Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

FLORIAN SCHINTKE

Ermittlung von Programmlaufzeiten anhand von Speicherzugriffen, Microbenchmarks und Simulation von Speicherhierarchien

ZIB-Report 00-33 (September 2000)

Diplomarbeit

Ermittlung von Programmlaufzeiten anhand von Speicherzugriffen, Microbenchmarks und Simulation von Speicherhierarchien

vorgelegt von

Florian Schintke

am Fachbereich Informatik
der Technischen Universität Berlin

betreut durch

Prof. Dr. Bernd Mahr
Prof. Dr. Alexander Reinefeld

August 2000

Danksagung:

Mein Dank gilt Prof. Dr. Alexander Reinefeld, Leiter des Bereichs Computer Science am Konrad-Zuse-Zentrum für Informationstechnik Berlin und Prof. Dr. Bernd Mahr vom Institut für Formale Modelle und Logik an der TU Berlin. Schließlich danke ich meinem Betreuer Dr. Jens Simon vom Konrad-Zuse-Zentrum für Informationstechnik Berlin für seine Unterstützung, sowie meinen Eltern, die mich während meines gesamten Studiums unterstützt haben.
Außerdem danke ich Donald E. Knuth und Leslie Lamport für die Entwicklung von TEX und LATEX, wodurch die Arbeit in dieser Form präsentiert werden kann.

Kurzfassung

Kostenmodelle dienen der Ermittlung von Programmlaufzeiten, zum Vergleich der Effizienz von Algorithmen und zur Analyse des Verhaltens von Speicherhierarchien. Ein neuartiges Kostenmodell ist das Latency-of-Data-Access (LDA) Modell, das mehrere hierarchische Speicherebenen mit unterschiedlichen Latenzzeiten berücksichtigt. In dieser Diplomarbeit wird ein Simulator für Speicherhierarchien präsentiert, der die Berechnung von Programmausführungszeiten nach dem LDA-Modell erlaubt. Mit Hilfe des Simulators wird die These geprüft, daß mit diesem Modell die Ausführungszeit eines Programms adäquat abgeschätzt werden kann. Mit dem Simulator ist es erstmals praktikabel möglich, Programmausführungszeiten nach dem LDA-Modell zu bestimmen. Der Simulator kann für Systeme mit unterschiedlichen Speicherarchitekturen konfiguriert werden und unterstützt Mehrprozessorsysteme mit gemeinsamem Speicher (SMP-Systeme) mit verschiedenen Kohärenzprotokollen. Der Simulator kann mit den Ergebnissen von Microbenchmarks konfiguriert werden, die die Architekturparameter einer Speicherhierarchie messen.
Die Ergebnisse bestätigen die These nicht nur für Einzelprozessorsysteme, sondern auch für SMP-Systeme, wo gleichzeitig interagierende Prozessoren gegenseitig ihre Zugriffssequenz auf Zwischenspeicher beeinflussen. Zusätzlich wurde eine neue Einsatzmöglichkeit des LDA-Modells entwickelt, um die Ausführungszeit von Programmteilen zu bestimmen. Einzelne Zugriffskosten können einem mehrerer parallel laufender Modelle zugeordnet werden. Dadurch können Kosten, die Zugriffe auf einzelne Speicherbereiche verursachen, separat bestimmt werden. Diese Profiling-Technik erlaubt Optimierungen an Datenstrukturen und Speicherzugriffsmustern durch präzise und gezielte Informationsproduktion.

Abstract

Cost models are used to determine the execution time of programs, to compare the efficiency of algorithms, and to analyse the behaviour of memory hierarchies. The Latency-of-Data-Access (LDA) model that takes into account multiple hierarchical memory levels with different latencies, is a newly proposed, innovative cost model. In this diploma-thesis, a simulator for memory hierarchies is presented that allows the calculation of execution times using the LDA model. The simulator is used to prove the claim that the execution time of a program can be accurately estimated with the LDA model. With the simulator, it is for the first time possible to determine the execution time of programs with this model in a practical way. The simulator can be configured for systems with various cache architectures and supports shared memory (SMP) multiprocessor systems with different cache coherence protocols. The simulator can be configured with the results from microbenchmarks which measure the architectural properties of a memory hierarchy.
The results confirm the claim not only for single processor systems, but also for SMP systems, where concurrently interacting processors influence each others cache access sequence. Additionally, a new field of usage of the LDA model was developed to determine execution times of program parts. Single access costs can be assigned to one of several parallel running models. As an example the costs of accesses to different memory areas can be split and determined separately. This profiling technique allows to optimise data structures and memory access patterns of sequential and parallel SMP programs by precise production of information.

Inhaltsverzeichnis

Einleitung

La perfection est atteinte non quand il ne reste rien à ajouter,
mais quand il ne reste rien à enlever.

– ANTOINE-MARIE-ROGER DE SAINT EXUPÉRY

1.1 Thema dieser Arbeit

Die Ausführungszeit von Programmen ist in der Informatik eines der grundlegenden Kostenmaße für Berechnungen. Durch sie werden verschiedene Algorithmen, verschiedene Speicherhierarchien und verschiedene Prozessoren im Bezug auf ihre Effizienz miteinander verglichen. Mit Hilfe von Modellen oder Kalkülen wird versucht, die Ausführungszeit von Programmen zu beschreiben und abzuschätzen. Am bekanntesten ist das Modell der „Random Access Machine“ (RAM-Modell) des von Neumann Rechners, beziehungsweise die „Parallel Random Access Machine“ (PRAM-Modell).

Ein neuartiges Modell, das unterschiedliche Zugriffszeiten auf Speicherzellen in einer Speicherhierarchie berücksichtigt, ist das „Latency-of-Data-Access Modell“ (LDA-Modell). Für eine Analyse der Ausführungszeit nach dem LDA-Modell muß die Anzahl der Zugriffe auf die verschiedenen Ebenen einer Speicherhierarchie bekannt sein. Mit einem im Rahmen dieser Arbeit entwickelten Simulator für Speicherhierarchien ist eine Gewinnung dieser Daten möglich.

Der entwickelte Simulator für Speicherhierarchien ist flexibel für die Speicherhierarchien von Einzelprozessorsystemen und symmetrischen Mehrprozessorsystemen (SMP-Systemen) konfigurierbar. Er unterstützt beliebig viele hierarchisch angeordnete Speicherebenen, verschiedene Ersetzungs- und Schreibstrategien und verschiedene Kohärenzprotokolle für SMP-Systeme. Für seine Konfiguration müssen architekturspezifische Daten der Speicherhierarchie, die simuliert werden soll, bekannt sein. Diese Daten können technischen Dokumentationen von Computern entnommen oder durch Microbenchmarks ermittelt werden. Microbenchmarks können auch zur Kontrolle der technischen Daten oder zur Ermittlung der aktuellen Einstellungen (bei Konfigurierbarkeit der Hardware) genutzt werden.

Es wird untersucht, ob mit den drei Hilfsmitteln Microbenchmarks, Informationen über Speicherzugriffe und Simulation von Speicherhierarchien Ausführungszeiten von Programmen unter Verwendung des LDA-Modells akkurat bestimmt werden können. Mit dem im Rahmen dieser Arbeit entwickelten Simulator für Speicherhierarchien ist es zusätzlich möglich, nicht nur die Gesamtkosten eines Programms oder eines Programmabschnitts, sondern auch die Kosten von Speicherzugriffen auf einzelne Speicherbereiche zu ermitteln, wodurch Informationen für mögliche Optimierungen der Nutzung von Zwischenspeichern durch Änderungen der Datenstrukturen des simulierten Programms zur Verfügung gestellt werden.

1.2 Struktur dieser Arbeit

Zuerst werden in Kapitel 2 die Konzepte von Speicherhierarchien dargestellt. Dabei wird erklärt, weshalb man Speicherhierarchien einsetzt, erläutert, wie Speicherhierarchien aufgebaut sind und wie einzelne Speicherebenen verwaltet und organisiert werden. Begriffe wie Blockgröße, Assoziativität, Ersetzungs- und Schreibstrategie werden eingeführt.

Kapitel 3 beschreibt zusätzliche Konzepte von Speicherhierarchien bei SMP-Systemen. Hierbei wird erläutert, weswegen Kohärenzprotokolle für die Speicherhierarchien dieser Systeme notwendig sind, und zwei Kohärenzprotokolle (MSI und MESI) werden vorgestellt.

In Kapitel 4 wird zunächst der Zweck von Microbenchmarks erläutert und es werden nützliche Einsatzmöglichkeiten diskutiert. Dann werden Microbenchmarks, die zur Ermittlung von architekturellen Daten einer Speicherhierarchie dienen, vorgestellt und untersucht. Die Ergebnisse der Microbenchmarks werden später teilweise dazu genutzt, einen flexiblen Simulator für Speicherhierarchien zu konfigurieren. Außerdem wird gezeigt, daß Microbenchmarks, die in höheren Programmiersprachen implementiert sind, Vor- und Nachteile haben, da die genutzten Compiler die Ergebnisse von Microbenchmarks stark beeinflussen können (siehe Abschnitt 4.6).

Verschiedene Arten von Simulatoren von Speicherhierarchien, zwischen denen man anhand ihrer Funktionsweise unterscheidet, sowie das Konzept der Instrumentierung und verschiedene Methoden der Instrumentierung werden in Kapitel 5 vorgestellt. Zusätzlich werden einige Simulatoren für Speicherhierarchien beschrieben.

In Kapitel 6 wird der entwickelte Simulator für Speicherhierarchien vorgestellt, nachdem das LDA-Modell, das dem Simulator zugrunde liegt, beschrieben wurde. Die Konfigurationsmöglichkeiten, die Schnittstellen und eine beispielhafte Instrumentierung eines Programms gehören unter anderem zu der Beschreibung des Simulators.

Daß mit Hilfe des LDA-Modells in Kombination mit einer Analyse der Speicherzugriffe, die mit einem über Ergebnisse von Microbenchmarks konfigurierten Simulator für Speicherhierarchien möglich wird, eine akkurate Bestimmung von Programmlaufzeiten möglich ist, wird anhand von Beispielen im Kapitel 7 gezeigt. Die auf realen Systemen gemessenen Programmlaufzeiten werden dafür mit simulierten Programmlaufzeiten verglichen. In Kapitel 7 wird auch die Möglichkeit des Simulators genutzt, die Kosten, die Zugriffe auf verschiedene Speicherbereiche verursachen, getrennt zu betrachten. Die Kosten der Speicherzugriffe einzelner Matrizen bei einer Matrix-Matrix-Multiplikation werden beobachtet und dann, durch Optimierung des Speicherformats einer Matrix, die Gesamtkosten auch für größere Matrizen reduziert.

Es bestehen im Prinzip zwei Haupteinsatzgebiete der in dieser Arbeit vorgestellten Methode zur Ermittlung von Programmlaufzeiten: Einerseits kann die Methode zur Analyse und Optimierung von Programmen für eine Speicherhierarchie oder zur Wahl eines geeigneten Algorithmus dienen, andererseits können, bei der Entwicklung einer neuen Speicherhierarchie, Programmlaufzeiten ermittelt werden, ohne Prozessor oder Speicherhierarchie in Hardware implementieren zu müssen. Eine Optimierung der Speicherhierarchie ist auf diese Art möglich, indem die Leistungen verschiedener Konfigurationen der Speicherhierarchie analysiert und miteinander verglichen werden.

Speicherhierarchien von Einzelprozessorsystemen

A memory hierarchy is a natural reaction to locality and technology.

– John L. Hennessy und David A. Patterson,
Computer Architecture – A Quantitative Approach

Speicherhierarchien werden benutzt, um die durchschnittliche Zeit, die für einen Speicherzugriff in einem Computer benötigt wird, zu reduzieren und so die Ausführungszeit von Programmen zu verkürzen. Speicherhierarchien umfassen mehrere Ebenen von Zwischenspeichern verschiedener Größe und Geschwindigkeit, die sich gegenseitig nutzen. Für diese Zwischenspeicher existieren verschiedene Realisierungsmöglichkeiten, die sich in Nutzbarkeit, dem Verwaltungsaufwand, der Geschwindigkeit und der benötigten Schaltung unterscheiden. Blockgröße, Zeilenanzahl, Assoziativität und Ersetzungs- und Schreibstrategie sind hierbei die zentralen Parameter, die das Verhalten eines Zwischenspeichers beeinflussen.

2.1 Zweck von Speicherhierarchien

Eine Speicherhierarchie besteht aus Speicherebenen, die verschiedene Geschwindigkeiten und Größen haben. In einer Speicherhierarchie werden von kleineren schnelleren Speichern, die im Bezug auf Kommunikationswege näher am Prozessor liegen, größere langsamere Speicher genutzt, die weiter entfernt vom Prozessor liegen. Eine Speicherhierarchie wird von einem oder mehreren Prozessoren genutzt, um sowohl das auszuführende Programm als auch seine Daten zu lesen und zu speichern.

Lokalität

Eine Speicherhierarchie soll die Ausführungszeiten von Programmen gegenüber einer Ausführung ohne Speicherhierarchie (nur Hauptspeicher) verkürzen. In vielen Programmen gibt es Bereiche, die in einem kleinen Abschnitt des Programms wiederholt auf die gleichen Speicherzellen und/oder auf benachbarte Speicherzellen zugreifen, also eine Lokalität der Daten aufweisen. Man spricht von räumlicher Lokalität (spatial locality), wenn nahe beieinanderliegender Adressen benötigt werden, und von zeitlicher oder temporaler Lokalität (temporal locality), wenn auf eine Adresse in einem Zeitintervall häufig zugegriffen wird. Ein wiederholtes Ausführen desselben, kleinen Programmteils führt zu einer Lokalität des Programmcodes. Der Ablauf solcher Programme kann beschleunigt werden, indem die in einem Teilabschnitt des Programms benötigten Daten in einem kleineren, aber dafür schnelleren Speicher (Cache) zwischengespeichert werden, weil sich dadurch die durchschnittliche Wartezeit (auch Latenzzeit genannt) des Prozessors auf angeforderte Daten reduzieren läßt. Wartezeiten können in einigen Fällen mit Rechenoperationen, die nur Daten benötigen, die bereits in Registern des Prozessors gespeichert sind, überbrückt werden. Je länger der Prozessor warten muß, ohne die Wartezeit mit Rechenoperationen überbrücken zu können, desto schlechter ist er ausgelastet. Um einen Prozessor gut auszulasten, sind also Speicher nötig, die schnell genug sind, dem Prozessor die benötigten Daten zu liefern. HENNESSY und PATTERSON (1996) haben statistische Analysen über die Lokalität von Daten in Programmen durchgeführt und beschrieben.

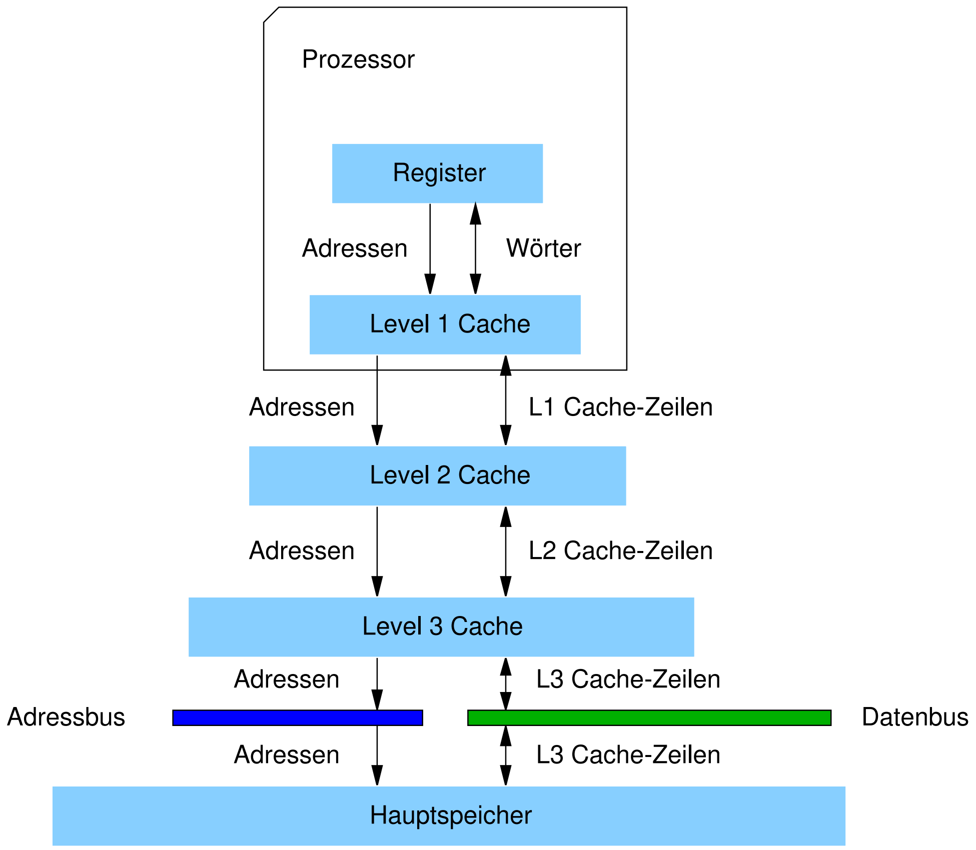


Abbildung 2.1: Je weiter oben in einer Speicherhierarchie eine Speicherebene angesiedelt ist, desto schneller sind Datenzugriffe, und desto kleiner ist ihre Kapazität.

Abbildung 2.1 zeigt den grundsätzlichen Aufbau einer Speicherhierarchie. Die kleinste Speicherebene, die nicht die Register des Prozessors sind, nennt man „Level 1 Cache“ oder „First Level Cache“ (im folgenden nur noch L1 Cache). Die weiteren Speicherebenen werden aufsteigend durchnumeriert, bis der Hauptspeicher erreicht ist (im folgenden als L2, L3, ..., Ln–1, Ln, Ln+1 Cache und so weiter bezeichnet). Üblich sind zwei bis drei Zwischenspeicher. Eine konkrete Speicherhierarchie nennt man auch Speicherarchitektur oder, wenn aus dem Zusammenhang klar ist, daß es um eine Speicherarchitektur geht, auch einfach nur Architektur. Die Caches sind miteinander verbunden und tauschen Adressen und Daten über ihre Verbindungen aus.

Fordert ein Prozessor Daten an, durchläuft diese Anfrage nacheinander die Zwischenspeicher, bis einer die Anfrage beantworten kann (die Daten gerade gespeichert hat). Sind die Daten nicht zwischengespeichert, beantwortet der Hauptspeicher die Anfrage.

Die Lösung kleinere Zwischenspeicher einzusetzen, um schnelle Speicherhierarchien zu realisieren und nicht einen großen, schnellen Speicher zu benutzen hat zwei Hauptgründe: Zum einen sind Kostengründe und unterschiedliche Technologien, zum anderen aber auch physikalische Gesetze dafür ausschlaggebend. Zuerst werden im folgenden die physikalischen Gründe betrachtet.

Große Speicher sind langsamer

Physikalisch bedingt werden Speicher im Durchschnitt immer langsamer, je größer sie werden. Je mehr Daten ein Speicher speichern kann, desto größer ist auch seine räumliche Ausdehnung gegenüber einem kleineren Speicher unter Verwendung der gleichen Technologie. Daten, die ein Prozessor vom Speicher anfordert, müssen zur Schnittstelle des Speichers transportiert werden, um zum Prozessor oder auf den Datenbus zu gelangen. Da Information maximal mit Lichtgeschwindigkeit transportiert werden kann, ist bei räumlich ausgedehnteren Speichern die durchschnittliche Zeit bis die Daten an der Schnittstelle anliegen höher als bei kleineren Speichern, weil die Information im Durchschnitt längere Wege zurücklegen muß. Deshalb sind größere Speicher langsamer.

Geschwindigkeitsunterschiede zwischen verschiedenen Speichern einer Speicherhierarchie kommen jedoch nicht nur durch geringere Entfernungen für Datentransporte in kleineren Speichern zustande, sondern auch durch den Einsatz besserer und teurerer Technik für kleinere Speicher (schnellere Schalter, kleinere physikalische Kapazitäten) ohne einen proportionalen Anstieg der Gesamtkosten für eine Speicherhierarchie zu verursachen.

Es wurden mit der Zeit verschiedene Methoden erdacht, Zwischenspeicher zu organisieren und zu verwalten, die jeweils unterschiedlich viel Verwaltungsinformation benötigen und den Zwischenspeicher verschieden gut ausnutzen. Zwischenspeicher, die sich von vielen Programmen gut nutzen lassen, haben meist einen höheren Verwaltungsaufwand als solche, deren Nutzung etwas eingeschränkter ist. Diese Aspekte werden detailliert im nächsten Abschnitt betrachtet. Bei der Konstruktion einer Speicherhierarchie muß abgewogen werden, welcher Kompromiß zwischen Nutzbarkeit und Verwaltungsaufwand sinnvoll ist.

2.2 Aufbau eines Caches

Der Aufbau eines Caches beeinflußt die Komplexität, den Platzverbrauch, die Nutzbarkeit und die Geschwindigkeit des Caches. Caches mit hoher Assoziativität (siehe unten) sind aufwendiger, aber im allgemeinen auch effektiver nutzbar als Caches mit niedriger Assoziativität. Ein Cache ist, wie in Abbildung 2.2 dargestellt, in Blöcke (auch Cache-Zeilen oder Cachelines genannt) aufgeteilt. Aus der Blockgröße und der Blockanzahl ergibt sich die Speicherkapazität des Caches. Für die Verwaltung der Daten im Cache wird eine Blockframe Adresse gespeichert, durch die eindeutig wird, welche Adressen in der entsprechenden Cache-Zeile gerade zwischengespeichert sind. Dafür werden die letzten Bits der Adresse weggelassen und so die Blockframe Adresse bestimmt. Die letzten Bits einer Adresse bestimmen die genaue Position der Daten innerhalb einer Cache-Zeile.

