Worst-Case Analyse vorgenommen wird. Sie erlauben es, eine differenziertere Abstufung des Verhaltens einer Instruktion bezüglich des Caches vorzunehmen. In [53] wird dieses Verfahren speziell für Set-assoziative Instruktionen-Caches verfeinert und analysiert. In [88] wird ein Tool-basierter Ansatz vorgestellt, der basierend auf der Arbeit in [34] die Analyse des Instruktionen-Caches um eine Analyse des Daten-Caches erweitert.

Ferdinand und Wilhelm stellen in [27] ein Verfahren vor, in dem mit der bekannten Technik der „Abstract Interpretation" das Cache-Verhalten eines Programms analysiert wird. „Abstract Interpretation" ist eine Methodik, die die Semantik eines Programms nutzt, um statisch, also vor der Ausführung eines Programms, dessen dynamisches Verhalten vorherzusagen. Die in dieser Arbeit beschriebene Vorgehensweise ermöglicht es, Speicherzugriffe dahingehend klassifizieren zu können, ob sie ein Cache-Hit oder ein Cache-Miss sein werden. Das Verfahren kann auf Instruktionen- und Daten-Caches angewendet werden.

Xue und Vera stellen in [89] ein analytisches Modell zur statischen Analyse der Datennutzung von Programmfragmenten vor. Im Gegensatz zu bisherigen Ansätzen ist dieses Modell nicht auf die Analyse in sich geschlossener Schleifenkonstruktionen beschränkt. Zentrale Punkte des Ansatzes sind das Auflösen von Funktionsaufrufen innerhalb von Schleifen, die Nutzung von Vektoren zur Beschreibung von Speicherzugriffen und eine Analyse, welche Speicherstellen wiederverwendet werden. Der beschriebene Ansatz bezieht sich ausschließlich auf virtuell indizierte Daten-Caches.

In [85] präsentieren Vera, Lisper und Xue einen Ansatz, Daten-Caches in Systemen mit preemptiven Multitasking anzuwenden. Sie untersuchen verschiedene Techniken wie die Partitionierung von Caches, dynamisches Cache-Locking und eine statische Cache-Analyse, ob diese für die Vorhersagbarkeit der Nutzung des Daten-Caches geeignet sind. Sie schlagen eine Kombination dieser drei Techniken vor, um die Daten-Nutzung besser analysieren zu können.

Cache-Locking für Realzeitsysteme

Unter Cache-Locking versteht man die Möglichkeit, Teile des Codes oder der Daten permanent im Cache zu halten. Man unterscheidet prinzipiell zwischen *statischem* und *dynamischem* Cache-Locking. Beim statischen Cache-Locking werden die entsprechenden Inhalte einmal in den Cache geladen und bleiben dann für die gesamte Lebenszeit des Systems dort. Beim dynamischen Cache-Locking werden immer wieder neue Inhalte in den Cache geladen und nur für die Laufzeit bestimmter Programmteile dort gehalten. Dies kann man beispielsweise bei Taskwechseln anwenden, wenn jede Task individuelle Daten im Cache vorrätig haben soll.

Vera, Lisper und Xue stellen in [84] eine Technik vor, die eine Daten-Cache Analyse zur Compilezeit mit Daten-Cache Locking kombiniert, um so die Speicherzugriffszeiten im Worst-Case

bestimmen zu können. Dazu werden zunächst jene Teile im Code, die statisch nicht analysiert werden können, im Cache abgelegt. Zusätzlich werden Daten, die mit hoher Wahrscheinlichkeit benötigt werden, ebenfalls in den Cache geladen. Zusammen mit einer statischen Analyse kann dann das Laufzeitverhalten des Programms bestimmt werden.

Puaut beschreibt in [60] Untersuchungen, die die Modellierung der Caches mit der Methodik des statischen Cache-Lockings vergleichen. Ziel ist es, die Vor- und Nachteile beider Verfahren zu beleuchten. Der Schwerpunkt der Betrachtungen liegt auf der Eignung für die Vorhersage der WCET und durch die Caches verursachte Laufzeitverlängerungen bei Taskunterbrechungen. Arnaud und Puaut gehen in [6] auf die Methode des statischen Cache-Lockings speziell für Instruktionen Caches ein. Sie beschreiben die aktuellen Vorgehensweisen und deren Umsetzung.

Campoy, Perles Ivars und Busquets-Mataix stellen in [17] und [19] einen genetischen Algorithmus vor, der ein Set an Instruktionen auswählt, welches in den Cache geladen werden soll. Diese Instruktionen werden so ausgewählt, dass das Programm einen maximalen Laufzeitgewinn erfährt. Einen ähnlichen Ansatz präsentieren die Autoren in [18] für ein dynamisches Cache-Locking. Jede Task hat ihr Set an Code, der im Cache vorrätig sein soll. Bei jedem Taskwechsel wird das jeweils alte Set durch das neue verdrängt.

Die Autoren untersuchen in [21] das Verhalten im Worst-Case für Systeme, die statisches Cache-Locking benutzen und für Systeme mit herkömmlichen Caches. Sie zeigen, dass eine bessere Vorhersagbarkeit mit Cache-Locking ohne Leistungsverlust des Systems erreicht werden kann.

Campoy, Perles Ivars, Rodríguez und Busquets-Mataix vergleichen in [20] statisches und dynamisches Cache-Locking in Bezug auf ihre Eignung für Realzeitsysteme. Statisches Cache-Locking verbessert die Vorhersagbarkeit eines Systems besser als dynamisches Cache-Locking. Dagegen erreicht man mit dynamischem Cache-Locking für die meisten Systeme eine höhere nutzbare Rechenleistung.

Anordnung von Code und Daten zur besseren Nutzung der Caches

Die Nutzung der Caches korrespondiert direkt mit der Nutzung des Speichers, unabhängig davon, ob es sich um physikalisch oder virtuell indizierte Caches handelt. Die bisher vorgestellten Ansätze haben sich damit beschäftigt, das Programmverhalten in Bezug auf die Caches zu analysieren und vorherzusagen, welche Speicherzugriffe in den Cache gehen und welche nicht. Ein weiterer Schritt ist, Teile eines Programms permanent im Cache zu haben. Die Ansätze in diesem Abschnitt behandeln die Frage, wie man Code und Daten im Speicher anordnen kann, um die Caches möglichst optimal zu nutzen.

Petrank und Rawitz zeigen in [54], dass es im Allgemeinen nicht möglich ist, für ein beliebiges Programm eine Anordnung im Speicher anzugeben, die die Anzahl der Cache-Misses minimiert. Sie schlagen daher vor, für bestimmte Szenarien passende Heuristiken zur Anordnung von Code und Daten zu entwickeln.

Hashemi, Kaeli und Calder stellen in [33] einen Algorithmus vor, der zur Compile-Zeit Code so anordnet, dass der Instruktionen-Cache möglichst optimal genutzt werden kann. Sie bilden dazu Funktionen auf Cachelines ab. Als Parameter verwenden sie die Funktionsgröße, Cachegröße, Cacheline-Größe und den CFG des Programms. Funktionsaufrufe sind im CFG entsprechend ihrer Aufrufhäufigkeit gewichtet. Zuerst werden die wichtigen Funktionen auf Cachelines abgebildet und dann schrittweise die nicht so häufig aufgerufenen. Der Algorithmus kann auf direct-mapped und Set-assoziative Caches angewandt werden. Statt auf Basis von Funktionen kann auch auf Basis von Basic Blocks angeordnet werden. Dies setzt jedoch voraus, dass Funktionen im Speicher aufgesplittet werden können. Die Autoren schreiben, dass ihre Vorgehensweise die Rate der Cache-Misses im Durchschnitt um 40% senkt.

Calder, Krintz, John und Austin stellen in [16] einen Algorithmus vor, um Daten so anzuordnen, dass der Daten-Cache möglichst optimal genutzt werden kann. Diese Arbeit baut auf [33] auf. Es werden globale Variablen, der Stack, Heap-Speicher und Konstanten berücksichtigt. Die Daten werden im Hinblick auf die Anzahl ihrer Referenzen, ihrer Größe und ihrer Lebensdauer analysiert. Die Lebensdauer eines Datums ist die Zeit zwischen der ersten und der letzten Referenz auf dieses Objekt. Diese Daten werden durch einen TRG (Temporal Relationship Graph) repräsentiert, der für die Optimierung verwendet wird. Die Autoren stellen fest, dass ihr Ansatz die Anzahl der Daten-Cache Misses im Durchschnitt um 24% verringert.

Tomiyana und Yasuura stellen in [80] einen Ansatz vor, um Code für Embedded Software so zu platzieren, dass die Hit-Rate maximal wird. Sie verwenden dafür nicht Funktionen oder Basic Blocks, sondern Traces. Ein Trace ist eine Abfolge von Basic Blocks (Programmpfad). Das Verfahren kann auf direct-mapped und Set-assoziative Caches angewandt werden. Die Autoren geben eine um 35% reduzierte Cache-Miss Rate im Durchschnitt an. Ein Nachteil der vorgeschlagenen Methodik ist, dass sie die Codegröße insgesamt erhöht, was gerade im Embedded-Bereich von Nachteil ist. Daher präsentieren sie in [81] eine verbesserte Vorgehensweise, die diesen Nachteil nicht aufweist. Der Nutzen bleibt dabei erhalten. In [82] fassen die Autoren beide Versionen zusammen und vergleichen sie.

Auf dem Gebiet der Nutzung von Caches für Realzeitsysteme sind in den letzten Jahren viele Fortschritte erzielt worden. Viele Ansätze für eine Analyse der Interaktionen eines Programms mit den Caches gehen von stark vereinfachten Szenarien aus, die die Wirklichkeit für PC-Prozessoren nicht hinreichend gut abbilden. Oft wird auch nur ein Instruktionen- oder nur ein Daten-Cache betrachtet. Die Ansätze zur Optimierung der Speicheranordnung gehen da schon weiter, berücksichtigen aber keine TLBs oder verschiedene Strategien des Cachings.

Auch wird meist davon ausgegangen, dass die Laufzeitverzögerungen für einen Cache-Miss immer gleich sind, egal ob es sich um einen Miss im Instruktionen- oder Daten-Cache handelt. Auch Misses mit vorherigem Zurückschreiben werden nicht betrachtet, ebenso wie mehrere Cache-Ebenen. Prefetching-Effekte, die eine große Rolle bei der praktischen Bewertung der Laufzeitverzögerungen durch Caches spielen, werden ebenso nicht berücksichtigt.

2.3 PCs als Realzeitsysteme

PC-Systeme sind nicht für den Einsatz in Realzeitanwendungen konzipiert. Sie sind darauf ausgelegt, im Durchschnitt eine hohe Rechenleistung zu erbringen. Sporadisch auftretende WCETs spielen dabei keine Rolle. Aber gerade die hohe Rechenleistung dieser Systeme macht sie auch für einen Einsatz im Realzeitbereich attraktiv. PC-Systeme sind sehr komplex und durch ihren modularen Aufbau auch sehr vielfältig. Es gibt daher nicht viele Forschungsgruppen, die sich mit den Realzeiteigenschaften von PC-Systemen befassen.

Sebek gibt in [67] einen umfassenden Überblick über Cachearchitekturen von PC-Prozessoren und deren Funktionsweise. Er legt detailliert alle Aspekte des Cachings dar und untersucht die Einflüsse der Caches auf die Rechenleistung anhand eines Pentium III, eines MPC750 und eines StrongARM SA-1110. Er gibt einen Überblick über die Einflüsse von PC-Hardware im Allgemeinen auf Realzeitsysteme, wobei der Schwerpunkt auch hier auf den Caches und Speicherzugriffen liegt. Er erklärt kurz gängige Methoden der Realzeitanalyse für Systeme mit Caches.

Einen Überblick über den Einsatz von PC-Komponenten in Realzeitsystemen gibt Burmberger in [11]. Er untersucht die Einsatzgebiete von PCs als Realzeitsysteme und vergleicht die dafür verwendeten Realzeitbetriebssysteme. Er teilt die untersuchten Komponenten verschiedenen Systemgruppen zu und untersucht sie hinsichtlich der Einhaltung bestimmter Kriterien für einen Einsatz in einem Realzeitsystem.

Srinivasan, Pather, Hill, Ansari und Niehaus stellen in [70] ihre Implementierung des „firm real-time system" KURT vor (Kansas University Real-Time). KURT basiert auf Linux und erweitert dies um die Eigenschaften eines Realzeitbetriebssystems. Es ist allerdings weder für harte Echtzeit, noch für weiche Echtzeit ausgelegt, daher die Bezeichnung „firm". Die Änderungen beruhen im Wesentlichen darauf, dass die Auflösung des Linux-Timers erhöht wird, um die Reaktivität des Systems zu steigern. Es wird neben den bereits existierenden Linux-Schedulern ein KURT-Scheduler implementiert, der den Tasks die Rechenzeit nach einem festen Plan zuteilt. Dieser Plan wird a-priori erstellt und ändert sich nicht zur Laufzeit des Systems (statisches Scheduling). KURT implementiert drei Betriebsmodi: „normal mode", in dem die KURT Erweiterungen inaktiv sind, „mixed real-time mode", in dem KURT-Applikationen nach Plan und Linux-Applikation in der restlichen Zeit ausgeführt werden und „focused real-time mode", in dem nur KURT-Applikationen ausgeführt werden.

Gopalan untersucht in [31] die Grundlagen und die Techniken für die Erweiterung eines GPOS zu einem RTOS. Er legt dar, wo die Unterschiede liegen und welche Techniken genutzt werden können, um ein Standardsystem mit GPOS echtzeitfähig zu machen. Er geht dabei auf die Ressourcenzuteilung und das Scheduling ein. Er untersucht verschiedene Implementierungen von Echtzeiterweiterungen für ein GPOS hinsichtlich ihrer Eignung für ein Realzeitsystem (darunter auch KURT und RT-Linux).

Mehnert, Hohmuth und Härtig vergleichen in [51] zwei Implementierungen von Echtzeiterweiterungen für Linux: RT-Linux und L4RTL. L4RTL ist eine Echtzeiterweiterung ähnlich wie RT-Linux, arbeitet jedoch mit unterschiedlichen Adressräumen für den Echtzeit- und den Nicht-Echtzeitanteil. Der Nicht-Echtzeitanteil wird komplett im User-Mode ausgeführt, nur der Echt-

zeitanteil arbeitet im Kernelmode. Sie untersuchen vor allem den Jitter der Interruptreaktionszeiten, der durch einen Wechsel der Adressräume unter verschiedenen Lastszenarien entsteht. Die Autoren stellen fest, dass die Verzögerungen für einen Wechsel der Adressräume in der Größenordnung jener Verzögerungen durch Cache Misses liegen.

In [46] stellen Liedtke, Härtig und Hohmuth ein Verfahren zum Aufteilen des Caches unter verschiedenen Tasks vor (Cache Colouring). Sie partitionieren mit Hilfe des Betriebssystems den Cache und weisen diese Partitionen unterschiedlichen Tasks zu. Im Vergleich zu anderen Ansätzen dieser Art benötigt dieses Verfahren keine Änderungen an der Hardware. Sie zeigen diese Vorgehensweise anhand von zwei einfachen Beispielen.

Schoenberg untersucht in [66] den Einfluss des PCI-Busses auf die Laufzeiten von Software. Die Untersuchungen beziehen sich auf einen Ein-Prozessor-PC. Er definiert einen „Slowdown Factor" um den Einfluss des PCI-Busses zu beschreiben. Dieser wird durch Messungen ermittelt. In [65] untersucht der Autor verschiedene Ansätze, um die Echtzeiteigenschaften des PCI-Arbiters zu verbessern. Diese gehen allerdings alle von Hardwaremodifikationen aus.

Der Trend für den Einsatz eines PC-Systems als Realzeitsystem geht dahin, ein Standardbetriebssystem um die Fähigkeiten eines Echtzeitbetriebssystems zu erweitern. Diese Ansätze verfolgen beispielsweise KURT, RT-Linux oder RTAI. Diese werden auch bereits in der Industrie genutzt. Die meisten implementieren ein eigenes Interruptmanagement und einen Realzeitscheduler. Aspekte des Speichermanagements, vor allem hinsichtlich der Nutzung der Caches, werden dabei kaum oder gar nicht berücksichtigt.

3 Grundlagen

In diesem Kapitel sollen die Grundlagen für das Verständnis der nachfolgenden Kapitel vermittelt werden. Zunächst wird auf die zentralen Begriffe *Realzeit* und *Realzeitbetriebssystem* eingegangen. Der nächste Abschnitt beschäftigt sich mit den wichtigsten Grundlagen zur PC-Architektur, auf deren Basis die Untersuchungen in dieser Arbeit durchgeführt wurden. Dabei wird auf die Prozessorarchitektur, den Aufbau von Multiprozessorsystemen und auf die verwendeten Bussysteme zur Ankopplung externer Peripherie eingegangen. Der Schwerpunkt in diesem Kapitel liegt auf dem Aufbau und der Funktionsweise von Caches. Dies bezieht sich nicht nur auf PC-Prozessoren, sondern gilt auch für andere Prozessoren mit ähnlichen Caches wie beispielsweise PowerPC. Abschließend wird noch ein Blick auf die verwendeten Speichertechnologien und deren Funktionsweise geworfen.

3.1 Realzeitsysteme

Realzeitsysteme unterscheiden sich von anderen Rechnersystemen dadurch, dass sie nicht nur korrekte Ergebnisse liefern müssen, sondern dass diese auch innerhalb einer bestimmten Zeit vorliegen müssen. Ist dies nicht der Fall, ist das Ergebnis unbrauchbar. Abhängig von den Folgen für den technischen Prozess wird zwischen *harter* und *weicher* Echtzeit unterschieden:

- **weiche Echtzeit:** Erfüllt ein System im Mittel die Echtzeitanforderungen, so spricht man von weicher Echtzeit. Die Ergebnisse, die zu spät vorliegen, verursachen keine schlimmen Folgen und sind in der Regel zumindest noch von geringem Nutzen.
- **harte Echtzeit:** Die Verletzung einer Zeitanforderung des Systems (Deadline) führt hier unmittelbar zum maximalen Schaden für den technischen Prozess. Ein zu spät eintreffendes Resultat ist absolut wertlos.

Beispiele für Prozesse, welche weichen Echtzeitanforderungen unterliegen, sind die Verarbeitung und Übertragung von Video- und Audiodaten. Harte Echtzeitanforderungen werden zum Beispiel im Automobilbereich an Fahrzeugsteuerungen oder an viele Teilsysteme im Luftfahrtbereich gestellt.

Der Begriff Echtzeit wird in der Informatik oft benutzt, um eine Abgrenzung zur *Modellzeit* herzustellen. Die *Modellzeit* ist jene Zeit, die von der Software als Laufzeit selbst verwaltet wird, die Echtzeit ist die Zeit, die Abläufe in der realen Welt benötigen. Sind diese beiden Zeiten für eine Anwendung identisch, so spricht man davon, dass diese Anwendung in Echtzeit abläuft.

Realzeitbetriebssysteme

Realzeitbetriebssysteme (Real-Time Operating System, RTOS) unterscheiden sich von Standardbetriebssystemen (General Purpose Operating System, GPOS) dadurch, dass sie auf die Einhaltung von Zeitbedingungen und die Vorhersagbarkeit des Prozessverhaltens optimiert wurden. Sie verfügen über echtzeitfähige Scheduler, während die Scheduler eines GPOS auf eine faire Zeitaufteilung zwischen allen Prozessen und eine möglichst hohe Reaktivität von interaktiven Prozessen ausgelegt sind.

Meist ist ein RTOS wesentlich kompakter ausgelegt als ein GPOS und interne Abläufe wie die Verarbeitung und Weitergabe von Prozess-Signalen laufen wesentlich schneller ab. Oft werden sie in kleinen *embedded* Systemen eingesetzt. Ihre Kompaktheit erreichen sie in der Regel dadurch, dass sie nur die unbedingt notwendige Funktionalität bieten.

Es gibt viele Beispiele für Realzeitbetriebssysteme wie VxWorks, RTEMS oder QNX. Speziell im PC-Bereich wird der Ansatz verfolgt, ein GPOS um die Fähigkeiten eines RTOS zu erweitern und Realzeit- und Standardapplikationen parallel auf einem Rechner zu bearbeiten. Beispiele für diese Ansätze sind RT-Linux und RTAI. Dieses Konzept wird in Abschnitt 5.3.2 näher erläutert.

3.2 PC-Architekturen

Dieser Abschnitt beschreibt den prinzipiellen Aufbau von modernen PC-Systemen. Dazu wird zunächst die Architektur der in diesem Bereich verwendeten Prozessoren beschrieben. Weitere Themen sind der generelle Aufbau eines Multiprozessorsystems und die Anbindung der Peripherie, insbesondere des Hauptspeichers. Die Ausführungen in diesem Abschnitt sollen dazu dienen, einen Überblick über den Aufbau und die Funktionsweise eines PC-basierten Rechensystems zu bekommen. Die Auswirkungen der jeweiligen Architekturmerkmale auf die Laufzeit von Software sind Gegenstand von Kapitel 4.

3.2.1 Prozessorarchitekturen

In heutigen PC-Systemen werden hauptsächlich Prozessoren verwendet, die die IA-32 Architektur implementieren. Das sind alle Prozessoren der Intel-Pentium/Celeron und der AMD-Athlon/Duron Familie. Bild 3.1 zeigt stellvertretend für diese Klasse von Prozessoren das Blockschaltbild des AMD-Athlon Prozessors. Dort kann man alle wesentlichen Merkmale dieser hoch komplexen Prozessoren erkennen. Die im Laufe der Entwicklung immer weiter gewachsene Komplexität ist auf das Bestreben zurückzuführen, die Ausführungsgeschwindigkeit von Software auf dem Prozessor zu steigern.

Der Prozessor hat zwei Cache-Ebenen, einen L1-Cache, der zu gleichen Hälften in einen Instruktionen- und einen Daten-Cache unterteilt ist, und einen L2-Cache, der für Code und Daten gleichermaßen genutzt werden kann. Details zu den Caches werden in Abschnitt 3.3 behandelt.

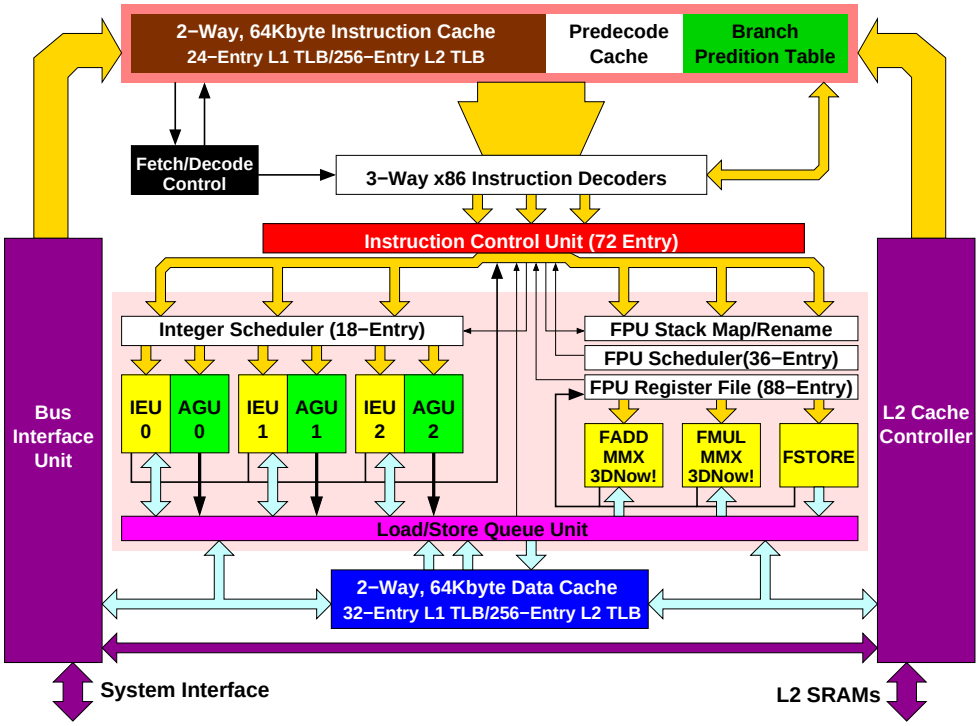

Bild 3.1: Blockschaltbild des AMD-Athlon Prozessors (aus [3])

Alle IA-32 kompatiblen Prozessoren verfügen über einen *CISC*-Befehlssatz (*Complex Instruction Set Computing*), der intern in einen *RISC*-Befehlssatz (*Reduced Instruction Set Computing*) übersetzt wird. Diese RISC-Befehle werden dann ausgeführt. Die Umsetzung von CISC nach RISC erfolgt über die Dekoder, welche die dekodierten Instruktionen in der *Instruction Control Unit* ablegen, von wo aus sie auf die *Instruction Execution Units (IEU)* verteilt werden. Es wird grundsätzlich zwischen Integer und Floating Point Operationen unterschieden. Die *Address Generation Units* (AGU) geben ihre Ergebnisse (Adressen) in die Load/Store Queue, von wo aus die Operationen ausgeführt werden (Speicherzugriffe oder die Weiterleitung an wartende, von den Daten abhängige Instruktionen).

Ein wesentliches Merkmal dieser Prozessoren ist auch die *Branch Prediction*, welche durch eine Vorhersage des Verhaltens von Sprungbefehlen versucht, Pipeline Stalls zu vermeiden und damit die Programmausführung zu beschleunigen. Zu diesem Mechanismus gehört auch die *Branch Prediction Table*. Als *Predecode Cache* wird eine Erweiterung des L1-Instruktionen-Caches bezeichnet: Zu jeder Cacheline werden zusätzliche Informationen gespeichert, welche angeben, wo innerhalb der Cacheline Befehlsgrenzen liegen und ob es sich um Sprungbefehle handelt. Diese Information wird während des Ladens einer Cacheline generiert.

Die Prozessoren implementieren alle eine *out-of-order execution*, das heißt, sie können Befehle in einer anderen Reihenfolge ausführen als jener, die vom Programmfluss vorgegeben ist. Auch diese Maßnahme dient dazu, die Befehlsabarbeitung zu beschleunigen. Eine detaillierte Betrachtung dieser Mechanismen und ihrer Auswirkungen auf die Laufzeit von Software wird in Abschnitt 4.2 vorgenommen.

3.2.2 Multiprozessorsysteme auf PC-Basis

Es gibt derzeit zwei unterschiedliche Formen von Multiprozessorsystemen in der PC-Welt: SMP- und NUMA-Systeme. Das Kürzel SMP steht für „Symmetric Multiprocessing", NUMA steht für „Non-Uniform Memory Access". Die beiden Systeme unterscheiden sich hauptsächlich in der Anbindung des Speichers an den Prozessor. Die Form der Anbindung spielt eine große Rolle im Hinblick auf die Frage, ob PC-Systeme als Realzeitsysteme eingesetzt werden können. Zunächst wird die SMP-Architektur näher beleuchtet.

Bild 3.2 zeigt den schematischen Aufbau eines Dual PC-Systems mit SMP-Architektur.

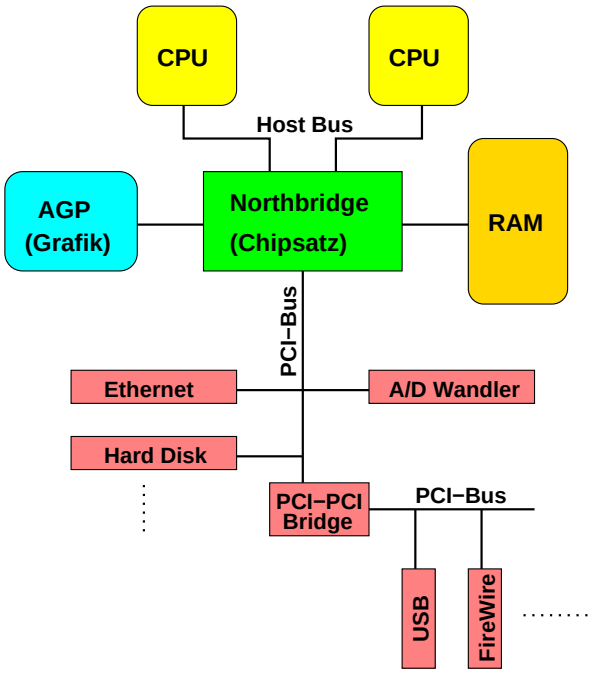

Bild 3.2: Dual-SMP PC-Architektur

Die Prozessoren sind über die Northbridge an die Peripherie gekoppelt. In Bild 3.2 sind zwei Prozessoren mit jeweils einer eigenen Verbindung zum Chipsatz gezeigt. Diese Variante wird von AMD implementiert, Intel verwendet einen gemeinsamen Host-Bus für alle Prozessoren. Dabei ist die Zahl der Prozessoren nicht auf zwei beschränkt, es gibt auch vier-, acht- oder 16-fach Systeme. Der gemeinsame Weg über die Northbridge bzw. den gemeinsamen Host-Bus ist ein Kollisionspunkt bei Speicherzugriffen in SMP-Systemen. Es kann nur ein Prozessor auf den Speicher zugreifen, währenddessen alle anderen Teilnehmer, auch die über den PCI-Bus angekoppelten Peripheriegeräte, warten müssen.

Die Peripheriegeräte sind in der Regel über den PCI-Bus mit dem Chipsatz verbunden. Dabei gibt es die Möglichkeit, mehrere PCI-Busse über Bridges miteinander zu koppeln. Auch andere Kommunikationssysteme wie USB und FireWire werden über den PCI-Bus an das System gekoppelt. Der AGP-Port, der für Grafikkarten konzipiert ist, ist direkt an den Chipsatz gekoppelt.

Im Vergleich zu einem SMP-System zeigt Bild 3.3 den Aufbau eines NUMA-Systems, wie es von AMD eingesetzt wird.

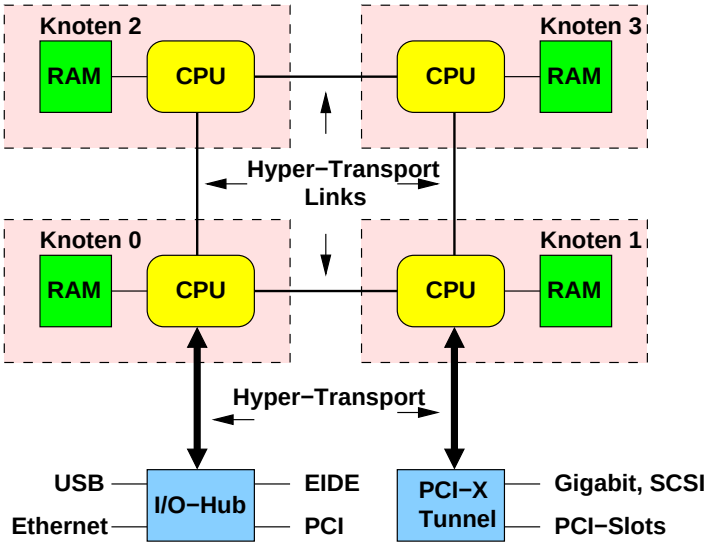

Bild 3.3: NUMA PC-Architektur

Im Gegensatz zu einem SMP-System hat in einem NUMA-System jeder Prozessor seinen eigenen Hauptspeicher. Dies verhindert bei entsprechender Nutzung Wartezeiten beim Zugriff auf den Speicher, wie sie in SMP-Systemen auftreten. Die Prozessoren sind über HyperTransport miteinander gekoppelt. Das ist eine paketorientierte Punkt-zu-Punkt Verbindung, mit der Daten und Interrupts von und zu den Prozessoren verschickt werden.

Die Ankopplung der Peripherie kann über jeden beliebigen Knoten erfolgen, im Beispiel sind dies Knoten 0 und Knoten 1. Über I/O-Hubs können weitere Kommunikationssysteme angeschlossen werden wie USB oder EIDE. Mit PCI-X Tunnel Bausteinen kann beispielsweise ein PCI-X Bussystem an das System gekoppelt werden. Solche Bausteine gibt es auch für andere Bussysteme (PCI, PCI-Express), so dass diese Architektur sehr flexibel ist. Es können durch die Kombination von Hubs und Tunnel-Bausteinen beliebige Anschlüsse von Peripheriegeräten vorgenommen werden. Mehr Details zur HyperTransport-Technologie finden sich in Abschnitt 3.2.4.

3.2.3 Chipsatz

Als Chipsatz werden integrierte Schaltkreise bezeichnet, die die Kopplung der zentralen Komponenten eines PC-Systems realisieren: Prozessor, Speicher, AGP-Port und Bussysteme wie PCI. Es gibt viele verschiedene Chipsätze von unterschiedlichen Herstellern, die meist auf ein bestimmtes System maßgeschneidert sind.

Die Rechenleistung, die ein System erbringen kann, hängt neben den Einzel-Komponenten ganz wesentlich vom Chipsatz ab. Der Chipsatz implementiert die Ansteuerung des Hauptspeichers

und ist daher mit verantwortlich für die Blockierungszeiten beim gleichzeitigen Zugriff mehrerer Geräte. Er regelt alle Datenflüsse von und zu beliebigen Komponenten, vor allem den Prozessoren. Interrupts werden in der Regel nicht vom zentralen Chipsatz verwaltet, sondern über extra Bausteine (IO-APIC) verarbeitet. Der Transport des Interrupt-Signals geschieht jedoch bei manchen Implementierungen über den Host-Bus und ist damit auch wieder abhängig vom Chipsatz.

Der Chipsatz ist eine zentrale Komponente im Bezug auf das Laufzeitverhalten eines PC-Systems. Bei SMP-Systemen ist er die zentrale Schaltstelle, über die nahezu alle Aktionen laufen. In NUMA-Systemen hat jeder Prozessor seinen eigenen Chipsatz, der für die Verbindung zum Speicher, zu den anderen Prozessoren und der Peripherie sorgt. Jeder Chipsatz implementiert eigene Strategien, um zwangsläufig auftretende Blockierzeiten so gering wie möglich zu halten. In [73] werden Chipsätze hinsichtlich ihrer Auswirkungen auf die Laufzeit von Software untersucht.

3.2.4 Bussysteme

Ein modernes PC-System verfügt in der Regel über einen zentralen Bus, meist ein PCI-Bus, an den sämtliche externe Hardware angekoppelt wird. Dazu gehören EIDE und SCSI-Systeme, USB, FireWire und Ethernet. Der weit verbreitete PCI-Bus wird zunehmend durch PCI-Express oder PCI-X Busse abgelöst, auf die in diesem Abschnitt kurz eingegangen werden soll. Noch relativ neu im Einsatz ist das HyperTransport System, was allerdings kein Bussystem im eigentlichen Sinne ist, sondern eine Punkt-zu-Punkt Verbindung.

PCI-Bus

Der auch heute noch am häufigsten eingesetzte Bus ist der PCI-Bus (*Peripheral Components Interconnect*). Er ist in der Nachfolge des ISA/EISA und VESA-Standards entstanden. Die erste Spezifikation 1.0 wurde am 22.6.1992 veröffentlicht, Version 2.0 folgte im April 1993, Version 2.1 wurde Anfang 1995 veröffentlicht. Heute wird meist die nur noch leicht geänderte Version 2.3 verwendet.

Der PCI-Bus ist unabhängig vom Prozessor des Rechners, in dem er betrieben wird. Geräte am PCI-Bus sind PCI-spezifisch, nicht prozessorspezifisch. Der PCI-Bus kann also nicht nur in PC-Systemen mit Intel-kompatiblen Prozessoren verwendet werden, sondern steht auch allen anderen Rechensystemen offen. Er kann mit 33 MHz (V. 2.0) oder 66 MHz (V. 2.1) betrieben werden und verfügt über eine Busbreite von 64 Bit. Alle Transaktionen auf dem Bus sind Burst-Transfers mit variabler Länge, die zwischen den Geräten individuell ausgehandelt wird.

Ein Datentransfer auf dem PCI-Bus läuft immer in zwei Phasen ab: Der Adressphase und der Datenphase. Zunächst wird die Startadresse für den Burst-Transfer übermittelt und dann folgen, je nach Länge des Transfers, mehrere Datenphasen. Das Zielgerät zählt dabei selbstständig die interne Zieladresse hoch. Während eines Datentransfers kann bereits ein neuer Busmaster ausgehandelt werden (*hidden arbitration*), so dass für die Arbitrierung keine zusätzliche Zeit

benötigt wird. Jedes Gerät, dass auf dem PCI-Bus Daten senden will, muss diesen vorher arbitrieren (Master-Slave Hierarchie).

Der PCI-Bus definiert für jedes Gerät einen Konfigurationsbereich, wo die benötigten Ressourcen wie DMA-Kanal oder IRQ-Nummer festgelegt werden. Damit ist es möglich, jedem Gerät automatisch die benötigten Ressourcen zuzuteilen. Pro PCI-Bus stehen vier physikalische Interrupts zur Verfügung. Reichen diese nicht aus, können sich mehrere Geräte einen Interrupt teilen (*interrupt sharing*). Über PCI-to-PCI Bridges können mehrere PCI-Busse gekoppelt werden. Es ist auch möglich, PCI-Busse mit 33 MHz und 66 MHz zu koppeln.

Das Laufzeitverhalten des PCI-Busses hängt zum einen von dem im Chipsatz implementierten Arbiter ab, der den Bus für einen Prozessor arbitriert. Meist ist dies ein Arbiter, der nach einem Round-Robin ähnlichen Schema arbeitet. Zum anderen hängt die Zugriffszeit auf eine Ressource davon ab, wie lange ein vorheriger Burst-Transfer dauert. Da die Burst-Länge individuell zwischen den Kommunikationspartnern verhandelt werden kann, kann diese Zeitspanne stark variieren. Eine eingehende Untersuchung des Zeitverhaltens des PCI-Busses im Hinblick auf eine Nutzung für Realzeitsysteme ist in [73] zu finden. Details zur Funktionsweise und zur technischen Implementierung des PCI-Busses finden sich in [69].

PCI-X ist eine Weiterentwicklung des PCI-Standards, die im Jahre 1998 in der Version 1.0 erstmals spezifiziert wurde. Die Version 2.0 des PCI-X Standards wurde 2002 veröffentlicht. PCI-X versucht, die Transferrate des PCI-Busses durch Double- und Quad-Data Übertragungsverfahren zu erhöhen. Zusätzliche Funktionen wie Split Transactions sollen helfen, die theoretisch möglichen Übertragungsraten auf dem Bus auch tatsächlich zu erreichen. PCI-X ist voll abwärtskompatibel zum PCI-Bus. Nähere Details zum PCI-X Bus finden sich in [68].

PCI-Express ist im Gegensatz zu PCI-X keine Weiterentwicklung des PCI-Busses, sondern ein völlig neues Konzept. Es soll den PCI-Bus in seiner heutigen Form und den AGP-Port ablösen und eine einheitliche Verbindung für alle Geräte bereitstellen. PCI-Express stellt im Gegensatz zum parallel übertragenden PCI-Bus eine serielle Punkt-zu-Punkt Verbindung dar. Aus Softwaresicht ist PCI-Express kompatibel zu PCI, es müssen demnach keine Änderungen am bestehenden PCI-Interface in Betriebssystemen vorgenommen werden. Eine ausführliche Diskussion von PCI-Express würde an dieser Stelle zu weit führen, detaillierte Informationen finden sich in [10]. Eine Tabelle mit den Übertragungskapazitäten der verschiedenen Busse befindet sich am Ende dieses Abschnitts.

HyperTransport

Das HyperTransport System (ursprünglich Lightning Data Transport, LDT) ist ein nicht-properitäres Kommunikationssystem, welches vom HyperTransport Konsortium spezifiziert wurde. Diesem Konsortium gehören über 40 Firmen an, darunter so namhafte wie AMD, Cisco, Apple, Sun oder Transmeta. Da die Spezifikation offen ist, kann jeder seine Hardware für den Einsatz mit HyperTransport konzipieren.

HyperTransport ist kein Bussystem, sondern eine paketorientiertes Punkt-zu-Punkt Netzwerk. Es soll alle Geräte in einem Rechnersystem miteinander verbinden, auch den Prozessor. Es wird

momentan hauptsächlich von AMD für seine Opteron-Systeme eingesetzt, aber auch Apple verwendet HyperTransport für seine G5-Systeme.

HyperTransport ist eine reine Chip-to-Chip Verbindungstechnik und unterstützt demnach keine Steckplätze wie PCI oder PCI-Express. Über Bridges können Bussysteme wie PCI oder PCI-Express angeschlossen werden, USB und FireWire werden beispielsweise über I/O-Hubs an einen HyperTransport Link angeschlossen. Das Software Interface ist kompatibel zu dem von PCI, so dass keine Änderungen am Interface in Betriebssystemen notwendig sind.

HyperTransport überträgt nicht nur Daten von einem Teilnehmer zu einem anderen, auch Interrupts werden als HyperTransport Pakete verschickt. Es werden *split transactions* unterstützt, um Verzögerungen durch wiederholte Datenanforderungen, Verbindungsabbrüche und Warte-Zyklen zu kompensieren. In [73] werden die Einflüsse von HyperTransport auf die Laufzeit von Software untersucht, Details zur technischen Umsetzung finden sich in [83].

Tabelle 3.1 gibt einen Überblick über die verschiedenen Systeme und ihre Transferraten.

Bus	Busbreite (Bit)	Taktfrequenz (MHz)	Transferrate (GB/s)
PCI 2.3	32	33	0,124
PCI 2.3	64	33	0,248
PCI 2.3	32	66	0,248
PCI 2.3	64	66	0,497
PCI-X 1.0	64	133	0,993
PCI-X 2.0/533	64	133 QDR[1)]	3,974
PCI-Express x1	-	2500	0,25
PCI-Express x16	-	2500	4
HyperTransport 1.0	2-32	400-800	bis zu 6,4
HyperTransport 2.0	2-32	1000-1400	bis zu 11,2

Tabelle 3.1: Technische Daten verschiedener Bussysteme

3.3 Caches in PC-Prozessoren

In diesem Abschnitt wird zunächst die Speicherhierarchie beschrieben, wie sie in modernen PC-Systemen implementiert ist. Im Anschluss wird auf den Aufbau und die Organisation von Caches in IA-32 kompatiblen Prozessoren eingegangen. Dazu zählen neben dem Pentium III und dem Pentium IV die AMD-Athlon und AMD-Opteron Prozessoren. Es wird die Abbildung des Hauptspeichers in den Cache beleuchtet und verschiedene Szenarien der Cache-Verdrängung beschrieben. Im letzten Unterpunkt wird auf den Mechanismus zur Synchronisierung der Caches in Multiprozessorsystemen (Cache-Snooping) eingegangen.

[1)] Quad Data Rate

3.3.1 Speicherhierarchie

Als *Speicherhierarchie* in einem Rechnersystem wird die nach Zugriffsgeschwindigkeit geordnete Abfolge der einzelnen Speichereinheiten bezeichnet. Die Speicherhierarchie in modernen PC-Systemen ist in Abbildung 3.4 gezeigt. Die schnellste Speichereinheit sind die Prozessorre-

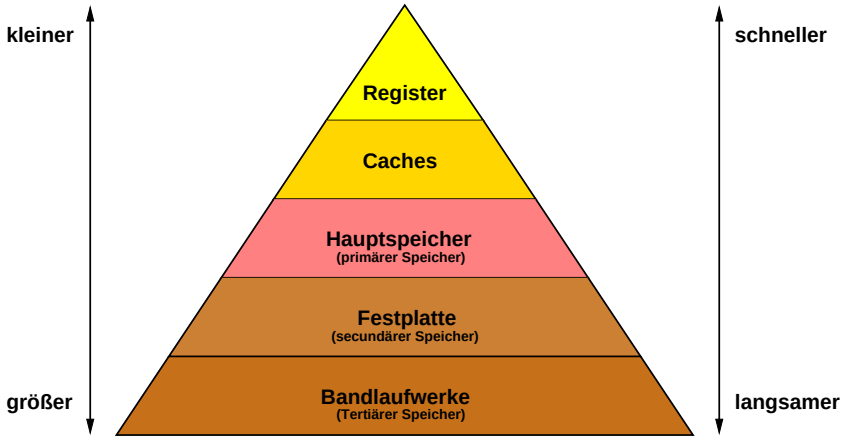

Bild 3.4: Speicherhierarchie

gister, gefolgt vom Cache. Der Cache selbst ist in mehrere Ebenen (Levels) unterteilt. Die erste Ebene (L1) ist die schnellste, gefolgt von der zweiten Ebene (L2) und je nach Implementierung einer dritten Ebene (L3). Die ersten beiden Ebenen befinden sich heute auf dem Prozessor-Die, die dritte Ebene besteht aus direkt an den Prozessor angeschlossenem externen RAM. Der Hauptspeicher bildet die nächste Stufe der Speicherhierarchie, gefolgt vom Festplattenspeicher. Danach folgen CD/DVD-Laufwerke oder andere Speichereinheiten wie Bandlaufwerke. Die Größe des verfügbaren Speichers nimmt mit fallender Zugriffsgeschwindigkeit zu.

3.3.2 Cache-Architekturen

Der Cachespeicher ist die schnellste Speicherebene nach den Prozessorregistern (vgl. Bild 3.4). Moderne Prozessoren verfügen über mindestens zwei Cache-Ebenen, den L1-Cache und den

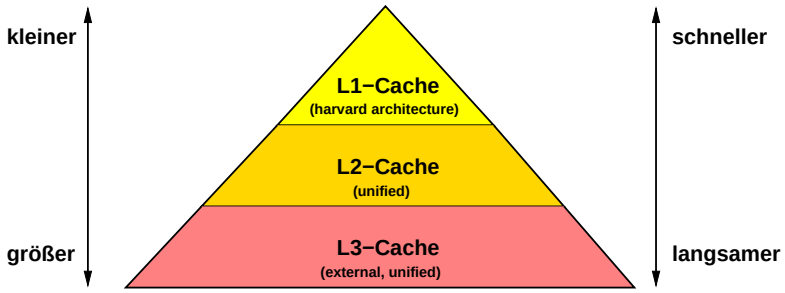

Bild 3.5: Cachehierarchie

L2-Cache. Je nach Prozessormodell kann noch ein L3 Cache vorgesehen sein. Für die Caches kann man eine Speicherhierarchie analog zu der in Bild 3.4 angeben, wie sie Bild 3.5 zeigt.

Es gibt verschiedene Organisationsformen für Caches: *direct-mapped*, *n-fach Set assoziativen* und *voll-assoziativen* Cache. Bild 3.6 zeigt den Aufbau für einen 2-fach Set-assoziativen Cache.

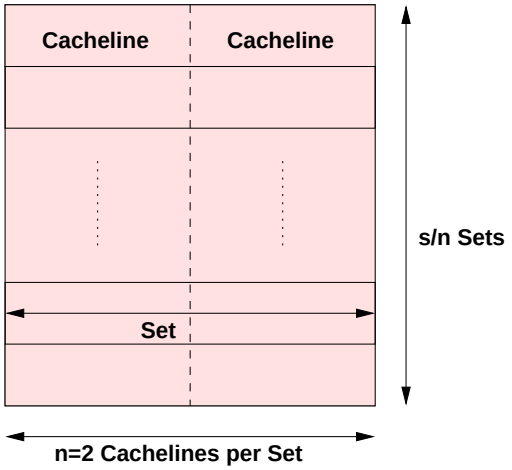

Bild 3.6: Aufbau eines 2-fach Set-assoziativen Caches

Ein *direct-mapped* und ein *voll-assoziativer* Cache können als Sonderfall eines *n-fach Set assoziativen* Caches gesehen werden:

- **direct mapped:** Diese Organisationsform kann auch als 1-fach Set-assoziativer Cache gesehen werden. Das heißt, die Cachelines sind identisch mit den Sets. Daher korrespondiert die Adresse einer Speicherzelle direkt mit der Cacheline, auf die der Speicherbereich abgebildet wird.
- **n-fach Set-assoziativ:** Hier teilen sich *n* Cachelines ein Set, wie in Bild 3.6 für $n = 2$ dargestellt. Das heißt, die Adresse einer Speicherzelle korrespondiert mit einem Set im Cache. Die Entscheidung, welche Cacheline innerhalb eines Sets verdrängt wird, ist für Software transparent. In der Regel wird eine sogenannte *Pseudo Least Recently Used* Strategie (PLRU) verwendet. Die PLRU-Strategie ist an die LRU-Strategie angelehnt welche besagt, dass immer die Cacheline verdrängt wird, die am längsten nicht mehr genutzt wurde. Details zu der Implementierung der PLRU-Strategie finden sich in Anhang B.2.
- **voll-assoziativ:** Einen *voll-assoziativen* Cache kann man auch als einen Set-assoziativen Cache mit $n \rightarrow s$ ansehen, wobei s die Cachegröße ist. Das heißt, dieser Cache besteht aus Cachelines, die sich ein Set in der Größe des gesamten Caches teilen. Es besteht kein direkter Zusammenhang zwischen Cacheline und Speicherstelle im Hauptspeicher, jeder Speicherbereich kann auf jede Cacheline abgebildet werden.

Prozessoren mit IA-32 Architektur nutzen Set-assoziative Caches (eine Ausnahme ist der Instruktionen-Cache des Pentium 4, siehe Anhang B.3). Im Folgenden werden die verschiedenen Ebenen der Caches miteinander verglichen.

L1 Cache

Die erste Cacheebene ist unterteilt in einen *Instruktionen*-Cache und in einen *Daten*-Cache, die jeweils gleich groß sind (Harvard Architektur). Die beiden Caches sind identisch organisiert (Assoziativität, Cacheline-Größe) und es kann parallel darauf zugegriffen werden. Typische Cachegrößen sind 32 oder 64 kB, die Cacheline-Größen sind meist 32 oder 64 Byte und der Cache ist meist 2- oder 4-fach Set-assoziativ.

L2 Cache

Die zweite Cacheebene ist ein *unified* Cache, das heißt, er kann für Code und Daten gleichermaßen genutzt werden (von Neumann Architektur). Es gibt bei den IA-32 Prozessoren zwei Varianten: Ein *inklusives* und ein *exklusives* Cachedesign. Unter *inklusivem* Cachedesign versteht man die Tatsache, dass der L2-Cache ein komplettes Abbild des aktuellen L1-Cache beinhaltet. Bei einem *exklusiven* Cachedesign ist das nicht der Fall. Das heißt auch, dass bei einem *inklusiven* L2-Cache nur die Größe des L2-Cache abzüglich der Größe des L1-Cache zur Verfügung steht. Typischerweise ist der L2-Cache wesentlich größer als der L1-Cache (Faktor 4 bis 8 oder 16). Die Cacheline-Größe entspricht der des L1-Cache. Die Assoziativität ist meist höher, in der Regel 8 oder 16.

Abbildung des Hauptspeichers in den Cache

Prozessoren der IA-32 Architektur nutzen physikalisch indizierte Caches, das heißt, Teile der physikalischen Adresse werden für die Set-Auswahl im Cache herangezogen. Es gibt auch virtuell indizierte Caches, die nutzen anstatt der physikalischen Adresse Teile der virtuellen Adresse zur Indizierung. Im Folgenden wird von einem n-fach Set-assoziativen Cache ausgegangen.

Bild 3.7 zeigt die Aufteilung des Adressworts für einen Cache mit 64 Byte Cacheline-Größe und 512 Sets (das entspricht bei einem 2-fach Set-assoziativen Cache einer Cachegröße von 64 kB, wie sie beispielsweise die Athlon Prozessoren verwenden).

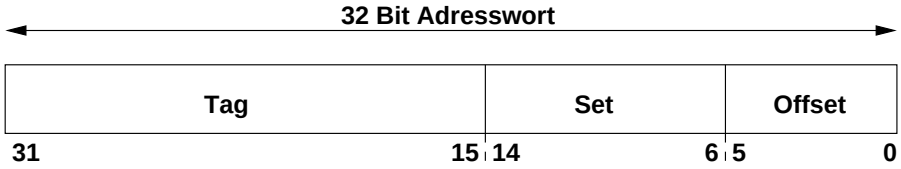

Bild 3.7: Aufteilung eines Adresswortes

Der niederwertigste Teil des Adressworts besteht aus m Bits, die für die Adressierung innerhalb einer Cacheline benötigt werden ($m = 6$ in Bild 3.7). Die folgenden Bits werden für die Auswahl eines Sets im Cache herangezogen, in Bild 3.7 sind dies 9 Bits für 512 Sets. Die restlichen Bits des Adresswortes werden als *Tag* verwendet, mit dessen Hilfe ein bestimmter Speicherbereich im Cache identifiziert werden kann.

Fortlaufende Adressen im Hauptspeicher werden demnach in aufeinander folgende Sets im Cache abgelegt. Ist das letzte Set im Cache erreicht, wird als nächstes wieder das erste Set verwendet. Bild 3.8 verdeutlicht diesen Zusammenhang anhand des Beispiels für einen 2-fach Set-assoziativen Cache mit 64 Byte Cacheline-Größe und 64 kB Gesamtgröße. Welche Cacheline innerhalb eines Sets verwendet wird, hängt von der Verdrängungsstrategie des Prozessors ab und ist für Software transparent.

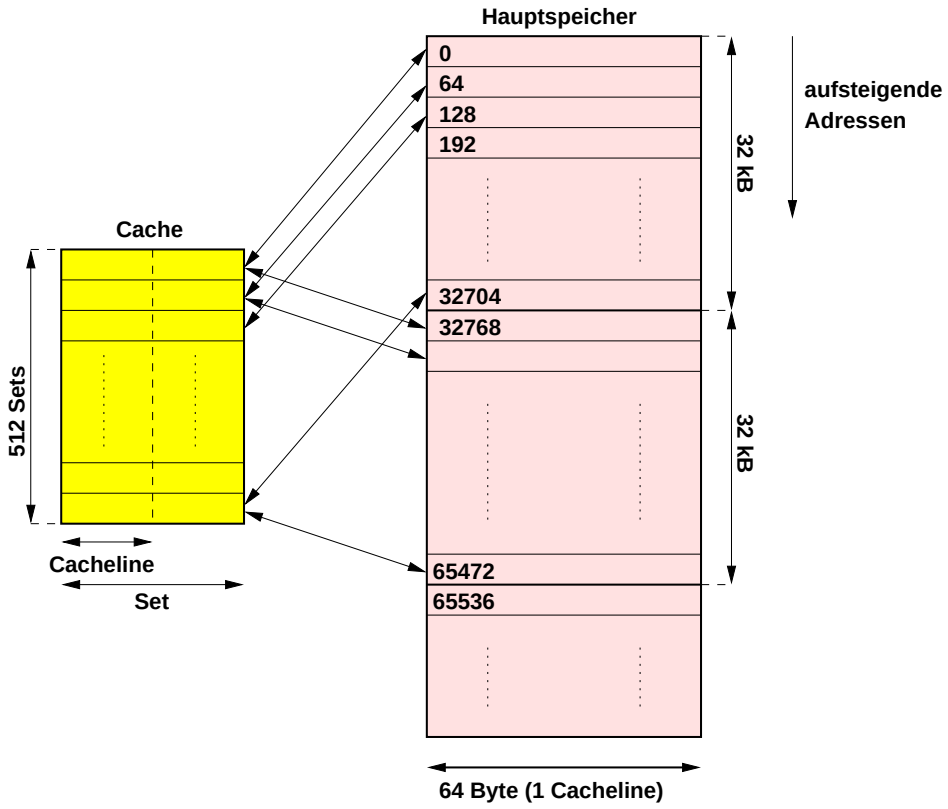

Bild 3.8: Abbildung von Hauptspeicheradressen in den Cache

Für das Beispiel aus Bild 3.8 bedeutet dies, dass in Abständen von 32 kB wieder dasselbe Set verwendet wird. Innerhalb eines solchen 32 kB großen Speicherbereichs können sich also keine Speicherinhalte gegenseitig verdrängen. Innerhalb eines 64 kB großen Bereichs *kann* dies passieren, abhängig davon, welche Cacheline aus dem jeweiligen Set tatsächlich verdrängt wird.

