

DigiSem
Wir beschaffen und
digitalisieren



b
UNIVERSITÄT
BERN
Universitätsbibliothek Bern

Dieses Dokument steht Ihnen online zur Verfügung
dank DigiSem, einer Dienstleistung der
Universitätsbibliothek Bern.

Kontakt: Gabriela Scherrer
Koordinatorin digitale Semesterapparate
E-Mail digisem@ub.unibe.ch, Telefon 031 631 93 26

David A. Patterson

John L. Hennessy

Rechnerorganisation und -entwurf

Die Hardware/Software-Schnittstelle

3. Auflage herausgegeben von Arndt Bode,
Wolfgang Karl und Theo Ungerer

Aus dem Amerikanischen übersetzt von Elke Jauch
und Judith Muhr

A-4'52'313

Universitätsbibliothek Bern
Zentralbibliothek

2007



Spektrum
AKADEMISCHER VERLAG

IT 150 42

Zuschriften und Kritik an:

Elsevier GmbH, Spektrum Akademischer Verlag, Dr. Andreas Rüdinger, Slevogtstraße 3–5, 69126 Heidelberg

Titel der Originalausgabe: Computer Organization and Design, the hardware/software interface, 3rd edition

First published in the United States by Morgan Kaufmann Publishers, San Francisco, CA

Morgan Kaufmann Publishers is an Imprint of Elsevier. Copyright © 2005 Elsevier Inc. All rights reserved.

Wichtiger Hinweis für den Benutzer

Verlag, Autoren, Übersetzerinnen und Herausgeber haben alle Sorgfalt walten lassen, um vollständige und akkurate Informationen in diesem Buch und der beiliegenden CD-ROM zu publizieren. Der Verlag übernimmt weder Garantie noch die juristische Verantwortung oder irgendeine Haftung für die Nutzung dieser Informationen, für deren Wirtschaftlichkeit oder fehlerfreie Funktion für einen bestimmten Zweck. Ferner kann der Verlag für Schäden, die auf einer Fehlfunktion von Programmen oder ähnliches zurückzuführen sind, nicht haftbar gemacht werden. Auch nicht für die Verletzung von Patent- und anderen Rechten Dritter, die daraus resultieren. Eine telefonische oder schriftliche Beratung durch den Verlag über den Einsatz der Programme ist nicht möglich. Der Verlag übernimmt keine Gewähr dafür, dass die beschriebenen Verfahren, Programme usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürfen. Der Verlag hat sich bemüht, sämtliche Rechteinhaber von Abbildungen zu ermitteln. Sollte dem Verlag gegenüber dennoch der Nachweis der Rechtsinhaberschaft geführt werden, wird das branchenübliche Honorar gezahlt.

Bibliografische Information Der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

Alle Rechte vorbehalten

3. Auflage 2005

© Elsevier GmbH, München

Spektrum Akademischer Verlag ist ein Imprint der Elsevier GmbH.

05 06 07 08 09 5 4 3 2 1 0

Für Copyright in Bezug auf das verwendete Bildmaterial siehe Abbildungsnachweis.

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Planung und Lektorat: Dr. Andreas Rüdinger, Bianca Alton

Redaktion: Martin Radke

Herstellung: Ute Kreutzer

Umschlaggestaltung: SpieszDesign, Neu-Ulm

Titelfotografie: © zefa/stockbyte

Satz: Steingraeber Satztechnik, Dossenheim

Druck und Bindung: LegoPrint S.p.A.; Lavis

Printed in Italy

ISBN 3-8274-1595-0

7.1 Einführung

Seit den Anfängen der Rechner wünschen sich Programmierer einen unbegrenzt großen und unendlich schnellen Speicher. Die Konzepte dieses Kapitels helfen, dem Benutzer einen scheinbar unbegrenzten, schnellen Speicher vorzuspiegeln. Bevor wir uns damit beschäftigen, wie diese Illusion entsteht, wollen wir eine einfache Analogie betrachten, die die wichtigsten Prinzipien und Mechanismen verdeutlicht, die wir verwenden werden.

Stellen Sie sich vor, Sie sind ein Student, der eine Arbeit über wichtige historische Entwicklungen auf dem Gebiet der Computerhardware schreibt. Sie sitzen an einem Schreibtisch in einer Bibliothek mit einer Auswahl von Büchern, die Sie aus den Regalen geholt haben, und lesen. Sie stellen fest, dass in den Ihnen vorliegenden Büchern einige wichtige Computer beschrieben sind, über die Sie schreiben wollen, dass es dort aber keine Informationen über den EDSAC-Rechner gibt. Deshalb gehen Sie zurück zu den Regalen und suchen nach einem weiteren Buch. Sie finden ein Buch über die ersten Rechner in Großbritannien, in dem auch der EDSAC beschrieben wird. Sofern Sie eine sinnvolle Auswahl an Büchern auf Ihrem Schreibtisch haben, besteht eine hohe Wahrscheinlichkeit, dass viele der Informationen, die Sie brauchen, in diesen Büchern zu finden sind, und Sie können einen Großteil Ihrer Zeit mit Nachforschungen in den Büchern auf Ihrem Schreibtisch verbringen, ohne zurück zu den Regalen gehen zu müssen. Haben mehrere Bücher auf Ihrem Schreibtisch Platz, sparen Sie Zeit, da Sie nicht jeweils nur ein Buch auf dem Schreibtisch haben und ständig zu den Regalen gehen müssen, um es zurückzubringen und ein anderes Buch zu holen.

Dasselbe Prinzip erlaubt uns, die Illusion eines großen Speichers zu erzeugen, auf den wir genauso schnell zugreifen können wie auf einen sehr kleinen Speicher. So wie Sie nicht gleichzeitig und mit derselben Wahrscheinlichkeit alle Bücher in der Bibliothek benötigen, muss ein Programm nicht auf seinen gesamten Code oder auf alle seine Daten gleichzeitig mit derselben Wahrscheinlichkeit zugreifen. Andernfalls wäre es unmöglich, einen Großteil der Speicherzugriffe schnell zu machen und dennoch viel Speicher im Computer zu haben, so wie es für Sie unmöglich wäre, alle Bücher aus der Bibliothek auf Ihrem Schreibtisch zu stapeln, und dennoch die gesuchte Information schnell zu finden.

Dieses *Lokalitätsprinzip* liegt sowohl Ihrer Arbeit in der Bibliothek als auch der Arbeitsweise von Programmen zugrunde. Das Lokalitätsprinzip besagt, dass Programme zu jedem beliebigen Zeitpunkt jeweils nur auf einen relativ kleinen Teil ihres Adressraums zugreifen, so wie Sie nur auf einen kleinen Teil der Bücher in der Bibliothek zugegriffen haben. Es gibt zwei verschiedene Arten von Lokalität:

- **Temporale Lokalität (temporal locality)**: Dieses Prinzip besagt, dass, wenn ein Zugriff auf eine Datenposition erfolgt ist, mit hoher Wahrscheinlichkeit bald wieder ein Zugriff darauf erfolgt.
- **Räumliche Lokalität (spacial locality)**: Das Lokalitätsprinzip besagt, dass nach einem Zugriff auf eine Datenposition mit hoher Wahrscheinlichkeit auch bald ein Zugriff auf benachbarte Adressen erfolgt.

Temporale Lokalität (temporal locality) Dieses Prinzip besagt, dass, wenn ein Zugriff auf eine Datenposition erfolgt ist, mit hoher Wahrscheinlichkeit bald wieder ein Zugriff darauf erfolgt.

Räumliche Lokalität (spacial locality) Das Lokalitätsprinzip besagt, dass nach einem Zugriff auf eine Datenposition mit hoher Wahrscheinlichkeit auch bald ein Zugriff auf benachbarte Adressen erfolgt.

zu verstärken. Wir werden später in diesem Kapitel noch sehen, wie die räumliche Lokalität in Speicherhierarchien genutzt wird.

So wie der Zugriff auf Bücher auf einem Schreibtisch eine ganz natürliche Lokalität aufweist, entsteht die Lokalität in Programmen aus einfachen und natürlichen Programmstrukturen. Beispielsweise enthalten die meisten Programme Schleifen, deshalb ist es wahrscheinlich, dass wiederholt ein Zugriff auf die darin verwendeten Befehle und Daten erfolgt, wodurch eine große temporale Lokalität entsteht. Weil der Zugriff auf Befehle normalerweise sequenziell erfolgt, weisen Programme eine hohe räumliche Lokalität auf. Der Zugriff auf Daten weist ebenfalls räumliche Lokalität auf. Insbesondere Zugriffe auf Elemente eines Arrays oder eines Datensatzes zeichnen sich durch hohe räumliche Lokalität aus.

Wir machen uns das Lokalitätsprinzip zu Nutze, indem wir den Speicher eines Computers als **Speicherhierarchie** aufbauen. Eine Speicherhierarchie besteht aus mehreren Speicherebenen mit unterschiedlichen Geschwindigkeiten und Größen. Die schnelleren Speicher sind pro Bit teurer als die langsameren Speicher – und deshalb kleiner.

Heute gibt es drei primäre Technologien für den Aufbau von Speicherhierarchien. Der Hauptspeicher wird mit DRAM (Dynamic Random Access Memory, dynamischer Speicher mit wahlfreiem Zugriff) implementiert, während Ebenen, die sich näher am Prozessor befinden (Caches), SRAM (Static Random Access Memory, statischer Speicher mit wahlfreiem Zugriff) verwenden. DRAM ist kostengünstiger pro Bit als SRAM, aber auch wesentlich langsamer. Die Preisdifferenz entsteht, weil DRAM wesentlich weniger Platz pro Speicherbit auf dem Chip verwendet, und DRAMs damit auf derselben Siliziumfläche eine größere Speicherkapazität aufweisen; die Geschwindigkeitsdifferenz entsteht aufgrund mehrerer Faktoren, die in Abschnitt B.9 in **Appendix B** auf CD beschrieben werden. Die dritte Technologie, die für die Implementierung der größten und langsamsten Hierarchieebene verwendet wird, ist die Festplatte. Die Zugriffszeit und der Preis pro Bit unterscheiden sich zwischen diesen Technologien ganz wesentlich, wie die nachfolgende Tabelle für typische Werte aus dem Jahr 2004 zeigt:

Speichertechnologie	Typische Zugriffszeit	\$ pro GB im Jahr 2004
SRAM	0,5–5 ns	\$4000–\$10 000
DRAM	50–70 ns	\$100–\$200
Festplatte	5 000 000–20 000 000 ns	\$0,50–\$2

Aufgrund dieser Unterschiede in Hinblick auf Kosten und Zugriffszeit ist es vorteilhaft, Speicher hierarchisch aufzubauen. Abbildung 7.1 zeigt, dass sich der schnellere Speicher näher am Prozessor befindet und dass der langsamere, billigere Speicher darunter angeordnet ist. Das Ziel ist, dem Benutzer so viel Speicher der billigeren Technologie wie möglich bereitzustellen, während der Zugriff mit der von dem schnellsten Speicher gebotenen Geschwindigkeit erfolgt.

Das Speichersystem ist als Hierarchie aufgebaut: Eine näher am Prozessor befindliche Ebene ist im Allgemeinen eine Untergruppe aller weiter entfernten Ebenen. Auf der untersten Ebene werden alle Daten gespeichert. In der hier verwendeten Analogie bilden die Bücher auf Ihrem Schreibtisch eine Untergruppe der Bibliothek, in der Sie arbeiten, die wiederum eine Untergruppe aller Bibliotheken auf dem Universitätsgelände ist. Wenn wir uns weiter vom Prozessor entfernen, weisen die Ebenen außerdem immer längere Zugriffszeiten auf, so wie es auch in einer Hierarchie von Bibliotheken auf dem Universitätsgelände sein könnte. Die Analogie geht jedoch an der folgenden Stelle fehl: Wenn ein Buch aus einem Regal entnommen und zum Arbeitstisch getragen wird, so ist es physisch nicht mehr im Regal vorhanden. Im Gegensatz dazu sind alle Datenelemente einer höheren Speicherebene Kopien der Datenelemente in den niedrigeren Speicherebenen. Die Daten sind somit physisch mehrfach vorhanden.

Speicherhierarchie (*memory hierarchy*) Eine Struktur, die mehrere Speicherebenen verwendet; je größer die Distanz zur CPU wird, desto größer werden die Speicher und desto länger ist die Zugriffszeit.



Geschwindigkeit	CPU	Größe	Kosten (\$/Bit)	Aktuelle Technologie
Schnellster	Speicher	Kleinstes	Höchste	SRAM
	Speicher			DRAM
Langsamster	Speicher	Größter	Niedrigste	Festplatte

Abb. 7.1 Der grundlegende Aufbau einer Speicherhierarchie. Durch die Implementierung des Speichersystems als Hierarchie entsteht beim Benutzer der Eindruck eines Speichers, der so groß wie der Speicher auf der untersten Ebene der Hierarchie ist, auf den er jedoch Zugriff hat, als wäre der Speicher vollständig aus dem schnellsten Speicher aufgebaut.

Eine Speicherhierarchie kann aus mehreren Ebenen bestehen, aber Daten werden nur jeweils gleichzeitig zwischen zwei benachbarten Ebenen übertragen, deshalb können wir unsere Aufmerksamkeit auf zwei Ebenen konzentrieren. Die obere Ebene – die näher am Prozessor liegt – ist kleiner und schneller (weil sie eine aufwändigeren Technologie verwendet) als die untere Ebene. Abbildung 7.2 zeigt die kleinstmögliche Informationseinheit, die in der zweistufigen Hierarchie vorhanden oder nicht vorhanden sein kann. Dieser wird *Block* oder *Zeile* genannt; in unserer Bibliotheksanalogie entspricht ein Buch einem solchen Block.