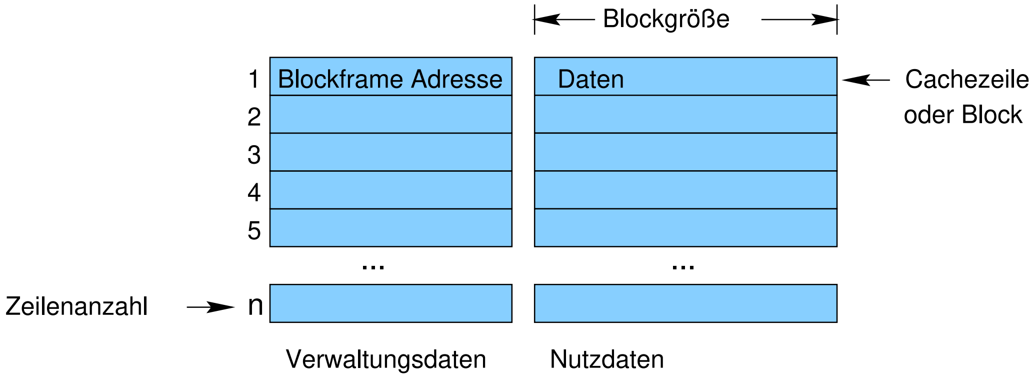


Abbildung 2.2: Ein Cache speichert Daten in Einheiten der Blockgröße. Damit die Daten ihrer Adresse zugeordnet werden können, muß Verwaltungsinformation gespeichert werden (Blockframe Adresse), bei der die letzten Bits der Adresse weggelassen werden. Die letzten Bits einer Adresse bestimmen die genaue Position der Daten innerhalb einer Cache-Zeile. Die Kapazität des Caches ergibt sich aus der Blockgröße und der Zeilenanzahl.

Ereignisse in einem Cache

Während der Nutzung eines Caches können verschiedene Ereignisse eintreten. Von einem „Cache-Hit" spricht man, wenn angeforderte Daten im Cache zwischengespeichert sind. Müssen Daten erst aus einer größeren Speicherebene in den Cache kopiert werden, spricht man von einem „Cache-Miss". Die Phrase „Zugriff auf eine Speicherebene" soll heißen, daß diese Speicherebene einen Cache-Hit hat.

Assoziativität

Als Assoziativität eines Caches bezeichnet man die Anzahl der verschiedenen Cache-Zeilen, in denen das Datum einer gegebenen Adresse zwischengespeichert werden kann. Die Assoziativität wird als ganze positive Zahl angegeben.

Bei Assoziativität 1 steht mit der Adresse auch fest, in welcher Cache-Zeile sie zwischengespeichert wird. Caches mit Assoziativität 1 werden auch „Direct-Mapped-Caches" genannt. Bei höherer Assoziativität wird der Cache in „Sets" eingeteilt, die jeweils wie ein Direct-Mapped-Cache arbeiten. Einen Cache mit einer Assoziativität, die der Cache-Zeilenanzahl entspricht, nennt man „vollassoziativ". Bei einem vollassiziativen Cache kann jede Adresse in jeder Cache-Zeile zwischengespeichert werden.

Einfluß der Assoziativität

Caches mit niedriger Assoziativität sind gut von Programmen nutzbar, die eine hohe Lokalität der benötigten Speicherzellen aufweisen (benachbarte Speicherzellen werden häufig gemeinsam benötigt). Je höher die Assoziativität eines Caches ist, desto besser kann er auch von Programmen, deren räumliche Datenlokalität nicht so hoch ist, durch temporäre Lokalität genutzt werden. Bei solchen Programmen entstehen bei Caches niedrigerer Assoziativität Verdrängungseffekte, obwohl andere Teile des Zwischenspeichers Daten speichern, die das Programm gerade nicht benötigt. Allerdings bedeutet eine höhere Assoziativität eine aufwendigere Schaltung und mehr administrative Daten und damit einen insgesamt höheren Verwaltungsaufwand. Auch die Zeit für das Testen auf einen Cache-Hit wird mit größerer Assoziativität höher, da eine Adresse an mehreren Stellen im Cache zwischengespeichert sein kann.

Ersetzungsstrategien

Wenn vom Prozessor eine Speicherzelle angefordert wird, die bisher nicht im Cache zwischengespeichert ist, und keine passenden freien Cache-Zeilen mehr im Cache vorhanden sind, muß entschieden werden, welche Cache-Zeile im Cache ersetzt wird. Eine Ersetzungsstrategie ist nur erforderlich, wenn der Cache eine Assoziativität größer als 1 hat (nicht Direct-Mapped), weil sonst ohnehin nur eine Cache-Zeile zur Speicherung der angeforderten Adresse in Frage kommt. Die Assoziativität des Caches gibt die Anzahl von möglichen Cache-Zeilen an, aus denen ausgewählt werden muß. Folgende Verfahren sind üblich:

FIFO: First in First out; Die Cache-Zeile, die sich zeitlich am längsten im Cache befindet, wird ersetzt.

LRU: Least Recently Used; Die Cache-Zeile, die am längsten nicht benutzt wurde, wird ersetzt.

Round Robin: Die Cachzeilen werden reihum ersetzt.

Random: Die zu ersetzende Cachzeile wird zufällig ausgewählt.

Je nach benutzter Ersetzungsstrategie werden unterschiedliche Verwaltungsinformationen im Cache benötigt: Bei LRU muß bekannt sein, welche Cache-Zeile am längsten nicht benutzt wurde, bei Round Robin, welche Cache-Zeile als nächste zu ersetzen ist, und bei Random muß eine Zufallszahl generiert werden.

Schreibstrategien

Schreibt der Prozessor Daten, wird dies je nach Schreibstrategie der Zwischenspeicher unterschiedlich behandelt. Es gibt folgende Schreibstrategien:

Write Back: Die Daten werden zunächst nur in den Cache geschrieben und erst in eine höhere Speicherebene, wenn die Cache-Zeile ersetzt werden soll. Diese Strategie führt zu etwas längeren Latenzzeiten bei einer Ersetzung einer Cache-Zeile, da eventuell noch die zwischengespeicherten Daten in die nächste Speicherebene geschrieben werden müssen. Zur Verwaltung wird je Cache-Zeile ein Bit benötigt, das speichert, ob die Daten in dieser Cache-Zeile modifiziert wurden (dirty-bit).

Write Through: Die Daten werden in die nächste Speicherebene und in den Cache geschrieben. Diese Variante ist einfacher als Write Back zu realisieren. Der Nachteil ist eine stärkere Speicherbusbelastung, da bei jeder Änderung die Daten in die nächste Speicherebene geschrieben werden. Die Latenzzeit zum Schreiben in den Hauptspeicher wird dabei vor dem Prozessor verborgen.

Lesestrategien

Ein Cache ersetzt immer eine ganze Cache-Zeile. Es gibt aber Architekturen, bei denen der Speicherbus wegen der Busbreite nicht eine komplette Cache-Zeile in einem Stück liefert, so daß bestimmte Daten eher ankommen. In diesem Fall arbeitet der Cache mit einer der folgenden Lesestrategien. Entweder die Cache-Zeile wird konsekutiv vom Anfang bis zum Ende gefüllt und die benötigten Daten an den Prozessor geschickt oder die vom Prozessor benötigten Daten werden zuerst angefordert, direkt an den Prozessor weitergeleitet und danach der Rest der Cache-Zeile gefüllt, wodurch die Latenzzeit für den Prozessor reduziert werden kann.

2.3 Neuartige Strategien für die Nutzung von Zwischenspeichern

Für eine bessere Ausnutzung der temporalen und räumlichen Lokalität von Daten schlagen PRVULOVIĆ et al. (1999a,b) eine statische Aufspaltung des Caches in zwei Teile vor. Der eine Teil soll für temporal lokale Daten genutzt werden und dementsprechend eine kleine Blockgröße und hohe Assoziativität aufweisen. Der andere Teil soll räumlich nahe beieinander liegende Speicherzugriffe optimieren und deshalb eine große Blockgröße aufweisen und zusätzlich mit einer Erkennung für Zugriffe mit gleicher Schrittweite ausgestattet werden, um Daten schon vor ihrem Zugriff in den Cache zu laden. Meßergebnisse zeigen, daß die durchschnittliche Latenzzeit der Caches durch diese Architektur in vielen Anwendungen verringert werden kann. Gemessen wurden die Ausführungszeiten von Programmen aus der SPEC95 Benchmark Suite (DIXIT und REILLY, 1995).

Eine noch höhere Flexibilität der Nutzung der Caches bieten RANGANATHAN et al. (2000) mit einer neuartigen Strategie für die Nutzung von Zwischenspeichern. Ihren rekonfigurierbaren Caches liegt die Idee zugrunde, einen Cache während der Ausführungszeit dynamisch in Bereiche zu unterteilen, die nach unterschiedlichen Strategien arbeiten. Bei einer Anfrage wird in allen Bereichen gleichzeitig geprüft, ob sie die angefragten Daten liefern können. Angedacht ist zum Beispiel auch, daß ein Bereich des Zwischenspeichers so konfiguriert werden kann, daß er durch den vom Compiler generierten Code kontrolliert wird und als schneller temporärer Speicher genutzt wird. RANGANATHAN et al. versprechen sich dadurch die Beschleunigung von Prozessen, die auf großen Datenmengen arbeiten (sogenanntes „Media Processing“), indem nur wiederverwendbare Daten zwischengespeichert werden, die bei einer normalen Ersetzungsstrategie eines Zwischenspeichers immer wieder durch die zu bearbeitenden Daten aus dem Zwischenspeicher verdrängt würden.

Speicherhierarchien von symmetrischen Multiprozessorsystemen

Der Kern des Lebens ist asymmetrisch.

– DOMINIK BENDER

In diesem Kapitel werden die zusätzlichen Konzepte für Speicherhierarchien von symmetrischen Multiprozessorsystemen (SMP-Systemen) eingeführt. Das wichtigste Konzept hierbei ist das Kohärenzprotokoll für Zwischenspeicher. Nach einer Erläuterung, weshalb Kohärenzprotokolle benötigt werden, und einer Unterteilung der möglichen Kohärenzprotokolle in zwei Gruppen (aktualisierende und invalidierende) werden zwei Protokolle (MSI und MESI) vorgestellt.

3.1 Symmetrische Multiprozessorsysteme (SMP-Systeme)

Symmetrische Multiprozessorsysteme (SMP-Systeme) sind Systeme, in denen sich mehrere Prozessoren einen gemeinsamen Hauptspeicher teilen und alle Prozessoren auf die gleiche Weise auf den Hauptspeicher zugreifen können. Jeder Prozessor hat seine eigenen Caches und der gemeinsame Speicher wird meist über einen gemeinsamen Speicherbus angesteuert. Diese Architektur ist für eine kleine bis mittlere Anzahl von Prozessoren geeignet. Bei einer größeren Anzahl von Prozessoren würde der Speicherbus zum Engpaß werden, und die Anwendungen würden nicht mehr gut skalieren.

Die Zugriffe auf den gemeinsamen Speicher finden aufgrund der Busarchitektur in SMP-Systemen sequentialisiert statt. Eine SMP-Architektur sorgt mit Hilfe von Kohärenzprotokollen dafür, daß auch durch die Nutzung der Caches für lesende und schreibende Zugriffe eine sequentialisierte Zugriffsweise auf Speicheradressen für alle Prozessoren als Modell der Speicherhierarchie gültig bleibt. Dadurch eignet sich diese Architektur gut zur parallelen Programmierung, da das Speichermodell aus Sicht der einzelnen Prozessoren sehr einfach ist und alle Speicherzugriffe die gleiche Semantik wie Hauptspeicherzugriffe haben. Der Datenaustausch zwischen Prozessoren geschieht über Zugriffe auf gemeinsame Adressen und verursacht keinen weiteren Protokollaufwand wie etwa das Kopieren der Daten in Zwischenpuffer, Einteilung in Pakete, Erstellen von Protokollpaketen und Kontrolle über die korrekte Übertragung, bei nachrichtengekoppelten Systemen. Welche Probleme durch prozessorlokale Caches auftreten können, und warum Kohärenzprotokolle notwendig sind, um für alle Prozessoren das Modell der sequentialisierten Speicherzugriffe aufrecht zu erhalten, soll der folgende Abschnitt klären.

3.2 Zweck und Arten von Kohärenzprotokollen

Sollen zwischen Prozessoren in einem SMP-System Daten ausgetauscht werden, geschieht dies über gemeinsam genutzte Adressbereiche der Prozessoren: Ein Prozessor schreibt Daten und ein anderer liest sie anschließend. Bei einem Arbeitsspeicher, der die Speicherzugriffe sequentialisiert abarbeitet, verhält sich alles so, wie der Programmierer es erwarten würde. Wird eine Adresse (egal von welchem Prozessor) gelesen, nachdem ein anderer dort Daten geschrieben hat, bekommt er die zuletzt geschriebenen Daten. Haben die Prozessoren eigene Zwischenspeicher, kann es passieren, daß ein Prozessor Daten nur in seine Zwischenspeicher schreibt (bei der Write Back Strategie) und ein anderer Prozessor veraltete Daten aus seinen Zwischenspeichern liest, so daß ein Datenaustausch fehlschlägt.

Um dieses Problem zu lösen, gibt es zwei Möglichkeiten. Entweder man überläßt den Prozessoren (und somit den Compilern) das Problem und gibt ihnen über einen geeigneten Instruktionssatz die Möglichkeit, zu wählen, ob Speicherzugriffe auf den Hauptspeicher oder die Zwischenspeicher erfolgen, und ob diese die Inhalte von Zwischenspeichern anderer Prozessoren beeinflussen, oder man sorgt mit Hilfe eines Kohärenzprotokolls dafür, daß die Zwischenspeicher für die Prozessoren transparent genutzt werden. Ein Kohärenzprotokoll hat die Aufgabe, die getrennten Zwischenspeicher der Prozessoren so zu verwalten, daß keine Inkonsistenzen gegenüber dem Verhalten eines Systems ohne Zwischenspeicher entstehen. Um diese Aufgabe zu erfüllen, wurden verschiedene Protokolle entwickelt, die sich in zwei Klassen einteilen lassen:

Update based: Protokolle, die nach diesem Verfahren arbeiten, aktualisieren bei Bedarf Zwischenspeicher anderer Prozessoren mit aktuell geschriebenen Daten, so daß diese Zwischenspeicher weiter genutzt werden können. Ein Protokoll dieser Art ist zum Beispiel das Dragon-Kohärenzprotokoll.

Invalidation based: Protokolle, die nach diesem Verfahren arbeiten, markieren bei Bedarf Inhalte von Zwischenspeichern anderer Prozessoren als ungültig, so daß ein Zugriff auf die betroffenen Adressen einen Zugriff auf den Speicherbus bewirkt. Beispiele für diese Art von Protokollen sind das MSI- und das MESI-Protokoll.

Für beide Arten von Protokollen gibt es verschiedene Möglichkeiten diese zu realisieren. Kohärenzprotokolle sind detailliert in CULLER und SINGH (1999) beschrieben. Folgend sollen zwei weit verbreitete Kohärenzprotokolle vorgestellt werden, da sie im weiteren Verlauf der Arbeit zur Simulation der Speicherhierarchie eines SMP-Systems mit zwei Pentium III Prozessoren genutzt werden (siehe Kapitel 7 ab Seite 46).

3.3 Kohärenzprotokoll MSI

Das MSI-Protokoll ist ein auf Invalidation beruhendes Kohärenzprotokoll, das auf drei Zuständen basiert, die jede Cache-Zeile einnehmen kann. Die drei Zustände, aus deren Anfangsbuchstaben der Name des Protokolls gebildet ist, sind:

Modified: In diesem Zustand befinden sich Cache-Zeilen, die nach dem Laden modifiziert wurden. Nur jeweils ein Prozessor kann für Adressen einer Cache-Zeile eine Cache-Zeile in diesem Zustand haben. Andere Prozessoren haben diese Adresse nicht gültig in ihren Zwischenspeichern. Lese- und Schreiboperationen können ohne Benachrichtigung anderer Prozessoren ausgeführt werden.

Shared: In diesem Zustand befinden sich Cache-Zeilen, auf die bisher nur lesend zugegriffen wurde. Es können mehrere Prozessoren die gleiche Adresse gleichzeitig in ihren Zwischenspeichern haben.

Invalid: In diesem Zustand befinden sich Cache-Zeilen, die keine gültigen Inhalte mehr enthalten. In diesen Zustand werden alle Cache-Zeilen anderer Prozessoren gesetzt, die vorher im Zustand *Shared* waren, wenn ein Prozessor eine Adresse in seinen Cache schreibt (seine Cache-Zeile auf *Modified* setzt).

Je nach Ausgangszustand und Ereignis wechseln die Cache-Zeilen zwischen diesen drei Zuständen, von denen auch abhängig ist, wie die Cache-Controller sich verhalten. Der Zustandsgraph in Abbildung 3.1 stellt die genauen Zustandsübergänge dar. Ist zum Beispiel eine Cache-Zeile A im Zustand *Modified* und ein anderer Cache B liest diese Adresse, wird das Bus-Ereignis „Bus Read“ oder „Bus Read Exclusive“ ausgelöst. Dadurch wird der Zustand der Cache-Zeile A auf *Shared* gesetzt und deren Inhalt auf den Bus geschrieben („Flush“), so daß der aktuelle Inhalt von Cache B übernommen werden kann. Modifizieren mehrere Prozessoren nahe beieinanderliegende oder gleiche Adressen, invalidieren sie sich gegenseitig die Zwischenspeicher, wodurch Daten häufiger aus dem Hauptspeicher gelesen werden müssen.

3.4 Kohärenzprotokoll MESI

Das MESI-Protokoll ist eine Erweiterung des MSI-Protokolls und wird in modernen Mehrprozessorsystemen wie zum Beispiel Systemen der Intel Pentium Reihe (MINDSHARE, INC. und SHANLEY, 1999) eingesetzt. Zusätzlich zu den drei Zuständen des MSI-Protokolls kommt ein weiterer Zustand *Exclusive* hinzu. In diesem Zustand befindet sich eine Cache-Zeile, deren Adressen kein anderer Prozessor in seinen Zwischenspeichern hat, so daß bei schreibenden Zugriffen keine anderen Cache-Zeilen auf *Invalid* gesetzt werden müssen. Es ist nicht zwingend, daß eine Cache-Zeile, die in keinem Zwischenspeicher anderer Prozessoren gespeichert ist, in diesem Zustand ist. Eine solche Cache-Zeile könnte auch im Zustand *Shared* sein. Nur beim Laden einer Cache-Zeile wird überprüft, ob andere Caches diese bereits zwischenspeichern. Ist dies nicht der Fall, wird die neue Cache-Zeile auf den Zustand *Exclusive* gesetzt. Bei einer Schreiboperation wechselt der Zustand dann ohne Bus-Ereignis auf *Modified*. Der zusätzliche Zustand *Exclusive* hilft Bandbreite auf dem Speicherbus bei sequentiellen Programmen zu sparen, die keine gemeinsamen Daten von Prozessoren nutzen.

Abbildung 3.2 zeigt den für das MESI-Protokoll modifizierten Zustandsgraphen. Je weiter oben in dieser Abbildung ein Zustand angeordnet ist, desto mehr sind Daten einer Cache-

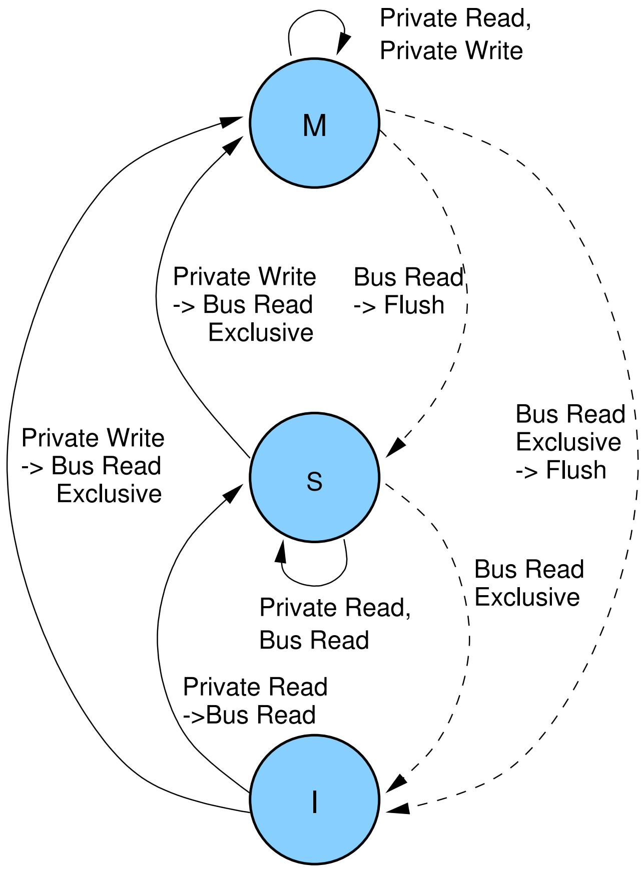


Abbildung 3.1: Der Zustandsgraph des MSI-Protokolls nach CULLER und SINGH (1999). Die Zustände M, S, I stehen für *Modified*, *Shared* und *Invalid*. Es werden Ereignisse, die vom Prozessor ausgehen, und Bus-Ereignisse, die von Cache-Controllern ausgehen, unterschieden. Gestrichelte Pfeile stellen Zustandsübergänge dar, die durch Bus-Ereignisse hervorgerufen werden. Eine Aktion kann weitere Bus-Ereignisse verursachen. Dies wird durch einen Pfeil A –> B dargestellt und bedeutet, daß durch Ereignis A das Bus-Ereignis B verursacht wird. Das Ereignis „Flush" bedeutet dabei, daß der Inhalt der Cache-Zeile auf den Bus geschrieben wird.

Zeile in diesem Zustand einem Prozessor zugeordnet und können schlechter von Caches verschiedener Prozessoren gleichzeitig zwischengespeichert werden.

Das Bus-Ereignis „Bus Read" liefert beim MESI-Protokoll zunächst ein Resultat zurück, ob der Inhalt in anderen Caches gefunden wurde. Anschließend werden die Daten übertragen und ein Zustandsübergang vom Zustand *Invalid* in den Zustand *Shared* oder *Exclusive* durchgeführt. Das Bus-Ereignis „Flush' " bedeutet, daß potenziell mehrere Prozessoren oder Caches die aktuellen Daten auf den Bus schreiben könnten, aber nur einer ausgewählt wird, der dies tut.

Auch beim MESI-Protokoll invalidieren sich Prozessoren gegenseitig ihre Caches, wenn sie gleichzeitig nahe beieinanderliegende Adressen modifizieren, was zu Leistungseinbußen führen kann. In Abschnitt 7.4 ab Seite 46 wird dies an einem Beispiel gezeigt.