3.3.3 Caching-Strategien

Beim Zugriff auf eine Speicherzelle wird zunächst im L1-Cache nach dieser Speicherzelle gesucht. Ist sie dort nicht vorhanden, wird in der nächsten Ebene, dem L2-Cache, gesucht. Ist die gesuchte Speicherzelle nicht im Cache, wird sie vom Hauptspeicher in den L1-Cache geladen. Es werden immer ganze Cachelines geladen. Manche Caches implementieren ein sogenanntes *prefetching*, das heißt, sie laden noch eine oder mehrere nachfolgende Cachelines mit.

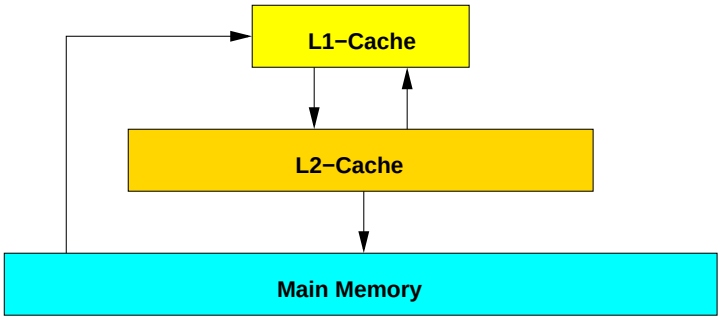

Bild 3.9: Zusammenspiel von Caches und Hauptspeicher

Bevor eine Cacheline in den L1-Cache geladen werden kann, muss zunächst eine andere Cacheline in den L2-Cache verdrängt werden. Um diese Cacheline aufnehmen zu können, muss auch aus dem L2-Cache eine Cacheline in den Hauptspeicher verdrängt werden. Bild 3.9 zeigt dieses Szenario.

Bei der Verdrängung in den Hauptspeicher muss man zwei Fälle unterscheiden:

- Ist der Inhalt der Cacheline unverändert geblieben, muss kein Schreibvorgang in den Hauptspeicher durchgeführt werden. Die Cacheline wird als *invalid* gekennzeichnet und kann einfach überschrieben werden.
- Wurde der Inhalt einer Cacheline im Cache verändert, aber noch nicht im Hauptspeicher, muss diese Cacheline in den Hauptspeicher zurückgeschrieben werden, um Datenkonsistenz zu gewährleisten. In einem Multiprozessorsystem müssen die Cacheinhalte aller Prozessoren konsistent gehalten werden. Dafür sorgt das *Cache-Snooping Protokoll*, das in Abschnitt 3.3.4 beschrieben wird.

Die Prozessoren mit IA-32 Architektur, die den Schwerpunkt dieser Arbeit bilden, unterstützen mehrere Caching-Strategien. Es kann beispielsweise für bestimmte Speicherbereiche festgelegt werden, ob Modifikationen an Cacheinhalten sofort in den Speicher übertragen werden *(write-through)* oder ob diese erst bei der Verdrängung der Cacheline zurückgeschrieben werden *(write-back)*. Details dazu befinden sich in Abschnitt 5.3.4.

3.3.4 Kohärenz der Caches in Multiprozessorsystemen

In einem Multiprozessorsystem teilen sich mehrere Prozessoren den Hauptspeicher. Insbesondere können mehrere Prozessoren dieselben Inhalte in ihren Caches speichern. Wenn nun ein Prozessor A den Inhalt einer Cacheline verändert, die auch ein anderer Prozessor B in seinem Cache hat, muss Prozessor B von der Änderung informiert werden. Die Modifikation der Cacheline muss auch im Cache von Prozessor B durchgeführt werden, bevor er mit dem Inhalt dieser Cacheline weiterarbeitet. Diese Aufgabe wird von dem *Cache-Snooping Protokoll* übernommen.

Das *Cache-Snooping Protokoll* ist bei den Prozessoren der IA-32 Architektur in Hardware realisiert und für Software transparent. Es sorgt automatisch dafür, dass die Caches aller Prozessoren konsistent bleiben. Wird eine Überschneidung von Speicherinhalten in den Caches festgestellt, wird der modifizierte Inhalt von Prozessor A in den Hauptspeicher zurückgeschrieben. Danach wird Prozessor B aufgefordert, den Inhalt der entsprechenden Cacheline neu zu lesen.

Die dadurch entstehenden Hauptspeicherzugriffe müssen für die Bestimmung von Ausführungszeiten für ein Realzeitsystem mit berücksichtigt werden. Diese Einflüsse können nicht durch eine Codeanalyse der Realzeitsoftware auf Prozessor A oder Prozessor B vorhergesagt werden. Ob sich zwei oder mehrere Prozessoren einen oder mehrere Speicherinhalte teilen, hängt von der Anordnung der Software im Hauptspeicher ab. Eine geeignete Speicherverwaltung für Realzeitsysteme wird in Kapitel 5 vorgestellt, auf die Realisierung des Cache-Snooping wird in Abschnitt 5.3.5 eingegangen.

3.4 Speichertechnologien

Die Taktraten von Prozessoren und Speichern haben sich in den vergangenen zehn Jahren stark auseinander entwickelt. Während die Taktraten der Prozessoren sehr stark angestiegen sind (von ca. 100 MHz bis aktuell ca. 3 GHz), sind die Speicher nur geringfügig schneller geworden. Dies ist der Grund dafür, dass der schnelle Cache-Speicher der Prozessoren immer mehr an Bedeutung gewonnen hat. Er wird dazu verwendet, den Geschwindigkeitsunterschied zwischen Prozessor und Speicher zu überbrücken. In diesem Abschnitt werden kurz die heutzutage verwendeten Speichertechnologien vorgestellt. Der verwendete Speicher hat neben anderen Faktoren einen bedeutenden Einfluss auf das Laufzeitverhalten des Systems.

Random Access Memory

Die Abkürzung RAM steht ganz allgemein für *Random Access Memory*. Damit wird ein Speicher bezeichnet, auf dessen Elemente in beliebiger Reihenfolge zugegriffen werden kann. In Bild 3.10 ist der schematische Aufbau eines solchen Speichermoduls gezeigt.

Die Speicherchips (*Dual Inline Memory Module, DIMM*) sind als eine Matrix mit Zeilen und Spalten aufgebaut. Jede Speicherstelle kann mit der entsprechenden Angabe von Zeile und Spalte angesprochen werden. Ein SDRAM-Baustein ist in der Regel aus mehreren solcher Matrizen (Bänke) aufgebaut. Zur Dekodierung der Zeileninformation steht ein *Row Decoder* zur Verfügung, der immer eine ganze Zeile auf einmal lädt. Über den *Column Decoder* wird eine Spalte ausgewählt. Die Signale zur Auswahl von Zeile und Spalte werden mit CAS (Column Address Strobe) und RAS (Row Address Strobe) bezeichnet. Die Zeit, die benötigt wird vom Anliegen des CAS-Signals bis die Spalteninformation tatsächlich vorliegt, wird als t_{CAS} bezeichnet. Diese Größe ist für die Geschwindigkeit eines Speicherbausteins von entscheidender Bedeutung. Eine weitere relevante Größe ist die Zeitspanne t_{RCD} (Row Column Delay), welche die Zeit angibt, die von einer gültigen Spaltenadressierung bis zum Anliegen eines gültigen RAS-Signals vergeht. Weiterhin wird meist noch die Zeit t_{RAS} angegeben, die analog zu t_{CAS}

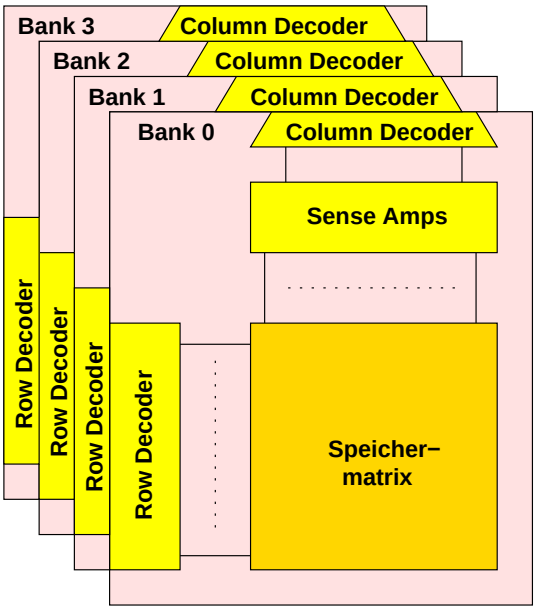

Bild 3.10: Schematischer Aufbau von RAM-Speicher

die Zeit bezeichnet, die vergeht, bis die angeforderte Zeileninformation vorliegt. Alle Zeiten werden in Vielfachen des Systemtakts angegeben.

Diese drei Zeiten werden oft zur Charakterisierung des Speicherchips in der Form $t_{CAS}-t_{RCD}-t_{RAS}$ angegeben. Ein Beispiel für einen schnellen SDRAM-Chip wäre die Bezeichnung 2-2-2 welche bedeutet, dass alle drei Zeiten zwei Taktzyklen betragen. Bei einem Takt von 133 MHz entspricht dies 7,5 ns. Ein weiterer wichtiger Parameter ist die Zeit t_{RP} (Row Precharge) welche die Zeit angibt, bis eine Zeile geladen ist. Dies ist von Bedeutung, wenn aufeinander folgende Speicherzugriffe in verschiedenen Zeilen des Speicherchips stattfinden. Dann muss zunächst die aktuelle Zeile geschlossen und die neue Zeile geladen werden.

Um diese Wartezeit t_{RP} möglichst zu vermeiden, ist ein Speicherchip in mehrere Bänke aufgeteilt (siehe Bild 3.10). Damit können mehrere Zeilen auf einmal geladen sein. Das heißt, sequentielle Speicherzugriffe werden nicht in einer Bank durchgeführt, sondern in verschiedenen Bänken. Wenn gerade eine Bank aktiv ist und am Speichercontroller, der die Zugriffe koordiniert, bereits weitere Anfragen anliegen, kann die nächste Zeile für die folgenden Zugriffe schon geladen werden. Das setzt voraus, dass die Daten über mehrere Bänke verteilt werden. Um dieses Verfahren zu optimieren, ist der Speichercontroller auch in der Lage, voneinander unabhängige Speicherzugriffe in einer anderen als der originalen Reihenfolge auszuführen. Damit können die vorhandenen Informationen optimal genutzt werden und während eines Transfers bereits die nächsten Zeilen geladen werden. Das bedeutet, dass die Zeit t_{RP} in diesen Fällen nicht als Verzögerungszeit gewertet werden muss.

Die Anzahl der Sense Amps, die nach einem Zugriff auf eine Spalte immer noch die Informationen gepuffert haben, wird auch als *Page Size* bezeichnet. Übliche Größen sind hier 2, 4 oder 8 kB. Diese Bezeichnung kommt daher, dass für Zugriffe innerhalb einer solchen Page keine

neuen Spalteninformationen geladen werden müssen, das heißt, diese Zugriffe können schneller abgearbeitet werden (Burst Transfers).

Weiterhin wird zwischen *registered* und *unbuffered* Speicherchips unterschieden. Beide Typen können nicht zusammen verwendet werden. Bei *unbuffered* Modulen kommuniziert der Speichercontroller direkt mit dem RAM. Dies ist die schnellste Methode des Speicherzugriffs. Der Nachteil ist, dass nur wenige Speicherchips hintereinander geschalten werden können, da die Signale auf dem Speicherbus getrieben werden müssen. Diesen Nachteil umgehen *registered* Module. Dort werden alle Signale vom Speichercontroller in einem Register zwischengespeichert. Dies hat eine niedrigere elektrische Kapazität der Module zur Folge, das heißt, es können mehr Speicherchips hintereinander geschalten werden. Nachteilig wirkt sich aus, dass die Zwischenspeicherung der Signale Zeit kostet und ein Speicherzugriff dadurch etwas langsamer ist als bei *unbuffered* Modulen.

Um die Zugriffsgeschwindigkeit auf den Speicher zu erhöhen, wird oft ein sogenanntes *Memory Interleaving* durchgeführt. Diese Technik kann genutzt werden, wenn in einem Rechner mehrere Speichermodule verwendet werden. Bei diesem Verfahren wird beispielsweise eine Cacheline des Prozessors (in aller Regel wird immer mindestens eine komplette Cacheline geladen) auf mehrere Speichermodule verteilt. Wird nun auf eine Cacheline zugegriffen, wird die Adresse gleichzeitig an mehrere Speichermodule gegeben. Die Latenzzeiten t_{CAS} und t_{RCD} treten dann nur einmal beim ersten Speichermodul auf, bei allen anderen Modulen, die den Rest der Cacheline beinhalten, sind die entsprechenden Zeilen und Spalten bereits geladen. Bei einem System mit zwei Speichermodulen spricht man von einem 2-way Interleaving, bei vier Modulen von einem 4-way Interleaving.

Um die Speicherzugriffszeiten anzugeben, wird oft eine Notation in der Form x-y-y-y angewandt. Der erste Parameter x bezeichnet die Latenzzeit für den ersten Speicherzugriff einer Cacheline, die y-Werte die Latenzzeiten für alle folgenden Speicherzugriffe, bis die Cacheline komplett ausgelesen oder geschrieben wurde. Diese Zeiten werden in Vielfachen von Taktzyklen des Front-Side Busses angegeben (im Unterschied zu den Kennzahlen des Speichermoduls, die in Vielfachen des Speichertakts angegeben werden).

DRAM, SRAM, SDRAM und DDR-Speicher

Grundsätzlich unterscheidet man zwischen statischem (SRAM) und dynamischen RAM (DRAM). Statisches RAM ist ein Speicher, dessen Informationen während des Betriebs nicht ständig neu geschrieben werden müssen (keine *refresh Zyklen*). Statisches RAM ist sehr schnell, hat einen vergleichsweise hohen Energieverbrauch und ist aufgrund der aufwändigen Realisierung sehr teuer. Es wird daher meist als Cache verwendet.

Im Gegensatz zu SRAM ist DRAM ein flüchtiger Speicher. Die Informationen müssen zyklisch neu geschrieben werden (*refresh Zyklen*). Der Energieverbrauch ist im Vergleich zu statischem RAM, ebenso wie die Kosten, relativ niedrig. Daher wird DRAM für größere Speichermodule verwendet und kommt heutzutage als Hauptspeicher zum Einsatz.

Die Bezeichnung SDRAM steht für *Synchronous Dynamic Random Access Memory*. Damit wird eine Speichertechnologie bezeichnet, bei der sich alle Steuersignale an einem konstanten Takt ausrichten (daher die Bezeichnung *synchron*). Pro Taktzyklus werden einmal Daten übertragen (steigende Flanke). Im Gegensatz dazu überträgt DDR-RAM (Double Date Rate) pro Taktzyklus zweimal Daten (steigende und fallende Taktflanke). Tabelle 3.2 gibt einen Überblick über SDRAM und DDR-RAM Speicherchips und ihre technischen Daten.

Speichertyp	Bustakt (MHz)	Max. Datenrate (MB/s)
PC66-SDRAM	66	533
PC100-SDRAM	100	800
PC133-SDRAM	133	1066
PC1600 (DDR1)	100	1600
PC2100 (DDR1)	133	2100
PC2700 (DDR1)	166	2700
PC3200 (DDR2)	200	3200
PC4300 (DDR2)	266	4300
PC5400 (DDR2)	333	5300

Tabelle 3.2: Verschiedene Speichertypen und ihre Transferraten

Die angegebenen Datentransferraten entsprechen dem Maximum, das bei optimalen Zugriffsmustern auf den Speicher zu erreichen ist. In der Praxis wird man, abhängig von der jeweiligen Software, niedrigere Datenraten erreichen. Der in Tabelle 3.2 angegebene Speichertyp DDR2 bezeichnet eine neue Realisierungsvariante des herkömmlichen DDR1-Speichers. Damit sind höhere Taktfrequenzen als bei DDR1 möglich und zugleich sinkt der Energieverbrauch.

4 Ausführungszeiten von Software auf modernen Prozessoren

In diesem Kapitel wird zunächst ein Ansatz zur Messung von Ausführungszeiten vorgestellt. Diese Messmethodik wurde zur Ermittlung aller Laufzeiten in dieser Arbeit verwendet und basiert auf dem Ansatz, den Petters in [56] vorgestellt hat. Im Anschluss werden die Mechanismen moderner Prozessorarchitekturen auf ihren Einfluss auf Ausführungszeiten von Software untersucht. Im dritten Abschnitt werden SMP- und NUMA-Systeme im Hinblick auf die Entstehung von Ausführungszeiten, speziell bei parallelen Speicherzugriffen, miteinander verglichen.

4.1 Messung von Ausführungszeiten

Grundsätzlich gibt es zwei Ansätze, um Ausführungszeiten auf einem Rechner zu ermitteln:

- Man bildet ein Modell der verwendeten Hardware und ermittelt die Ausführungszeiten durch Simulation. Hierzu ist es nötig, genaue Kenntnisse über den Aufbau der einzelnen Hardwarekomponenten (z.B. Prozessorarchitektur, Speicher) zu haben. Je komplexer der Aufbau der Hardware ist, desto umfangreicher, schwieriger und auch fehleranfälliger ist eine Simulation. Um eine WCET aus dem Modell simulativ bestimmen zu können, muss sichergestellt sein, dass das Modell hinreichend genau ist.
- Die Alternative zur Modellierung ist die Messung von Ausführungszeiten direkt auf der Zielhardware. Dazu ist eine Messtechnik nötig, mit der man die Laufzeiten hinreichend genau messen kann. Optimal wäre eine Messung auf einen Prozessorzyklus genau. Um die WCET eines beliebigen Stück Codes messen zu können, muss die Worst-Case Situation, die diese WCET hervorruft, vor der Messung eingestellt werden können. Je komplexer der Code und die Zielhardware sind, desto schwieriger ist es, den Worst-Case zu identifizieren und künstlich für eine Messung zu erzeugen.

Für den Einsatz auf PC-Architekturen kommt nur der messtechnische Ansatz in Frage. Zum einen sind PC-Architekturen sehr komplex, angefangen bei den verwendeten Prozessoren über die Bussysteme bis hin zu den Speichertechnologien, so dass eine Modellierung sehr aufwändig und fehleranfällig wäre. Zum anderen ist die genaue Funktionsweise der Hardware oft nicht dokumentiert, so dass eine hinreichend genaue Modellierung des Gesamtsystems nicht möglich ist.

Auf der anderen Seite verfügen alle IA-32 kompatiblen Prozessoren oder auch der PowerPC über eine Möglichkeit, Laufzeiten zyklengenau bestimmen zu können. Mit Hilfe der Messun-

gen können die Auswirkungen einzelner Mechanismen auf die Laufzeiten von Software und deren Wirkungen untereinander untersucht werden. Somit wird es möglich, auch hoch komplexe Systeme mit vergleichsweise geringem Aufwand auf ihr Laufzeitverhalten hin zu untersuchen. Im folgenden Abschnitt wird das Prinzip der Messung von Ausführungszeiten vorgestellt.

4.1.1 Prinzip der Messung

Alle moderneren PC-Prozessoren verfügen über ein Register namens *Time-Stamp Counter* (TSC). Dieses Register wird mit jedem Taktzyklus des Prozessors um eins erhöht. Es ist für Software jederzeit möglich, den Wert dieses Registers auszulesen. Um die Laufzeit eines bestimmten Codestücks zu messen, wird vor der Ausführung des Codestücks und unmittelbar nach dessen Ausführung der Wert des TSC-Registers gelesen. Die Differenz der beiden Werte ist die Ausführungszeit des Codes in Prozessortaktzyklen.

Moderne Prozessoren verfügen über die Möglichkeit, die Befehle eine Codestücks in einer anderen Reihenfolge auszuführen, als sie im Programmcode angegeben ist (*out-of-order execution*, siehe Abschnitt 4.2.1 und 4.2.2). Das kann dazu führen, dass die Operation, die den Wert des TSC ermittelt, vor der letzten Instruktion des Codes ausgeführt wird, dessen Laufzeit ermittelt werden soll. In diesem Fall wäre die gemessene Zeit zu klein. Um dies zu verhindern, muss man vor dem Lesen des TSC-Registers die Befehlsabarbeitung *serialisieren*. Das bedeutet, dass die Befehlsabarbeitung aller vorigen Instruktionen beendet wird, bevor der aktuelle, serialisierende Befehl ausgeführt wird. Der Overhead, der durch diese Serialisierung entsteht, ist sehr gering (siehe auch Abschnitt 4.2.1). In Bild 4.1 ist der detaillierte Aufbau der Messroutine gezeigt. Der Zeitstempel in Schritt 2 beendet eine Messung, der Zeitstempel in Schritt 13 startet eine neue Messung.

Die Messroutine beginnt mit der Serialisierung der Befehlsabarbeitung. Nach dem Lesen des TSC in Schritt 2 werden in Schritt 3 die Werte der *Performance-Monitoring Counter* (PMCs) gelesen. Diese Register können so programmiert werden, dass sie bestimmte Ereignisse während der Codeausführung mitzählen. Beispielsweise kann die Anzahl der Cache Misses im L1- oder L2-Cache gezählt werden oder auch die Anzahl der TLB Misses. Eine ausführliche Beschreibung zu den PMCs befindet sich in [3] und [39]. Mit Hilfe der PMCs kann zum einen die Korrektheit der gemessenen Ergebnisse verifiziert werden und zum anderen das Zusammenwirken verschiedener Mechanismen bei der Codeausführung untersucht werden.

In Schritt 4 werden der Zeitstempel und die Werte der PMC-Register im Hauptspeicher abgelegt. Um durch diese Speicherzugriffe nicht den Cache des Prozessors zu beeinträchtigen, ist dieser Speicherbereich als *uncacheable* markiert. Die Zugriffe gehen demnach am Cache vorbei direkt in den Hauptspeicher. Die folgende Abfrage entscheidet anhand eines Übergabeparameters, ob für die folgende Messung der Worst-Case bezüglich der Prozessorarchitektur hergestellt werden soll oder nicht.

Ist dies erwünscht, wird entweder der komplette L1 Instruktionen-Cache (I-Cache) oder der komplette L1 Daten-Cache (D-Cache) oder beide invalidiert. Dadurch können die Auswirkungen von Cache Misses auf Code und Daten getrennt untersucht werden. Bei der Invalidierung

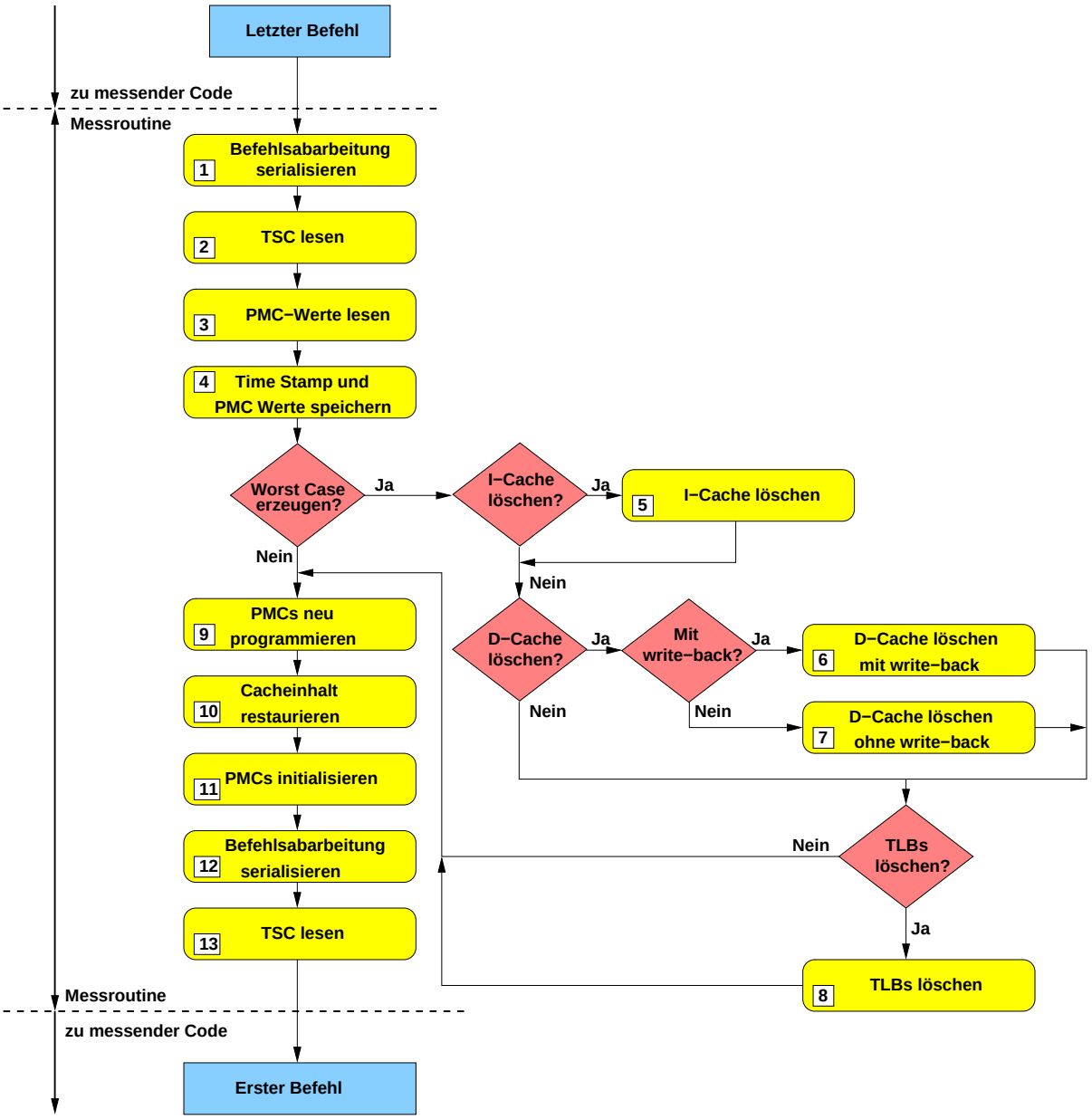

Bild 4.1: Die Messroutine

des Daten-Caches wird noch unterschieden zwischen einer Invalidierung mit Daten, die hinterher zurückgeschrieben werden müssen (Schritt 6) oder solchen, die nicht zurückgeschrieben werden müssen (Schritt 7).

Anschließend können die *Translation Lookaside Buffers* (TLBs) in Schritt 8 invalidiert werden. Die TLBs sind kleine Caches, die die notwendige Information für die Umrechnung einer virtuellen in eine physikalische Adresse beinhalten. Bei jedem Speicherzugriff wird auch auf die TLBs zugegriffen. Details zu den TLBs befinden sich in Anhang B.4.

In Schritt 9 werden die PMCs neu programmiert, um das Auftreten bestimmter Ereignisse während der anschließenden Codeausführung mitzählen zu können. In Schritt 10 werden Variablen in den Daten-Cache geladen, die von der Messroutine im restlichen Verlauf noch benötigt werden und zuvor gegebenenfalls in Schritt 6 oder Schritt 7 aus dem Cache verdrängt wurden. Diese Maßnahme dient dazu, dass die Messroutine selbst nicht schon Daten-Cache Misses verursacht, die dann dem zu messenden Code zugeordnet werden.

Anschließend werden in Schritt 11 die PMC-Register auf den Wert null gesetzt. Ab diesem Zeitpunkt werden die zuvor angegebenen Ereignisse mitgezählt. Um nicht die Ausführung von Code der Messroutine mitzumessen, wird die Befehlsabarbeitung nochmals serialisiert, bevor die neue Messung mit dem Lesen des Zeitstempels in Schritt 13 gestartet wird. Da auch dieser Wert abgespeichert werden muss, sind zwei Speicherzugriffe an dieser Stelle nötig, die sich aber nicht vermeiden lassen. Während der Ausführung der Messroutine sind die Interrupts gesperrt, was deren Implementierung vereinfacht und damit auch den Overhead für die Messung reduziert.

Bei der Implementierung der Messroutine muss man auch das automatische *Prefetching* des Prozessors beachten. Muss eine Cacheline in den Instruktionen-Cache geladen werden, so werden automatisch mindestens zwei Cachelines geladen: Jene, die benötigt wird und der Speicherbereich, der im Hauptspeicher direkt anschließt. Man muss jetzt darauf achten, dass durch die Messroutine kein Prefetch ausgelöst wird, der den nachfolgenden Code in seiner Ausführungszeit begünstigt. Dazu wird der Code, der den Zeitstempel in Schritt 13 nimmt, am Anfang einer neuen Cacheline angeordnet (*Alignment*). Dadurch ist sichergestellt, dass diese Cacheline und der physikalisch folgende Speicher in den Cache geladen werden. Der Rest der ersten Cacheline, der nicht von Code benötigt wird, und die folgende Cacheline werden mit *nop*-Operationen (*nop*: no operation) aufgefüllt. Dadurch wird sichergestellt, dass das automatische Prefetching keinen Vorteil für den nachfolgenden Code bringt.

Je nach Lage der Messroutine im Speicher benötigt die Messroutine keinen neuen Eintrag in den TLBs oder höchstens einen für den Code und einen für die Daten. Durch eine geeignete Speicherbelegung, wie sie in Kapitel 5 diskutiert wird, kann genau bestimmt werden, wieviele TLB Einträge für eine Messung nötig sind.

Ablauf einer Messung

Bild 4.2 zeigt den Ablauf einer Messung. Vor dem Codestück, dessen Laufzeit ermittelt werden soll, und nach diesem Codestück wird jeweils einmal die Messroutine aus Bild 4.1 aufgerufen. Der Zeitstempel `TSC2` des ersten Aufrufs der Messroutine startet die Messung, `TSC1` des zweiten Aufrufs beendet die Messung.

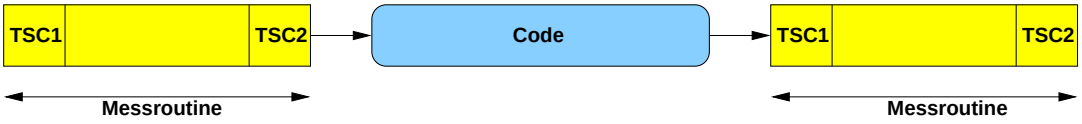

Bild 4.2: Der Ablauf einer Messung

Auf diese Art und Weise können beliebig viele Messpunkte hintereinander gesetzt werden. Da jeder Messpunkt eine eindeutige ID-Nummer bekommt, können die Ergebnisse einer Messung hinterher eindeutig den verschiedenen Codestücken zugeordnet werden.

Es ist auch möglich, parallel auf mehreren Prozessoren zu messen. Darauf wird ausführlich im folgenden Abschnitt eingegangen.

4.1.2 Messen auf Multiprozessorsystemen

Um auf mehreren Prozessoren parallel Messungen durchführen zu können, muss es möglich sein, die Messroutine gleichzeitig mehrmals aufzurufen. Um die Messergebnisse nach der Messung einem bestimmten Prozessor zuordnen zu können, muss die Messroutine in der Lage sein festzustellen, von welchem Prozessor sie gerade aufgerufen wurde.

Zur Synchronisierung der Aufrufe setzt die Messroutine zwischen Schritt 2 und Schritt 3 (siehe Bild 4.1) einen *Spinlock*, der direkt vor Schritt 12 wieder freigegeben wird. Ein *Spinlock* ist ein Mechanismus, der es ermöglicht, dass bestimmte Codeteile zur selben Zeit nur von einem Prozessor ausgeführt werden können.

Dazu wird bei Eintritt in den *Spinlock* zunächst überprüft, ob dieser frei ist. Dazu wird der Wert einer Variablen *atomar* dahingehend überprüft, ob er größer oder gleich null ist. Ist dies der Fall, wird diese Variable *atomar* von einem Prozessor dekrementiert und der anschließende Code ausgeführt. Möchte ein anderer Prozessor denselben Code ausführen, stellt er bei Eintritt in den *Spinlock* fest, dass dieser bereits von einem anderen Prozessor belegt ist und wartet nun solange an diesem *Spinlock*, bis er wieder freigegeben wird.

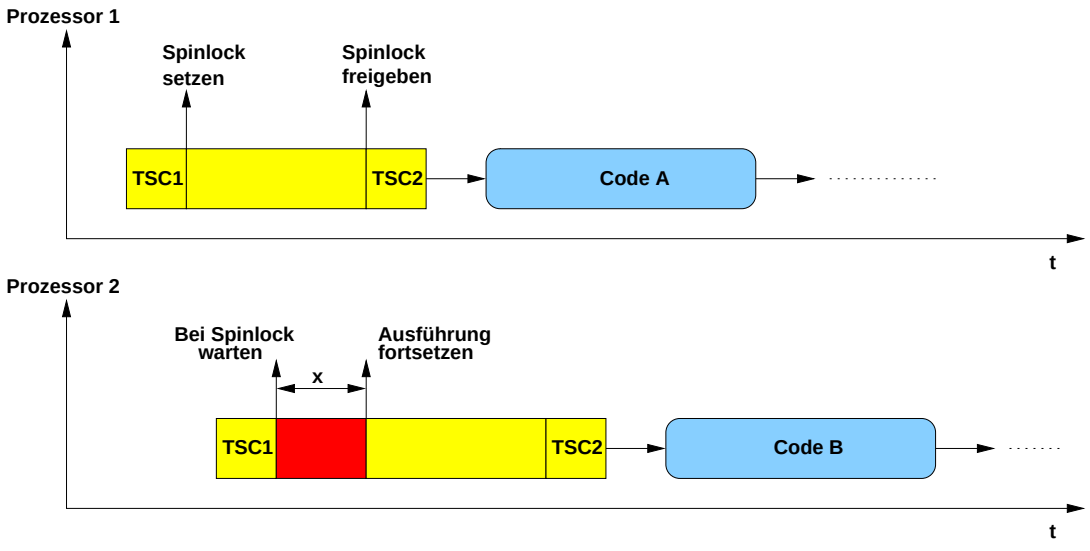

Bild 4.3: Synchronisierung bei der Messung auf mehreren Prozessoren

Bild 4.3 zeigt die Synchronisierung beim Ablauf paralleler Messungen. Die Zeitspanne x bezeichnet die Blockierzeit, die Prozessor 2 warten muss, bis er die Codeausführung fortsetzen

kann. Die Spinlock-Operationen tragen *nicht* zum Overhead der Messroutine bei. Die Blockierzeit selbst wird *nicht* mitgemessen, da sie erst nach dem Lesen des TSC auftritt.

Anhand der *APIC-ID* (Advanced Programmable Interrupt Controller) eines jeden Prozessors kann die Messroutine feststellen, auf welchem Prozessor sie gerade läuft. Die Intel-Spezifikation für Multiprozessorsysteme schreibt vor, dass die APIC-ID für jeden Prozessor eines Rechnersystems eindeutig sein muss. Diese ID lässt sich leicht durch Lesen eines bestimmten Registers ermitteln. Sie wird zusammen mit den Zeitstempeln und den PMC-Werten abgespeichert.

4.1.3 Auswertung und Messgenauigkeit

Für die Messroutine wird ein bestimmter Speicherbereich reserviert, in dem die Messdaten abgelegt werden. Ist dieser Speicher voll, muss er ausgelesen werden. Die Daten können dann weiterverarbeitet und beispielsweise auf Festplatte abgespeichert werden.

Das Auslesen der Messdaten übernimmt eine Steuerapplikation, die nicht unter Realzeitbedingungen laufen muss (siehe auch Abschnitt 5.3.2). Die Messroutine erkennt selbstständig, wenn der Speicherbereich voll ist und meldet dies der Steuerapplikation. Da die Realzeitsoftware weiterlaufen soll, speichert die Messroutine in dieser Zeit keine Messwerte ab, sondern springt sofort nach dem Funktionsaufruf zum Aufrufer zurück. Die Steuerapplikation kann nun die Messdaten auslesen und zur späteren Weiterverarbeitung abspeichern. Bild 4.4 zeigt den schematischen Messaufbau.

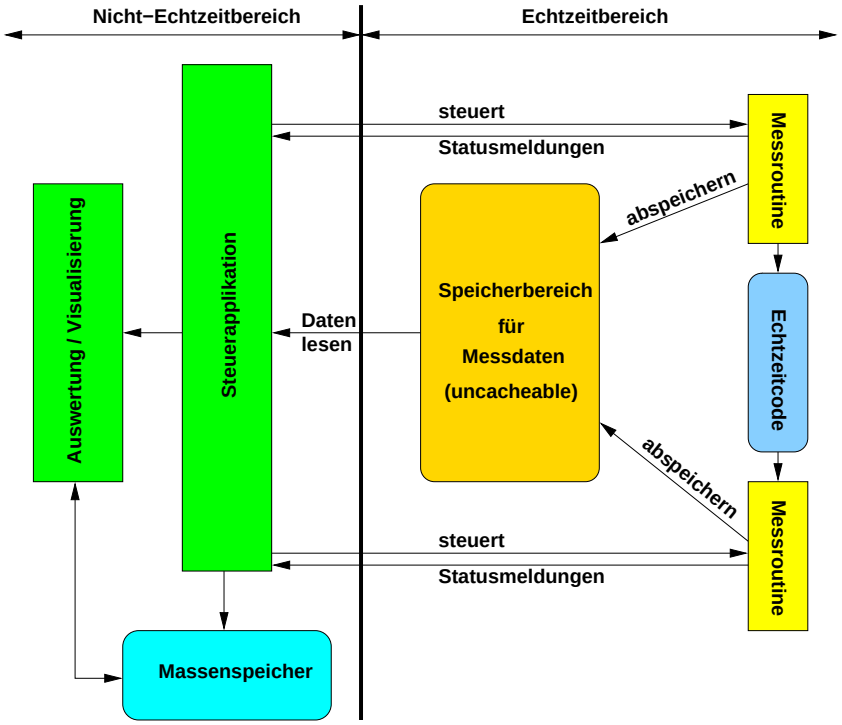

Bild 4.4: Softwareumgebung für die Messung

Sind die Daten vollständig ausgelesen, meldet das die Steuerapplikation der Messroutine und es können wieder Messdaten aufgenommen werden. Die Realzeitapplikation wird von diesen Vorgängen nicht beeinträchtigt.

Messgenauigkeit

Die Zeiten werden jeweils als Differenz zweier Zeitstempel berechnet. Diese Differenz gibt die Zeit in Prozessortaktzyklen an. Das TSC-Register ist 64 Bit breit und wird beim Start des Systems von null beginnend hochgezählt. Bis zu einem Überlauf des Registers würde es demnach $2^{64} - 1$ Taktzyklen dauern. Bei einer Taktfrequenz von 1,5 GHz wären das knapp 390 Jahre. Vorausgesetzt, dass die Taktfrequenz des Prozessors über den jeweiligen Messzeitraum stabil ist, ist diese Methode der Zeitmessung sehr genau und zuverlässig.

Ungenauigkeiten gibt es zum einen durch die Eigenschaften der Prozessorarchitektur, und zum anderen durch die Messroutine selbst. Die out-of-order execution der Prozessoren macht es erforderlich, serialisierende Befehle vor dem Lesen des TSC-Registers zu verwenden. Dadurch wird die Befehlsabarbeitung leicht verlängert. Die Laufzeitverlängerung ist jedoch nur sehr klein (siehe auch 4.2.1) und kann die Ausführung des Codes nie verkürzen. Das heißt für Messungen im Worst-Case, dass die gemessenen Zeiten nie kleiner als die tatsächliche Ausführungszeit sind.

Neben der Serialisierung der Befehlsabarbeitung entsteht ein Overhead für das Abspeichern des zweiten Zeitstempels, der die neue Messung startet. Dazu kommt noch die Übergabe von zwei Parametern (Steuervariable, Mess-ID) an die Messroutine und die Zeit für den Sprung zur Messroutine und den Rücksprung von der Messroutine.

Die Größe des Overheads hängt davon ab, unter welchen Voraussetzungen gemessen wird. Bei der Variante *NO_CACHE* werden nur die Zeitstempel genommen und die PMCs programmiert. Die Variante *CACHE* invalidiert alle Caches inklusive der TLBs. Bei den Varianten *ICACHE* und *DCACHE* werden jeweils nur der Instruktionen- bzw. der Daten-Cache invalidiert. *DCACHE_WB* invalidiert den Daten-Cache mit Daten, die hinterher zurückgeschrieben werden müssen (im Gegensatz zu *DCACHE*). Tabelle 4.1 stellt den Overhead der Messroutine (in Prozessortaktzyklen und Nanosekunden) auf einem AMD-Athlon (32 Bit) und einem AMD-Opteron (64 Bit) Prozessor gegenüber. Von den gemessenen Zeiten muss der jeweilige

Konfiguration	AMD-Athlon (1533 MHz)	AMD-Opteron (1793 MHz)
NO_CACHE	116 (75.67 ns)	112 (62.47 ns)
CACHE	1132 (738.42 ns)	1594 (889.01 ns)
ICACHE	1092 (712.33 ns)	1562 (871.17 ns)
DCACHE	116 (75.67 ns)	112 (62.47 ns)
DCACHE_WB	116 (75.67 ns)	112 (62.47 ns)

Tabelle 4.1: Overhead der Messroutine

Overhead abgezogen werden. Die Ergebnisse aus Tabelle 4.1 zeigen, dass der Overhead für die Messungen, die den Instruktionen-Cache invalidieren, relativ hoch ist im Vergleich zu den

Messarten, die den Instruktionen-Cache nicht berühren. Die Messwerte zeigen, dass die Messroutine selbst den Daten-Cache *nicht* beeinflusst.

Damit die Messergebnisse verglichen werden können, muss sichergestellt sein, dass die Messungen immer unter denselben Rahmenbedingungen ausgeführt werden. Dazu gehört insbesondere auch die Lage des Codes (Messroutine und zu messender Code) im Hauptspeicher, da diese mit der Position im Cache korrespondiert. Auf die Positionierung von Code und Daten im Hauptspeicher wird ausführlich in Kapitel 5 eingegangen.

4.1.4 Eingrenzen der WCET

Die WCET von Software hängt von vielen Faktoren ab: Der Prozessorarchitektur, der Anordnung im Hauptspeicher, der physikalischen Anbindung des Speichers und von parallel laufenden Transaktionen anderer Geräte, die den Speicher oder auch den Prozessor selbst (*Interrupt*) benötigen. Der Schwerpunkt dieser Arbeit liegt auf dem Einfluss der Prozessorarchitektur (Abschnitt 4.2) und Unterbrechungen durch *Interrupts* auf die WCET. Die Einflüsse durch die Aktivität von Peripheriegeräten (z. B. Massenspeicher) werden ausführlich in [73] behandelt.

Möchte man die WCET von einem Stück Code mit Hilfe von Messungen ermitteln, muss man vor der Messung das Worst-Case Szenario für dieses Codestück konstruieren. Da das für komplexe Systeme sehr aufwändig und schwierig zu realisieren ist, muss man die einzelnen Einflussfaktoren identifizieren, um dann deren Auswirkungen auf die WCET getrennt zu untersuchen und zu quantisieren. Kennt man dann noch die Wechselwirkungen einzelner Faktoren untereinander, kann man für eine konkrete Worst-Case Situation eine WCET rechnerisch ermitteln.

Die Eigenschaften der Prozessorarchitektur müssen von der Messroutine selbst berücksichtigt werden. Sie sind unabhängig von der Peripherie. Der nächste Schritt ist die Analyse der Speicherzugriffszeiten, die von der physikalischen Anbindung, der Art des Speichers und den parallelen Aktivitäten anderer Teilnehmer abhängen (Abschnitt 4.3).

Eine Messung kann immer nur die Ausführungszeiten ermitteln, die unter den zum Messzeitpunkt gegebenen Bedingungen entstehen. Sind diese Bedingungen über den Messzeitraum hinweg nicht konstant, liefern wiederholte Messungen unterschiedliche Ergebnisse. Das heißt, wenn man dieselbe Messung mehrmals wiederholt, ist die Streuung der Messwerte ein Maß für die Zuverlässigkeit der Messung. Erhält man bei wiederholten Messungen immer unterschiedliche Werte, kann man nicht sicher sagen, dass der größte gemessene Wert auch tatsächlich der WCET entspricht. Auch ein sehr langer Messzeitraum ändert daran nichts.

Daher ist es notwendig, immer nur so wenig Einflussfaktoren wie möglich mit Hilfe von Messungen zu untersuchen und zu versuchen, deren Einfluss auf die WCET mathematisch darzustellen. Gelingt dies für alle Faktoren, kann die WCET für ein bestimmtes Worst-Case Szenario berechnet werden. Eine Messung kann die Größenordnung der berechneten WCET dann verifizieren. Die WCET kann direkt nur dann gemessen werden, wenn *alle* Faktoren, die an der Entstehung der WCET beteiligt sind, über den Messzeitraum hinweg konstant im Worst-Case gehalten werden können.

4.2 Einfluss der Prozessorarchitektur auf die WCET

In diesem Abschnitt werden die verschiedenen Mechanismen moderner Prozessoren, wie sie beispielsweise in PC-Systemen verwendet werden, auf ihren Einfluss auf die Ausführungszeit von Software untersucht. Zunächst wird die Befehlsausführung moderner Prozessoren mit Hilfe von *Pipelines* näher untersucht und der *Tomasulo-Algorithmus* vorgestellt, der die parallele Ausführung mehrerer Befehle in einer anderen Reihenfolge als der vom Programmcode vorgegebenen ermöglicht. Darauf aufbauend wird der Einfluss der *Branch Prediction* näher untersucht. Dann folgt eine genaue Betrachtung der Einflüsse der Caches und TLBs und deren Kohärenz bei Multiprozessorsystemen. Am Ende des Abschnitts werden die Vorgänge bei einer Programmunterbrechung (Interrupt) näher beleuchtet.

4.2.1 Befehlsausführung mit Hilfe von Pipelines

Unter einer *Pipeline* versteht man die aufeinanderfolgenden Schritte bei der Ausführung eines Befehls. Pro Taktzyklus kann eine Stufe einer Pipeline abgearbeitet werden. Es gibt in der Regel unterschiedliche Pipelines für Integer- und für Floating Point Befehle. Die Befehlsausführung mit Hilfe von Pipelines dient dazu, möglichst viele Befehle parallel ausführen zu können. Das wird dadurch erreicht, dass die Abarbeitung in mehrere Schritte untergliedert wird (*Pipeline Stages*) und sich diese Schritte jeweils zeitlich überlappen. So kann ein Befehl beispielsweise gerade in Schritt fünf seiner Ausführung sein und ein nachfolgender Befehl in Schritt vier. Das Verhältnis von Stufen der Pipeline zu aktuell genutzten Stufen wird als *Füllgrad* der Pipeline bezeichnet. Eine Pipeline ist dann optimal ausgelastet, wenn alle Stufen gerade von Befehlen genutzt werden (Füllgrad = 1). In Anhang B.1 wird als Beispiel die Integer Pipeline des AMD-Athlon Prozessors vorgestellt.

Eine Instruktion kann auch mehrere Taktzyklen in einer Stufe der Pipeline verbringen, wenn die nötigen Daten zur weiteren Ausführung noch nicht vorhanden sind. Dadurch entsteht ein „Stau" in der Pipeline (*pipeline stall*), das heißt, die folgenden Instruktionen müssen ebenfalls warten.

Um nicht die gesamte Befehlsabarbeitung dadurch zu blockieren, implementieren moderne Prozessoren mehrere, gleichartige Pipelines, die parallel zueinander arbeiten. Um dieses Potential vollständig ausnutzen zu können, werden Befehle bearbeitet, sobald alle Operanden vorhanden sind, ungeachtet der ursprünglichen Reihenfolge der Befehle (*out-of-order execution*).

Um die korrekte Funktionsweise der Software zu garantieren, werden die Ergebnisse der Befehle solange in einem Puffer zwischengespeichert, bis die blockierende Operation auch ausgeführt wurde. Erst dann wird die Bearbeitung der einzelnen Befehle in der ursprünglichen Reihenfolge abgeschlossen (*in-order retirement*).

Möchte man für einen Prozessor mit Pipelines Ausführungszeiten bestimmen, hat man zunächst das Problem, dass nicht jeder Befehl alle Stufen der Pipeline durchläuft. Man muss für jeden Befehl wissen, welche Stufen er durchläuft. Hinzu kommt, dass man nicht davon ausgehen

kann, dass ein Befehl, der beispielsweise elf Stufen benötigt, auch wirklich nach elf Taktzyklen fertig abgearbeitet ist.

Für die Bestimmung von Ausführungszeiten von Software bedeutet dies, dass man für jeden Befehl wissen muss, wieviele Taktzyklen (Pipelinestufen) er zur ungestörten Ausführung benötigt. Das wird im Allgemeinen von den Herstellern dokumentiert. Hinzu kommen dann Verzögerungszeiten durch Pipeline Stalls, die von den Abhängigkeiten der Befehle untereinander verursacht werden. Man muss vier Arten von Ursachen für Pipeline Stalls unterscheiden:

1. **Ressourcen bedingte Stalls (Structural Stalls):** Wenn die *Reservation Stations* (siehe 4.2.2) des Prozessors keine freien Plätze mehr haben, können keine weiteren Befehle bearbeitet werden. Gleiches gilt für andere Ressourcen des Prozessors. Diese Verzögerungen sind nur von der Codestruktur abhängig und können sicher durch Messungen bestimmt werden.

2. **Datenabhängigkeiten (Data Hazard Stalls):** Stalls können durch Datenabhängigkeiten von Befehlen verursacht werden, das heißt, ein Befehl berechnet ein Ergebnis (z. B. eine Adresse), die ein nachfolgender Befehl benötigt. Diese Stalls hängen nur von der Programmstruktur selbst ab und sind für ein bestimmtes Stück Code immer gleich. Solche Verzögerungszeiten können sicher gemessen werden.

3. **Hauptspeicherzugriffe (Data Hazard Stalls with Memory):** Diese Stalls entstehen durch Zugriffe auf den Hauptspeicher, wenn die benötigten Daten nicht im Cache vorrätig sind. Die Wartezeiten dafür können stark schwanken und sind von mehreren Faktoren abhängig. Verzögerungszeiten dieser Art können nicht einfach mitgemessen werden.

4. **Programmverzweigungen (Control Stalls):** Stalls entstehen auch, wenn eine bedingte Sprunganweisung im Programmcode auftritt. Nach der Dekodierung des Sprungbefehls ist bis zur Berechnung der Sprungbedingung nicht bekannt, an welcher Stelle das Programm fortgesetzt werden soll. Diese Wartezeit kann etliche Taktzyklen betragen. Um dieses Problem zu lösen, implementieren moderne Prozessoren eine Sprungvorhersage (*Branch Prediction*), um schon bei der Dekodierung des Sprungbefehls sagen zu können, welcher Befehl als Nächstes ausgeführt werden soll. Die Auswirkungen der Branch Prediction auf Ausführungszeiten von Software werden in Abschnitt 4.2.3 untersucht.

Die Verzögerungszeiten, die durch Datenabhängigkeiten (Punkt 2) oder ressourcenbedingte Stalls (Punkt 1) entstehen, liegen in der Größenordnung weniger Taktzyklen. Je länger die Pipeline des Prozessors wird, desto größer können auch die Verzögerungszeiten werden. Sie sind abhängig von der Codestruktur des Programms und damit für ein bestimmtes Codestück konstant. Diese Zeiten können mit einer Laufzeitmessung der Software mitgemessen werden.

Auf die Auswirkungen von Speicherzugriffen (Punkt 3) wird in Abschnitt 4.3 eingegangen, die Auswirkungen der Branch Prediction (Punkt 4) sind Gegenstand von Abschnitt 4.2.3. Der folgende Abschnitt stellt den *Tomasulo-Algorithmus* vor, auf dem die *out-of-order* Befehlsausführung aller modernen Prozessoren basiert.

4.2.2 Tomasulo-Algorithmus

Der *Tomasulo-Algorithmus* ist benannt nach seinem Entwickler Robert M. Tomasulo, der diesen Algorithmus 1967 im IBM Journal - Research and Development [79]- veröffentlicht hat. Er ist die Grundlage für parallel arbeitende Pipelines, um die vorhandenen Ressourcen des Prozessors optimal ausnutzen zu können. Er ermöglicht die Ausführung der Befehle in einer anderen Reihenfolge als der vom Programmcode vorgegebenen. Die folgenden Abschnitte befassen sich mit den Methoden, die dieser Algorithmus anwendet. Es sollen hier nicht alle Details wiedergegeben werden. Es werden die zu Grunde liegende Prinzipien und Techniken, die auch für die Entstehung von Ausführungszeiten wichtig sind, näher betrachtet.

Register Renaming

Unter *Register Renaming* versteht man die Abbildung von extern sichtbaren Registern auf interne Register. Die Prozessoren haben in der Regel deutlich mehr interne als externe Register, die für den Programmierer sichtbar sind (z.B. 48 interne und 8 externe (reguläre) Register beim AMD-Athlon). Durch diese Abbildung können Register-Abhängigkeiten zwischen einzelnen Instruktionen aufgelöst werden, um diese dann parallel ausführen zu können. Das folgende kleine Beispiel illustriert dieses Vorgehen:

```
div eax,ebx
add edx,eax
sub ecx,ebx
mul edx,ebx
```

Bild 4.5: Codebeispiel

Der `add`-Befehl schreibt sein Ergebnis nach `edx` und ist datenabhängig vom `div`-Befehl, der sein Ergebnis nach `eax` schreibt. Der `mul`-Befehl schreibt sein Ergebnis ebenfalls nach `edx`, er darf demnach nicht vor dem `add`-Befehl ausgeführt werden. Der `sub`-Befehl ist abhängig von `ebx`, sein Ergebnis ist jedoch unabhängig von allen anderen Instruktionen. Um diesen Befehl parallel ausführen zu können, kann diese Abhängigkeit durch *Register Renaming* aufgelöst werden, indem man das `ebx`-Register durch ein internes Register ersetzt.

Die Datenabhängigkeit zwischen `div` und `add` lässt sich mittels *Register Renaming* nicht lösen. Das *Register Renaming* wird mit Hilfe von *Reservation Stations* realisiert, mit denen sich der folgende Abschnitt befasst.

Reservation Stations

Jede Ausführungseinheit (*execution unit*) besitzt eine *Reservation Station*. Dies sind Register, die die Operanden einzelner Instruktionen speichern. Die Referenzen der Instruktionen auf Register werden durch Referenzen auf einzelne Positionen der Reservation Stations ersetzt. Dieser

Prozess entspricht dem Register Renaming. Die Operanden einer Instruktion werden direkt an die Ausführungseinheit übergeben, sobald der Befehl ausgeführt werden kann.

Dadurch wird durch das Renaming auch gleich eine Zuteilung der Instruktionen auf die verschiedenen Ausführungseinheiten getroffen. Die Instruktionen werden den Ausführungseinheiten zugewiesen, die gerade genügend Platz in ihrer Reservation Station haben. Bei mehreren Schreibzugriffen auf ein und dasselbe Register (vgl. das Codebeispiel in Bild 4.5) wird nur der letzte Zugriff tatsächlich ausgeführt.