Wenn die vom Prozessor angeforderten Daten in einem Block auf der oberen Ebene liegen, spricht man von einem *Treffer* (was analog dazu ist, wenn Sie die gesuchte Information in einem Ihrer Bücher auf Ihrem Schreibtisch finden). Werden die Daten auf der oberen Ebene nicht gefunden, wird die Anforderung als *Fehlzugriff* bezeichnet. Es erfolgt ein Zugriff auf die untere Ebene der Hierarchie, um den Block mit den angeforderten Daten zu finden. (Um es mit unserer Analogie zu veranschaulichen: Sie

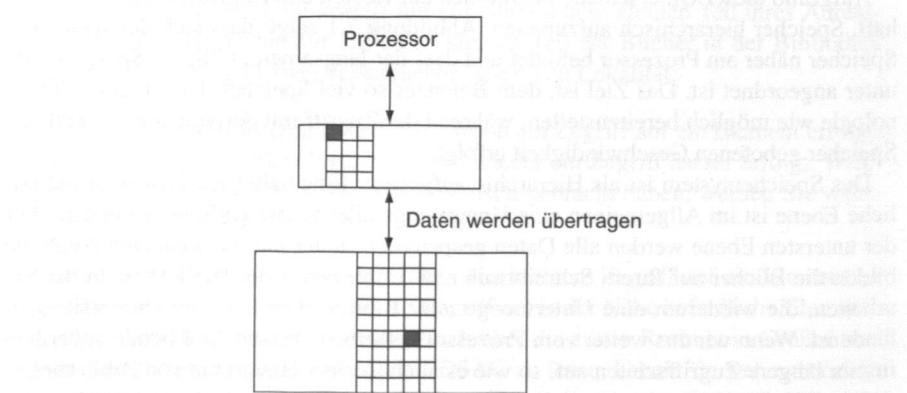


Abb. 7.2 Datentransfer zwischen den Ebenen der Speicherhierarchie. Innerhalb jeder Ebene ist die Informationseinheit, die vorhanden oder nicht vorhanden ist, ein so genannter *Block*. Normalerweise wird beim Kopieren zwischen zwei Ebenen ein ganzer Block übertragen.

gehen von Ihrem Schreibtisch zu den Regalen, um das gewünschte Buch zu suchen.) Die **Trefferrate (hit rate)** oder das **Trefferverhältnis** ist der Anteil der Speicherzugriffe, die auf der oberen Ebene befriedigt werden; sie wird häufig als Leistungsmaß für die Speicherhierarchie verwendet.

Die **Fehlzugriffsrate (miss rate)** ($1 - \text{Trefferrate}$) ist der Anteil der Speicherzugriffe, die nicht auf der oberen Ebene befriedigt werden.

Weil die Ausführungsgeschwindigkeit der wichtigste Grund für die Verwendung einer Speicherhierarchie ist, spielt die Zeit für die Verarbeitung von Treffern und von Fehlzugriffen eine große Rolle.

Die **Zugriffszeit bei Treffer (hit time)** ist die Zeit für den Zugriff auf die obere Ebene der Speicherhierarchie, worin auch die Zeit enthalten ist, die benötigt wird, um festzustellen, ob der Zugriff ein Treffer oder ein Fehlzugriff ist (d.h. die Zeit, die benötigt wird, um die Bücher auf dem Schreibtisch zu durchsuchen).

Der **Fehlzugriffsaufwand (miss penalty)** ist die Zeit für den Austausch eines Blocks in der oberen Ebene durch den entsprechenden Block aus der unteren Ebene, zuzüglich der Zeit, dem Prozessor diesen Block bereitzustellen (oder in unserem Beispiel die Zeit, ein neues Buch aus den Regalen zu holen und auf den Schreibtisch zu legen). Weil die obere Ebene kleiner ist und unter Verwendung schnellerer Speicherbausteine aufgebaut wird, ist die Zugriffszeit bei einem Treffer kleiner als die Zugriffszeit auf die nächste Hierarchieebene, was den größten Anteil des Fehlzugriffsaufwands ausmacht. (In den Büchern auf dem Schreibtisch können Sie viel schneller nachsehen, als wenn Sie aufstehen und ein neues Buch aus dem Regal holen müssen.)

Wie wir in diesem Kapitel sehen werden, wirken sich die Konzepte für den Aufbau von Speichersystemen auf viele andere Aspekte eines Rechners aus, unter anderem darauf, wie das Betriebssystem den Speicher sowie Ein-/Ausgaben verwaltet, wie Compiler Code erzeugen, und sogar darauf, wie Anwendungen den Computer nutzen. Weil alle Programme einen Großteil ihrer Ausführungszeit mit Speicherzugriffen verbringen, ist das Speichersystem notwendigerweise ein wesentlicher Faktor für die Leistung, d.h. die Ausführungsgeschwindigkeit. Die Abhängigkeit der Leistung von der Speicherhierarchie bedeutet, dass Programmierer, die daran gewöhnt waren, sich den Speicher als flaches Speichergerät mit wahlfreiem Zugriff vorzustellen, jetzt die Speicherhierarchien verstehen müssen, um eine gute Leistung zu erzielen. Wir zeigen anhand von zwei Beispielen, wie wichtig dieses Verständnis ist.

Weil Speichersysteme von so großer Bedeutung für die Ausführungsgeschwindigkeit sind, haben die Computerentwickler diesen Systemen große Aufmerksamkeit gewidmet und komplexe Mechanismen für eine verbesserte Leistung des Speichersystems geschaffen. In diesem Kapitel werden wir die wichtigsten Konzepte betrachten, wobei jedoch viele Vereinfachungen und Abstraktionen genutzt werden, um das Material sowohl vom Umfang als auch von der Komplexität her überschaubar zu machen. Wir hätten mit Leichtigkeit Hunderte von Seiten über Speichersysteme schreiben können, wie viele neuere Doktorarbeiten zeigen.

Welche der folgenden Aussagen sind allgemein gültig?

Caches

1. nutzen die temporale Lokalität.
 2. Beim Lesen ist der zurückgegebene Wert davon abhängig, welche Blöcke sich im Cache befinden.
 3. Die größten Kosten der Speicherhierarchie entstehen auf der obersten Ebene.
-

Trefferrate (hit rate) Der Anteil der Speicherzugriffe, bei denen der gesuchte Block in einer Ebene der Speicherhierarchie (z.B. in einem Cache) gefunden wird.

Fehlzugriffsrate (miss rate) Der Anteil der Speicherzugriffe, bei denen der gesuchte Block nicht innerhalb einer Ebene der Speicherhierarchie gefunden wird.

Zugriffszeit bei Treffer (hit time) Die Zeit für den Zugriff auf eine Ebene der Speicherhierarchie, einschließlich der Zeit, die benötigt wird, um festzustellen, ob der Zugriff ein Treffer oder ein Fehlzugriff ist.

Fehlzugriffsaufwand (miss penalty) Die Zeit, die benötigt wird, um einen Block von einer unteren Ebene in eine höhere Ebene der Speicherhierarchie zu laden. Diese beinhaltet die Zeit für die Übertragung und das Einfügen des Blocks in die höhere Ebene, auf der der Fehlzugriff stattgefunden hat, sowie die Zeit für den Zugriff auf den Block durch den Prozessor.





Programme weisen sowohl temporale Lokalität (die Tendenz zur Wiederverwendung zuvor bereits benutzter Datenelemente) als auch räumliche Lokalität (die Tendenz, auf Datenelemente zuzugreifen, die in der Nähe von Datenelementen liegen, auf die bereits zugegriffen wurde) auf. Speicherhierarchien nutzen die temporale Lokalität, indem sie Datenelemente, auf die vor kurzem zugegriffen wurde, näher am Prozessor vorhalten. Speicherhierarchien nutzen die räumliche Lokalität, indem sie ganze Blöcke aus mehreren benachbarten Wörtern in Speicher auf höhere Hierarchieebenen verschieben.

Abbildung 7.3 zeigt, dass eine Speicherhierarchie in der Nähe des Prozessors kleinere Speicher mit schnelleren Speichertechnologien nutzt. Treffer in der höchsten Hierarchieebene führen zu einer schnelleren Verarbeitung. Zugriffe, die fehlschlagen, gehen weiter auf niedrigere Ebenen der Hierarchie, die durch größere aber langsamere Speicher realisiert sind. Wenn die Trefferrate hoch genug ist, hat die Speicherhierarchie also eine effektive Zugriffszeit, die nahe an der Zugriffszeit der höchsten (und schnellsten) Ebene liegt, und eine Größe, die gleich der Größe der untersten (und größten) Ebene ist.

In den meisten Systemen ist der Speicher eine echte Hierarchie, d.h. Daten können nicht auf Ebene i vorliegen, wenn sie nicht auch auf Ebene $i+1$ vorliegen.

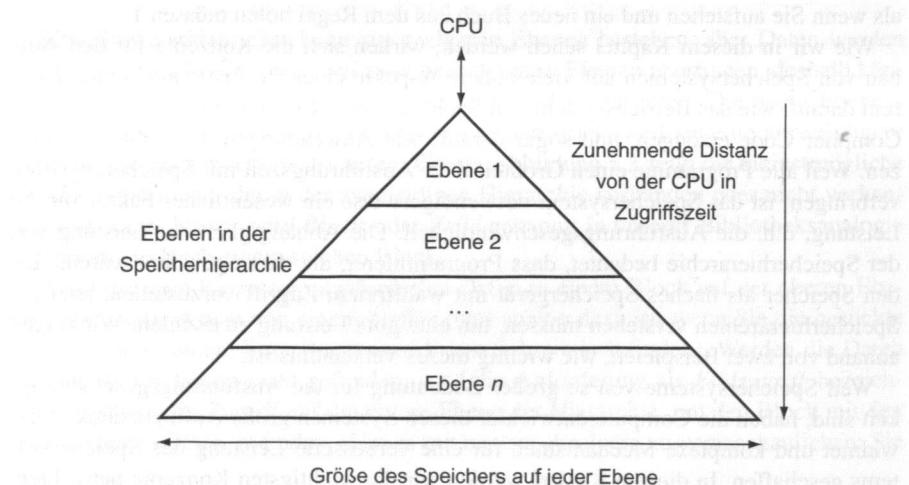


Abb. 7.3 Diese Zeichnung zeigt den Aufbau einer Speicherhierarchie: Je größer die Distanz zum Prozessor wird, desto größer ist auch der Speicher. Dieser Aufbau erlaubt in Kombination mit den richtigen Betriebsmechanismen, dass der Prozessor eine Zugriffszeit erzielt, die hauptsächlich durch Ebene 1 der Hierarchie bestimmt wird, und dennoch über einen Speicher der Größe von Ebene n verfügt. Die Bewahrung dieser Illusion ist Thema dieses Kapitels. Obwohl die unterste Ebene der Hierarchie normalerweise von der Festplatte gebildet wird, verwenden einige Systeme Bandlaufwerke oder einen Dateiserver über ein lokales Netzwerk als weitere Ebene der Hierarchie.

7.2

Caches – Grundlagen

In unserem Bibliotheksbeispiel hat der Schreibtisch als Cache gedient – als sicherer Platz, an dem wir Dinge (Bücher) aufbewahren, die wir genauer betrachten wollen.

Der Name *Cache* wurde beim ersten kommerziell verfügbaren Computer, der mit dieser zusätzlichen Ebene ausgestattet war, gewählt, um die Ebene der Speicherhierarchie zwischen dem Prozessor und dem Hauptspeicher zu bezeichnen. Auch heute wird das Wort *Cache* noch hauptsächlich in diesem Sinne verwendet, aber der Begriff wird auch für die Bezeichnung beliebigen Speichers eingesetzt, der Zugriffslokalität nutzt. Caches erschienen erstmals Anfang der 60er-Jahre in den ersten Forschungsrechnern und noch im selben Jahrzehnt auch in kommerziellen Rechnern; heute enthalten alle Allzweckrechner Caches, von den Servern bis hin zu eingebetteten Prozessoren.

In diesem Abschnitt betrachten wir zunächst einen sehr einfachen Cache, in dem die Prozessoranforderungen jeweils ein Wort umfassen und auch die Blöcke nur ein Wort groß sind. (Leser, die bereits mit den Grundlagen von Caches vertraut sind, können Abschnitt 7.2 überspringen.) Abbildung 7.4 zeigt einen solchen einfachen Cache vor und nach der Anforderung eines Datenelements, das sich anfänglich nicht im Cache befindet. Vor dem Zugriff enthält der Cache eine Menge von Wörtern X_1, X_2, \dots, X_{n-1} . Der Prozessor fordert ein Wort X_n an, das sich nicht im Cache befindet. Diese Anforderung führt zu einem Fehlzugriff und das Wort X_n wird aus dem Speicher in den Cache geladen.

Betrachtet man das in Abbildung 7.4 gezeigte Szenario, stellen sich zwei Fragen: Woher wissen wir, ob sich ein Datenelement im Cache befindet? Und wenn es sich dort befindet, wie finden wir es? Die Antworten auf diese beiden Fragen hängen eng zusammen. Wenn jedes Wort an genau einer Stelle im Cache stehen kann, ist es ganz einfach, das Wort zu finden, falls es sich im Cache befindet. Die einfachste Methode, wie man für jedes Wort im Speicher eine Position im Cache zuweist, ist es, die Cache-Position abhängig von der *Adresse* des Worts im Speicher zuzuweisen. Diese Cache-Struktur wird als **direkt abgebildet** bezeichnet, weil jede Speicheradresse auf genau eine Position im Cache abgebildet wird. Die Abbildung von Adressen auf Cache-Positionen ist

Cache: a safe place for hiding or storing things.