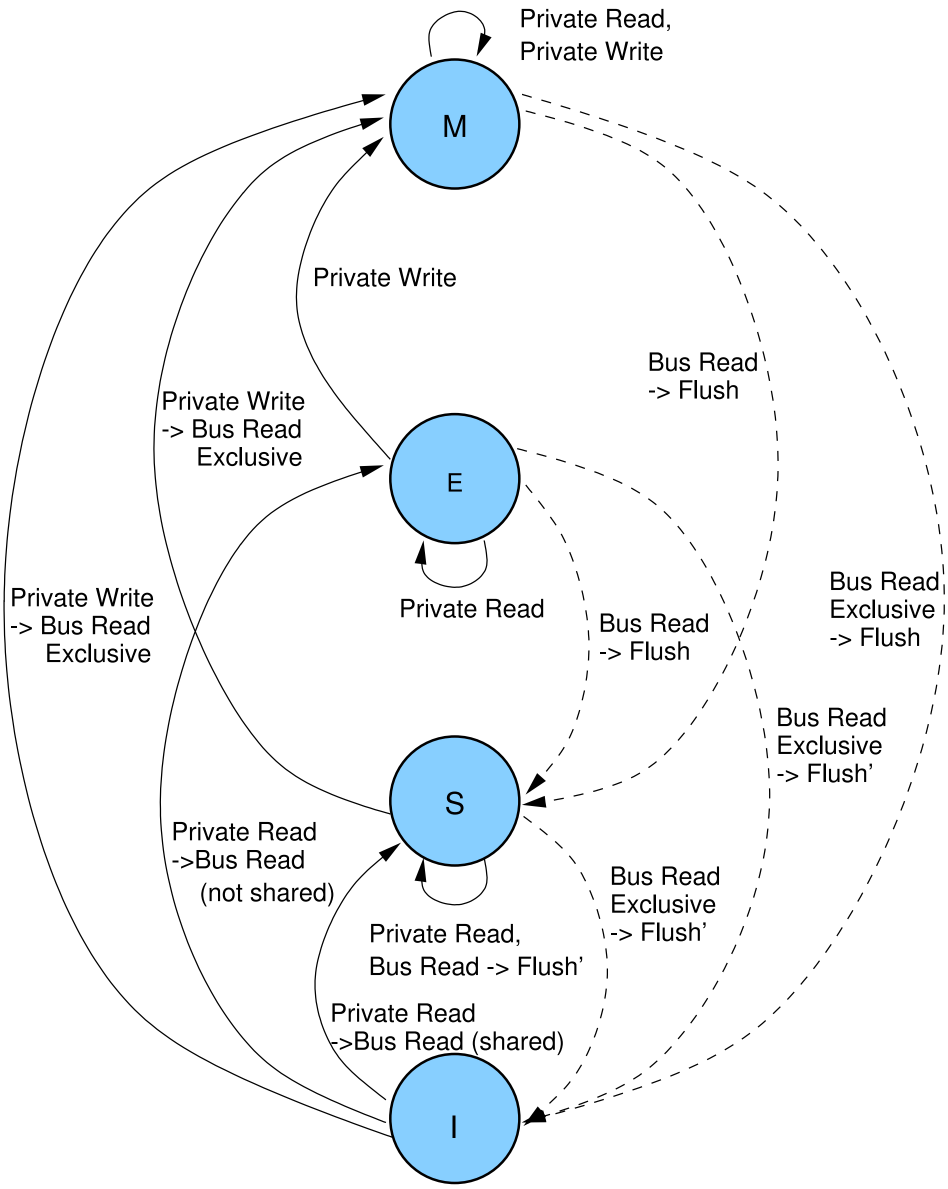


Abbildung 3.2: Der Zustandsgraph des MESI-Protokolls nach CULLER und SINGH (1999). Die Zustände M, E, S, I stehen für *Modified*, *Exclusive*, *Shared* und *Invalid*. Wie in Abbildung 3.1 werden Ereignisse, die vom Prozessor ausgehen, und Bus-Ereignisse, die von Cache-Controllern ausgehen, unterschieden. Gestrichelte Pfeile stellen Zustandsübergänge dar, die durch Bus-Ereignisse hervorgerufen werden.

Microbenchmarks

Durch das Einfache geht der Eingang zur Wahrheit.

– GEORG CHRISTOPH LICHTENBERG (1742 – 1799)

Microbenchmarks sind kleine, einfache Benchmarks, die spezielle Systemparameter wie Prozessortakt oder Speicherlatenzzeiten messen. Für diese Messungen ist es wichtig, die Genauigkeit der Zeitmeßroutinen zu bestimmen, damit Messungen so lange ausgeführt werden, daß zuverlässige Resultate ermittelt werden können. Hierfür wird ein Microbenchmark zum Messen der Meßgenauigkeit vorgestellt. Wie man unabhängig von der Prozessorarchitektur den Prozessortakt bestimmen kann, wird in Abschnitt 4.3 erläutert.

Mit Microbenchmarks können außerdem wichtige Parameter einer Speicherhierarchie, wie die Anzahl der Speicherebenen, die Größe, die Blockgröße und die Latenzzeitsen von Speicherebenen, durch Messen ermittelt werden. Diese Daten können dann für die Konfiguration eines Simulators für Speicherhierarchien genutzt werden. Daß die Speicherhierarchie, zu der auch die Register eines Prozessors gehören, sehr wichtig für die Leistung ist, die ein Prozessor Anwendungen zur Verfügung stellt, wird am Beispiel des Horner-Schema-Benchmarks ab Seite 21 gezeigt.

4.1 Zweck von Microbenchmarks

Nach CULLER und SINGH (1999) dienen Microbenchmarks dazu, die Leistung einer bestimmten Teilfunktion eines Systems zu messen. Dabei versucht man, andere Einflußfaktoren auszuschließen, so daß die Ergebnisse einfach zu interpretieren sind und mit der Leistung der korrespondierenden Hardware verglichen werden können. Typische Microbenchmarks messen beispielsweise Latenzzeiten von Speichern, Speicherbandbreiten, Taktraten von Prozessoren, Rechengeschwindigkeit von Prozessoren oder den Aufwand für einen Unterprogrammaufruf.

Hochsprachen

Diese Daten, die Microbenchmarks ermitteln, sind oft auch schon in Beschreibungen der Hardwarehersteller vorhanden. Der Einsatz von Microbenchmarks ist trotzdem gerechtfertigt, um einerseits die Daten der Hardwarehersteller zu überprüfen, und andererseits die Daten unter realen Bedingungen zu messen. Benutzt man verschiedene Compiler, um einen Microbenchmark zu übersetzen, kann man den generierten Code vergleichen und auf Leistungsunterschiede untersuchen (siehe Abschnitt 4.6). Ein aus einer Hochsprache mit einem Compiler übersetzter Microbenchmark wird Leistungswerte, die eine reale Applikation erreichen kann, besser widerspiegeln, als ein direkt in Maschinensprache implementierter und optimierter Microbenchmark.

In den folgenden Abschnitten sollen verschiedene Microbenchmarks vorgestellt werden. Da Benchmarking und Leistungsanalyse zentrale Gebiete als Voraussetzung für Optimierungen sind, gibt es mittlerweile sehr viele Microbenchmarks, so daß ein Anspruch auf eine vollständige Betrachtung hier nicht erhoben werden kann. Vielmehr soll eine kleine Auswahl getroffen werden, an der man erkennen kann, wie unterschiedliche Microbenchmarks funktionieren und welche Daten man mit Microbenchmarks über eine Rechner- und Speicherarchitektur ermitteln kann.

4.2 Genauigkeit der Zeitmessung

Dadurch, daß die Geschwindigkeit von Prozessoren zur Zeit immer weiter steigt, ist es schwierig, Benchmarks zu schreiben, die für mehrere Prozessorgenerationen zuverlässige Ergebnisse liefern. Kann ein Benchmark von schnellen Prozessoren zu schnell abgearbeitet werden, gelangt man an die Grenze der Meßgenauigkeit der Zeitmeßfunktionen. Gestaltet man andererseits den Benchmark aufwendiger, benötigt eine Messung auf einem langsameren Prozessor unnötig viel Zeit.

Durch das Messen der Genauigkeit der Zeitmessung kann eine Mindestausführungszeit für einen Microbenchmark festgelegt und die Anzahl der Wiederholungen des Benchmarks variabel gestaltet werden. Beispielsweise benutzen McVoy und Staelin (1996) für die Ermittlung der Genauigkeit der Funktion `gettimeofday()` ein Verfahren, bei dem sie einen Block von Anweisungen solange vergrößern, bis eine geringe Änderung der Anzahl der wiederholten Ausführungen dieses Blocks eine entsprechend kleine Änderung der dafür benötigten Zeit ergibt. Würde sich zum Beispiel die gemessene Zeit gar nicht ändern, egal ob man den Anweisungsblock einmal oder zweimal ausführt, ist der Block und die Zeit, die für seine Ausführung benötigt wird, zu klein, um aussagekräftige Ergebnisse zu erhalten. Der Anweisungsblock wird inkrementell soweit vergrößert, daß die Ungenauigkeit höchstens 1 % beträgt. Dieses Verfahren ist üblich und wird auch in anderen Benchmarksammlungen wie PARKBENCH (Hockney und Berry, 1993) eingesetzt, um die Genauigkeit von Zeitmeßroutinen zu bestimmen.

4.3 Bestimmung des Prozessortaktes

Der Prozessortakt eines Prozessors läßt sich einfach bestimmen, wenn von einer Instruktion des Prozessors bekannt ist, wieviele Takte sie benötigt. In diesem Fall führt man diese Funktion so häufig aus, daß die Meßgenauigkeit ausreicht und kann sich so den Prozessortakt ausrechnen. Diese Methode funktioniert nur für bekannte Prozessortypen zuverlässig. Außerdem wird dadurch der Microbenchmark in der Portabilität eingeschränkt. Diese Methode wurde noch in lmbench 1.0 (1996), einer Benchmarksammlung von McVoy und Staelin (1996), eingesetzt.

McVoy und Staelin haben für lmbench 2.0 (1998) ihren Microbenchmark *mhz* in ANSI-C implementiert und eine portable Methode gefunden, den Prozessortakt zu bestimmen. Zunächst werden die Ausführungszeiten für neun verschiedene Anweisungsfolgen gemessen, die so ausgesucht sind, daß sie sich möglichst geringfügig in der Anzahl der benötigten Takte unterscheiden. Die Berechnung des größten gemeinsamen Teilers dieser Zeiten ergibt dann die Zeit, die ein Takt des Prozessors benötigt, wenn mindestens zwei der Anweisungsfolgen eine zueinander prime Taktanzahl (kein gleicher Primfaktor) benötigen. Um wegen der Meßgenauigkeit lange genug zu rechnen, wird die gleiche Anweisungsfolge (zum Beispiel die 5. Anweisungsfolge) wiederholt, bis die Ausführungszeit lang genug ist. Damit der Prozessor mindestens einen Takt je Anweisung benötigt und nicht zwei Anweisungen in einem Takt bearbeiten kann, sind die Anweisungen so gestaltet, daß für die Ausführung einer Anweisung das Ergebnis der vorherigen benötigt wird (streng sequentiell).

Die neun verschiedenen ANSI-C Anweisungsfolgen sind:

1.	p = *p;	6.	a ^= a « b;
2.	a ^= a + a;	7.	a ^= a + b;
3.	a ^= a + a + a;	8.	a += (a + b) & 07;
4.	a »= b;	9.	a ++; a ^= 1; a «= 1;
5.	a »= a + a;		

Angenommen die 4. Anweisungsfolge benötigt 12 ns und die 5. Anweisungsfolge benötigt 18 ns, dann kann über den größten gemeinsamen Teiler (in diesem Fall 6) die Zeit für einen Takt errechnet werden (maximal 6 ns). Wenn die Anzahl der tatsächlich benötigten Takte zweier Anweisungsfolgen zueinander prim ist, ist die wirkliche Taktdauer des Prozessors bestimmt (6 ns), weil kein gemeinsamer Faktor das Ergebnis verfälschen kann (sonst zum Beispiel auch 3 ns Taktdauer bei 2 als gemeinsamem Faktor oder 2 ns Taktdauer bei 3 als gemeinsamem Faktor). Da die Anweisungsfolgen aber extra so gewählt sind, daß sie sich möglichst wenig in der Anzahl der Takte, die für die Bearbeitung benötigt werden, unterscheiden, ist die Wahrscheinlichkeit, daß es nicht zwei Anweisungsfolgen gibt, die eine zueinander relativ prime Taktanzahl benötigen, gering.

Daß dieses Verfahren tatsächlich funktioniert, haben McVoy und Staelin dadurch gezeigt, daß sie es an vielen Prozessorarchitekturen erfolgreich ausprobiert haben. Zu ihnen gehören PA-RISC (PA-7000, PA-7200, PA-8000), Intel (486, Pentium, PentiumPro, Pentium II), DEC Alpha, PowerPC (PPC-603, PPC-604, PPC-604e), AMD (K5, K6), Sun (MicroSPARC, SuperSPARC, Ultra-I, Ultra-II), MIPS (R4000, R5000, R6000), Cyrix und Motorola 68020. Außerdem wurden für die Tests verschiedene Betriebssysteme genutzt.

VLIW McVoy und Staelin erwarten, daß mit diesen Anweisungen aufgrund der sequentiellen Abhängigkeit der Anweisungen auch auf Prozessoren mit „Very Long Instruction Word" (VLIW) Architektur, wie der IA-64 Architektur, richtige Ergebnisse ermittelt werden können.

4.4 Latenzzeiten von Speicherebenen

Mit einem Microbenchmark, der die Latenzzeiten der verschiedenen Speicherebenen bei Lesezugriffen mißt, kann man herausfinden, wie lange der Prozessor auf angeforderte Daten warten muß, ehe sie in einem Register vorliegen. Das Messen der Latenzzeiten für das Schreiben von Daten ist schwieriger, da man nicht so leicht erkennen kann, wann ein Schreibvorgang beendet ist. Verschiedene Schreibstrategien, siehe Abschnitt 2.2 auf Seite 7, verbergen tatsächliche Latenzzeiten beim Schreiben vor dem Prozessor.

Mit genauer Kenntnis über Speicherlatenzzeiten kann ein Programmierer oder Compilerbauer versuchen, die Wartezeiten mit sinnvollen Instruktionen zu füllen, indem Anweisungen umgeordnet werden.

Wartezeiten nutzen

Im Rahmen dieser Arbeit wurde ein nach dem im folgenden beschriebenen Verfahren arbeitender Microbenchmark zum Messen von Latenzzeiten implementiert und angewandt.

Messen von Speicherlatenzzeiten

Um die Latenzzeiten des L1 Caches, L2 Caches und Hauptspeichers zu messen, werden Ladeoperationen, mit verschiedenen Abständen der geladenen Adressen zueinander, durchgeführt. Ladeoperationen über verschieden große Speicherbereiche hinweg können Kapazitätsgrenzen von Zwischenspeichern anhand steigender Latenzzeiten deutlich machen, weil größere Zwischenspeicher üblicherweise auch längere Latenzzeiten haben.

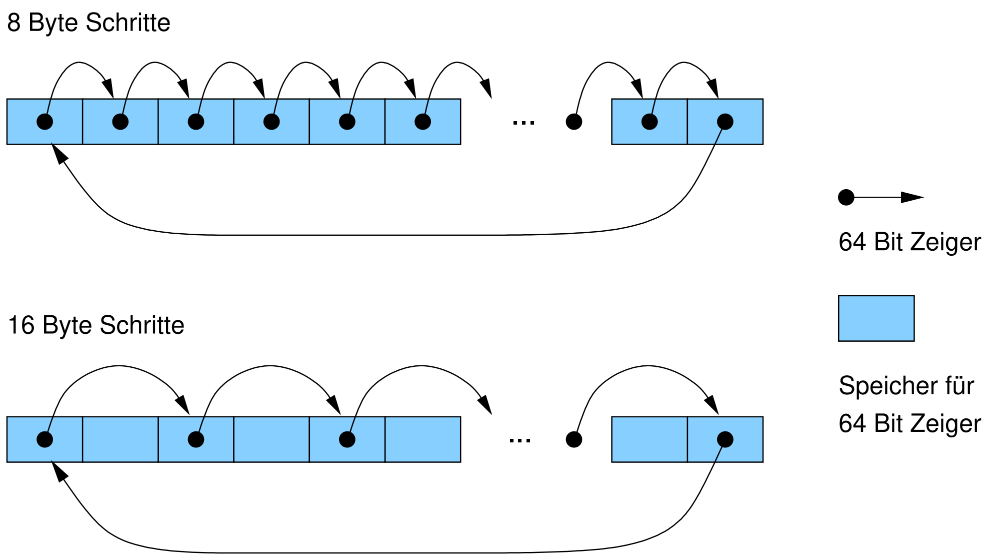


Abbildung 4.1: Die obige Datenstruktur, mit der Ladeoperationen sukzessive gezielt auf Adressen zugreifen, kann zum Messen von Speicherlatenzzeiten genutzt werden. Um verschiedene Speicherebenen zu erreichen, wird der Abstand zwischen aufeinanderfolgend geladenen Adressen und der zu durchlaufende Speicherbereich immer weiter vergrößert. Irgendwann sind Blockgröße und Schrittweite so groß, daß der Ln Cache komplett überschrieben wird und jeder Speicherzugriff ein Laden aus dem Ln+1 Cache bewirkt.

Dazu werden Speicherbereiche, wie in Abbildung 4.1 dargestellt, benutzt. Der Benchmark lädt entlang der Referenzen immer den Inhalt der nächste Zelle und benutzt diesen Inhalt als Referenz auf die nächste Zelle. Der Prozessor kann nicht mehrere Ladeoperationen gleichzeitig verarbeiten, da die Ladeoperationen streng sequentiell voneinander abhängen. Der Benchmark benötigt außer der Ladeoperation und ein paar Operationen zur Schleifenverwaltung keine anderen Operationen. Die durchschnittliche Latenzzeit ergibt sich aus der gemessenen Gesamtausführungszeit geteilt durch die Anzahl der durchgeführten Ladeoperationen. Um die Messung zu verbessern, kann jede Ladeoperation mit einem Verwaltungsaufwand von einem Prozessortakt angenommen werden und zusätzlich zum Schleifenverwaltungsaufwand vom gemessenen Ergebnis subtrahiert werden.

Architektur:	**Alpha 21164**	**Pentium II/III**	**IBM RS/6000**
Prozessor			
MHz	600	400/500	25
Gleitkommazahl Register	32	8	32
L1 Cache			
Gesamtgröße	8 kB	16 kB	128 kB
Blockgröße	32 Bytes	32 Bytes	1024 Bytes
Cache-Zeilen	256	512	128
Assoziativität	1	2	4
Ersetzungsstrategie	entfällt	lru	random
Latenzzeit	2 Takte	3 Takte	1 Takt
L2 Cache			**Hauptspeicher**
Gesamtgröße	96 kB	256 kB	48 MB
Blockgröße	64 Bytes	32 Bytes	4096 Bytes
Cache-Zeilen	1536	8192	12288
Assoziativität	3	4	–
Ersetzungsstrategie	random	lru	random
Latenzzeit	8 Takte	20 Takte	40 Takte

Tabelle 4.1: Die Parameter verschiedener Speicherhierarchien beeinflussen deren Ausnutzung. Angegeben sind die Daten eines Alpha 21164 in einer Cray T3E, eines Pentium II Prozessors in einem PC mit 100 MHz Frontsidebus und einer IBM RS/6000.

Die Speicherbereiche werden für jede Messung mehrmals durchlaufen. Die Größe des zu durchlaufenden Speicherblocks wird für verschiedene Messungen variiert, um die Kapazitätsgrenzen der einzelnen Zwischenspeicher zu ermitteln. Bei kleinen zu durchlaufenden Blöcken passen alle Daten komplett in einen Zwischenspeicher. Ist der zu durchlaufende Block größer als der Zwischenspeicher, müssen die Daten aus dem Ln+1 Cache geladen werden, der längere Latenzzeiten hat.

Wenn Daten noch nicht im Zwischenspeicher enthalten sind, wird immer eine ganze Cache-Zeile in den Zwischenspeicher kopiert. Zugriffe auf folgende, benachbarte Adressen sind dann solange schneller, bis die Cache-Zeile diese Daten wegen ihrer begrenzten Länge nicht mehr bereitstellt und erneut Daten in den Cache kopiert werden müssen. Wenn die Cache-Zeilenlänge zu gering ist, können schrittweise Speicherzugriffe dazu führen, daß jeder Zugriff das Kopieren einer Cache-Zeile in den Cache erfordert.

An den Latenzzeitdiagrammen, die mit dem Latenzzeit-Microbenchmark erstellt werden, kann man die Latenzzeiten der verschiedenen Speicherebenen, deren Gesamtgröße und die jeweilige Blockgröße ablesen, wie die folgenden Abschnitte an zwei Beispielen zeigen werden. Aus der Blockgröße und der Gesamtgröße läßt sich dann einfach die Cache-Zeilenanzahl berechnen. Die tatsächlichen Daten der Speicherhierarchien der beiden im folgenden betrachteten Systeme sind in Tabelle 4.1 dargestellt.

Speicherlatenzzeiten auf einer Cray T3E

Die Cray T3E benutzt einen DEC Alpha 21164 Prozessor mit L1 und L2 Cache. Außerdem hat die Cray T3E sogenannte „Stream Buffers“ (ANDERSON et al. 1997; OED 1996), die konsekutive Zugriffe auf den Hauptspeicher beschleunigen können. Für die Messungen wurden diese Stream Buffers jedoch abgeschaltet, um die Ergebnisse besser mit denen anderer Speicherhierarchien vergleichen zu können. Die Resultate des Benchmarks zum Messen von Latenzzeiten sind in Abbildung 4.2 abgebildet. Es sind hauptsächlich drei verschiedene Latenzzeiten erkennbar. Für kleine Speicherblöcke bis zu 8 kB passen alle Daten in den L1 Cache, der eine Latenzzeit von 3,4 ns hat (übrigens: Licht legt im Vakuum in dieser Zeit eine Strecke von etwa 1 m zurück).