Die Resultate, die einzelne Instruktionen produzieren, werden direkt an die nächste Reservation Station weitergegeben, die dieses Resultat benötigt. Dafür ist ein internes Bus-System (*Common Data Bus, CDB*) nötig, das die einzelnen Reservation Stations miteinander verbindet. Gibt es mehrere gleichartige Ausführungseinheiten (Pipelines) sind auch mehrere solcher Bussysteme nötig.

Spekulative Programmausführung

Das Register Renaming und die Reservation Stations sind die zentralen Bestandteile des Tomasulo Algorithmus. Damit kann die im Instruktionsfluss vorhandene Parallelität für die parallele Ausführung der Instruktionen mit Hilfe einer Pipeline realisiert werden. Diese Art der Befehlsbearbeitung nennt man auch *dynamische Befehlsausführung*.

Diese Vorgehensweise stößt aber an ihre Grenzen, wenn der Code bedingte Sprungbefehle enthält: Es können solange keine neuen Befehle in die Pipeline geladen werden, bis der Sprungbefehl komplett ausgeführt ist. Erst dann ist bekannt, an welcher Adresse der nächste auszuführende Befehl steht. Um dieses Problem zu lösen, wird zunächst eine *Sprungvorhersage (Branch Prediction)* durchgeführt, welche unmittelbar nach der Dekodierung des Sprungbefehls die nächste auszuführende Instruktion liefert. Wie diese Vorhersage realisiert ist, wird in Abschnitt 4.2.3 behandelt.

Die Sprungvorhersage trifft eine Vermutung, ob der Sprungbefehl verzweigt oder nicht. Daraufhin führt der Prozessor *spekulativ* die entsprechenden Befehle aus. Wenn der Sprungbefehl komplett abgearbeitet ist, wird das tatsächliche Ergebnis mit der Vorhersage verglichen. War die Vorhersage korrekt, können die spekulativ ausgeführten Befehle beendet (*retired*) werden, ansonsten wird die Pipeline geleert (*pipeline flush*), und es wird mit der Befehlsabarbeitung an der korrekten Stelle fortgefahren.

Um die spekulative Ausführung realisieren zu können, muss es eine Möglichkeit geben, die Ausführung (*execution*) und die Beendigung (*retirement*) eines Befehls zu trennen. Dazu wird der *Reorder Buffer (ROB)* eingeführt. Dieser Puffer nimmt alle dekodierten Befehle auf und verteilt sie inklusive ihrer Operanden an die Reservation Stations der Ausführungseinheiten. Fertig bearbeitete Befehle werden solange im ROB gespeichert, bis sie nicht mehr spekulativ sind. Erst dann werden die Ergebnisse von den internen Registern (ROB) in die physikalischen Register übertragen (*instruction retirement*).

Spekulativ ausgeführte Befehle können keine Exceptions auslösen. Eine Exception kann erst dann auftreten, wenn die Befehle beendet werden und damit nicht mehr spekulativ sind. Ähnliches gilt für „teure" Spekulationen: Befinden sich die spekulativ auszuführenden Befehle nicht im Cache sondern im Hauptspeicher, wird kein Speicherzugriff initiiert, da dieser im Falle einer Fehlspekulation nur mit sehr viel Aufwand (Zeitverlust) wieder rückgängig gemacht werden könnte. In der Regel wird nur ein L1-Cache Miss durchgeführt. Wie viele Aktionen tatsächlich spekulativ angestoßen werden, wird auch als *Spekulationstiefe* bezeichnet. In der Regel hat diese den Wert eins, das heißt, nur L1-Cache Misses werden durchgeführt, keine L2-Cache Misses oder TLB Misses.

Bild 4.6 zeigt in einer schematischen Übersicht, wie der Tomasulo-Algorithmus prinzipiell arbeitet. Die Instruktionen werden vom I-Cache geholt (*fetch*) und dann dekodiert (*decode*). Die

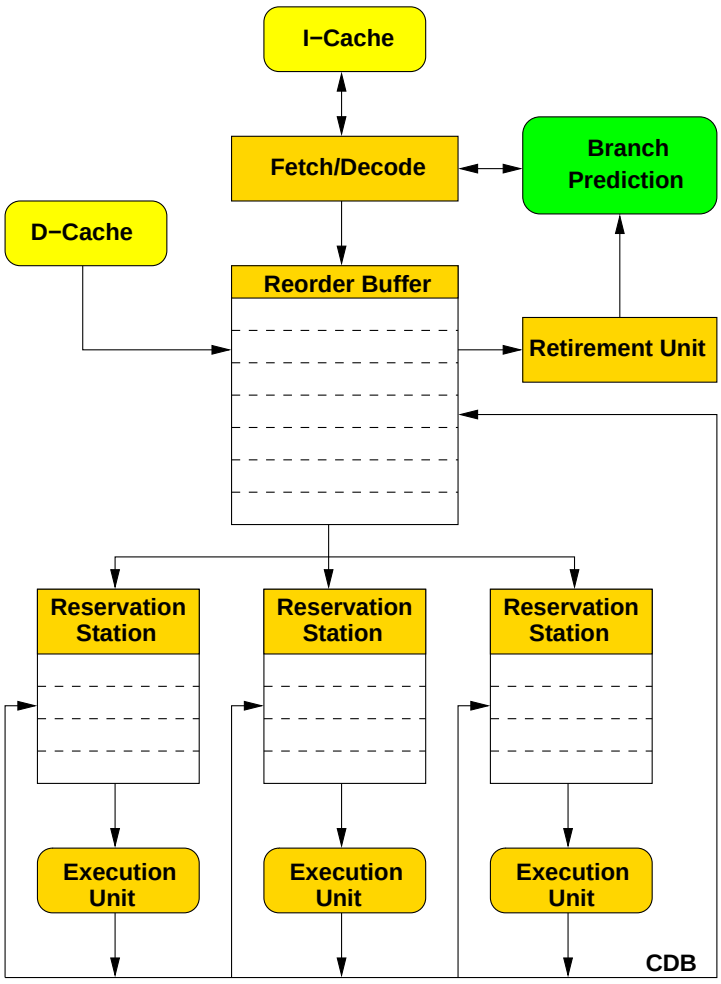

Bild 4.6: Befehlsabarbeitung nach dem Tomasulo-Algorithmus (schematisch)

dekodierten Befehle werden in einem freien Eintrag im ROB abgelegt. Ist kein freier Eintrag mehr vorhanden, muss der Dekoder warten. Während der Dekodierung wird im Falle eines Sprungbefehls mittels der Branch Prediction eine Vorhersage getroffen, wo im Programmco-

de die Ausführung fortgesetzt werden soll. Ab dieser Stelle werden dann die nächsten Befehle geladen.

Vom ROB werden die Instruktionen auf freie Plätze in den Reservation Stations verteilt. Die Registerreferenzen werden dabei durch interne Referenzen (Platz in einer Reservation Station) ersetzt. Sind alle Operanden vorhanden, wird die Instruktion ausgeführt (out-of-order execution). Die Ergebnisse der Befehle werden im entsprechenden Eintrag im ROB abgespeichert. Dort wartet die Instruktion dann solange, bis sie beendet (*retired*) werden kann. Dazu muss sie warten, bis alle vorherigen Instruktionen im Programmfluss ausgeführt worden sind (*in-order retirement*) und der Status nicht mehr spekulativ ist. Dann werden die Ergebnisse auf die externen Register übertragen und der Eintrag im ROB wieder freigegeben.

Mehr Details und Beispiele zu diesem Thema finden sich in [36]. Der nächste Abschnitt befasst sich mit der Funktionsweise der Branch Prediction und deren Auswirkungen auf die Laufzeit.

4.2.3 Branch Prediction

Unter *Branch Prediction* versteht man einen Mechanismus, der für jeden Sprungbefehl vorhersagt, ob er an seine Sprungzieladresse verzweigt oder nicht. Ist diese Entscheidung getroffen, muss die *Branch Prediction* auch die Zieladresse liefern können. Ohne diesen Mechanismus würde bei jedem Sprungbefehl die Pipeline solange keine neuen Befehle mehr bearbeiten können, bis der Sprungbefehl komplett ausgeführt wäre. Die Folge wären Pipeline Stalls, die den Geschwindigkeitsvorteil durch die parallele Abarbeitung von Befehlen in einer Pipeline erheblich reduzieren würden.

Man unterscheidet grundsätzlich zwischen einer *statischen* und einer *dynamischen* Branch Prediction (BP). Die statische BP wird angewandt, wenn ein Sprungbefehl zum ersten Mal bearbeitet wird. Anhand einer festen Regel wird entschieden, ob der Befehl verzweigen wird oder nicht. Ist der Befehl ausgeführt, wird die Sprungzieladresse in einen internen Cache abgelegt, dem *Branch Target Buffer*. Aus diesem Puffer kann dann bei einer erneuten Ausführung des Sprungbefehls die Zieladresse entnommen werden.

Die dynamische Variante der BP wird angewandt, wenn ein Sprungbefehl zum wiederholten Mal ausgeführt wird und sich dessen Zieladresse im Branch Target Buffer befindet. Anhand eines Schemas, das sowohl die Historie des aktuellen Sprungbefehls als auch die der letzten Sprünge berücksichtigt, wird entschieden, ob der Befehl verzweigen wird oder nicht. Wenn auf Verzweigung entschieden wird, werden die Befehle ab der Zieladresse, die sich im Branch Target Buffer befindet, geladen, und mit deren Ausführung begonnen. Ist der Sprungbefehl komplett bearbeitet, wird verglichen, ob die Vorhersage richtig war. Erst dann können die auf Verdacht bearbeiteten Befehle fertig bearbeitet (*retired*) werden. War die Vorhersage falsch, werden die entsprechenden Befehle aus der Pipeline gelöscht und die richtigen Befehle geladen. Details, wie der Branch Prediction Mechanismus bei den AMD oder Intel Prozessoren funktioniert, befinden sich in [14].

Die Verzögerungszeit, die für einen falsch vorhergesagten Sprung zu berechnen ist, hängt von der Länge der Pipeline des Prozessors ab. Beim AMD-Athlon beträgt sie beispielsweise zehn

Taktzyklen. Die Befehle, die auf Verdacht geladen und bearbeitet werden, können keine Cache Misses auslösen, da sie erst komplett ausgeführt werden, wenn bestätigt wird, dass die Vorhersage richtig war. Für die Ausführungszeiten von Software müssen demnach keine Cache Misses betrachtet werden, die auf eine falsche Sprungvorhersage zurückzuführen sind.

Worst-Case Szenario

Ein Worst-Case Szenario bezüglich der Branch Prediction setzt sich aus folgenden beiden Bedingungen zusammen:

- Für jeden Sprungbefehl müssen die Mechanismen der BP so eingestellt sein, dass sie die maximal mögliche Anzahl an falschen Vorhersagen liefern. Was das im Detail bedeutet, hängt von der Implementierung der BP ab und wird für den AMD-Athlon Prozessor in [14] gezeigt.
- Die Eingangsdaten für datenabhängige Verzweigungen, die mehrmals durchlaufen werden, müssen so gewählt werden, dass möglichst unregelmäßige Sprungmuster entstehen bzw. die jeweiligen Sprungmuster sich nicht so schnell wiederholen. Dadurch kann die BP nicht von Lerneffekten profitieren.

Des Weiteren spielen die Positionen der Sprungbefehle im Speicher eine Rolle für die BP. Beispielsweise kann die BP des AMD-Athlon Prozessors innerhalb eines 16 Byte großen Codeabschnitts maximal zwei Sprungbefehle bearbeiten. Befinden sich mehr als zwei Sprungbefehle innerhalb eines solchen Codeabschnitts, kann für diese Befehle keine Vorhersage durchgeführt werden und es entsteht ein Pipeline Stall von zehn Taktzyklen. Betrachtet man jedoch ein gegebenes, unveränderbares Codestück, spielt dies für das Worst-Case Szenario keine Rolle.

Mechanismen der Branch Prediction

Die erste Bedingung setzt voraus, dass man die Mechanismen des Prozessors für die BP (z. B. Branch Target Buffer, Global Branch History Counter und Global Branch History Register beim AMD-Athlon Prozessor, siehe auch [14]) manipulieren kann. Das ist in der Regel auf direktem Wege nicht möglich. Dies kann nur indirekt geschehen, indem man bestimmte Sprungmuster vor dem betrachteten Codestück ausführt, die die Mechanismen der BP mit den gewünschten Werten hinterlassen. Der Branch Target Buffer kann leicht gelöscht werden.

Im Falle des AMD-Athlon Prozessors kann ein Sprung aufgrund der Ausgangswerte der BP-Mechanismen maximal zweimal falsch vorhergesagt werden. Indem man den Branch Target Buffer löscht, kann bei der ersten Ausführung keine vollständige Vorhersage getroffen werden, da die Sprungzieladresse nicht zur Verfügung steht. Als Unsicherheit bleibt die zweite mögliche falsche Vorhersage.

Manche Prozessoren stellen *Branch Hints* zur Verfügung. Dies sind Präfixe für Sprungbefehle, die es ermöglichen, die Vorhersage für diesen Sprungbefehl anzugeben. Dies würde allerdings eine Änderung im Code des Programms erfordern und ist daher keine zweckmäßige Methode.

Eine andere Möglichkeit ist, vor der Messung ein Sprungmuster durchzuführen, was die Mechanismen für die betrachteten Sprünge so hinterlässt, dass sie die zweite falsche Vorhersage liefern. Diese Sprungmuster müssten allerdings für jedes Stück Code neu implementiert werden und können sehr aufwändig werden.

Eine weitere Möglichkeit wäre, die Messung hinreichend oft zu wiederholen und vor jeder neuen Messung die Mechanismen der BP zufällig zu manipulieren. Diese Methode wurde von Petters in [56] vorgeschlagen. Der Nachteil davon ist, dass man nie garantieren oder nachweisen kann, dass man jetzt tatsächlich für jeden Sprungbefehl den Worst-Case gemessen hat.

Eine praktikablere Lösung ist für jeden bedingten Sprungbefehl im Code die Verzögerungszeit für einen falsch vorhergesagten Sprung hinzuzurechnen. Die Anzahl der betreffenden Sprünge kann durch eine statische Codeanalyse leicht ermittelt werden. Somit kann man garantieren, dass die tatsächliche Verzögerungszeit, die durch die BP verursacht wird, weder unterschätzt noch viel zu pessimistisch angegeben wird. Bei der Beurteilung der Verzögerungszeiten muss man auch die *Spekulationstiefe* beachten (siehe Abschnitt 4.2.2). Auch korrekt vorhergesagte Sprünge können eine zusätzliche Verzögerungszeit mit sich bringen, abhängig davon, ob der spekulative Code im Cache ist oder nicht. Ein Hauptspeicherzugriff wird erst durchgeführt, nachdem der verursachende Code nicht mehr spekulativ ist. Die dadurch entstehende Laufzeitverlängerung entspricht jener für einen falsch vorhergesagten Sprung. Ob dieses Problem auftritt, hängt von der Anordnung von Code und Daten im Speicher ab und kann für jeden Sprungbefehl durch eine statische Codeanalyse ermittelt werden.

Datenabhängigkeit der Branch Prediction

Die zweite Bedingung beschäftigt sich mit falschen Vorhersagen, die nicht durch die internen Mechanismen der BP verursacht werden. Ob ein bedingter Sprungbefehl im Programmablauf verzweigt oder nicht, ist datenabhängig. Sind diese Eingangsdaten variabel, stellt sich die Frage, für welches Muster an Eingangsdaten die BP die meisten falschen Vorhersagen produziert.

Die BP kann ein regelmäßiges Verhalten sehr gut erkennen und vorhersagen. Ändern die Sprungbefehle ihr Verhalten oft und scheinbar undeterministisch, ist keine echte Vorhersage mehr möglich. Wie groß die Sprungmuster sein dürfen (das heißt, nach wievielen Sprüngen sich ein Muster wiederholt), damit sie die BP noch erkennt, hängt von der konkreten Implementierung ab. In der Regel liegen diese bei einer Länge von vier bis acht.

Von dem Verhalten der Sprünge hängt ab, welcher Programmpfad durchlaufen wird. Solche Untersuchungen werden für jede WCET-Analyse benötigt bei der es darum geht, welcher Pfad durch das Programm die WCET erzeugt. Die Ausführungszeit des Codes, dessen Ausführung durch die Programmverzweigung bedingt ist, wird in der Regel sehr viel größer sein als die wenigen Zyklen, die die BP am Anfang ausmacht. Daher genügt es, das zu untersuchende Programm mit dem Datenmuster zu vermessen, das den Worst-Case Pfad erzeugt, den man zuvor durch eine statische WCET Analyse gewonnen hat. Erzeugt man dann noch den Worst-Case bezüglich der internen BP Mechanismen, kann die WCET hinsichtlich der Einflüsse der Branch Prediction sicher ermittelt werden.

4.2.4 Caches und TLBs

Die Caches, deren Aufbau und Funktionsweise bereits in Abschnitt 3.3 vorgestellt wurde, spielen eine ganz wesentliche Rolle bei der Entstehung von WCETs. Befinden sich sowohl Code als auch Daten des auszuführenden Programms im Cache, sind die Ausführungszeiten schnell und in ihrem Verhalten deterministisch. Das gilt für den Cache wie auch für die TLBs.

Sind nicht alle benötigten Daten im Cache verfügbar, müssen diese aus dem Hauptspeicher geladen werden. Diese Speicherzugriffe dauern im Vergleich zu Zugriffen in den Cache sehr lang. Darüberhinaus sind die Zugriffszeiten abhängig davon, ob gerade noch andere Prozessoren oder Peripheriegeräte auf den Hauptspeicher zugreifen. Die Folge sind stark schwankende Zugriffszeiten, die für eine WCET-Analyse berücksichtigt werden müssen. Dasselbe gilt für die TLBs, da vor dem eigentlichen Speicherzugriff zunächst einmal die physikalische Adresse berechnet werden muss, wofür bei einem TLB-Miss wiederum Speicherzugriffe nötig sind. Diese

```
1   unsigned long a[LENGTH];

    void task1(void) {
      unsigned int i;
5
      for (i=0; i<LENGTH; i++)
        a[i] = LENGTH-i;
    }
```

Bild 4.7: Testprogramm für Speicherzugriffe

Laufzeitunterschiede sollen anhand eines AMD-Athlon SMP-Systems (2 × 1533 MHz) gezeigt werden. Dafür wird das Programm aus Bild 4.7 herangezogen. Es wird die Zeit gemessen, die nötig ist, um die Daten in das Datenfeld `a` zu laden (Zeile 6 bis Zeile 7 in Bild 4.7).

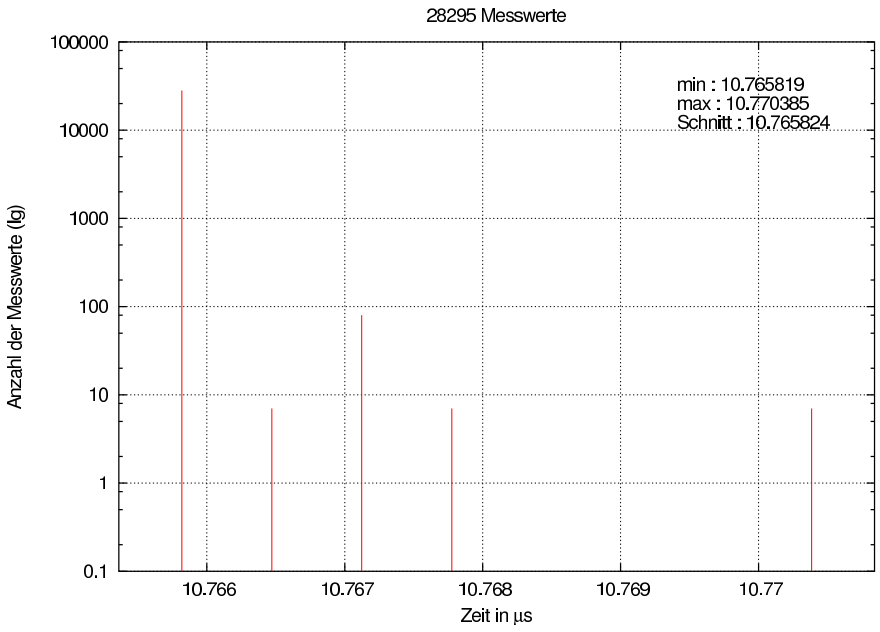

Bild 4.8: Laufzeitmessung, wenn Code und Daten im Cache sind

Bild 4.8 zeigt das Ergebnis der Messung. Die Messung wurde mit der Messroutine aus Abschnitt 4.1.1 durchgeführt. Code und Daten des Programms befinden sich vollständig im Cache. Es wurden 32 kB Daten (LENGTH = $32 \cdot 1024/4 = 8192$) in das Datenfeld geladen. Die Schwankungsbreite der gemessenen Zeiten ist sehr klein und entspricht sieben Taktzyklen. Das liegt in der Größenordnung einer Verzögerung, wie sie beispielsweise durch Pipeline Stalls verursacht wird.

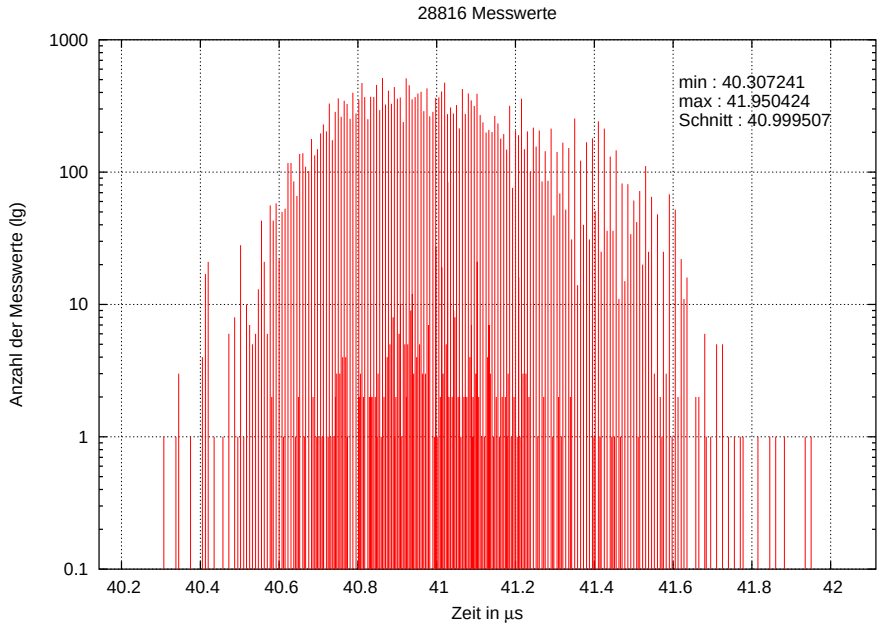

Bild 4.9: Laufzeitmessung, wenn Code und Daten *nicht* im Cache sind

Bild 4.9 zeigt dieselbe Messung mit dem Unterschied, dass die Daten für das Datenfeld `a` aus dem Hauptspeicher geladen werden müssen. Vor der Messung wurde sichergestellt, dass kein anderes Gerät in dieser Zeit auf den Hauptspeicher zugreift. Es handelt sich demnach um ungestörte Speicherzugriffe. Die Schwankungsbreite beträgt ca. 1,6 μs bei einer Gesamtlaufzeit von ca. 41 μs, das entspricht 4%. Die Ausführungszeit steigert sich im Vergleich zur vorherigen Messung um den Faktor 3,8. Es handelt sich hier nur um Lesezugriffe, zusätzliche Schreibzugriffe würden diesen Faktor nochmals verdoppeln. Die Abstände zwischen den Messwerten in Bild 4.9 entsprechen jenen, die für Transferoperationen vom L2-Cache in den L1-Cache oder vom L1-Cache in den Victim-Buffer benötigt werden (siehe dazu auch Abschnitt 3.3).

Bild 4.10 zeigt nochmals dieselbe Messung, diesmal jedoch mit parallelen Speicherzugriffen von einem zweiten Prozessor und PCI-Geräten (Festplatte, Ethernet). Die großen Laufzeitschwankungen sind deutlich zu erkennen. Sie werden durch Blockierzeiten beim Zugriff auf den Hauptspeicher erzeugt. Der gemessene maximale Wert liegt bei 66,8 μs, die Laufzeitschwankung bei 26,32 μs, das entspricht 39,4%, rund dem Zehnfachen der Laufzeitschwankung aus Bild 4.9. Verglichen mit der Laufzeit des Codes im Cache aus Bild 4.8 erhöht sich die Laufzeit im Worst-Case um den Faktor 6,2.

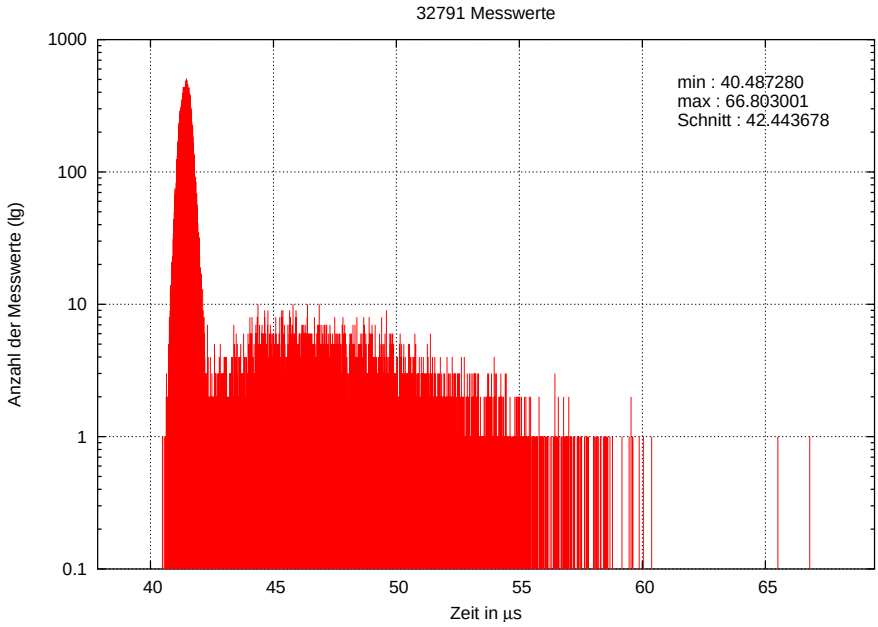

Bild 4.10: Laufzeitmessung mit parallelen Hauptspeicherzugriffen

Diese Messungen zeigen den enormen Laufzeitunterschied zwischen Zugriffen auf den Cache und Zugriffen auf den Hauptspeicher. Für ein Realzeitsystem müssen diese Laufzeitunterschiede unbedingt mit berücksichtigt werden, da sie großen Einfluss auf die WCET von Software haben. Auch die Schwankungsbreite der gemessenen Laufzeiten spielt eine große Rolle. Für eine WCET-Analyse muss man die jeweils größten ermittelten Zeiten verwenden, auch wenn diese sehr selten auftreten und deutlich größer als die durchschnittlichen Ausführungszeiten sind. Die WCET sollte möglichst dicht an der BCET liegen. Ansonsten kann die eigentliche Rechenkapazität nicht für Realzeitsysteme genutzt werden.

Das Ziel ist daher, die Caches möglichst optimal für Realzeitsoftware nutzen zu können. Die Ausführungszeiten für Software in den Caches sind am schnellsten und mit den geringsten Laufzeitschwankungen behaftet. Die WCET liegt dicht an der BCET und kann sicher bestimmt werden. Die Verzögerungen, die durch Hauptspeicherzugriffe entstehen, sind sehr viel größer als jene, die durch Pipeline Stalls oder falsch vorhergesagte Sprungbefehle verursacht werden. Dies betrifft nicht nur Code- oder Daten-Cache Misses, sondern auch Misses in den TLBs. Alle diese Misses hängen neben der Codestruktur vor allem von der Lage der Programme im Hauptspeicher ab (siehe auch Abschnitt 3.3).

In Kapitel 5 wird ein Verfahren vorgestellt, mit dem man die Caches für Realzeitsoftware optimal nutzen kann. Cache Misses und deren Folgen werden durch dieses Verfahren in ihrem Verhalten vorhersagbar, so dass sie in einer statischen WCET-Analyse berücksichtigt werden können. Diese Methoden berücksichtigen sowohl die Caches als auch die TLBs.

Der folgende Abschnitt beschäftigt sich mit den Auswirkungen des *Cache-Snooping* auf die Laufzeiten von Software.

4.2.5 Cache-Snooping in Multiprozessorsystemen

Das *Cache-Snooping* Protokoll sorgt bei Multiprozessorsystemen dafür, dass der Inhalt der Caches der einzelnen Prozessoren und der des Hauptspeichers zu jedem Zeitpunkt konsistent ist. Das Protokoll ist in der Regel komplett in Hardware realisiert. Es werden sämtliche Speichertransfers innerhalb der Caches eines Prozessors und zwischen den Prozessoren und dem Hauptspeicher beobachtet (daher *snooping* = *schnüffeln*). Wird dabei eine drohende Inkonsistenz bemerkt, wird diese durch entsprechende Maßnahmen sofort behoben. Bild 4.11 zeigt ein Beispiel.

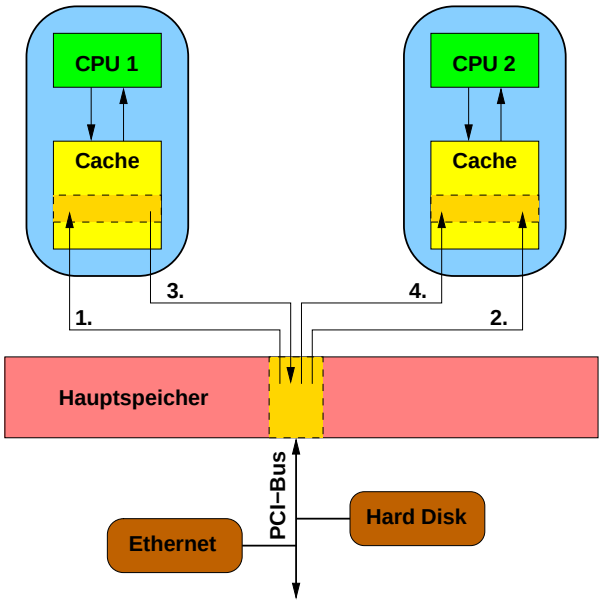

Bild 4.11: Cache-Snooping Szenario

Prozessor 1 (CPU 1) lädt in Schritt 1 einen Speicherbereich in seinen Cache. In Schritt 2 lädt Prozessor 2 (CPU 2) denselben Speicherbereich in seinen Cache. Verändert nun Prozessor 1 den Inhalt dieses Speicherbereichs, wird diese Änderung zunächst nur innerhalb des Caches von Prozessor 1 durchgeführt. Jetzt könnte es passieren, dass Prozessor 2 mit dem mittlerweile veralteten Inhalt seines Caches weiterarbeitet und daher falsche Ergebnisse produziert.

Um das zu verhindern, greift an dieser Stelle der Cache-Snooping Mechanismus ein. Er erzwingt zunächst einen Schreibzugriff von Prozessor 1 in den Hauptspeicher (Schritt 3). Danach wird Prozessor 2 aufgefordert, diesen Speicherbereich neu zu lesen (Schritt 4). Somit enthält dann auch der Cache von Prozessor 2 den aktuellen Inhalt des betreffenden Speicherbereichs. Während dieses Synchronisierungsvorgangs muss Prozessor 2 mit der Befehlsausführung warten, sofern diese von den Daten dieser Cacheline abhängt. Somit ist gewährleistet, dass die Daten jederzeit über alle Prozessoren hinweg konsistent sind.

Es gibt auch Implementierungen, die die Daten direkt zwischen den Caches austauschen, ohne Umweg über den Speicher (siehe Abschnitt 5.3.5). Dadurch nimmt der Synchronisierungsvor-

gang insgesamt weniger Zeit in Anspruch und ist geringeren Laufzeitschwankungen unterworfen.

Die Notwendigkeit, den Inhalt der Caches und den des Hauptspeichers kohärent zu halten, tritt nicht nur auf, wenn sich zwei oder mehrere Prozessoren den Speicher teilen. Peripheriegeräte, die autonom auf den Hauptspeicher zugreifen können (in Bild 4.11 beispielsweise PCI-Geräte), können auch Daten verändern, die der Prozessor gerade in seinem Cache hält. In diesem Fall muss diese Inkohärenz auch durch das Cache-Snooping Protokoll behoben werden.

Tritt der Fall ein, dass der Cache-Snooping Mechanismus die Caches von zwei oder mehreren Prozessoren synchronisieren muss, bedeutet dies für jeden Prozessor der synchronisiert wird, dass er auf zwei Speicherzugriffe warten muss. Diese Zugriffe können die Ausführungszeit deutlich verlängern, wie im vorhergehenden Abschnitt gezeigt wurde. Hinzu kommt, dass diese Zugriffe nicht durch den Prozessor selbst, sondern durch einen anderen Prozessor verursacht wurden. Das heißt, sie können nicht durch eine Analyse der Software des betroffenen Prozessors erkannt und gegebenenfalls für eine WCET-Analyse mit berücksichtigt werden.

Daher ist es wichtig, solche Laufzeitverzögerungen von vornherein auszuschließen. Eine Synchronisierung von Speicherinhalten kann nur dann erforderlich werden, wenn identische Speicherbereiche in den Caches unterschiedlicher Prozessoren liegen. Das kann durch eine geeignete Speicherverwaltung, wie sie in Kapitel 5 vorgestellt wird, vermieden werden. Jeder Prozessor bekommt seinen eigenen, physikalischen (für physikalisch indizierte Caches, sonst virtuell) Adressraum zugewiesen. Für Speicherbereiche, die für den Datenaustausch zwischen Tasks auf unterschiedlichen Prozessoren verwendet werden, kann der Einfluss durch Cache-Snooping dadurch verhindert werden, dass diese Bereiche als *uncacheable* markiert werden und folglich nicht in den Cache geladen werden.

Mehr Details dazu finden sich in Kapitel 5. In Abschnitt 5.3.5 wird das Cache-Snooping Protokoll der AMD-Athlon Prozessoren detailliert beschrieben. Der folgende Abschnitt widmet sich der Frage, welchen Einfluss Interrupts auf die Laufzeit von Software haben.

4.2.6 Interrupts

Ein Interrupt ist eine Unterbrechung des aktuell ausgeführten Programms, um ein anderes, in der Regel sehr kurzes Programm (*Interrupt Service Routine*, ISR), auszuführen. Nachdem die ISR beendet ist, wird das zuvor unterbrochene Programm fortgesetzt. Interrupts können von einer externen Quelle (Peripheriegeräte) ausgelöst werden (*Hardware Interrupts*) oder programmgesteuert durch Software (*Software Interrupts*). Ist ein Fehler im Programmablauf eingetreten, können *Exceptions* generiert werden. Das sind ebenfalls Interrupts, die vom Prozessor selbst ausgelöst werden und einen kritischen Fehler im Programmablauf anzeigen (z. B. Division durch null). Die dazugehörigen Fehlerbehandlungsroutinen nennt man in Analogie zu den ISRs *Exception Handler*.

Nach der Ausführung einer ISR soll das unterbrochene Programm fortgesetzt werden können. Dazu ist es nötig, den aktuellen Programmkontext (Register, Stack) zu sichern und die Befehlsabarbeitung zu serialisieren (siehe Abschnitt 4.1.1). Dann muss ein Kontext für die ISR

eingerichtet und eventuell die Privilegierungsstufe des Prozessors gewechselt werden. Die meisten Prozessoren implementieren mehrere Privilegierungsstufen, welche einzelne Prozesse, die in verschiedenen Privilegierungsstufen ausgeführt werden, voneinander abschotten. Das heißt, ein Prozess in einer niedrigeren Stufe darf nicht auf Daten eines Prozesses auf einer höheren Stufe zugreifen. Diese Eigenschaft kann zur Implementierung von Speicherschutzmechanismen verwendet werden.

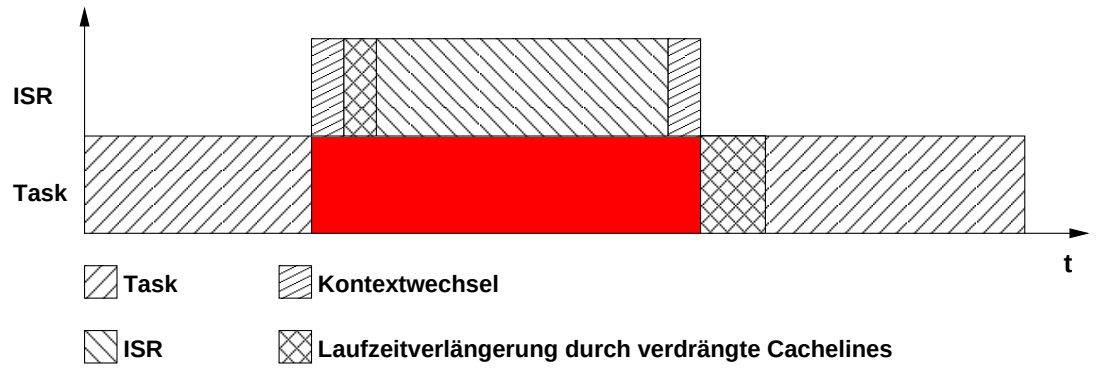

Bild 4.12: Laufzeitverlängerung durch Interrupts

Eine ISR wird ausgeführt wie ein normales Programm, das heißt, sie benutzt ebenso die Caches und TLBs. Daher kann es dazu kommen, dass eine ISR Cachelines der zuvor unterbrochenen Task verdrängt. Diese Cachelines müssen nach Beendigung der ISR wieder in den Cache geladen werden, was eine zusätzliche Verzögerung der Laufzeit bedeutet. Bild 4.12 verdeutlicht den Einfluss von Interrupts auf die Ausführungszeit einer Task. Eine Laufzeitverlängerung aufgrund von zuvor verdrängten Cachelines betrifft natürlich auch die ISR selbst.

Die Laufzeitverzögerung durch Interrupts lässt sich allgemein wie folgt angeben:

$$t_{delay} = 2 \cdot t_{context} + t_{ISR} + t_{cache}$$

Die Zeit $t_{context}$ ist die Zeit, die für einen Wechsel der Privilegierungsstufe benötigt wird. Sie ist spezifisch für den jeweiligen Prozessor und konstant. Verglichen mit der Zeit t_{cache}, die für das Nachladen von Cachelines benötigt wird, ist diese Zeit klein. t_{cache} hängt von der Anzahl der Cachelines ab, die geladen werden müssen. Die Zeit t_{ISR} ist die WCET der ISR.

Die gesamte Verzögerungszeit t_{delay} hängt im Wesentlichen von der Anzahl der verdrängten Cachelines ab, die wieder geladen werden müssen und der Laufzeit der ISR. Das Ziel ist daher, die Anzahl der Cache Misses zu minimieren und unvermeidbare Cache Misses für eine WCET-Analyse vorhersagbar zu machen. Dies wird mit der Speicheranordnung erreicht, die in Kapitel 5 vorgestellt wird.

Das zeitliche Verhalten von Hauptspeicherzugriffen spielt eine zentrale Rolle bei der Betrachtung der WCET auf modernen Mikroprozessorarchitekturen. Dieses Verhalten wird maßgeblich von der Anbindung des Prozessors an den Hauptspeicher bestimmt. Die folgenden Abschnitte befassen sich mit dieser Anbindung in SMP- und NUMA-Systemen.

4.3 Speicheranbindung in SMP- und NUMA-Systemen

Der Hauptspeicher ist eine Ressource, die sich oft mehrere Geräte in einem System teilen müssen. Daher kann es vorkommen, dass mehrere Geräte zur selben Zeit auf den Speicher zugreifen wollen und es zu Laufzeitverzögerungen kommt, weil ein Gerät warten muss, bis ein anderes Gerät seinen Speicherzugriff beendet hat. Die Länge dieser Blockierzeiten hängt davon ab, wie die Geräte in einem System mit dem Hauptspeicher verbunden sind.

4.3.1 SMP-Systeme

Als SMP-System (siehe Abschnitt 3.2.2) wird eine Architektur bezeichnet, welche n gleichartige Prozessoren mit identischen Rechten beherbergt. Alle Prozessoren teilen sich den Hauptspeicher, jeder Prozessor darf Interrupts bekommen und abarbeiten oder selbst Interrupts generieren. Jeder Prozessor kann jede Task bearbeiten und gleichberechtigt auf jede Ressource des Systems zugreifen. Bild 4.13 zeigt den schematischen Aufbau eines SMP-Systems mit Host-Bus.

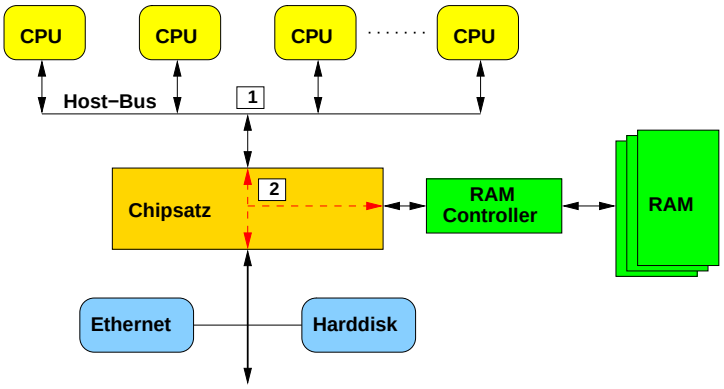

Bild 4.13: Aufbau eines SMP-Systems

Bild 4.13 zeigt zwei Konfliktpunkte, an denen sich die Datenpfade auf dem Weg zum Speicher kreuzen: Die Prozessoren teilen sich den Host-Bus (Punkt 1), und die Peripheriegeräte teilen sich mit allen Prozessoren den Speicher (Punkt 2). Es gibt auch Systeme, in denen jeder Prozessor eine eigene Punkt-zu-Punkt Verbindung zum Chipsatz hat. Bei diesen Systemen fallen Punkt 1 und Punkt 2 zusammen.

Die Prozessoren in einem SMP-System behindern sich beim Zugriff auf den Hauptspeicher, unabhängig davon, welche Adressen sie gerade verwenden. Untersuchungen in [73] haben gezeigt, dass diese Konkurrenzsituation Laufzeitverzögerungen bis zu 100% verursachen kann. Die genaue Verzögerung pro Zugriff eines Prozessors kann nicht angegeben werden. Aufgrund der Symmetrie des Systems ist die Blockierdauer jedoch begrenzt. Diese Grenze hängt von der Anzahl der Prozessoren im System ab. Wenn t_B die maximal erlaubte Zugriffsdauer für einen Prozessor bezeichnet und das System aus n Prozessoren besteht, beträgt die maximale Blockierzeit pro Prozessor $t_{B,CPU} = (n-1) \cdot t_B$. Die Größe von t_B hängt vom jeweiligen Chipsatz ab

und kann herstellerbedingt leicht schwanken. Diese Zeit kann aber durch Messungen ermittelt und so für eine WCET-Analyse genutzt werden.

Auch Peripheriegeräte können autonom auf den Hauptspeicher zugreifen und stehen in Konkurrenz zu den Prozessoren. Wie die Zugriffsvergabe genau geregelt wird, hängt wiederum vom Chipsatz ab. Untersuchungen haben gezeigt, dass die auftretenden Blockierzeiten von Hersteller zu Hersteller merklich schwanken können. Diese müssen durch Messungen auf der jeweiligen Zielhardware ermittelt werden. In der Regel ist die Zugriffsvergabe jedoch so organisiert, dass die Peripheriegeräte insgesamt so viel Zugriffszeit auf den Speicher bekommen, wie ein Prozessor. Das heißt, die Blockierungszeiten durch die Peripheriegeräte wirken sich aus Sicht eines Prozessors so aus, als ob noch ein Prozessor mehr im System wäre.

Ein Beispiel für den Einfluss von konkurrierenden Speicherzugriffen auf die Laufzeit von Software zeigen die Bilder 4.8, 4.9 und 4.10 aus Abschnitt 4.2.4. Weiterführende Untersuchungen zu dieser Problematik sind in [73] zu finden. Der folgende Abschnitt befasst sich mit der Situation bei NUMA-Systemen.

4.3.2 NUMA-Systeme

Als NUMA-System (siehe Abschnitt 3.2.2) wird eine Architektur bezeichnet, die im Gegensatz zu SMP (vgl. Abschnitt 4.3.1) für jeden Prozessor einen eigenen, physikalischen Speicher vorsieht. Bild 4.14 zeigt den schematischen Aufbau eines NUMA-Systems.

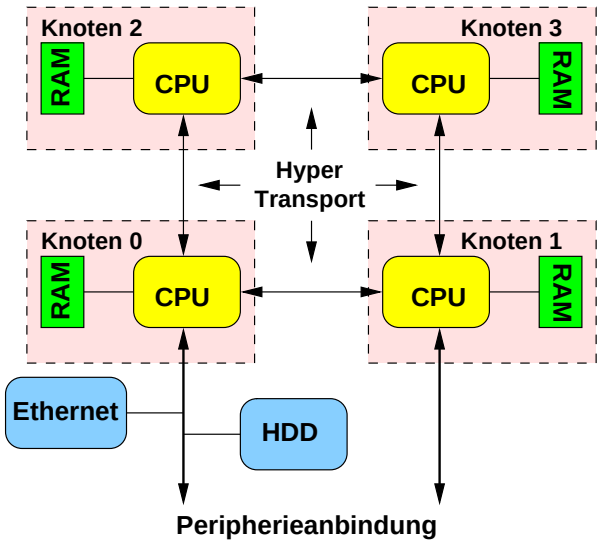

Bild 4.14: Aufbau eines NUMA-Systems

Im Gegensatz zur SMP-Architektur müssen die Prozessoren nicht den Weg durch einen zentralen Chipsatz gehen, wenn sie auf ihren Speicher zugreifen wollen. Einer oder mehrere Knoten sind direkt mit einem Peripheriebus verbunden, die anderen Prozessoren können über die HyperTransport Links (siehe Abschnitt 3.2.4) auf die Peripheriegeräte zugreifen.

Jeder Prozessor kann auch auf den Speicher anderer Prozessoren zugreifen. Diese Zugriffe erfolgen über die HyperTransport Links. Da nicht jeder Knoten mit jedem anderen Knoten direkt verbunden ist, müssen bestimmte Zugriffe einen dazwischen liegenden Knoten „tunneln", das heißt, der entsprechende Zugriff wird weitergeleitet. In Bild 4.14 ist das zum Beispiel für einen Zugriff von Knoten 2 auf Knoten 1 der Fall. Tabelle 4.2 zeigt die Messergebnisse für das Lesen von 64 kB Daten von verschiedenen Knoten eines NUMA-Systems, dessen Aufbau dem aus Bild 4.14 entspricht. Die Ergebnisse zeigen den deutlichen Unterschied zwischen Zugriffen

Nach / Von	Knoten 1	Knoten 2	Knoten 3
Knoten 1	83 μs	110 μs	97 μs
Knoten 2	110 μs	83 μs	97 μs
Knoten 3	97 μs	97 μs	83 μs

Tabelle 4.2: Zugriffszeiten auf Speicherbereiche verschiedener Knoten

auf benachbarte Knoten und solchen, die einen anderen Knoten tunneln müssen. Die Richtung der Zugriffe spielt dabei keine Rolle. An den Ergebnissen ändert sich auch nichts, wenn zwei Zugriffe parallel über Kreuz erfolgen, beispielsweise von Knoten 1 nach Knoten 3 und umgekehrt. Das liegt an dem HyperTransport System, das jeweils einen unabhängigen Link in jede Richtung vorsieht.

Aus der Sicht eines Realzeitsystems gesehen, bringen NUMA-Systeme einige Vorteile gegenüber SMP-Systemen:

- Jeder Prozessor hat seinen lokalen Speicher. Solange ein Prozessor nur Software ausführt, die in seinem lokalen Speicher liegt, gibt es keine Einflüsse durch Cache-Snooping (siehe Abschnitt 4.2.5). Zugriffe auf den lokalen Speicher erfolgen ohne Konkurrenz anderer Prozessoren (höchstens durch Peripheriegeräte) und sind somit schneller und deterministischer.
- Peripheriebusse hängen an mehreren Knoten. Das heißt, man kann die Geräte, die nur von Tasks auf einem bestimmten Prozessor benutzt werden, alle an diesem einen Peripheriebus anschließen. Damit treten entsprechende Konflikte nur auf diesem Knoten auf, alle anderen Prozessoren bleiben unbehelligt.
- Die HyperTransport Links sind Punkt-zu-Punkt Verbindungen. Da es sich hier nicht um einen klassischen Bus handelt, sind die Laufzeiten für Zugriffe deutlich weniger Schwankungen unterworfen. Die Zugriffszeiten sind dadurch kleiner und deterministischer. Auch die Zeit zum Tunneln einzelner Knoten ist nahezu lastunabhängig.

Um diese Vorteile nutzen zu können, müssen die Realzeittasks entsprechend ihrer Ressourcennutzung auf die verschiedenen Knoten verteilt werden. Mit diesem Aspekt beschäftigt sich Abschnitt 5.4. Es ist auch möglich, die SMP- und die NUMA-Architektur zu kombinieren. Ein Knoten kann aus mehreren Prozessoren bestehen, die innerhalb des Knotens ein SMP-System bilden. Innerhalb dieses Knotens treten dann dieselben Probleme auf, wie in einem klassischen SMP-System. Nach Außen verhält sich der Knoten, als ob er nur aus einem Prozessor bestehen würde.

5 Anordnung von Code und Daten im Speicher

In diesem Kapitel wird zunächst ein Algorithmus vorgestellt, mit dessen Hilfe Code und Daten eines Realzeitsystems im Hauptspeicher so angeordnet werden können, dass die Caches des Rechners aus Sicht eines Realzeitsystems optimal genutzt werden. Verdrängungsvorgänge im Cache können kontrolliert werden und die Zeit für einen Verdrängungsvorgang aus dem Cache in den Hauptspeicher kann besser vorhergesagt werden. Mit diesem Algorithmus ist es möglich, Teile eines Realzeitsystems permanent im Cache zu platzieren (Cache-Locking).

Zunächst werden die Ziele des Algorithmus definiert und dessen Einsatz für Realzeitsysteme motiviert. Der nächste Abschnitt beschreibt detailliert die Ansätze zur Anordnung von Code und Daten im Cache. Anschließend wird die Transformation einer Anordnung von Code und Daten im Cache in eine entsprechende Anordnung im Hauptspeicher diskutiert. Dabei wird auch ausführlich auf die Nutzung der TLBs eingegangen.

Der folgende Abschnitt befasst sich mit Strategien zur intelligenten Speicherverwaltung für Realzeitsysteme. Dabei wird auf den parallelen Betrieb eines Echtzeit- und eines Standardbetriebssystems und auf die Kommunikation in Echtzeit zwischen Realzeit-Tasks und Tasks des Standardbetriebssystems und Realzeit-Tasks untereinander eingegangen. Schwerpunkt ist die Anordnung entsprechender Mechanismen zur Interprozesskommunikation im Speicher (z. B. shared memory) und der Einfluss dieser Anordnung auf die WCET von Software. Es wird eine Methodik vorgestellt, wie man das MOESI-Protokoll zum schnellen Austausch kleiner Datenmengen zwischen verschiedenen Prozessoren nutzen kann.

Aus den Überlegungen für eine optimale Nutzung der Caches für Realzeit-Tasks ergeben sich wichtige Aspekte für eine Zuordnung von Realzeit-Tasks auf mehrere Prozessoren, die im Anschluss diskutiert werden. Das Kapitel wird von Betrachtungen für einen Echtzeitnachweis abgeschlossen.

5.1 Motivation und Zielsetzungen

Der Einsatz von Caches in modernen Mikroprozessoren beschleunigt die *durchschnittliche* Ausführungszeit von Software erheblich. Die Beschleunigung, die durch den Einsatz von Caches zu erreichen ist, hängt von dem Quotienten der Zugriffszeit auf den Hauptspeicher und der Zugriffszeit auf den Cache (in der Regel volle Taktfrequenz des Prozessors) ab. Dieser *Beschleunigungsfaktor* liegt bei modernen PC-Prozessoren in einer Größenordnung von 15.

Durch diese großen Unterschiede bei den Zugriffszeiten ergeben sich sehr unterschiedliche Laufzeiten für Software, die davon abhängen, wie oft der Prozessor bei der Programmausführung auf den Cache oder auf den Hauptspeicher zugreifen muss. Moderne Prozessoren verfügen nicht nur über Caches für Code und Daten, sondern auch über Translation Lookaside Buffers (TLBs), die die Informationen, die für die Umrechnung einer virtuellen (logischen) in eine physikalische Adresse benötigt werden, zwischenspeichern. Wenn auf eine Speicheradresse zugegriffen werden soll, muss zunächst die virtuelle in die physikalische Adresse umgerechnet werden, wofür ein Eintrag in den TLBs oder alternativ mehrere Hauptspeicherzugriffe benötigt werden. Erst danach folgt ein Zugriff auf den Code- bzw. Daten-Cache oder den Hauptspeicher.

Ob ein benötigtes Datum in den Caches vorliegt oder nicht, hängt zum einen von der Anordnung von Code und Daten im Hauptspeicher ab, zum anderen von der Struktur des ausgeführten Codes. Die interne Struktur von Programmen im Hinblick auf die Nutzung der Caches zu optimieren, ist in erster Linie die Aufgabe von Codegeneratoren und Compilern. In dieser Arbeit wird die Software als gegeben und unveränderlich betrachtet. Das Ziel ist es, für diese Software, deren Code- und Datenobjekte frei im Speicher platzierbar sind, eine für Realzeitsysteme optimale Anordnung im Hauptspeicher zu finden. Der enorme Geschwindigkeitszuwachs bei der Ausführung von Software, den die Caches bieten können, soll für Realzeitsysteme genutzt werden. Dadurch sinkt die WCET der Software und das zeitliche Verhalten des Systems wird besser vorhersagbar.

Ziele der Speicherorganisation

Der Schwerpunkt in dieser Arbeit liegt auf Strategien, um für ein gegebenes Softwaresystem eine Anordnung von Code und Daten im Hauptspeicher zu finden, so dass

- die Anzahl der theoretisch möglichen Cache-Misses minimiert wird;
- die Art und Anzahl der verbleibenden Misses vorhersagbar ist. Das heißt, der Ort und die Art der Verdrängung im Cache und die zu berechnende zusätzliche Ausführungszeit sollen a priori ermittelbar sein;
- die zur Verfügung stehenden TLBs optimal genutzt werden können, um zusätzliche Hauptspeicherzugriffe für die Adressumrechnung zu vermeiden. Die Anzahl der möglichen TLB-Misses soll a priori bestimmt werden können;
- Cache-Locking auf Softwareebene möglich ist;
- Cache-Snooping Effekte vermieden oder kontrollierbar werden;

Diese Strategien sollen den Code nicht modifizieren, so dass beispielsweise bereits zertifizierte Programme nicht noch einmal neu zertifiziert werden müssen. Ziel dieser Anordnung ist es, die WCET von Software zu minimieren *und* die Ausführungszeiten der Realzeitsoftware in Bezug auf die Cachenutzung des Prozessors prädizierbar zu machen. Darüber hinaus soll es möglich sein, einzelne Codeabschnitte oder Daten permanent im Cache zu halten (*Cache-Locking*). Diese Möglichkeit bieten Prozessorarchitekturen, wie beispielsweise PowerPC, in Hardware an, bei der x86-Architektur ist dies nicht vorgesehen. Aus der Sicht eines Realzeitsystems kann man

mit Hilfe des Cache-Lockings die Ausführungszeiten kleiner, besonders zeitkritischer Applikationen minimieren und dafür sorgen, dass diese nahezu konstant sind.