Webster's New World
Dictionary of the American
Language, Third College
Edition (1988)

Direkt abgebildeter Cache (*direct-mapped cache*) Eine Cache-Struktur, bei der jede Speicheradresse auf genau eine Position im Cache abgebildet wird.

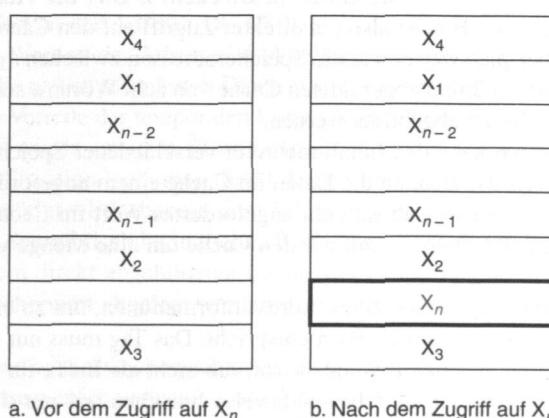
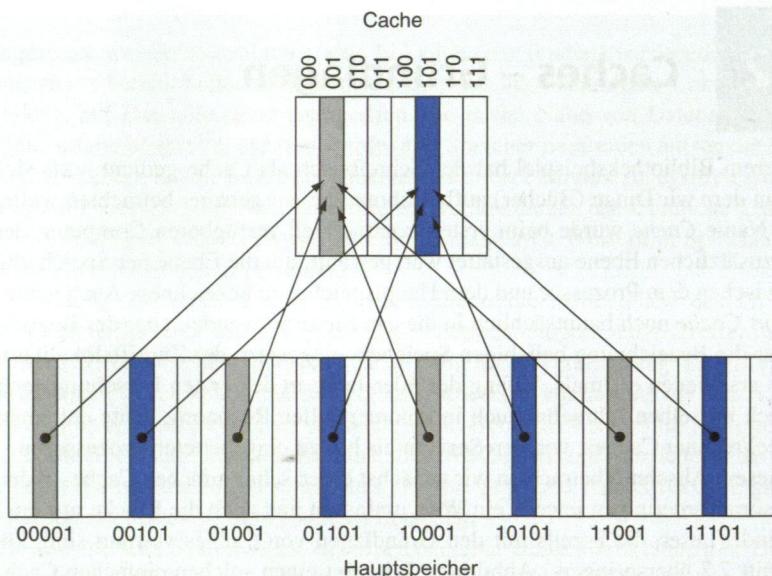


Abb. 7.4 Der Cache vor und nach dem Zugriff auf ein Wort X_n , das sich anfänglich nicht im Cache befindet. Diese Speicheranforderung verursacht einen Fehlzugriff, der den Cache veranlasst, X_n aus dem Speicher zu laden und in den Cache einzufügen.



Der **Modulo-Wert n** wird als maximale Anzahl der Blöcke gewählt, die in den Cache passen. Dieser Wert ist immer eine Zweierpotenz. $\log_2 n$ ist dann die Anzahl der zur Adressierung benötigten Adressbits, die als niedrigwertigster Teil einer Speicheradresse gewählt werden. Das gilt jedoch nur, falls, wie hier vereinfacht dargestellt, die Blockgröße der Speicherwertgröße entspricht.

Abb. 7.5 Direkt abgebildeter Cache mit acht Einträgen, bei dem Adressen von Speicherwörtern zwischen 0 und 31 auf dieselben Cache-Positionen abgebildet werden. Weil es acht Wörter im Cache gibt, wird eine Adresse X auf das Cache-Wort X modulo 8 abgebildet. Das heißt, die unteren $\log_2(8) = 3$ Bit werden als Cache-Index verwendet. Somit werden die Adressen 00001_B , 01001_B , 10001_B und 11001_B auf Eintrag 001_B des Caches abgebildet, während die Adressen 00101_B , 01101_B , 10101_B und 11101_B alle auf Eintrag 101_B des Caches abgebildet werden.

für einen direkt abgebildeten Cache normalerweise einfach. Beispielsweise verwenden fast alle direkt abgebildeten Caches die Abbildung:

$$(\text{Blockadresse}) \bmod (\text{Anzahl der Cache-Blöcke im Cache})$$

Diese Abbildung ist geschickt gewählt, denn wenn die maximale Anzahl n der Blöcke im Cache eine Zweierpotenz ist, dann kann die Cache-Position einfach unter Verwendung der unteren \log_2 (Cache-Größe in Blöcken) n Bits der Adresse berechnet werden. Mit den unteren Bits ist also ein direkter Zugriff auf den Cache möglich. Abbildung 7.5 zeigt beispielsweise, wie die Speicheradressen zwischen 1_D (00001_B) und 29_D (11101_B) in einem direkt abgebildeten Cache von acht Wörtern auf die Positionen 1_D (001_B) und 5_D (101_B) abgebildet werden.

Jede Cache-Position kann den Inhalt mehrerer verschiedener Speicheradressen enthalten. Woher wissen wir also, ob die Daten im Cache einem angeforderten Wort entsprechen? Woher wissen wir, ob sich ein angefordertes Wort im Cache befindet oder nicht? Wir lösen das Problem, indem wir den Cache um eine Menge von **Tags** erweitern.

Die Tags enthalten die notwendigen Adressinformationen, um zu erkennen, ob ein Wort im Cache dem angeforderten Wort entspricht. Das Tag muss nur den oberen Teil der Adresse enthalten, der den Bits entspricht, die nicht als Index für den Cache verwendet werden. In Abbildung 7.5 beispielsweise brauchen wir nur die oberen 2 der 5 Adressbits für das Tag, weil das Indexfeld der Adresse mit den unteren 3 Bits den Block auswählt. Wir nehmen die Index-Bits nicht in das Tag mit auf, da sie redundant wären, weil das Indexfeld jeder Adresse per Definition denselben Wert haben muss.

Außerdem müssen wir erkennen können, wenn ein Cache-Block keine gültige Information enthält. Beispielsweise enthält der Cache beim Programmstart keine brauchbaren Daten, und die Tag-Felder sind bedeutungslos. Selbst nach der Ausführung vieler Befehle können einige der Cache-Einträge immer noch leer sein, wie in Abbildung 7.4

Tag Ein Feld in einer Tabelle, die für eine Ebene der Speicherhierarchie verwendet wird. Dieses Feld enthält die Adressinformation, die man benötigt, um zu erkennen, ob der zugehörige Block in der entsprechenden Hierarchieebene einem angeforderten Wort entspricht.

gezeigt. Wir müssen also wissen, dass das Tag für solche Einträge ignoriert werden soll. Die gebräuchlichste Methode ist es, ein **Gültigkeits-Bit (valid bit)** vorzusehen, das anzeigt, ob ein Eintrag eine gültige Adresse enthält. Ist das Bit nicht gesetzt, dann kann der Block nicht der gesuchte sein.

Im restlichen Abschnitt konzentrieren wir uns auf die Erklärung, wie Leseoperationen in einem Cache ausgeführt werden, und wie sich die Cache-Steuerung beim Lesen verhält.

Im Allgemeinen ist die Verarbeitung von Leseoperationen etwas einfacher als die Verarbeitung von Schreiboperationen, weil bei Leseoperationen der Inhalt des Caches nicht geändert werden muss. Nachdem wir die Grundlagen für Leseoperationen im Cache und die Verarbeitung von Cache-Fehlzugriffen betrachtet haben, beschäftigen wir uns mit Cache-Designs für echte Computer und zeigen, wie diese Caches Schreiboperationen verarbeiten.

Zugriff auf einen Cache

Die Tabelle 7.1 zeigt den Inhalt eines acht Wörter großen, direkt abgebildeten Caches, der auf eine Folge von Speicherzugriffen durch den Prozessor reagiert. Weil es acht Blöcke im Cache gibt, geben die unteren 3 Bits einer Adresse die Blocknummer an. Für jeden Zugriff wird die folgende Aktion ausgeführt:

Dezimal-adresse der Referenz	Binäradresse der Referenz	Treffer oder Fehl-zugriff im Cache	Zugeordneter Cache-Block (wo die Daten gefunden oder platziert werden)
22	10110 _B	Fehlzugriff (7.1 b)	(10110 _B mod 8) = 110 _B
26	11010 _B	Fehlzugriff (7.1 c)	(11010 _B mod 8) = 010 _B
22	10110 _B	Treffer	(10110 _B mod 8) = 110 _B
26	11010 _B	Treffer	(11010 _B mod 8) = 010 _B
16	10000 _B	Fehlzugriff (7.1 d)	(10000 _B mod 8) = 000 _B
3	00011 _B	Fehlzugriff (7.1 e)	(00011 _B mod 8) = 011 _B
16	10000 _B	Treffer	(10000 _B mod 8) = 000 _B
18	10010 _B	Fehlzugriff (7.1 f)	(10010 _B mod 8) = 010 _B

Wenn das Wort an der Adresse 18 (10010_B) in den Cache-Block 2 (010_B) geladen wird, muss das Wort an der Adresse 26 (11010_B), das sich im Cache-Block 2 (010_B) befand, durch die neu angeforderten Daten ersetzt werden. Dieses Verhalten erlaubt einem Cache, die Vorteile der temporalen Lokalität zu nutzen: Wörter, auf die vor kurzer Zeit zugegriffen wurde, ersetzen Wörter, auf die schon längere Zeit kein Zugriff mehr erfolgte. Diese Situation kann man direkt damit vergleichen, dass ein neues Buch aus den Regalen benötigt wird, aber auf dem Schreibtisch kein Platz mehr frei ist – ein bereits auf Ihrem Schreibtisch befindliches Buch muss deshalb ins Regal zurückgestellt werden. In einem direkt abgebildeten Cache gibt es nur einen Platz, wo das neu angeforderte Speicherwort abgelegt werden kann, und damit nur eine Wahl, was ersetzt werden soll.

Wir wissen, wo wir im Cache nach möglichen Adressen suchen müssen: Die unteren Bits einer Adresse werden verwendet, um den eindeutigen Cache-Eintrag zu finden, auf den die Adresse abgebildet werden könnte. Abbildung 7.6 zeigt, wie eine referenzierte Adresse unterteilt wird in

- einen Cache-Index, der für die Auswahl des Blocks verwendet wird, und
- ein Tag-Feld, das mit dem Wert des Tag-Felds des Caches verglichen wird.

Gültigkeits-Bit (valid bit) Ein Feld in den Tabellen einer Speicherhierarchie, das angibt, ob der zugehörige Block gültige Daten enthält.

Tab. 7.1 Der Cache-Inhalt nach jeder Zugriffsanforderung, die einen Fehlzugriff verursacht, wobei Index- und Tag-Felder binär dargestellt sind. Der Cache ist anfänglich leer und die Gültigkeits-Bits (V-Eintrag im Cache) sind nicht gesetzt (N). Der Prozessor fordert die folgenden Adressen an: 10110_B (Fehlzugriff), 11010_B (Fehlzugriff), 10110_B (Treffer), 11010_B (Treffer), 10000_B (Fehlzugriff), 00011_B (Fehlzugriff), 10000_B (Treffer) und 10010_B (Fehlzugriff). Die Abbildungen zeigen den Cache-Inhalt, nachdem die aufeinander folgenden Fehlzugriffe verarbeitet wurden. Wenn ein Zugriff auf die Adresse $10010_B(18)$ erfolgt, muss der Eintrag für die Adresse $11010_B(26)$ ersetzt werden, und ein Zugriff auf 11010_B verursacht einen nachfolgenden Fehlzugriff. Das Tag-Feld enthält nur den oberen Teil der Adresse. Die vollständige Adresse eines Worts im Cache-Block i mit Tag-Feld j für diesen Cache ist $j \times 8 + i$ oder äquivalent die Konkatenation des Tag-Feldes j und des Index i. Im unten gezeigten Cache f. beispielsweise hat der Index 010 das Tag 10 und entspricht der Adresse 10010_B .

Index	V	Tag	Daten
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a. Der Ausgangszustand des Caches nach dem Einschalten.

Index	V	Tag	Daten
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	J	10_B	Speicher(10110_B)
111	N		

b. Nach Verarbeitung eines Fehlzugriffs auf Adresse (10110_B).

Index	V	Tag	Daten
000	N		
001	N		
010	J	11_B	Speicher(11010_B)
011	N		
100	N		
101	N		
110	J	10_B	Speicher(10110_B)
111	N		

c. Nach Verarbeitung eines Fehlzugriffs auf Adresse (11010_B).

Index	V	Tag	Daten
000	J	10_B	Speicher(10000_B)
001	N		
010	J	11_B	Speicher(11010_B)
011	N		
100	N		
101	N		
110	J	10_B	Speicher(10110_B)
111	N		

d. Nach Verarbeitung eines Fehlzugriffs auf Adresse (10000_B).

Index	V	Tag	Daten
000	J	10_B	Speicher(10000_B)
001	N		
010	J	11_B	Speicher(11010_B)
011	J	00_B	Speicher(00011_B)
100	N		
101	N		
110	J	10_B	Speicher(10110_B)
111	N		

e. Nach Verarbeitung eines Fehlzugriffs auf Adresse (00011_B).

Index	V	Tag	Daten
000	J	10_B	Speicher(10000_B)
001	N		
010	J	10_B	Speicher(10010_B)
011	J	00_B	Speicher(00011_B)
100	N		
101	N		
110	J	10_B	Speicher(10110_B)
111	N		

f. Nach Verarbeitung eines Fehlzugriffs auf Adresse (10010_B).