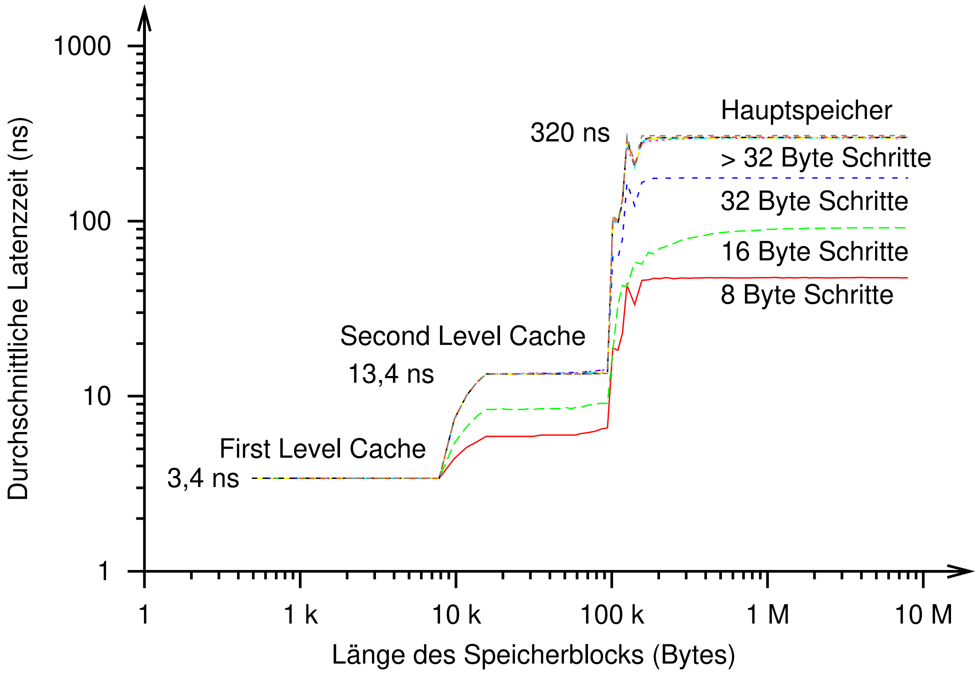


Abbildung 4.2: Speicherlatenzzeiten auf einem 600 MHz Alpha 21164 Prozessor in einer Cray T3E bei aufeinanderfolgenden Ladeoperationen. Die Schrittweite zwischen zwei Ladeoperationen variiert von Kurve zu Kurve.

Für Speichergrößen zwischen 8 und 96 kB sind die Daten im L2 Cache zwischengespeichert, der eine Latenzzeit von 13,4 ns hat. Speicherblöcke, die mit einer Schrittweite von 8 oder 16 Bytes durchlaufen werden, haben im Durchschnitt kürzere Latenzzeiten, weil ein Teil der Speicherzugriffe aus dem L1 Cache beantwortet werden kann. Bei Speicherzugriffen in 8 Byte Schritten kann jeder vierte Speicherzugriff *nicht* vom L1 Cache erfüllt werden (durchschnittliche Latenzzeit ist (3 L1-Latenzzeit + 1 L2-Latenzzeit) / 4). Bei einer Schrittweite von 16 Bytes kann jeder zweite Speicherzugriff nicht vom L1 Cache erledigt werden (durchschnittliche Latenzzeit ist (1 L1-Latenzzeit + 1 L2-Latenzzeit) / 2). Daraus kann man folgern, daß der L1 Cache Cache-Zeilen mit 32 Bytes hat, da bei 16 Byte Schritten, jedoch nicht mehr bei 32 Byte Schritten, noch ein Teil der Ladeoperationen vom L1 Cache beantwortet werden können.

Im Bereich zwischen 100 kB und 10 MB der Abbildung 4.2 kann man die Latenzzeiten des Hauptspeichers (320 ns) ablesen. Wie schon beim L2 Cache können auch beim Hauptspeicher bei kleiner Schrittgröße noch einige Anfragen vom L1 oder L2 Cache beantwortet werden. Entsprechend läßt sich folgern, daß der L2 Cache eine Blockgröße von 64 Bytes hat, weil dies die erste Schrittweite ist, bei der jeder Zugriff zu einem Cache-Miss im L1 und L2 Cache führt.

Wechsel der Speicherebene

Allgemein steigt beim Erreichen der Kapazitätsgrenze eines Caches (Ln Cache) und dem Wechsel in den nächsten Cache (Ln+1 Cache) die Latenzzeit nicht direkt auf die Latenzzeit des Ln+1 Caches an, sondern nähert sich dieser langsam. Dies liegt daran, daß der durchlaufene Speicherblock beim Übergang Größen erreicht, bei denen bei einem Durchlaufen des Blocks der Ln Cache nicht komplett überschrieben wird. Dadurch beantwortet der Ln Cache bei einem weiteren Durchlaufen des Speicherblocks noch einen Teil der Speicherzugriffe mit seiner Latenzzeit.

Speicherlatenzzeiten eines Pentium II Prozessors

Abbildung 4.3 zeigt die Ergebnisse des Benchmarks zum Messen von Speicherlatenzzeiten für einen Intel Pentium II Prozessor. An dem Diagramm läßt sich ablesen, daß die Latenzzeit des L1 Cache 7,5 ns, die des L2 Cache 49,9 ns und die des Hauptspeichers maximal 190 ns beträgt. Außerdem zeigt die Abbildung, daß der L1 Cache eine Größe von 16 kB und der L2 Cache eine Größe von 256 kB hat. L1 und L2 Cache haben die gleiche Blockgröße von je 32 Bytes.

Im Gegensatz zu einem Alpha 21164 in einer Cray T3E sind die Latenzzeiten für einen Pentium II Prozessor für den L1 und L2 Cache größer, die Anbindung zum Hauptspeicher ist aber mit geringeren Latenzzeiten verbunden, wie Abbildung 4.3 zeigt. Kleine Abweichungen beim Wechsel zum L2 Caches werden dadurch hervorgerufen, daß es beim Pentium II zwar für Programmcodes und Daten getrennte L1 Caches gibt, der L2 Cache aber für das Zwischenspeichern von Programm und Daten genutzt wird. Das Zwischenspeichern des Programmcodes verhindert also die optimale Nutzung des L2 Caches.

Im Bereich des Hauptspeichers entstehen mehr Abweichungen als erwartet, weil die Latenzzeiten der Speicherchips je nach Zugriffsmuster variieren können (WINDECK, 2000). Speicher sind so gebaut, daß sie Zugriffe auf nahe beieinander liegende Adressen schnell bearbeiten können. Bei weiter auseinanderliegenden Adressen muß speicherintern auf andere Bereiche umgeschaltet werden, so daß Schalt- und Latenzzeiten ansteigen. Außerdem muß der Inhalt der Speicher regelmäßig aufgefrischt werden. Wie oft dies geschieht hängt von den Speicherbausteinen ab. Üblich sind Auffrischungen (Refreshrate) im Abstand von 64 ms bei dynamischen Speichern (WINDECK, 2000).

4.5 Speicherbandbreiten

STRICKER und GROSS (1995) beschreiben ein Modell zur Abschätzung von Speicherbandbreiten mittels ihrer „Extended Copy Transfer Characterization“. Um ihr Modell zu überprüfen, haben sie einen Microbenchmark geschrieben, mit dem man Speicherbandbreiten messen kann. Genauso wie beim Messen von Speicherlatenzzeiten benutzen sie

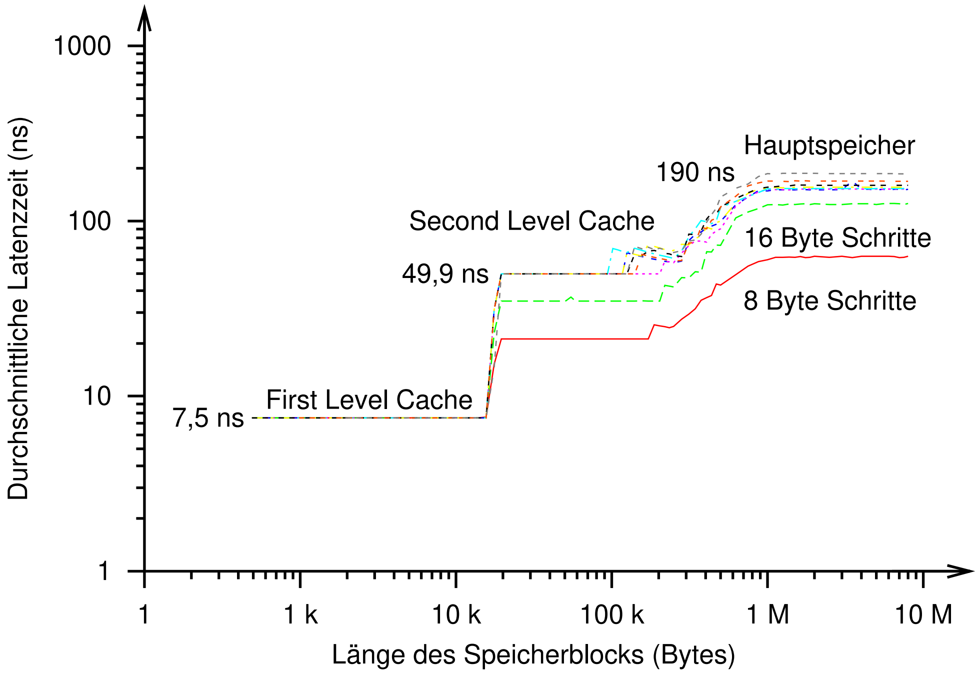


Abbildung 4.3: Speicherlatenzzeiten eines Pentium II Prozessors mit 400 MHz bei aufeinanderfolgenden Ladeoperationen. Die Schrittweite zwischen zwei Ladeoperationen variiert von Kurve zu Kurve.

einen Speicherbereich, dessen Größe variiert wird, und laden Daten mit unterschiedlicher Schrittweite. Im Unterschied zum Messen von Speicherlatenzzeiten sind die Ladeoperationen voneinander unabhängig und nicht streng sequentiell, so daß der Prozessor mehrere Ladeoperationen in schneller Folge ausführen kann. Dieser Benchmark kann auch für das Messen der Bandbreite nicht lokaler Speicherzugriffe genutzt werden, wie STRICKER und GROSS (1997) zeigen.

4.6 Speicheroperationen im Verhältnis zu Rechenoperationen: Horner-Schema

Bei einer Analyse verschiedener Verhältnisse von Speicheroperationen zu Rechenoperationen erhält man Daten darüber, unter welchen Bedingungen ein Prozessor mit hoher Leistung rechnen kann. Durch diese Daten kann bestimmt werden, wieviele Rechenoperationen ein Prozessor auf den in Registern vorhandenen Daten durchführen muß, ehe er auf angeforderte, externe Daten zugreift, wenn man möglichst nahe an die Leistungsgrenze des Prozessors kommen möchte. Werden zu häufig Daten angefordert, können die Speicher diese nicht mehr mit der nötigen Geschwindigkeit liefern.

Horner-Schema

Um diese Eigenschaft zu analysieren, eignet sich das Horner-Schema (HORNER, 1819) zum Auswerten von Polynomen. Das im folgenden beschriebene Verfahren wurde auch vom PARKBENCH Comittee eingesetzt (HOCKNEY und BERRY, 1993, Seite 22). Mit diesem Schema läßt sich die Anzahl der Rechenoperationen im Verhältnis zu den zu ladenden Daten sehr einfach variieren. Die einzigen benötigten Operationen sind Laden, Speichern, Addieren und Multiplizieren.

Nach dem Horner-Schema wird der Wert eines Polynoms an einer Stelle x folgendermaßen berechnet:

$$\sum_{i=0}^{n} a_i x^i = \left(\left(\dots\left((a_n)x + a_{n-1}\right)\dots\right)x + a_1\right)x + a_0$$

In einem Microbenchmark nach dem Horner-Schema wertet man ein Polynom in einer Schleife an verschiedenen Stellen aus. Es lassen sich die Anzahl der Auswertungsstellen und der Grad des Polynoms (die Anzahl der benötigten Koeffizienten und damit der Rechenaufwand pro geladenen Daten) variieren. Dadurch bekommt man einen Überblick über die Charakteristik der Speicheranbindung des Prozessors.

---

polynomal [−h] [−l max_length] [−d max_degree]
    [−m {muladd, mul, add}] [−s {memory, register}]

Die Parameter bedeuten im einzelnen:

−h		Eine Beschreibung der Aufrufstruktur wird als Hilfe ausgegeben.
−l	max_length	Die maximale Anzahl der Auswertungsstellen (Voreinstellung 1.048.576 = 1024^2).
−d	max_degree	Der maximale Grad des Polynoms, das ausgewertet werden soll (Voreinstellung 18, implementationsbedingt höchstens 40).
−m	{muladd, mul, add}	Die Art der Operation, die benutzt wird. Entweder eine muladd Operation, oder nur add oder nur mul Operationen (Voreinstellung muladd).
−s	{memory, register}	Bei memory werden die Resultate der Auswertung in einen Vektor im Speicher geschrieben, bei register werden die Resultate an eine feste Adresse geschrieben (Voreinstellung memory).

Abbildung 4.4: Die Handbuchseite des Horner-Schema-Benchmarks zeigt, daß mit dem Benchmark auch die benutzten Operationen modifiziert werden können, um beispielsweise die Geschwindigkeit von Multiplikation und Addition zu vergleichen. Mathematisch berechnen diese Varianten etwas anderes als eine Auswertung eines Polynoms nach dem Horner-Schema.

Der Horner-Schema-Benchmark, der in der Benchmarksammlung PARKBENCH in Fortran implementiert ist, wurde im Rahmen dieser Arbeit in ANSI-C reimplementiert und flexibler gestaltet. Die Aufrufsyntax und Variationsmöglichkeiten der Reimplementation zeigt die Handbuchseite in Abbildung 4.4. Bei der Implementation wurde darauf geachtet, daß ein Compiler für jeden Polynomgrad optimalen Code erzeugen kann. Um dies zu erreichen, wurde für jeden Grad eine Funktion mit fester Anzahl von Schleifendurchläufen generiert, so daß der Compiler diese Schleifen komplett ausrollen kann, damit weniger Verwaltungskosten für Schleifen entstehen und die Register optimal genutzt werden können.

Alpha 21164 Prozessor mit 600 MHz

Die in Abbildung 4.5 dargestellen Resultate des Horner-Schema-Benchmarks für einen Alpha 21164 lassen sich wie folgt interpretieren. Bei niedrigen Graden des Polynoms (2 – 6) und wenig Auswertungsstellen (1 – 50) wird wenig gerechnet, und es entsteht ein hoher Verwaltungsaufwand für die Schleifen, die oft durchlaufen werden, damit die Meßzeit ausreichend lang ist. Bei einer mittleren Anzahl von Auswertungsstellen (50 – 10000) und niedrigem Polynomgrad bleiben die Koeffizienten des Polynoms in Registern, und nur die Daten für die Auswertungsstellen müssen geladen werden, so daß der Speicherbus zum Engpaß wird. Je höher der Grad des Polynoms wird, desto mehr wird mit der auszuwertenden Stelle ohne weitere Speicheroperationen gerechnet, und um so besser wird die Leistung, weil die Wartezeiten durch Rechenoperationen überbrückt werden können. Bei Graden von 6 – 10 bleibt die Leistung relativ konstant, weil der Prozessor nicht mehr ohne dabei zu rechnen auf Daten vom Speicher warten muß und so seine Höchstleistung erreicht. Steigt der Grad des Polynoms noch weiter, sinkt die Leistung wieder ab, weil nicht mehr alle Koeffizienten des Polynoms in Registern des Prozessors gehalten werden können. Dies macht sich mit wachsendem Polynomgrad immer deutlicher bemerkbar.

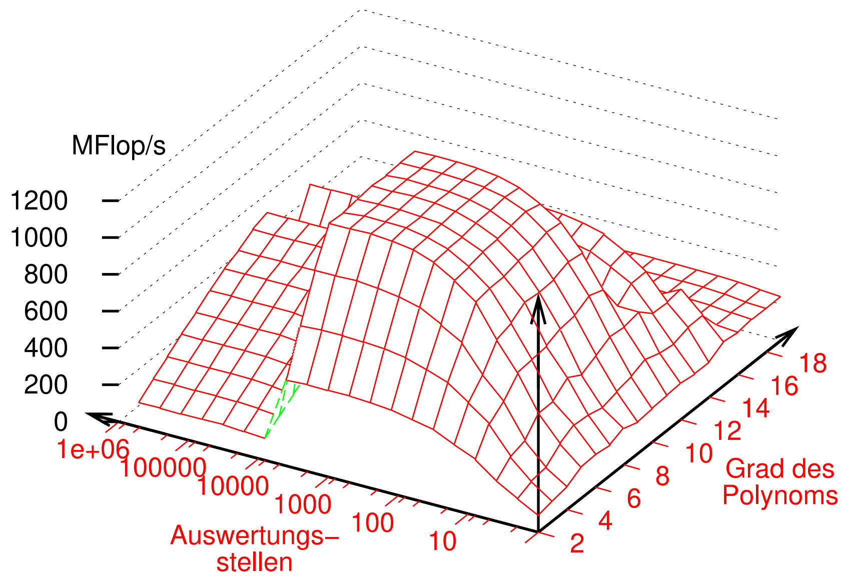


Abbildung 4.5: Die Ergebnisse des Horner-Schema-Benchmarks für einen 600 MHz Alpha 21164 in einer Cray T3E zeigen, daß das Verhältnis von Lade- zu Rechenoperationen starken Einfluß auf die erreichbare Leistung hat.

Bei einer großen Anzahl von Auswertungsstellen (mehr als 10000) können diese nicht mehr im L1 Cache untergebracht werden, und die Latenzzeiten zum Laden der benötigten Daten steigt, so daß der Prozessor nicht mehr bis zur Höchstleistung ausgelastet wird.

An die theoretische Höchstleistung von 1200 MFlop/s kommt man mit diesem Microbenchmark mit im besten Fall 1161 MFlop/s bei Polynomgrad 11 und 2048 Auswertungsstellen sehr nahe heran. Dafür lief der Prozeß auf einem einzelnen Knoten auf einer Cray T3E und wurde nicht vom Betriebssystem unterbrochen, so daß die Caches nur von diesem Prozeß genutzt wurden.

Pentium II mit 400 MHz

Bei einem Pentium II System unter Linux bemerkt man die Bedeutung eines guten Compilers, der effizienten Code erzeugt. Die Abbildungen 4.6 und 4.7 zeigen die Ergebnisse, die mit zwei verschiedenen Compilern in der gleichen Testumgebung gemessen wurden. Durch die relativ geringe Registeranzahl (8 Register für Gleitkommazahlen im Gegensatz zu 32 entsprechenden Registern im Alpha 21164 und höhere Latenzzeiten für den L1 Cache) ist es für die Compiler schwierig, die Leistung optimal zu nutzen.

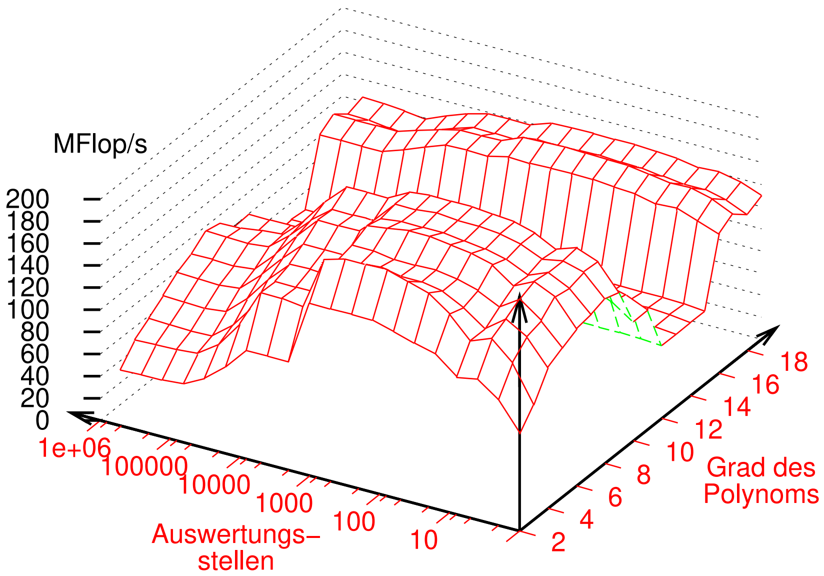


Abbildung 4.6: Auf einem Pentium II mit 400 MHz liefert eine Messung mit der Evaluierungsversion des Fujitsu C-Compilers unter Linux beim Horner-Schema-Benchmark diese Charakteristik.

Beim Fujitsu C-Compiler gibt es einen großen Leistungseinbruch bei Polynomen mit Graden zwischen 12 und 15. Dies läßt sich ohne Kenntnis interner Compilerdetails nur schwer begründen, könnte aber daran liegen, daß eine Optimierungsstufe, die für kleinere Grade funktioniert hat, nicht mehr anwendbar ist und eine andere, die für Grade ab 16 optimieren kann, noch nicht einsetzbar ist. Davon abgesehen zeigt sich ein mit dem Alpha 21164 grundsätzlich vergleichbares Bild. Die theoretische Höchstleistung von 400 MFlop/s wird allerdings nicht annähernd erreicht (Maximum 190 MFlop/s bei Polynomgrad 3 und 128 bis 512 Auswertungsstellen). Dies liegt an der sehr geringen Registeranzahl, wodurch häufig auf den Speicher zugegriffen werden muß. Die relativ schlechte Speicheranbindung des L1 und L2 Caches in PCs kann diese Speicherzugriffe nicht mit der nötigen Geschwindigkeit abarbeiten, so daß der Prozessor oft auf Daten warten muß.

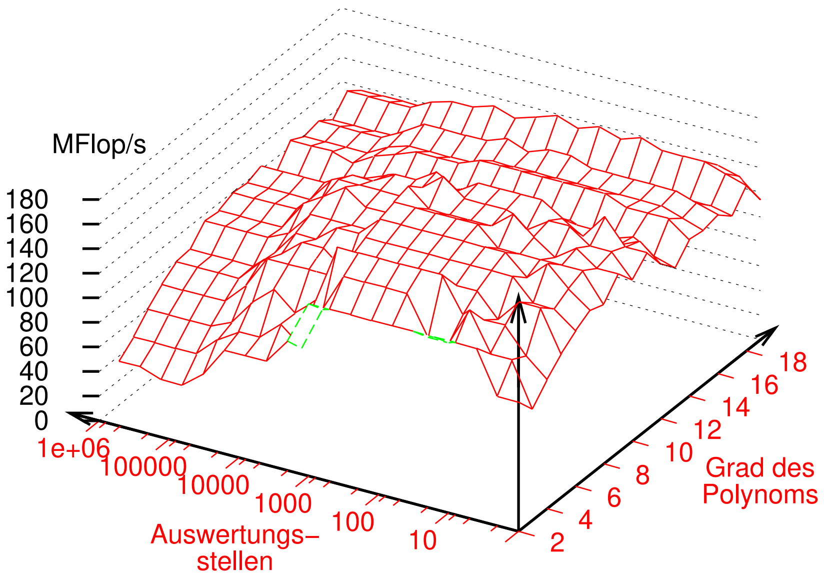


Abbildung 4.7: Die Messungen mit der GCC 2.95.2 für Linux auf einem Pentium II mit 400 MHz ergeben ein nicht ganz gleichmäßiges Diagramm für den Horner-Schema-Benchmark. Daran läßt sich erkennen, daß die optimale Nutzung der wenigen Register der IA-32 Architektur schwierig ist.