Das Cache-Snooping wurde bereits in Abschnitt 4.2.5 ausführlich beschrieben. Das Ziel ist hier, die Folgen eines Eingreifens des Cache-Snooping Mechanismus' möglichst zu vermeiden. Wo dies nicht möglich ist, sollen die Folgen für die WCET abschätzbar sein.

Die TLBs spielen ebenfalls eine wichtige Rolle bei Speicherzugriffen. Sie werden bei jedem Speicherzugriff für die Adressrechnung benötigt. Befinden sich die benötigten Informationen nicht in den TLBs, werden zusätzliche Speicherzugriffe nötig. Dies soll durch geeignete Strategien der Speicheranordnung vermieden werden.

5.2 Anordnung von Code und Daten

In diesem Abschnitt wird ein neuer Algorithmus zur Anordnung von Code und Daten eines Realzeitsystems im Cache und im Hauptspeicher vorgestellt. Dieser Algorithmus basiert auf den Arbeiten von Hashemi [33] und Calder [16]. Die dort vorgeschlagenen Methodiken werden für den Einsatz in Realzeitsystemen mit modernen Prozessorarchitekturen erweitert und verfeinert. Zunächst wird das Szenario definiert, für das der Algorithmus zum Einsatz kommen soll. Im Anschluss werden die Ansätze zur optimalen Anordnung von Code und Daten im Cache vorgestellt. Daraus ergibt sich unter Berücksichtigung aller Mechanismen moderner Prozessorarchitekturen, insbesondere der TLBs, ein Algorithmus zur Abbildung dieser Anordnung auf den Hauptspeicher.

5.2.1 Szenario

Es werden grundsätzlich Realzeitsysteme betrachtet, die aus einem Realzeitbetriebssystem und Realzeitapplikationen bestehen. Das Gesamtsystem wird als eine Menge F von Funktionen f_i betrachtet:

$$F = \{f_i : i = 1...N\}$$

Jede Funktion f_i besteht aus Code und Daten:

$$\begin{aligned} f_i &= \{A_i, S_i, D_i\} \text{ mit} && (5.1)\\ A_i &= \{ro, rw, locked\} \text{ und} \\ D_i &= \{A_{i,k_i}, S_{i,k_i} : k_i = 1...K_i\} && (5.2) \end{aligned}$$

Der Code einer Funktion f_i wird durch sein Attribut A_i und seine Größe S_i in Bytes beschrieben. Die Datenobjekte D_i einer Funktion werden ebenfalls durch ihr Attribut A_{i,k_i} und ihre Größe S_{i,k_i} beschrieben.

Die Attribute A, die jedem Code- und Datenobjekt zugewiesen werden, beschreiben die Eigenschaften eines Objekts:

- **ro (read-only)**: Das Objekt wird während seiner Lebensdauer nicht verändert. Die Lebensdauer eines Objektes ist die Zeitspanne von der ersten bis zur letzten Referenz auf dieses Objekt.
- **rw (read-write)**: Das Objekt wird während seiner Lebensdauer mindestens einmal verändert. Bei einer Verdrängung aus dem Cache muss der korrespondierende Bereich im Hauptspeicher aktualisiert werden.
- **locked**: Das Objekt befindet sich während seiner gesamten Lebensdauer (das heißt, schon vor der ersten Referenz) im Cache und wird nie verdrängt.

Objekte mit dem Attribut *read-only* müssen bei einer Verdrängung aus dem Cache nicht in den Speicher zurückgeschrieben werden. Die entsprechenden Einträge werden einfach als ungültig markiert. Dadurch erspart man sich einen Speicherzugriff während des Verdrängungsvorganges. Im Gegensatz dazu müssen Objekte mit dem Attribut *read-write* bei einer Verdrängung in den Speicher zurückgeschrieben werden. Objekte mit diesem Attribut können auch eine Interaktion des Cache-Snooping Protokolls hervorrufen, im Gegensatz zu Objekten mit dem Attribut *read-only*.

Ein spezielles Attribut ist *locked*: Diese Objekte bleiben während ihrer Lebenszeit im Cache und werden nie verdrängt. Dabei spielt es keine Rolle, ob sie während ihrer Lebenszeit modifiziert werden oder nicht. Dieses sogenannte *Cache-Locking* wird von Prozessoren der IA-32 oder AMD64 Architektur nicht in Hardware unterstützt. Bei diesen Prozessoren muss das Cache-Locking durch eine geeignete Speicherabbildung des Realzeitsystems erreicht werden.

Jede Funktion besteht nach Gleichung 5.1 aus einem Code- und einem oder mehreren Datenobjekten. Jedes Datenobjekt wird nach Gleichung 5.2 durch ein Attribut und dessen Größe in Bytes beschrieben. Jedes Datenobjekt kann dabei ein anderes Attribut besitzen. Datenobjekte sind alle genutzten Daten einer Funktion. Das können globale Variablen sein oder ein Bereich auf dem Stack. Auch Speicherregionen für den Austausch von Daten zwischen den Tasks (shared memory) sind Datenobjekte.

Es wird ohne Beschränkung der Allgemeinheit angenommen, dass alle Informationen vor Inbetriebnahme des Realzeitsystems vorliegen. Alle Parameter bleiben konstant während der Laufzeit des Systems. Als Cachearchitektur wird ein *n-fach Set-assoziativer* Cache (siehe Abschnitt 3.3.2) mit l Sets angenommen:

$$C = \{c_j(A_j) : j = 0...l\} \tag{5.3}$$

Eine Cacheline hat die Größe m in Bytes. Die Größe eines Sets c_j mit Attribut A_j beträgt demnach $n \cdot m$ Bytes. Die Cachegröße ergibt sich damit allgemein zu $s = n \cdot m \cdot l$ Bytes. Die Methoden lassen sich auch einfach auf einen *direct mapped* ($n = 1$) Cache übertragen. Der Cache wird physikalisch indiziert, das heißt, die physikalische Adresse eines Objekts im Hauptspeicher korrespondiert mit der Position des Objekts im Cache. Die Methodik der Anordnung kann aber auch leicht auf virtuell indizierte Caches übertragen werden. Zunächst wird die Anordnung im Cache von nur einem Prozessor betrachtet.

5.2.2 Ein Ansatz zur Anordnung von Code und Daten im Cache

Die Anordnung erfolgt statisch vor Inbetriebnahme des Systems. Diese Anordnung ist dynamisch zur Laufzeit nicht veränderbar. Das heißt nicht, dass das zeitliche Verhalten des Systems genau bekannt sein muss: Es muss nur bekannt sein, welche Codeabschnitte (z. B. Tasks, ISRs, Systemcalls) und welche Daten benutzt werden, nicht wann sie genutzt werden. Erst eine Hinzunahme von zuvor nicht berücksichtigten Code- oder Datenobjekten erfordert eine neue Anordnung des Systems.

Die Anordnung selbst erfolgt zunächst im Cache, bei mehreren Cache-Ebenen in der höchsten (z. B. L1-Cache). In einem zweiten Schritt wird diese Anordnung dann auf den Hauptspeicher abgebildet (siehe Abschnitt 5.2.3). Dieses Verfahren wird später auf mehrere Cache-Ebenen und die Berücksichtigung von TLBs erweitert (Abschnitt 5.2.4 und 5.2.5).

Es werden Objekte auf Sets abgebildet. Die Abbildung $F \rightarrow C$ wird wie folgt definiert:

- Ist ein Objekt i größer als eine Cacheline, suche eine Folge von zusammenhängenden Cachelines für das Objekt i. Zusammenhängend sind alle Cachelines für die gilt (r ist die Nummer der Cacheline): $(r \ mod \ n \cdot l) \ mod \ n = const.$
- Ist ein Objekt kleiner als eine Cacheline, suche eine Cacheline, die schon ein oder mehrere Objekte enthält und noch ausreichend freien Platz bietet. Der benötigte freie Platz b errechnet sich als $b = align - (S \ mod \ align) + S$. Die Größe $align$ bezeichnet die nötige Ausrichtung (Alignment) eines Objekts an einer Speicheradresse. Der Wert von $align$ ist von der Architektur festgelegt. Ist keine solche Cacheline vorhanden, nimm eine unbenutzte Cacheline.
- Innerhalb einer Cacheline dürfen sich nur Objekte befinden, die dieselben Attribute haben. Dieses Attribut ist dann das Attribut A_j des Sets j, dem die Cacheline angehört. Innerhalb eines Sets dürfen nur Cachelines mit demselben Attribut sein.
- Ist nicht mehr genügend Speicherplatz im Cache frei, müssen Cachelines mehrfach belegt werden. Auch hier ist darauf zu achten, dass nur Objekte mit den korrekten Attributen verwendet werden.

Um eine unnötig hohe Speicherfragmentierung in Zonen mit unterschiedlichen Attributen zu vermeiden, sollte jeweils eine zusammenhängende Zone eines Attributs gebildet werden. Bild 5.1 zeigt eine mögliche Belegung eines 2-fach Set-assoziativen Caches nach diesem Schema. Der obere Teil des Caches ist mit Objekten belegt, die das Attribut *locked* haben. Danach folgen zwei Zonen mit den Attributen *read-only* und *read-write*. Diese Anordnung hat folgende Vorteile:

- Muss eine Cacheline verdrängt werden, ist a-priori bekannt, ob dafür mit einem Hauptspeicherzugriff zum Abgleich der Daten gerechnet werden muss. Bei Cachelines mit dem Attribut *read-only* ist das nicht der Fall, bei Cachelines mit dem Attribut *read-write* muss man damit rechnen.
- Mit Hilfe des Attributs *locked* ist es möglich, Objekte permanent im Cache zu halten. Solche Objekte werden nie verdrängt. Dies minimiert die WCET und erleichtert die statische WCET-Analyse.

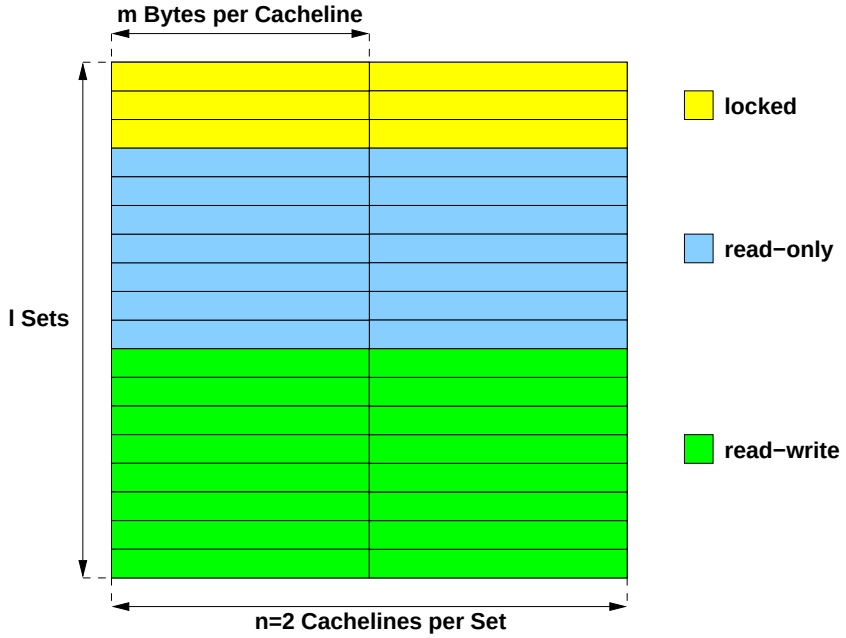

Bild 5.1: Beispiel für eine Anordnung im Cache

- Die Anzahl der Cachelines, die mit einem zusätzlichen Speicherzugriff verdrängt werden müssen, ist minimal. Das wird durch die Gruppierung der Objekte nach Attributen erreicht. Wären Objekte mit den Attributen *read-only* und *read-write* innerhalb einer Cacheline, kann eine Verdrängung, die durch das *read-only* Objekt verursacht wird, einen Speicherzugriff auslösen, weil das *read-write* Objekt mit verdrängt werden muss.

Aus der Sicht eines Realzeitsystems wird mit dieser Methodik zur Anordnung von Code und Daten vor allem die Prädizierbarkeit des Systems erhöht. Des Weiteren sinkt die Zahl der Verdrängungsvorgänge mit zusätzlichem Speicherzugriff auf ihr Minimum. Sie ist nur noch vom Code und den Eingangsdaten abhängig. Die Möglichkeit, Code und Daten permanent im Cache zu halten, erhöht nochmals sowohl die Vorhersagbarkeit als auch die Geschwindigkeit des Systems.

Fragmentierung

Ein Nachteil dieser Vorgehensweise ist, dass unter Umständen nicht der komplette Speicherplatz des Caches genutzt werden kann. Sobald ein Objekt einer Cacheline in einem ungenutzten Set zugewiesen wird, kann dieses Set nur noch mit Objekten mit demselben Attribut belegt werden. Gibt es nicht genug Objekte, um diesen Speicherplatz zu füllen, muss dieser ungenutzt bleiben. Bei x Zonen verschiedener Attribute errechnet sich der ungenutzte Speicherplatz im Worst-Case zu

$$S_{Cache,ungenutzt} = x \cdot (n \cdot m - S_{min}) \tag{5.4}$$

Der Parameter S_{min} bezeichnet dabei die minimale Objektgröße. Diese entspricht in der Regel der Datenwortbreite. Für ein 32 Bit System wären dies 4 Byte, für ein 64 Bit System 8 Byte. Im Best-Case kann der komplette Speicherplatz genutzt werden.

5.2.3 Abbildung in den Hauptspeicher

Die Anordnung von Code und Daten im Cache korrespondiert direkt mit den Positionen der Objekte im Hauptspeicher (siehe Abschnitt 3.3.2). Für die Umsetzung der gewünschten Anordnung im Cache muss die entsprechende Anordnung im Hauptspeicher realisiert werden.

Bild 5.2 zeigt diese Abbildung für einen Cache, der wie in Bild 5.1 belegt ist.

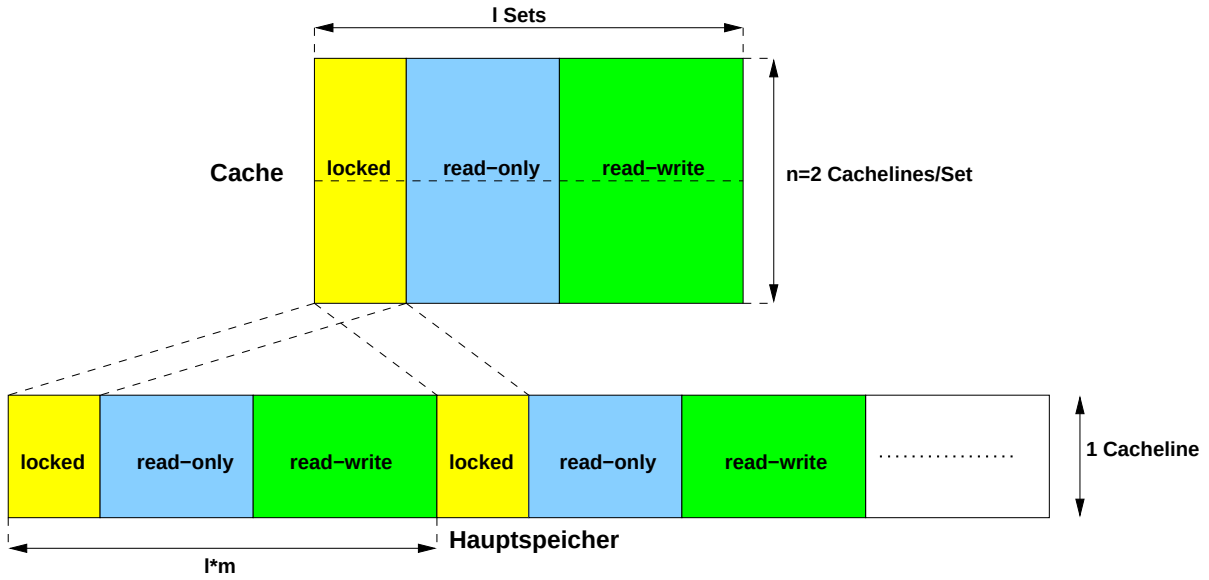

Bild 5.2: Abbildung des Caches in den Hauptspeicher

Die Aufteilung des Caches aus Bild 5.1 in drei Zonen mit jeweils einem Attribut muss auch im Hauptspeicher umgesetzt werden. In dem Beispiel aus Bild 5.2 wird eine Zone im Cache in $n = 2$ Zonen im Hauptspeicher aufgeteilt. Das liegt daran, dass über die physikalische Adresse eines Objekts nur das Set festgelegt wird (siehe auch Abschnitt 3.3.2). An welche Position im Set eine Cacheline platziert wird, ist für Software transparent und kann durch die Anordnung im Hauptspeicher nicht beeinflusst werden.

Die Abfolge der Zonen aus dem Cache im Hauptspeicher wiederholt sich im Abstand $l \cdot m$. Das heißt, wenn eine Zone $p\%$ des Caches belegt, müssen auch $p\%$ des Hauptspeichers für diese Zone reserviert werden. Dabei spielt es keine Rolle, ob soviel Speicherplatz wirklich benötigt wird.

Wieviel Speicher *tatsächlich* für eine Anwendung genutzt werden kann, hängt davon ab, wie sich die Objekte, aus denen die Anwendung besteht, auf die drei Attribute aufteilen lassen.

Hat man beispielsweise wenige Objekte mit dem Attribut *locked*, so können alle korrespondierenden Zonen im Hauptspeicher nicht von anderen Objekten genutzt werden. Haben die Objekte zusammen die Größe S_{locked}, so ergibt sich der zu reservierende Speicher zu

$$S_{HS,locked} = \left(n \cdot m \cdot \left\lceil \frac{S_{locked}}{n \cdot m} \right\rceil\right) \cdot \frac{S_{HS}}{l \cdot m} \tag{5.5}$$

Das heißt in diesem Fall, dass eine Speichergröße von

$$S_{Rest} = S_{HS,locked} - S_{locked} \tag{5.6}$$

für die Anwendung nicht nutzbar ist. Dies betrifft jeweils den kompletten Speicher, der von einem Cache genutzt wird. Dies ist ein wichtiger Aspekt für Multiprozessorsysteme und wird in Abschnitt 5.3.3 aufgegriffen.

Diese sich automatisch ergebende Aufteilung des Speichers in Zonen mit identischen Attributen hat auch zur Folge, dass man die Größe der Zonen mit *read-only* und *read-write* Attributen, die in der Praxis die meisten Objekte beinhalten, sorgfältig gegeneinander abwägen muss. Macht man eine Zone sehr groß, muss die andere entsprechend klein sein. Das hat zur Folge, dass alle Objekte, die in dieser kleinen Zone sein müssen, weit über den Hauptspeicher gestreut werden. Dies hat negative Folgen in Bezug auf die TLBs und wird in Abschnitt 5.2.4 näher betrachtet. Hat man mehr Objekte, die zur kleinen Zone gehören und weniger Objekte, die zur großen Zone gehören, steigt wiederum der ungenutzte Speicherplatz.

Details der Transformation einer Anordnung im Cache auf Adressen im Hauptspeicher werden in Abschnitt 5.2.9 diskutiert.

5.2.4 Berücksichtigung der TLBs

Die TLBs sind kleine Caches, die die nötigen Informationen für die Umrechnung einer virtuellen in eine physikalische Adresse zwischenspeichern. Ein Eintrag in den TLBs deckt immer einen gewissen Adressbereich (Seite) ab. Typische Werte sind 4 kB, 2 MB oder 4 MB. Wieviele Einträge für welche Seitengröße vorhanden sind, ist architekturabhängig und kann von Software nicht beeinflusst werden. Manche Architekturen erlauben es, zwei Seitengrößen parallel zu nutzen.

Die TLBs sind ähnlich aufgebaut wie die Caches: Es gibt einen TLB für Code und einen für Daten. Analog zum L2-Cache gibt es einen L2-TLB, der sowohl Einträge für Daten- als auch Einträge für Codeadressen aus den L1-TLBs aufnimmt. Im Gegensatz zum Cache sind die L1-TLBs in der Regel *voll-assoziativ* (siehe auch Abschnitt 3.3.2). Das heißt, jede Adresse im Speicher kann auf einen beliebigen Eintrag im TLB abgebildet werden. Details zur Adressumrechnung befinden sich in Anhang B.4. Bild 5.3 zeigt die Funktionsweise der TLBs.

Ist für eine Adressrechnung kein entsprechender Eintrag in den TLBs vorhanden, muss dieser aus dem Hauptspeicher geladen werden. Dies bedeutet einen oder mehrere zusätzliche Hauptspeicherzugriffe für die Umrechnung einer Adresse, unabhängig davon, ob es sich um Code oder Daten handelt.

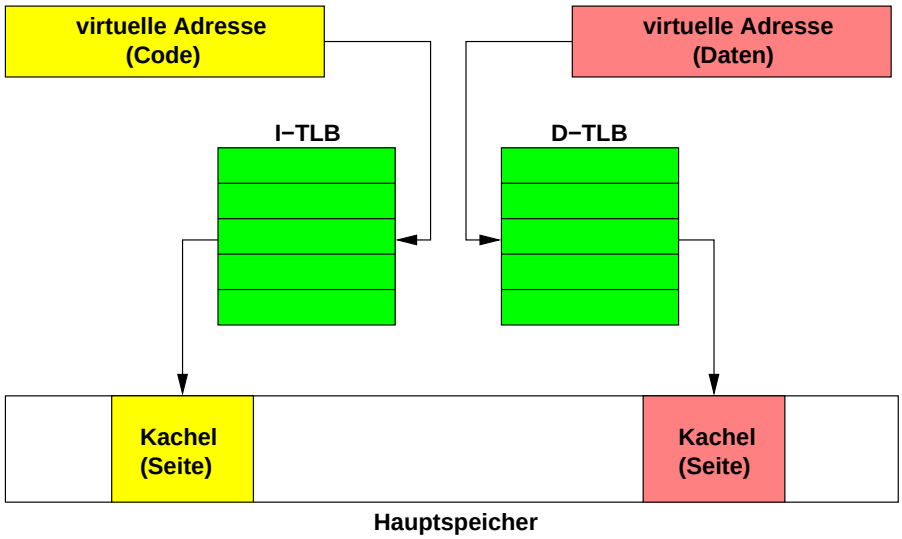

Bild 5.3: Adressumsetzung mit Hilfe von TLBs

Da einer Seite im virtuellen Adressraum eine gleichgroße Kachel im physikalischen Adressraum gegenüber steht, wird die Anzahl der TLB-Einträge, die eine Anwendung benötigt, von der Anordnung der Code- und Datenobjekte im Hauptspeicher beeinflusst. Verteilt man mehrere Objekte, die zu einer Anwendung gehören, auf einen großen Bereich im Hauptspeicher, werden auch mehrere Kacheln und damit TLB-Einträge zur Laufzeit benötigt. Das Ziel einer intelligenten Speicheranordnung muss es demnach sein, inhaltlich zusammengehörige Objekte im Speicher nahe zueinander zu positionieren. Bild 5.4 zeigt dieses Szenario.

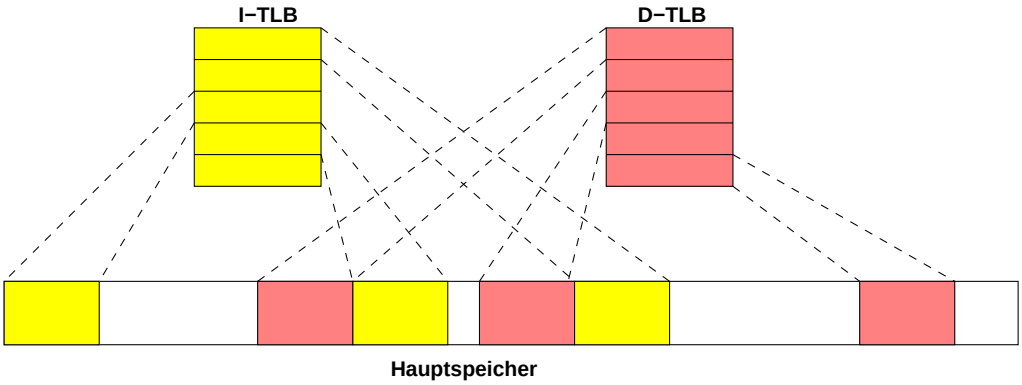

Bild 5.4: Einfluss der physikalischen Speicheranordnung auf die TLBs

Werden beispielsweise Datenobjekte, die von einem Codeobjekt referenziert werden, auf mehrere physikalische Kacheln verteilt, werden für die Referenzierung auch mehrere Einträge im Daten-TLB benötigt. Idealerweise liegen solche Objekte in einer Seite.

Erweiterung der Strategie zur Speicheranordnung

Es gibt zwei Möglichkeiten, die Nutzung der TLBs zu optimieren:

- **Lokale Optimierung:** Hier wird die Anzahl der benötigten TLB-Einträge pro Funktion minimiert. Es werden immer mindestens zwei Einträge benötigt (ein Eintrag für Code und einer für Daten). Durch eine kompakte Anordnung der Datenobjekte im Speicher, die zu einer Funktion gehören, wird die Anzahl an zusätzlich nötigen TLB-Einträgen minimiert.
- **Globale Optimierung:** Es wird die Anzahl an TLB-Einträgen, die für das gesamte System benötigt werden, minimiert. Bei dieser Art der Optimierung kann es passieren, dass die Anzahl der TLB-Einträge für einzelne Funktionen des Systems steigt.

Um die nötige Anzahl der TLB-Einträge für beide Arten der Optimierung zu berechnen, muss man die Aufteilung von Code und Daten in *Sections* vom Compiler mit berücksichtigen.

Der Compiler legt während des Kompiliervorgangs Objekte mit verschiedenen Eigenschaften in eigens dafür vorgesehenen Adressbereichen an, den sogenannten *Sections*. Diese Adressbereiche repräsentieren immer *einen* physikalisch zusammenhängenden Speicherbereich.

Im Allgemeinen gibt es einen Speicherabschnitt für Code (*.text section*) und einen für Daten (*.data section*). Nicht initialisierte Daten werden im sogenannten *.bss* Abschnitt angelegt. Es gibt noch weitere vordefinierte Abschnitte wie beispielsweise den *.rodata* Abschnitt, in den Datenobjekte abgelegt werden, die zum Zeitpunkt des Kompilierens einen fest zugewiesenen Wert haben und diesen zur Laufzeit der Software nicht mehr ändern. Objekte können immer nur innerhalb eines Abschnitts angeordnet werden. Die Startadressen der Abschnitte sind variabel. Es können auch zusätzliche Abschnitte vom Anwender definiert werden.

Die Anzahl der TLB-Einträge berechnet sich wie folgt: Sei M_i die Menge aller Q_i *Sections*, die für eine Funktion f_i nach Gleichung 5.1 benötigt werden:

$$M_i = \{m_{i,q_i} : q_i = 1 \dots Q_i\} \tag{5.7}$$

Alle Codeobjekte liegen in einem Adressbereich $m_{i,1}$ (*.text Section*). Die Datenobjekte D_i aus Gleichung 5.2 können über mehrere Adressbereiche m_{i,q_i} aufgeteilt sein. Um diese Abhängigkeit zu beschreiben, wird Gleichung 5.2 wie folgt erweitert:

$$D_i \;=\; \{A_{i,k_i}, S_{i,k_i}, m_{i,q_i} : k_i = 1...K_i,\ q_i \in [2; Q_i]\} \tag{5.8}$$

Die Anzahl an TLB-Einträgen hängt davon ab, auf wieviele Adressbereiche die Datenobjekte verteilt sind und wie groß diese Adressbereiche im Verhältnis zur Seitengröße p sind. Sei $S_{m_{i,q_i}}$ die Größe des Adressbereichs m_{i,q_i} in Bytes, der von einer Funktion benötigt wird. Die Anzahl der TLB-Einträge $n_{TLB,lokal}$ für die Funktion i ergibt sich dann zu

$$n_{TLB,lokal} = \left\lceil \frac{1}{p} \sum_{q_i=1}^{Q_i} S_{m_{i,q_i}} \right\rceil \tag{5.9}$$

Die Größe eines Adressbereichs für das gesamte System ergibt sich aus der Addition der Anteile, die die Objekte der einzelnen Funktionen belegen. Die Anzahl der TLB-Einträge für das gesamte System $n_{TLB,global}$ ergibt sich damit zu

$$n_{TLB,global} = \left\lceil \frac{1}{p} \sum_{i=1}^{N} \sum_{q_i=1}^{Q_i} S_{m_{i,q_i}} \right\rceil \tag{5.10}$$

Lokale Optimierung

Für die lokale Optimierung werden zunächst das Codeobjekt und dann die zugehörigen Datenobjekte angeordnet. Die Datenobjekte einer Funktion werden für beide Arten der Optimierung nach ihrer Größe sortiert angeordnet. Ist die Größe eines *TLB-Fensters* erreicht, werden die nächsten Cachelines derselben Sets belegt. Ein *TLB-Fenster* ist ein zusammenhängender Bereich im Cache, für dessen Abbildung auf den Hauptspeicher ein Eintrag in den TLBs benötigt wird.

Für das Beispiel des 2-fach Set assoziativen Caches mit 64 Byte Cachelinegröße und einer Seitengröße von $p = 4096$ Byte ist das *TLB-Fenster* 64 Cachelines groß. Das heißt, zuerst werden die ersten 64 Sets mit jeweils einer Cacheline belegt, danach dieselben Sets wiederum mit jeweils einer Cacheline. Erst danach werden die nächsten 64 Sets belegt.

Bei der funktionsweisen Anordnung der Code- und Datenobjekte kann es zu einer weiteren Fragmentierung (Zerstückelung) des Cache-Speichers innerhalb von Zonen mit demselben Attribut kommen (siehe auch Abschnitt 5.2.2). Diese tritt auf, wenn zwischen den einzelnen Blöcken der jeweiligen Funktionen Lücken auftreten, die später nicht gefüllt werden können. Solch eine Lücke kann dann nicht gefüllt werden, wenn sie kleiner als der Speicherbedarf der kleinsten anzuordnenden Objekte ist.

Zwischen den Blöcken gleichen Attributs zweier aufeinander folgender Funktionen können keine Lücken entstehen, weil alle Objekte eine Größe besitzen, die einem ganzzahligen Vielfachen der Speicherausrichtung (Alignment) entspricht. Es können sich nur Lücken zwischen Zonen verschiedener Attribute bilden oder zum Rand des Caches hin.

Benötigt man für ein Objekt für die Anordnung eine neue Cacheline, hat man prinzipiell die Möglichkeit, den Block in Richtung höherer Setnummern oder in Richtung niedrigerer Setnummern wachsen zu lassen. Die Entscheidung muss hier so getroffen werden, dass entweder eine vorhandene Lücke geschlossen wird oder keine neue Lücke entsteht.

Bild 5.5 zeigt dieses Szenario für zwei Zonen (beispielsweise *read-write* und *read-only*).

Soll jetzt beispielsweise ein Objekt zur Zone 1 hinzugefügt werden, so wird der Algorithmus sich entscheiden, zuerst, wenn möglich, die vorhandene Lücke zum Cacheanfang zu schließen. Ebenso würde mit einem Objekt verfahren, das zu Zone 2 hinzugefügt werden soll. Hier würde versucht werden, zuerst die Lücke zum Ende des Caches hin zu schließen.

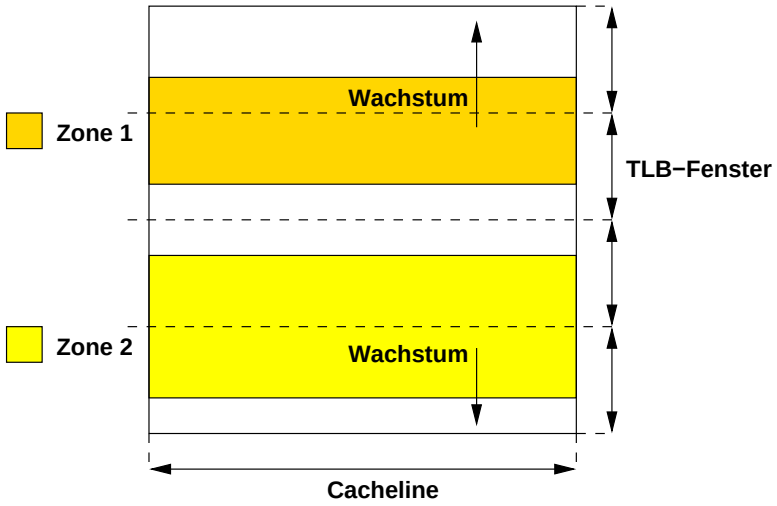

Bild 5.5: Fragmentierung während der Anordnung im Cache

Globale Optimierung

Bei der globalen Optimierung ist die Vorgehensweise etwas anders. Hier werden die Objekte nicht nach Funktionen angeordnet, sondern sie werden zunächst nach ihrer Größe sortiert (sowohl Code- als auch Datenobjekte) und dann der Reihe nach, beginnend mit dem jeweils größten Objekt, angeordnet. Diese Sortierung erstreckt sich über *alle* Objekte, ist also funktionsübergreifend.

Die Vorgehensweise der Anordnung bleibt dabei gleich: Zunächst werden die Objekte solange in neue Sets platziert, bis die Größenordnung eines TLB-Fensters erreicht ist. Dann werden die nächsten Cachelines innerhalb derselben Sets gewählt, wie bei der lokalen Optimierung auch. Durch die Sortierung der Objekte nach ihrer Größe ergibt sich eine kompaktere Anordnung als bei der lokalen Optimierung. Die Anzahl der benötigten TLB-Einträge kann nach Gleichung 5.10 ermittelt werden.

Lokale vs. globale Optimierung

Nun stellt sich die Frage, in welcher Situation eine lokale Optimierung und in welcher Situation eine globale Optimierung sinnvoll ist. Passt das gesamte System in den Cache des Prozessors, sind die beiden Optimierungen gleichwertig. Es existieren jeweils soviele Einträge in den TLBs, dass für alle Objekte im Cache ein Eintrag in den TLBs zur Verfügung steht. Ist der Platz im Cache nicht ausreichend, kommt es auch in den TLBs zu Verdrängungen.

Die lokale Optimierung ist sinnvoll, wenn mehrere Funktionen im Verhältnis deutlich öfter aufgerufen werden als andere und damit ebenso oft ihre TLB-Inhalte aktualisieren müssen. Dadurch sind die TLB-Misses bei einem erneuten Aufruf einer Funktion minimal. Das kann zwar auf Kosten zusätzlicher TLB-Einträge für das Gesamtsystem gehen, was aber trotzdem

sinnvoll sein kann, wenn nur wenige Funktionen die meiste Rechenzeit eines Systems für sich beanspruchen.

Eine globale Optimierung ist vorzuziehen, wenn die einzelnen Funktionen des Systems relativ gleichmäßig genutzt werden. Dann macht es keinen Sinn, einzelne Komponenten auf Kosten anderer zu optimieren.

Für eine globale Optimierung kann man auch die parallele Nutzung von zwei verschiedenen Seitengrößen in Betracht ziehen. Damit ist es möglich, für zwei unterschiedliche Speicherbereiche jeweils eine eigene Seiteneinteilung vorzunehmen. So können beispielsweise für einen Speicherbereich 4 kB Seiten und für einen anderen 4 MB große Seiten definiert werden. Das kann dann sinnvoll sein, wenn ein Speicherbereich größere zusammenhängende Objekte beinhaltet (z. B. RTOS). Auf diese Art und Weise kann man Seiten und damit TLB-Einträge sparen. Die TLBs haben separate Einträge für verschiedene Seitengrößen, das heißt, Bereiche mit unterschiedlichen Seitengrößen können sich in den TLBs nicht beeinflussen. Daher lässt sich ein solches Szenario auch mit den Gleichungen 5.9 und 5.10 bewältigen, wenn man für die unterschiedlichen Speicherbereiche jeweils die richtige Seitengröße p einsetzt.

5.2.5 Caches mit mehreren Ebenen

Bisher wurde für den Algorithmus zur Anordnung von Code und Daten von einer Cache-Ebene ausgegangen. Moderne Prozessoren haben jedoch meist mindestens zwei Cache Ebenen (siehe auch Kapitel 3.3). Wenn im L1-Cache eine Cacheline verdrängt werden muss, wird sie zunächst im L2-Cache abgelegt. Vom L2-Cache werden die Cachelines dann in den Hauptspeicher verdrängt (bei einer zweistufigen Cache Hierarchie). Wird eine Cacheline, die sich im L2-Cache befindet, wieder benötigt, wird sie von dort direkt in den L1-Cache geladen. Die Zugriffszeit ist dabei bei On-Chip Caches kaum länger als jene bei Zugriffen auf den L1-Cache.

Der L2-Cache ist in der Regel wesentlich größer als der L1-Cache (Faktor vier bis acht oder mehr). Betrachtet man die Anordnung in einer Cacheebene, heißt das für den Algorithmus, dass er von einem Cache mit höherer Assoziativität ausgehen kann. Bild 5.6 verdeutlicht dies am Beispiel eines 2-fach Set-assoziativen L1-Caches und eines 16-fach Set-assoziativen L2-Caches. Die Cacheline-Größe beträgt in beiden Caches m.

Angenommen, der L1-Cache hat l_1 Sets mit n_1 Cachelines pro Set, der L2-Cache analog dazu l_2 Sets mit n_2 Cachelines pro Set, wobei $n_1 < n_2$ gilt. Der L1-Cache habe die Größe S_1 in Bytes und der L2-Cache die Größe S_2 in Bytes. Die Anzahl der Sets im L2-Cache ergibt sich dann zu

$$l_2 = \frac{S_2}{n_2 \cdot m}$$

Allgemein bedeutet das, dass für ein Set im Cache aus Sicht des Algorithmus'

$$n' = \frac{n_2 \cdot l_2}{l_1} + n_1 = \frac{S_2}{l_1 \cdot m} + n_1 \tag{5.11}$$

Cachelines zur Verfügung stehen, wenn man den L2-Cache mit berücksichtigt. Für den Fall $l_1 = l_2$ steht die volle Assoziativität des L2-Caches zur Verfügung. Das gilt nur für *exklusive*

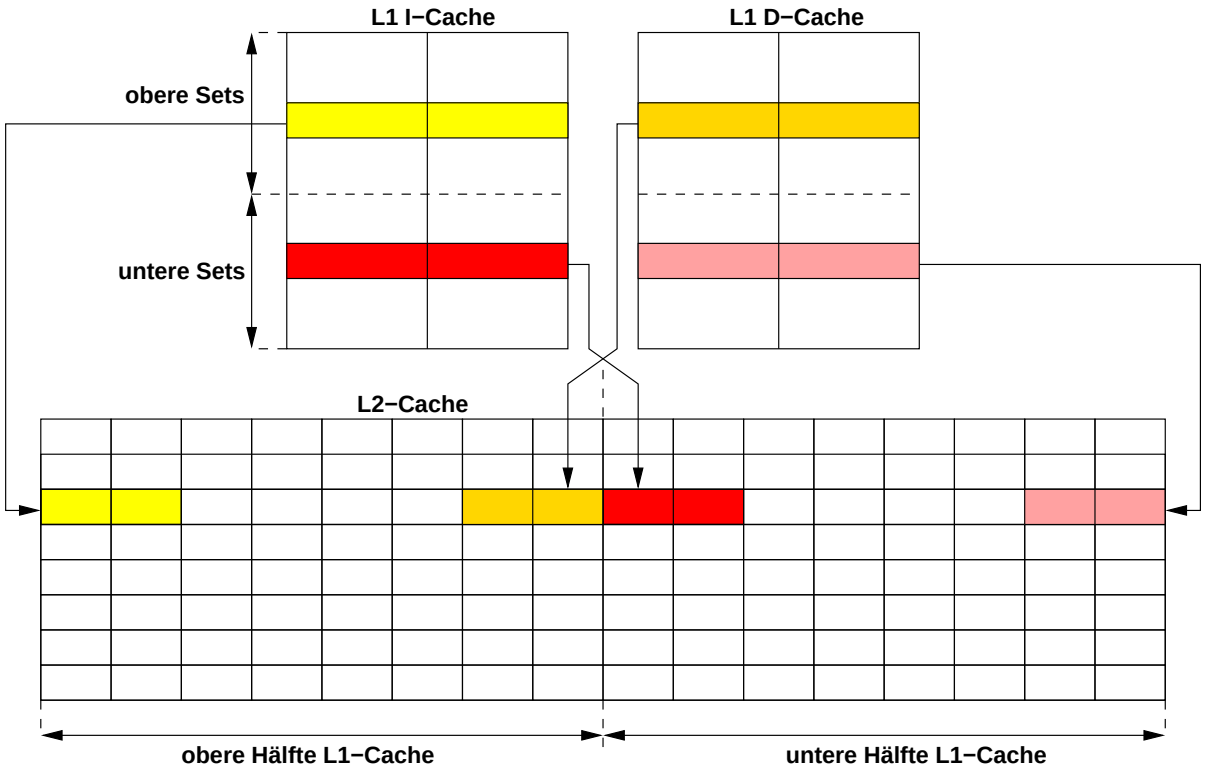

Bild 5.6: Verdrängung vom L1-Cache in den L2-Cache

Cachedesigns (siehe auch 3.3), bei einem *inklusiven* Cachedesign müsste man die Cachegröße S_2 um die Größe S_1 des L1-Caches reduzieren und daraus eine niedrigere zur Verfügung stehende Anzahl an Sets n' berechnen.

Die Größe n' gibt an, wievielfach ein Set belegt werden darf, bevor es zu einem Hauptspeicherzugriff kommt. Nimmt man als Beispiel $S_2 = 256$ kB und $l_1 = 512$ Sets im L1-Cache ($n_1 = 2$, $n_2 = 16$) mit einer Cacheline-Größe von $m = 64$ Byte, ergibt sich ein $n' = 10$. Das heißt, man kann den Cache für die Anordnung wie *einen* Cache mit der Assoziativität zehn betrachten. Der L2-Cache wird dann zu gleich großen Teilen für Instruktionen und Daten verwendet (acht Cachelines pro Set für den L1-I-Cache und acht für den L1-D-Cache).

Je nach Anwendung muss dies aber nicht die beste Lösung sein. Man kann auch mehr oder weniger Cachelines für Code oder Daten verwenden. Der folgende Abschnitt diskutiert verschiedene Szenarien.

5.2.6 Anwendungsszenarien

Der Algorithmus, wie er in Abschnitt 5.2.2 und 5.2.3 vorgestellt und in den Abschnitten 5.2.4 und 5.2.5 erweitert wurde, behandelt Code- und Datenobjekte mit gleicher Priorität. Je nach Anwendung muss das aber nicht der beste Weg sein. Prinzipiell kann man folgende Anwendungsszenarien unterscheiden:

1. Eine Task besteht in etwa aus gleich viel Code und Daten.
2. Eine Task besteht aus wenig Code und braucht große Datenmengen.
3. Das Gegenteil von Punkt 2: Die Task besteht aus viel Code und benötigt nur wenig Daten.

Das vorrangige Ziel besteht darin, die WCET für eine Task zu minimieren. Gleichzeitig soll die WCET möglichst sicher und wenig pessimistisch bestimmt werden können.

Betrachtet man ein Szenario wie in Fall 1, ist der Algorithmus, so wie er vorgestellt wurde, die beste Lösung. Es wird versucht, alle Objekte im Cache gleichermaßen anzuordnen. Es gibt nicht mehr Verdrängungsvorgänge als nötig und für jede Verdrängung ist bekannt, wie sie abläuft: Mit oder ohne zusätzlichen Hauptspeicherzugriff zum Abgleich der Daten. Dadurch wird die WCET – in Bezug auf die Prozessorarchitektur – minimiert und gleichzeitig so gut wie möglich vorhersagbar. Für den Fall, dass die Hauptspeicherzugriffe in Folge einer Verdrängung ungestört ablaufen können, ist dies die beste Lösung.

Anders sieht die Situation hingegen für Fall 2 aus. Wenn der Codeumfang klein genug ist, dass er in den L1-Cache passt, ist es sinnvoll, den *kompletten* L2-Cache für die Daten zu verwenden. Das heißt, die modellierte Assoziativität n' des Caches aus Gleichung 5.11 aus Abschnitt 5.2.5 muss dann folgendermaßen geändert werden:

$$\begin{aligned} n'_{code} &= n_1 \\ n'_{data} &= n_1 + n_2 \end{aligned} \tag{5.12}$$

Die Größe n'_{code} bezeichnet die Assoziativität, mit der die Codeobjekte angeordnet werden und n'_{data} die Assoziativität, mit der die Datenobjekte angeordnet werden. Somit wird ein kleiner Instruktionen-Cache modelliert (L1 Instruktionen-Cache) und ein großer Daten-Cache (L1 Daten-Cache und der L2-Cache). Analog kann man das für Fall 3 machen, indem man n'_{code} und n'_{data} aus Gleichung 5.12 vertauscht.

Indem man die Größen n'_{code} und n'_{data} variiert, kann man entweder mehr Platz für Code- oder mehr Platz für Datenobjekte vorsehen. Auf diese Art und Weise kann der Algorithmus sehr flexibel an unterschiedliche Anwendungsszenarien angepasst werden. Wichtig ist nur, dass die Summe $n'_{code} + n'_{data}$ nicht größer als $2 \cdot n'$ (Gleichung 5.11) wird. Ansonsten kommt es zu Verdrängungen.

5.2.7 Der Algorithmus als Pseudo-Code

Bild 5.7 zeigt die Vorgehensweise bei der Anordnung von Code und Daten für eine lokale Optimierung als Pseudo-Code. Es wird davon ausgegangen, dass die Daten aus Abschnitt 5.2.1 bereits vorliegen.

Für eine lokale Optimierung der TLB-Nutzung werden die Codeobjekte f_i nach ihren Abhängigkeiten sortiert. Das heißt, die Objekte, die voneinander abhängig sind, stehen im Feld `f` hintereinander. Die Abhängigkeiten werden von der Funktion `find_dependency` ermittelt, die jeweils das nächste Element liefert.

```
        *** Algorithmus zur Anordnung von Code und Daten ***

/* Codeobjekte f[i] nach Abhängigkeiten sortieren */

  f[0] = code_object[0];
  for(i=1; i<N; i++)
    f[i] = find_dependency(f[i-1]);

/* Alle Objekte anordnen */

  for(i=0; i<N; i++) {
    placed = FALSE;
    /* suche freien Platz und überprüfe Attribute */
    while(placed == FALSE) {
      p      = search_free_place(f[i],m,l,n'_code);
      placed = check_attributes(f[i],p);
    }
    place_item_in_memory(f[i],p);
    for(k=0; k<f[i].K; k++) {
      placed = FALSE;
      /* suche freien Platz und überprüfe Attribute */
      while(placed == FALSE) {
        p      = search_free_place(f[i].d[k],m,l,n'_data);
        placed = check_attributes(f[i].d[k],p);
      }
      place_item_in_memory(f[i].d[k],p);
    }
  }
```

Bild 5.7: Der Algorithmus als Pseudo-Code

Der Algorithmus arbeitet dann schrittweise alle Elemente f_i ab. Zunächst wird mit Hilfe der Funktion `search_free_place` der nächste freie Speicherplatz im Cache gesucht. Ist kein Speicherplatz mehr frei, wird damit begonnen, den Cache mehrfach zu belegen. Die Funktion `check_attributes` überprüft dann die Attribute der gefundenen Speicherposition. Stimmen diese mit dem Objekt überein, ist ein passender Platz gefunden (`placed=TRUE`), ansonsten muss eine neue Position im Cache gesucht werden.

Dieselbe Vorgehensweise wiederholt sich mit den Datenobjekten, die jedem Codeobjekt zugeordnet sind. Der Algorithmus endet, wenn allen Objekten eine korrekte Position im Speicher zugewiesen ist. Nach jeder gefundenen Position im Cache erfolgt die Abbildung auf physikalische Adressen im Speicher mit der Funktion `place_item_in_memory`.

Soll eine globale Optimierung angewendet werden, werden zunächst alle Code- und Datenobjekte ihrer Größe nach sortiert. Dann werden die Code- und Datenobjekte nach obigem Schema angeordnet.

Komplexität

Die Komplexität dieser Vorgehensweise lässt sich wie folgt abschätzen: Sei N die Anzahl an Codeobjekten und K_{max} die maximale Anzahl an Datenobjekten, die einem Codeobjekt zugeordnet sind. Die Ausführungszeit der Funktion `find_dependency` steigt linear mit der An-

zahl der Codeobjekte N. Die Ausführungszeit der Funktion `search_free_place` hängt von der Größe des Caches (Suchraum) und der Größe des jeweiligen Objekts ab. Im Worst-Case ergeben sich $\frac{S_{Cache}}{S_{min}}$ Versuche, bis eine freie Position gefunden wird. Dieser Faktor ist nur abhängig von der Architektur des Rechners. Die Funktion `check_attributes` zum Überprüfen der Attribute ist in ihrer Komplexität unabhängig von der Zahl der Objekte oder der Cachearchitektur. Dasgleiche gilt für die Funktion `place_item_in_memory`: Im Worst-Case benötigt diese Funktion $\frac{S_{HS}}{S_{Cache}} \cdot n$ Versuche, um eine entsprechende Position im Speicher zu finden. Demnach ergibt sich für die Ordnung $\mathcal{O}$

$$\begin{aligned} \mathcal{O} &= N + N\Big(\frac{S_{Cache}}{S_{min}} + \frac{S_{HS}}{S_{Cache}} \cdot n + K_{max}(\frac{S_{Cache}}{S_{min}} + \frac{S_{HS}}{S_{Cache}} \cdot n)\Big) \\ \mathcal{O} &\sim N(K_{max} + 2) \end{aligned} \tag{5.13}$$

Die Komplexität steigt demnach mit dem Produkt aus der Anzahl der Codeobjekte und der maximalen Anzahl der Datenobjekte pro Codeobjekt. Da ein Realzeitsystem meist aus einer überschaubaren Menge an Code und Datenobjekten besteht, ist diese Komplexität für den praktischen Einsatz kein Hindernis.

5.2.8 Ein Anwendungsbeispiel

Die Funktionsweise des Algorithmus zur Anordnung von Code und Daten im Hauptspeicher soll an einem einfachen Beispiel gezeigt werden. Es wird ein kleines Realzeitsystem betrachtet, das aus folgenden Softwarekomponenten besteht:

- **Echtzeitscheduler:** Diese Funktion (rt_schedule) realisiert den Schedulingmechanismus des Realzeitbetriebssystems. Sie wird entweder über einen Timer-Interrupt aufgerufen oder direkt durch eine Betriebssystemfunktion (rt_task_wait_period).
- **Zwei Tasks:** Diese beiden Tasks repräsentieren eine einfache Applikation (task0, task1).
- **Interrupt Service Routine:** Diese wird vom RTOS dazu genutzt, auf alle Interrupts zu reagieren (dispatch_global_irq) und dann gegebenenfalls eine vom Anwender installierte Routine aufzurufen.

Tabelle 5.1 zeigt alle Funktionen und deren Parameter (Attribut A, Größe S in Bytes).

Funktionen f_i	A_i	S_i	Datenobjekte D_k	A_{i,k_i}	S_{i,k_i}
task0	ro	357	a	rw	8192
			flags, meas_ctrl	rw	4
task1	ro	357	b	rw	8192
			flags, meas_ctrl	rw	4
rt_task_wait_period	ro	807	rthal[1)]	rw	72
			locked_cpu	rw	4
			rt_smp_current	rw	8
			rt_smp_time_h	rw	16

Funktionen f_i	A_i	S_i	Datenobjekte D_k	A_{i,k_i}	S_{i,k_i}
			rt_smp_linux_task	rw	1920
sortA	ro	807	-	-	-
sortB	ro	807	-	-	-
rt_schedule	ro	807	rthal[1)]	rw	72
			sched_rqsted, rt_smp_current	rw	8
			locked_cpus, scheduling_cpus	rw	4
			rt_smp_times	rw	64
			rt_smp_oneshot_running, rt_smp_half_tick	rw	8
			rt_smp_time_h	rw	16
			rt_smp_linux_task	rw	1920
			rt_smp_shot_fired, rt_smp_fpu_task	rw	8
			boot_cpu_data	rw	192
			switch_time, rt_smp_linux_cr0	rw	16
			tuned	rw	28
rt_get_time	ro	50	rthal[1)]	rw	72
			rt_smp_oneshot_timer	rw	8
			rt_smp_times	rw	64
rt_switch_to_linux	ro	41	global	rw	44
			processor	rw	2592
rt_switch_to_real_time	ro	94	rthal[1)]	rw	72
			processor	rw	2592
			global	rw	44
dispatch_global_irq	ro	408	Stack	rw	36
			rthal[1)]	rw	72
			processor	rw	2592
			global_irq	rw	384
			internal_ic_ack_irq, global_vector, ic_ack_irq	rw	128
			linux_isr	rw	1024

Tabelle 5.1: Parameter des Beispielszenarios

Die Größen der einzelnen Objekte und deren Beziehungen untereinander wurden durch eine Analyse des Objectcodes ermittelt. Das Attribut der Codeobjekte ist immer *read-only (ro)*. Die Datenobjekte sind in diesem Fall alle mit dem Attribut *read-write (rw)* versehen. Die Anordnung soll für einen 2-fach Set assoziativen Cache mit 64 Byte Cacheline Größe und 64 kB Instruktionen- und Daten-Cache gezeigt werden. Bild 5.8 zeigt den Kontrollflussgraphen (CFG) der Funktionen.

Für jedes Objekt muss ein ganzzahliges Vielfaches der Speicherausrichtung (Alignment) als Speicherplatz reserviert werden. Dadurch wird sichergestellt, dass jedes Objekt eine korrekte Ausrichtung im Speicher erhält. Auf einer 32 Bit Architektur beträgt diese Ausrichtungsgrenze

[1)] Variable im Betriebssystemkern, feste Position im Speicher

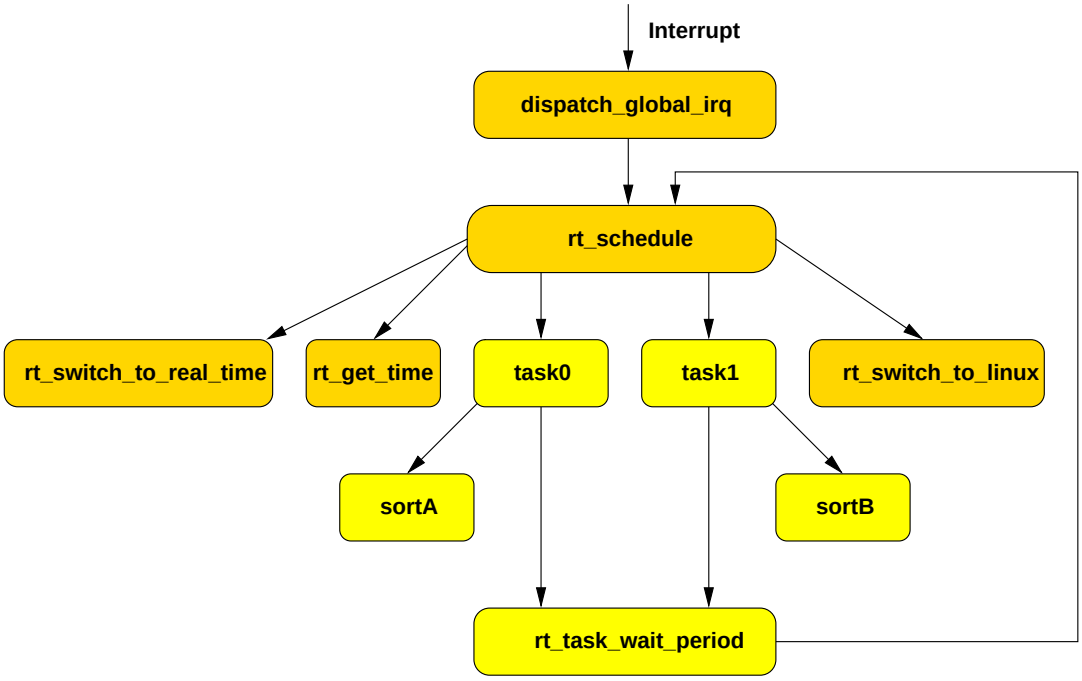

Bild 5.8: Kontrollflussgraph des Szenarios aus Tabelle 5.1

in der Regel vier, demnach müssen beispielsweise für das Codeobjekt *rt_switch_to_real_time* 96 Byte Speicherplatz (anstatt 94 Byte) reserviert werden. Im Folgenden wird die Anordnung für eine lokale Optimierung gezeigt.