Der Index eines Cache-Blocks gibt in Kombination mit dem Tag-Inhalt dieses Blocks eindeutig die Speicheradresse des in dem Cache-Block enthaltenen Worts an. Weil das Indexfeld als Adresse für den Zugriff auf den Cache verwendet wird und weil ein n Bit großes Feld 2^n Werte haben kann, muss die Gesamtzahl der Einträge in einem direkt abgebildeten Cache eine Zweierpotenz sein. In der MIPS-Architektur werden Wörter

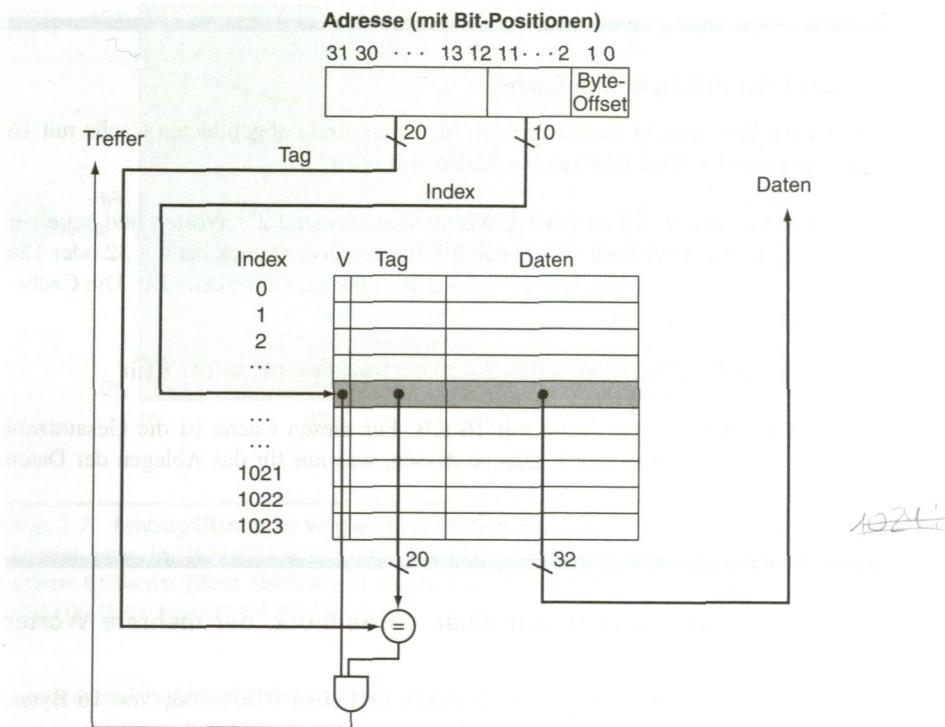


Abb. 7.6 Bei diesem Cache wird der untere Teil der Adresse verwendet, um einen Cache-Eintrag auszuwählen, der aus einem Datenwort und einem Tag besteht. Das Tag für den Cache wird mit dem oberen Teil der Adresse verglichen, um festzustellen, ob der Eintrag im Cache der angeforderten Adresse entspricht. Weil der Cache 2^{10} (oder 1024) Wörter enthält und eine Blockgröße von 1 Wort aufweist, werden 10 Bits verwendet, um den Cache zu indizieren, so dass $32 - 10 - 2 = 20$ Bits bleiben, die mit dem Tag verglichen werden müssen. Wenn das Tag und die oberen 20 Bits der Adresse gleich sind und das Gültigkeits-Bit (V) gesetzt ist, erzeugt die Anforderung einen Treffer im Cache und das Wort wird dem Prozessor bereitgestellt. Andernfalls erfolgt ein Fehlzugriff.

an Vielfachen von 4 Bytes ausgerichtet, deshalb geben die beiden niedrigstwertigen Bits jeder Adresse ein Byte innerhalb eines Worts an und werden bei der Auswahl des Worts im Block ignoriert.

Die Gesamtzahl der für einen Cache benötigten Bits ist eine Funktion der Cache-Größe und der Adressgröße, weil der Cache sowohl den Speicherplatz für die Daten als auch für die Tags umfasst. Die Größe des oben gezeigten Blocks betrug ein Wort, aber normalerweise ist ein Block mehrere Wörter groß. Unter Voraussetzung einer 32-Bit-Adresse benötigt ein direkt abgebildeter Cache mit einer Größe von 2^n Blöcken mit 2^m -Wort-(2^{m+2} -Byte)-Blöcken ein Tag-Feld der Größe $32 - (n + m + 2)$ Bit, weil n Bit für den Index verwendet werden, m Bit für das Wort innerhalb des Blocks und 2 Bit für den Byte-Teil der Adresse. Die Gesamtzahl der Bits in einem direkt abgebildeten Cache beträgt $2^n \times (\text{Blockgröße} + \text{Tag-Größe} + \text{Gültigkeitsfeldgröße})$. Weil die Blockgröße 2^m Wörter (2^{m+5} Bit) und die Adressgröße 32 Bit beträgt, ist die Anzahl der Bits in einem solchen Cache gleich $2^n \times (m \times 32 + (32 - n - m - 2) + 1) = 2^n \times (m \times 32 + 31 - n - m)$. Die Namenskonvention schließt jedoch die Größe des Tag-Feldes und das Gültigkeitsfeld aus und zählt nur die Größe der Daten.

BEISPIEL**ANTWORT****Anzahl der Bits in einem Cache**

Wie viele Bits braucht man insgesamt für einen direkt abgebildeten Cache mit 16 KB Daten und 4-Wort-Blöcken bei 32-Bit-Adressen?

Wir wissen, dass 16 KB gleich 4K Wörter sind, das sind 2^{12} Wörter, und bei einer Blockgröße von 4 Wörtern (2^2) gleich 2^{10} Blöcke. Jeder Block hat 4×32 oder 128 Bit Daten plus ein Tag mit $32 - 10 - 2 - 2$ Bit plus ein Gültigkeits-Bit. Die Cache-Gesamtgröße beträgt also

$$2^{10} \times (128 + (32 - 10 - 2 - 2) + 1) = 2^{10} \times 147 = 147 \text{ KBit}$$

oder 18,4 KB für einen Cache mit 16 KB. Für diesen Cache ist die Gesamtzahl der Bits im Cache etwa das 1,15fache dessen, was nur für das Ablegen der Daten benötigt wird.

BEISPIEL**ANTWORT****Abbildung einer Adresse auf einen Cache-Block, der mehrere Wörter umfasst**

Betrachten wir einen Cache mit 64 Blöcken und einer Blockgröße von 16 Bytes. Auf welche Blocknummer wird die Byteadresse 1200 abgebildet?

Die Formel wurde auf Seite 386 gezeigt. Der Block wird berechnet mit

$$\left(\frac{\text{Byteadresse}}{\text{Bytes pro Block}} \right) \bmod (\text{Anzahl der Cache-Blöcke})$$

Dabei ist die Adresse des Blocks gleich

$$\left[\frac{\text{Byteadresse}}{\text{Bytes pro Block}} \right]$$

Beachten Sie, dass diese Blockadresse der Block ist, der alle Adressen zwischen

$$\left[\frac{\text{Byteadresse}}{\text{Bytes pro Block}} \right] \times \text{Bytes pro Block}$$

und

$$\left[\frac{\text{Byteadresse}}{\text{Bytes pro Block}} \right] \times \text{Bytes pro Block} + (\text{Bytes pro Block} - 1)$$

enthält. Bei 16 Bytes pro Block ist die Byteadresse 1200 also die Blockadresse

$$\left[\frac{1200}{16} \right] = 75$$

Die Blockadresse wird auf die Cache-Blocknummer ($75 \bmod 64 = 11$) abgebildet. Dieser Block bildet tatsächlich alle Adressen zwischen 1200 und 1215 ab.

Größere Blöcke nutzen räumliche Lokalität, um die Fehlzugriffsraten zu senken. Wie Abbildung 7.7 zeigt, sinkt die Fehlzugriffsraten bei steigender Blockgröße. Die Fehlzugriffsraten steigt jedoch wieder, wenn die Blockgröße zu einem wesentlichen Teil der Cache-Größe wird. Dann können nur noch sehr wenige Blöcke im Cache abgelegt

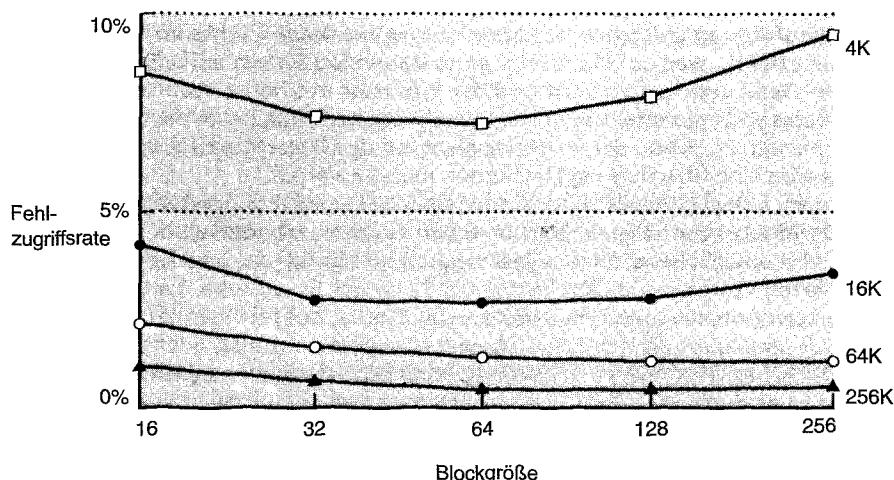


Abb. 7.7 Fehlzugriffsrate im Vergleich zur Blockgröße. Beachten Sie, dass die Fehlzugriffsrate ansteigt, wenn die Blockgröße im Verhältnis zur Cache-Größe zu groß ist. Jede Linie stellt einen Cache anderer Größe dar. (Diese Abbildung ist unabhängig von der Assoziativität, die später beschrieben wird.) Die Daten basieren auf den SPEC92-Benchmarks.

werden, und es gibt sehr viel Konkurrenz um diese Blöcke. Demzufolge wird ein Block aus dem Cache geworfen, noch bevor ein Zugriff auf viele seiner Wörter erfolgt ist. Anders ausgedrückt, die räumliche Lokalität zwischen den Wörtern in einem Block ist bei großen Blöcken kleiner und damit der Vorteil geringer.

Ein noch bedeutenderes Problem, das bei Vergrößerung der Blöcke entsteht, ist, dass die Kosten eines Fehlzugriffs steigen. Der Fehlzugriffsaufwand wird bestimmt durch die Zeit, die erforderlich ist, um den Block von der nächst niedrigeren Hierarchieebene zu holen und in den Cache zu laden. Die Zeit für das Laden besteht aus zwei Komponenten: der Latenz bis zum Laden des ersten Wortes und der Übertragungszeit für den Rest des Blocks. Wenn wir das Speichersystem nicht ändern, steigt mit der Blockgröße die Übertragungszeit – und damit der Fehlzugriffsaufwand.

Darüber hinaus beginnt die Verbesserung der Fehlzugriffsrate zu sinken, wenn die Blöcke größer werden. Das Ergebnis ist, dass die Erhöhung des Fehlzugriffsaufwands die Verminderung der Fehlzugriffsrate für große Blöcke überwiegt, und die Cache-Leistung damit sinkt. Wenn wir allerdings den Speicher darauf auslegen, dass größere Blöcke effizienter übertragen werden, können wir die Blockgröße steigern und erhalten weitere Verbesserungen der Cache-Leistung. Wir werden im nächsten Abschnitt auf dieses Thema zurückkommen.

Vertiefung: Der größte Nachteil bei der Erhöhung der Blockgröße ist, dass der Fehlzugriffsaufwand für den Cache steigt. Es ist zwar schwierig, etwas gegen die Latenzkomponente des Fehlzugriffsaufwands zu unternehmen, aber möglicherweise können wir einen Teil der Übertragungszeit verbergen, so dass der Fehlzugriffsaufwand effektiv kleiner wird. Die einfachste Methode ist der so genannte *Early Restart*. Dabei wird die Programmausführung bereits fortgesetzt, sobald das angeforderte Wort geladen ist, statt auf das Laden des gesamten Blocks zu warten. Viele Prozessoren verwenden diese Technik für den Zugriff auf Befehle, wo sie auch am besten funktioniert. Zugriffe auf Befehle erfolgen größtenteils sequenziell. Wenn also das Speichersystem zu jedem Taktzyklus ein Wort bereitstellen kann, ist der Prozessor möglicherweise in der Lage, die Ausführung des Befehls, der den Fehlzugriff ausgelöst hat, zu starten, sobald das angeforderte Befehlswort geladen wurde, und das Speichersystem liefert die weiteren Befehlwörter gerade rechtzeitig („just in time“), so dass keine weiteren Verzögerungen in der Pro-



grammausführung entstehen. Diese Technik ist für Daten-Caches im Allgemeinen weniger effektiv, weil auf die Wörter eines Datenblocks meist auf weniger vorhersehbare Weise zugegriffen wird und der Prozessor mit hoher Wahrscheinlichkeit ein anderes Wort aus einem anderen Cache-Block benötigt, bevor die Übertragung abgeschlossen ist. Wenn der Prozessor nicht auf den Daten-Cache zugreifen kann, weil gerade eine Übertragung stattfindet, muss er warten.