Bei der GNU Compiler Collection (GCC) wird die maximale Leistung bei Polynomgrad 10 und 512 Auswertungsstellen mit 174 MFlop/s erreicht. Auch wenn mit der GCC nicht die Leistung erreicht wird, die der Fujitsu C-Compiler maximal liefert, ist das Leistungsbild wesentlich gleichmäßiger, was dem Programmierer erleichtert, diese Systemleistung effektiv für seine Applikation zu nutzen. Daß die maximale Leistung bei einem im Verhältnis zur Registeranzahl hohen Polynomgrad auftritt, überrascht, wird aber durch die relativ ähnlichen Leistungswerte in der Umgebung relativiert und könnte auf Einflüsse des Betriebssystems während der Messung zurückzuführen sein.

Diese zwei Beispiele (Alpha 21164 und Pentium II) zeigen, daß eine sinnvoll gestaltete Speicherhierarchie für eine gute Auslastung des Prozessors wichtig ist. Beide Prozessoren benutzen zwei Zwischenspeicher, und obwohl der Hauptspeicher des Alpha 21164 längere Latenzzeiten hat als der des Pentium II, kann man die Leistung des Alpha 21164, weil er mehr Register hat und der L1 Cache schneller ist als der des Pentium II, besser nutzen.

Simulation von Speicherhierarchien

A good simulation, be it a religious myth or scientific theory,
gives us a sense of mastery over experience.

– HEINZ R. PAGELS (1939 – 1988)

Simulatoren für Speicherhierarchien dienen der Analyse und Optimierung von Speicherhierarchien. Unterschiedliche Verfahren zur Realisierung von solchen Simulatoren wurden mit der Zeit entwickelt und werden in diesem Kapitel beschrieben. Verschiedene Techniken der Instrumentierung werden präsentiert und ihre Vor- und Nachteile betrachtet. Abschließend werden einige Simulatoren für Speicherhierarchien vorgestellt, um einen Einblick in die unterschiedlichen Möglichkeiten zu geben.

5.1 Zweck und Arten von Simulatoren für Speicherhierarchien

Eine gute Nutzung einer Speicherhierarchie eines Prozessors ist sehr wichtig, um die Rechenleistung eines Prozessors so gut wie möglich auszunutzen, wie bereits die Untersuchung des Horner-Schema-Benchmarks in Kapitel 4 ab Seite 21 gezeigt hat. Mit einem Simulator für Speicherhierarchien kann man Speicherzugriffe auf die verschiedenen Speicherebenen analysieren und versuchen, Speicherzugriffsmuster einer Anwendung zu optimieren. Außerdem werden Simulatoren von Speicherhierarchien dazu eingesetzt, Speicherhierarchien für Prozessoren zu optimieren und Kosten-Nutzen-Abschätzungen durchzuführen. Dabei werden Beispielapplikationen mit verschiedenen Speicherhierarchien simuliert und die Ausnutzung der Zwischenspeicher (Verhältnis von Cache-Hits zu Cache-Misses) berechnet.

Die Ansteuerung von Simulatoren von Speicherhierarchien können im wesentlichen auf zwei verschiedene Arten durchgeführt werden. Dementsprechend unterscheidet man zwei Arten von Simulatoren:

Trace Driven: Bei einem durch ein Laufprotokoll gesteuerten Cachesimulator (Trace Driven) wird eine Anwendung so modifiziert, daß sie alle nötigen Daten, die der Cachesimulator benötigt, während der Ausführungszeit speichert (Laufprotokoll, Trace). Der Simulator analysiert dann nach dem Programmlauf die aufgezeichneten Daten. Die gespeicherten Daten können eventuell auch mehrmals mit verschiedenen Einstellungen analysiert werden, solange sie auch für diese Einstellungen gültig sind.

Execution Driven: Bei einem ausführungsgesteuerten Simulator (Execution Driven) wird ein Programm so modifiziert, daß es Routinen des Cachesimulators aufruft, während es läuft. Für diese Cachesimulatoren wird kein Laufprotokoll benötigt.

5.2 Instrumentierung von Programmen

Unabhängig von der Art des Simulators muß das zu simulierende Programm so modifiziert werden, daß es die nötigen Eingaben für den Simulator liefert (entweder in Form von gespeicherten Daten oder in Form von Funktionsaufrufen). Diese Modifikation nennt man Instrumentierung.

Bei protokollgesteuerten Simulatoren ist das Ziel der Instrumentierung, daß das Programm ein Protokoll erzeugt, das dann vom Simulator gelesen werden kann. Bei ausführungsgesteuerten Simulatoren wird durch die Instrumentierung das Programm so modifiziert, daß an allen nötigen Stellen Routinen des Simulators aufgerufen werden.

Unabhängig von der Ansteuerung des Simulators sind zwei verschiedene Arten der Instrumentierung üblich: entweder wird ein ausführbares Programm oder der Quelltext eines Programms instrumentiert.

Instrumentierung des ausführbaren Programms

Für die Instrumentierung von ausführbaren Programmen gibt es Bibliotheken, wie die „Executable Editing Library" (EEL) von LARUS und SCHNARR (1995), die eine Vielzahl verschiedener Plattformen und Binärformate unterstützen. Die EEL ist eine in C++ geschriebene Bibliothek und benutzt ihrerseits die „Binary File Descriptor Library" (CHAMBERLAIN, 1991). Mit Hilfe der EEL läßt sich der Kontrollflußgraph eines ausführbaren Programms berechnen. Die Knoten dieses Graphen beschreiben Blöcke von sequentiell ausgeführten Codeteilen, die Kanten Sprunganweisungen für Funktionsaufrufe oder Schleifen. Mit der EEL können Programmfragmente an diese Kanten angebracht werden. Die EEL erzeugt dann aus dem modifizierten Kontrollflußgraphen ein neues, ausführbares Programm, das so arbeitet wie das Originalprogramm und zusätzlich bei einem Kontrollflußwechsel die vorher im Graphen angebrachten Programmfragmente (zum Beispiel einen Aufruf einer Simulatorfunktion) ausführt.

Einfluß auf die Portabilität

Die Portabilität ist bei einer Instrumentierung von ausführbaren Programmen nicht in dem Maße gegeben wie bei einer Instrumentierung des Quelltextes, da sich auf einer neuen Rechnerarchitektur der Instruktionssatz, die Registeranzahl, die Semantik von Instruktionen oder das Format der ausführbaren Dateien ändern können. Bei Nutzung einer Bibliothek wie der EEL ist diese für eine neue Architektur in jedem Fall anzupassen, wofür detaillierte Kenntnisse über die Maschineninstruktionen nötig sind.

Instrumentierung des Quelltextes

Eine Instrumentierung des Quelltextes setzt voraus, daß der Quelltext der zu untersuchenden Anwendung verfügbar ist. Allerdings ist eine Analyse durch einen Simulator für Speicherhierarchien zur Optimierung von Programmen nur sinnvoll, wenn man das Programm danach auch ändern kann, so daß diese Einschränkung selten Einfluß hat. Die Instrumentierung des Quelltextes kann entweder automatisch über ein Instrumentierungsprogramm erfolgen oder von Hand vorgenommen werden.

Eine Quelltextinstrumentierung von Hand ist zwar aufwendig, dafür aber auch wesentlich flexibler als eine automatische Instrumentierung, weil der Simulator ganz gezielt aufgerufen werden kann. Eine automatische Instrumentierung dagegen ist nicht so fehleranfällig, kann aber auch dazu führen, daß sich das Programm mit dem eingesetzten Compiler nicht mehr kompilieren läßt, weil durch die Instrumentierung Konstrukte erzeugt wurden, die der genutzte Compiler nicht übersetzen kann.

5.3 Funktionsweise von Simulatoren für Speicherhierarchien

Es gibt viele Simulatoren für Speicherhierarchien, die sich zum Beispiel im Detaillierungsgrad der Simulation oder der simulierbaren Speicherhierarchien unterscheiden, unterschiedliche Informationen für die Berechnung benötigen und auch unterschiedliche Informationen produzieren.

Ein Simulator für Speicherhierarchien benötigt Informationen darüber, auf welche Speicheradressen ein Programm zugreift und ob diese Zugriffe lesend oder schreibend sind. Diese Informationen stellen eine entsprechende Instrumentierung des Programms zur Verfügung. Außerdem benötigt ein Simulator für Speicherhierarchien zusätzlich Daten über den Aufbau der Speicherhierarchie, die simuliert werden soll.

Stehen diese Informationen zur Verfügung, vollzieht der Simulator die Zugriffe auf den Speicher nach und errechnet, in welcher Speicherebene die Daten zwischengespeichert werden, zählt Cache-Hits und Cache-Misses, zählt, wie häufig eine Cache-Zeile genutzt wurde, bevor sie wieder ersetzt wurde, summiert die Latenzzeiten, die mit Speicherzugriffen auf verschiedene Speicherebenen verbunden sind oder berechnet und sammelt andere Daten, die schließlich abgefragt werden können.

Ein Cachesimulator benötigt intern Datenstrukturen und Mechanismen, die die Verwaltung von Zwischenspeichern nachbilden. Mit Hilfe dieser Datenstrukturen wird festgestellt, welche Cache-Zeilen belegt und welche frei sind, welche als nächste zu ersetzen sind und welche Adressen in welchen Cache-Zeilen gespeichert sind. Die internen Datenstrukturen eines Cachesimulators hängen vom Detaillierungsgrad und der Flexibilität des Simulators ab.

Einige Simulatoren simulieren sehr detailliert das Verhalten des Prozessors oder der Auftragspuffer der Cache-Controller, um bessere Berechnungen der Ausführungszeit durchführen zu können. Je nachdem wie detailliert unterschiedliche Aspekte einer Speicherhierarchie simuliert werden, differieren auch die Daten, die für die Simulation benötigt werden und der Aufwand, den die Simulation verursacht.

5.4 Beispiele für Simulatoren von Speicherhierarchien

Bei der Entwicklung neuer Computer werden Simulatoren für Speicherhierarchien besonders häufig eingesetzt, um die Leistungsfähigkeit verschiedener Speicherhierarchien abschätzen zu können. Aber auch um den Einfluß neuer Architekturen von Zwischenspeichern auf verschiedene Programme zu analysieren, werden Simulatoren programmiert. In den folgenden Abschnitten sollen verschiedene Simulatoren für Speicherhierarchien exemplarisch vorgestellt werden.

Der Active Memory Cachesimulator

Der Speichersimulator „Active Memory“ von LEBECK und WOOD (1995) ist ein protokollgesteuerter Simulator. Dabei können während der sich an die Ausführung des Programms anschließenden Analyse eigene Funktionen an Ereignisse wie Cache-Miss oder Cache-Hit gekoppelt werden, so daß der Nutzer große Freiheit hat, die erzeugten Daten auszuwerten. Der Idee, an Ereignisse wie Cache-Miss oder Cache-Hit benutzerdefinierte Funktionen binden zu können, hat dieser Simulator auch seinen Namen zu verdanken.

Trigger Funktionen

Die Instrumentierung findet an ausführbaren Programmen mit Hilfe der Executable Editing Library (LARUS und SCHNARR, 1995) statt (siehe Abschnitt 5.2).

Der Performance Estimator for RISC Microprocessors

Der „Performance EstimatoR for RISC Microprocessors“ (PERFORM) ist ein ausführungsgesteuerter Cachesimulator. DUNLOP und HEY (1997) verfolgen mit ihm den Ansatz alle Daten und Variablen, die nicht den Kontrollfluß eines Programms beeinflussen, zu eliminieren, wodurch der Speicherplatzbedarf und der Simulationsaufwand stark reduziert werden. Diese Reduzierung wird automatisch am Quelltext, der in Fortran vorliegen muß, in mehreren Schritten durchgeführt. Zuerst werden für die Ermittlung der Programmlaufzeit unnötige Operationen eliminiert, dann wird das Programm mit Aufrufen des Cachesimulators versehen, und schließlich können Teilergebnisse aus vorhergegangenen Simulationen übernommen werden, um die Simulationszeit weiter zu verringern.

Der Simulator ist maschinenunabhängig und liest Maschinendaten aus einer Konfigurationsdatei. Dadurch daß alle Änderungen am Quelltext automatisch ausgeführt werden, kann PERFORM auch in anderen Werkzeugen, vor dem Benutzer verborgen, eingesetzt werden. Allerdings ist PERFORM auf Fortran beschränkt, was das Einsatzgebiet eingrenzt.

Der Multi Lateral Cache Simulator

Der Simulator mlcache („Multi Lateral Cache Simulator“) von TAM et al. (1998) kann Cache-Konfigurationen simulieren, bei denen parallele Caches horizontal Daten austauschen können (lateral). Zum Beispiel kann ein Direct-Mapped-Cache von einem kleinen vollassoziativen Cache unterstützt werden, so daß seltener Cache-Misses auftreten. Parallele

Caches können unterschiedliche Ersetzungsstrategien haben, wodurch ein Cache manchmal Daten bereithält, die ein anderer schon wieder verworfen hat. Dadurch, daß nicht ein großer Cache vollassoziativ sein muß, sondern nur ein kleiner, wird Verwaltungsaufwand gespart und eventuell Geschwindigkeit gewonnen.

Der in ANSI-C implementierte Simulator wird über eine Quelltextdatei konfiguriert, in der man zum Beispiel programmiert, in welcher Reihenfolge die Caches prüfen, ob ein zugegriffener Block im Cache ist oder wann die Caches welche Daten austauschen.

Der Rice Simulator for ILP Multiprocessors

Der „Rice Simulator for ILP Multiprocessors“ (RSIM) ist ein Simulationssystem, das nicht nur Speicherhierarchien, sondern auch prozessorinternes Verhalten, wie „Instruction Level Parallelism“ (ILP) simulieren kann. Dadurch wird eine sehr genaue Simulation möglich. Mit RSIM können parallele Speicherstrukturen simuliert werden, wobei jeder Prozessor zwei eigene Speicherebenen (L1 und L2 Cache) haben kann, ehe auf gemeinsamen Hauptspeicher zugegriffen wird. Für die Simulation benutzt RSIM eine Bibliothek zur ereignisbasierten Simulation, wie sie COVINGTON et al. (1991) beschreiben. Der Simulator ist weder protokoll- noch ausführungsgesteuert, sondern interpretiert ein ausführbares Programm.

Der Linux Memory Simulator

Der Simulator Limes („Linux Memory Simulator“) ist ein unter Linux laufender ausführungsgesteuerter Simulator (MAGDIC, 1997a; 1997b). Die Instrumentierung des Programms wird durch die Modifikation des Assemblercodes, den die GCC erzeugt, erreicht. Dadurch ist der Simulator auf die Nutzung der GCC angewiesen und kann auch nicht unbedingt mit jeder Version der GCC genutzt werden, weil sich die Assemblergenerierung ändern könnte. Mit Limes können auch mehrere parallel arbeitende Prozessoren simuliert werden. Allerdings muß die Applikation bei Änderung des Kohärenzprotokolls neu kompiliert und gelinkt werden. Die Parallelität muß in der Applikation mit PARMACS Macros (HEMPEL, 1991) ausgedrückt werden, die zum Beispiel Threaderzeugung und Synchronisation zulassen.

Auf dem LDA-Modell basierender Simulator

With one clock you can tell the exact time,
but with two clocks you can't be sure.
Still, having two clocks is better than having only one.

– AUTOR UNBEKANNT

Kostenmodelle wie das RAM-Modell oder das LDA-Modell erlauben die Ermittlung von Programmlaufzeiten anhand von Maschinenmodellen. Dieser Ansatz ist, im Gegensatz zur Ermittlung der Programmlaufzeit mit Hilfe einer Stoppuhr, dazu geeignet, Programmlaufzeiten auf bisher nicht existierender Hardware und/oder ohne störende Einflußfaktoren wie Betriebssystemunterbrechungen oder ähnliches zu ermitteln. Vergleiche zwischen verschiedenen Algorithmen oder unterschiedlichen Speicherarchitekturen sind dadurch einfacher und ohne Fehler durch äußere Faktoren möglich.

Bisher wurden in dieser Arbeit Speicherhierarchien und Microbenchmarks zur Ermittlung von Systemparametern sowie die grundlegenden Arten von Simulatoren von Speicherhierarchien vorgestellt. Im Rahmen dieser Arbeit ist ein Simulator für Speicherhierarchien entstanden, der Programmlaufzeiten nach dem LDA-Modell bestimmt. Dieser Simulator wird mit den Ergebnissen der Microbenchmarks zur Bestimmung der Parameter einer Speicherhierarchie konfiguriert und simuliert dann die Speicherzugriffe eines Programms für diese Speicherarchitektur.

6.1 Das RAM-Modell

Das sehr einfache Maschinenmodell der „Random-Access-Machine“ (RAM) (AHO et al., 1974) besteht aus einer Recheneinheit und einem Speicher, der beliebig groß sein kann und auf den wahlfrei zugegriffen werden kann. Für Parallelrechner wurde dieses Modell zu dem der „Parallel Random Access Machine“ (PRAM) erweitert, bei dem mehrere Recheneinheiten auf den Speicher zugreifen können (FORTUNE und WYLLIE, 1978). Diese Modelle werden oft dafür benutzt, den Aufwand von Algorithmen und Programmen abzuschätzen und dann mit Hilfe des O-Kalküls, beschrieben im zweiten Kapitel von CORMAN et al. (1990) und erstmals eingeführt von BACHMANN (1892), in Aufwandsklassen einzuteilen. Für eine solche Aufwandsabschätzung haben alle Basisoperationen gleiche, konstante Kosten, und das Laden von Operanden aus dem Speicher wird entweder vernachlässigt oder auch mit konstanten Kosten berechnet. Über die Entwicklung der so aufsummierten Gesamtkosten mit wachsender Eingabegröße wird dann eine Zuordnung zu einer Aufwandsklasse des O-Kalküls vorgenommen. Wie bereits in Kapitel 2 und Kapitel 4 gezeigt wurde, sind aber nicht alle Speicherzugriffe mit konstanten Kosten verbunden, sondern die Kosten hängen von der Speicherarchitektur ab. Diese Unterschiede werden vom RAM-Modell vernachlässigt, was zu deutlichen Fehlabschätzungen führen kann. Dies wird in Kapitel 7 ab Seite 42 am Beispiel der Matrix-Matrix-Multiplikation gezeigt.

6.2 Das Latency-of-Data-Access Modell (LDA-Modell)

Das Latency-of-Data-Access Modell (LDA-Modell) von SIMON und WIERUM (1998) ist ein neuartiges Kostenmodell, das im Hinblick auf Kostenentwicklungen in realen Rechnersystemen entwickelt wurde. Bei diesem Kostenmodell werden Gleitkommazahloperationen konstante Kosten und Speicheroperationen Kosten zugrunde gelegt, die von den Latenzzeiten der Speicherebenen abhängig sind, auf die zugegriffen wird. Ganzzahloperationen werden vernachlässigt.

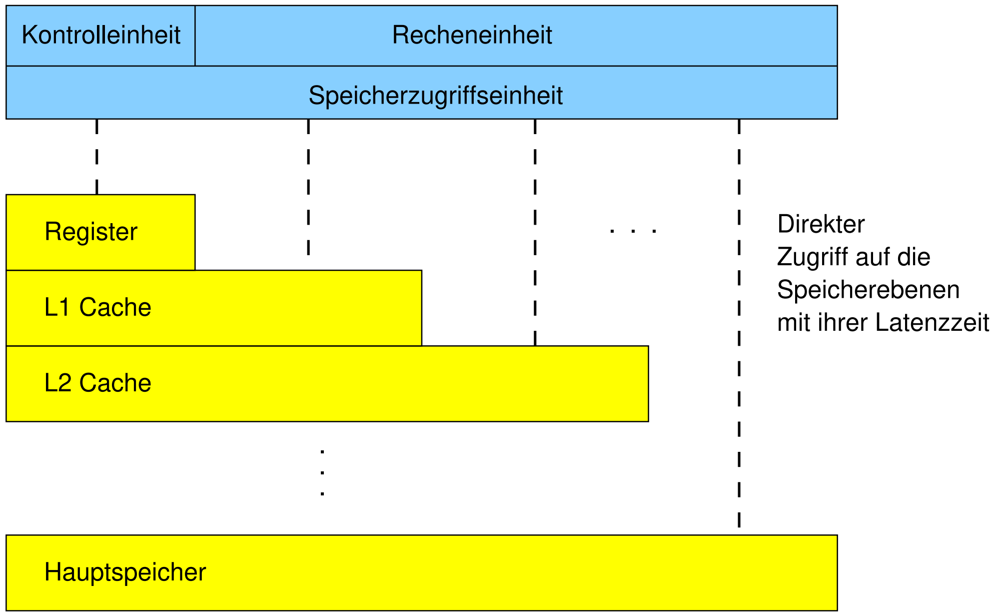


Abbildung 6.1: Das Maschinenmodell, das dem LDA-Modell zugrunde liegt, läßt den Prozessor direkt auf jede Speicherebene zugreifen. Die Gesamtkosten ergeben sich aus der Summe der Latenzzeiten und den ausgeführten Operationen.

Dieses Kostenmodell ist besser an die Gegebenheiten von Speicherhierarchien realer Systeme angepaßt als das RAM-Modell, wodurch das LDA-Modell auch besser zur Abschätzung von Ausführungszeiten auf realen Maschinen dienen kann. Dieser Vorteil wird durch wesentlich mehr benötigte Informationen für eine Abschätzung erkauft. Sowohl Information über die Architektur der Speicherhierarchie, als auch darüber, wieviele Speicherzugriffe auf welche Speicherebene zugreifen, sind notwendig, um eine Kostenberechnung nach dem LDA-Modell durchzuführen.

Gewinnung der Daten Die Daten über die Architektur der Speicherhierarchie können entweder durch Hardwarebeschreibungen gegeben sein oder, wie in Kapitel 4 beschrieben, über Microbenchmarks gemessen werden. Wie häufig auf die Speicherebenen zugegriffen wird, ist durch eine statische Analyse nicht zu beantworten, läßt sich aber durch Ausführung des Programms ermitteln. Einige Speicherbausteine und Cache-Controller besitzen Ereigniszähler (PRVULOVIĆ et al., 1999a), in denen gespeichert wird, wieviele Cache-Hits oder Cache-Misses sie hatten. Aus diesen Ereigniszählern können dann die nötigen Daten mit Tools wie SProf (SIMON, 1999) ausgelesen werden. Ist der Zugriff auf solche Ereigniszähler nicht möglich oder sollen Daten für eine andere Speicherhierarchie als die vorhandene ermittelt werden, kann man die Speicherzugriffe durch einen Simulator für Speicherhierarchien simulieren lassen. Eine mathematische Funktion aufzustellen, aus der sich die Anzahl der Speicherzugriffe berechnen läßt, ist aufwendig und komplex, so daß eine Simulation sinnvoll ist.