Zuerst wird der Code der Funktion *dispatch_global_irq* platziert (erste Funktion im Kontrollflussgraph). Anschließend werden die Daten dieser Funktion angeordnet. Zunächst gibt es das Objekt *Stack* welches den Stackbereich widerspiegelt, der von der Funktion *dispatch_global_irq* benötigt wird. Die Variable *rthal* ist Bestandteil des laufenden Betriebssystem-Kernels und kann zur Laufzeit nicht im Speicher verschoben werden. Die Position dieses Objekts im Cache ist demnach vorgegeben. Alle anderen, in der Position variablen Objekte, müssen entsprechend drumherum platziert werden. Diese Variablen werden nacheinander möglichst dicht zueinander platziert.

Tabelle 5.2 zeigt das Ergebnis der Anordnung, wenn für *rthal* eine Speicherposition angenommen wird, die auf die Cachelines 5-6 abgebildet wird. Begonnen wird mit *rthal*, um von dort ausgehend einen möglichst kompakten Block mit Objekten desselben Attributs zu bilden.

Durch Objekte mit fest vorgegebener Position ergeben sich Startpunkte für Speicherbereiche mit denselben Attributen. Daher platziert der Algorithmus immer von solchen Fixpunkten ausgehend die zur Verfügung stehenden Objekte. Die Größe der einzelnen Blöcke ist vor Beginn der Anordnung bekannt. Sie ergibt sich aus den Größen der Teilobjekte (siehe auch Abschnitt 5.2.3).

Die Objekte im Daten-Cache werden jeweils in ein neues Set platziert, damit sie möglichst wenige Einträge in den TLBs benötigen. Im vorliegenden Beispiel deckt ein Eintrag in den TLBs

Objekt	belegter Platz	Cache[2)]	Set(s)
dispatch_global_irq	**408**	**I**	**0-6**
Stack	36	D	0
rthal	72	D	5-6
internal_ic_ack_irq	128	D	3-4
global_vector	128	D	1-2
ic_ack_irq	128	D	67-69
processor	2592	D	6-46
global_irq	384	D	61-67
linux_isr	1024	D	46-61

Tabelle 5.2: Anordnung für dispatch_global_irq

bei einer Seitengröße $p = 4096$ Byte einen Bereich von 64 Cachelines ab. Die Datenobjekte der nächsten Funktion können dann in die jeweils zweiten Cachelines der jeweiligen Sets platziert werden.

Sind alle Datenobjekte des ersten Codeobjekts angeordnet, wird das nächste Codeobjekt im Kontrollfluss genommen. In diesem Beispiel ist das die Funktion *rt_schedule* (siehe Bild 5.8). Auch hier wird zunächst das Codeobjekt und dann die zugehörigen Datenobjekte angeordnet.

Die Anordnung der restlichen Objekte aus Tabelle 5.1 zeigt Tabelle 5.3.

Objekt	belegter Platz	Cache	Set(s)
rt_schedule	**808**	**I**	**6-17**
sched_rqsted, rt_smp_current	jeweils 8	D	4
locked_cpus, scheduling_cpus	jeweils 4	D	5
rt_smp_oneshot_running, rt_smp_half_tick	jeweils 8	D	5
rt_smp_shot_fired, rt_smp_fpu_task	jeweils 8	D	5
rt_smp_times	64	D	3
rt_smp_time_h	16	D	4
switch_time, rt_smp_linux_cr0	jeweils 16	D	4
boot_cpu_data	192	D	0-2
tuned	28	D	0
rt_get_time	**52**	**I**	**18**
rt_smp_oneshot_timer	8	D	5
rt_switch_to_linux	**44**	**I**	**18-19**
rt_switch_to_real_time	**96**	**I**	**19-20**
global	44	D	5-6
task0	**360**	**I**	**21-27**
a	8192	D	6-134
flags, meas_ctrl	jeweils 4	D	69
task1	**360**	**I**	**27-33**

[2)] I: Instruktionen Cache D: Daten Cache

Objekt	belegter Platz	Cache	Set(s)
b	8192	D	69-197
rt_task_wait_period	**808**	**I**	**33-45**
locked_cpu	4	D	164
rt_smp_time_h	16	D	164
rt_smp_linux_task	1920	D	134-164
sortA	**808**	**I**	**45-58**
sortB	**808**	**I**	**58-71**

Tabelle 5.3: Anordnung des Szenarios aus Tabelle 5.1

5.2.9 Algorithmus zur Abbildung einer Anordnung in den Speicher

Die Vorgehensweise bei der Abbildung einer Anordnung im Cache in den Speicher soll an dem Beispiel aus Abschnitt 5.2.8 gezeigt werden. Die Grundlagen zur Abbildung einer Anordnung im Cache in den Hauptspeicher wurden bereits in Abschnitt 5.2.3 erläutert.

```
#define ALIGNMENT 4 /* 32 Bit , 8 für 64 Bit Architektur */

a = offset + section_offset + startSet*m;
while(isFree(a,size) == FALSE) {
  a += l*m;
  if(a > hs_size) { print('Out of memory!'); exit(1); }
}

bool isFree(address,size) {
  if(size < m) {
    r = 0;
    while((memFree(address,size) == FALSE) &&
          (r <= m-(address % m))) {
      r += ALIGNMENT; address += ALIGNMENT;
    }
  }
  return(memFree(address,size));
}
```

Bild 5.9: Algorithmus zur Berechnung von Hauptspeicheradressen

Die Berechnung der Speicheradressen a für jedes Objekt erfolgt mit dem Algorithmus, der in Bild 5.9 als Pseudocode dargestellt ist. Die Variable `offset` bezeichnet die Startadresse im Speicher, ab der das Realzeitsystem abgelegt werden soll. Die Funktion `memFree` prüft, ob ein Speicherbereich, der durch die Startadresse `address` und die Größe `size` bestimmt ist, frei ist. Die Variable `startSet` beinhaltet die Setnummer, ab der das Objekt im Cache abgelegt werden soll. l ist die Anzahl an Sets im Cache, m die Größe einer Cacheline in Bytes.

Zunächst wird überprüft, ob die Startadresse verfügbar ist. Das ist der Fall, wenn für das Objekt ab dieser Adresse genügend freier zusammenhängender Speicherplatz zur Verfügung steht. Ist

das nicht der Fall, wird die nächste passende Speicherstelle überprüft. Diese ergibt sich, indem zu der vorherigen Adresse $l \cdot m$ Bytes hinzu addiert werden (siehe auch Abschnitt 5.2.3).

Ist ein Objekt kleiner als eine Cacheline, muss für jede mögliche Position innerhalb einer Cacheline überprüft werden, ob für dieses Objekt genug freier Speicherplatz zur Verfügung steht. Dazu werden innerhalb einer Cacheline alle Positionen im Abstand `ALIGNMENT` überprüft.

Die Ergebnisse der Speicheranordnung für das Beispiel aus Tabelle 5.1 in Abschnitt 5.2.8 sind in Tabelle 5.4 zusammengestellt. Es wird hier ein *.text*, ein *.data* und ein *.bss* Abschnitt benötigt. Welches Objekt zu welchem Abschnitt gehört, kann durch ein Analyse des Objectcodes ermittelt werden. Die entsprechenden Angaben sind in Tabelle 5.4 enthalten.

Objekt	Section	Startadresse
dispatch_global_irq	**.text**	**0x8000000**
Stack	.bss	0x8020000
rthal	.data	0x8010140
internal_ic_ack_irq	.data	0x8010040
global_vector	.data	0x80100c0
ic_ack_irq	.data	0x8011128
processor	.data	0x8010188
global_irq	.data	0x8010fa8
linux_isr	.data	0x8010ba8
rt_schedule	**.text**	**0x8000198**
sched_rqsted	.data	0x8018150
rt_smp_current	.data	0x8018158
locked_cpus	.data	0x8018160
scheduling_cpus	.data	0x8018164
rt_smp_oneshot_running	.data	0x8018140
rt_smp_half_tick	.data	0x8018148
rt_smp_shot_fired	.data	0x8018130
rt_smp_fpu_task	.data	0x8018138
rt_smp_times	.data	0x80180c0
rt_smp_time_h	.data	0x8018100
switch_time	.data	0x8018110
rt_smp_linux_cr0	.data	0x8018120
boot_cpu_data	.data	0x8018000
tuned	.data	0x8010000
rt_get_time	**.text**	**0x80004c0**
rt_smp_oneshot_timer	.data	0x8018168
rt_switch_to_linux	**.text**	**0x80004f4**
rt_switch_to_real_time	**.text**	**0x8000520**
global	.data	0x8018170
task0	**.text**	**0x8000580**
a	.bss	0x8020024

Objekt	Section	Startadresse
flags	.data	0x801811a8
meas_ctrl	.data	0x801811ac
task1	**.text**	**0x80006e8**
b	.bss	0x8022024
rt_task_wait_period	**.text**	**0x8000850**
locked_cpu	.data	0x8018219c
rt_smp_time_h	.data	0x8018291c
rt_smp_linux_task	.data	0x8018292c
sortA	**.text**	**0x8000b78**
sortB	**.text**	**0x8000ea8**

Tabelle 5.4: Anordnung des Szenarios aus Tabelle 5.1 im Speicher

Das Realzeitsystem soll bei `offset`= $0x8000000$ (128 MB) im Speicher beginnen. Der *.text* Abschnitt beginnt dann ebenfalls bei $0x8000000$ (`section_offset` = 0), der *.data* Abschnitt bei $0x8010000$ (`section_offset` = 0x10000) und der *.bss* Abschnitt bei $0x8020000$ (`section_offset` = 0x20000). Ein Abschnitt muss mindestens die Größe $l \cdot m$ haben oder ein ganzzahliges Vielfaches davon. Bild 5.10 zeigt die Anordnung der Objekte im Speicher in schematischer Darstellung.

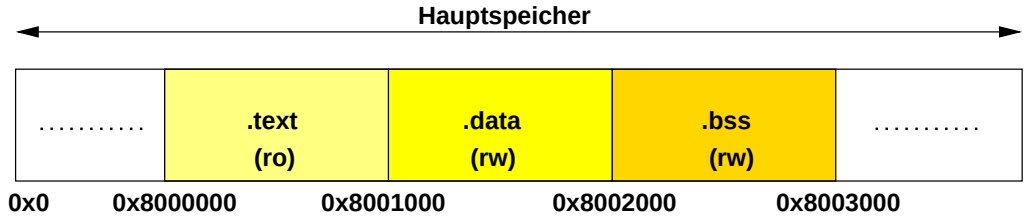

Bild 5.10: Anordnung der Objekte aus Tabelle 5.4

Die *read-only (ro)*-Zone fällt mit dem *.text* Bereich zusammen. Die *read-write (rw)*-Zone erstreckt sich über den *.data* und den *.bss*-Bereich. Die Lücken, die durch diese Aufteilung in zwei Bereiche entstehen, könnten von weiteren Objekten gefüllt werden. Für den Code, der sich vollständig im Abschnitt *.text* befindet, werden zwei Einträge in den TLBs benötigt. Für die Daten werden fünf Einträge für den *.bss* Abschnitt benötigt und für die Daten im *.data* Abschnitt ebenfalls fünf Einträge. An diesem Beispiel sieht man, dass dies nicht der minimalen Anzahl an TLB-Einträgen für Daten entspricht, die nach Gleichung 5.9 neun hätte betragen müssen.

Wären ein oder mehrere Objekte mit dem Attribut *locked* versehen, so dürften die zugehörigen *Sets* kein weiteres Mal vergeben werden. Möchte man beispielsweise die Funktion *rt_schedule* dauerhaft im Cache haben, so wären 13 Sets nicht mehr für andere Objekte verfügbar. Im vorliegenden Beispiel wären dies 2,5% des Caches. Von diesen 2,5% (entspricht 1664 Byte) wären 856 Byte ungenutzt (51,44%).

5.3 Speicherverwaltung

In diesem Abschnitt wird zunächst allgemein auf die Einflüsse der Hauptspeicherbelegung auf die WCET von Software eingegangen. Basierend darauf wird im Anschluss die Anwendung von PC-Architekturen als Realzeitsysteme diskutiert. In diesem Zusammenhang wird auch das Prinzip bereits existierender Echtzeiterweiterungen für Systeme dieser Art kurz erläutert. Es wird die Frage diskutiert, welche Vor- und Nachteile Uni- und Multiprozessorsysteme im Einsatz als Realzeitsysteme bieten. Danach werden weiterführende Methoden zur Verbesserung der Echtzeiteigenschaften vorgestellt.

Zunächst wird eine Aufteilung des Hauptspeichers in einen Bereich für das GPOS und einen Bereich für das RTOS vorgeschlagen. Aufbauend darauf wird die Zuteilung von Speicherbereichen für Tasks beider Systeme diskutiert und deren Auswirkung auf den Algorithmus aus Abschnitt 5.2 besprochen. Es folgt eine Betrachtung von Mechanismen zur Interprozesskommunikation in Echtzeit, sowohl zwischen RT-Tasks als auch zwischen RT- und GPOS-Tasks. In diesem Zusammenhang wird auch auf die Möglichkeit eingegangen, Cache-Snooping gezielt für den Datenaustausch zwischen Tasks auf zwei Prozessoren einzusetzen. Der Abschnitt schließt mit Betrachtungen über einen Realzeitnachweis für PC-Systeme.

5.3.1 Einflüsse der Speicherbelegung auf die WCET

Die Anordnung von Code und Daten im Speicher korrespondiert direkt mit der Belegung der Caches. Das gilt insbesondere auch für Speicherregionen, die für den Datenaustausch zwischen Tasks genutzt werden. Sollen die Daten zwischen Tasks ausgetauscht werden, die auf unterschiedlichen Prozessoren ausgeführt werden, kommt das Problem der Kohärenz der Caches hinzu. Die Kohärenz der Caches wird durch das Cache-Snooping Protokoll (siehe Abschnitt 4.2.5) sichergestellt. Das kann dazu führen, dass eine Aktion auf einem Prozessor eine Verzögerung auf einem anderen Prozessor verursacht.

Die Aufgabe der Speicherverwaltung ist es, allen Komponenten eines Systems einen geeigneten Speicherplatz zuzuweisen. Ein Algorithmus, der dies im Hinblick auf die Nutzung der Caches tut, wurde in Abschnitt 5.2 vorgestellt. Dieses Verfahren muss von der globalen Speicherverwaltung mitgeteilt bekommen, in welcher Adressregion im Speicher die einzelnen Komponenten abgelegt werden können. Diese dürfen nicht mit anderen Komponenten des Systems kollidieren.

Mit Hilfe einer geeigneten Speicherverwaltung kann der Speicher in Regionen unterteilt werden, die von unterschiedlichen Prozessoren genutzt werden (siehe Abschnitt 5.3.3). Die Anwendung unterschiedlicher Caching-Strategien, wie sie in Abschnitt 5.3.4 vorgestellt wird, ist ebenso Aufgabe einer Speicherverwaltung. Die Zuteilung von Speicherregionen erfolgt in der Initialisierungsphase des Realzeitsystems. Dynamische Änderungen sind nicht vorgesehen.

Eine ungünstige Speicherbelegung kann zu unnötigen und meist nur schwer kontrollierbaren Cache-Misses führen, welche wiederum zu Hauptspeicherzugriffen führen können. Diese dauern sehr viel länger als ein Zugriff im Cache und sie sind in ihrer Dauer abhängig von anderen parallel ablaufenden Aktivitäten im System.

5.3.2 Ein PC als Realzeitsystem

Aufgrund ihrer hohen Rechenleistung und ihrer Flexibilität möchte man PC-Systeme auch als Echtzeitsysteme nutzen können. Prinzipiell sind diese Systeme aber nicht für dieses Einsatzgebiet konzipiert. Das Optimierungskriterium ist die durchschnittliche Rechenleistung, nicht die Minimierung selten auftretender, maximaler Laufzeiten (WCETs).

Der grundlegende Gedanke vieler Forschungsgruppen auf diesem Gebiet ist, die Anwendung in einen Echtzeit- und einen Nicht-Echtzeitanteil aufzuspalten. Beide Teile sollen auf einem Rechner bearbeitet werden können. Dazu wird ein Standardbetriebssystem für die Nicht-Echtzeitanteile verwendet und ein Realzeitbetriebssystem für den Echtzeitanteil.

Standard- und Realzeitbetriebssysteme

Standardbetriebssysteme wie Linux verfügen über eine breite Hardwareunterstützung von Standardkomponenten im PC-Bereich. Die Integration neuer Hardware erfolgt sehr schnell, ebenso wie die Weiterentwicklung vorhandener Anwendungssoftware. Dies gilt insbesondere auch für Tools zur Softwareentwicklung für verschiedene Plattformen. Diese Standardbetriebssysteme sind aber nicht geeignet für Realzeitsysteme, da sie unabdingbare Eigenschaften eines Realzeitbetriebssystems nicht erfüllen:

- Die verwendeten Scheduler sind nicht echtzeitfähig. Das Ziel dieser Scheduler ist es, die zur Verfügung stehende Rechenleistung möglichst gerecht auf die Prozesse im System zu verteilen. Dabei steht eine hohe Reaktivität von Prozessen, die mit dem Benutzer interagieren, im Vordergrund („gefühlte Rechenleistung").
- Der Anwender hat keine Möglichkeit, das Sperren und Freigeben von Interrupts durch das Betriebssystem zu kontrollieren. Daher ist es nicht möglich, die Ausführungszeit von Tasks zu bestimmen, da diese jederzeit von beliebigen Interrupts unterbrochen werden können.
- Viele Systemaufrufe, beispielsweise zur Interprozesskommunikation, sind blockierend, das heißt, ihre Ausführungszeit ist abhängig vom aktuellen Systemzustand. Diese Blockierungszeiten können sehr lang werden und sind im Allgemeinen nicht abschätzbar.

Ein Ansatz, um einen PC als Realzeitsystem einzusetzen, besteht darin, ein Standardbetriebssystem um Echtzeitfähigkeiten zu erweitern. Damit wird es möglich, sowohl Standard- als auch Realzeitapplikationen auf demselben Rechner zu betreiben.

Echtzeiterweiterungen für Linux

Für die Untersuchungen in dieser Arbeit wurde die Echtzeiterweiterung RTAI (Real Time Application Interface [24]) für Linux verwendet. Das Konzept von RTAI besteht darin, die Interruptverwaltung von Linux zu übernehmen und die Realzeitapplikationen mit Hilfe eines

Echtzeitschedulers zu verwalten. Es werden geeignete Mechanismen zur Interprozesskommunikation bereitgestellt. Die Realzeit-Tasks werden im Kernelspace des Prozessors ausgeführt und haben alle Rechte für einen direkten Zugriff auf die Hardware.

Linux wird als niederpriorste Realzeit-Task angesehen und bekommt vom Echtzeitscheduler entsprechend Rechenzeit zugewiesen. Eine Linux-Task kann nie eine Realzeit-Task unterbrechen. Die Linux-Tasks müssen sich die Rechenzeit, die Linux von RTAI zugewiesen bekommt, teilen. Sie werden nach wie vor vom Linux-Scheduler verwaltet. Die Rechenzeit für den Realzeitanteil geht demnach auf Kosten der Rechenzeit von Linux. Bild 5.11 zeigt die Systemstruktur von RTAI.

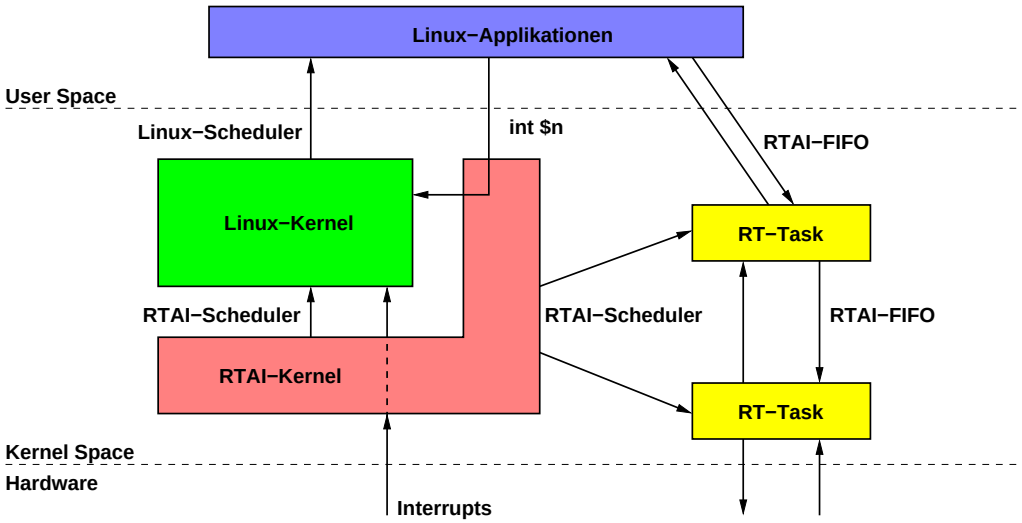

Bild 5.11: Die Systemstruktur von RTAI

Mit Hilfe von RTAI ist es möglich, Standard- und Realzeitapplikationen auf demselben Rechner zu bearbeiten. Die Trennung in einen Echtzeit- und einen Nicht-Echtzeitanteil ist für viele praktische Anwendungen sinnvoll. Oft müssen nur Teile eines technischen Prozesses unter harten Echtzeitbedingungen ausgeführt werden, wie zum Beispiel eine Datenerfassung oder die Berechnung eines neuen Stellwerts innerhalb eines Regelkreises. Aufgaben wie die Visualisierung von Prozess-Daten müssen hingegen nicht in harter Echtzeit durchgeführt werden.

RTAI überlässt die Speicherverwaltung, ebenso wie ähnliche Ansätze wie beispielsweise RT-Linux, komplett Linux. Der Speicher für die Realzeit-Tasks oder für einzelne Komponenten des RTOS, wie den Echtzeitscheduler, wird dynamisch von Linux vergeben. Es ist keine Möglichkeit vorgesehen, bestimmte Komponenten an bestimmte Speicherplätze zu binden. Es kann somit auch vorkommen, dass ein und dasselbe System bei einem Neustart an ganz anderen Adressen im Speicher abgelegt wird. Dadurch können sich die WCETs der Tasks und damit das Laufzeitverhalten des Systems ändern.

Uni- und Multiprozessorsysteme

PC-Systeme gibt es als Einprozessor- wie auch als Mehrprozessormaschinen (in der Regel Dual-Systeme). Beide Varianten werden sowohl von Standard- als auch von Realzeitbetriebssystemen unterstützt. Daher stellt sich die grundlegende Frage, ob eher die Ein- oder die Mehrprozessorvariante für einen Realzeitbetrieb geeignet ist. Um diese Frage zu beantworten, muss man die Entstehung von WCETs auf PC-Systemen genauer betrachten.

Die WCET hängt im Allgemeinen von der Codestruktur und den Eingangsdaten einer Realzeit-Task ab, sowie von der Hardware, auf der diese Task ausgeführt wird. Die Hardwareabhängigkeiten kann man in Abhängigkeiten von der Prozessorarchitektur und Abhängigkeiten von der Peripherieanbindung unterteilen. Der Einfluss der Prozessorarchitektur wurde ausführlich in Kapitel 4 behandelt. Er wird geprägt durch die Caches des Prozessors und durch die Speicheranbindung.

Der Aufbau eines Mehrprozessorsystems ist nahezu identisch zu dem eines Uniprozessorsystems (siehe Bild 3.2 auf Seite 17). Bei SMP-Systemen kommen nur weitere identische Prozessoren hinzu, die alle gleichberechtigt über den Host-Bus an den Chipsatz angeschlossen werden. Bei NUMA-Systemen werden alle weiteren Prozessoren über HyperTransport-Links mit den vorhandenen Prozessoren verknüpft, wobei jeder Prozessor einen eigenen physikalischen Speicher hat.

Alle CPUs müssen sich die Speicheranbindung mit allen PCI-Geräten teilen. Das heißt, die Zugriffszeiten auf den Speicher hängen zum einen von der Aktivität der Peripheriegeräte ab, und zum anderen von der Konkurrenzsituation der CPUs untereinander. Die Anzahl der Speicherzugriffe, die eine CPU während der Ausführung einer Realzeit-Task durchführen muss, hängt wiederum von der Softwarestruktur und den Eingangsdaten ab.

Muss nun eine CPU alle Software ausführen, müssen sich auch alle Tasks den Cache teilen. Das ist insbesondere dann von Bedeutung, wenn auf dem Rechner sowohl ein Standard- als auch ein Realzeitbetriebssystem ausgeführt werden soll. Dann kann es passieren, dass eine GPOS-Task Cacheinhalte einer RTOS-Task verdrängt, wodurch sich deren Laufzeit verlängern würde. Hinzu kommt ein Overhead für den ständigen Wechsel zwischen den Betriebssystemen. Interrupts, die nur für das Standardbetriebssystem von Bedeutung sind, müssen auch während des Echtzeitbetriebs empfangen werden und können so die Laufzeit von Realzeit-Tasks verlängern.

Es ist daher von Vorteil, ein Mehrprozessorsystem einzusetzen. Ein Prozessor kann exklusiv für das Standardbetriebssystem genutzt werden und die anderen Prozessoren exklusiv für Realzeitaufgaben. Dies hat den Vorteil, dass das Standardbetriebssystem die Realzeit-Tasks bezüglich der Prozessorarchitektur nicht beeinflussen kann (keine direkten Cacheverdrängungen). Interrupts können so verteilt werden, dass jeder Prozessor nur die Interrupts bekommt, die er auch bearbeiten soll.

Bezüglich der Einflüsse bei den Speicherzugriffszeiten unterscheiden sich Uni- und Mehrprozessorsysteme nur durch die zusätzlichen Prozessoren, die auf den Speicher zugreifen müssen. Die dadurch resultierenden Laufzeitverlängerungen können im Worst-Case pro Speicherzugriff bis zu 100% bei SMP-Systemen betragen (siehe Abschnitt 4.3). Diesem Nachteil steht jedoch

die deutliche Trennung von GPOS und RTOS und damit eine geringere Anzahl an Speicherzugriffen entgegen. Auch unerwünschte Unterbrechungen durch Interrupts können vermieden werden.

Für Applikationen, die aufgrund ihrer Struktur die Caches gut nutzen können oder die sogar ganz in den Cache passen, sind Multiprozessorsysteme die bessere Lösung. Sehr speicherintensive Anwendungen hingegen können von den Caches nicht profitieren. Solche Anwendungen benötigen vor allem kurze und deterministische Speicherzugriffszeiten und können daher gut auf einem Uniprozessorsystem ausgeführt werden. Hat man sowohl speicherintensive Anwendungen als auch solche, die die Caches gut nutzen können, ist wiederum eine Multiprozessorlösung die bessere Variante: Ein Prozessor kann die speicherintensive Anwendung ausführen und einer (oder mehrere) die restlichen Anwendungen.

Es hängt von der konkreten Anwendung ab, welche Variante die geeignetere für einen Betrieb als Echtzeitsystem ist. Im Allgemeinen überwiegen jedoch die Vorteile eines Multiprozessorsystems. Dies gilt insbesondere für Systeme mit NUMA-Architektur (siehe Abschnitt 4.3.2), bei denen sich die Prozessoren weit weniger gegenseitig beeinflussen als bei SMP-Systemen.

5.3.3 Speicherbereiche für GPOS und RTOS

Um ein Multiprozessorsystem sinnvoll für Echtzeitanwendungen einsetzen zu können, werden sowohl dem GPOS als auch dem RTOS eigene Bereiche des Hauptspeichers zugewiesen. Damit werden beide Systeme in Bezug auf die Prozessorarchitektur voneinander entkoppelt. Es gibt keine gegenseitigen Verdrängungen mehr in den Caches oder den TLBs. Die beiden Bereiche werden jeweils einem oder mehreren Prozessoren zugewiesen. Bild 5.12 zeigt eine mögliche Aufteilung für ein System mit vier Prozessoren.

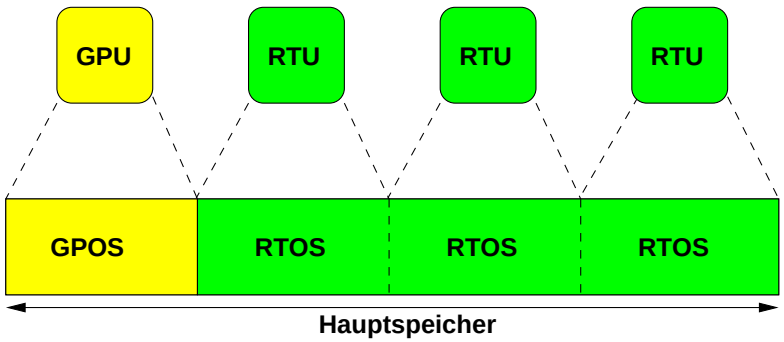

Bild 5.12: Aufteilung des Hauptspeichers unter vier Prozessoren

Ein Prozessor bearbeitet entweder nur GPOS-Code oder nur RTOS-Code. Dazu zählt auch jeweils der Code entsprechender Anwendungen. Ein Prozessor, der nur GPOS-Code bearbeitet, wird als *General Purpose Unit* (GPU) bezeichnet, ein Prozessor, der nur RTOS-Code bearbeitet, als *Real-Time Unit* (RTU).

Eine Aufteilung des Speichers wie in Bild 5.12 sorgt dafür, dass Standard-Tasks keine Cachelines oder TLB-Einträge von Realzeit-Tasks verdrängen können. Dasselbe gilt auch für Realzeit-Tasks, die auf unterschiedlichen RTUs ausgeführt werden. Dadurch werden unerwünschte Interaktionen vermieden, die die Laufzeit von Realzeit-Tasks verlängern können. Auch durch das Cache-Snooping Protokoll sind keine Verzögerungen zu erwarten, solange jeder Prozessor nur auf Daten innerhalb seines eigenen Speicherbereichs zugreift. Dieser Aspekt wird in Abschnitt 5.3.4 aufgegriffen, in dem es um den Datenaustausch zwischen Tasks geht.

Diese Trennung der Speicherbereiche kann durch eine entsprechende Anordnung von Code und Daten des Gesamtsystems erreicht werden. Dies lässt sich leicht mit dem Algorithmus aus Abschnitt 5.2 verbinden, indem man bei der Abbildung einer Anordnung im Cache (siehe Abschnitt 5.2.9) den Anfang des gewünschten Adressbereichs als `offset` angibt und den Parameter `hs_size` auf die Größe des Adressbereichs setzt. Dazu muss vor der Anordnung festgelegt sein, welche Komponenten des Systems auf welchem Prozessor ausgeführt werden sollen. Mit diesem Aspekt beschäftigt sich Abschnitt 5.4.

Die Größen der einzelnen Adressbereiche müssen nicht gleich sein, so wie es in Bild 5.12 gezeigt ist. Die Größe kann je nach Anwendung variieren. Auch die Aufteilung, wieviele Prozessoren als GPU und wieviele als RTU verwendet werden, ist prinzipiell beliebig. Als minimale Speichergröße pro Prozessor kann die Größe des Caches angesehen werden, da es keinen Sinn macht, einem Prozessor weniger Speicher zuzuteilen, als er im Cache speichern kann.

SMP- und NUMA-Systeme

In SMP-Systemen besitzt jeder Prozessor die gleichen Rechte. Jeder Prozessor darf mit derselben Priorität auf den gesamten Speicherbereich zugreifen. Neben Blockierungszeiten auf dem gemeinsam benutzten Bus zum Speicher kann es dabei jederzeit zu gegenseitigen Verdrängungen in den Caches und den TLBs der Prozessoren kommen. Vor allem die automatische Synchronisierung der Caches durch das Cache-Snooping Protokoll (siehe Abschnitt 4.2.5) kann zu unkontrollierbaren Schwankungen bei der Ausführungszeit der Software führen.

Wenn jeder Prozessor nur noch auf einen bestimmten Adressbereich zugreifen muss (dies entspricht einer Trennung wie in Bild 5.12), können solche Verzögerungseffekte vermieden werden. Voraussetzung dafür ist, dass die Bereichsgrenzen für die unterschiedlichen Adressbereiche auf ganzzahlige Vielfache der Cachegröße gelegt werden. Die Blockierungen bei konkurrierenden Speicherzugriffen auf dem Bus werden durch diese Maßnahme nicht beeinflusst.

Bei NUMA-Systemen (siehe Bild 4.14) hat jeder Prozessor seinen eigenen physikalischen Speicherbereich, auf den er schnell zugreifen kann. Jeder Prozessor kann auf den lokalen Speicher eines anderen Prozessors zugreifen, so dass es bei einem gleichzeitigen Zugriff mehrerer Prozessoren auf einen lokalen Speicherbereich wiederum zu Verzögerungen kommen kann. Um diese zu umgehen ist es sinnvoll, die Adressbereiche der einzelnen Prozessoren so festzulegen, dass sie auf den jeweiligen lokalen Speicherbereich zeigen. Gegenüber einem SMP-System hat man dann den Vorteil, dass es bei Speicherzugriffen zu keinen Verzögerungen durch konkurrierende Zugriffe anderer Prozessoren kommen kann.

5.3.4 Interprozesskommunikation in Echtzeit

Interprozesskommunikation (IPC) bezeichnet im Allgemeinen den Austausch von Daten zwischen zwei oder mehreren Teilnehmern (Tasks, ISRs). Gängige Realisierungen des Datenaustauschs sind ein allen Teilnehmern zugänglicher Speicherbereich (shared memory), FIFO-Mechanismen oder ein Mailboxsystem. Alle diese Methoden benötigen bestimmte Speicherbereiche zur Zwischenspeicherung der Daten.

In Abschnitt 5.3.3 wurde der Speicherbereich zwischen den Prozessoren des Systems aufgeteilt, um beispielsweise Interaktionen durch das Cache-Snooping Protokoll vorzubeugen. Zur Realisierung von IPC-Mechanismen ist es nötig, dieses Konzept um spezielle Speicherbereiche zu erweitern. Bild 5.13 zeigt eine mögliche Anordnung für vier Prozessoren. Die Speicherbereiche

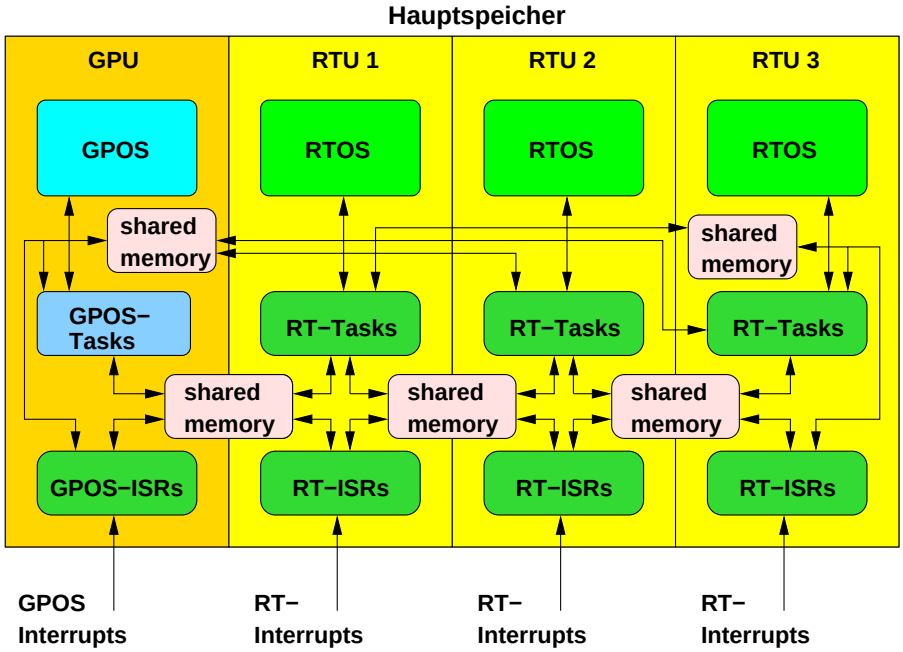

Bild 5.13: Speicheraufteilung mit IPC-Bereichen

für die IPC sind in Bild 5.13 als *shared memory* gekennzeichnet. Alle IPC-Mechanismen, wie FIFOs oder Mailboxsysteme, benötigen solch einen Bereich. Um eine Antwort auf die Frage zu finden, wo und wie diese Speicherbereiche angelegt werden sollen, muss man einen Blick auf die verschiedenen Caching-Strategien des Prozessors werfen.

Verschiedenen Speicherbereichen kann eine bestimmte Caching-Strategie zugewiesen werden, welche angibt, wie das Caching bei einem Zugriff (lesen oder schreiben) erfolgt. Die meisten modernen Prozessoren kennen folgende Strategien:

Strong Uncacheable: Zugriffe auf diesen Speicherbereich gehen am Cache vorbei direkt in den Hauptspeicher bzw. zum Prozessor. Adressbereiche, die als *strong uncacheable* markiert sind, verändern den Inhalt der Caches nicht und jeder Zugriff resultiert in einem Hauptspeicherzugriff.

Write Combining: Im Prinzip dasselbe wie *strong uncacheable*. Der einzige Unterschied ist, dass Schreibzugriffe durch einen *Write Buffer* gebündelt und in einer anderen Reihenfolge ausgeführt werden können. Der Cache ist davon nicht betroffen.

Write-Through: Es werden sowohl Schreib- als auch Lesezugriffe über den Cache abgewickelt. Jeder Schreibzugriff wird sowohl im Cache als auch in dem korrespondierenden Speicherbereich durchgeführt.

Write-Back: Dasselbe wie *Write-Through* mit dem Unterschied, dass Schreibzugriffe *nur* im Cache durchgeführt werden. Wird eine Cacheline verdrängt, deren Inhalt im Cache verändert wurde, muss sie in den Speicher zurückgeschrieben werden.

Write Protected: Es werden nur Lesezugriffe über den Cache abgewickelt. Schreibzugriffe gehen direkt über den Bus.

In Standardbetriebssystemen wird meist *Write-Back* als Strategie für den gesamten Speicherbereich gewählt, der für Anwendungen zur Verfügung steht. Mit Hilfe dieser Strategie lässt sich der Cache in der Regel effektiv nutzen. Die *Strong Uncacheable* Strategie ist für *memory mapped I/O* Bereiche gedacht, wo der Einsatz von Caches keinen Sinn macht. *Write Combining* ist sinnvoll für Framebuffer Geräte, bei denen es darauf ankommt, dass Schreibzugriffe den Graphikspeicher schnell aktualisieren.

Die Interprozesskommunikation soll möglichst schnell ablaufen und dabei andere Komponenten des Realzeitsystems so wenig wie möglich beeinflussen. Da die Daten im Speicher abgelegt werden müssen, können sie prinzipiell die Caches und TLBs der beteiligten Prozessoren beeinträchtigen. Man muss zwischen einem Datenaustausch von Tasks auf einem Prozessor und Tasks auf unterschiedlichen Prozessoren unterscheiden.

Für eine Kommunikation zwischen Prozessoren macht es grundsätzlich für den schreibenden Prozessor keinen Sinn, die Daten im Cache zu speichern (eine Ausnahme kann das MOESI-Protokoll sein, siehe Abschnitt 5.3.5). Diese Daten werden nur von anderen Prozessoren benötigt und sollten so schnell wie möglich im Speicher ankommen. Für den lesenden Prozessor kann es sinnvoll sein, die gelesenen Daten im Cache vorrätig zu haben, wenn öfters auf dieses Datum zugegriffen werden muss und sich die Daten zwischendurch nicht ändern.

Sollen Speicherbereiche für den Datenaustausch im Cache gehalten werden, müssen diese bei einer Anordnung mit dem Algorithmus aus Abschnitt 5.2 mit berücksichtigt werden. Ändern sich die Daten häufig, bringt eine Speicherung in den Caches keinen Vorteil. Dies gilt sowohl für die IPC auf einem Prozessor als auch für mehrere Prozessoren.

Es gibt nun mehrere Möglichkeiten, wie man Adressbereiche zum Datenaustausch konfigurieren kann. Prinzipiell muss man unterscheiden zwischen einem Datenaustausch zwischen Realzeit-Tasks und einem Datenaustausch zwischen Realzeit- und Standard-Tasks.

Datenaustausch mit dem GPOS

Die Kommunikation mit Tasks des Standardbetriebssystems spielt eine große Rolle, wenn man eine Applikation in einen Standard- und einen Realzeitanteil aufteilt. Dies kann beispielsweise sinnvoll sein, wenn der Echtzeitanteil periodisch Daten liefert, die eine Standardapplikation visualisieren soll. Oder in der umgekehrten Richtung, wenn ein Benutzer Daten über ein GUI (Graphical User Interface) eingibt, die eine Realzeit-Task parametrieren sollen.

Der Speicherbereich für diesen Datenaustausch sollte so liegen, dass er nur im Adressbereich der GPU oder der beteiligten RTU liegt. Sollen Daten mit Prozessen von mehreren RTUs ausgetauscht werden ist es besser, für jedes GPU-RTU Paar einen eigenen Speicherbereich zu definieren. Je nach dem, ob die Caching-Strategie in beide Richtungen der Kommunikation dieselbe sein soll oder nicht, muss man einen eigenen Speicherbereich für jede Kommunikationsrichtung vorsehen. Als Caching-Strategien bieten sich entweder *Strong Uncacheable* oder *Write Protected* an.

Teilen sich mehrere RTUs und die GPU einen Speicherbereich für wechselseitige Kommunikation, kann es zu Interaktionen des Cache-Snooping Mechanismus' kommen, falls mehrere CPUs Teile dieses Bereichs in ihren Caches haben. Insbesondere kann es dann zu Verdrängungen im Cache einer RTU kommen, die durch eine Aktion auf der GPU verursacht wurden. Dieses Szenario sollte unbedingt vermieden werden.

Datenaustausch zwischen Realzeitapplikationen

Die Kommunikation zwischen Realzeit-Tasks kann entweder mit Tasks auf demselben Prozessor oder mit Tasks auf einem anderen Prozessor stattfinden. Sind alle Teilnehmer auf einem Prozessor, ist die schnellste Methode ein Austausch direkt im Cache des Prozessors. Dieser Speicherbereich muss bei der Anordnung des Gesamtsystems mit berücksichtigt werden. Das macht im Allgemeinen jedoch nur Sinn, wenn der benötigte Speicherbereich nicht zu groß wird.

Müssen größere Mengen an Daten transferiert werden ist es besser, diese im Speicher abzulegen. Dasselbe gilt für den Fall, dass die Geschwindigkeit der Kommunikation keine große Rolle spielt. Dann ist es günstiger, den Cache für andere Komponenten des Systems zu nutzen. Solange nur ein Prozessor auf einen Bereich im Speicher zugreift, sind die Zugriffszeiten auf den Hauptspeicher relativ gut vorhersagbar, insbesondere bei NUMA-Systemen.

Werden die Kommunikationspartner auf unterschiedlichen Prozessoren ausgeführt, gelten die gleichen Überlegungen wie für den Datenaustausch zwischen GPU und RTU. Eine Aktion der einen RTU sollte nicht die Ausführungszeiten der anderen RTU beeinflussen.

Der nächste Abschnitt beschäftigt sich mit einer besonderen Form des Datenaustauschs zwischen zwei Prozessoren. Dabei werden die Eigenschaften des Cache-Snooping Protokolls ausgenutzt. Dies ermöglicht den schnellen Austausch kleiner Datenmengen direkt über die Caches der Prozessoren.

5.3.5 Datenaustausch mit Hilfe des Cache-Snooping Protokolls

Man kann das Cache-Snooping, wenn es mit Hilfe des MOESI-Protokolls implementiert ist, als besonders schnellen und effektiven Weg für den Datenaustausch zwischen Tasks auf zwei unterschiedlichen Prozessoren nutzen. Um dies zu verstehen, muss man sich die Details der Implementierung der Cache-Snooping Techniken genauer ansehen.

Es gibt zum einen die MESI-Technik. Die Abkürzung steht für Modified, Exclusive, Shared und Invalid. Dies sind die Bezeichnungen für verschiedene Zustände, in der sich eine Cacheline befinden kann:

Modified: Der Inhalt der Cacheline ist gültig, aber die korrespondierende Speicherstelle ist ungültig, das heißt, sie enthält andere Daten. Wird die Cacheline verdrängt, müssen die Daten im Speicher aktualisiert werden.

Exclusive: Die Cacheline ist gültig, die dazugehörigen Daten im Speicher auch. Kein anderer Prozessor hat diese Cacheline in seinem Cache.

Shared: Die Cacheline ist gültig, ebenso die Daten im Speicher. Die Cacheline kann auch in den Caches anderer Prozessoren sein.

Invalid: Die Cacheline ist ungültig.

Das MESI-Protokoll wird von den Prozessoren der Intel-Pentium Familie verwendet. Die Prozessoren der AMD-Athlon Familie und die AMD-Opteron Prozessoren erweitern das MESI-Protokoll zum MOESI-Protokoll. Der neue Zustand *Owned* ist mit einer geschickteren Umsetzung des Datenabgleichs zwischen verschiedenen Prozessoren verbunden.

Der Zustand *Owned* bedeutet, dass die Cacheline auch in den Caches anderer Prozessoren vorkommen kann, und dass die Daten im Speicher ungültig sein können. Alle anderen Prozessoren müssen diese Cacheline als *Shared* kennzeichnen. Nur die Cacheline im *Owned* Zustand enthält die aktuell gültigen Daten. Möchte nun ein Prozessor, der diese Cacheline als *Shared* hält, auf diese Daten zugreifen, bekommt er die aktuelle Cacheline direkt von dem Prozessor, der diese Cacheline als *Owned* besitzt. Es ist kein Umweg über den Speicher notwendig wie bei MESI.

Parallel zu dieser Aktion können andere Prozessoren oder Peripheriegeräte frei auf den Speicher zugreifen. Bild 5.14 verdeutlicht diesen Vorgang.

Jeder Prozessor verfügt über drei Punkt-zu-Punkt Verbindungen mit dem System Controller:

processor-to-system Bus 1 : Dieser Bus wird dazu verwendet, Anfragen oder Kommandos an das System zu senden (z.B. Speicher lesen oder schreiben). Dieser Bus wird nur vom Prozessor kontrolliert.

system-to-processor Bus 2 : Auf diesem Bus signalisiert der System Controller Kommandos und Anfragen an den Prozessor. Dieser Bus wird nur vom System Controller kontrolliert.

Data Bus 3 : Dies ist ein bidirektionaler Bus, auf dem die Daten transportiert werden. Daten werden *nur* auf diesem Bus transportiert.

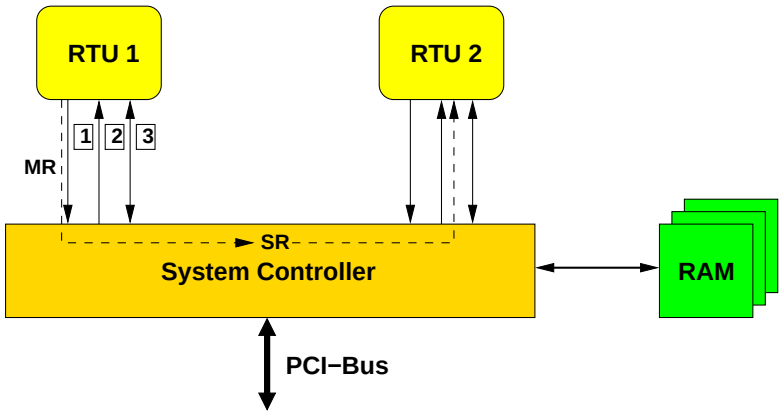

Bild 5.14: Cache-Snooping mit dem MOESI-Protokoll (nach [4])

Die drei Bussysteme können parallel zueinander arbeiten. Die Datenübertragung erfolgt paketorientiert, wobei jedes Paket mit einer ID versehen ist, mit deren Hilfe die Daten zu den entsprechenden Kommandos zugeordnet werden können. Damit ist es auch möglich, Befehle asynchron zu bearbeiten und so Wartezeiten anderer Befehle zu nutzen (*split transactions*).

Führt RTU 1 nun einen *Memory Request* (MR) durch, wird dieser zunächst vom System Controller in einen *Snoop Request* (SR) übersetzt. Auf diesen SR muss RTU 2 antworten und die aktuelle Cacheline direkt an RTU 1 senden. Dazu ist keine Interaktion mit dem Memory Controller notwendig. Ein Peripheriegerät könnte parallel dazu beispielsweise Daten in den Speicher schreiben. Teilt sich ein Prozessor die Daten mit einem Peripheriegerät, würde das Snooping genauso funktionieren.

Im Vergleich zum MESI-Protokoll, was ein ständiges Aktualisieren des Speichers voraussetzt (siehe auch Abschnitt 4.2.5), arbeitet das MOESI-Protokoll sehr effektiv. Dies kann dazu genutzt werden, einen schnellen Datenaustausch zwischen zwei Tasks auf zwei unterschiedlichen Prozessoren zu realisieren. Hat man einen gemeinsamen Speicherbereich für diese Daten, kann man ihn mit dem Algorithmus aus Abschnitt 5.2 im Speicher anordnen und dann die Daten dort ablegen. Die Aktualisierung der Caches erledigt das Cache-Snooping Protokoll automatisch und sehr schnell (siehe Abschnitt 6.2). Weitere Details zur Funktionsweise des Cache-Snooping Protokolls finden sich in [4],[5], [36] und [40].

5.4 Taskverteilung auf mehrere Prozessoren

In diesem Abschnitt geht es um das Problem, auf welchen Prozessoren einzelne Tasks und ISRs ausgeführt werden sollen. Diese Problemstellung soll in Bezug auf die Cache- und Speichernutzung des Systems untersucht werden. Grundsätzlich gibt es zwei Ansätze: Eine statische und eine dynamische Taskverteilung. Bei der statischen Verteilung wird von vornherein festgelegt, welcher Prozessor welche Task oder ISR bearbeitet. Diese Zuordnung ändert sich nicht zur Laufzeit des Systems. Bei einer dynamischen Verteilung wird zur Laufzeit entschieden, welcher

Prozessor welche Task oder ISR bearbeiten soll. Diese Entscheidung wird jedesmal neu getroffen, wenn eine Rechenzeitanforderung an das System gestellt wird. Entscheidungskriterium ist hier meist die Lastverteilung innerhalb des Systems, das heißt, der Prozessor, der momentan die meiste freie Rechenkapazität hat, muss die Task oder ISR ausführen.

Dynamische Taskzuteilung

Der Algorithmus aus Abschnitt 5.2 setzt ein statisches Szenario voraus. Die Anordnung aller Code- und Datenobjekte muss vor dem Start des Systems durchgeführt werden. Ändert sich etwas an dem Szenario, indem etwa eine weitere Task hinzukommt, kann sich dies auf die komplette Anordnung auswirken. Das heißt, diese neue Task muss nicht nur hinzugefügt werden, sondern die Anordnung aller anderen Objekte kann sich ebenfalls ändern. Eine optimale Anordnung in einem sich dynamisch ändernden System ist ohne Neuberechnung der Anordnung im Speicher nicht möglich. Diese würde nicht nur eine Unterbrechung zur Laufzeit erfordern, sondern auch einen erneuten Realzeitnachweis des Systems, da sich die WCETs einzelner Tasks ändern können. Die einzige Ausnahme wäre, Tasks mit gleichartigen Objekten (das heißt, gleiche Anzahl, Größe und Attribute) gegeneinander auszutauschen. Dies ist aber ein sehr unwahrscheinliches Szenario.

Ein weiterer Aspekt in einem dynamischen Szenario ist die Kommunikationsstruktur der Tasks. Wie in Abschnitt 5.3.4 gezeigt, spielt es für den Datenaustausch zweier Tasks eine Rolle, auf welchen Prozessoren diese Tasks ausgeführt werden. Da sich dies in einem dynamischen Szenario ständig ändern würde, müssten auch die Speicherbereiche zur Interprozesskommunikation neu angeordnet werden oder unter Umständen sogar neue Speicherbereiche angelegt werden. Dieser Aufwand ist zur Laufzeit praktisch nicht durchführbar.

Möchte man ein dynamisches System realisieren, könnte man höchstens von vornherein festlegen, welche Tasks auf welchen Prozessoren ausgeführt werden dürfen. Dann kann man die Anordnung so gestalten, dass man für jeden Prozessor mit allen potentiell dort laufenden Tasks rechnet. Dann erhält man ein Worst-Case Szenario, das im laufenden Betrieb nie eintreten wird, da ein und dieselbe Task nicht gleichzeitig auf allen Prozessoren laufen wird, wo sie laufen dürfte. Dieses Szenario kann sehr unrealistisch sein, abhängig von der Anzahl und dem Umfang der Tasks, die mehrfach berücksichtigt werden müssen. Dadurch muss man mit sehr pessimistischen WCETs rechnen und die Auslastbarkeit des Systems sinkt. Kennt man mögliche Abhängigkeiten der Tasks, zum Beispiel, dass Task A und Task B nie gleichzeitig im System aktiv sein können, kann man die Abschätzung der nötigen Rechenkapazitäten verbessern.

Statische Taskzuteilung

Die Möglichkeiten zur Anordnung eines Systems im Speicher und damit in den Caches der Prozessoren kann auch Anhaltspunkte für eine Zuteilung der Tasks liefern. Je nach dem, welche Tasks man auf welchem Prozessor bearbeiten möchte, ergibt sich eine andere Anzahl an zu erwartenden Cache-Misses.

Ein Ansatz zur statischen Taskzuteilung besteht nun darin, die Anzahl an Cache-Misses für verschiedene generische Taskzuteilungen zu berechnen. Unter all den verschiedenen Szenarien kann man dann jenes wählen, welches die geringste Anzahl an Cache-Misses erwarten lässt. Dabei muss nicht unbedingt nur die Anzahl der Cache-Misses eine Rolle spielen, es kann auch von Bedeutung sein, ob es sich hauptsächlich um Cache-Misses mit oder ohne anschließendes Zurückschreiben handelt. Ein System mit mehreren Cache-Misses ohne Zurückschreiben kann schneller laufen als ein System mit weniger Cache-Misses mit Zurückschreiben. Das genaue Verhältnis, bis zu dem Cache-Misses ohne Zurückschreiben kürzere Laufzeiten hervorrufen, hängt von der Speicherzugriffszeit und von den parallelen Zugriffen anderer Prozessoren oder Peripheriegeräte ab (siehe Abschnitt 4.2.4 und [73]).