Ein noch komplexeres Schema ist, den Speicher so zu organisieren, dass das angeforderte Wort zuerst vom Speicher in den Cache übertragen wird. Der restliche Block wird anschließend übertragen, beginnend mit der Adresse hinter dem angeforderten Wort bis zum Blockende. Am Ende des Blocks wird die Übertragung im direkten Anschluss mit den Adressen am Blockanfang fortgesetzt. Diese Technik, auch als *Requested Word First* (*Angefordertes Wort zuerst*) oder *Critical Word First* (*Kritisches Wort zuerst*) bezeichnet, kann etwas schneller als ein Early Restart sein, ist aber durch dieselben Eigenschaften eingeschränkt, die den Early Restart beschränken.

Verarbeitung von Cache-Fehlzugriffen

Cache-Fehlzugriff (cache miss)
Eine Anforderung von Daten aus dem Cache, die nicht erfüllt werden kann, weil die Daten nicht im Cache vorliegen.

Bevor wir den Cache eines realen Systems betrachten, wollen wir überprüfen, wie die Steuerungseinheit mit **Cache-Fehlzugriffen (cache misses)** umgeht. Die Steuerungseinheit muss einen Fehlzugriff erkennen und diesen verarbeiten, indem die angeforderten Daten aus dem Speicher (oder, wie wir noch sehen werden, aus einem Cache auf einer darunter liegenden Ebene) geladen werden. Wenn der Cache einen Treffer anzeigt, verwendet der Rechner die Daten weiter, als sei nichts geschehen. Wir können also dieselbe grundlegende Steuerung verwenden, wie wir sie in Kapitel 5 entwickelt und in Kapitel 6 für das Pipelining angepasst haben. Die Speicher im Datenpfad in den Kapiteln 5 und 6 werden einfach durch Caches ersetzt.

Die Anpassung der Steuerung eines Prozessors für Cache-Treffer ist trivial. Für Fehlzugriffe ist jedoch ein gewisser Zusatzaufwand erforderlich. Die Behandlung eines Cache-Fehlzugriffs erfolgt mit der Prozessor-Steuerungseinheit und mit einem separaten Controller, der den Speicherzugriff initiiert und den Cache wieder füllt. Die Verarbeitung eines Cache-Fehlzugriffs erzeugt einen Stillstand, ähnlich dem Pipeline-Stillstand, der in Kapitel 6 beschrieben wurde (im Gegensatz zu einem Interrupt, bei dem der Status aller Register gespeichert werden müsste). Für einen Cache-Fehlzugriff können wir den gesamten Prozessor stillstehen lassen, wobei wir den Inhalt der temporären und für den Programmierer sichtbaren Register einfrieren, während wir auf den Speicher warten. Im Gegensatz dazu sind die in Kapitel 6 beschriebenen Pipeline-Stillstände komplexer, weil wir einige Befehle weiter ausführen müssen, während andere stillstehen.

Jetzt betrachten wir genauer, wie die von Befehlen ausgelösten Cache-Fehlzugriffe für den Multizyklen- oder Pipeline-Datenpfad verarbeitet werden. Derselbe Ansatz kann ganz einfach auf die Verarbeitung der von Daten ausgelösten Fehlzugriffe erweitert werden. Wenn ein Befehlszugriff zu einem Fehlzugriff führt, ist der Inhalt des Befehlsregisters ungültig. Um den richtigen Befehl in den Cache zu laden, müssen wir in der Lage sein, die untere Ebene in der Speicherhierarchie anzugeben, eine Leseoperation auszuführen. Weil der Befehlszähler im ersten Taktzyklus der Ausführung inkrementiert wird, und zwar sowohl in Pipeline- als auch in Multizyklen-Prozessoren, ist die Adresse des Befehls, der einen Fehlzugriff für den Befehls-Cache erzeugt hat, gleich dem Wert des Befehlszählers minus 4. Sobald wir die Adresse haben, müssen wir den Hauptspeicher anwählen, eine Leseoperation auszuführen. Wir warten darauf, dass der Speicher antwortet (weil der Zugriff mehrere Zyklen lang dauert), und schreiben dann die Wörter in den Cache.

Jetzt können wir die Schritte definieren, die bei einem Befehls-Cache-Fehlzugriff auszuführen sind:

1. Den ursprünglichen Befehlszählerwert (aktueller Befehlszähler – 4) an den Speicher senden.
2. Den Hauptspeicher anweisen, eine Leseoperation auszuführen, und darauf warten, dass der Speicher seinen Zugriff abschließt.
3. Den Cache-Eintrag füllen, wobei die Daten aus dem Speicher in den Datenteil des Eintrags eingefügt, die oberen Bits der Adresse (aus der ALU) in das Tag-Feld geschrieben und das Gültigkeits-Bit gesetzt werden.
4. Den Befehl erneut laden, wobei er diesmal im Cache zu finden ist.

Die Steuerung des Caches bei einem Datenzugriff ist im Wesentlichen identisch: Bei einem Fehlzugriff bleibt der Prozessor einfach im Stillstand, bis der Speicher mit den Daten antwortet.

Schreiboperationen verarbeiten

Für Schreiboperationen funktioniert das Ganze etwas anders. Angenommen, wir haben bei einem Speicherbefehl die Daten nur in den Daten-Cache geschrieben (ohne den Hauptspeicher zu ändern); nachdem wir dann in den Cache geschrieben haben, enthält der Speicher einen anderen Wert als der Cache. In einem solchen Fall sagt man, Cache und Speicher sind *inkonsistent*. Die einfachste Methode, Hauptspeicher und Cache konsistent zu halten, ist es, Daten immer sowohl in den Speicher als auch in den Cache zu schreiben. Dieses Schema wird als **Durchschreibetechnik (write-through)** bezeichnet.

Der andere wichtige Aspekt bei Schreiboperationen ist, was bei einem Schreib-Fehlzugriff passiert. Zuerst holen wir die Wörter des Blocks aus dem Speicher. Nachdem der Block geladen und in den Cache gespeichert wurde, können wir das Wort überschreiben, das den Fehlzugriff im Cache-Block verursacht hat. Außerdem schreiben wir das Wort unter Verwendung der vollständigen Adresse in den Hauptspeicher.

Dieses Design verarbeitet Schreiboperationen auf sehr einfache Weise, bietet aber keine gute Leistung. Bei einem Durchschreibeschema bewirkt jede Schreiboperation, dass die Daten in den Hauptspeicher geschrieben werden. Diese Schreiboperationen dauern lang, oftmals mindestens 100 Prozessortaktzyklen, was den Prozessor wesentlich verlangsamen könnte. Bei den SPEC2000 Integer-Benchmarks beispielsweise sind 10 % der Befehle Speicheroperationen. Wenn der CPI ohne Cache-Fehlzugriffe gleich 1,0 ist, würde der Aufwand von 100 zusätzlichen Taktten bei jeder Schreiboperation zu einem CPI von $1,0 + 100 \times 10\% = 11$ führen, wodurch sich die Ausführungsgeschwindigkeit um mehr als den Faktor 10 verschlechtern würde.

Eine Lösung für dieses Problem ist die Verwendung eines **Schreibpuffers (write buffer)**. Ein Schreibpuffer speichert die Daten, die darauf warten, dass sie in den Speicher geschrieben werden. Nachdem die Daten in den Cache und in den Schreibpuffer geschrieben wurden, kann der Prozessor die Ausführung fortsetzen. Wenn eine Schreiboperation in den Hauptspeicher abgeschlossen ist, wird der Eintrag aus dem Schreibpuffer gelöscht. Ist der Schreibpuffer voll und der Prozessor will eine Schreiboperation ausführen, muss der Prozessor stillstehen, bis ein Platz im Schreibpuffer frei ist. Wenn natürlich die Geschwindigkeit, in der der Speicher Schreiboperationen ausführt kann, kleiner ist als die Geschwindigkeit, in der der Prozessor Schreiboperationen erzeugt, kann keine noch so große Pufferung helfen, weil die Schreiboperationen schneller erzeugt werden, als das Speichersystem sie entgegennehmen kann.

Die Geschwindigkeit, in der Schreiboperationen erzeugt werden, kann auch *kleiner* sein als die Geschwindigkeit, in der der Speicher sie entgegennehmen kann, und dennoch können Stillstände auftreten. Das passiert, wenn die Schreiboperationen in

Durchschreibetechnik (write-through) Ein Schema, bei dem Schreiboperationen immer sowohl den Cache als auch den Speicher aktualisieren, so dass sichergestellt ist, dass die Daten zwischen Speicher und Cache immer konsistent sind.

Schreibpuffer (write buffer) Ein FIFO-Puffer, der die Daten aufnimmt, die darauf warten, dass sie in den Speicher geschrieben werden.

Bündeln (bursts) auftreten. Um das Auftreten solcher Stillstände zu reduzieren, erhöhen die Prozessoren üblicherweise die Tiefe des Schreibpuffers auf mehr als einen einzigen Eintrag.

Die Alternative zu einem Durchschreibeschema ist ein Schema, das als **Rückschreibetechnik (write-back)** bezeichnet wird. Bei einem Rückschreibeschema wird bei einer Schreiboperation der neue Wert nur in den Block im Cache geschrieben. Der veränderte Block wird erst dann in die untere Hierarchieebene geschrieben, wenn er ausgetauscht wird. Rückschreibeschemata können die Ausführungsgeschwindigkeit erhöhen, insbesondere, wenn Prozessoren Schreiboperationen schneller erzeugen, als sie vom Hauptspeicher verarbeitet werden können; ein Rückschreibeschema ist jedoch komplexer zu implementieren als ein Durchschreibeschema.

Im restlichen Abschnitt beschreiben wir Caches aus realen Prozessoren. Wir betrachten dabei insbesondere die Verarbeitung von Lese- und Schreiboperationen. In Abschnitt 7.5 beschreiben wir die Verarbeitung von Schreiboperationen detaillierter.



Vertiefung: Schreiboperationen verursachen verschiedene Komplikationen für Caches, die bei Leseoperationen nicht auftreten. Hier beschreiben wir zwei davon: Die Vorgehensweise bei Schreib-Fehlzugriffen sowie die effiziente Implementierung von Schreiboperationen in Rückschreibe-Caches.

Betrachten wir einen Fehlzugriff in einem Durchschreibe-Cache. Die in den meisten Entwürfen von Durchschreibe-Caches angewandte Strategie wird als *Fetch-on-miss*, *Fetch-on-write* oder manchmal als *Allocate-on-miss* bezeichnet. Sie reserviert einen Cache-Block für die Adresse, die den Fehlzugriff verursacht hat, und lädt den restlichen Block in den Cache, bevor die Daten geschrieben werden und die Ausführung fortgesetzt wird. Alternativ könnten wir entweder den Block im Cache reservieren, aber die Daten nicht laden (als *no-fetch-on-write* bezeichnet), oder den Block überhaupt nicht reservieren (als *no-allocate-on-write* bezeichnet). Ein anderer Name für diese Strategien, die geschriebenen Daten nicht im Cache abzulegen, ist *Write-around*, weil die Daten um den Cache herum geschrieben werden, um in den Speicher zu gelangen. Die Motivation für diese Schemata ist die Beobachtung, dass Programme manchmal ganze Datenblöcke schreiben, bevor sie sie lesen. In solchen Fällen ist der Ladevorgang, der dem ersten Schreib-Fehlzugriff zugeordnet ist, möglicherweise unnötig. Es gibt viele subtile Aspekte bei der Implementierung dieser Schemata in Mehrwort-Blöcken, einschließlich einer komplizierteren Verarbeitung der Schreibtreffer, weil Mechanismen erforderlich sind, die denen für Rückschreibe-Caches sehr ähnlich sind.

Die effiziente Implementierung von Speicheroperationen für einen Rückschreibe-Cache ist komplexer als für einen Durchschreibe-Cache. In einem Rückschreibe-Cache müssen wir den Block zurück in den Speicher schreiben, wenn die Daten im Cache „dirty“ sind (also verändert wurden) und wir einen Cache-Fehlzugriff haben. Würden wir den Block durch einen Speicherbefehl überschreiben, bevor wir wissen, ob die Speicheroperation einen Treffer im Cache erzeugt hat (wie wir es für einen Durchschreibe-Cache machen könnten), würden wir den Inhalt des Blocks zerstören, der im Speicher nicht gesichert ist. Ein Durchschreibe-Cache kann die Daten in den Cache schreiben und das Tag lesen; wenn das Tag nicht übereinstimmt, tritt ein Fehlzugriff auf. Weil der Cache ein Durchschreibe-Cache ist, ist das Überschreiben des Blocks im Cache keine Katastrophe, weil der Speicher den korrekten Wert enthält.

In einem Rückschreibe-Cache können wir den Block nicht überschreiben, deshalb sind für Speicheroperationen entweder zwei Takte erforderlich (ein Takt für die Überprüfung auf einen Treffer, gefolgt von einem Takt für die eigentliche Ausführung der Schreiboperation), oder man braucht einen zusätzlichen Puffer, einen so genannten *Speicherpuffer (store buffer)*, der diese Daten aufnimmt – wodurch es letztlich möglich wird, dass die Speicheroperation nur einen Takt benötigt, weil ein Pipelining dafür stattfindet. Bei Verwendung eines Speicherpuffers führt der Prozessor den Tag-Vergleich durch und speichert die Daten während des normalen Cache-Zugriffstakts in den Speicherpuffer. Bei einem Cache-Treffer werden

Rückschreibetechnik (write-back) Ein Schema, das Schreiboperationen verarbeitet, indem es Werte nur in dem Block im Cache aktualisiert, und den veränderten Block erst dann in die untere Hierarchieebene schreibt, wenn der Block ausgetauscht wird.

die neuen Daten aus dem Speicherpuffer im nächsten freien Zugriffszyklus in den Cache geschrieben.