Alternative zur Simulation

Mit der Methode der Simulation läßt sich die Ausführungszeit für ein Problem und eine Problemgröße gut bestimmen. Es ist aber unklar wie eine Prognose für andere Eingabedaten oder Problemgrößen aus diesen Daten funktionieren kann, da sich die Speichernutzung bei anderen Problemen (andere Eingabedaten) oder Problemgrößen drastisch ändern kann. Das LDA-Modell wurde eher im Hinblick auf Benchmarking und Programmanalyse hin entworfen, wofür die Analyse von konkreten Programmläufen oft ausreicht.

6.3 Das LDA-Modell für SMP-Systeme

Das LDA-Modell läßt sich einfach auf parallele Rechnerarchitekturen und SMP-Systeme erweitern (SIMON und WIERUM, 1998). Dabei werden die Gesamtkosten durch Gleitkommazahloperationen und Latenzzeiten von Speicheroperationen bestimmt. Jeder Prozessor kann dabei eigene Zwischenspeicher haben und nutzt schließlich gemeinsamen Speicher, der von allen Prozessoren genutzt wird. Ob und wie sich Latenzzeiten ändern, wenn Prozessoren gleichzeitig (zum Beispiel schreibend) auf die gleiche Speicheradresse zugreifen, ist unklar und kann im LDA-Modell nur über ein stochastisches Modell berücksichtigt werden. In SMP-Systemen werden solche Zugriffe sequentialisiert und keine Reihenfolge der Sequentialisierung garantiert. Gleichzeitige Lesezugriffe auf den gemeinsamen Speicher müssen ebenfalls sequentialisiert werden und können zu verlängerten Latenzzeiten führen. Ein solches Verhalten kann im LDA-Modell nur über angepaßte durchschnittliche Latenzzeiten für den gemeinsamen Speicher modelliert werden.

6.4 LDA-Modell basierter Simulator für Speicherhierarchien

Um Programmlaufzeiten durch Simulation von Speicherhierarchien zu ermitteln, eignet sich das LDA-Modell sehr gut. Eine Simulation, die nach dem LDA-Modell Ausführungszeiten ermittelt, kann auf eine detaillierte Simulation von Hardwarekomponenten verzichten und muß „nur“ das logische Verhalten der Zwischenspeicher simulieren. Im Rahmen dieser Arbeit wurde ein flexibler Simulator für Speicherhierarchien entwickelt, dem das LDA-Modell zugrunde liegt.

Um eine Berechnung der Ausführungszeit eines parallelen Programms nach dem LDA-Modell durchzuführen, werden Informationen darüber benötigt, auf welche Speicherebene wie häufig zugegriffen wird. Diese Anzahlen hängen unter anderem vom Kohärenzprotokoll des SMP-Systems ab, da die Inhalte der Zwischenspeicher aller Prozessoren von einem Prozessor beeinflußt werden können. Diese Beeinflussung hat in einigen Fällen starken Einfluß auf die Nutzung der Zwischenspeicher.

Der im Rahmen dieser Arbeit entstandene Simulator für Speicherhierarchien erlaubt auch die Definition von Speicherhierarchien von SMP-Systemen, wobei zwischen den beiden invalidierenden Kohärenzprotokollen MSI und MESI (siehe Kapitel 3 auf Seite 9) gewählt werden kann.

6.5 Konfigurationsmöglichkeiten

Der Simulator für Speicherhierarchien wird über eine Konfigurationsdatei konfiguriert, in der verschiedene Speicherhierarchien definiert werden können. Jede Speicherhierarchie bekommt einen symbolischen Namen. Alle Zeitangaben werden in Takten des Prozessors angegeben.

```
# Konfigurationsdatei
[Alpha.21164]
mhz = 600
processors = 1
firstLevelCaches = Alpha.21164.L0   # durch Kommata getrennte Liste der
                                    # L1 Caches der Prozessoren

[Alpha.21164.L0]
numberOfBlocks = 256                # in x * blocksize
blocksize = 32                      # in bytes
associativity = 2                   # als Zahl oder 'FULL'
replaceStrategy = lru               # lru, random oder round_robin
writeStrategy = write_through       # write_back oder write_through
readLatency = 3                     # in cpu cycles
next = Alpha.21164.L1

[Alpha.21164.L1]
numberOfBlocks = 4096               # in x * blocksize
blocksize = 128                     # in bytes
associativity = FULL                # als Zahl oder 'FULL'
readLatency = 30                    # in cpu cycles
writeLatency = 60                   # in cpu cycles
next = NULL
```

Abbildung 6.2: Der Simulator wird über eine Konfigurationsdatei konfiguriert, in der die zu simulierende Speicherhierarchie beschrieben wird. Hier ist die Konfigurationsdatei für einen Alpha 21164 Prozessor mit 600 MHz ohne Register und ohne Hauptspeicher abgebildet.

Die Definition der Speicherhierarchie eines Alpha 21164 ohne Register und ohne Hauptspeicher zeigt Abbildung 6.2. Die 64-Bit Register würde man als vollassoziative Speicherebene vor dem L0 Cache mit einer Blockgröße von 8 Bytes ohne Latenzzeit realisieren. Für eine Mehrprozessorkonfiguration muß man die Anzahl der Prozessoren und jede Speicherhierarchie der Prozessoren einzeln definieren, bei denen die vorletzte Speicherebene auf eine allen Prozessoren gemeinsame Speicherebene zeigt.

Bei der Erstellung der Konfigurationsdatei helfen die in Kapitel 4 vorgestellten Microbenchmarks. Der Prozessortakt, die Anzahl der Speicherebenen und die Größe und Latenzzeit der Zwischenspeicher lassen sich mit Hilfe dieser Microbenchmarks ermitteln. Ist eine Konfigurationsdatei syntaktisch fehlerhaft, wird vom Simulator eine Fehlermeldung ausgegeben. Ist zum Beispiel eine nicht unterstützte Ersetzungsstrategie für eine Speicherebene angeben, wird eine Liste der unterstützten Ersetzungsstrategien „lru, random, round_robin" in der Fehlermeldung ausgegeben.

6.6 Schnittstelle und Implementation des Simulators

Das UML Klassendiagramm (OMG, 1999) in Abbildung 6.5 zeigt den internen Aufbau des in C++ geschriebenen Simulators. Die Klassen lassen sich dabei in zwei Gruppen einteilen. Die Klassen der einen Gruppe dienen der statischen Beschreibung einer Speicherhierarchie (MachineDef, Protocol, CacheDef, WStrategy und RStrategy), und die Klassen der anderen Gruppe (Caches, Cachelevel und Cacheline) simulieren das dynamische Verhalten der Speicherhierarchie abhängig von der statischen Beschreibung und den Methodenaufrufen an der Schnittstelle. Die zwei Klassen MachineDef und Caches bilden die Schnittstelle für die Instrumentierung, deren Interface in den Abbildungen 6.3 und 6.4 dargestellt ist.

In der Klasse Cachelevel wird die Semantik der Ersetzungs- und Schreibstrategien sowie die Semantik der Kohärenzprotokolle implementiert. Ladeoperationen über Grenzen von Cache-Zeilen werden aufgeteilt und nacheinander verarbeitet. Durch interne Zwischenspeicher ist eine schnelle, perfekte LRU-Ersetzungsstrategie implementiert worden.

```
class MachineDef
{
public:
  MachineDef(const string & machine, const config & config);
  ~MachineDef();
  CacheDef *         getFirstLevel(const uint16 processor) const;
  const string       name()                                const;
  const uint32       mhz()                                 const;
  const uint16       processors()                          const;
  const protocol_t   protocol()                            const;
private:
  ...
};
```

Abbildung 6.3: Die Klasse MachineDef repräsentiert die Definition einer Speicherhierarchie. Ein Objekt dieser Klasse wird benutzt um ein Objekt der Klasse Caches (siehe Abbildung 6.4) zu erzeugen.

Eine weitere in Abbildung 6.5 nicht dargestellte Klasse Config dient zum Einlesen einer Konfigurationsdatei und stellt die Informationen einer Konfigurationsdatei der Klasse MachineDef zur Verfügung, so daß das Einlesen der Konfigurationsdatei vom restlichen Simulator getrennt ist. Eine Umstellung auf ein anderes Dateiformat für die Konfigurationsdatei, wie zum Beispiel XML, ist dadurch einfach möglich.

```
class Caches
{
public:
  Caches(const MachineDef * machine);
  ~Caches();
  void read         (const uint16 processor,
                     const uint64 address,
                     const uint64 size /* in bytes */,
                     const account withAccount = defaultAccount);
  void write        (const uint16 processor,
                     const uint64 address,
                     const uint64 size /* in bytes */,
                     const account withAccount = defaultAccount);
  bool busAction    (const Cachelevel * invokingLevel,
                     const cacheaction_t action,
                     const uint64 address,
                     const uint64 size /* in bytes */)
  uint64 totalCostInCycles     (const account withAccount = sumAll) const;
  uint64 costForReadOfLevel    (const string & inLevelName,
                                const account withAccount = sumAll) const;
  uint64 costForWriteOfLevel   (const string & inLevelName,
                                const account withAccount = sumAll) const;
  void   doFloatOps(uint64 number);
  uint64 totalMemOps() const;
  uint64 totalFloatOps() const;
  // initialize the caches and reset all counters
  void init();
  // reset the counters, but keep the content of the caches untouched
  void resetCounters();
  // get the machinedef of this object
  const MachineDef * getMachineDef() const;
  // Write a checkpoint and give the user the possibility to
  // write additional data to the checkpoint to restart work
  // at the correct place
  ofstream * writeCheckpoint(const string & filename);
  // close the checkpoint file and move it to the final location
  void commitWriteCheckpoint(const string & filename);
  // Read a checkpoint and give the user the ifstream at the position
  // that contains his data
  ifstream * readCheckpoint(const string & filename);
  // close the checkpointfile from reading
  void finishReadCheckpoint(const string & filename);
private:
  ...
};
```

Abbildung 6.4: Die Klasse Caches bietet die Hauptschnittstelle zum Simulator. Die Klasse besitzt Methoden, um Speicher- und Rechenoperationen zu simulieren, die bisherigen Kosten abzufragen und Checkpoints zu schreiben und zu lesen.

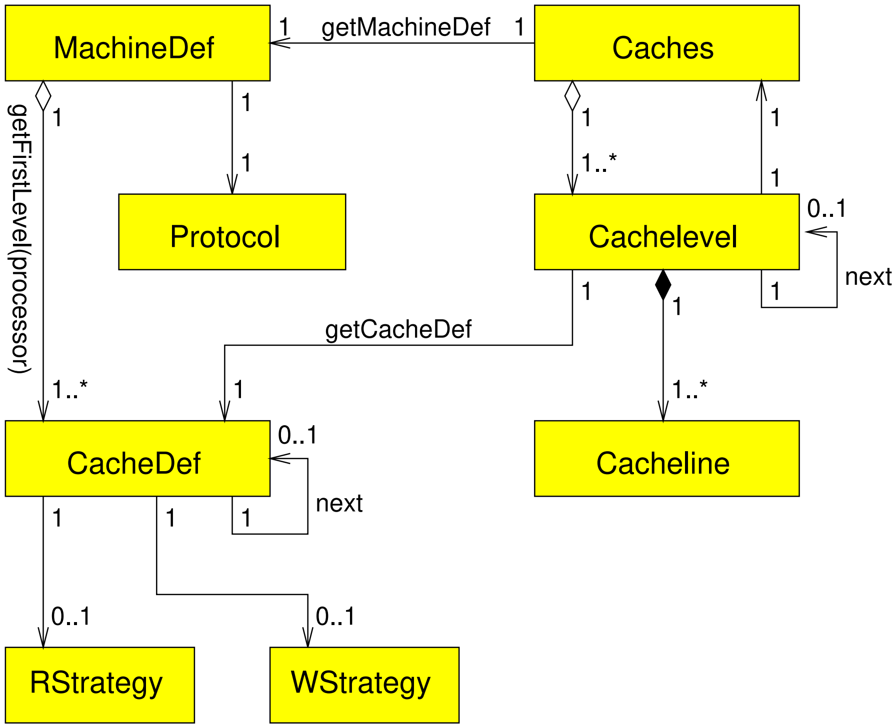


Abbildung 6.5: Das UML Klassendiagramm (OMG, 1999) des Simulators zeigt den groben internen Aufbau des Simulators und die Assoziationen zwischen den Klassen. Dabei sind im linken Teil der Abbildung die Klassen, die statisch eine Speicherhierarchie beschreiben, dargestellt. Die Klassen im rechten Teil der Abbildung realisieren durch ihre Methoden das dynamische Verhalten der Speicherhierarchie. Die Schnittstelle für Benutzer des Simulators bilden die zwei oben abgebildeten Klassen MachineDef und Caches.

6.7 Checkpointing

Die Ausführungszeit einer Simulation einer Speicherhierarchie ist deutlich höher als die Ausführungszeit auf einer entsprechenden Hardware. Um die Flexibilität zur Verfügung zu stellen, eine Simulation zu unterbrechen und später fortzusetzen, wurde eine sogenannte „Checkpointing-Funktion“ implementiert, mit der der Simulator seinen internen Zustand abspeichern und später (auch auf einem anderen Rechner oder mit einem anderen Programm) wieder herstellen kann (INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, 1995).

Diese Funktionalität kann auch dazu genutzt werden, die Analyse verschiedener Varianten eines Programms schneller zu simulieren, wenn es dabei für alle Varianten einen gemeinsamen Anfangsteil gibt. Dann kann man zuerst diesen Anfangsteil simulieren und den Zustand des Simulators abspeichern, um dann die verschiedenen Varianten des Programms auf diesem Zustand aufsetzend weiter zu simulieren. Dadurch wird eine Simulation des Anfangsteils für jede Variante einzeln gespart.

Läßt man in regelmäßigen Abständen einen Checkpoint schreiben, ist es unerheblich, wenn der Computer oder der Strom ausfällt, weil man auf dem letzen Checkpoint wieder aufsetzen kann. Die Applikation, die den Simulator nutzt, muß natürlich auch ein Checkpointing unterstützen. Dazu wird die Möglichkeit gegeben, applikationsspezifische Daten mit in den Checkpoint des Simulators zu schreiben und diese später auch wieder zu lesen.

Platzbedarf Der Platzbedarf eines solchen Checkpoints hängt von der Speicherhierarchie und von den zwischengespeicherten Adressen ab. Es werden für jede Cache-Zeile Informationen wie letzter Zugriff, Zustand im Kohärenzprotokoll, gespeicherte Adresse und für jede Speicherebene die bisher angefallenen Kosten abgespeichert.

6.8 Getrennte Teilkostenermittlung

Um detaillierten Aufschluß über die Kostenverteilung von Speicherzugriffen in Programmen bekommen zu können, kann bei jeder Speicheroperation ein „Konto“ angegeben werden, zu dessen Lasten die Speicheroperation ausgeführt werden soll. Diese Konten können vom Nutzer selbst definiert werden und zum Beispiel zur getrennten Kostenaufzeichnung für verschiedene Speicherbereiche genutzt werden. Diese Möglichkeit wird bei der ungeblockten Matrix-Matrix-Multiplikation in Abschnitt 7.3 ab Seite 42 eingesetzt. Diese Art der Kostenermittlung entspricht einer Anwendung des LDA-Modells auf einen Teil des Speichers oder des Programms.

Mit der getrennten Kostenermittlung könnten auch die Kosten verschiedener Programmabschnitte ermittelt werden oder bei einem Iterationsverfahren die Kosten für verschiedene Iterationstiefen einzeln errechnet werden. Es sind vielfältige, sinnvolle Nutzungen der getrennten Teilkostenermittlung denkbar, die dazu dienen, genauer zu analysieren, welcher Teil der Kosten wo in einem Programm verursacht wird. Durch solche Erkenntnisse kann ein Programm dann gezielt optimiert werden.

6.9 Instrumentierung

Zur Instrumentierung von Programmen müssen zusätzliche Simulatoraufrufe in den Quelltext eingefügt werden, wenn auf Daten zugegriffen wird. Diese Zugriffe können lesend oder schreibend sein. Programme arbeiten oft mit virtuellen Adressen, die sehr hoch sein können, und die Adressen des Simulators liegen zwischen Null und der höchsten Adresse der größten Speicherebene. Der Simulator unterstützt keine automatische Umsetzung von virtuellen auf simulierte Adressen. Bei der Instrumentierung können deshalb meist nicht direkt die Adressen der Speicherzellen im Programm benutzt werden, sondern sie müssen auf niedrigere Adressen für den Simulator umgerechnet werden. Dies kann man meist durch Subtraktion der niedrigsten auftretenden Adresse bei jedem Speicherzugriff tun, so daß alle Adressen verschoben werden, ihre relativen Abstände zueinander aber erhalten bleiben. Diese Verschiebung kann die relative Lage der Cache-Zeilen der Zwischenspeicher zu den Adressen beeinflussen. Will man dies verhindern, darf man die Adressen nur um ein Vielfaches des kleinsten gemeinsamen Vielfachen aller Blockgrößen der Speicherhierarchie verschieben. Die meisten Betriebssysteme verschieben Adressen beim Laden eines Programms in Schritten von 4096 Bytes. Ein Beispiel für einen instrumentierten Quelltext, der diese Größe berücksichtigt, zeigt Abbildung 6.6.

```
int main()
{
    int          i, j, k;
    uint64       base;
    float        reg;
    float        a[1000][1000], b[1000][1000], c[1000][1000];
    Config       config("machines.def");
    MachineDef   machine("alpha",config);
    Caches       caches(&machine);

    base = (uint64) &a[0][0];
    base -= base % 4096;
    for (i = 0; i < 1000; i++)
      for (j = 0; j < 1000; j++)
      {
        reg = 0.0;
        for (k = 0; k < 1000; k++)
        {
          reg += a[i][k] * b[k][j];
          caches.read(0, (uint64) &a[i][k] - base, 4);
          caches.read(0, (uint64) &b[k][j] - base, 4);
          caches.doFloatOps(2);
        }
        caches.write(0, (uint64) &c[i][j] - base, 4);
      }
    cout << "Total cost is : " << caches.totalCostInCycles() << endl;
    return 0;
}
```

Abbildung 6.6: Die Instrumentierung des Quelltextes sorgt für die Aufrufe des Simulators. Durch eine Verschiebung der virtuellen Adressen bekommt man niedrigere Adressen für den Simulator, dessen simulierte Speicheradressen von Null bis zur höchsten Adresse der größten Speicherebene reichen. Die virtuellen Adressen des Programms könnten wesentlich größer sein.

Eliminierung der unnötigen Berechnungen und Daten

Bei den meisten Simulatoren muß der Simulationscomputer mindestens genauso viel Speicher haben, wie das zu simulierende Programm und Problem in Anspruch nehmen. Für eine Abschätzung der Rechenzeit mit einer Simulation sind aber die Speicherinhalte oft irrelevant. Wichtig für die Simulation nach dem LDA-Modell ist lediglich, auf welche Adressen schreibend beziehungsweise lesend zugegriffen wird, um ermitteln zu können, auf welche Speicherebene zugegriffen wird.

Durch Eliminierung der Dateninhalte können so auf einem Rechner Ausführungszeiten von Problemen ermittelt werden, für die der Arbeitsspeicher sonst nicht ausreichen würde. Algorithmen, deren Kontrollflüsse von berechneten Daten abhängen, können auf diese Art aber nicht simuliert werden. Für solche Algorithmen muß man auf die Speicherplatzeinsparungen zumindest teilweise verzichten.

Ermittlung von Programmlaufzeiten mit dem Simulator

Die Zukunft vorauszusagen, ist ganz einfach.
Bei der Vorhersage richtig zu liegen, ist der schwierige Teil.

– HOWARD FRANK

In den folgenden Abschnitten soll durch Vergleich von Meß- und Simulationsergebnissen gezeigt werden, daß durch den Simulator und das LDA-Modell eine gute Abschätzung der Ausführungszeit von Programmen möglich ist. Hierzu werden einige Microbenchmarks simuliert, deren gemessene Ergebnisse bereits in Kapitel 4 präsentiert wurden und zusätzlich eine sequentielle und zwei parallele Matrix-Matrix-Multiplikationen untersucht.

7.1 Simulation der Messung von Latenzzeiten

In Kapitel 4 wurde eine Methode vorgestellt, wie man Latenzzeiten der unterschiedlichen Speicherebenen messen kann. Dabei werden unterschiedlich große Speicherbereiche mit unterschiedlichen Schrittweiten durchlaufen. Läßt man diesen Benchmark durch den Simulator ablaufen, um eine Analyse der Ausführungszeiten nach dem LDA-Modell durchzuführen, sollten die Ergebnisse ein ähnliches Bild zeigen wie die Messung mit einem realen System, deren Resultate auf Seite 21 in Abbildung 4.3 dargestellt sind.

Abbildung 7.1 zeigt die nach dem LDA-Modell berechneten Ausführungszeiten. Die Ergebnisse stimmen im wesentlichen mit denen der realen Messung auf Seite 21 in Abbildung 4.3 überein. Unterschiede im Bereich des L2 Caches werden dadurch hervorgerufen, daß es beim Pentium II zwar für Programmtext und Daten getrennte L1 Caches gibt, der L2 Cache aber sowohl für das Zwischenspeichern vom Programm als auch von Daten genutzt wird. Für die Simulation wurde eine optimale LRU-Ersetzungsstrategie angenommen, der Pentium II benutzt aber nur 2 Bits für die Ersetzungsstrategie. Dadurch treten im realen System, schon bevor der L2 Cache voll ausgenutzt wurde, Ladeoperationen auf, die auf den Hauptspeicher zugreifen. Für die Simulation wurden Latenzzeiten von 190 ns für den Hauptspeicher angenommen. Die durchschnittliche Latenzzeit für den Hauptspeicher steigt bei der Messung in Abbildung 4.3 langsamer an, als in der Simulation, weil die Speicherchips von internen Schaltvorgängen abhängige Latenzzeiten haben. Je nach Zugriffsmuster ist die „Row Adress Strobe to Column Address Strobe“-Latenzzeit im Gegensatz zur „Column Address Strobe Latenzzeit“ nicht immer nötig (WINDECK, 2000). Während der Messung konnten also Hauptspeicherzugriffe bis zu Blockgrößen von circa einem Megabyte mit der CAS-Latenzzeit beantwortet werden, was die Simulation nicht widerspiegelt. Die architekturellen Daten der simulierten Speicherhierarchie können Tabelle 4.1 auf Seite 18 entnommen werden.