Möchte man in einem System einzelne Tasks oder ISRs dauerhaft im Cache ablegen (Cache-Locking), so ist es sinnvoll, alle diese Tasks auf einen Prozessor zu legen. Je mehr Platz im Cache von solchen Tasks belegt wird, desto besser kann der Hauptspeicher genutzt werden. Wie in Abschnitt 5.2.3 gezeigt wurde, dürfen alle Regionen im Speicher, die mit einem Set im Cache korrespondieren, das Objekte mit dem Attribut *locked* enthält, nicht genutzt werden. Lässt man nun alle Tasks mit dem Attribut *locked* von einem Prozessor ausführen, kann man den Adressbereich dieses Prozessors entsprechend klein machen. Er muss nur so groß wie der Cache sein. Der verbleibende Adressbereich kann dann von anderen Prozessoren genutzt werden.

Ein weiterer Aspekt ist die Kommunikationsstruktur der Tasks. Es ist sinnvoll, Tasks die mit geringen Datenmengen pro Nachricht miteinander kommunizieren, auf einem Prozessor zu platzieren. Dies gilt insbesondere auch für ISRs, die mit Tasks kommunizieren müssen. Bei geringen Datenmengen kann der Datenaustausch über den Cache erfolgen, was die schnellste Methode ist (siehe auch Abschnitt 5.3.4).

Werden größere Datenmengen benötigt, ist ein Austausch über den Cache nicht mehr effizient. In diesem Fall sollte der Datenaustausch über einen gemeinsamen Speicherbereich erfolgen. Prinzipiell spielt es keine Rolle, ob die beteiligten Tasks auf einem oder auf verschiedenen Prozessoren ausgeführt werden. Im Hinblick auf die jeweilige Caching-Strategie (siehe Abschnitt 5.3.4) ist es jedoch sinnvoll, Speicherbereiche mit derselben Strategie nebeneinander zu platzieren. Der Grund dafür ist, dass die Caching-Strategien immer nur für größere Speicherbereiche (z.B. 4 kB) vergeben werden können. Hat man viele kleinere Speicherbereiche für die Interprozesskommunikation (beispielsweise wenn die Tasks auf mehreren Prozessoren ausgeführt werden) und platziert diese in weiter auseinanderliegenden Speicherbereichen, muss man für jeden kleinen Bereich einen größeren mit der gewünschten Caching-Strategie belegen. Dadurch geht unter Umständen viel Speicherplatz für andere Tasks verloren.

Die Grenze, ab der ein Austausch über den Cache ineffizient wird, hängt davon ab, wieviel Speicherplatz im Cache für andere Tasks benötigt wird und wie kritisch deren Zeitanforderungen sind. Die Größe eines Speicherbereichs für den Datenaustausch im Cache kann als Parameter für den Algorithmus zur Anordnung von Code und Daten im Cache angesehen werden. Durch Variieren dieses Parameters verändert sich die Anzahl der zu erwartenden Cache-Misses für das Gesamtsystem. Damit lässt sich eine anwendungsspezifische Effizienzgrenze ermitteln.

5.5 Realzeitnachweis

Für jedes Realzeitsystem muss ein Echtzeitnachweis erbracht werden können. Es muss mathematisch nachgewiesen werden können, dass alle Tasks ihre Deadlines unter allen Umständen einhalten können. Dazu wird eine Worst-Case Betrachtung des Szenarios durchgeführt. Dafür muss man alle relevanten Parameter des Systems kennen, wie zum Beispiel das verwendete Schedulingverfahren und – ganz wichtig – die WCET aller beteiligten Tasks.

Jedes Realzeitnachweisverfahren setzt die Kenntnis der WCET der einzelnen Tasks voraus. Die WCET hängt von vielen Faktoren ab: Ganz allgemein von der Codestruktur und den Eingangsdaten, von der Prozessorarchitektur und dem Verhalten der benötigten und parallel arbeitenden Peripherie. Der Schwerpunkt in dieser Arbeit liegt auf den Einflüssen der Prozessorarchitektur.

Wie schon in Kapitel 4 gezeigt wurde, hängt die Ausführungszeit einer Task auf einem bestimmten Prozessor ganz wesentlich von der nötigen Anzahl der Hauptspeicherzugriffe ab. Diese dauern im ungestörten Fall schon wesentlich länger als Zugriffe in den Cache und können vor allem durch den parallelen Zugriff anderer Geräte auf den Speicher noch erheblich verlängert werden. Die Ursachen und das genaue Verhalten dieser Verzögerungen wird eingehend in [73] untersucht. Das Ziel der Methoden aus Abschnitt 5.2 ist es, Speicherzugriffe soweit als möglich zu verhindern. Wo dies nicht möglich ist, sind sowohl der Ort im Code als auch die Art der Verdrängung bekannt.

Die WCET einer Task setzt sich bei identischen Eingangsdaten ohne Unterbrechungen durch Interrupts wie folgt zusammen:

$$t_{WCET} = t_{BCET} + n_1 \cdot t_{IC} + n_2 \cdot t_{DC} + n_3 \cdot t_{DCWB} + n_4 \cdot t_{TLB}$$

Die Zeit t_{BCET} bezeichnet die ungestörte Ausführungszeit unter optimalen Bedingungen, die *Best-Case Execution Time*. Dies ist zugleich die Mindestlaufzeit. Die weiteren Zeiten sind jene, die für bestimmte Ereignisse (Störungen), die bei der Befehlsausführung auftreten, hinzugerechnet werden müssen: t_{IC} ist die Zeit für einen Miss im Instruktionen-Cache, t_{DC} jene für einen Miss im Daten-Cache ohne Zurückschreiben und t_{DCWB} ist die Zeit, die für einen Miss im Daten-Cache mit Zurückschreiben benötigt wird. Hinzu kommt noch die Zeit t_{TLB} für einen TLB-Miss. Die Parameter n_1 bis n_4 können mit dem Algorithmus aus Abschnitt 5.2 eingestellt und optimiert werden.

Zusätzlich können Laufzeitverlängerungen durch Interrupts entstehen. Dabei muss man nicht nur die Zeit berücksichtigen, die die Interrupt Service Routine im Worst-Case zur Ausführung benötigt, sondern auch die unter Umständen auftretenden Laufzeitverlängerungen der unterbrochenen Task durch Cacheverdrängungen. Dieses Szenario wurde bereits in Abschnitt 4.2.6 ausführlich dargestellt. Weitere Einflussfaktoren, wie zum Beispiel die Branch Prediction (siehe Abschnitt 4.2.3), sind zum einen im Vergleich mit Cache-Misses zu vernachlässigen. Zum anderen sind die Beeinflussungen für einen festen Programmablauf mit identischen Eingangsdaten immer gleich. Ob ein Sprungbefehl verzweigt oder nicht, ist datenabhängig. Die einzige Variable bei diesem Mechanismus ist die Abhängigkeit der Vorhersage von der Ausführung der vorherigen Sprünge (in der Regel vier bis acht). Dieser Einfluss ist jedoch vernachlässigbar.

Betrachtung des Gesamtsystems

Das Hauptproblem bei der Durchführung eines Realzeitnachweises besteht darin, für ein gegebenes System den Worst-Case zu ermitteln. Wie im vorherigen Absatz bereits erläutert, hängt dieser von vielen Faktoren ab. Um diese Komplexität zu bewältigen ist es nötig, die einzelnen Einflussfaktoren des Worst-Case Szenarios soweit möglich voneinander zu trennen, um diese dann einzeln zu untersuchen und zu bewerten. Kann man die Worst-Case Situation durch eine Systemanalyse ermitteln, kann man dann die zu erwartende WCET abschätzen.

Nimmt man als Beispiel eine PC-Architektur, so lassen sich folgende Parameter identifizieren, die sich auf die Laufzeit auswirken (siehe Bild 5.15 und [73]):

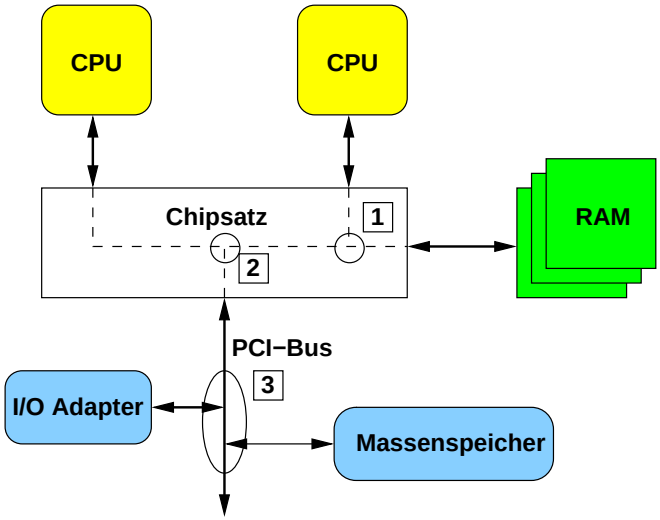

Bild 5.15: Allgemeine Struktur eines PC-Systems (SMP-Architektur, AMD)

Parallele Speicherzugriffe (1 und 2): Parallele Zugriffe auf den Speicher können die Zugriffszeiten aus Sicht des zugreifenden Geräts verlängern, da die Zugriffe in der Regel nur sequentiell durchgeführt werden können. Die Dauer eines Zugriffs hängt dabei von der Anzahl der parallel arbeitenden Geräte (Geräte am Bus, Prozessoren) ab, von der Dauer eines Zugriffs (Burst Transfers) und der Arbitrierungsstrategie des Chipsatzes bzw. Memory Controllers. Punkt 1 in Bild 5.15 kennzeichnet die Verzögerungen durch parallele Zugriffe der CPUs, Punkt 2 jene durch parallele Zugriffe von Peripheriegeräten.

Arbitrierung des Busses: (3): Teilen sich mehrere Geräte einen Bus, wie in Bild 5.15 anhand des I/O Adapters und des Massenspeichers gezeigt, hängt die Zugriffszeit zum einen von der Dauer ab, bis ein Gerät den Bus für sich arbitrieren kann. Diese Zeit wiederum hängt vom Bus Protokoll ab, das heißt in der Regel, wieviele andere Geräte gerade aktiv sind und wieviele Daten diese senden. Zum anderen hängt die Zugriffszeit natürlich auch von den Konflikten in Punkt 1 und 2 ab.

Um für solch ein System nun eine Abschätzung der WCET durchführen zu können, muss zunächst die Worst-Case Situation ermittelt werden. Diese hängt davon ab, was eine Task ma-

chen muss. Hat man beispielsweise eine Task, welche n-mal auf den Speicher zugreifen muss, so muss man für jeden dieser n Zugriffe die maximal mögliche Zugriffszeit im Realzeitnachweis berücksichtigen. Diese tritt auf, wenn zuerst der andere Prozessor einen kompletten Burst Zugriff (in der Regel eine oder mehrere Cachelines) und dann ein Peripheriegerät einen Burst Zugriff maximaler Länge durchführen darf.

Möchte man diese Zeit durch eine Messung bestimmen, müsste man genau diese Situation vor der Messung herstellen. In der Praxis ist dies oft nicht möglich. Daher muss man die jeweiligen Laufzeitverlängerungen einzeln untersuchen und sich dann daraus die Gesamtlaufzeit im Worst-Case ableiten. Dazu ist es nötig, alle Faktoren in Betracht zu ziehen. Es macht keinen Sinn, einfach zu messen und zu versuchen, das System gleichzeitig einer hohen Last auszusetzen. Der tatsächliche Worst-Case würde höchstens durch Zufall auftreten und man könnte nach der Messung nie sagen, ob man den Worst-Case nun mitgemessen hat oder nicht.

Um einen Realzeitnachweis führen zu können, muss man alle Einflussfaktoren in Betracht ziehen. Je komplexer das System ist, desto schwieriger wird es, den Worst-Case zu identifizieren und für eine direkte Messung sicher zu erzeugen. Daher muss man die einzelnen Einflussfaktoren lokalisieren und so weit wie möglich getrennt untersuchen. Für den Fall einer PC-Architektur ist dies möglich, wie es in [73] für die Einflüsse gezeigt wird, die außerhalb des Prozessors liegen. Kennt man dann noch die Laufzeiteigenschaften des Prozessors, kann man die WCET analytisch ermitteln.

6 Ergebnisse und Anwendungsbeispiele

In diesem Kapitel werden zunächst Untersuchungsergebnisse vorgestellt, die sich mit der zu erwartenden Laufzeitverlängerung von Software durch Cache-Misses beschäftigen. Dabei wird der Laufzeitunterschied zwischen L1- und L2-Cache untersucht und verschiedene Strategien zum Speicherzugriff im Hinblick auf Mechanismen zur Interprozesskommunikation analysiert. Ein weiterer Schwerpunkt sind Untersuchungen zu den Einflüssen des Cache-Snooping auf die Laufzeit von Software.

Im zweiten Teil dieses Kapitels werden zwei Beispiele für die Anwendung eines PC-Prozessors für Realzeitaufgaben vorgestellt. Zum einen wird auf die Durchführung einer Matrixmultiplikation eingegangen und deren Laufzeitverhalten in Abhängigkeit der Caches analysiert. Zum anderen wird die *Fast Fourier-Transformation* untersucht, die eine wichtige Rolle in der Signalverarbeitung spielt.

Alle Messungen in diesem Kapitel wurden auf einem Quad-Opteron Rechner mit NUMA-Architektur (1792 MHz pro Prozessor, 64 Bit) durchgeführt. Dieses System ist mit 2 GB DDR-RAM pro Prozessor ausgestattet. Die Prozessoren verfügen über einen 128 kB großen L1-Cache, der je zur Hälfte in einen Instruktionen- und einen Daten-Cache unterteilt ist. Der L2-Cache ist 1 MB groß, die Cacheline-Größe beträgt jeweils 64 Byte. Alle Messungen wurden mit der Messmethodik aus Kapitel 4 durchgeführt. Es wird hier im Wesentlichen der Fall ungestörter Hauptspeicherzugriffe betrachtet. Eine Analyse der Einflüsse beim gleichzeitigen Zugriff konkurrierender Peripheriegeräte wurde in [73] durchgeführt. Die Ergebnisse aus dieser Arbeit werden dazu genutzt, die Laufzeiten bei gestörten Speicherzugriffen abzuschätzen.

6.1 Laufzeitverlängerungen durch Cache-Misses

Der Aufbau und die Funktionsweise des Caches wurden in Abschnitt 3.3 bereits vorgestellt. In modernen Prozessoren ist der Cache meist in zwei Ebenen, den L1- und den L2-Cache unterteilt. Der L1-Cache ist wiederum unterteilt in einen Cache nur für Instruktionen (I-Cache) und einen nur für Daten (D-Cache). Auf I- und D-Cache kann parallel zugegriffen werden. Code oder Daten werden immer zunächst im L1-Cache gesucht, dann im L2-Cache, eventuell in einem L3-Cache und zum Schluss im Hauptspeicher. Die Dauer eines Zugriffs hängt davon ab, in welcher Ebene der Speicherhierarchie der Prozessor die benötigten Daten findet. Zunächst sollen die Unterschiede zwischen einem Zugriff im Cache und ungestörten Zugriffen im Hauptspeicher untersucht werden.

6.1.1 Verzögerungen durch Cache-Misses

Diese Laufzeitverzögerungen entstehen dadurch, dass ein Zugriff auf den Hauptspeicher länger dauert als ein Zugriff in den Cache. Gegenstand der Untersuchungen sind ungestörte Hauptspeicherzugriffe. Blockierungen durch parallelen Zugriff mehrerer Teilnehmer auf den Speicher werden in [73] untersucht.

Man muss zwischen Misses im Instruktionen- und Daten-Cache unterscheiden. Wird eine Instruktion im Cache nicht gefunden, wird diese aus dem Hauptspeicher geladen. Bevor eine neue Cacheline geladen werden kann, muss eine Cacheline in den L2-Cache verdrängt werden. Während die neue Cacheline geladen wird, fängt der Prozessor bereits mit der Dekodierung der ersten Befehle an, das heißt, die Befehlsausführung wird bereits während des Ladevorgangs fortgesetzt. Abhängig davon, wie schnell der Prozessor die neuen Befehle bearbeiten kann, muss er eventuell nochmals kurz warten, bis der nächste Block der Cacheline geladen wurde. Bild 6.1 verdeutlicht dieses Szenario.

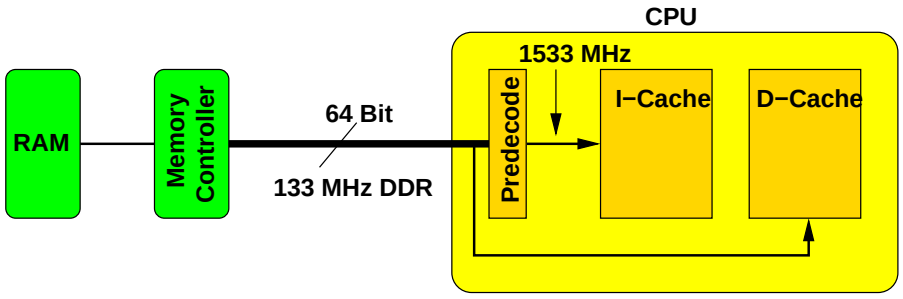

Bild 6.1: Szenario beim Laden einer Cacheline

Nimmt man die Zahlen aus Bild 6.1, so ergibt sich folgende Überlegung: Auf dem Bus können mit einem Taktzyklus 2x64 Bit übertragen werden (16 Byte). Der Prozessor teilt eine Cacheline in sogenannte *Instruction Windows* dieser Größe ein und unterzieht diese zunächst einer *Predecode*-Phase. In dieser Phase wird ein 16 Byte großer Datenblock hinsichtlich der dort enthaltenen Befehle untersucht. Es werden die Grenzen der Befehle detektiert (die Befehle des x86-Befehlssatzes haben unterschiedliche Längen zwischen einem und 15 Byte) und Sprungbefehle indiziert. Im nächsten Schritt gehen die Befehle zum Decoder, der bei den Athlon- bzw. Opteron-Prozessoren bis zu drei Befehle pro Zyklus dekodieren kann.

Das Zeitfenster, bis ein neuer Datenblock eintrifft, beträgt 7,52 ns. Wenn der Prozessor für einfache Befehle im Schnitt zwei Taktzyklen zur Ausführung benötigt, könnte er in dieser Zeit 5,76 Befehle bearbeiten. Das heißt, wenn ein Instruction Window sechs oder mehr Befehle enthält oder die enthaltenen Befehle eine längere Ausführungsdauer benötigen, muss der Prozessor nur auf die ersten 16 Byte wirklich warten, der Rest der Cacheline wird aus Sicht des Prozessors im Hintergrund geladen. Das Gleiche gilt, wenn der Prozessor nur einen Teil der Befehle einer Cacheline benötigt, weil er sich beispielsweise innerhalb einer Schleife befindet.

Aus Sicht eines Realzeitsystems sind nur die Zeiten für den Worst-Case von Interesse. Dieser besteht darin, dass der Prozessor tatsächlich während der Befehlsabarbeitung ständig auf den

langsamen Bus warten muss. Wieviel Zeit der Prozessor dadurch verliert hängt davon ab, wieviele Befehle in einem Instruction Window enthalten sind und wieviel Zeit der Prozessor für die Abarbeitung benötigt. Damit er warten muss, müsste er wenige Befehle pro Instruction Window vorfinden. Dies wären dann allerdings lange Befehle, die im Normalfall auch relativ viel Zeit zur Ausführung benötigen. Die Wartezeit des Prozessors wird daher im Wesentlichen von der Zeit bestimmt, die benötigt wird, bis das erste Instruction Window einer Cacheline geladen wird. Die restlichen Instruction Windows kommen direkt im Anschluss, da eine Cacheline immer in einem Burst-Transfer (d.h. ohne Unterbrechung) übertragen wird.

Diese Überlegungen lassen sich auch auf Misses im Daten-Cache anwenden. Auch hier erfolgt die Übertragung aus dem Speicher in 16 Byte großen Abschnitten und ein Datum ist verfügbar, sobald es im Cache angekommen ist. Ein wesentlicher Unterschied zum Instruktionen-Cache besteht darin, dass möglicherweise zunächst eine Cacheline aufgrund ungesicherter Daten in den Speicher geschrieben werden muss[1)], bevor eine neue geladen werden kann. Auch dieser Effekt wird durch die Prozessorarchitektur oft ausgeglichen, indem eine Verdrängung zunächst in den Victim-Buffer erfolgt und parallel dazu bereits die nächste Cacheline geladen wird, noch bevor die alte zurückgeschrieben wird. Allerdings können sich auch zwei Cachelines gegenseitig stören, wenn zum Beispiel gerade eine Instruktionen-Cacheline noch geladen wird und in der Zwischenzeit schon eine Daten-Cacheline geladen werden soll. Diese Verzögerung ist dann in der Zeit, die benötigt wird, bis die ersten 16 Byte der Daten-Cacheline ankommen, bereits enthalten.

Um die Zugriffszeiten zu ermitteln, wird der Code aus Bild 6.2 benutzt.

```
unsigned long a[LENGTH], b[LENGTH];

void LoadData(void)
{
  unsigned long i;

  for(i=0; i<LENGTH; i++)
    b[i] = a[i];
}
```

Bild 6.2: Code zum Ermitteln der Laufzeitunterschiede

Um die Laufzeitverlängerung für den Fall zu messen, wenn nur der Code nicht im Cache ist, wird vor der Messung der L1-Code und der L2-Cache mit irrelevantem Code gefüllt. Die Datenfelder `a` und `b` werden dabei so groß gewählt, dass die Daten komplett in den L1-Data-Cache passen. Wird das Codestück ausgeführt, muss es zunächst seinen Code aus dem Speicher laden, die Daten sind bereits im Cache. Die gemessene Laufzeitverlängerung spiegelt demnach die Kosten für das Nachladen einer Code-Cacheline wieder. Die meisten Prozessoren führen ein *Prefetching* durch, das heißt, es wird nicht nur die benötigte Cacheline geladen, sondern auch gleich die im Speicher folgende. Das vorliegende Codestück braucht jedoch nur eine Code-Cacheline, es kann demnach von diesem Effekt nicht profitieren.

[1)] Es gibt auch *self-modifying code*. Da dies jedoch für Realzeitsysteme nicht sinnvoll ist, wird dieser Fall hier nicht betrachtet.

Um die Laufzeitverzögerungen durch Zugriffe auf die Daten zu messen, wird ganz ähnlich vorgegangen. Vor jeder Messung wird nur der L1-Daten-Cache und der L2-Cache mit nicht nutzbaren Daten gefüllt, der Code liegt komplett im L1-Cache. Damit ist die gemessene Ausführungszeit ein Maß für die Auswirkungen von Misses im Daten-Cache. Je nach dem, ob die Fülldaten vor der Messung modifiziert werden oder nicht, hat man den Fall, dass die Daten jeweils zurückgeschrieben werden müssen oder nicht.

Bei allen Messungen sind die jeweiligen Adressbereiche komplett in den TLBs, man misst nur die Auswirkungen durch Cache-Misses. Tabelle 6.1 zeigt die gemessenen Zeiten für Misses im Daten- und Instruktionen-Cache für `LENGTH`=4096 (32 kB).

Messgröße	gemessene Ausführungszeit	Laufzeitverlängerung
t_{BCET}	8,09 μs	-
$t_{I-CACHE}$	12,5 μs	54,51%
$t_{D-CACHE}$	33,38 μs	312,61%
$t_{D-CACHE-WB}$	57,59 μs	611,87%

Tabelle 6.1: Laufzeitverlängerungen durch Cache-Misses

Die Zeit t_{BCET} gibt jene Zeit an, die der Code zur Ausführung braucht, wenn keinerlei Cache-Misses auftreten. Dies ist die schnellstmögliche Ausführung. Die Zeit $t_{I-CACHE}$ ist die Ausführungszeit für den Fall, dass nur Misses im Instruktionen-Cache vorkommen, alle Daten sind im Cache. Im Vergleich zu t_{BCET} verlängert sich die Laufzeit in diesem Fall um 54,51%, obwohl der Code nur eine Cacheline benötigt.

Die Zeiten $t_{D-CACHE}$ und $t_{D-CACHE-WB}$ geben die Ausführungszeit an, wenn der Code im Cache ist und die Daten aus dem Speicher gelesen werden müssen. Im Fall von $t_{D-CACHE}$ ist dazu nur ein Lesevorgang nötig, im Fall von $t_{D-CACHE-WB}$ enthält die jeweils verdrängte Cacheline ungesicherte Daten und muss in den Speicher zurückgeschrieben werden. Die Ausführungszeiten sind im Vergleich zu t_{BCET} ca. 4,13 bzw. 7,12 mal so groß.

Diese Ergebnisse zeigen zum einen den immensen Einfluss, den Cache-Misses auf die Ausführungszeit von Software haben. Die gemessenen Zeiten geben auch „nur" die Laufzeitverlängerungen im Fall von ungestörten Hauptspeicherzugriffen an. Kommen noch Wartezeiten hinzu, die durch den gleichzeitigen Zugriff externer Geräte auf den Hauptspeicher verursacht werden, können sich die Faktoren nochmals um ein Vielfaches vergrößern. Näheres dazu ist in der Dissertation von Jürgen Stohr [73] zu finden.

Zum anderen zeigen diese Ergebnisse aber auch, dass nicht alle Cache-Misses in ihrer Wirkung auf die Ausführungszeit von Software gleich sind. Vergleicht man die Laufzeitverlängerungen für Misses im Instruktionen- und im Daten-Cache zeigt sich, dass diese unterschiedlich ins Gewicht fallen. Dies liegt daran, dass der Inhalt von einer Instruktionen Cacheline in der Regel sehr schnell vom Prozessor verarbeitet wird (im Fall serieller Programmausführung). Eine Daten-Cacheline wird normalerweise geladen, wenn der Prozessor ein Datum (beim Opteron acht Byte) daraus benötigt. Bis ein weiteres Datum aus derselben Cacheline benötigt wird, wie im Beispiel aus Bild 6.2, ist in aller Regel genug Zeit vergangen, dass dieses Datum dann auch

bereits im Cache enthalten ist. Damit erklärt sich dann auch die kleinere Zugriffszeit pro Cacheline bei den Datenzugriffen im Vergleich zu dem Zugriff auf die Instruktionen-Cacheline.

Falls für einen Zugriff auf eine Daten-Cacheline erst eine Cacheline mit ungesichertem Inhalt in den Speicher geschrieben werden muss, verlängern sich die Zugriffszeiten nochmals deutlich. Dies hat mit der Konkurrenzsituation zu tun, die entsteht, wenn zunächst Platz für eine neue Cacheline geschaffen werden muss. Ist der Victim-Buffer voll, muss zunächst ein kompletter Schreibzugriff durchgeführt werden, bevor die neue Cacheline geladen werden kann.

Die Ergebnisse aus Tabelle 6.1 zeigen, dass die Art der Verdrängung eine große Rolle spielt. Man kann nicht ganz allgemein von einem Cache-Miss sprechen und diesem eine bestimmte Verzögerungszeit zuordnen. Man muss unterscheiden zwischen Misses im Instruktionen- und Daten-Cache und ob diese mit einem Schreibzugriff verbunden sind oder nicht. Dies berechtigt die Vorgehensweise des Algorithmus aus Abschnitt 5.2, jeweils die Art einer Cacheverdrängung festzulegen und Verdrängungen mit zusätzlichem Schreibzugriff zu minimieren.

Einfluss der Translation Lookaside Buffers

In den Ergebnissen aus Tabelle 6.1 sind keinerlei TLB-Misses (siehe Abschnitt 5.2.4) enthalten, alle Zugriffe waren TLB-Hits. Sind die Daten für die Adressrechnung nicht in den TLBs, müssen diese zunächst aus dem Speicher gelesen werden. Dies ist im Prinzip nichts anderes als ein Cache-Miss. Das Codestück aus Bild 6.2 benötigt 17 Einträge in den Daten-TLBs und einen Eintrag im Code-TLB. Ein Eintrag wird für die Umrechnung aller Adressen innerhalb einer 4 kB großen Seite (page) benötigt.

Misst man nun die Ausführungszeit des Codes, wenn nur TLB-Misses aber keine Cache-Misses auftreten, so erhält man ein $t_{TLB} = 10,8603\ \mu s$. Dies entspricht einer Laufzeitverlängerung gegenüber dem günstigsten Fall von 34,42%. Die Auswirkungen von TLB-Misses auf die Laufzeit sind gegenüber Code- oder Daten-Cache Misses klein. Ein Eintrag in den TLBs ist ausreichend für einen bestimmten Adressraum (z.B. 4 kB), der deutlich größer ist als die Daten, die eine Cacheline liefert (z.B. 4 kB / 64 Byte = 64). Daher kommen TLB-Misses im Normalfall auch deutlich seltener vor als Cache-Misses.

Ein TLB-Miss ist nie mit dem Zurückschreiben eines Eintrags verbunden. Ändert sich etwas an den Adresstabellen, welche die TLBs zwischenspeichern, so werden alle Einträge in den TLBs als ungültig markiert und müssen in der Folge neu gelesen werden. Für ein Realzeitsystem sollte sich daher die Adressraumaufteilung während der Laufzeit nicht ändern. Das lässt sich in der Regel leicht realisieren, da es die Aufgabe des Betriebssystems ist, die Adresstabellen zu verwalten.

Die Messung ist wiederum für einen ungestörten Zugriff auf den Hauptspeicher durchgeführt worden. Bei einer entsprechenden Konkurrenzsituation können sich die Laufzeiten deutlich erhöhen. Die Messung zeigt dennoch, dass die TLBs bei einer Speicheroptimierung mit berücksichtigt werden müssen. Eine Unterscheidung nach Code- oder Daten-TLBs macht keinen Sinn, da die Auswirkungen dieselben sind.

6.1.2 Laufzeitunterschiede zwischen L1- und L2-Cache

Um diese Laufzeitunterschiede messen zu können, muss man sich das Zusammenspiel zwischen L1- und L2-Cache näher ansehen. Bild 6.3 zeigt die Datenpfade für einen Prozessor mit zwei Cache-Ebenen. Die angegebenen Zyklenzahlen im Worst-Case sind Angaben von AMD aus [2].

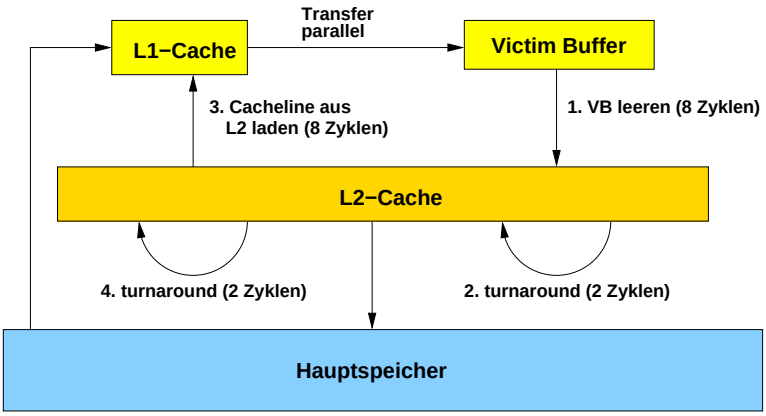

Bild 6.3: Datenpfade zwischen L1- und L2-Cache

Wird eine Cacheline aus dem Hauptspeicher in den L1-Cache geladen, muss dafür eine andere Cacheline verdrängt werden, sofern keine freie Cacheline mehr vorhanden ist. Diese wird – je nach Implementierung – zunächst in einen *Victim-Buffer* verdrängt und von dort zusammen mit dem gesamten Inhalt dieses Buffers (bei AMD acht Cachelines) in den L2-Cache. Nimmt man die von AMD genannten Zahlen als Grundlage, müsste das Laden einer Cacheline vom L2-Cache in den L1-Cache im Worst-Case 20 Prozessortaktzyklen dauern, ansonsten 10 Prozessortaktzyklen. Es gibt auch Implementierungen ohne Victim-Buffer, bei denen die Cachelines direkt in den L2-Cache verdrängt werden (z.B. Intel-Pentium).

Der Sinn des Victim-Buffers ist es, die Zeit für den Verdrängungsvorgang aus dem L1-Cache in den L2-Cache möglichst niedrig zu halten, im Optimalfall bei null. Dies wird dadurch erreicht, dass die Cacheline zunächst in den Victim-Buffer zwischengespeichert wird und parallel dazu die neue Cacheline geladen werden kann. Ansonsten müsste man mit dem Laden der neuen Cacheline warten, bis die alte in den L2-Cache verdrängt wurde.

Interessant aus Sicht eines Realzeitsystems ist die Frage, wie groß die Laufzeitunterschiede für Software sind in Abhängigkeit davon, ob sich der Code und die Daten im L1- oder L2-Cache befinden. Die Zeiten für einen Transfer vom L2- in den L1-Cache lassen sich nicht direkt messen. Mit dem Code aus Abbildung 6.2 kann man die Laufzeiten messen für den Fall, dass sich die Datenfelder `a` und `b` komplett im L1-Cache befinden und für den Fall, dass sie sich über den L1- und den L2-Cache erstrecken. Dies kann mit Hilfe des Parameters `LENGTH` erreicht werden.

Die Laufzeit dieses Codes wird praktisch nur durch die Dauer der Datenzugriffe auf die Datenfelder `a` und `b` bestimmt. Wählt man den Parameter `LENGTH` so, dass `a` und `b` so groß

wie der L1-Daten-Cache sind, erhält man die Laufzeit für die Datenmenge s_D in Bytes (s_D ist die Größe des Daten-Cache in Bytes). Verdoppelt man nun die Datenmenge, ist immer die eine Hälfte eines Datenfeldes im L2-Cache und die andere im L1-Cache. Das hat zur Folge, dass die Datenzugriffe immer aus dem L2-Cache kommen müssen. Wären die Zugriffszeiten gleich, müsste sich die Ausführungszeit verdoppeln. Aus der Abweichung von der doppelten Ausführungszeit kann man auf die unterschiedlichen Zugriffszeiten schließen.

Die Laufzeit für das Programm, wenn es komplett im L1-Cache läuft, beträgt

$$t_1 = t_I + 2 \cdot 4 \cdot LENGTH \cdot \frac{s_B}{m} \cdot t_{D1} \tag{6.1}$$

wobei t_{D1} die Zeit für einen Zugriff auf eine Cacheline im L1-Daten-Cache bezeichnet und t_I die Zeit für einen Zugriff im L1-Instruktionen-Cache. Der Faktor zwei kommt von den beiden Datenfeldern (zwei Zugriffe) und der Faktor vier spiegelt die Tatsache wider, dass immer 16 Byte auf einmal vom Cache kommen. Die Größe der Datenfelder `LENGTH` muss mit der Größe der Daten s_B in Byte multipliziert werden und durch die Größe einer Cacheline m dividiert werden, um die Anzahl der Cachelines zu erhalten, auf die zugegriffen wird.

Die Laufzeit t_2, wenn die Datenfelder `a` und `b` doppelt so groß sind, beträgt

$$t_2 = t_I + 2 \cdot LENGTH \cdot \frac{s_B}{m} \cdot (t_{D2} + 3 \cdot t_{D1}) \tag{6.2}$$

Der Faktor zwei berücksichtigt wiederum die beiden Datenfelder. Die doppelte Datenmenge wird durch den Faktor `LENGTH` berücksichtigt. Der Klammerausdruck berücksichtigt die Tatsache, dass nur der erste 16 Byte Zugriff aus dem L2-Cache kommt, die folgenden drei 16 Byte Zugriffe kommen dann schon aus dem L1-Cache.

Die Größen t_1 und t_2 können mit der Messroutine aus Kapitel 4 sehr zuverlässig gemessen werden. Einflüsse der Peripherie spielen keine Rolle, da sich das Geschehen ausschließlich im Cache abspielt. Trifft man die Annahme, dass $t_I = t_{D1}$, so kann man aus Gleichung 6.1 die Zeit t_{D1} bestimmen. Aus Gleichung 6.2 kann man dann die Zeit t_{D2} berechnen. Tabelle 6.2 fasst die Ergebnisse zusammen.

Größe	gemessener/berechneter Wert	`LENGTH`
t_1	8,092683 μs	4096
t_2	24,208467 μs	8192
t_{D1}	1,9752704 ns	-
t_{D2}	5,8937648 ns	-
t_{D2}/t_{D1}	2,98	-

Tabelle 6.2: Unterschiedliche Zugriffszeiten auf L1- und L2-Cache

Die Zeit t_1 ist die Dauer für Zugriffe auf ein 64 kB großes Datenfeld, t_2 für ein 128 kB großes Feld. Ein Zugriff auf den L2-Cache dauert ca. dreimal länger als ein Zugriff auf den L1-Cache. Die Zugriffszeiten skalieren mit der Taktfrequenz der Prozessoren, da sie mit voller Taktfrequenz auf den Cache zugreifen können.

6.2 Auswirkungen des Cache-Snooping

In Abschnitt 5.3.5 wurde das MOESI-Protokoll beschrieben, mit dessen Hilfe das Cache-Snooping bei den Prozessoren der AMD-Athlon und AMD-Opteron Familie realisiert wird. Dieses Protokoll vermeidet den Zugriff auf den Hauptspeicher, wenn eine Cacheline aktualisiert werden muss. Statt dessen wird die betreffende Cacheline direkt zwischen den betroffenen Prozessoren ausgetauscht. Die Frage, die sich nun stellt, ist, wie das Cache-Snooping die Ausführungszeit von Software beeinflusst.

Um diese Zeiten zu ermitteln, wird wiederum auf den Code aus Bild 6.2 zurückgegriffen. Die Zeit, die der Transport einer Cacheline durch den Snooping-Mechanismus benötigt, wird gemessen, indem eine weitere Task auf einem anderen Prozessor ein oder mehrere Elemente des Datenfeldes `a` modifiziert. Die sich ergebende Abweichung in der Laufzeit des Codes aus Bild 6.2 ist ein Maß für den Einfluss des Cache-Snooping. Voraussetzung dafür ist, dass sich der komplette Code und die Daten im Cache des jeweiligen Prozessors befinden. Ein weiterer interessanter Punkt ist, ob die Entfernung der Prozessoren in dem Quad-Opteron System einen nennenswerten Einfluss auf die Ausführungszeiten hat. Tabelle 6.3 fasst die Messergebnisse zusammen. Der Parameter `LENGTH` beträgt jeweils 1024. Bild 6.4 zeigt dazu den schematischen Aufbau des Quad-Opteron Rechners.

Messgröße	gemessene Zeit	Knoten: von-nach	Laufzeitverlängerung
t	3,599896 μs	kein Snooping	-
t_{Snoop}	4,044668 μs	1-3	15,53%
t_{Snoop}	4,068118 μs	1-2	16,20%
t_{Snoop}	4,067560 μs	2-3	16,18%

Tabelle 6.3: Auswirkungen des Cache-Snooping

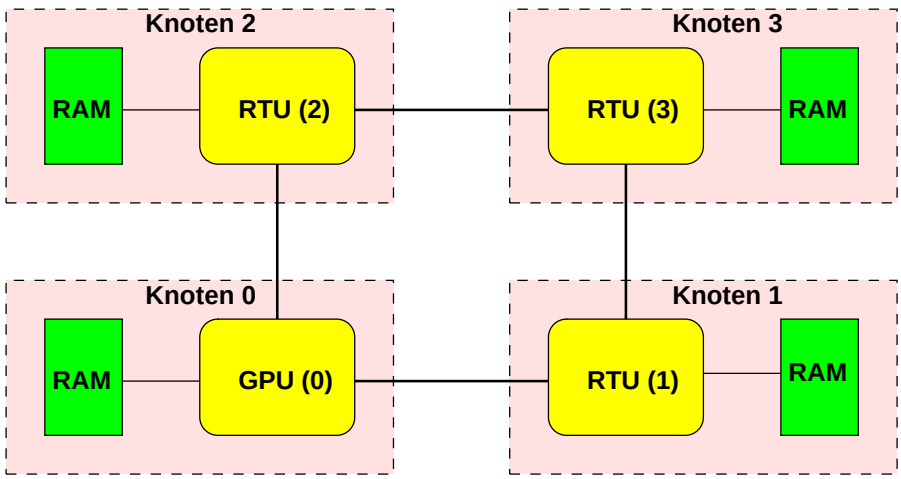

Bild 6.4: Schematischer Aufbau des Quad-Opteron Rechners

Die Zeit t_{Snoop} gibt jeweils die gemessene Zeit für den Code aus Bild 6.2 an, wenn gleichzeitig auf einem anderen Knoten eine Task läuft, die vier Werte des Datenfeldes `a` verändert. Diese vier Werte liegen in unterschiedlichen Cachelines, das heißt, es müssen vier Cachelines mit Hilfe des Snooping-Mechanismus' ausgetauscht werden.

In Bild 6.5 ist der Verlauf der Ausführungszeiten in Abhängigkeit von der Anzahl der Cachelines aufgetragen, die mittels Cache-Snooping aktualisiert werden müssen. Die Stützwerte wurden alle für den Datenaustausch zwischen Prozessor 1 und Prozessor 3 gemessen.

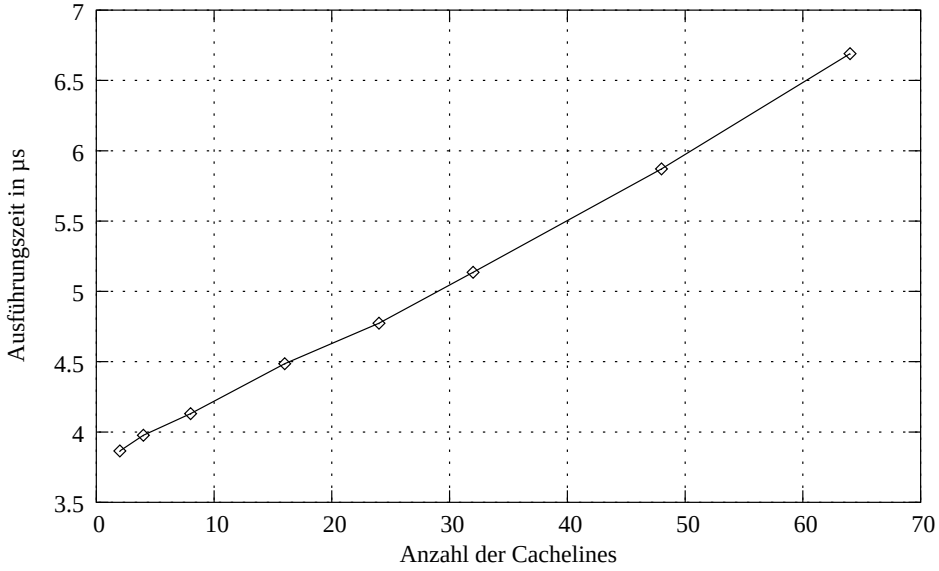

Bild 6.5: Einfluss des Cache-Snooping auf die Ausführungszeit

Man kann deutlich den linearen Zusammenhang erkennen. Im Vergleich zur ungestörten Ausführungszeit ergibt sich ein Offset und dann ein linearer Verlauf. Auch ein parallel zu dem Cache-Snooping erzeugter Datenstrom ändert nichts an diesem Ergebnis. Die Ergebnisse in Tabelle 6.3 zeigen auch, dass es keinen nennenswerten Unterschied gibt, ob die beiden Prozessoren direkt miteinander gekoppelt sind, oder ob ein weiterer Prozessor dazwischen liegt.

Kennt man die Laufzeit einer Routine, wenn alle Daten im Cache sind, kann man die Laufzeit abschätzen, wenn eine bestimmte Anzahl an Daten mittels Cache-Snooping synchronisiert wird. Dies ist auch ein wichtiges Ergebnis für die Interprozesskommunikation (siehe Abschnitt 5.3.4), die diesen Mechanismus nutzen kann. Damit ist es möglich, die Dauer eines Datentransfers in Abhängigkeit der Datenmenge anzugeben. Der gemessene lineare Verlauf in Bild 6.5 zeigt auch, dass der Jitter sehr gering ist. Somit können die Laufzeiten sehr zuverlässig bestimmt werden.

Dies gilt natürlich nur unter Verwendung des MOESI-Protokolls. Wird das MESI-Protokoll eingesetzt, sind jeweils zwei Speicherzugriffe zu berücksichtigen (siehe Abschnitt 5.3.5). Diese können je nach Konkurrenzsituation beim Zugriff auf den Speicher stark variieren und sind daher im Allgemeinen nur schwer vorhersagbar. Auch wird dort die Aktualisierung einer Cacheline insgesamt mehr Zeit in Anspruch nehmen.

6.3 Interprozesskommunikation

In Abschnitt 5.3.4 wurden verschiedene Ansätze zum Datenaustausch zwischen Tasks vorgestellt. Dabei wurde zum einen auf verschiedene Caching-Strategien der Prozessoren eingegangen, die man zu diesem Zweck nutzen kann. Zum anderen wurde die Möglichkeit diskutiert, wie man das Cache-Snooping mit Hilfe des MOESI-Protokolls zum schnellen Datenaustausch nutzen kann. Diese Möglichkeit wurde im vorherigen Abschnitt bereits untersucht.

In diesem Abschnitt werden die Auswirkungen der verschiedenen Caching-Strategien auf die Laufzeit von Software untersucht. Auch hier wird wiederum davon ausgegangen, dass alle nötigen Hauptspeicherzugriffe nicht von weiteren Komponenten gestört werden.

Bild 6.6 zeigt die gemessenen Laufzeiten für die Strategien *Strong Uncacheable* (UC), *Write Combining* (WC), *Write-Through* (WT) und *Write Protected* (WP). Dargestellt ist die Zeit zum Lesen und Schreiben in Abhängigkeit der Datenmenge in kB. Die Zeiten wurden in einem Diagramm zusammengefasst, da sie sich praktisch nicht unterscheiden.

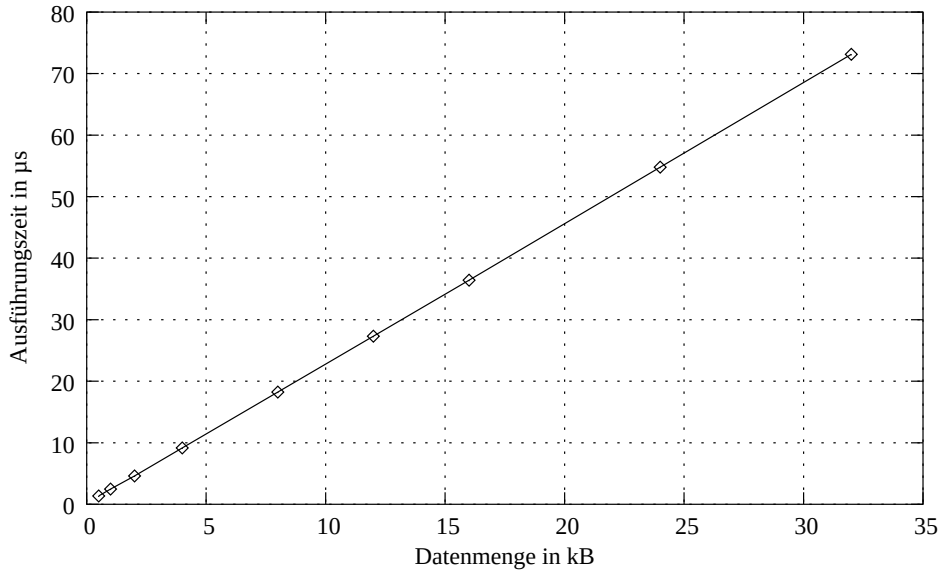

Bild 6.6: Zugriffszeiten der Strategien UC,WC,WT und WP

Die einzelnen Messpunkte wurden jeweils mit Geradenstücken verbunden, so dass die Linearität gut sichtbar wird. Die UC-Strategie schaltet die Caches praktisch ab, es ist auch kein Write-Combining erlaubt, das heißt, auch die Schreibpuffer des Prozessors sind deaktiviert. Die WC-Strategie ändert daran nichts, außer dass Write-Combining möglich ist. Dieser Effekt bringt aber bei aufeinander folgenden Speicherzugriffen, wie sie bei der Interprozesskommunikation auftreten, nichts. Die WT-Strategie lädt zwar beim Lesen den Speicherinhalt in den Cache, beim Schreiben wird er jedoch sowohl im Cache als auch im Speicher aktualisiert, so dass auch hier keine wesentlichen Änderungen der Laufzeit zu erwarten sind. Die WP-Strategie verhält sich sehr ähnlich wie die WT-Strategie, außer dass beim Schreiben der Cache nicht aktualisiert, sondern invalidiert wird.

Für den Datenaustausch zwischen Prozessen auf unterschiedlichen Prozessoren bedeutet dies, dass die Zeit, die ein Prozessor benötigt, um die Daten in den Speicher zu schreiben, in Abhängigkeit der Datenmenge sehr gut abgeschätzt werden kann. Diese Speicherzugriffsstrategien erlauben es, die Zugriffsart für bestimmte Bereiche im Speicher festzulegen, ohne dass andere Bereiche davon betroffen sind.

Vergleicht man diese Werte, die faktisch Hauptspeicherzugriffe darstellen, mit den Werten von Tabelle 6.1 aus Abschnitt 6.1.1, so fällt auf, dass die Werte dort kleiner sind als jene aus Bild 6.6. In Abschnitt 6.1.1 wurde die Zeit für einen Cache-Miss unter verschiedenen Bedingungen ermittelt. Das heißt, der Rechner hat prinzipiell mit aktiven Caches gearbeitet und hat damit auch von Prefetching-Effekten profitiert. Als Folge eines Cache-Misses wird immer mindestens eine ganze Cacheline geladen und die Ausführung wird bereits fortgesetzt, nachdem ein Teil dieser Cacheline geladen wurde. Arbeitet der Prozessor ganz ohne Caches, fallen diese Effekte weg, und der Prozessor muss für jedes zu ladende Datum einen Speicherzugriff initiieren. Dadurch ergeben sich längere Ausführungszeiten.

6.4 Anwendungsbeispiele

In diesem Abschnitt werden die Auswirkungen von Cache-Misses auf die Laufzeit von Software anhand verschiedener Beispielprogramme untersucht. Betrachtet werden die Verzögerungen durch Instruktionen- und Daten-Cache-Misses und der Einfluss der TLBs. Dabei wurden Beispielszenarien gewählt, wie sie auch in der Praxis häufig vorkommen. Diese Untersuchungen sollen Aufschluss darüber geben, wie sich die Maßnahmen zur Speicheroptimierung aus Kapitel 5 in der Praxis auf die Laufzeit von Software auswirken und wann sich eine Optimierung lohnt.

6.4.1 Matrixmultiplikation

Eine in der Praxis häufig vorkommende Aufgabe ist die Multiplikation zweier Matrizen. Der dafür nötige Code hat in der Regel einen sehr kleinen Umfang verglichen mit der Menge an Daten, die bearbeitet werden müssen. Bild 6.7 zeigt einen Beispielcode, der zwei Matrizen `a` und `b` miteinander multipliziert.

```
void matrixmult(void) {
  unsigned long i,j,k;
  k = 0;
  for(i=0; i<ROW*COLUMN; i+=ROW) {
      for(j=0; j<COLUMN; j++)
        c[i+j] = a[k]*b[j] + a[k+1]*b[j+ROW];
      k += ROW;
    }
}
```

Bild 6.7: Matrixmultiplikation

Die beiden Matrizen sind als Datenfelder implementiert, wobei die ersten `COLUMN` Felder der ersten Zeile der Matrix entsprechen, die zweiten `COLUMN` Felder der zweiten Zeile etc. Die Größe eines Datenfeldes entspricht demnach `ROW·COLUMN`. Die Codegröße beträgt 95 Byte, das heißt, der Code benötigt zwei Cachelines im Instruktionen-Cache. Die Datenmenge ist variabel und hängt davon ab, wie die Parameter `ROW` und `COLUMN` gewählt werden. Der Code wurde bewusst nicht auf eine bestimmte Prozessorarchitektur hin optimiert, um die Auswirkungen von Speicherzugriffen an einem verständlichen und überschaubaren Beispiel studieren zu können. Im Folgenden wird untersucht, wie die Laufzeit dieses Codes von Misses im Instruktionen- und Daten-Cache abhängt, auch in Abhängigkeit der Datenmenge. Daraus lässt sich dann die Frage beantworten, was eine Speicheroptimierung an dieser Stelle leisten kann.

Die Parameter `ROW` und `COLUMN` sind beide auf 100 gesetzt. Bild 6.8 zeigt die Ausführungszeit des Programms für den Fall, dass sowohl Code als auch Daten komplett im Cache (L1- und L2-Cache) sind, ebenso wie alle benötigten TLB-Einträge.

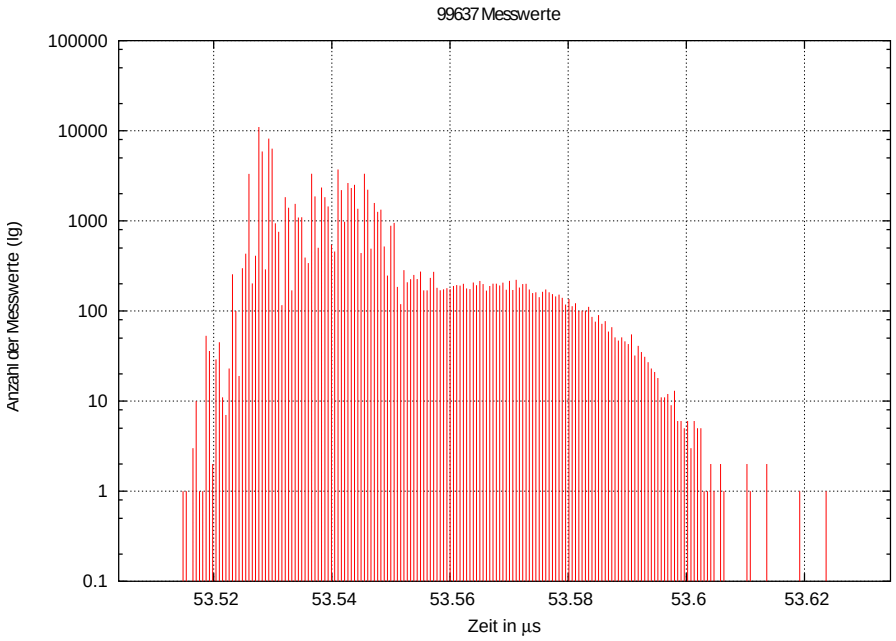

Bild 6.8: Ausführungszeit im ungestörten Fall

Die minimale gemessene Ausführungszeit beträgt 53,51 μs, die maximale 53,62 μs, der Jitter beträgt 0,2%. Man sieht die diskreten Zeiten, die sich durch Umladevorgänge im L1- und L2-Cache und den Victim-Buffer ergeben. Dieser Fall stellt aus Sicht eines Realzeitsystems den Best-Case dar, da alle Daten im Cache sind und daher die Ausführungszeit nicht von externen Peripheriegeräten durch parallele Speicherzugriffe verlängert werden kann. Die einzig mögliche Verzögerung wären Interrupts.

Im Vergleich dazu zeigt Bild 6.9 den Fall, dass weder Code noch Daten im Cache sind und keine Einträge in den TLBs genutzt werden können. Die Verdrängungen aus dem L2-Cache sind jeweils mit dem Zurückschreiben ungesicherter Daten verbunden.