Im Gegensatz dazu können Schreiboperationen in einem Durchschreibe-Cache immer in einem Takt ausgeführt werden. Es gibt jedoch einige zusätzliche Komplikationen bei Mehrwort-Blöcken, weil wir das Tag nicht einfach überschreiben können, wenn wir Daten schreiben. Stattdessen lesen wir das Tag und schreiben den Datenteil des ausgewählten Blocks. Wenn das Tag mit der Adresse des zu aktualisierenden Blocks übereinstimmt, kann der Prozessor normal weiterarbeiten, weil der richtige Block aktualisiert wurde. Stimmt das Tag nicht überein, erzeugt der Prozessor einen Schreib-Fehlzugriff, um den restlichen Block zu laden, der zu der Adresse gehört, an die geschrieben wird. Weil es immer sicher ist, die Daten zu überschreiben, benötigen Schreibtreffer ebenfalls nur einen Zyklus.

Viele Rückschreibe-Caches enthalten auch Schreibpuffer (*write buffer*), die verwendet werden, um den Fehlzugriffsaufwand zu reduzieren, wenn bei einem Fehlzugriff ein Dirty-Block ersetzt wird. In einem solchen Fall wird, während der angeforderte Block aus dem Speicher gelesen wird, der Dirty-Block in einen Rückschreibepuffer (*write-back buffer*) verschoben, der dem Cache zugeordnet ist. Der Inhalt des Rückschreibepuffers wird später, beispielsweise wenn der Bus wieder frei ist, in den Speicher geschrieben. Vorausgesetzt, es passiert nicht unmittelbar danach ein weiterer Fehlzugriff, halbiert diese Technik den Fehlzugriffsaufwand für den Fall, dass ein Dirty-Block ersetzt werden muss.

Ein Beispiel-Cache: Der Intrinsity-FastMATH-Prozessor

Der Intrinsity FastMATH ist ein schneller, eingebetteter Mikroprozessor, der eine MIPS-Architektur sowie eine einfache Cache-Implementierung verwendet. Am Ende des Kapitels werden wir das komplexere Cache-Design des Intel Pentium 4 betrachten, aber wir beginnen aus pädagogischen Gründen mit diesem einfachen aber doch realen Beispiel. Abbildung 7.8 zeigt den Aufbau des Daten-Caches des Intrinsity FastMATH.

Dieser Prozessor hat eine zwölfstufige Pipeline, ähnlich der in Kapitel 6 beschriebenen. Der Prozessor wird unter Höchstlast in jedem Takt sowohl ein Befehlswort als auch ein Datenwort anfordern. Um die Anforderungen der Pipeline zu erfüllen, ohne einen Stillstand zu erzeugen, werden separate Befehls- und Daten-Caches verwendet. Jeder Cache umfasst 16 KB oder 4 K Wörter mit 16-Wort-Blöcken.

Leseanforderungen für den Cache sind einfach zu implementieren. Weil es separate Daten- und Befehls-Caches gibt, braucht man separate Steuerungssignale, um die Caches zu lesen oder in sie zu schreiben. (Sie wissen, dass wir den Befehls-Cache aktualisieren müssen, wenn ein Fehlzugriff auftritt.) Die Schritte für eine Leseanforderung eines der Caches sehen also wie folgt aus:

1. Die Adresse wird an den entsprechenden Cache gesendet. Die Adresse stammt entweder vom Befehlszähler (für einen Befehl) oder von der ALU (für Daten).
2. Wenn der Cache einen Treffer signalisiert, steht das angeforderte Wort auf den Datenleitungen zur Verfügung. Weil es 16 Wörter in dem gewünschten Block gibt, müssen wir das richtige Wort auswählen. Ein Blockindexfeld wird verwendet, um den Multiplexer zu steuern (unten in der Abbildung gezeigt), der das angeforderte Wort aus den 16 Wörtern in dem indizierten Block auswählt.
3. Wenn der Cache einen Fehlzugriff signalisiert, senden wir die Adresse an den Hauptspeicher. Wenn der Speicher die Daten zurückgibt, schreiben wir sie in den Cache und lesen sie dann, um die Anforderung zu erfüllen.

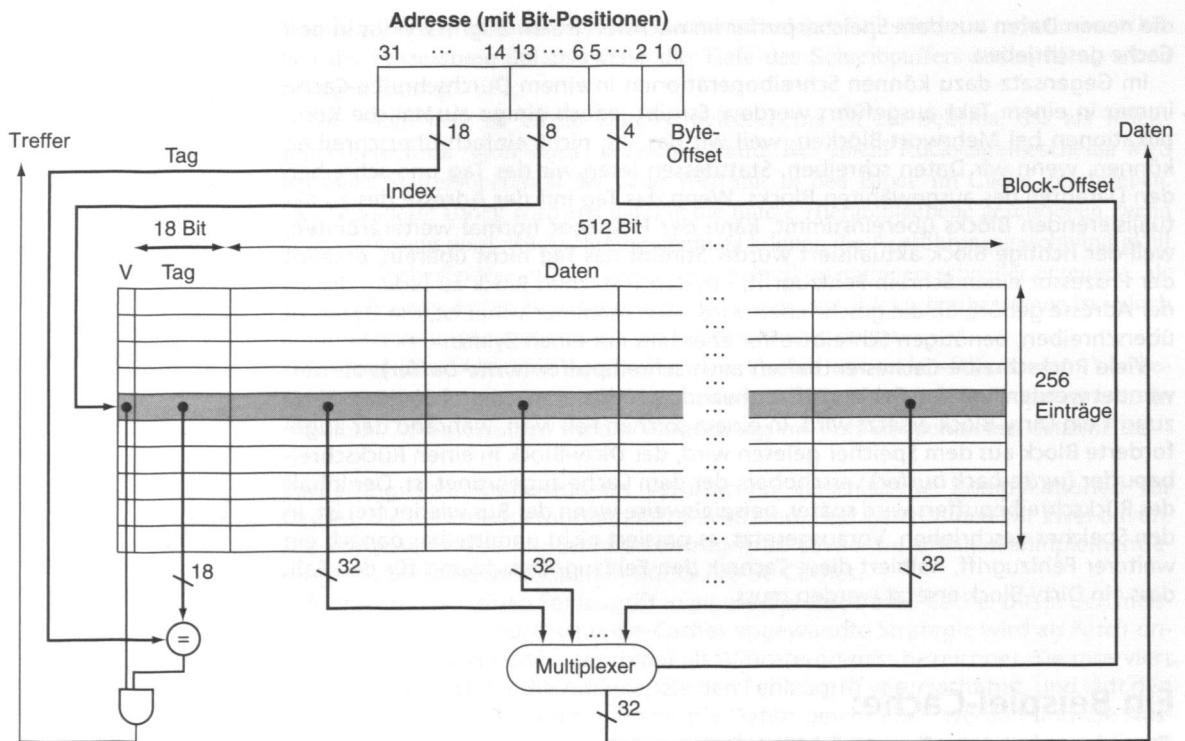


Abb. 7.8 Die 16 KB großen Caches im Intrinsity FastMATH enthalten je 256 Blöcke mit 16 Wörtern pro Block. Das Tag-Feld ist 18 Bit breit und das Indexfeld ist 8 Bit breit, während ein 4-Bit-Feld (Bits 5–2) verwendet wird, um den Block zu indizieren und das Wort unter Verwendung eines 16:1-Multiplexers auszuwählen. Das Gültigkeits-Bit (V) signalisiert, ob ein Block Gültigkeit besitzt. In der Praxis verwenden Caches ein separates großes RAM für die Daten und ein kleineres RAM für die Tags, um den Multiplexer zu eliminieren, wobei der Block-Offset die zusätzlichen Adressbits für das große Daten-RAM bereitstellt. In diesem Fall ist das RAM 32 Bit breit und muss 16-mal so viele Wörter wie Blöcke im Cache haben.

Für Schreiboperationen unterstützt der Intrinsity FastMATH sowohl die Durchschreibetechnik als auch die Rückschreibetechnik und überlässt dem Betriebssystem die Auswahl, welche Strategie für eine Anwendung verwendet werden soll. Der Prozessor besitzt einen Schreibpuffer, der einen Eintrag aufnehmen kann.

Welche Cache-Fehlzugriffsichten entstehen bei einer Cache-Struktur wie der im Intrinsity FastMATH verwendeten? Tabelle 7.2 zeigt die Fehlzugriffsichten für die Befehls- und Daten-Caches für die SPEC2000-Integer-Benchmarks. Die kombinierte Fehlzugriffsrate ist die effektive Fehlzugriffsrate pro Referenz für jedes Programm nach Berücksichtigung der unterschiedlichen Häufigkeiten von Befehls- und Datenzugriffen.

Tab. 7.2 Angenäherte Fehlzugriffsichten für Befehle und Daten für den Prozessor Intrinsity FastMATH mit SPEC2000-Benchmarks. Die kombinierte Fehlzugriffsrate ist die effektive Fehlzugriffsrate, die für den Befehls-Cache (16 KB) und Daten-Cache (16 KB) aufgetreten ist. Sie wird ermittelt, indem die Fehlzugriffsichten für Befehle und Daten nach der Häufigkeit von Befehls- und Datenzugriffen gewichtet werden.

Fehlzugriffsrate für Befehle	Fehlzugriffsrate für Daten	Kombinierte effektive Fehlzugriffsrate
0,4 %	11,4 %	3,2 %

Obwohl die Fehlzugriffsrate eine wichtige Eigenschaft von Cache-Entwürfen darstellt, ist das ultimative Maß die Auswirkung des Speichersystems auf die Programm-ausführungszeit. Wir werden gleich sehen, in welchem Verhältnis Fehlzugriffsrate und Ausführungszeit zueinander stehen.

Vertiefung: Ein kombinierter Cache mit einer Gesamtgröße gleich der Summe der beiden getrennten Caches weist im Allgemeinen eine bessere Trefferrate auf. Diese höhere Rate entsteht, weil der kombinierte Cache die Anzahl der Einträge, die von Befehlen verwendet werden können, nicht streng von denen trennt, die von Daten verwendet werden können. Nichtsdestotrotz verwenden viele Prozessoren einen getrennten Befehls- und Daten-Cache, um die Cache-Bandbreite zu erhöhen.

Nachfolgend sehen Sie die Fehlzugriffsichten für Caches einer Größe, wie man sie beim Intrinsic-FastMATH-Prozessor findet, und für einen kombinierten Cache, dessen Größe gleich der Gesamtgröße der beiden Caches ist:

- Cache-Gesamtgröße: 32 KB
- Effektive Fehlzugriffsrate der getrennten Caches: 3,24 %
- Fehlzugriffsrate des kombinierten Caches: 3,18 %

Die Fehlzugriffsrate der getrennten Caches ist nur unmerklich schlechter.

Der Vorteil bei der Verdopplung der Cache-Bandbreite durch die Unterstützung eines gleichzeitigen Befehls- und Datenzugriffs überwiegt den Nachteil einer etwas schlechteren Fehlzugriffsrate bei Weitem. Diese Beobachtung ist ein weiterer Hinweis darauf, dass wir die Fehlzugriffsrate nicht als einziges Maß für die Cache-Leistung verwenden können, wie in Abschnitt 7.3 gezeigt.



Getrennte Caches (*split caches*) Ein Schema, bei dem sich eine Ebene der Speicherhierarchie aus zwei voneinander unabhängigen Caches zusammensetzt, die parallel zueinander arbeiten, wobei der eine Befehle, der andere Daten verarbeitet.

Design des Speichersystems zur Unterstützung von Caches

Cache-Fehlzugriffe werden aus dem Hauptspeicher bedient, der aus DRAM-Bausteinen aufgebaut ist. In Abschnitt 7.1 haben wir gesehen, dass DRAMs hauptsächlich auf Dichte und nicht auf Zugriffszeit ausgelegt sind. Es ist zwar schwierig, die Latenz zu reduzieren, die anfällt, bis das erste Wort aus dem Speicher geholt wird, aber wir können den Fehlzugriffsaufwand reduzieren, wenn wir die Bandbreite vom Speicher zum Cache erhöhen. Diese Reduzierung erlaubt, größere Blöcke zu verwenden, und dabei dennoch einen geringen Fehlzugriffsaufwand beizubehalten, ähnlich dem für einen kleineren Block.

Der Prozessor ist normalerweise über einen Bus an den Speicher angeschlossen. Die Taktrate des Busses ist im Allgemeinen viel langsamer als diejenige des Prozessors und zwar um einen Faktor von bis zu 10. Die Geschwindigkeit dieses Busses wirkt sich auf den Fehlzugriffsaufwand aus.

Um den Einfluss der verschiedenen Speicherorganisationen zu verstehen, definieren wir eine Menge hypothetischer Speicherzugriffszeiten. Wir nehmen folgendes an:

- 1 Speicherbus-Taktzyklus für das Senden der Adresse
- 15 Speicherbus-Taktzyklen für jeden initiierten DRAM-Zugriff
- 1 Speicherbus-Taktzyklus für das Senden eines Datenworts

Wenn wir einen Cache-Block von vier Wörtern und eine ein Wort breite DRAM-Bank haben, wäre der Fehlzugriffsaufwand gleich $1 + 4 \times 15 + 4 \times 1 = 65$ Speicherbus-Taktzyklen. Die Anzahl der Bytes, die pro Speicherbus-Taktzyklus für einen einzigen Fehlzugriff anfallen, sind also

$$\frac{4 \times 4}{65} \approx 0,25$$

Abbildung 7.9 zeigt drei Optionen für das Design eines Speichersystems. Die erste Option folgt dem, was wir bisher angenommen haben: Der Speicher ist ein Wort breit und alle Zugriffe erfolgen sequenziell. Die zweite Option erhöht die Bandbreite zum Speicher, indem sie den Speicher sowie die Busse zwischen Prozessor und Speicher verbreitert; auf diese Weise erhalten wir parallelen Zugriff auf alle Wörter des Blocks. Die dritte Option erhöht die Bandbreite, indem sie den Speicher, nicht aber den Verbindungsbus verbreitert. Wir haben also immer noch einen Aufwand für die Übertragung jedes Worts, aber wir können vermeiden, dass der Aufwand für die Zugriffslatenz mehr als einmal anfällt. Wir wollen untersuchen, wie die beiden letzteren Optionen den Fehlzugriffsaufwand von 65 Taktzyklen verbessern, der für die erste Option anfällt (Abbildung 7.9a).