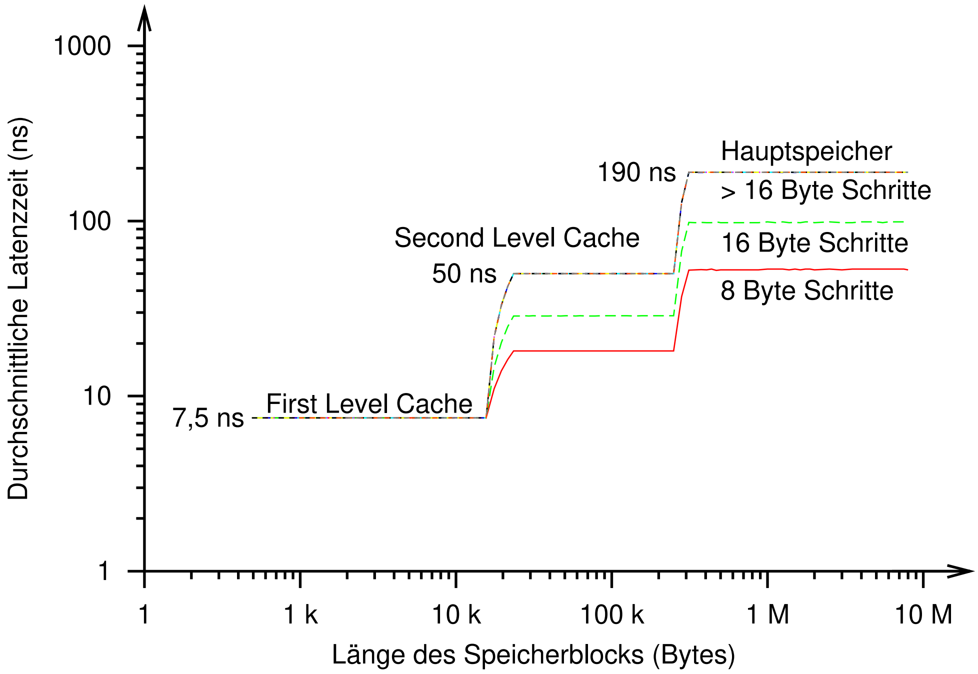


Abbildung 7.1: Die nach dem LDA-Modell berechneten Ausführungszeiten (Speicherhierarchie eines Pentium II Prozessors mit 400 MHz) zeigt für Speicherlatenzzeiten im wesentlichen gleiche Ergebnisse wie die reale Messung in Abbildung 4.3.

7.2 Simulation des Horner-Schema-Benchmarks

Auch der im Kapitel 4 im Abschnitt 4.6 ab Seite 21 vorgestellte Horner-Schema-Benchmark wurde genutzt, um den im Rahmen dieser Arbeit geschriebenen Simulator für Speicherhierarchien und das LDA-Modell zu testen. Der Horner-Schema-Benchmark wurde für den Simulator instrumentiert, um dann zu sehen, ob die in der Praxis auftretenden Effekte auch von der Simulation adäquat nachgebildet werden.

Die Simulation des Horner-Schema-Benchmarks mit der Speicherhierarchie eines Pentium II mit 400 MHz zeigt bei Graden ab 6 ein mit der realen Messung vergleichbares Bild wie Abbildung 7.2 zeigt. Insgesamt zeigt die Simulation etwas bessere Leistungswerte und die von der Speicherhierarchie verursachten Leistungsänderungen sind deutlicher zu erkennen, als bei der realen Messung in Abbildung 4.7 auf Seite 25, finden aber an den gleichen Stellen statt.

Bei niedrigen Graden zeigt die Simulation deutlich bessere Leistungswerte als die reale Messung, was zeigt, daß die Compiler für diesen Bereich keinen optimalen Code erzeugt haben. Die Simulation kommt bei einem Polynomgrad von 4 auf maximal 355 MFlop/s, was viel näher an der theoretisch maximalen Leistung von 400 MHz liegt als die Messungen in Kapitel 4. Die Leistungswerte bei nur einer Auswertungsstelle liegen bei allen Graden höher als die restlichen, da diese Auswertungsstelle dann im Register gehalten werden kann (jede Messung oder Simulation wird mehrmals hintereinander durchgeführt, um zuverlässigere Resultate zu erhalten).

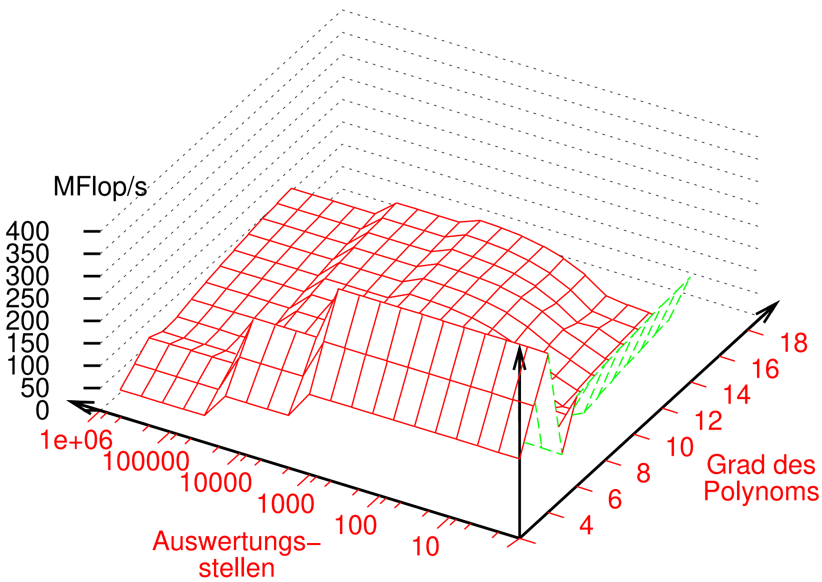


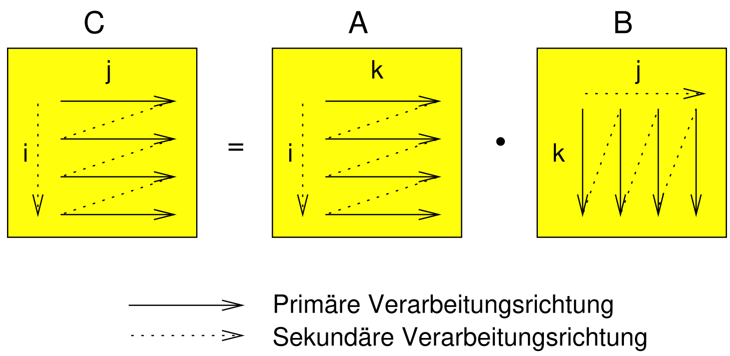
Abbildung 7.2: Die Simulation des Horner-Schema-Benchmarks auf einem Pentium II Prozessors mit 400 MHz zeigt die Unterschiede einer realen Messung (Abbildung 4.7) und den Resultaten, die eine Analyse nach dem LDA-Modell für diese Abbildung liefert.

7.3 Sequentielle Matrix-Matrix-Multiplikation

Es gibt verschiedene Verfahren, eine sequentielle Matrix-Matrix-Multiplikation effizient unter Nutzung einer Speicherhierarchie durch geeignete Aufteilung des Gesamtproblems (Blockung) zu berechnen. Im folgenden soll aber die einfache, ungeblockte Matrix-Matrix-Multiplikation als Beispiel dienen, um daran die Einflüsse von Speicherhierarchien auf die Ausführungszeit zu beobachten. Es soll geprüft werden, ob diese Einflüsse durch das LDA-Modell adäquat nachgebildet werden. Im folgenden Text sei deshalb mit Matrix-Matrix-Multiplikation die ungeblockte Matrix-Matrix-Multiplikation gemeint.

Die erwartete Entwicklung der Ausführungszeit, die man für eine Matrix-Matrix-Multiplikation bei wachsender Matrixgröße benötigt, ist $O(n^3)$. Um die benötigte Zeit für eine Berechnung auf einer Maschine vorherzusagen, mißt man eine Matrix-Matrix-Multiplikation fester Größe und schätzt dann die zeitliche Entwicklung bei wachsender Matrixgröße ab.

Durch Messungen und Simulation der Speicherzugriffe (Abbildung 7.4) läßt sich erkennen, daß die benötigte Zeit schneller wächst als erwartet, wenn man die Zeit einer kleinen Matrix-Matrix-Multiplikation als Basis für eine Prognose der Ausführungszeit einer größeren benutzt. Die gemessenen Ausführungszeiten müßten alle auf einer horizontalen Geraden liegen, wenn die Ausführungszeiten kubisch mit steigender Matrixgröße wüchsen. Die Ausführungszeiten steigen zusätzlich zur kubischen Entwicklung an, weil bei steigendem Grad der Matrizen größere und langsamere Speicherebenen genutzt werden müssen. Dies kann durch Kapazitätsgrenzen der Zwischenspeicher oder durch schlechte Nutzung




```
for (i = 0; i < matrixSize; i++)
  for (j = 0; j < matrixSize; j++)
    for (k = 0; k < matrixSize; k++)
      c[i][j] += a[i][k] * b[k][j];
```

Abbildung 7.3: Bei der Matrix-Matrix-Multiplikation erfolgen die Zugriffe auf die Matrizen A und C zeilenweise und die Zugriffe auf die Matrix B spaltenweise. Bei zeilenweisem Zugriff werden nacheinander im Speicher benachbarte Adressen benötigt, während bei spaltenweisem Zugriff die benötigten Elemente nicht nebeneinander im Speicher abgelegt sind.

der Zwischenspeicher verursacht sein.

Eine genauere Analyse der Kostenverteilung auf die einzelnen Matrizen ist möglich, indem die Zugriffe auf die Matrizen A, B und C auf verschiedene Konten aufgezeichnet werden. Abbildung 7.5 zeigt für eine IBM RS/6000 die Gesamtausführungszeit und die nach Matrizen aufgeteilten Anteile für verschiedene Matrixgrößen. Dabei ist ein Wechsel zwischen Speicherebenen deutlich an einem höheren Zeitbedarf für eine Operation erkennbar. Die Resultatmatrix C verursacht, solange sie in eine Speicherebene paßt, mit wachsender Matrixgröße immer weniger Kosten, weil im Verhältnis zur Anzahl der Gesamtoperationen immer seltener auf sie zugegriffen wird.

Bei den Matrizen A und B ist erkennbar, daß für Matrix B eher auf größere und langsamere Speicherebenen zugegriffen wird, Matrix A aber noch in kleineren, schnelleren Speicherebenen zwischengespeichert wird, so daß die Zugriffe auf Matrix B bald die Gesamtausführungszeit des Programms dominieren. Dies liegt daran, daß Zugriffe auf Matrix B die Zwischenspeicher schlecht nutzen. Ein Zugriff auf Matrix B speichert eine ganze Cache-Zeile in den Zwischenspeichern, von der nur wenige Bytes benötigt werden, da die im Speicher benachbarten Adressen nicht direkt benötigt werden. Die Cache-Zeile wird durch andere Cache-Zeilen aus dem Zwischenspeicher verdrängt, ehe weitere Bytes der Cache-Zeile genutzt werden könnten. Bei Zugriffen auf Matrix A dagegen kann direkt von dem Laden ganzer Cache-Zeilen profitiert werden, weil einmaliges Laden einer Cache-Zeile in einen schnelleren Zwischenspeicher mehrere darauffolgende Speicherzugriffe beschleunigt.

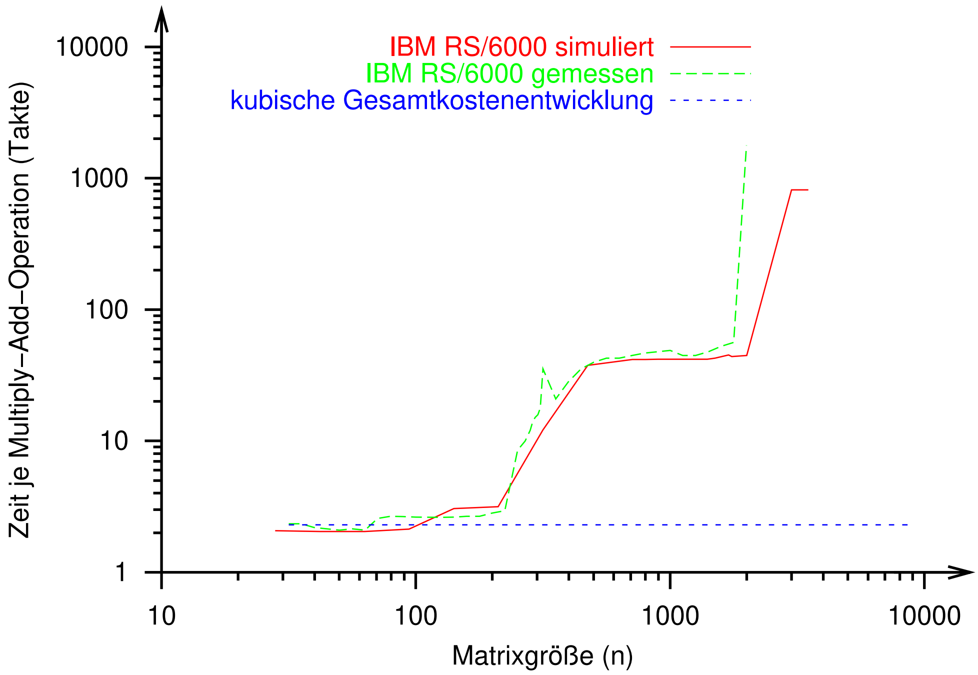


Abbildung 7.4: Die Kosten einer Matrix-Matrix-Multiplikation (n×n-Matrizen) mit der Speicherhierarchie einer IBM RS/6000 steigen durch die Nutzung verschiedener Speicherebenen mehr als kubisch mit dem Grad der Matrizen an. Die gemessenen Daten sind SIMON (1999) entnommen.

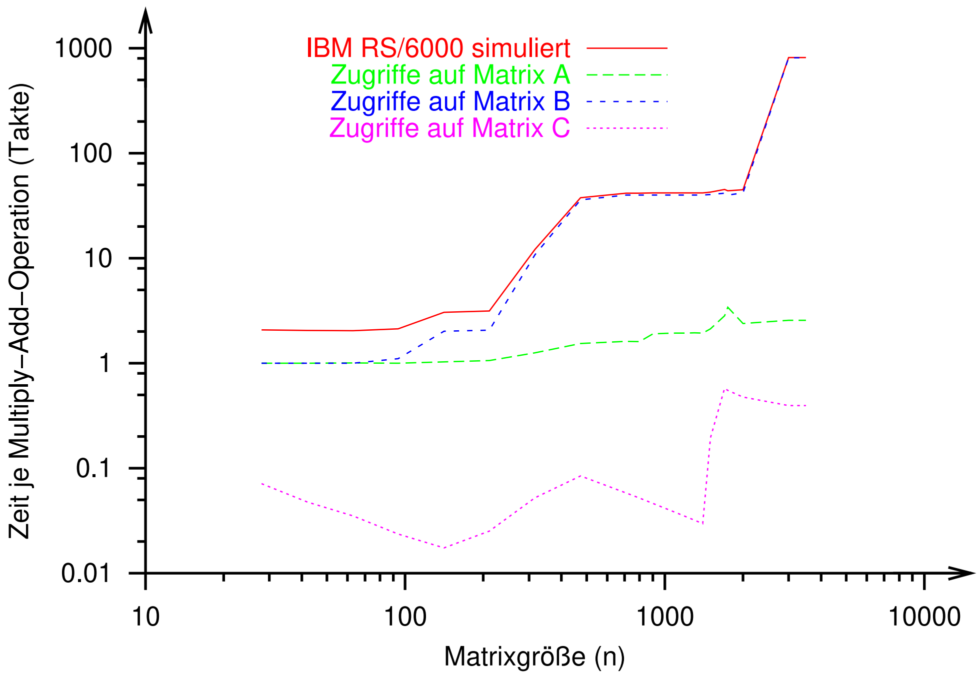


Abbildung 7.5: Die Kosten, die durch die Zugriffe auf drei Matrizen bei einer Matrix-Matrix-Multiplikation $C = A \cdot B$ verursacht werden, lassen erkennen, daß die Zugriffe auf Matrix B die Gesamtkosten bei wachsender Matrixgröße (n×n-Matrizen) dominieren. Simuliert wurde die Speicherhierarchie einer IBM RS/6000.

Optimierte Matrix-Matrix-Multiplikation

Für eine Optimierung der Cachenutzung muß ein spaltenweises Durchlaufen der Matrix B verhindert werden. Man kann Matrix B transponiert im Speicher ablegen, so daß die ursprünglich spaltenweisen Zugriffe zu zeilenweisen Zugriffen werden und dann die Zwischenspeicher wie bei Matrix A genutzt werden.

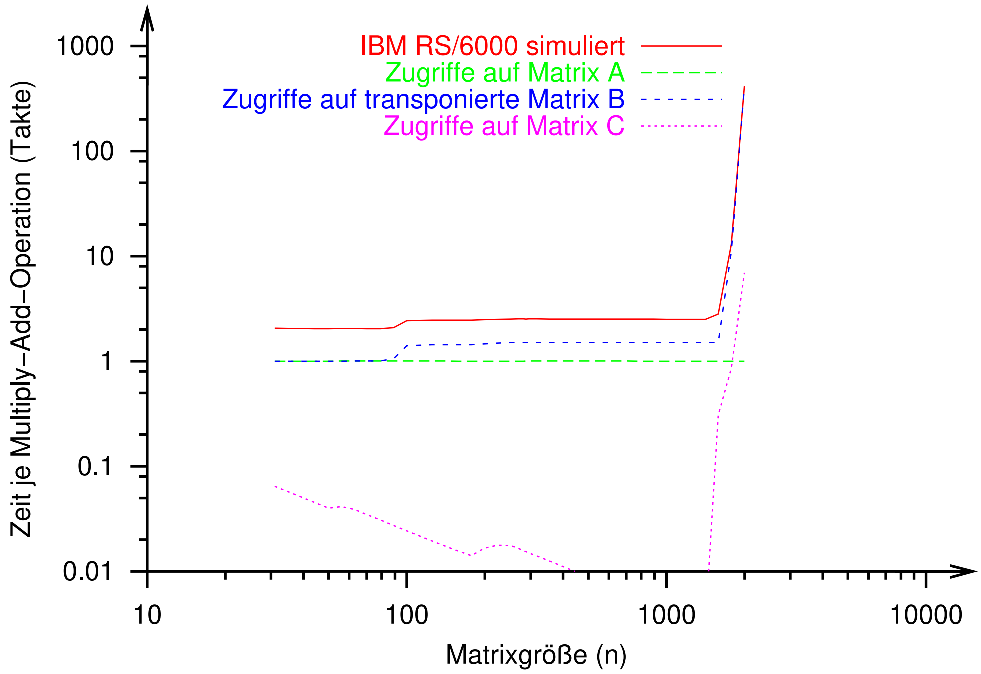


Abbildung 7.6: Matrix B wurde transponiert im Speicher abgelegt, so daß ein spaltenweiser Zugriff benachbarte Adressen anfordert. Durch diese Optimierung der Zugriffe auf die vorher kostendominierende Matrix konnten die Ausführungszeiten der Matrix-Matrix-Multiplikation deutlich reduziert werden (n×n-Matrizen). Simuliert wurde die Speicherhierarchie einer IBM RS/6000.

Wie Abbildung 7.6 zeigt, tritt der erwartete Leistungszuwachs tatsächlich ein. Durch die optimierten Zugriffe auf Matrix B kann die Zeit je Multiply-Add-Operation auch für größere Matrizen niedrig gehalten werden, weil die Zwischenspeicher wesentlich besser genutzt werden und sich die durchschnittlichen Latenzzeiten reduzieren.

Daß schließlich bei einer Matrixgröße von etwa 1700 die Ausführungszeit trotzdem deutlich ansteigt, liegt daran, daß die Kapazität der Zwischenspeicher nicht mehr ausreicht und häufiger auf den Hauptspeicher zugegriffen werden muß. Daß die Zugriffszeiten für Matrix B höher sind als die für Matrix A, liegt daran, daß Matrix B in einer höheren Speicherebene liegt als Matrix A (Index i bleibt gleich, aber Index j ändert sich).

Dieses einfache Beispiel zeigt, daß es für Optimierungen sinnvoll sein kann, Kenntnisse über die Kosten von Speicherzugriffen auf bestimmte Speicherbereiche zu haben. Übliche Profiling-Techniken hätten alle Kosten der Anweisung in der innersten Schleife zugeordnet und nicht detaillierten Aufschluß über die Kosten der Speicherzugriffe auf die verschiedenen Matrizen geben können.

7.4 Messung parallelisierter Matrix-Matrix-Multiplikationen

Für die Untersuchung des Einflusses von Speicherzugriffen auf die Leistung und Ausnutzung einer Speicherhierarchie eines SMP-Systems wurden zwei verschiedene, einfache Parallelisierungen der Matrix-Matrix-Multiplikation für ein Dualprozessorsystem vorgenommen. Bei beiden Parallelisierungen wurde die mittlere Schleife der Matrix-Matrix-Multiplikation (siehe Abbildung 7.3, Seite 43) in zwei Teile zerlegt, die von zwei Threads bearbeitet werden. Bei der ersten Variante werden alle geraden Indizes der mittleren Schleife (siehe Abbildung 7.3, Seite 43) von einem und alle ungerade Indizes vom anderen Thread bearbeitet (alternierend). Bei der zweiten Variante wird die erste Hälfte der Indizes von einem und die andere Hälfte der Indizes vom anderen Thread bearbeitet (blockweise).

Bei der alternierenden Aufteilung wird erreicht, daß beide Prozessoren häufig gleichzeitig auf nahe beeinanderliegende Adressen zugreifen, wodurch Konflikte entstehen und die Zwischenspeicher der einzelnen Prozessoren aufgrund des Kohärenzprotokolls schlecht genutzt werden können. Bei der blockweisen Aufteilung werden diese Konflikte stark reduziert, da die Adressen, auf die die Prozessoren gleichzeitig zugreifen, weiter auseinander liegen. Abbildung 7.7 zeigt die durchschnittliche Dauer für die Ausführung einer Multiply-Add-Operation in Takten bei verschiedenen Matrixgrößen für die zwei Varianten der Matrix-Matrix-Multiplikation auf einem Dualprozessorsystem mit zwei Pentium III Prozessoren mit je 500 MHz (weitere Daten siehe Tabelle 4.1 auf Seite 18).