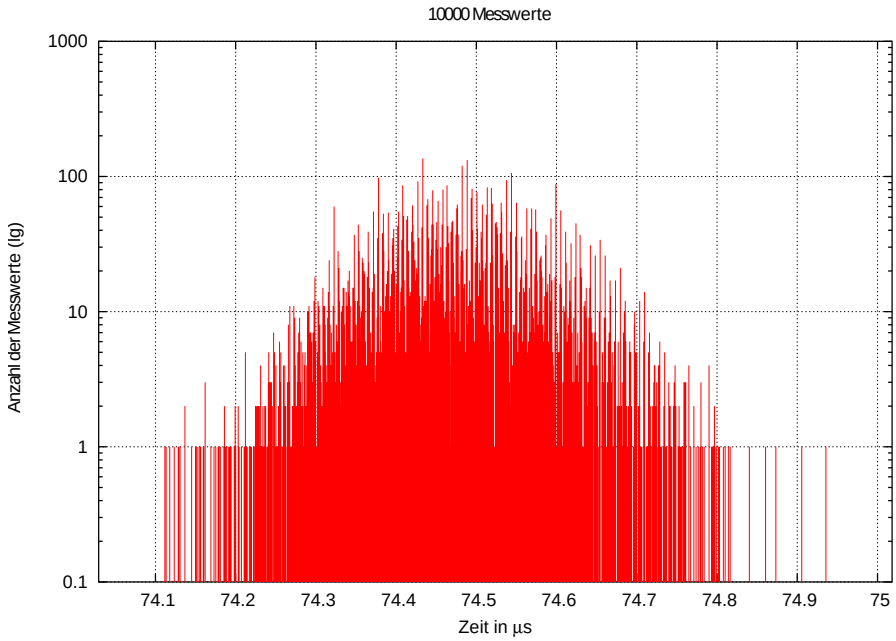

Bild 6.9: Ausführungszeit, wenn weder Code noch Daten im Cache sind

Die minimale Ausführungszeit beträgt 74,11 μs, die maximale 74,94 μs, das ergibt einen Jitter von 1,1%. Dieser verhältnismäßig geringe Jitter ist darauf zurückzuführen, dass diese Messung ohne den Einfluss externer Peripheriegeräte durchgeführt wurde. Das heißt, der Prozessor konnte ungestört auf seinen lokalen Speicher zugreifen. Würden parallel dazu Zugriffe parallel arbeitender Geräte oder Prozessoren auftreten, würde sich ein Ergebnis ähnlich wie in Bild 4.10 ergeben. Eine Abschätzung, um wieviel sich die Ausführungszeiten in diesem Fall erhöhen würden, wird am Ende dieses Abschnitts vorgenommen.

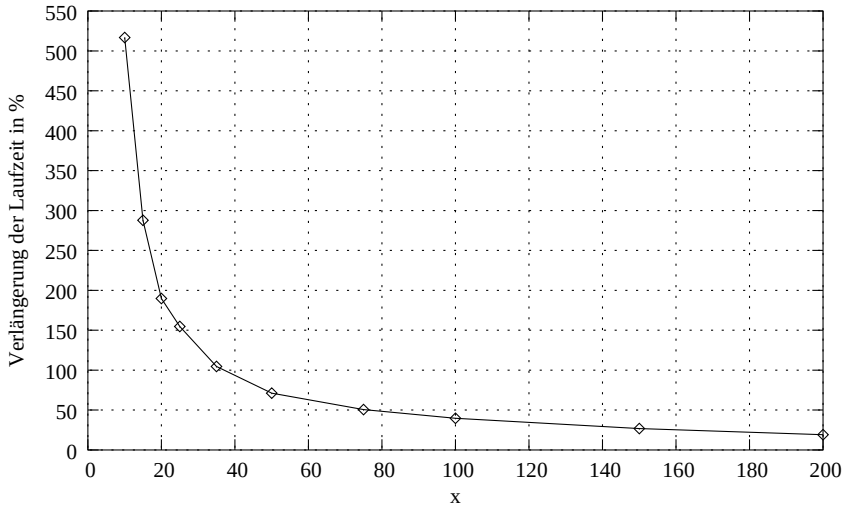

Bild 6.10: Laufzeitverlängerung in Abhängigkeit der Datenmenge

Im Vergleich zu der vorangegangenen Messung in Bild 6.8 verlängert sich die Laufzeit um 21,32 μs, das entspricht 39,76%. Bild 6.10 zeigt die Verlängerung der Laufzeit in Abhängigkeit der Dimension x der Matrizen. Es werden immer quadratische Matrizen benutzt. Die Anzahl der benötigten Daten-Cachelines in Abhängigkeit von x ergibt sich zu $3/8 \cdot x^2$.

Man kann deutlich erkennen, dass die Laufzeitunterschiede, je nachdem, ob die Daten alle im Cache oder alle nicht im Cache sind, mit steigender Datenmenge schnell abnehmen. Bei kleinen Datenmengen ist es demnach sehr entscheidend, ob die Daten im Cache sind oder nicht. Bei steigender Datenmenge profitiert der Code von dem Prefetching-Effekt, dass immer eine komplette Cacheline geladen wird und nicht nur das benötigte Datum. Ein Datum ist hier acht Byte groß, das heißt, mit einer Cacheline werden acht Daten auf einmal geladen. Im Hinblick auf eine Speicheroptimierung bedeutet das, dass es sehr wichtig ist, kleine Datenobjekte so im Speicher zu platzieren, dass es keine Verdrängungen im Cache gibt. Hat man große Datenobjekte, so ist eine effektive Platzierung oft nicht mehr möglich, die Auswirkungen sind aber auch nicht mehr so gravierend.

Bild 6.11 zeigt den Einfluss des Instruktionen-Caches auf die Laufzeit des Beispielprogramms.

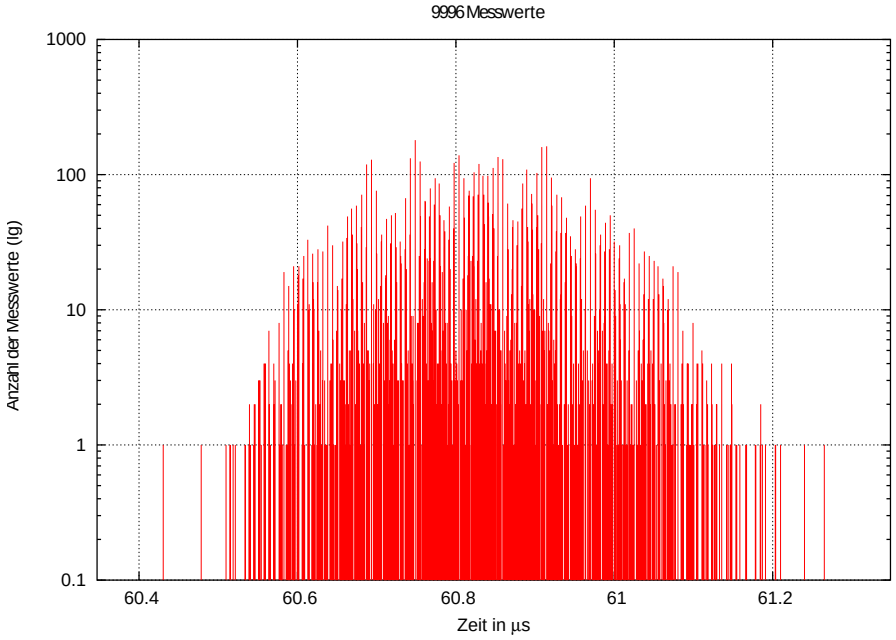

Bild 6.11: Code nicht im Cache, alle Daten im Cache

Die gemessenen Ausführungszeiten schwanken zwischen 60,43 μs und 61,27 μs. Im Vergleich zu den Zeiten aus Bild 6.8 bedeutet dies eine Laufzeitverlängerung von 14,25%, obwohl nur zwei Cachelines betroffen sind.

Eine weitere interessante Fragestellung ist, inwieweit sich das Zurückschreiben bei den Cacheverdrängungen auf die Laufzeit auswirkt. Ein weiteres wichtiges Kriterium der in Kapitel 5

vorgestellten Speicheroptimierung ist die Sortierung der Objekte nach Attributen, damit im Falle einer Cacheverdrängung bekannt ist, ob die Daten zuvor gesichert werden müssen oder nicht. Bild 6.12 zeigt dieselbe Messung wie sie in Bild 6.8 und Bild 6.9 dargestellt sind, mit dem Unterschied, dass der Code im Cache ist und die Verdrängungsvorgänge im Daten-Cache ohne Zurückschreiben stattfinden.

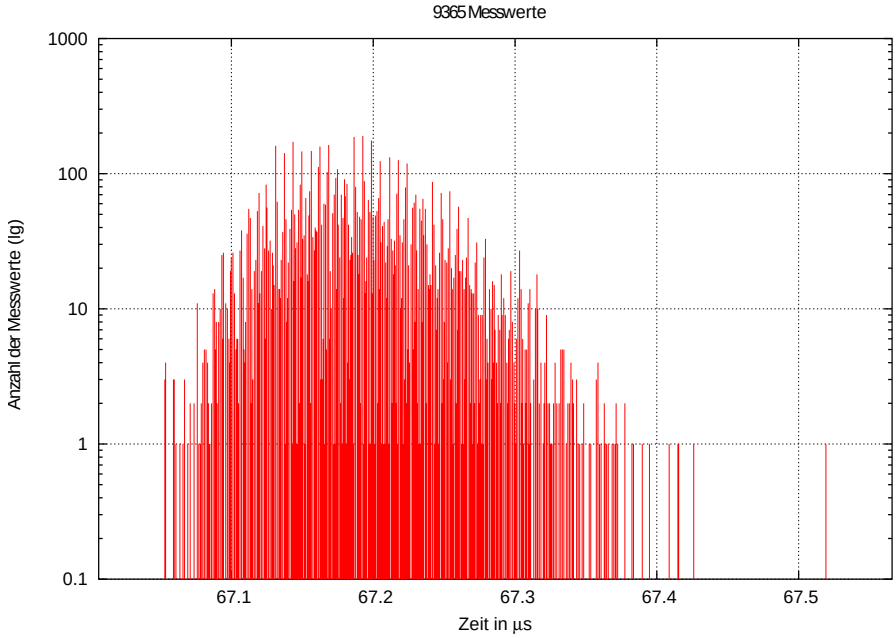

Bild 6.12: Code im Cache und Datenverdrängungen ohne Zurückschreiben

Die gemessenen Zeiten bewegen sich zwischen 67,05 μs und 67,52 μs. Misst man dasselbe Szenario mit Zurückschreiben (Code im Cache), so ergeben sich Zeiten zwischen 71,56 μs und 72.66 μs. Gegenüber dem Fall ohne Zurückschreiben ist das eine Steigerung von ca. sechs Prozent. Diese ist allerdings nur deshalb so gering, da die Verdrängungsvorgänge ungestört ablaufen. Wäre dies nicht der Fall, wären die Verzögerungen weitaus größer.

Führt man die Messung durch, wenn Code und Daten im Cache sind, aber die TLBs vor der Messung invalidiert werden, so ergeben sich Ausführungszeiten zwischen 54,1 μs und 54,28 μs. Verglichen mit der Messung aus Bild 6.8 ist das eine Steigerung von 1,4%, obwohl das Programm einen Eintrag für den Code und 43 Einträge für die Daten benötigt. Dieses Beispiel zeigt, dass der Einfluss der TLBs im Vergleich zu den Caches eher gering ist.

Die Ausführungszeit wird wesentlich davon beeinflusst, ob die drei benötigten Datenfelder `a`, `b` und `c` im Daten-Cache sind oder nicht. Betrachtet man den Beispielcode aus Bild 6.7 sieht man, dass das Datenfeld `a` pro Wert mit `k` und `k+1` indiziert wird, also zwei aufeinander folgenden Feldern. Datenfeld `b` wird mit `j` und `j+ROW` indiziert, das heißt, wenn `ROW` größer als eine Cacheline ist, müssen für die Berechnung eines Eintrags der Ergebnismatrix `c` zwei Cachelines geladen werden. Im Gegensatz dazu ist für Datenfeld `a` immer eine Cacheline ausreichend, außer die beiden aufeinander folgenden Indices liegen in aufeinander folgenden Cachelines.

Datenfeld c wird immer in aufsteigender Folge indiziert und profitiert demnach voll vom Daten-Cache.

Misst man nun die Ausführungszeiten, wenn eines der drei Datenfelder nicht im Cache ist, ergibt sich folgendes Bild: Ist a nicht im Cache, verlängert sich die Laufzeit auf 62,5 μs gegenüber dem ungestörten Fall aus Bild 6.8, ist b nicht im Cache verlängert sich die Laufzeit auf 54,2 μs und ist c nicht im Cache, ergeben sich 57,84 μs. Dieses Szenario zeigt, dass verschiedenen Datenobjekten bei der Optimierung unterschiedliches Gewicht zukommen kann, je nachdem, wie auf diese Objekte zugegriffen wird. In diesem Beispiel würde eine Optimierung von a, das heißt, wenige oder gar keine Cache-Misses beim Zugriff auf a, mehr Geschwindigkeitszuwachs bringen als ein Optimierung von b oder c.

Tabelle 6.4 fasst die Messergebnisse für den Fall ungestörter Hauptspeicherzugriffe zusammen.

Messung	Laufzeitverlängerung	Jitter
ohne Störungen	-	0,2%
Daten nicht im Cache	25,6%	0,7%
Daten nicht im Cache mit Zurückschr.	34,6%	1,5%
Code nicht im Cache	14,25%	0,8%
TLBs invalidiert	1,4%	0,3%
TLBs und Caches invalidiert	39,76%	1,1%
a nicht im Cache	16,56%	0,16%
b nicht im Cache	1,1%	0,53%
c nicht im Cache	7,87%	0,86%

Tabelle 6.4: Zusammenfassung der Ergebnisse

Der geringe Jitter auch bei vielen Verdrängungsvorgängen zeigt, wieviel man erreichen kann, wenn der Prozessor ohne Blockierungen auf den Speicher zugreifen kann. Der nächste Abschnitt gibt eine Abschätzung, welche Zeiten und Jitter zu erwarten sind, falls die Hauptspeicherzugriffe durch parallele Aktivitäten im System verzögert werden.

Abschätzung der Auswirkungen paralleler Zugriffe

Die Ergebnisse in Tabelle 6.4 spiegeln nicht die Verhältnisse wider, wie sie in einem realen PC-System auftreten würden. Diese Zahlen zeigen, wieviel Laufzeitgewinn man mindestens erreicht, wenn man die Caches entsprechend nutzt. Vor allem der sehr geringe Jitter fällt hier auf. In einem realen System werden aber immer auch noch Peripheriegeräte sein, die parallel zu einem Prozessor auf den Speicher zugreifen. Das können Prozesssignaladapter sein, Massenspeicher oder ein Ethernetanschluss. Aber auch weitere Prozessoren in einem Multiprozessorsystem nutzen den Speicher parallel, wenn auch nicht unbedingt dieselben Speicherbereiche.

Bei parallelen Zugriffen auf den Hauptspeicher kommt es zu Blockierungen, da in aller Regel immer nur ein Teilnehmer zur selben Zeit auf den Speicher zugreifen kann. Die dadurch ent-

stehenden Blockierungszeiten können sehr groß werden. Jürgen Stohr hat in seiner Dissertation ([73]) unter anderem die Auswirkungen paralleler Zugriffe auf die Dauer von Hauptspeicherzugriffen einer CPU untersucht.

Er hat dabei einen linearen Zusammenhang zwischen der Zugriffszeit t_{WCAT} (WCAT: Worst-Case Access Time) im Worst-Case und der Anzahl der kontinuierlich durchgeführten Zugriffe festgestellt:

$$t_{WCAT}(x) = t_{WCAT,1} + m_{WCAT} \cdot (x - 1) \tag{6.3}$$

Die Zeit $t_{WCAT,1}$ in Gleichung 6.3 ist die Zeit, die für den ersten Hauptspeicherzugriff benötigt wird. Mit x wird die Anzahl der nötigen Speicherzugriffe insgesamt bezeichnet. Mit dieser Abschätzung ist es möglich, die Auswirkungen paralleler Aktivitäten im System auf die Laufzeit von Software zu bestimmen. Je nach Art der Einflüsse ergibt sich ein anderer Wert $t_{WCAT,1}$ und eine andere Steigung m_{WCAT} der Geraden.

Für das Beispiel der Matrixmultiplikation soll nun mit Hilfe von Gleichung 6.3 abgeschätzt werden, um wieviel sich die Ausführungszeiten und der Jitter steigern würden bei parallel auftretenden Hauptspeicherzugriffen. Nimmt man als Worst-Case Szenario an, dass sowohl externe Peripheriegeräte auf den lokalen Speicher zugreifen, als auch zwei weitere Prozessoren (Fall 1), so ergibt sich nach [73] eine Steigung $m_{WCAT} = 0,47\mu s$ und eine Zeit $t_{WCAT,1}$ von $0,55\mu s$. Die so errechneten Resultate fasst Tabelle 6.5 zusammen.

Messung	Laufzeitverlängerung	Jitter
TLBs und Caches invalidiert	6592%	4732%
Daten nicht im Cache (ohne Zurückschreiben)	3295%	2609%
TLBs invalidiert	51,97%	50,31%

Tabelle 6.5: Abgeschätzte Zeiten bei parallelen Hauptspeicherzugriffen (Fall 1)

Im Vergleich zu Tabelle 6.4 ergeben sich erhebliche Unterschiede. Dies liegt daran, dass die Ausführungszeit der Routine zur Matrixmultiplikation ganz entscheidend von der Dauer der Hauptspeicherzugriffe bestimmt wird. Für die Berechnung der Werte wurde ein Worst-Case Szenario angenommen, welches man in der Praxis für ein Realzeitsystem unbedingt vermeiden sollte (Peripheriezugriffe *und* gleichzeitig Zugriffe zweier Prozessoren, jeweils mit maximaler Datenrate). Würde man das Szenario dahingehend ändern, dass nur Peripheriegeräte parallele Hauptspeicherzugriffe durchführen (Fall 2), so erhält man Werte, wie sie in Tabelle 6.6 gezeigt sind. Diese ergeben sich für ein $t_{WCAT,1}$ von $0,28\mu s$ und $m_{WCAT} = 0,14\mu s$.

Messung	Laufzeitverlängerung	Jitter
TLBs und Caches invalidiert	1964%	1390%
Daten nicht im Cache (ohne Zurückschreiben)	982%	764%
TLBs invalidiert	15,69%	14,42%

Tabelle 6.6: Abgeschätzte Zeiten bei parallelen Hauptspeicherzugriffen (Fall 2)

Die Werte aus Tabelle 6.5 und Tabelle 6.6 sind Überabschätzungen des realen Worst-Case Verhaltens, da sie davon ausgehen, dass *alle* Hauptspeicherzugriffe um den maximal möglichen Betrag verlängert werden. In der Praxis wird dies sicherlich nur für einen bestimmten, wenn auch hohen Prozentsatz, der Fall sein. Würde man versuchen, diese Zeiten direkt zu messen, würde man in wiederholten Messungen immer verschiedene Werte mit einer hohen Streuung messen, da die Blockierzeiten sehr davon abhängen, wie die einzelnen Ereignisse, die zu den Blockierungen führen, zusammentreffen.

Bild 6.13 zeigt die Werte aus den Tabellen 6.4, 6.5 und 6.6 nochmal im Vergleich. Man sieht deutlich die verhältnismäßig geringen Laufzeitunterschiede für den Fall ungestörter Hauptspeicherzugriffe und die sehr großen Laufzeitunterschiede für den Fall gestörter Hauptspeicherzugriffe. Auch der Unterschied zwischen verschiedenen Störszenarien wird deutlich (Fall 1 und Fall 2).

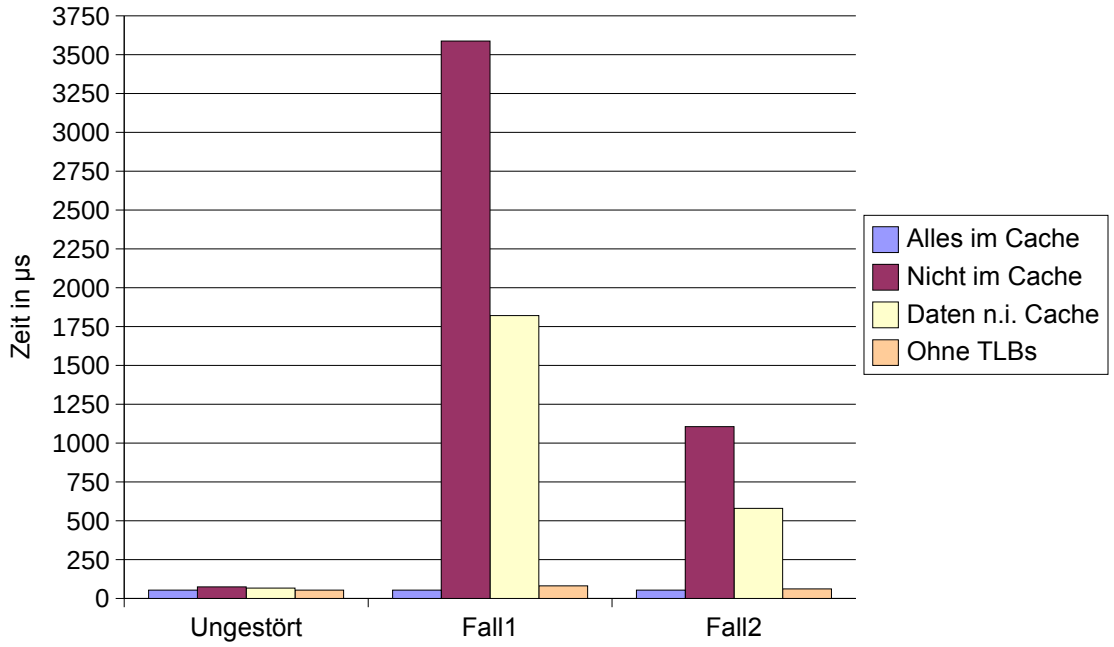

Bild 6.13: Ausführungszeiten für die Matrixmultiplikation

Diese Abschätzungen zeigen sehr deutlich, wie stark die Ausführungszeit des Beispielprogramms von den Hauptspeicherzugriffen abhängt. Es wird auch der Unterschied zwischen Verdrängungen mit und ohne Zurückschreiben deutlich. Die Werte für Verdrängungen mit Zurückschreiben entsprechen in etwa jenen, wenn weder Code noch Daten im Cache sind und sind daher in den Tabellen 6.5 und 6.6 nicht extra angegeben. Auch der Jitter steigt im Vergleich zu ungestörten Zugriffen sehr stark an. Diese Ergebnisse zeigen, wie wichtig es ist, die Anzahl an Verdrängungsvorgängen mit Zurückschreiben zu minimieren. Der Vergleich zwischen Fall 1 und Fall 2 zeigt aber auch, dass man mit einer geeigneten Anordnung seines Systems im Speicher, auch wenn große Teile nicht in den Cache passen, ein deutlich besseres Laufzeitverhalten erzielen kann.

6.4.2 Fast Fourier-Transformation (FFT)

Ein in der digitalen Signalverarbeitung häufig eingesetzter Algorithmus ist die „Schnelle Fourier-Transformation" oder *Fast Fourier-Transformation (FFT)*. Dabei handelt es sich um einen Algorithmus zur schnellen Berechnung der *Diskreten Fourier-Transformation (DFT)*. Der Algorithmus erreicht seine Beschleunigung gegenüber der normalen DFT, indem er die Berechnung sich aufhebender Terme vermeidet. Die einzige Einschränkung des Verfahrens ist, dass die Anzahl der Abtastwerte (Stützstellen) eine Zweierpotenz sein muss. Da die Anzahl der Messwerte in der Praxis jedoch meist beliebig ist, ist das keine bedeutende Einschränkung.

Die Entwicklung des Algorithmus wird J.W. Cooley und J.W. Tukey (1965) zugeschrieben. Das Prinzip beruht darauf, dass der Eingangsvektor der Stützstellen in zwei Hälften aufgeteilt wird: Die eine Hälfte mit Werten mit geraden Indices, die andere Hälfte mit Werten ungerader Indices. Die beiden Hälften werden getrennt berechnet und zum Schluss wieder zu einem Ergebnisvektor zusammengefügt. Dabei können die Eigenschaften der Einheitswurzeln aus der Fourier-Matrix genutzt werden. Der Algorithmus lässt sich wie folgt beschreiben:

$$f_j = \sum_{k=0}^{n-1} x_k \cdot e^{-\frac{2\pi i}{n} jk} \text{ mit } j = 0, 1, ..., n-1 \text{ und } i^2 = -1 \tag{6.4}$$

Gleichung 6.4 zeigt die Definition der DFT f_j, wobei n die Anzahl der Stützstellen bezeichnet. Setzt man nun $n' = n/2$ und bezeichnet mit f' die Fourier-Transformierte des Eingangsvektors der geraden Indices und mit f'' die Fourier-Transformierte des Eingangsvektors der ungeraden Indices, so ergibt sich

$$f_j = \begin{cases} f'_j + e^{-\frac{2\pi i}{n} j} f''_j & \text{falls } j < n' \\ f'_{j-n'} - e^{-\frac{2\pi i}{n}(j-n')} f''_{j-n'} & \text{falls } j \geq n' \end{cases}$$

Mehr über die mathematischen Grundlagen zur Fourier-Transformation findet sich in [52]. Im Folgenden soll der Einfluss der Speicheranordnung auf die Ausführungszeit des FFT-Algorithmus untersucht werden.

Die Untersuchungen stützen sich auf eine Implementierung des Algorithmus der TU Wien. Es wird sowohl die FFT als auch die IFFT (Inverse FFT) betrachtet. Interessant ist hier die Fragestellung, welchen Einfluss die Caches und die TLBs auf die Ausführungszeit des Algorithmus haben. Die Abhängigkeit von der Anzahl der Stützstellen wird durch die Ordnung des Algorithmus beschrieben:

$$\mathcal{O} = n \log_2 n$$

Im Vergleich zum vorherigen Beispiel der Matrixmultiplikation aus Abschnitt 6.4.1 besteht dieser Algorithmus aus wesentlich mehr Code. Für die Untersuchungen wurden jeweils $n = 8192$ Stützstellen als Eingangsdaten verwendet. Die Codegröße beträgt 1870 Byte, das entspricht ca. 30 Cachelines. Die Datenmenge beträgt ca. 258 kB. Der Algorithmus kann von seiner Struktur her die Daten-Caches nicht so gut nutzen wie die Matrixmultiplikation, da die jeweils benutzten Datenobjekte – bedingt durch den Algorithmus – einen größeren Abstand im Speicher haben. Der Code ist linear und enthält Schleifenkonstrukte. Misst man die Ausführungszeit, wenn alle Komponenten im Cache sind, ergeben sich 649,08 μs für die FFT und 650,88 μs für die IFFT.

Misst man im Vergleich dazu die Zeit für den Fall, dass weder Code noch Daten im Cache sind, ergeben sich 716,23 μs für die FFT und 709,36 μs für die IFFT. Dies entspricht einer Laufzeitverlängerung von 10,34% bzw. 8,98%. An diesen Zahlen sieht man, dass der Cache keinen so großen Einfluss auf die Laufzeit des Algorithmus hat wie das beim Beispiel der Matrixmultiplikation der Fall war. Die Laufzeit für den Fall, dass die Daten im Cache sind, aber der Code erst geladen werden muss, beträgt 682,32 μs bzw. 683,43 μs, was einer Laufzeitverlängerung von 5,12% bzw. 5% entspricht. Das ist deutlich weniger als bei der Matrixmultiplikation, was daran liegt, dass der Code auch im optimalen Zustand viele Cachelines benötigt und daher vom Cache nicht so gut profitieren kann.

Untersucht man die Abhängigkeit des Algorithmus von den Daten, so ergibt sich folgendes Bild: Sind die Daten nicht im Cache und die verdrängten Daten müssen erst zurückgeschrieben werden, ergibt sich eine Laufzeit von 692.48 μs bzw. 684,68 μs, müssen die verdrängten Daten nicht zurückgeschrieben werden, ergeben sich 672.88 μs bzw. 683,23 μs. Man sieht, dass die Laufzeitverlängerungen durch Instruktionen- und Daten-Cache-Misses in derselben Größenordnung liegen. Auch hier zeigt sich, dass der Einfluss des Instruktionen-Caches im Vergleich zum Daten-Cache sehr erheblich ist, da es sich bei der FFT um wesentlich weniger Code als Daten handelt, die Einflüsse auf die Laufzeit aber nahezu dieselben sind.

Der Einfluss der TLBs ist, wie schon beim Beispiel der Matrixmultiplikation gesehen, sehr gering: Die Ausführungszeit der FFT erhöht sich auf 650,24 μs, die der IFFT auf 652,46 μs. Dies entspricht einer Steigerung von 0,18% bzw. 0,24%, wobei das Programm in der vorliegenden Implementierung einen Eintrag für Code und 37 Einträge für Daten benötigt. Diese Werte erklären sich dadurch, dass immer nur ein halber Ergebnisvektor berechnet wird. Tabelle 6.7 fasst die Ergebnisse der Messungen zusammen.

Messung	FFT			IFFT		
	Zeit in μs	Steigerung	Jitter	Zeit in μs	Steigerung	Jitter
im Cache	649,08	-	0,55%	650,88	-	0,55%
nicht im Cache	716,23	10,34%	0,28%	709,36	8,98%	1,49%
Code nicht im Cache	682,32	5,12%	1%%	683,43	5%	1,29%
Daten n.i.C., ohne WB	672,88	3,67%	1,62%	683,23	5%	2,1%
Daten n.i.C., mit WB	692,48	6,69%	1,2%	684,68%	5,19%	1,1%
ohne TLBs	650,24	0,18%	0,05%	652,46%	0,24%	0,08%
1 Vektor n.i.C.	672,26	3,57%	0,18%	674,19	3,58%	0,3%

Tabelle 6.7: Zusammenfassung der Messergebnisse

Die Messungen zeigen den Einfluss von Instruktionen- und Daten-Cache auf die Laufzeit des FFT-Algorithmus. Wie schon bei der Matrixmultiplikation fällt der vergleichsweise große Einfluss des Instruktionen-Cache auf. Die Unterschiede bei den Verdrängungen aus dem Daten-Cache zeigen, je nachdem, ob zuvor ein Schreibzugriff notwendig ist oder nicht, die Notwendigkeit zwischen beiden Arten der Verdrängung zu unterscheiden. Des Weiteren rechtfertigen sie auch das Ziel der Optimierung aus Kapitel 5, die Anzahl der Verdrängungen mit

Zurückschreiben zu minimieren. Der Einfluss der TLBs ist vergleichsweise gering und kann im Vergleich mit den Einflüssen von Instruktionen- und Daten-Cache vernachlässigt werden.

Die Ergebnisse zeigen auch, dass es sich lohnt, für einen Algorithmus, der mit vielen Daten arbeitet, zumindest den Code im Cache zu haben. Dies ist oft relativ einfach zu realisieren, wobei es bei großen Datenmengen meist unmöglich ist, alle benötigten Daten im Cache zu haben. Bei den Ergebnissen aus Tabelle 6.7 muss wie beim Beispiel der Matrixmultiplikation beachtet werden, dass die Ausführungszeiten für einen *ungestörten* Zugriff auf den Hauptspeicher gemessen wurden. Verwendet man wieder die Abschätzung für die Laufzeiten bei gestörten Hauptspeicherzugriffen (Fall 1 und Fall 2) aus Abschnitt 6.4.1, so ergeben sich die Werte aus Tabelle 6.8.

Szenario	FFT		IFFT	
	Steigerung	Jitter	Steigerung	Jitter
Fall 1				
nicht im Cache	601%	535%	599%	541%
Daten n.i.C., ohne WB	302%	287%	301%	282%
ohne TLBs	4,72%	4,54%	4,7%	4,51%
Fall 2				
nicht im Cache	179%	153%	179%	156%
Daten n.i.C., ohne WB	90%	84%	90%	81%
ohne TLBs	1,42%	1,24%	1,42%	1,17%

Tabelle 6.8: Abgeschätzte Werte für gestörte Hauptspeicherzugriffe (Fall 1 und Fall 2)

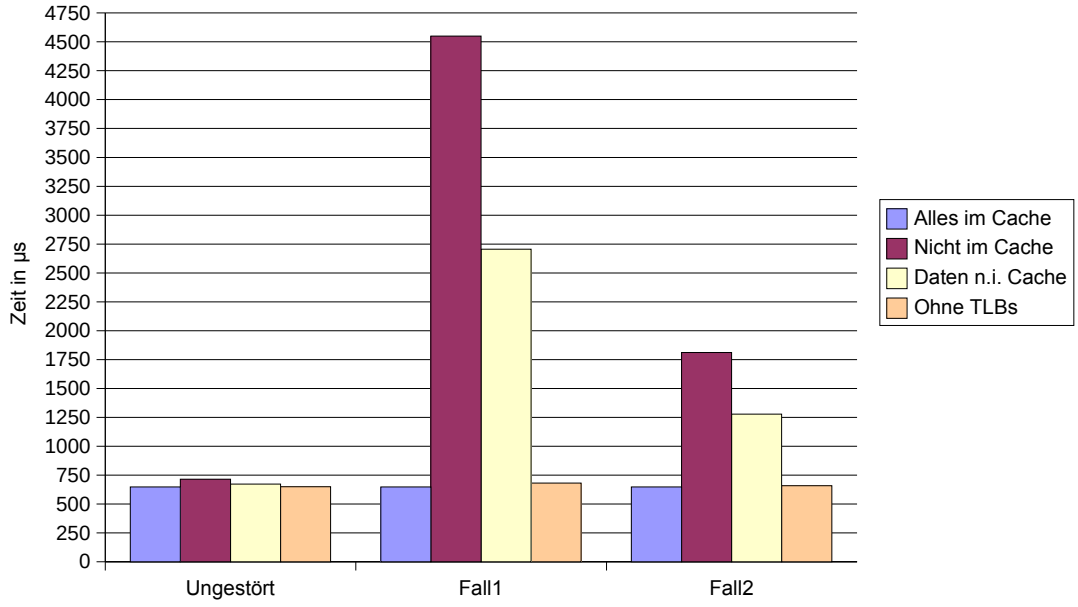

Bild 6.14: Ausführungszeiten für die FFT

Bild 6.14 stellt die Ausführungszeiten für die FFT unter verschiedenen Laufzeitbedingungen nochmals grafisch nebeneinander. Man sieht die deutlichen Laufzeitunterschiede zwischen der Ausführung mit ungestörten Speicherzugriffen und jener mit gestörten Zugriffen (Fall 1 und Fall 2).

Auch in diesem Beispiel sieht man, wie abhängig die Ausführungszeiten von den Hauptspeicherzugriffen sind. Da das Beispielprogramm die Caches nicht so gut nutzen kann wie die Matrixmultiplikation, fallen die Steigerungsraten für die Ausführungszeiten und den Jitter hier auch geringer aus. Die relativ starke Abhängigkeit der Ausführungszeit vom Instruktionen-Cache tritt bei gestörten Hauptspeicherzugriffen nicht mehr auf, da die Anzahl der Hauptspeicherzugriffe für den Zugriff auf Daten deutlich größer ist.

6.5 Was bringt die Speicheroptimierung?

Eine Frage, die sich jedem Anwender stellt, ist welchen Nutzen die in Kapitel 5 vorgestellte Speicheroptimierung für eine Applikation haben kann. Ein weiterer wichtiger Punkt ist das Kosten/Nutzen-Verhältnis der Optimierung, das heißt, wieviel Aufwand muss man für die Realisierung der Optimierung treiben und wieviel Nutzen bringt das für ein konkretes Anwendungsszenario. Weitere Fragestellungen, denen in diesem Abschnitt nachgegangen werden soll, beziehen sich auf die Anwendbarkeit dieser Strategien für andere Prozessorarchitekturen als den konkret untersuchten, und dem Einsatz der Optimierung für Systeme, die mit Hilfe eines Realzeitbetriebssystems realisiert sind.

Nutzen der Speicheroptimierung

Die wesentlichen Vorteile der Speicheroptimierung, wie sie in Kapitel 5 vorgestellt wurde, sind die bessere Vorhersagbarkeit des Laufzeitverhaltens des Systems und die Erhöhung der Ausführungsgeschwindigkeit der Software im Allgemeinen. Dies gilt ganz unabhängig davon, auf welcher Architektur die Optimierung vorgenommen wird. Diese Vorteile werden umso deutlicher, je mehr Geräte in einem System auf einen gemeinsamen Speicher zugreifen müssen.

Der größte Zuwachs an Geschwindigkeit und Determinismus im Laufzeitverhalten ergibt sich, wenn die Software komplett im Cache ist. Dies kann durch eine geschickte Anordnung der Software im Speicher erreicht werden. Ist dies nicht möglich, können dennoch Teile der Software dauerhaft im Cache gehalten werden, welche dann mit geringem Jitter und schnellen Laufzeiten ausgeführt werden können. Dies ist insbesondere für Interrupt-Service-Routinen interessant.

Haben nicht alle Bestandteile der Software Platz im Cache, lassen sich Verdrängungen nicht vermeiden. Wie die Ergebnisse der vorangegangenen Abschnitte gezeigt haben, sind nicht alle Cache-Misses gleich in Bezug auf die Laufzeit von Software zu beurteilen. Man muss prinzipiell zwischen Misses im Instruktionen- und im Daten-Cache unterscheiden. Beide Caches können unabhängig voneinander optimiert werden. Die Messungen haben gezeigt, dass der Instruktionen-Cache einen erheblichen Einfluss auf die Laufzeit hat, auch wenn im Vergleich

zu den Daten nur sehr wenige Cachelines betroffen sind. Umgekehrt hat sich gezeigt, dass die Optimierung der Daten bei Programmen mit kleinen Datenmengen wesentlich mehr ins Gewicht fällt, als bei Programmen mit sehr großen Datenmengen. Auch der Unterschied, ob eine Daten-Cacheline mit oder ohne vorheriges Zurückschreiben verdrängt wird, spielt eine Rolle.

Wieviel Nutzen eine konkrete Applikation aus der Nutzung der Caches ziehen kann, hängt ganz wesentlich von der Struktur der Speicherzugriffe ab. Prinzipiell können Programme, die die meiste Zeit in einem bestimmten Bereich des Speichers arbeiten, am meisten von den Caches profitieren (örtliche Lokalität). Dies hat man am Beispiel der Matrixmultiplikation gesehen, die vom Code her nur aus einer kleinen Schleife besteht. Ist der Code nicht im Cache, sind relativ große Verzögerungszeiten zu beobachten. Sind die Programme eher seriell aufgebaut, können sie keinen großen Nutzen aus den Caches ziehen, das heißt, eine Optimierung an dieser Stelle hat keinen großen Einfluss auf das Laufzeitverhalten des Programms.

Wie in Abschnitt 6.3 gezeigt, gibt es aber dennoch einen Unterschied zwischen einem seriellen Zugriff auf Speicherstellen, die in den Cache geladen werden, und demselben Zugriffsmuster auf Speicherbereiche, die als *uncacheable* markiert sind. Die Zugriffe, die in den Cache geladen werden, sind schneller, da sie von Prefetching-Effekten profitieren können. Beim Zugriff auf Instruktionen werden immer zwei Cachelines geladen, so dass auch bei einem seriellen Zugriff von den schnelleren Zugriffszeiten im Cache profitiert werden kann. Etwas weniger ausgeprägt ist dieser Effekt beim Zugriff auf Daten, da hier immer nur eine Cacheline geladen wird. Allerdings bieten moderne Prozessoren die Möglichkeit, einen Daten-Prefetch in Software zu programmieren. Sie stellen dazu spezielle Befehle zur Verfügung, mit denen ein gezieltes Prefetching möglich ist. Kennt die Software das Zugriffsmuster auf ihre Daten, kann sie den Cache effektiv einsetzen.

Je besser ein Programm die Caches nutzen kann, desto höher wird der Nutzen einer Speicheroptimierung ausfallen. Am Beispiel der Matrixmultiplikation wurde eine Verbesserung der Laufzeit mit Caches von ca. 40% gemessen. Der Code kann den Cache hier optimal nutzen, auch die Daten können dies fast optimal. Vergleicht man dies mit dem Beispiel der FFT, die den Cache bedingt durch ihre Programmstruktur nicht so gut nutzen kann, ergibt sich eine Laufzeitverringerung von nur ca. 10%. Bei der Interpretation dieser Ergebnisse muss man sich aber immer vor Augen führen, dass diese ohne Beeinflussungen durch externe Peripherie ermittelt wurden. Kommen parallele Zugriffe auf den Hauptspeicher hinzu, steigen die Verzögerungszeiten durch die Wartezeiten beim Speicherzugriff stark an. Die verwendete Abschätzung liefert hier Werte von bis zu Faktor 67 für die Matrixmultiplikation und Faktor 7 für die FFT.

Dieselbe Überlegung gilt auch für den Jitter der Ausführungszeiten. Neben der reinen Ausführungszeit ist auch der Jitter eine aus Sicht eines Realzeitsystems wichtige Größe. Die gemessenen Jitter bei einer Ausführung im Cache sind sehr klein und unabhängig davon, was externe Peripheriegeräte zur selben Zeit tun. Kommt es zu Verdrängungen und damit zu Speicherzugriffen, sieht das ganz anders aus: Die in den beiden vorangegangenen Beispielen gemessenen Jitter sind relativ klein, da die Hauptspeicherzugriffe ungestört sind. Sind sie das nicht, ergeben sich erheblich größere Jitter. Die Abschätzung von Stohr liefert hier Werte von bis zu 4732% für die Matrixmultiplikation und bis zu 541% für die FFT.

Um für eine konkrete Applikation einen Anhaltspunkt zu bekommen, ob sich eine Optimierung lohnen könnte, kann man die Ausführungszeit dieser Applikation im Cache messen (sofern möglich) und anschließend die Ausführungszeit, wenn sowohl Instruktionen- als auch Daten-Cache (incl. TLBs) vor der Messung mit unbrauchbaren Objekten gefüllt wurden. Die Differenz zwischen beiden Messungen ergibt die Größenordnung der Laufzeitverbesserungen, die bei einer Speicheroptimierung maximal zu erwarten ist. Oft hat man die Situation, dass man eine Applikation mit vergleichsweise wenig Code aber viel Daten hat. Hier lohnt es sich, zunächst den Code im Speicher optimal anzuordnen, wie das Beispiel der Matrixmultiplikation gezeigt hat. Bei den Daten ist es sinnvoll, jene Objekte anzuordnen, auf die oft zugegriffen wird auf Kosten jener, die von Haus aus nicht viel vom Cache profitieren können. Man kann dann mit relativ wenig angeordneten Objekten im Cache hohe Laufzeitgewinne erzielen.

Aufwand einer Speicheroptimierung

Der Aufwand, den man zur Realisierung einer bestimmten Anordnung von Code- und Datenobjekten im Speicher treiben muss, ist nicht ganz unerheblich. Es stellt sich die grundlegende Frage, welche Objekte vorhanden sind und wo sie platziert werden sollen. Die Identifikation der Objekte kann aus dem *Objectcode* automatisiert erfolgen. Der *Objectcode* ist der (target-spezifische) Maschinencode, den ein Compiler bzw. Linker erzeugt. Aus einer Objectcode-Analyse lassen sich die gesuchten Objekte inklusive ihrer Größe und auch ihrer Attribute ermitteln.

Die Frage, wo die Objekte platziert werden sollen, lässt sich allgemein nicht beantworten, es kommt auf die Programmstruktur an. Grundlegend gelten jedoch die Regeln, die in Abschnitt 5.2 formuliert wurden. Dieses Verfahren lässt sich durch entsprechende Software automatisieren, so dass der Aufwand für die Optimierung selbst nicht sehr hoch ist. Kennt man den Applikationscode sehr gut, kann man auch ein manuelles Eingreifen vorsehen, da nicht jedes spezielle Einsatzgebiet von dem Verfahren aus Kapitel 5 abgedeckt werden kann.

Die Realisierung einer Anordnung im Speicher ist in Bild 6.15 dargestellt.

Einzelne Objectcode-Dateien, wie sie ein Compiler erzeugt, werden mit Hilfe des Linkers zu einer Objectcode-Datei zusammengefasst. Diese Datei wird analysiert, und die erforderlichen Umstellungen innerhalb dieser Datei werden dann wiederum vom Linker erledigt. Das Ergebnis ist eine Objectcode-Datei, die dann zur Ausführung gebracht werden kann. Es sind keine Änderungen an Standardtools wie Compiler oder Linker notwendig. Es ändert sich auch nichts am Verfahren des Software-Entwurfs.

Das Verfahren, wie es in Bild 6.15 dargestellt ist, wurde mit Hilfe des *gcc* und des GNU-Linker *ld* realisiert. Zur Manipulation einer Objectcode-Datei wurde ein Tool *rcmc (RECOMS*[2)] *Coloured Memory Creator)* entwickelt, zur Analyse der Object-Datei und der Umsetzung der Speicheroptimierung das Tool *mmo (Main Memory Optimizer)*. Um das fertig arrangierte Executable in den Speicher laden und starten zu können, wurde ein spezieller Treiber entwickelt,

[2)] *REaltime with Commercial Off-the Shelf Multiprocessor Systems*, ein von der DFG gefördertes Projekt, in dessen Rahmen wesentliche Teile dieser Arbeit entstanden.

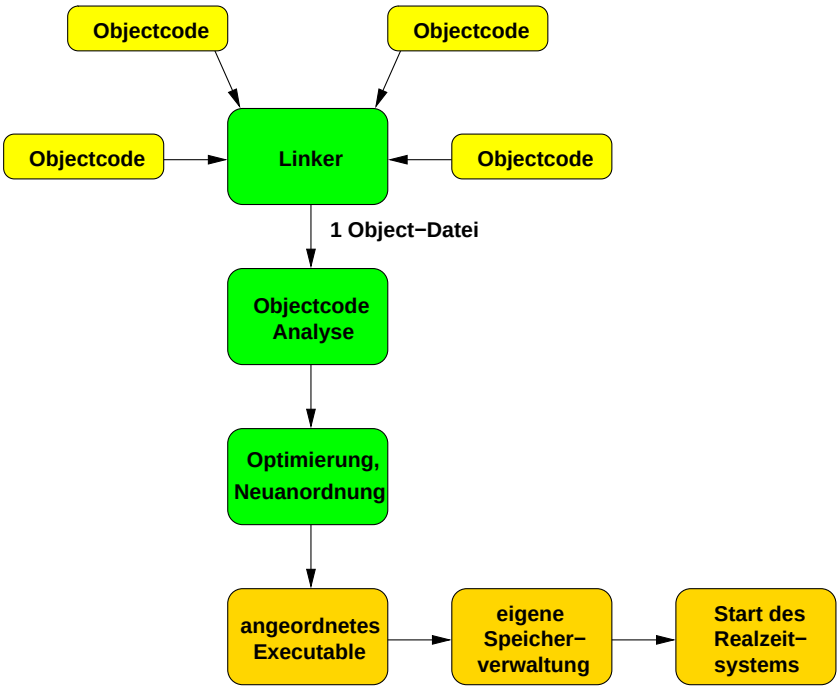

Bild 6.15: Toolkette zur Anordnung von Code und Daten im Speicher

der auch dafür sorgt, dass Speicherbereiche, die nicht im Executable enthalten sind (Stack, shared memory) den Vorgaben entsprechend angelegt werden. Als Betriebssystem wurde Linux mit der Echtzeiterweiterung RTAI verwendet.

Der Aufwand zur Umsetzung einer optimierten Speicheranordnung ist zum einen die Entwicklung geeigneter Tools wie *rcmc* und *mmo* unter Einbeziehung einer Standardtoolkette. Zum anderen muss die Möglichkeit bestehen, in das Speichermanagement des Betriebssystems soweit eingreifen zu können, dass man sich für das Realzeitsystem einen eigenen, physikalischen Speicherbereich reservieren kann, über den man dann mittels eines eigenen Treibers frei verfügen kann.

Die architekturspezifischen Teile der Umsetzung liegen in der Analyse des Objectcodes und der Ermittlung der relevanten Parameter der Cache-Architektur. Letztere können von den meisten Prozessoren mit Hilfe einer speziellen Instruktion (*cpuid* bei Intel-kompatiblen Prozessoren) abgefragt werden. Die Analyse des Objectcodes muss dem Befehlssatz der jeweiligen Architektur angepasst werden. Hat man eine geeignete Toolkette zur Verfügung, läuft der Prozess der Optimierung weitgehend automatisch ab.

Einsatz der Optimierung auf anderen Architekturen

Das Ziel der Optimierung ist eine effiziente Nutzung der Caches. Das bedeutet zum einen, dass der zur Verfügung stehende Speicher im Cache ausgenutzt werden soll und zum anderen, dass nicht zu vermeidende Verdrängungsvorgänge in vorhersagbarer Art und Weise ablaufen. Das

heißt, es soll bekannt sein, ob nur eine Cacheline gelesen werden oder zusätzlich auch eine Cacheline geschrieben werden muss. Ebenso soll es möglich sein, Teile des Caches dafür zu nutzen, Code oder Datenobjekte permanent zu speichern.

Letztere Eigenschaft, das sogenannte Cache-Locking, wird von einigen Prozessorarchitekturen in Hardware unterstützt. Das heißt, man kann bestimmte Speicherbereiche im Cache per Software halten, ohne dies bei der Hauptspeicherbelegung berücksichtigen zu müssen. Die meisten Prozessoren unterstützen Set-assoziative Caches, wie sie in Kapitel 5 verwendet wurden. Die Architektur eines *direct mapped* Caches wird oft für kleinere Caches verwendet, lässt sich aber im Prinzip genauso handhaben wie die der Set-assoziativen Caches. Der Algorithmus zur Anordnung kann mit den entsprechenden Parametern wie Cachegröße, Cacheline-Größe oder Set-Assoziativität angewendet werden.

Neben den physikalisch indizierten Caches gibt es auch virtuell indizierte Caches. Der einzige Unterschied besteht darin, dass die virtuelle Adresse statt der physikalischen verwendet wird, um ein Objekt im Cache zu platzieren. Bei der Vorgehensweise der Optimierung ändert sich nichts, nur die Umsetzung der Optimierung muss angepasst werden. Es muss keine Anordnung innerhalb einer Object-Datei erfolgen, sondern die virtuelle Speicherverwaltung muss die gewünschte Abbildung realisieren.

Es gibt auch Systeme, die keinen Instruktionen-Cache im eigentlichen Sinn haben, sondern einen sogenannten *Trace-Cache*. Der wesentliche Unterschied ist hier, dass nicht mehr einzelne Speicherbereiche in Form von Cachelines im Cache abgelegt werden, sondern die sogenannten Traces abgespeichert werden. Als Trace wird eine Folge von Mikrobefehlen bezeichnet, die von den Dekodern des Prozessors aus einem Assemblerbefehl erzeugt wird. Es wird also kein Speicherbereich im Cache abgelegt, sondern dekodierte Befehle. Der Sinn dabei ist, dass man sich die wiederholte Dekodierung von Befehlen im Cache sparen möchte. Aus der Sicht der Speicheroptimierung ändert sich dabei allerdings nichts. Die Bereiche, die normalerweise direkt im Cache abgelegt werden würden, werden nun eben in Form von Traces abgelegt. Das einzig Schwierige dabei ist, dass man das Fassungsvermögen des Trace-Caches schwer abschätzen kann. Angaben hierzu werden vom Hersteller nicht gemacht, auch Angaben, wie der Trace-Cache intern aufgebaut ist und wie Verdrängungen gehandhabt werden, sind nicht zu finden.

Berücksichtigung des Umfelds einer Realzeitapplikation

Grundsätzlich müssen alle Komponenten eines Systems in die Optimierung mit einbezogen werden. Dies gilt sowohl für Speicherbereiche zur Interprozesskommunikation (IPC), als auch für Teile des Betriebssystems. Dienste des RTOS, wie beispielsweise der Scheduler, sind genauso in die Optimierung mit einzubeziehen wie die Applikation selbst.

Gerade die Berücksichtigung eines RTOS spielt eine große Rolle, da gerade hier große Beeinflussungen mit den Applikationen auftreten können. Häufig genutzte Funktionen wie der Scheduler oder Mechanismen der IPC müssen unbedingt mit berücksichtigt werden. Sie müssen als Teile der Applikation gesehen werden. Auch Speicherbereiche, die von externen Geräten als Ziel für DMA-Transfers benutzt werden, müssen bei der Optimierung mit berücksichtigt wer-

den. Sie sind im Prinzip wie Speicherbereiche zur IPC zu behandeln. Ein Beispiel hierfür wäre der kontinuierliche Transfer von Bilddaten einer mittels FireWire angebundenen Kamera.

Ändern sich die Bedingungen im Umfeld des Realzeitsystems, so müssen diese Änderungen mit berücksichtigt werden. Dazu ist eine neue Optimierung erforderlich. Ein dynamisches Hinzufügen von Komponenten ist nicht ohne Weiteres möglich, da sich dadurch das ganze System ändern kann. Das bedeutet, dass alle Parameter des Systems vor einer Optimierung bekannt sein müssen und sich diese zur Laufzeit des Systems nicht ändern dürfen.

7 Zusammenfassung

In dieser Arbeit wurde ein Algorithmus zur effizienten Nutzung von Caches moderner Prozessoren für Realzeitsysteme vorgestellt. Ziel ist es, die Ausführungszeiten von Software zu minimieren und die Schwankungen in der Laufzeit (Jitter) zu begrenzen. Dadurch wird es möglich, die Worst-Case Execution Time (WCET) auf diesen Prozessoren zu ermitteln. Die Untersuchungen in dieser Arbeit wurden auf AMD-Athlon und AMD-Opteron Prozessoren durchgeführt, stellvertretend für die Familie Intel-kompatibler Prozessoren. Der AMD-Opteron ist ein Vertreter der neuen 64-Bit Architekturen, der aber voll rückwärts kompatibel zu den bisherigen 32-Bit Prozessoren ist.

Die Komponenten der Architektur dieser Prozessoren wurden im Hinblick auf ihren Einfluss auf die Laufzeit von Software untersucht. Gegenstand der Untersuchungen waren die out-of-order execution mit Hilfe von Pipelines, in deren Zusammenhang auch der Tomasulo-Algorithmus näher beleuchtet wurde. Ein weiterer wichtiger Bestandteil der Architektur moderner Prozessoren ist die Branch Prediction, deren Funktionsweise und Auswirkungen ebenfalls untersucht wurden. Im Vergleich zu den Einflüssen der Caches auf die Laufzeit sind die Auswirkungen der Branch Prediction und der out-of-order execution zu vernachlässigen. Die Einflüsse der Caches und der TLBs wurden mit und ohne Einflüsse externer Störungen dargelegt. Diese Beobachtungen motivieren den Ansatz zur Optimierung der Speicherbelegung, um die Caches für Realzeitsysteme nutzen zu können.

Um präzise und zuverlässig Laufzeiten von Software ermitteln zu können, wurde eine Methodik entwickelt, diese auf dem Zielsystem messen zu können. Diese Vorgehensweise basiert auf den Ideen, die Stefan Petters in seiner Dissertation [56] vorgestellt hat. Dieser Ansatz wurde weiterentwickelt, gerade im Hinblick auf Multiprozessorsysteme und die Genauigkeit der Messung. Es ist möglich, die Ausführungszeit nahezu ohne Beeinflussung durch die Messroutine zu bestimmen oder wahlweise vor der Messung bestimmte Störungen zu erzeugen, wie beispielsweise unbrauchbare Daten im Cache oder ungültige Einträge in den TLBs. Zusätzlich können die PMCs der Prozessoren genutzt werden, um die Messergebnisse besser interpretieren zu können.

Es hat sich gezeigt, dass die Laufzeit von Software sehr stark schwanken kann, wenn es zu Verdrängungen im Cache kommt. Um nun moderne Prozessoren in Realzeitsystemen nutzen zu können, wurde ein Algorithmus vorgestellt, der die Caches effizient nutzt. Können Verdrängungen nicht vermieden werden, so ist zumindest bekannt, wie viele Verdrängungen auftreten, wo im Programmkontext sie auftreten und welcher Art sie sind (Code, Daten, mit oder ohne Zurückschreiben). Dadurch lässt sich die WCET einer Realzeitapplikation zuverlässiger abschätzen.