Wenn man die Breite des Speichers und des Busses erhöht, erhöht sich proportional dazu die Speicherbandbreite, womit sowohl die Zugriffszeit- als auch die Übertragungszeitanteile des Fehlzugriffsaufwands sinken.

Mit einer Hauptspeicherbreite von zwei Wörtern fällt der Fehlzugriffsaufwand von 65 Speicherbus-Taktzyklen auf $1 + 2 \times 15 + 2 \times 1 = 33$ Speicherbus-Taktzyklen. Bei einem vier Wörter breiten Speicher beträgt der Fehlzugriffsaufwand gerade 17 Speicherbus-Taktzyklen. Die Bandbreite für einen einzigen Fehlzugriff beträgt dann 0,48 Byte pro Bus-Taktzyklus für einen Speicher, der zwei Wörter breit ist (fast das Doppelte), und 0,94 Byte pro Bus-Taktzyklus, wenn der Speicher vier Wörter breit ist

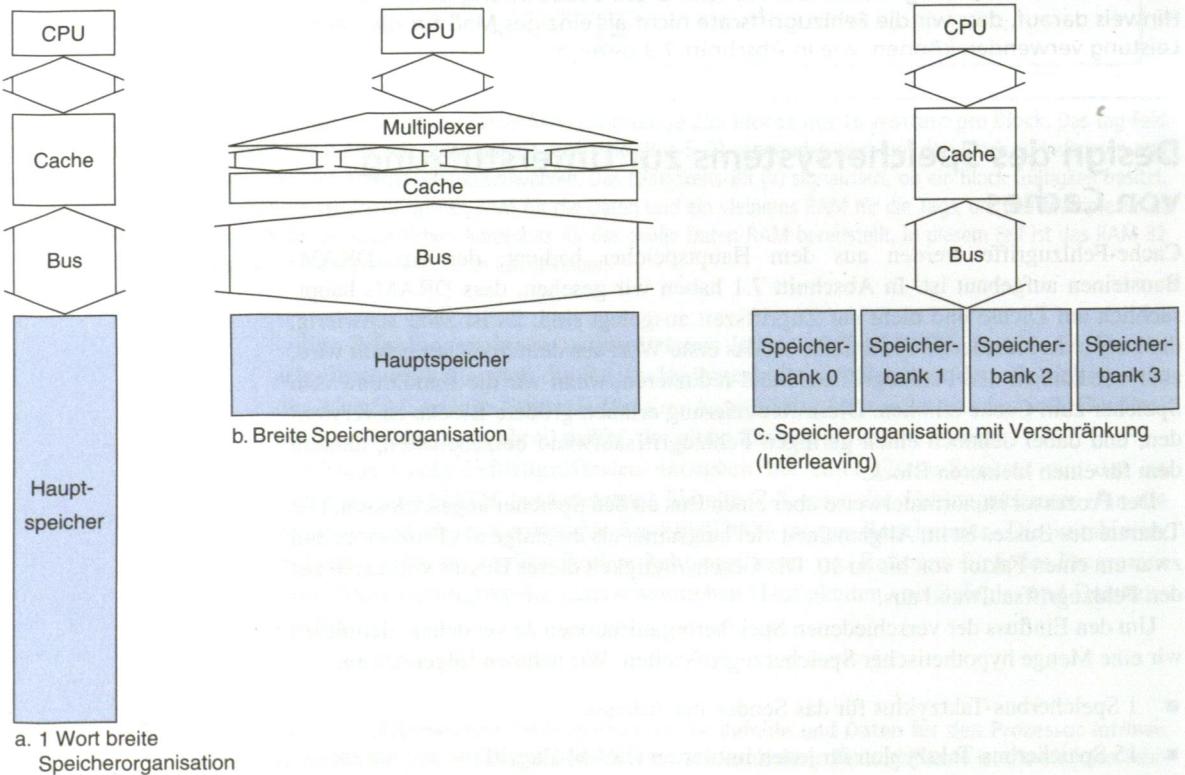


Abb. 7.9 Die primäre Methode, eine höhere Speicherbandbreite zu erzielen, ist die Steigerung der physischen oder logischen Breite des Speichersystems. In dieser Abbildung wird die Speicherbandbreite auf zwei Arten erhöht. Das einfachste Design, (a), verwendet einen Speicher, bei dem alle Komponenten ein Wort breit sind; (b) zeigt einen breiteren Speicher, Bus und Cache; während (c) einen schmäleren Bus und Cache mit einem verschrankten Speicher (*Interleaving*) zeigt. In (b) besteht die Logik zwischen dem Cache und dem Prozessor aus einem Multiplexer, der für Leseoperationen verwendet wird, und einer Steuerungslogik, die bei Schreiboperationen die entsprechenden Wörter im Cache aktualisiert.

(fast das Vierfache). Der wesentliche Aufwand für diese Verbesserung sind der breitere Bus sowie der mögliche Anstieg der Zugriffszeit aufgrund der Multiplexer- und Steuerungslogik zwischen Prozessor und Cache.

Statt den gesamten Pfad zwischen dem Speicher und dem Cache zu verbreitern, können die Speicherchips in Bänken angeordnet werden, um mehrere Wörter statt jeweils nur ein einziges Wort innerhalb einer Zugriffszeit zu lesen oder zu schreiben. Jede Bank könnte ein Wort breit sein, so dass die Breite des Busses und des Caches sich nicht ändern müssen, wobei das Senden einer Adresse an mehrere Bänke diesen das gleichzeitige Lesen erlaubt. Dieses Schema, das als *Interleaving* (Verschränkung) bezeichnet wird, behält den Vorteil bei, dass die vollständige Speicherlatenz nur ein einziges Mal anfällt. Mit vier Bänken beispielsweise besteht die Zeit, einen 4-Wort-Block zu laden, aus 1 Zyklus für die Übertragung der Adresse und der Leseanforderung an die Bänke, 15 Zyklen für den Zugriff auf den Speicher aller vier Bänke, und 4 Zyklen, um die 4 Wörter zurück an den Cache zu senden. Das ergibt einen Fehlzugriffsaufwand von $1 + 1 \times 15 + 4 \times 1 = 20$ Speicherbus-Taktzyklen. Das ist eine effektive Bandbreite pro Fehlzugriff von 0,80 Byte pro Takt, oder etwa die dreifache Bandbreite gegenüber dem 1 Wort breiten Speicher und Bus. Auch für Schreiboperationen sind Bänke sehr praktisch. Jede Bank kann unabhängig voneinander schreiben, wodurch die Schreibbandbreite vervierfacht wird und weniger Stillstände in einem Durchschreibe-Cache stattfinden. Wie wir sehen werden, macht eine alternative Strategie für Schreiboperationen die Verschränkung noch attraktiver.

Vertiefung: Speicherchips sind so aufgebaut, dass sie eine bestimmte Anzahl von Ausgabebits erzeugen, normalerweise 4 bis 32, wobei im Jahr 2004 8 oder 16 Bits am gebräuchlichsten waren. Wir beschreiben den Aufbau eines RAM als $d \times w$, wobei d die Anzahl der adressierbaren Positionen (die Tiefe) ist und w die Ausgabe (oder Breite jeder Position). Eine Möglichkeit zur Verbesserung der Übertragungsgeschwindigkeit der Daten vom Speicher in die Caches ist die Ausnutzung der DRAM-Struktur. DRAMs sind logisch als rechteckige Felder aufgebaut, und die Zugriffszeit setzt sich aus der Zeilen- und der Spaltenzugriffszeit zusammen. DRAMs puffern eine Bitzeile innerhalb des DRAM für den Spaltenzugriff. Außerdem sind sie mit optionalen Timing-Signalen ausgerüstet, die wiederholte Zugriffe auf den Bitzeilenpuffer erlauben, ohne dass ein erneuter Zeilenzugriff nötig ist. Diese Methode, ursprünglich als *Seitenmodus* (*Page mode*) bezeichnet, hat eine Reihe von Verbesserungen durchlaufen. Im Seitenmodus verhält sich der Puffer wie ein SRAM. Durch die Änderung der Spaltenadresse kann bis zum nächsten Zeilenzugriff ein Zugriff auf beliebige Bits erfolgen. Diese Fähigkeit ändert die Zugriffszeit ganz wesentlich, weil die Zugriffszeit auf Bits in der Zeile sehr viel geringer ist. Tabelle 7.3 zeigt, wie sich Dichte, Kosten und Zugriffszeit von DRAMs über die Jahre verändert haben.



Die neueste Entwicklung sind DDR SDRAMs (Double Data Rate Synchronous DRAMs, Synchrone DRAMs mit doppelter Datengeschwindigkeit). SDRAMs unterstützen einen Bündelzugriff (*Burst Access*) auf Daten, die an sequenziell hintereinander stehenden Positionen im DRAM angeordnet sind. Ein SDRAM gibt eine Startadresse und eine Bündellänge vor. Die Daten im Bündel werden von einem Takt signal gesteuert übertragen, das im Jahr 2004 bis zu 300 MHz schnell war. Die beiden wichtigsten Vorteile von SDRAMs sind die Verwendung eines Takts, so dass eine Synchronisierung überflüssig wird, sowie der Wegfall der Notwendigkeit, aufeinander folgende Adressen im Bündel bereitzustellen. Der DDR-Teil des Namens bedeutet, dass Datenübertragungen sowohl während der ansteigenden als auch der abfallenden Flanke des Takts stattfinden, womit man doppelt soviel Bandbreite erhält, als man basierend auf Taktgeschwindigkeit und Datenbreite erwarten würde. Um eine solch hohe Bandbreite bieten zu können, ist das interne DRAM in Form von verschrankten Speicherbänken aufgebaut.

Der Vorteil dieser Optimierungen ist, dass sie die Schaltungen verwenden, die in den DRAMs größtenteils schon enthalten sind, wodurch geringe Kosten für das System entstehen, während wesentliche Verbesserungen der Bandbreite möglich



sind. Die interne Architektur von DRAMs und die Implementierung dieser Optimierungen sind auf CD in Abschnitt B.9 von **Appendix B** beschrieben.

Zusammenfassung

Wir haben den vorigen Abschnitt mit einer Betrachtung der einfachsten Form eines Caches begonnen: einem direkt abgebildeten Cache mit einem 1-Wort-Block. In einem solchen Cache sind sowohl Treffer als auch Fehlzugriffe einfach zu verarbeiten, weil ein Wort an genau einer Position stehen kann, und es für jedes Wort ein separates Tag gibt. Um Cache und Speicher konsistent zu halten, kann ein Durchschreibeschema verwendet werden, so dass jede Schreiboperation in den Cache auch eine Aktualisierung des Speichers bewirkt. Die Alternative zum Durchschreiben ist ein Rückschreibeschema, das einen Block zurück in den Speicher kopiert, sobald er ersetzt wird; dieses Schema werden wir in nachfolgenden Abschnitten noch genauer beschreiben.

Um die räumliche Lokalität zu nutzen, muss ein Cache eine Blockgröße größer einem Wort haben. Die Verwendung eines größeren Blocks senkt die Fehlzugriffsrate und verbessert die Effizienz des Caches, indem die Menge der Tag-Speicherungen im Verhältnis der Menge der Datenspeicherungen im Cache reduziert wird. Obwohl ein größerer Block die Fehlzugriffsrate senkt, kann sich der Fehlzugriffsaufwand erhöhen. Wenn der Fehlzugriffsaufwand linear mit der Blockgröße steigt, können größere Blöcke schnell zu einer schlechteren Leistung führen. Um dies zu vermeiden, wird die Bandbreite des Hauptspeichers erhöht, um Cache-Blöcke schneller übertragen zu können. Die beiden gebräuchlichsten Methoden dafür sind Speicherverbreiterung und -verschränkung. In beiden Fällen reduzieren wir die Zeit für das Laden des Blocks, indem wir die Anzahl der Speicherzugriffe minimieren, die benötigt werden, um einen Block zu laden. Mit einem breiteren Bus können wir auch die Zeit senken, die man braucht, um den Block vom Speicher in den Cache zu übertragen.

Tab. 7.3 Die DRAM-Größen sind bis 1996 alle drei Jahre um den Faktor 4 gestiegen, danach haben sie sich alle zwei Jahre etwa verdoppelt. Die Verbesserungen der Zugriffszeiten erfolgten langsamer, aber stetig. Die Kosten gehen eng mit den Dichteverbesserungen einher, obwohl die Kosten häufig auch durch andere Faktoren beeinflusst werden, wie etwa Verfügbarkeit oder Nachfrage. Die Kosten pro Megabyte wurde nicht inflationsbereinigt.

Jahr der Einführung	Chip-Größe	\$ pro MB	Gesamtzugriffszeit auf eine neue Zeile/Spalte	Spaltenzugriffszeit auf eine vorhandene Zeile
1980	64 Kbit	\$1500	250 ns	150 ns
1983	256 Kbit	\$500	185 ns	100 ns
1985	1 Mbit	\$200	135 ns	40 ns
1989	4 Mbit	\$50	110 ns	40 ns
1992	16 Mbit	\$15	90 ns	30 ns
1996	64 Mbit	\$10	60 ns	12 ns
1998	128 Mbit	\$4	60 ns	10 ns
2000	256 Mbit	\$1	55 ns	7 ns
2002	512 Mbit	\$0,25	50 ns	5 ns
2004	1024 Mbit	\$0,10	45 ns	3 ns

Die Geschwindigkeit des Speichersystems wirkt sich auf die Entscheidung des Designers aus, welche Größe der Cache-Block haben soll. Welche der folgenden Richtlinien für Cache-Designer sind allgemein gültig?