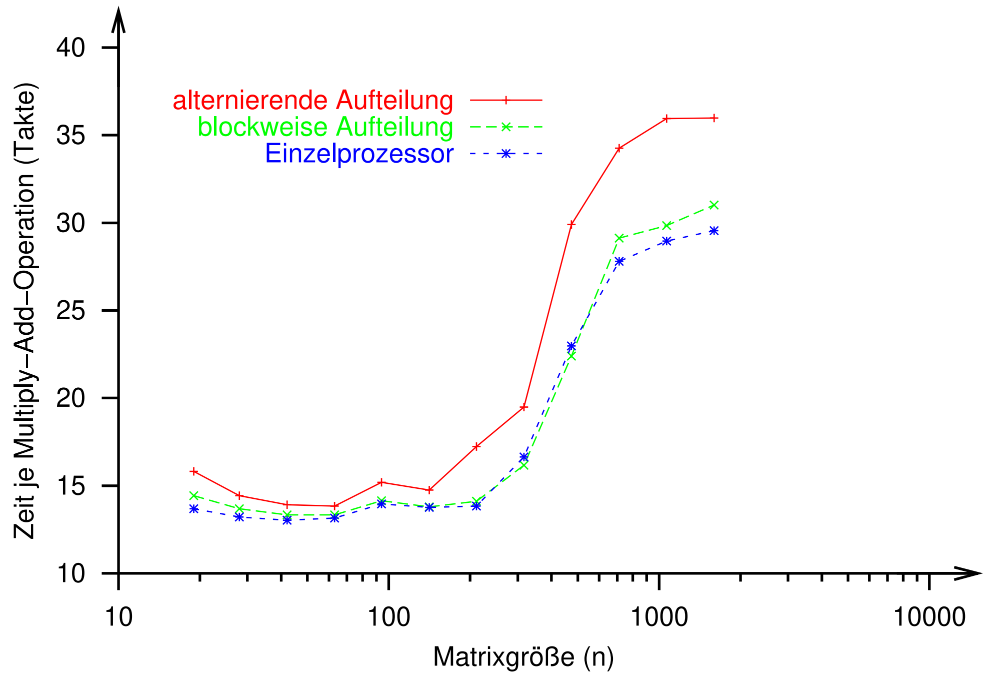


Abbildung 7.7: Die Ausführungszeiten einer Matrix-Matrix-Multiplikation für zwei verschiedenen Aufteilungsweisen der Operationen auf zwei Threads. Einmal wurden die Indizes der mittleren Schleife (siehe Abbildung 7.3, Seite 43) alternierend von den Threads berechnet, bei der anderen Aufteilung wurden die Indizes in zwei Blöcken von den Threads auf einem Dual-Pentium III berechnet. Die höheren Ausführungszeiten kommen durch konkurrierende Speicherzugriffe auf Cache-Zeilen wegen des Kohärenzprotokolls der Zwischenspeicher zustande.

Abbildung 7.7 zeigt, daß die gegenseitigen Verdrängungseffekte bei der alternierenden Aufteilung die Ausführungszeit des Programms deutlich erhöhen. Bei einer blockweisen Aufteilung wird fast ein linearer Speedup erreicht. An zwei Meßpunkten ist die blockweise Aufteilung sogar schneller als die Einzelprozessorvariante, was dadurch verursacht sein kann, daß gemeinsam genutzte Daten (Matrix A) nur von einem Prozessor aus dem Hauptspeicher in seinen Cache geholt werden müssen und dieser dann die Daten für den anderen Prozessor mit geringerer Latenzzeit zur Verfügung stellt (per Flush auf den Bus, siehe Abschnitt 3.4).

Die Ausführungszeit steigt für größere Matrizen, wie bei einer sequentiellen Matrix-Matrix-Multiplikation an, weil die Kapazitäten der schnelleren Zwischenspeicher nicht mehr so gut genutzt werden können und weil Zugriffe auf Matrix B die Ausführungszeit schließlich dominieren.

7.5 Simulation parallelisierter Matrix-Matrix-Multiplikationen

Bei einer Simulation der parallelisierten Matrix-Matrix-Multiplikation können die zwei Threads wesentlich genauer synchronisiert werden als bei einer realen Messung wie in Abschnitt 7.4. Synchronisationspunkte bei einer Messung würden die Inhalte der Zwischenspeicher beeinflussen und zwei gleichzeitig gestartete Threads laufen nicht ganz synchron. Trotzdem zeigt auch die Simulation ein ähnliches Verhältnis zwischen der alternierenden und der blockweisen Aufteilung der Matrix-Matrix-Multiplikation.

```
size = 1000 * 1000 * 4;
for (i = 0; i < 1000; i++)
  for (j = 0; j < 1000; j += 2)
  {
    for (k = 0; k < 1000; k++)
    {
      caches.read(0, a + i * size + k * 4, 4);
      caches.read(1, a + i * size + k * 4, 4);
      caches.read(0, b + k * size + j * 4, 4);
      caches.read(1, b + k * size + (j + 1) * 4, 4);
      caches.doFloatOps(2);
    }
    caches.write(0, c + i * size + j * 4, 4);
    caches.write(1, c + i * size + (j + 1) * 4, 4);
  }
```

Abbildung 7.8: Die Instrumentierung der zentralen Schleife der parallelen Matrix-Matrix-Multiplikation für die alternierende Aufteilung und eine Matrixgröße von 1000.

Abbildung 7.8 zeigt die Instrumentierung der parallelen Matrix-Matrix-Multiplikation für die alternierende Aufteilung. Bei einer Instrumentierung eines parallelen Programms muß immer eine Sequentialisierung der Ladeoperationen für den Bus angegeben werden, damit reproduzierbare Ergebnisse berechnet werden können. Diese Tatsache schränkt weder die Nutzbarkeit des Simulators noch des LDA-Modells ein (konkurrierende, schreibende Zugriffe auf den Hauptspeicher werden normalerweise von der Hardware sequentialisiert).

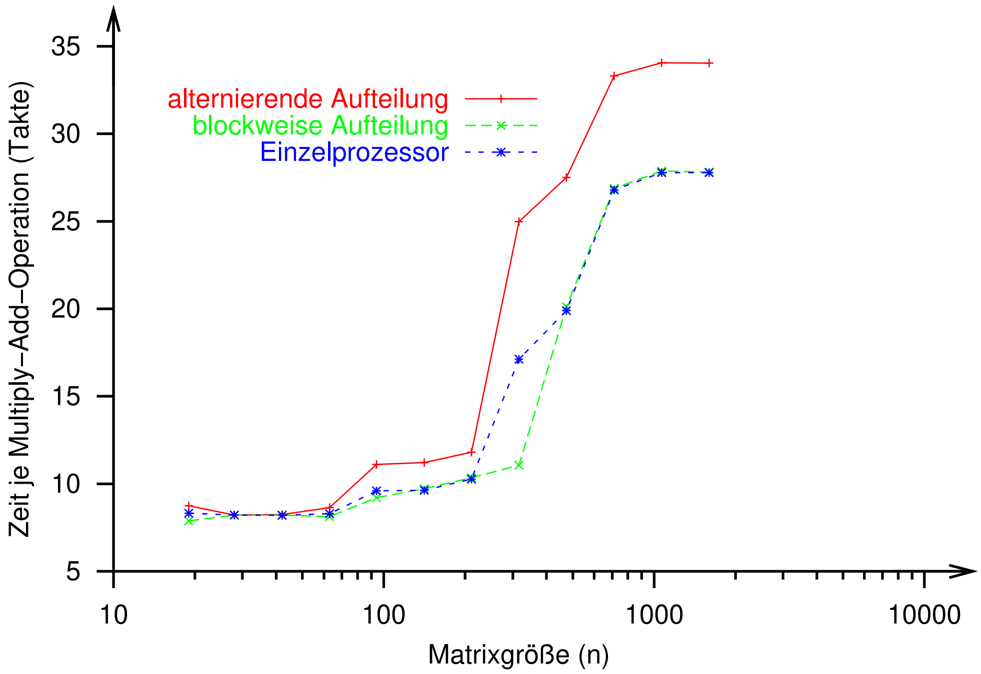


Abbildung 7.9: Die mit dem Simulator errechneten Ausführungszeiten nach dem LDA-Modell einer Matrix-- Matrix-Multiplikation (n×n-Matrizen) für zwei Threads zeigen ähnliche Unterschiede zwischen den zwei verschiedenen Aufteilungen, wie die tatsächliche Messung in Abbildung 7.7.

Abbildung 7.9 zeigt die Ergebnisse der Simulation der parallelisierten Matrix-Matrix-Multiplikation. Die Ausführungszeiten sind im Gegensatz zu der Messung (Abbildung 7.7) generell niedriger, was einerseits dadurch zu erklären ist, daß für die acht Gleitkommaregister bei der Simulation eine optimale LRU-Ersetzungsstrategie angenommen wurde, bei einem realen Pentium III-System die Registernutzung jedoch wesentlich eingeschränkter ist, und andererseits bei der Simulation die Caches nicht durch das Zwischenspeichern von Instruktionen oder durch Verwaltungsaufwände beeinflußt wurden.

Trotzdem werden durch eine Ermittlung der Ausführungszeit nach dem LDA-Modell durch den Simulator die wesentlichen Unterschiede durch Verdrängungseffekte bei einem Dualprozessorsystem deutlich und akkurat widergespiegelt. Daß bei kleinen Matrizen die Ausführungszeiten zwischen den unterschiedlichen Varianten bei der Messung weiter auseinander liegen, als bei der Simulation, liegt an dem Hohen statistischen Einfluß der Threadsynchronisation bei kleinen Matrixgrößen. Das LDA-Modell ist also auch für SMP-Systeme geeignet und kann für Optimierungen und Analysen im Zusammenhang mit einem Simulator für Speicherhierarchien gut verwendet werden.

Zusammenfassung

Iucundi acti labores.

– MARCUS TULLIUS CICERO

8.1 Ergebnisse

In dieser Diplomarbeit wurde das neuartige „Latency-of-Data-Access Model“ (LDA-Modell) von SIMON und WIERUM (1998) genauer untersucht und getestet, ob es Programmausführungszeiten adäquat widerspiegelt. Das Modell berücksichtigt Latenzzeiten von unterschiedlichen Speicherebenen einer Speicherhierarchie. Für die Berechnung der Ausführungszeit nach dem LDA-Modell sind Informationen über die Anzahl der Zugriffe auf die verschiedenen Speicherebenen nötig. Um die Anzahl der Speicherzugriffe auf jede Speicherebene zu bestimmen, wurden in dieser Arbeit die Speicherzugriffe von Programmen und das Verhalten der Zwischenspeicher simuliert und dabei aufgezeichnet, wie häufig auf die unterschiedlichen Speicherebenen zugegriffen wurde. Für SMP-Systeme wurde untersucht, ob das LDA-Modell die gegenseitige Beeinflussung der Zwischenspeicher zweier Prozessoren adäquat widerspiegelt.

Die Untersuchungen haben gezeigt, daß das LDA-Modell die Beeinflussung der Ausführungszeiten durch die Nutzung von Speicherhierarchien adäquat widerspiegelt. Sowohl die Simulation von Microbenchmarks als auch die Simulation einer Matrix-Matrix-Multiplikation haben Veränderungen der Ausführungszeit bei unterschiedlichen Problemgrößen wie erwartet nachgebildet. Sogar Verdrängungseffekte aufgrund eines Kohärenzprotokolls in einem SMP-System konnten durch das LDA-Modell so nachgebildet werden, daß sie realen Messungen entsprachen.

Zusätzlich wurde das LDA-Modell auf eine neue Art und Weise eingesetzt, bei der es sich nicht nur zur Ermittlung von Gesamtausführungszeiten, sondern auch zur Ermittlung von Teilausführungszeiten eignet. Hierfür wurden mehrere Analysen nach dem LDA-Modell gleichzeitig durchgeführt. Bei jedem Speicherzugriff und jeder Operation wird gewählt, zu welchem der parallelen LDA-Modelle die dabei entstehenden Warte- oder Ausführungszeiten hinzugerechnet werden sollen. So können zum Beispiel die Wartezeiten auf Speicheroperationen von verschiedenen Speicherbereichen in einem Programm getrennt ermittelt werden (siehe Abschnitt 7.3 ab Seite 42). Die Summe der Ausführungszeiten aller gleichzeitig simulierten LDA-Modelle entspricht der Gesamtausführungszeit, die eine Simulation mit einem einzigen LDA-Modell ergeben hätte.

Die Hypothese, daß sich mit Hilfe des LDA-Modells durch Simulation des Verhaltens von Speicherhierarchien Programmausführungszeiten adäquat berechnen lassen, hat sich für die untersuchten Programme bestätigt.

8.2 Realisierung der Untersuchung

Für die Untersuchung des LDA-Modells wurde ein flexibler Simulator entwickelt, der das Verhalten verschiedener, beliebig tiefer Speicherhierarchien nachbilden kann und mitprotokolliert, wie häufig auf die unterschiedlichen Speicherebenen zugegriffen wird. Der Simulator unterstützt auch Speicherhierarchien von SMP-Systemen mit zwei verschiedenen Kohärenzprotokollen (MSI und MESI) und kann für weitere Protokolle erweitert werden. Der Simulator wurde in C++ entwickelt und wird über eine Konfigurationsdatei, in der beliebig viele verschiedene Speicherhierarchien definiert werden können, konfiguriert. Die Simulation einer Speicherhierarchie ist unabhängig von der Speicherhierarchie des Computers, auf dem der Simulator ausgeführt wird. Für ein zu simulierendes Programm muß der Quelltext von Hand instrumentiert werden, was eine hohe Portabilität und Anpassungsfähigkeit an spezielle Bedürfnisse im Gegensatz zu anderen Ansätzen gewährleistet.

Um eine Simulation zeitweise zu unterbrechen oder bisher berechnete Teilergebnisse zu sichern, bietet der Simulator die Möglichkeit, die internen Zustände der Speicherebenen einer simulierten Speicherhierarchie zu sichern und später auf diesem Stand wieder aufzusetzen.

Zur Ermittlung von architekturellen Daten einer Speicherhierarchie und der Taktrate des Prozessores, die zur Konfiguration des Simulators notwendig sind, wurden Microbenchmarks vorgestellt und implementiert, deren Resultate in der Arbeit diskutiert wurden. Hierzu zählen ein Benchmark zur Messung von Latenzzeiten, mit dem neben den Latenzzeiten der einzelnen Speicherebenen auch deren Größe, die Länge von Cache-Zeilen und die Anzahl der Cache-Zeilen bestimmt werden können (siehe Abschnitt 4.4 ab Seite 17) und der Horner-Schma-Benchmark zur Messung der Speicheranbindung eines Prozessors im Verhältnis zu seiner Rechenleistung (siehe Abschitt 4.6 ab Seite 21).

Durch die Kombination von Microbenchmarks, Information über Speicherzugriffe und Simulation von Speicherhierarchien mit dem LDA-Modell können Ausführungszeiten von Programmen adäquat bestimmt werden, indem die Microbenchmarks zur Konfiguration des Simulators für Speicherhierarchien benutzt werden, der wiederum die Daten für eine Berechnung nach dem LDA-Modell liefert, beziehungsweise direkt eine Analyse nach dem LDA-Modell durchführt.

8.3 Ausblick

Für weitere Untersuchungen mit dem Simulator für Speicherhierarchien könnte dieser um weitere Kohärenzprotokolle, zum Beipiel um das Dragon-Protokoll, erweitert werden, so daß vergleichende Untersuchungen zwischen aktualisierenden und invalidierenden Kohärenzprotokollen möglich würden.

Zur Optimierung von Programmen oder Kernstücken von Programmen scheint besonders die Möglichkeit, Teilausführungszeiten von Programmen unabhängig vom Kontrollfluß ermitteln zu können, sehr interessant, da dadurch die Nutzung einer Speicherhierarchie gut analysiert werden kann. Untersuchungen von weiteren Applikationen oder Benchmarks könnten hier interessante, neue Ergebnisse liefern.

Literaturverzeichnis

AHO, ALFRED V.; HOPCROFT, JOHN E. und ULLMAN, JEFFREY D.
The Design and Analysis of Computer Algorithms.
Addison-Wesley, 1974.

ANDERSON, ED; BROOKS, JEFF und HEWITT, TOM.
The Benchmarker's Guide to Single-processor Optimization for Cray T3E Systems.
Benchmarking Group, Cray Research, Juni 1997.

BACHMANN, PAUL GUSTAV HEINRICH.
Die Analytische Zahlentheorie.
B.G. Teubner, Leipzig, 1892.

CHAMBERLAIN, STEVE.
libbfd: The Binary File Descriptor Library.
Cygnus Support, April 1991.

CORMAN, THOMAS H.; LEISERSON, CHARLES E. und RIVEST, RONALD L.
Introduction to algorithms.
MIT Press, 1990.
ISBN 0-262-53091-0.

COVINGTON, R. G.; DWARKADAS, SANDHYA; JUMP, J. ROBERT; SINCLAIR, JAMES B. und MADALA, SRIDHAR.
Efficient simulation of parallel computer systems.
International journal in computer simulation, 1,1:31–58, 1991.

CULLER, DAVID E. und SINGH, JASWINDER.
Parallel Computer Architecture A Hardware/Software Approach.
Morgan Kaufmann Publishers, 1999.
ISBN 1-55860-343-3.

DIXIT, KAIVALYA und REILLY, JEFF.
SPEC95 questions and answers.
SPEC Newsletter, 7:7 – 10, September 1995.
URL: `http://www.specbench.org/osg/cpu95/`.

DUNLOP, ALISTAIR und HEY, TONY.
PERFORM - a fast simulator for estimating program execution time.
The Journal of Performance Evaluation and Modelling for Computer Systems, November 1997.
URL: `http://hpc-journals.ecs.soton.ac.uk/PEMCS/`.

FORTUNE, STEVEN und WYLLIE, JAMES.
Parallelism in random access machines.
Proc. of 10th ACM Symposium on Theory of Computing, Seiten 114 – 118, 1978.

HEMPEL, ROLF.
The ANL/GMD Macros (PARMACS) in Fortran for Portable Parallel Programming Using the Message Passing Programming Model – Users' Guide and Reference Manual.
Technischer Bericht, GMD, November 1991.

HENNESSY, J. L. und PATTERSON, D. A.
Computer Architecture: A Quantitative Approach.
Morgan Kaufmann, San Mateo, CA, USA, zweite Auflage, 1996.

HOCKNEY, ROGER W. und BERRY, MICHAEL.
Public international benchmarks for parallel computers.
Technical report, CS-93-213, PARKBENCH Committee, Computer Science Department, University of Tennessee, November 1993.

HORNER, WILLIAM GEORGE.
A new method of solving numerical equations of all orders, by continuous approximation.
In *Philosophical Transactions of the Royal Society of London*, Seiten 308–335, Juli 1819.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION.
Basic Reference Model of Open Distributed Processing, 1995.
URL: `http://www.iso.ch/`.
ISO/IEC International Standard 10746. Parts 1 – 4.

LARUS, JAMES R. und SCHNARR, ERIC.
EEL: Machine-independent executable editing.
In *Proceedings of the SIGPLAN'95 Conference on Programming Language Design and Implementation*, Seiten 291–300, 1995.

LEBECK, ALVIN R. und WOOD, DAVID A.
Active memory: A new abstraction for memory-system simulation.
In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, Seiten 220–230, New York, NY, USA, Mai 1995. ACM Press.
ISBN 0-89791-695-6.
URL: `http://www.cs.duke.edu/~alvy/fast-cache/`.

MAGDIC, DAVOR.
Limes: A multiprocessor environment for PC Platforms.
In *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, März 1997a.

MAGDIC, DAVOR.
Limes: An execution-driven multiprocessor simulation tool for the i486+-based PCs.
User's Guide, University of Belgrade, 1997b.

McVoy, Larry und Staelin, Carl.
lmbench: portable tools for performance analysis.
In *USENIX 1996 Annual Technical Conference, January 22–26, 1996. San Diego, CA, USA*, Seiten 279–294, Berkeley, CA, USA, Januar 1996. USENIX.
URL: `http://bitmover.com/lmbench/`.

Mindshare, Inc. und Shanley, Tom.
Pentium Pro and Pentium II system architecture.
Addison-Wesley, Reading, MA, USA, zweite Auflage, 1999.
ISBN 0-201-30973-4.

Oed, Wilfried.
Massiv-paralleles Prozessorsystem Cray T3E.
Cray Research GmbH, Riesstraße 25, 80992 München, November 1996.

OMG.
Unified Modeling Language Specification.
Open Management Group, Version 1.3 alpha R2 Auflage, Januar 1999.
URL: `http://www.omg.org/`.

Prvulović, M.; Marinov, D.; Dimitrijević, Z. und Milutinović, V.
Split Temporal/Spatial Cache: A Survey and Reevaluation of Performance.
In *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, Juli 1999a.

Prvulović, M.; Marinov, D.; Dimitrijević, Z. und Milutinović, V.
The Split Spatial/Non-Spatial Cache: A Performance and Complexity Evaluation.
In *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, Juli 1999b.

Ranganathan, Parthasarathy; Adve, Sarita V. und Jouppi, Norman P.
Reconfigurable caches and their application to media processing.
In *Proceedings of the 27th International Symposium on Computer Architecture*, Juni 2000.

Simon, Jens.
Werkzeugunterstützte effiziente Nutzung von Hochleistungsrechnern.
Doktorarbeit, Universität-GH Paderborn, 1999.
ISBN 3-931445-03-9.

Simon, Jens und Wierum, Jens-Michael.
The Latency-of-Data-Access Model for Analyzing Parallel Computation.
Information Processing Letters, 66(5):255–261, Juni 1998.

Staelin, Carl; McVoy, Larry und BitMover, Inc.
mhz: anatomy of a micro-benchmark.
In *Proceedings of the USENIX 1998 Annual Technical Conference*, Seiten 155–166, Berkeley, USA, Juni 15–19 1998. USENIX Association.
ISBN 1-880446-94-4.

STRICKER, THOMAS M. und GROSS, THOMAS.
Optimizing memory system performance for communication in parallel computers.
In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Seiten 308–319, Santa Margherita Ligure, Italy, Juni 22–24, 1995. ACM SIGARCH and IEEE Computer Society TCCA.

STRICKER, THOMAS M. und GROSS, THOMAS.
Global address space, non-uniform bandwidth: A memory system performance characterization of parallel systems.
In *Proceedings of the Third International Symposium on High Performance Computer Architecture (HPCA '97)*, Seiten 168–181, Los Alamitos, CA, USA, Februar 1997. IEEE Computer Society Press.
ISBN 0-8186-7764-3.

TAM, EDWARD; RIVERS, JUDE; TYSON, GARY und DAVIDSON, EDWARD S.
mlcache: A flexible multi-lateral cache simulator.
Technical Report CSE-TR-363-98, Computer Science and Engineering, University of Michigan, Mai 1998.

WINDECK, CHRISTOF.
Speicherriegel enträtselt. Wie Mainboards die Hauptspeicher-Zeitparameter einstellen (sollten).
c't Magazin für Computertechnik, (17):166–173, 2000.

Index