Das Prinzip beruht darauf, die Software in Code- und Datenobjekte zu gliedern. Jedes Objekt bekommt ein Attribut zugewiesen (read-only, read-write oder locked) und wird dementsprechend im Speicher platziert. Es dürfen immer nur Objekte gleichen Attributs in ein Set im Cache abgebildet werden. Damit ist für jede mögliche Verdrängung bekannt, um welche Art der Verdrängung es sich handelt. Code- und Datenobjekte werden dabei getrennt behandelt. Objekte mit dem Attribut „locked" dürfen nie aus dem Cache verdrängt werden. Dies lässt sich durch eine entsprechende Anordnung im Hauptspeicher realisieren. Dies ist wichtig für Prozessoren, die diese Möglichkeit nicht in Hardware unterstützen.

Es wurde auch das Umfeld von Realzeitapplikationen behandelt. Oft wird ein Realzeitbetriebssystem eingesetzt, welches bei der Speicheranordnung als Teil des Gesamtsystems mit berücksichtigt werden muss. Wichtig aus der Sicht der Speicherverwaltung sind auch Speicherbereiche, die für den Austausch von Daten zwischen Realzeit-Tasks verwendet werden. Diese Bereiche müssen bei der Anordnung ebenso mit einbezogen werden wie Speicherregionen, in die Daten mittels DMA-Transfers der Peripherie abgelegt werden. In diesem Zusammenhang wurden verschiedene Strategien für den Zugriff auf den Speicher analysiert.

Um die Auswirkungen der vorgestellten Verfahren untersuchen zu können, wurde auf Mehrprozessorsysteme auf PC-Basis zurückgegriffen. Der Grundgedanke ist, einen Prozessor als GPU (General-Purpose Unit) für ein Standardbetriebssystem zu verwenden und alle weiteren Prozessoren als RTUs (Real-Time Unit) exklusiv für Realzeitanwendungen zu verwenden. Mit dieser Architektur ist es möglich, Anwendungen in einen Realzeit- und einen Nicht-Realzeit Anteil aufzuspalten. Dies kann beispielsweise für Datenerfassungssysteme nützlich sein, bei denen die eigentliche Datenerfassung unter Realzeitbedingungen erfolgen muss, die Auswertung, Visualisierung oder Speicherung der Daten aber keine Realzeitanforderungen stellt.

Im Vergleich zu einem Einprozessorsystem gibt es durch die zusätzlichen Prozessoren bei einem Multiprozessorsystem auf SMP-Basis mehr Beeinflussungen bei einem Zugriff auf den gemeinsamen Speicher. Bei einem NUMA-System treten diese Beeinflussungen durch die physikalisch getrennten Speicher für jeden Prozessor nicht mehr auf. Auch wenn ein Prozessor parallel zu dem lokalen Prozessor in den Speicher schreibt, sind die Auswirkungen nicht so deutlich wie bei SMP-Systemen. Mehrprozessorsysteme ermöglichen es, Beeinflussungen des GPOS und des RTOS im Prozessor zu vermeiden. Durch eine entsprechende Trennung der Objekte im Speicher kommt es zu keinen gegenseitigen Verdrängungen in den Caches. Auch die gegenseitigen Beeinflussungen der Realzeit-Tasks untereinander oder jene, die durch die Unterbrechung von Interrupts verursacht werden, lassen sich minimieren.

Unter dem Gesichtspunkt der Speicheranordnung wurde auch die Verteilung von Tasks auf verschiedene RTUs untersucht. Der Mechanismus des Cache-Snooping, welcher dafür sorgt, dass die Inhalte aller Caches in einem Multiprozessorsystem konsistent sind, wurde ebenso analysiert. Es wurde eine Methode aufgezeigt, wie dieser Mechanismus vorteilhaft zum Datenaustausch zwischen zwei Realzeit-Tasks genutzt werden kann. Die Durchführung eines Realzeitnachweises unter Einbeziehung der vorgestellten Methoden und der daraus gewonnenen Erkenntnisse wurde ebenso behandelt.

Die Auswirkungen einer Optimierung der Speicheranordnung wurden auf einem Quad-Opteron Rechner studiert. Als Anwendungsbeispiele wurde eine Routine zur Matrixmultiplikation und

eine Umsetzung der Fast Fourier-Transformation (FFT) untersucht. Die Ergebnisse zeigen, dass die Anwendungen im Cache sehr schnell und nahezu ohne Jitter ausgeführt werden können. Kommt es zu Verdrängungen im Cache, gibt es große Unterschiede, je nachdem, ob es sich um Verdrängungen im Instruktionen-Cache oder im Daten-Cache mit oder ohne Zurückschreiben handelt. Diese Ergebnisse rechtfertigen die Vorgehensweise des Algorithmus. Die gemessenen Werte wurden alle für einen störungsfreien Zugriff auf den Hauptspeicher ermittelt, daher spiegeln sie auch nur die minimal zu erwartenden Verzögerungszeiten wider. Maximale Verzögerungszeiten treten auf, wenn mehrere Geräte parallel auf den Speicher zugreifen wollen. Zur Abschätzung dieser Verzögerungszeiten und der zu erwartenden Jitter wurde eine Abschätzung aus der Dissertation von Jürgen Stohr [73] verwendet. Die Ergebnisse zeigen deutlich die Abhängigkeit der Ausführungszeiten von Hauptspeicherzugriffen und belegen, wie wichtig eine Speicheroptimierung für ein Realzeitsystem sein kann.

Neben diesen Studien an Applikationen, wie sie in der Praxis verwendet werden, wurden auch die Auswirkungen des Cache-Snooping auf die Laufzeit untersucht. Die Laufzeitverlängerung steigt linear mit der Anzahl der zu aktualisierenden Cachelines. Somit sind die Verzögerungen durch Cache-Snooping gut prädizierbar. Eine weitere Messung hat sich mit den Laufzeitunterschieden zwischen L1- und L2-Cache beschäftigt, die im Wesentlichen für den Jitter einer Anwendung verantwortlich sind, welche in beiden Caches Platz findet. Bei allen Untersuchungen wurden auch die Effekte des Prefetching analysiert, welches dafür sorgt, dass die Laufzeitverzögerungen in der Praxis oft deutlich unter dem zu erwartenden Maximum liegen. Dadurch werden die negativen Folgen durch Cache-Misses zumindest teilweise wieder aufgefangen.

In dieser Arbeit wurde gezeigt, dass es durchaus möglich ist, moderne Prozessoren in Realzeitsystemen einzusetzen. Zentraler Punkt bei der Abschätzung der WCET auf diesen Prozessoren ist die Nutzung der Caches. Die durch Cache-Misses hervorgerufenen Verzögerungszeiten sind bei einem ungestörten Zugriff auf den Hauptspeicher gut vorhersagbar und unterliegen nur einem geringen Jitter. Erst wenn sich mehrere Geräte den Zugang zum Speicher teilen müssen, kann es zu starken Verzögerungszeiten und einem großen Jitter kommen. Es ist daher unabdingbar, immer das Gesamtsystem zu betrachten.

A Messroutine

Die Messroutine, die in Kapitel 4 vorgestellt wurde, besteht aus zwei Teilen: Einem Makro zum Aufruf einer Messung, welches in den Quellcode der Anwendung eingefügt wird, und der Messfunktion *tstrace*. Diese ist in Assembler geschrieben und ist daher für jede Prozessorfamilie (Intel-Pentium, AMD-Athlon, AMD-Opteron) leicht unterschiedlich. Bild A.1 zeigt das Makro zum Aufruf von *tstrace* aus einer Anwendung heraus für die AMD-Opteron Architektur:

```
#define MEAS_FL(nr,fl) \
  asm(" pushfl\n\t" \
      " cli\n\t" \
      " movl  meas_ctrl,%eax\n\t" \
      " andl  $0x7,%eax\n\t" \
      " orl   $"SYMBOL_NAME_STR(fl)",%eax\n\t" \
      " pushl %eax\n\t" \
      " pushl $"SYMBOL_NAME_STR(nr)"\n\t" \
      " call  tstrace\n\t" \
      " addl  $0x8,%esp\n\t" \
      " popfl\n\t" \
      " jmp 1f\n\t" \
      " nop; nop; nop; nop; nop; nop; nop; nop; nop; nop\n\t" \
      " nop; nop; nop; nop; nop; nop; nop; nop; nop; nop\n\t" \
      " nop; nop; nop; nop; nop; nop; nop; nop; nop; nop\n\t" \
      " nop; nop; nop; nop; nop; nop; nop; nop; nop; nop\n\t" \
      " nop; nop; nop; nop; nop; nop; nop; nop; nop; nop\n\t" \
      " nop; nop; nop; nop; nop; nop; nop; nop; nop; nop\n\t" \
      " nop; nop; nop; nop; nop; nop; nop; nop; nop; nop\n\t" \
      " nop; nop; nop; nop; nop; nop; nop; nop; nop; nop\n\t" \
      " nop; nop; nop; nop; nop; nop; nop; nop; nop; nop\n\t" \
      " nop; nop; nop; nop; nop; nop; nop; nop; nop; nop\n\t" \
      " nop; nop; nop; nop; nop; nop; nop; nop; nop; 1:");
```

Bild A.1: Das Makro MEAS_FL

Mit Hilfe des Parameters `fl` gibt man an, ob nur ein Zeitstempel genommen und abgespeichert werden soll, oder ob die Caches mit unnützen Daten bzw. Instruktionen gefüllt werden sollen. Die `nop`-Instruktionen zum Schluss bewirken, dass der nachfolgende Code, dessen Laufzeit gemessen werden soll, nicht vom Prefetching des Prozessors profitiert. Die `nop`-Instruktionen werden nicht ausgeführt, sie füllen nur die bereits angefangene Cacheline und die direkt nachfolgende. Die Messroutine selbst zeigt Bild A.2:

```
tstrace:                                  # interrupts are off on entry
    xor      %rax,%rax                    # serialize
    movq     %rax,%dr0
    rdtsc
    movq     %rax,%r10                    # save lower bits of time stamp
    movq     %rdx,%r11
```

```
    movl     $0xC0010000,%ecx             # select PerfEvtSel0
    rdmsr
    btrl     $0x16,%eax                   # turn it off
    wrmsr
    movl     $0xC0010001,%ecx             # select PerfEvtSel1
    rdmsr
    btrl     $0x16,%eax                   # turn it off
    wrmsr
    movl     $0xC0010002,%ecx             # select PerfEvtSel2
    rdmsr
    btrl     $0x16,%eax                   # turn it off
    wrmsr
    movl     $0xC0010003,%ecx             # select PerfEvtSel3
    rdmsr
    btrl     $0x16,%eax                   # turn it off
    wrmsr

#if defined(CONFIG_SMP)

# Here comes our spinlock

1:  lock     decb meas_spinlock           # atomic decrement
    jns      3f                           # it is zero now, then continue
2:  cmpb     $0,meas_spinlock             # else compare with zero (locked)
    repz                                  # do exactly nothing
    nop
    jle      2b                           # spinlock was zero -> once again
    jmp      1b                           # not zero, lock and continue
3:
#end of spinlock

#endif

# get meas_ctrl and measid from the caller's registers
# this is necessary for sync between GPU and RTU!

    movq     meas_ctrl,%rdx      # this is the actual one, not from stack!!!

    movq     %rdi,meas_ctrl
    movq     %rsi,measid

    movq     meas_ctrl,%rax
    bt       $0x1,%rdx           # bit no. 1, corresponds to M_NO_MEASUREMENT
    jc       1f                  # set -> ok
    bt       $0x1,%rax
    jnc      1f                  # not set -> ok

    bts      $0x1,%rax           # not ok, so we have to sync
    movq     %rax,meas_ctrl

1:  movq     meas_ctrl,%rdx
    andq     $M_NO_MEASUREMENT,%rdx       # should I do the whole procedure?
    jnz      5f                           # no, then return

# set the parameters depending on the CPU we are running on
# the CPU is determined by the APIC-ID

    lea      apic_id,%rax  # we must use lea here, so rax is an address!
    movq     (%rax),%rax   # rax has now the address of the APIC-ID register
    movq     $0xFFFFFFFF00000000,%r8    # bring it to canonical form!
    or       %r8,%rax
    movq     (%rax),%rax                # read the APIC register
    and      $0xff000000,%eax           # determine the ID
    shr      $0x18,%eax
    movl     %eax,cpu_id        # save the cpu id where we are running on

    movq     meascounter,%r8
    incq     %r8
    imul     $ENTRY_LENGTH,%r8  # (measurements+1)*(byte per measurement)->r8
    movq     $RECOMS_ALIGNMENT,%rcx
    imul     $RECOMS_BUFFER,%rcx        # buffer size in bytes ->ecx
    imul     $CONFIG_NR_CPUS,%rcx
    cmpq     %r8,%rcx                   # space left?
    jle      6f                         # no, then abort
```

```
# Store timestamps

    lea      meas_ptr,%r8          # get pointer for data placement in memory
    movq     (%r8),%r8
    movq     $0xFFFFFFFF00000000,%rax
    or       %rax,%r8
    movq     %r10,(%r8)            # move low 32 bit to array (of timestamp)
    movq     %r11,8(%r8)           # move high 32 bit to array
    movq     measid,%rax           # move measurement ID into array
    movq     %rax,16(%r8)

    movq     meas_ctrl,%rdx
    andq     $M_FAST,%rdx
    jz       2f
    movq     cpu_id,%rax           # save cpu_id
    movq     %rax,40(%r8)
    jmp      11f

2:  movl     $0x0,%ecx             # select counter 0
    rdpmc                          # read PMC
    movl     %eax,40(%r8)          # write it to memory
    movl     $0x1,%ecx             # the same for the other counters
    rdpmc
    movl     %eax,48(%r8)
    movl     $0x2,%ecx
    rdpmc
    movl     %eax,56(%r8)
    movl     $0x3,%ecx
    rdpmc
    movl     %eax,64(%r8)

    movq     cpu_id,%rax            # save cpu_id
    movq     %rax,72(%r8)

11: addq     $ENTRY_LENGTH,meas_ptr # set the measurement pointer to next set
    addl     $1,meascounter         # total number of measurements

    movq     meas_ctrl,%rdx
    andq     $M_FAST,%rdx
    jnz      10f
    movq     meas_ctrl,%rdx # should the acceleration units be set to the WC?
    andq     $M_NO_CACHEFLUSH,%rdx
    jnz      4f          # avoiding call so not to disturb return stack buffer
    jmp      8f

# setup the event counters to count certain events

4:
    movq    meas_ctrl,%rdx
    andq    $M_TLB,%rdx
    jz      1f
    movq    %cr4,%rax           # invalidate TLBs, first turn off global pages
    andq    $0xFFFFFFFFFFFFFF7F,%rax
    movq    %rax,%cr4
    movq    %cr3,%rax                # now invalidate TLBs
    movq    %rax,%cr3
    movq    %cr4,%rax                # turn on global pages
    orq     $0x80,%rax
    movq    %rax,%cr4
```

```
1:  movl     $0xC0010000,%ecx        # select PerfEvtSel0
    rdmsr
    andl     $0x00FF0000,%eax        # mask the reserved bits
    movl     unitmask0,%edi          # set the unitmask
    shl      $8,%edi                 # shift it to the right place
    orl      %edi,%eax               # set the according bits in %eax
    movl     pmc_event0,%edi         # the event code itself
    orl      %edi,%eax               # set the according bits in %eax
    bts      $0x10,%eax              # count events in CPL 1,2,3
    bts      $0x11,%eax              # and also in CPL 0
    btr      $0x12,%eax              # no edge detection
    bts      $0x16,%eax              # enable the counter
    wrmsr                            # the event counter starts to count here!

    movl    $0xC0010001,%ecx         # select PerfEvtSel1
    rdmsr
    andl    $0x00FF0000,%eax
    movl    unitmask1,%edi
    shl     $8,%edi
    orl     %edi,%eax
    movl    pmc_event1,%edi
    orl     %edi,%eax
    bts     $0x10,%eax
    bts     $0x11,%eax
    btr     $0x12,%eax              # no edge detection
    bts     $0x16,%eax
    wrmsr

    movl    $0xC0010002,%ecx         # select PerfEvtSel1
    rdmsr
    andl    $0x00FF0000,%eax
    movl    unitmask2,%edi
    shl     $8,%edi
    orl     %edi,%eax
    movl    pmc_event2,%edi
    orl     %edi,%eax
    bts     $0x10,%eax
    bts     $0x11,%eax
    btr     $0x12,%eax              # no edge detection
    bts     $0x16,%eax
    wrmsr

    movl    $0xC0010003,%ecx         # select PerfEvtSel1
    rdmsr
    andl    $0x00FF0000,%eax
    movl    unitmask3,%edi
    shl     $8,%edi
    orl     %edi,%eax
    mov     pmc_event3,%edi
    orl     %edi,%eax
    bts     $0x10,%eax
    bts     $0x11,%eax
    btr     $0x12,%eax              # no edge detection
    bts     $0x16,%eax
    wrmsr

10:
    movq     cycles,%r8
    movq     meascounter,%rcx
    cmpq     %rcx,%r8
    jg       2f
    movq     $M_END,meas_err
    orq      $M_NO_MEASUREMENT,meas_ctrl

2:
# tstrace needs two entries in data TLBs, invalidate them now

    lea     meas_ptr,%r8
    movq    (%r8),%r8
    subq    $ENTRY_LENGTH,%r8
    movq    $0xFFFFFFFF00000000,%rcx
    orq     %rcx,%r8
```

```
    movq    (%rsp),%rdx                 # load data cache lines
    movq    8(%rsp),%rax
    movq    16(%rsp),%rdx
    movq    24(%r8),%rcx
    movq    32(%r8),%rdx

    movq    meas_ctrl,%rdx
    andq    $M_FAST,%rdx
    jnz     10f

# reset the event counters

    xor     %eax,%eax          # Event Counters have to be set to 0 explicitly
    xor     %edx,%edx
    movl    $0xC0010004,%ecx
    wrmsr
    movl    $0xC0010005,%ecx
    wrmsr
    movl    $0xC0010006,%ecx
    wrmsr
    movl    $0xC0010007,%ecx
    wrmsr

10:

#if defined(CONFIG_SMP)

# release the spinlock

    movb     $1,meas_spinlock           # 1 means unlocked

#endif
.align CACHELINE_SIZE

    xorq    %rax,%rax                   # serialize execution
    movq    %rax,%dr0

    rdtsc

    movl     %eax,24(%r8)
    movl     %edx,32(%r8)

    retq

5:  # skip_execution

#if defined(CONFIG_SMP)

# release the spinlock

    movb    $1,meas_spinlock

#endif

    retq                               # just return, because nothing is to do,
                                       # or no more space left

6:  # overflow

    movl     $M_OVERFLOW,meas_err
    orl      $M_NO_MEASUREMENT,meas_ctrl

#if defined(CONFIG_SMP)
    movb     $1,meas_spinlock
#endif

    retq
```

```
8:  # kill_units

        movq    meas_ctrl,%rcx
        andq    $M_DCACHEFLUSH,%rcx
        jnz     2f
        movq    meas_ctrl,%rcx
        andq    $M_DCACHEFLUSH_WB,%rcx
        jnz     2f

        movq    meas_ctrl,%rcx
        andq    $M_CACHEFLUSH,%rcx
        jz      1f

#.align CACHELINE_SIZE

.rept CACHESIZE/16   # 16 is the window size
1:
        jmp  1f
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
.endr

1:      movq    meas_ctrl,%rcx
        andq    $M_ICACHEFLUSH,%rcx
        jnz     4b

2:      movq    $CACHELINES,%rax         # the size in cachelines (L1+L2)
        lea     invalidate_cache_area,%rdx

        movq    meas_ctrl,%rcx
        andq    $M_DCACHEFLUSH,%rcx
        jnz     6f
2:      addq    $0x1,(%rdx)              # touch in data cache
        addq    $CACHELINE_SIZE,%rdx     # goto next cacheline
        subq    $0x1,%rax                # count the cachelines visited so far
        jne     2b                       # repeat for every cacheline
        jmp     4b
6:      movq    (%rdx),%r8               # read cacheline
        addq    $CACHELINE_SIZE,%rdx     # goto next cacheline
        subq    $0x1,%rax                # count the cachelines visited so far
        jne     6b
        jmp     4b
```

Bild A.2: Die Messroutine *tstrace*

B Ergänzungen zur Prozessorarchitektur

In diesem Kapitel werden kurz einige Details zur Umsetzung bestimmter Techniken in Prozessoren der IA-32 Architektur erläutert. Diese sollen dem besseren Verständnis der Ausführungen aus Kapitel 4, 5 und 6 dienen. Zunächst wird kurz die Pipeline des AMD-Athlon Prozessors vorgestellt, dann wird die PLRU-Stratgie kurz erläutert, bevor die wichtigsten Daten bezüglich der Caches einiger Prozesoren zusammengestellt werden. Das Kapitel schließt mit einer kurzen Erläuterung der Adressumrechnung für die Intel-Pentium / AMD-Athlon und AMD-Opteron Prozessoren.

B.1 Pipelines

Bild B.1 zeigt als Beispiel die Integer-Pipeline des AMD-Athlon Prozessors. Pro Taktzyklus des Prozessors wird eine Stufe der Pipeline abgearbeitet. Es gibt einen Zweig für *Direct Path* und einen für *Vector Path* Instruktionen. *Direct Path* Befehle werden direkt in eine MacroOP dekodiert, *Vector Path* Befehle werden durch eine Sequenz von MacroOPs ersetzt, die aus dem MROM des Prozessors entnommen werden.

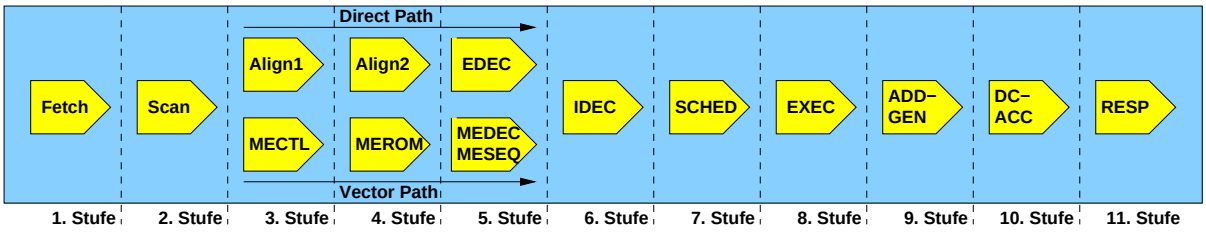

Bild B.1: Integer Pipeline des AMD-Athlon Prozessors (nach [3])

Die einzelnen Stufen der Abarbeitung bedeuten:

- **Fetch:** Lädt die nächsten Befehle (immer 16 Byte) und berechnet die Adresse der folgenden 16 Byte (*Instruction Window*).
- **Scan:** Identifiziert die Instruktionen im aktuellen *Instruction Window*

Direct Path:

- **Align 1:** Transferiert die Instruktionen in einen Puffer.
- **Align 2:** Identifiziert den Opcode incl. Prefixes der Instruktionen im Puffer aus *Align 1*.
- **EDEC (Early Decode):** Dekodiert die Instruktionen und bildet MacroOPs.

Vector Path:

- **MECTL (Microcode Engine Control):** Generiert die Einsprungpunkte für die MROM Dekodierung.
- **MEROM (Microcode Engine ROM):** Mit Hilfe der Einsprungpunkte aus der vorherigen Stufe wird der Microcode im MROM indiziert.
- **MEDEC/MESEQ (Microcode Engine Decode/Sequencer):** Die Instruktionen werden in MacroOPs dekodiert.

Direct Path und Vector Path:

- **IDEC/Rename (Instruction Decoder Rename):** Renaming für die einzelnen Instruktionen (siehe auch Abschnitt 4.2.2).
- **SCHED (Scheduler):** Die Instruktionen, die ausgeführt werden können (alle Operanden vorhanden), werden auf die Ausführungseinheiten verteilt.
- **EXEC (Execution):** Führt den Befehl aus.
- **ADDGEN (Address Generation):** Wenn der Befehl eine Adresse benötigt, wird sie hier berechnet und an die TLBs/Caches weitergeleitet.
- **DCACC (Data Cache Access):** Die Caches/TLBs werden nach der zuvor generierten Adresse durchsucht.
- **RESP (Response)**: Die Daten werden vom Cache geliefert bzw. ein Miss-Status wird gesetzt.

Am Beispiel des AMD-Athlon Prozessors kann man deutlich die Umsetzung des Tomasulo-Algorithmus aus Abschnitt 4.2.2 erkennen. Weitere Details zur Befehlsabarbeitung beim AMD-Athlon Prozessor finden sich in [3].

B.2 Die PLRU-Strategie

In n-fach Set-assoziativen Caches teilen sich n Cachelines ein Set. Ist ein Set bereits mit n Cachelines belegt und es soll eine weitere Cacheline dort abgelegt werden, so muss eine Cacheline aus dem Set verdrängt werden. Diese Entscheidung wird optimalerweise nach der Least-Recently-Used (LRU) Strategie getroffen. Das heißt, die Cacheline, die am längsten nicht mehr benötigt wurde, soll verdrängt werden.

Die Entscheidung, welche Cacheline betroffen ist, muss möglichst schnell getroffen werden, da sonst die Leistungsfähigkeit der Caches eingeschränkt wird. Die LRU-Startegie wäre ziemlich

aufwändig zu implementieren, daher wird meist die PLRU-Strategie (Pseudo-LRU) angewandt, welche die LRU-Strategie annähert. Die PLRU-Strategie soll am Beispiel eines 4-fach Set-assoziativen Caches kurz erläutert werden.

Pro Set werden n-1 Bits benötigt, für n=4 sind dies B0, B1 und B2. In Abhängigkeit der Position einer Cacheline im Set, auf die gerade zugegriffen wird, werden die Bits gesetzt. Bild B.2 zeigt den binären Entscheidungsbaum.

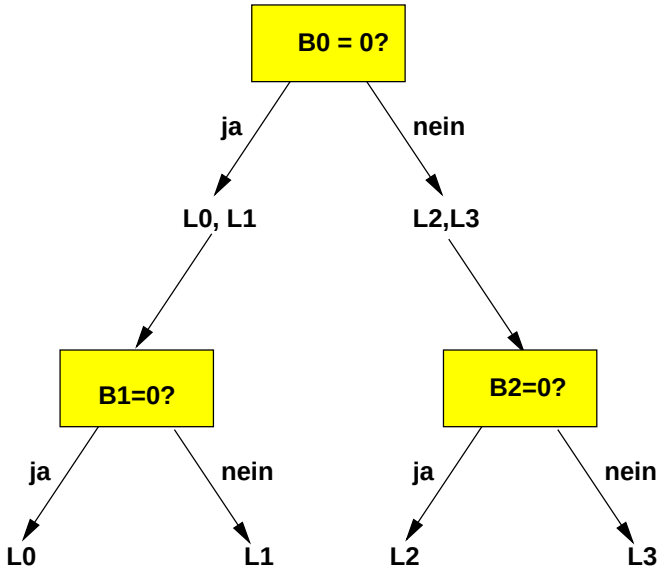

Bild B.2: PLRU-Strategie für n=4

Wird zum Beispiel auf die erste Cacheline (L0) im Set zugegriffen, werden die Bits B0 und B2 auf 0 (false) gesetzt. Soll nun eine Cacheline verdrängt werden, wird diese entsprechend den Bits B0, B1 und B2 bestimmt. Haben diese beispielsweise die Werte B0=1 und B2=1, so wird L3 verdrängt. B1 spielt in dem Fall keine Rolle.

Je höher die Assoziativität des Caches ist, desto aufwändiger wird die Implementierung der PLRU-Strategie nach diesem Schema. Es werden immer n-1 Bits benötigt und der Entscheidungsbaum wächst entsprechend an. Dies kostet Zeit zur Entscheidungsfindung und zum Aktualisieren der jeweiligen Bits, sowie Energie. In [1] wird ein Überblick über verschiedene Verfahren der Cacheverdrängung und deren Effizienz gegeben.

B.3 Daten der Caches von PC-Prozessoren

In diesem Abschnitt werden als Ergänzung die wichtigsten Daten zu den Caches in PC-Prozessoren genannt. Tabelle B.1 zeigt die für den Algorithmus aus Kapitel 5 relevanten Parameter.

Prozessor	L1-Cache				L2-Cache (unified)		
	Typ	Größe (kB)	Ass.	CL (Byte)	Größe (kB)	Ass.	CL (Byte)
Pentium III, Celeron	I-Cache	16	4	32	256, 512	8,16	32
	D-Cache	16	4	32			
Pentium M	I-Cache	32	4	32	1024, 2048	8	32
	D-Cache	32	4	32			
Pentium 4, Xeon	I-Cache	ca. 96, ca. 120	Trace-Cache		256, 512, 1024, 2048	8	128
	D-Cache	8, 16	4	64			
Athlon, Duron	I-Cache	64	2	64	256, 512	16	64
	D-Cache	64	2	64			
Opteron	I-Cache	64	2	64	1024, 2048	16	64
	D-Cache	64	2	64			

Tabelle B.1: Daten der Caches für verschiedene x86-Prozessoren

B.4 Adressumrechnung

Die Adressumrechnung von einer virtuellen in eine physikalische Adresse erfolgt bei den Prozessoren der IA-32 Architektur in zwei Schritten. Bild B.3 zeigt den Vorgang.

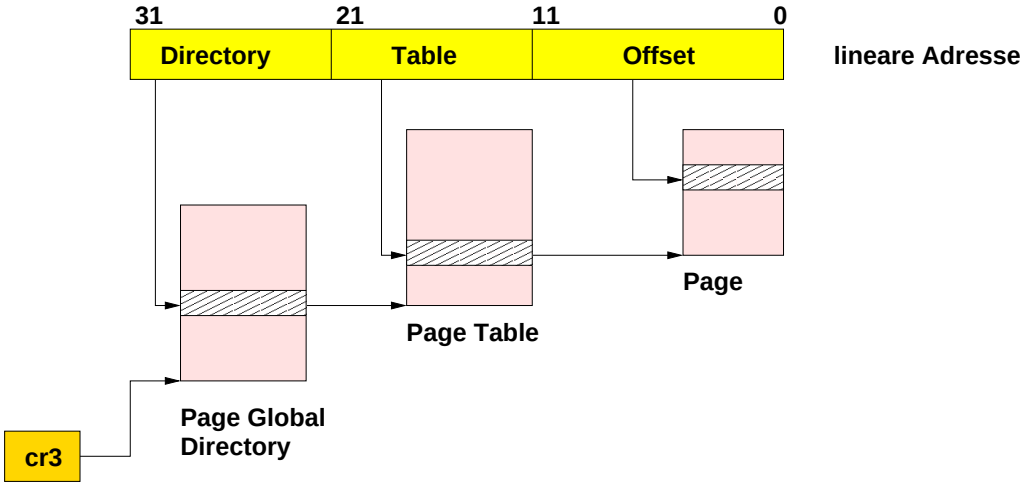

Bild B.3: Adressumrechung bei den Intel-Pentium und AMD-Athlon Prozessoren

Die untersten n Bits werden für den Offset innerhalb einer Page benötigt, wenn die Page-Größe 2^n Byte beträgt (hier $n = 12$ für 4 kB große Pages). Die folgenden zehn Bit adressieren einen

Eintrag im Page Directory, welches wiederum über einen Eintrag des Page Global Directory adressiert wird. Das Page Global Directory selbst wird über das Register cr3 adressiert, der Eintrag wird über die höchsten zehn Bit des Adresswortes ausgewählt.

Für jeden Speicherzugriff muss die virtuelle in die physikalische Adresse umgerechnet werden. Dafür muss jeweils ein Eintrag im Page Global Directory und ein Eintrag im Page Directory referenziert werden. Damit dazu nicht jedesmal zwei Hauptspeicherzugriffe generiert werden müssen, werden diese Einträge in den Translation Lookaside Buffers (TLBs) zwischengespeichert. Diese sind ähnlich wie die Caches aufgebaut. Es gibt einen L1- und einen L2-TLB, der L1-TLB ist wie der L1-Cache in einen TLB für Instruktionen und einen für Daten aufgeteilt. Die TLBs sind so groß, dass im Normalfall alle Instruktionen und Daten, die sich im Cache befinden, über Einträge in den TLBs adressiert werden können. Nähre Details zu den TLBs der einzelnen Prozessoren finden sich in den jeweiligen Handbüchern und können mit Hilfe der *cpuid*-Funktion ermittelt werden.

Beim AMD-Opteron findet dasselbe Schema Anwendung, jedoch sind hier noch zwei weitere Tabellen involviert. Bild B.4 zeigt diese Umrechnung.

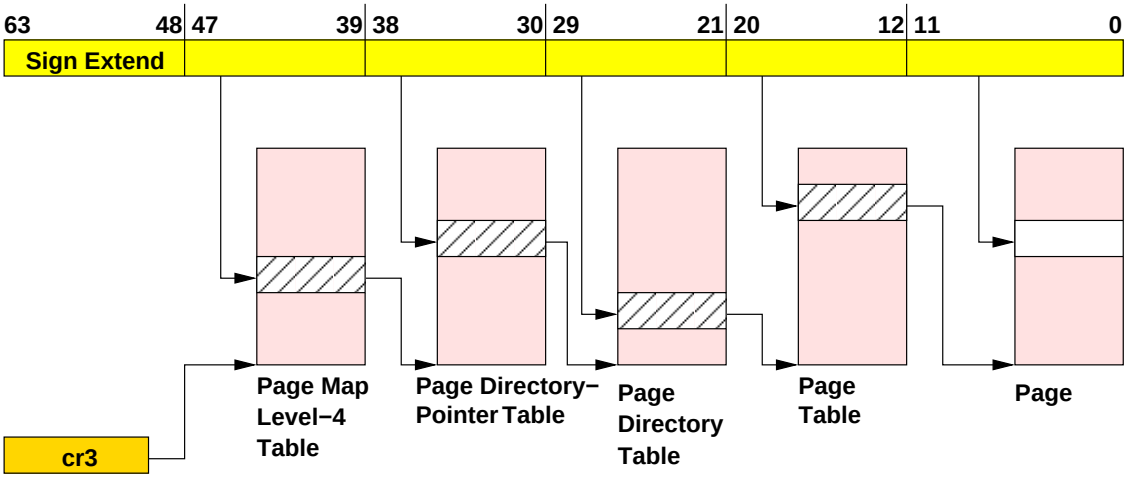

Bild B.4: Adressumrechung beim AMD-Opteron Prozessor

Hier werden für die Adressumrechnung vier Einträge benötigt. Auch hier speichern die TLBs diese Einträge, um langsame Hauptspeicherzugriffe zu verhindern. Bild B.4 zeigt die Umrechnung für eine Seitengröße von 4 kB. Für andere Seitengrößen ändert sich am Prinzip der Umrechnung nichts, es werden nur entsprechend weniger Tabellen benötigt (für 2 MB große Pages fällt z.B. die Page Table weg).

Literaturverzeichnis

[1] AL-ZOUBI, HUSSEIN, ALEKSANDAR MILENKOVIC, and MILENA MILENKOVIC: *Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite*. In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, pages 267–272, New York, NY, USA, 2004. ACM Press.

[2] AMD, One AMD Place, Sunnyvale, CA 94088, USA: *Whitepaper: AMD Athlon Processor and AMD Duron Processor with Full Speed On-Die L2 Cache*, June 2000.

[3] AMD, One AMD Place, Sunnyvale, CA 94088, USA: *AMD Athlon Processor x86 Code Optimization Guide*, February 2002.

[4] AMD, One AMD Place, Sunnyvale, CA 94088, USA: *Whitepaper: The AMD-760 MPX Platform for the AMD Athlon MP Processor*, January 2002.

[5] AMD, One AMD Place, Sunnyvale, CA 94088, USA: *AMD64 Architecture Programmers's Manual Volume 2: System Programming*, September 2003.

[6] ARNAUD, ALEXIS and ISABELLE PUAUT: *Towards a Predictable and High Performance Use of Instruction Caches in Hard Real-Time Systems*. In *Proc. of the Work-in-Progress Session of the 15th Euromicro Conference on Real-Time Systems*, pages 61–64, Porto, Portugal, July 2003.

[7] BATE, I. and R. REUTEMANN: *Worst-Case Execution Time Analysis for Dynamic Branch Predictors*. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, Catania, Italy, June 2004.

[8] BERNAT, G., A. COLIN, and S. M. PETTERS: *WCET Analysis of Probabilistic Hard Real-Time Systems*. In *Proceedings of the 23rd Real-Time Systems Symposium RTSS 2002*, pages 279–288, Austin, Texas, USA, December 2002.

[9] BOVET, DANIEL P. and MARCO CESATI: *Understanding the Linux Kernel*. O'Reilly, Sebastopol, 2nd edition, 2003.

[10] BUDRUK, RAVI, DON ANDERSON, and TOM SHANLEY: *PCI Express System Architecture*. Addison–Wesley Publishing Company, Boston, 2003.

[11] BURMBERGER, GREGOR: *PC–basierte Systemarchitekturen für zeitkritische technische Prozesse*. Doktorarbeit, Lehrstuhl für Realzeit–Computersysteme, Technische Universität München, Februar 2002.

[12] BURNS, ALAN and STEWART EDGAR: *Predicting Computation Time for Advanced Processor Architectures*. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, Stockholm, Sweden, June 19–21 2000.

[13] BÜLOW VON, A., J. STOHR, G. FÄRBER, P. MÜLLER, and J. B. SCHRAML: *Using the RECOMS Architecture for Controlling a Radio Telescope*. Technical report, Institute for Real–Time Computer Systems, Technische Universität München, May 2004.

[14] BÜLOW VON, ALEXANDER: *Bestimmung der WCET auf AMD–Athlon Prozessoren unter Berücksichtigung eines Realzeit–Betriebssystems*. Diplomarbeit, Lehrstuhl für Realzeit–Computersysteme, Technische Universität München, 2001.

[15] BÜLOW VON, ALEXANDER, JÜRGEN STOHR, and GEORG FÄRBER: *Towards an Efficient Use of Caches in State of the Art Processors for Real–Time Systems*. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems – Work in Progress Session*, Catania, Italy, June 2004.

[16] CALDER, B., K. CHANDRA, S. JOHN, and T. AUSTIN: *Cache-Conscious Data Placement*. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, 1998.

[17] CAMPOY, A. MARTÍ, A. PERLES IVARS, and J.V. BUSQUETS-MATAIX: *Using Locking Caches in Preemptive Real-Time Systems*. In *Proceedings of the 12th IEEE Real Time Congress on Nuclear and Plasma Sciencies*, pages 157–159, Valencia, Spain, June 2001.

[18] CAMPOY, A. MARTÍ, A. PERLES IVARS, and J.V. BUSQUETS-MATAIX: *Dynamic Use of Locking Caches in Multitasking, Preemptive Real-Time Systems*. In *Proceedings of the 15th Triennial World Congress*, Barcelona, Spain, 2002.

[19] CAMPOY, A. MARTÍ, A. PERLES IVARS, and J.V. BUSQUETS-MATAIX: *Static Use of Locking Caches in Multitask Preemptive Real-Time Systems*. In *Proceedings of the IEEE Real-Time Embedded System Workshop*, December 2003.

[20] CAMPOY, A. MARTÍ, A. PERLES IVARS, F. RODRÌGUEZ, and J.V. BUSQUETS-MATAIX: *Static Use of Locking Caches vs. Dynamic Use of Locking Caches for Real-Time Systems*. In *Proceedings of the CCECE 2003*, May 2003.

[21] CAMPOY, A. MARTÍ, A. PERLES, S. SÁEZ, and J.V. BUSQUETS-MATAIX: *Performance Analysis of the Static Use of Locking Caches*. In *Proceedings of the 3rd WSEAS Int. Conference on Automation and Information*, Tenerife, Spain, December 2002.

[22] COLIN, A. and S. M. PETTERS: *Experimental Evaluation of Code Properties for WCET Analysis*. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, Cancun, Mexico, December 3–5 2003.

[23] COLIN, ANTOINE and ISABELLE PUAUT: *Worst Case Execution Time Analysis for a Processor with Branch Prediction*. Journal of Real-Time Systems, 18:249 – 274, 2000.

[24] DIAPM, Dipartimento di Ingegneria Aerospaziale Politecnico di Milano: *A Hard Real Time Support for LINUX*, 2002.

[25] ENGBLOM, JAKOB: *Analysis of the Execution Time Unpredictability caused by Dynamic Branch Prediction*. In *Proceedings of the 9th IEEE Real-Time/Embedded Technology and Applications Symposium (RTAS 2003)*, Toronto, Canada, May 2003.

[26] FERDINAND, CHRISTIAN, REINHOLD HECKMANN, MARC LANGENBACH, FLORIAN MARTIN, MICHAEL SCHMIDT, HENRIK THEILING, STEPHAN THESING, and REINHARD WILHELM: *Reliable and Precise WCET Determination for a Real-Life Processor*. In *Proceedings of the Embedded Software First International Workshop (EMSOFT 2001), Tahoe City, CA, USA*, volume 2211, pages 469–485, October 2001.

[27] FERDINAND, CHRISTIAN and REINHARD WILHELM: *Efficient and Precise Cache Behavior Prediction for Real-Time Systems*. Journal of Real-Time Systems, 17:131–181, 1999.

[28] FÄRBER, GEORG: *Prozessrechentechnik*. Springer, Berlin, 3. Auflage, 1994.

[29] GOEBL, MATTHIAS: *Einflüsse der Bussysteme moderner PCs auf das Laufzeitverhalten von Realzeitsoftware*. Diplomarbeit, Lehrstuhl für Realzeit–Computersysteme, Technische Universität München, 2003.

[30] GONZALEZ-SCHILLER, CHRISTIAN: *Realzeiteigenschaften von IDE Festplatten*. Diplomarbeit, Lehrstuhl für Realzeit–Computersysteme, Technische Universität München, 2004.

[31] GOPALAN, K.: *Real–Time Support in General Purpose Operating Systems*, January 2001. Research Proficiency Exam Report.

[32] GÖTZ, SIEGFRIED: *Implementierung einer alternativen Speicherverwaltung für Realzeitsysteme*. Diplomarbeit, Lehrstuhl für Realzeit–Computersysteme, Technische Universität München, 2003.

[33] HASHEMI, AMIR H., DAVID R. KAELI, and BRAD CALDER: *Efficient Procedure Mapping Using Cache Line Coloring*. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 171–182, 1997.

[34] HEALY, CHRISTOPHER A., ROBERT D. ARNOLD, FRANK MUELLER, DAVID B. WHALLEY, and MARION G. HARMON: *Bounding Pipeline and Instruction Cache Performance*. In *Proceedings of the IEEE International Real-Time Systems Symposium*, pages 172–181, December 1994.

[35] HECKMANN, REINHOLD, MARC LANGENBACH, STEPHAN THESING, and REINHARD WILHELM: *The Influence of Processor Architecture on the Design and the Results of WCET Tools*. In *Proceedings of the IEEE*, volume 91, July 2003.

[36] HENNESSY, JOHN L. and DAVID A. PATTERSON: *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann Publishers, San Francisco, 3rd edition, 2003.

[37] HERGENHAN, A. and W. ROSENSTIEL: *Static Timing Analysis of Embedded Software on Advanced Processor Architectures*. In *Proceedings of the Date 2000 Conference*, Paris, France, 2000.

[38] HOPFNER, THOMAS, JÜRGEN STOHR, WOLFRAM FAUL und GEORG FÄRBER: *RTCPU – Realzeitanwendungen auf Dual-Prozessor PC Architekturen*. it+ti — Informationstechnik und Technische Informatik, 43(6):291, Dezember 2001.

[39] INTEL, P.O. Box 7641, Mr. Prospect IL 60056-7641: *IA-32 Intel Architecture, Software Developer's Manual, Volume 1-3*, 2001.

[40] INTEL, P.O. Box 7641, Mr. Prospect IL 60056-7641: *IA-32 Intel Architecture, Software Developer's Manual, Volume 3: System Programmer's Guide*, April 2005.

[41] KIRNER, RAIMUND and PETER PUSCHNER: *Classification of WCET Analysis Techniques*. In *Proc. 8th IEEE International Symposium on Object-oriented Real-time distributed Computing*, pages 190–199, Seattle, WA, May 2005.

[42] LI, JINGYUAN: *Analysis of the Real-Time Performance of a NUMA-Based Multiprocessor System*. Master's thesis, Institute for Real–Time Computer Systems, Technische Universität München, 2004.

[43] LI, XIANFENG, ABHIK ROYCHOUDHURY, and TULIK MITRA: *Modeling Out-of-Order Processors for Software Timing Analysis*. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 92–103, 2004.

[44] LI, Y.-T. S., S. MALIK, and A. WOLFE: *Efficient microarchitecture modeling and path analysis for real-time software*. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*, pages 298–307, Washington, DC, USA, 1995. IEEE Computer Society.

[45] LI, Y.-T. S., S. MALIK, and A. WOLFE: *Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches*. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, pages 254–263, December 1996.

[46] LIEDTKE, JOCHEN, HERMANN HÄRTIG, and MICHAEL HOHMUTH: *OS–Controlled Cache Predictability for Real–Time Systems*. In *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, Montreal, Canada, June 9–11 1997.

[47] LIU, JANE W. S.: *Real–Time Systems*. Prentice Hall, New Jersey, 2000.

[48] LUNDQVIST, THOMAS and PER STENSTRÖM: *Integrating Path and Timing Analysis using Instruction-Level Simulation Techniques*. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, Montreal, Canada, June 1998.

[49] LUNDQVIST, THOMAS and PER STENSTRÖM: *Timing Anomalies in Dynamically Scheduled Microprocessors*. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, December 1999.

[50] MAIER-KOMOR, THOMAS, ALEXANDER VON BÜLOW, and GEORG FÄRBER: *MetaC and its use for Automated Source Code Instrumentation of C programs for Real-Time Anal-*

ysis. In *Proceedings of the Work-in-Progress Session of the 17th Euromicro Conference on Real-Time Systems, (ECRTS'05)*, Palma de Mallorca, Spain, July 2005.

[51] MEHNERT, FRANK, MICHAEL HOHMUTH, and HERMANN HÄRTIG: *Cost and benefit of separate address spaces in real-time operating systems*. In *Proceedings of the 23th IEEE Real-Time Systems Symposium*, Austin, TX, (USA), December 2002.

[52] MEYBERG, KURT und PETER VACHENAUER: *Höhere Mathematik 2*. Springer, 1997.

[53] MUELLER, FRANK: *Timing Analysis for Instruction Caches*. Journal of Real-Time Systems, 18:209–239, 2000.

[54] PETRANK, EREZ and DROR RAWITZ: *The hardness of cache conscious data placement*. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 101–112, New York, NY, USA, 2002. ACM Press.

[55] PETTERS, STEFAN M.: *Bounding the Execution Time of Real–Time Tasks on Modern Processors*. In *Proc. of the 7th Int. Conf. on Real–Time Computing Systems and Applications*, Cheju Island, South Korea, December 12–14 2000.

[56] PETTERS, STEFAN M.: *Worst Case Execution Time Estimation for Advanced Processor Architectures*. PhD thesis, Institute for Real–Time Computer Systems, Technische Universität München, September 2002.

[57] PETTERS, STEFAN M.: *Comparison of Trace Generation Methods for Measurement Based WCET Analysis*. In *3rd Intl. Workshop on Worst Case Execution Time Analysis*, Porto, Portugal, July 1 2003. Satellite Workshop of the 15th Euromicro Conference on Real-Time Systems.

[58] PETTERS, STEFAN M. und ALEXANDER VON BÜLOW: *Laufzeitbestimmung von Realzeitsoftware*. it+ti — Informationstechnik und Technische Informatik, 43(4):206–214, August 2001.

[59] PETTERS, STEFAN M. and GEORG FÄRBER: *Making Worst Case Execution Time Analysis for Hard Real–Time Tasks on State of the Art Processors Feasible*. In *6th Int. Conf. on Real–Time Computing Systems and Applications (RTCSA'99)*, Hongkong, ROC, December 13–15 1999. IEEE, IEEECSP.

[60] PUAUT, ISABELLE: *Cache Analysis vs. Static Cache Locking for Schedulability Analysis in Multitasking Real-Time Systems*. In *Proc. of the 2nd International Workshop on Worst-Case Execution Time Analysis, in conjunction with the 14th Euromicro Conference on Real-Time Systems*, Vienna, Austria, June 2002.

[61] PUSCHNER, P. and A. BURNS: *A Review of Worst-Case Execution-Time Analysis (editorial)*. Journal of Real-Time Systems, 18(2/3):115–128, 2000.

[62] PUSCHNER, PETER and GUILLEM BERNAT: *WCET Analysis of Reusable Portable Code*. In *Proc. 13th Euromicro International Conference on Real-Time Systems*, pages 45–52, June 2001.

[63] PUSCHNER, PETER P. and ANTON V. SCHEDL: *Computing Maximum Task Execution Times – A Graph Based Approach*. Journal of Real-Time Systems, 13:67–91, 1997.

[64] SCHNEIDER, J. and CH. FERDINAND: *Pipeline Behavior Prediction for Superscalar Processors by Abstract Interpretation*. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 35–44, May 1999.

[65] SCHÖNBERG, SEBASTIAN: *Using PCI-Bus Systems in Real-Time Environments*. PhD thesis, Department of Computer Science, Institute for System Architecture, Technische Universität Dresden, June 2002.

[66] SCHÖNBERG, SEBASTIAN: *Impact of PCI-Bus Load on Applications in a PC Architecture*. In *Proceedings of 24th IEEE International Real–Time Systems Symposium*, Cancun, Mexico, December 2003.

[67] SEBEK, FILIP: *Cache Memories and Real-Time Systems*. Technical report, Department of Computer Engineering, Mälardalen University, Sweden, October 2001.

[68] SHANLEY, TOM: *PCI-X System Architecture*. Addison–Wesley Publishing Company, Boston, 2001.

[69] SHANLEY, TOM and DON ANDERSON: *PCI System Architecture*. Addison–Wesley Publishing Company, 3rd edition, 1995.

[70] SRINIVASAN, B., S. PATHER, R. HILL, F. ANSARI, and D. NIEHAUS: *A Firm Real–Time System Implementation Using Commercial Off–The–Shelf Hardware and Free Software*. In *Proceedings of Real–Time Technology and Applications Symposium, Denver*, June 1998.

[71] STOHR, J., A. VON BÜLOW und M. GOEBL: *Einflüsse des PCI–Busses auf das Laufzeitverhalten von Realzeitsoftware*. Technischer Bericht, Lehrstuhl für Realzeit–Computersysteme, Technische Universität München, Dezember 2003.

[72] STOHR, JÜRGEN: *Untersuchungen zur Eignung der Intel SMP Architektur für Realzeitsysteme*. Diplomarbeit, Lehrstuhl für Realzeit–Computersysteme, Technische Universität München, 2002.

[73] STOHR, JÜRGEN: *Auswirkungen der Peripherieanbindung auf das Realzeitverhalten PC-basierter Multiprozessorsysteme*. Doktorarbeit, Lehrstuhl für Realzeit–Computersysteme, Technische Universität München, September 2005.

[74] STOHR, JÜRGEN, ALEXANDER VON BÜLOW, and GEORG FÄRBER: *Controlling the Influence of PCI DMA Transfers on Worst Case Execution Times of Real–Time Software*. In *Proceedings of the 4th International Workshop on Worst Case Execution Time Analysis in conjunction with the 16th Euromicro Conference on Real–Time Systems*, Catania, Italy, June 2004.

[75] STOHR, JÜRGEN, ALEXANDER VON BÜLOW, and GEORG FÄRBER: *Using State of the Art Multiprocessor Systems as Real-Time Systems – The RECOMS Software Architec-*

ture. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems – Work in Progress Session*, Catania, Italy, June 2004.

[76] STOHR, JÜRGEN, ALEXANDER VON BÜLOW, and GEORG FÄRBER: *Bounding Worst-Case Access Times in Modern Multiprocessor Systems*. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, Palma de Mallorca, Balearic Islands, Spain, July 2005.

[77] THEILING, HENRIK, CHRISTIAN FERDINAND, and REINHARD WILHELM: *Fast and Precise WCET Prediction by Seperated Cache and Path Analyses*. The International Journal of Time-Critical Computing Systems, 18:157–179, 2000.

[78] THIELE, LOTHAR and REINHARD WILHELM: *Design for Timing Predictability*. Journal of Real-Time Systems, 28:157–177, 2004.

[79] TOMASULO, R. M.: *An efficient algorithm for exploiting multiple arithmetic units*. IBM Journal, Research and Developement 11:1, pages 25–33, January 1967.

[80] TOMIYAMA, HIROYUKI and HIROTO YASUURA: *Optimal Code Placement of Embedded Software for Instruction Caches*. In *Proceedings of the ED&TC*, pages 96–101, 1996.

[81] TOMIYAMA, HIROYUKI and HIROTO YASUURA: *Size-Constrained Code Placement for Cache Miss Rate Reduction*. In *Proceedings of the 9th International Symposium on System Synthesis*, pages 96–101, La Jolla, CA, USA, 1996.

[82] TOMIYAMA, HIROYUKI and HIROTO YASUURA: *Code Placement Techniques for Cache Miss Rate Reduction*. ACM Transactions on Design Automation of Electronic Systems, 2:410–429, October 1997.

[83] TRODDEN, J. and D. ANDERSON: *HyperTransport System Architecture*. Addison–Wesley Publishing Company, Boston, 2003.

[84] VERA, X., B. LISPER, and J. XUE: *Data Cache Locking for Higher Program Predictability*. In *In Proceedings of International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'03)*, pages 272–282, June 2003.

[85] VERA, XAVIER, BJÖRN LISPER, and JINGLING XUE: *Data Caches in Multitasking Hard Real-Time Systems*. In *International Real-Time Systems Symposium (RTSS)*, December 2003.

[86] WENZEL, INGOMAR, RAIMUND KIRNER, BERNHARD RIEDER, and PETER PUSCHNER: *Measurement-Based Worst-Case Execution Time Analysis*. In *Proc. 3rd IEEE Workshop on Future Embedded and Ubiquitious Systems*, pages 7–10, Seattle, WA, May 2005.

[87] WENZEL, INGOMAR, BERNHARD RIEDER, RAIMUND KIRNER, and PETER PUSCHNER: *Automatic Timing Model Generation by CFG Partitioning and Model Checking*. In *Proc. Conference on Design, Automation, and Test in Europe*, Mar 2005.

[88] WHITE, RANDALL T., FRANK MUELLER, CHRISTOPHER A. HEALY, DAVID B. WHALLEY, and MARION G. HARMON: *Timing Analysis for Data Caches and Set-Associative*

Caches. In *Proceedings of the Real-Time Technology and Applications Symposium*, pages 192–202, June 1997.

[89] XUE, JINGLING and XAVIER VERA: *Efficient and Accurate Analytical Modeling of Whole-Program Data Cache Behavior*. IEEE Transactions on Computers, January 2004.

[90] YODAIKEN, VICTOR: *The RTLinux Manifesto*. In *Proceedings of the 5th Linux Expo*, Raleigh, NC, March 1999.

[91] YU, LANG: *Anbindung einer AD/DA Wandlerkarte an ein Quad-Opteron Multiprozessorsystem*. Diplomarbeit, Lehrstuhl für Realzeit–Computersysteme, Technische Universität München, 2004.

Index