1. Je kürzer die Speicherlatenz, desto kleiner der Cache-Block.
2. Je kürzer die Speicherlatenz, desto größer der Cache-Block.
3. Je größer die Speicherbandbreite, desto kleiner der Cache-Block.
4. Je größer die Speicherbandbreite, desto größer der Cache-Block.

7.3

Cache-Leistung messen und verbessern

In diesem Abschnitt betrachten wir zunächst, wie man die Cache-Leistung messen und analysieren kann. Anschließend untersuchen wir zwei unterschiedliche Techniken zur Verbesserung der Cache-Leistung. Die erste Technik konzentriert sich darauf, die Fehlzugriffsrate zu reduzieren, indem sie die Wahrscheinlichkeit reduziert, dass zwei unterschiedliche Speicherblöcke um dieselbe Cache-Position konkurrieren. Die zweite Technik reduziert den Fehlzugriffsaufwand, indem sie der Hierarchie eine zusätzliche Ebene hinzufügt. Diese Technik, auch als *Cache-Speicherhierarchie* bezeichnet, wurde 1990 zunächst in Rechnern für über \$100 000 eingeführt; heute ist sie bereits in den Mikroprozessoren von Desktop-Computern für weniger als \$1000 vorhanden!

Die CPU-Zeit kann in die Taktzyklen unterteilt werden, während derer die CPU das Programm ausführt, und die Taktzyklen, die die CPU damit verbringt, auf das Speichersystem zu warten. Normalerweise gehen wir davon aus, dass die Kosten für Cache-Treffer Teil der normalen CPU-Ausführungszyklen sind. Damit gilt,

$$\text{CPU-Zeit} = (\text{CPU-Ausführungstaktzyklen} + \text{Speicherstillstands-Taktzyklen}) \\ \times \text{Taktzykluszeit}$$

Die Speicherstillstands-Taktzyklen (memory-stall clock cycles) stammen hauptsächlich aus Cache-Fehlzugriffen. Außerdem beschränken wir die Diskussion auf ein vereinfachtes Modell des Speichersystems. In realen Prozessoren können die durch Lese- und Schreiboperationen verursachten Stillstände relativ komplex sein, und eine genaue Leistungsvorhersage bedingt im Allgemeinen sehr detaillierte Simulationen des Prozessors und des Speichersystems.

Speicherstillstands-Taktzyklen können definiert werden als die Summe der Stillstandszyklen aus Leseoperationen plus derjenigen aus Schreiboperationen:

$$\text{Speicherstillstands-Taktzyklen} = \text{Lesestillstands-Zyklen} + \text{Schreibstillstands-Zyklen}$$

Die Lesestillstands-Zyklen können definiert werden durch die Anzahl der Lesezugriffe pro Programm, den Fehlzugriffsaufwand in Taktzyklen für eine Leseoperation und die Lese-Fehlzugriffsraten:

$$\text{Lesestillstands-Zyklen} = \frac{\text{Leseoperationen}}{\text{Programm}} \times \text{Lese-Fehlzugriffsraten} \\ \times \text{Lese-Fehlzugriffsaufwand}$$

Schreiboperationen sind komplizierter. Für ein Durchschreibeschema haben wir zwei Ursachen für Stillstände: Schreib-Fehlzugriffe, für die es normalerweise erforderlich ist, dass der Block geladen wird, bevor die Schreiboperation fortgesetzt wird (weitere Informationen über die Verarbeitung von Schreiboperationen finden Sie im Abschnitt Vertiefung auf Seite 394), und Schreibpuffer-Stillstände, die auftreten, wenn eine Schreiboperation stattfindet, aber der Schreibpuffer voll ist. Die Stillstandzyklen für Schreiboperationen sind also gleich der Summe dieser beiden:

Schreib-Stillstandszyklen

$$= \left(\frac{\text{Schreiboperationen}}{\text{Programm}} \times \text{Schreib-Fehlzugriffsrate} \times \text{Schreib-Fehlzugriffsaufwand} \right) + \text{Schreibpuffer-Stillstände}$$

Weil die Schreibpuffer-Stillstände vom zeitlichen Auftreten der Schreiboperationen und nicht nur von deren Häufigkeit abhängig sind, ist es nicht möglich, solche Stillstände mithilfe einer einfachen Gleichung zu berechnen. In einem System mit ausreichender Schreibpuffertiefe (d.h. vier oder mehr Wörter) und einem Speicher, der Schreiboperationen in einer Geschwindigkeit akzeptieren kann, die die durchschnittliche Schreibfrequenz in Programmen wesentlich übersteigt (z.B. um einen Faktor von 2), sind glücklicherweise die Schreibpuffer-Stillstände selten, und wir können sie problemlos ignorieren. Wenn ein System diese Kriterien nicht erfüllt, wäre es nicht gut entworfen. Der Designer hätte stattdessen einen tieferen Schreibpuffer oder eine Rückschreibtechnik verwenden sollen.

Rückschreibbeschemata haben auch potenzielle zusätzliche Stillstände, die aus der Notwendigkeit entstehen, einen Cache-Block in den Speicher zurückzuschreiben, wenn der Block ersetzt wird. Wir werden in Abschnitt 7.5 genauer darauf eingehen.

In den meisten Caches mit Durchschreibetechnik ist der Fehlzugriffsaufwand für Lese- und Schreiboperationen gleich (die Zeit, um den Block aus dem Speicher zu laden). Wenn wir davon ausgehen, dass die Schreibpuffer-Stillstände zu vernachlässigen sind, können wir die Lese- und Schreiboperationen unter Verwendung einer einzigen Fehlzugriffsrate und eines einzigen Fehlzugriffsaufwands zusammenfassen:

$$\text{Speicherstillstands-Taktzyklen} = \frac{\text{Speicherzugriffe}}{\text{Programm}} \times \text{Fehlzugriffsrate} \times \text{Fehlzugriffsaufwand}$$

Dies können wir auch umschreiben in

$$\text{Speicherstillstands-Taktzyklen} = \frac{\text{Befehle}}{\text{Programm}} \times \frac{\text{Fehlzugriffe}}{\text{Befehl}} \times \text{Fehlzugriffsaufwand}$$

Wir betrachten ein einfaches Beispiel, anhand dessen wir den Einfluss der Cache-Leistung auf die Prozessorleistung besser erkennen können.

BEISPIEL

Berechnung der Cache-Leistung

Wir gehen von einer Fehlzugriffsrate des Befehls-Caches für ein Programm von 2 % und einer Fehlzugriffsrate des Daten-Caches von 4 % aus. Ein Prozessor hat einen CPI von 2 ohne Speicherstillstände, und der Fehlzugriffsaufwand beträgt für alle Fehlzugriffe 100 Zyklen. Berechnen Sie, um wie viel schneller ein Prozessor mit einem perfekten Cache ohne Fehlzugriffe laufen würde. Verwenden Sie die Befehshäufigkeiten für SPECint2000 aus Kapitel 3, Tabelle 3.14 auf Seite 194.

Die Anzahl der Speicherfehlzugriffs-Zyklen für Befehle in Hinblick auf die Gesamtzahl ausführter Befehle (I) beträgt

ANTWORT

$$\text{Befehlsfehlzugriffs-Zyklen} = I \times 2\% \times 100 = 2,00 \times I$$

Die Häufigkeit aller Lade- und Speicheroperationen in den SPECint2000-Benchmarks beträgt 36 %. Damit können wir die Anzahl der Speicherfehlzugriffs-Zyklen für Datenzugriffe ermitteln:

$$\text{Datenfehlzugriffs-Zyklen} = I \times 36\% \times 4\% \times 100 = 1,44 \times I$$

Die Gesamtzahl der Speicherstillstands-Zyklen beträgt $2,00 \times I + 1,44 \times I = 3,44 \times I$. Das sind mehr als 3 Zyklen Speicherstillstand pro Befehl. Der CPI mit Speicherstillständen beträgt also $2 + 3,44 = 5,44$. Weil es keine Änderung im Befehlszähler oder in der Taktgeschwindigkeit gibt, ist das Verhältnis der CPU-Ausführungszeiten gleich

$$\begin{aligned} \frac{\text{CPU-Zeit mit Stillständen}}{\text{CPU-Zeit mit perfektem Cache}} &= \frac{I \times \text{CPI}_{\text{Stillstand}} \times \text{Taktzyklus}}{I \times \text{CPI}_{\text{perfekt}} \times \text{Taktzyklus}} \\ &= \frac{\text{CPI}_{\text{Stillstand}}}{\text{CPI}_{\text{perfekt}}} = \frac{5,44}{2} = 2,72 \end{aligned}$$

Die Leistung mit perfektem Cache wäre um den Faktor 2,72 besser.

Was passiert, wenn der Prozessor schneller getaktet wird, nicht aber das Speichersystem? Die mit Speicherstillständen verbrachte Zeit nimmt einen wachsenden Anteil der Ausführungszeit ein. Das Gesetz von Amdahl, das wir in Kapitel 4 vorgestellt haben, erinnert uns an diese Tatsache. Einige einfache Beispiele zeigen, wie ernsthaft dieses Problem sein kann. Angenommen, wir beschleunigen den Computer aus dem vorigen Beispiel, indem wir seinen CPI von 2 auf 1 reduzieren, ohne die Taktgeschwindigkeit zu ändern, beispielsweise durch eine verbesserte Pipeline. Das System mit Cache-Fehlzugriffen hätte damit einen CPI von $1 + 3,44 = 4,44$, und das System mit dem perfekten Cache wäre

$$\frac{4,44}{1} = 4,44 \text{ mal schneller.}$$

Die Ausführungszeit, die für Speicherstillstände aufgewendet wird, wäre von

$$\frac{3,44}{5,44} = 63\%$$

auf

$$\frac{3,44}{4,44} = 77\%$$

gestiegen. Analog dazu steigert eine höhere Taktgeschwindigkeit ohne Änderung des Speichersystems auch den Leistungsverlust aufgrund von Cache-Fehlzugriffen, wie das nächste Beispiel verdeutlicht.

Cache-Leistung bei gesteigerter Taktgeschwindigkeit

BEISPIEL

Angenommen, wir erhöhen die Leistung des Computers aus dem vorigen Beispiel, indem wir seine Taktgeschwindigkeit verdoppeln. Weil sich die Geschwindigkeit des Hauptspeichers sehr wahrscheinlich nicht ändern wird, nehmen wir an, dass

ANTWORT

sich die absolute Zeit für die Verarbeitung eines Cache-Fehlzugriffs nicht ändert. Um wie viel schneller ist der Computer mit dem schnelleren Takt, wenn wir von derselben Fehlzugriffssrate wie im vorigen Beispiel ausgehen?

In den schnelleren Taktzyklen gemessen, beträgt der neue Fehlzugriffsaufwand doppelt so viele Taktzyklen, d.h. 200 Taktzyklen. Damit gilt:

$$\text{Gesamt-Fehlzugriffs-Zyklen pro Befehl} = (2\% \times 200) + 36\% \times (4\% \times 200) = 6,88$$

Der schnellere Computer mit Cache-Fehlzugriffen hat also einen CPI von $2 + 6,88 = 8,88$, im Vergleich zu einem CPI mit Cache-Fehlzugriffen von 5,44 für den langsameren Computer.

Unter Verwendung der Formel für die CPU-Zeit aus dem vorigen Beispiel können wir die relative Leistung wie folgt berechnen:

$$\begin{aligned} \frac{\text{Leistung mit schnellem Takt}}{\text{Leistung mit langsamem Takt}} &= \frac{\text{Ausführungszeit mit langsamem Takt}}{\text{Ausführungszeit mit schnellem Takt}} \\ &= \frac{IC \times CPI_{\text{langsam}} \times \text{Taktzyklus}}{IC \times CPI_{\text{schnell}} \times \frac{\text{Taktzyklus}}{2}} = \frac{5,44}{8,88 \times \frac{1}{2}} = 1,23 \end{aligned}$$

Der Computer mit dem schnelleren Takt ist also nur etwa 1,2-mal schneller statt doppelt so schnell, was er gewesen wäre, wenn wir die Cache-Fehlzugriffe hätten ignorieren können.

Wie diese Beispiele zeigen, steigt der relative Cache-Aufwand, wenn ein Prozessor schneller wird. Verbessert ein Prozessor darüber hinaus sowohl die Taktgeschwindigkeit als auch den CPI, erleidet er eine doppelte Einbuße:

1. Je kleiner der CPI, desto schwerwiegender ist der Einfluss von Stillstandszyklen.
2. Das Hauptspeichersystem kann voraussichtlich nicht so schnell wie die Prozessorykluszeit verbessert werden, hauptsächlich deshalb, weil die Zugriffsgeschwindigkeit des zugrunde liegenden DRAM-Speichers nicht viel schneller wird. Bei Berechnung des CPI wird der Cache-Fehlzugriffsaufwand in Prozessortaktzyklen gemessen, die für einen Fehlzugriff anfallen. Wenn die Hauptspeicher von zwei Prozessoren dieselben absoluten Zugriffszeiten haben, führt eine höhere Prozessortaktrate zu einem höheren Fehlzugriffsaufwand.

Die Bedeutung der Cache-Leistung für Prozessoren mit geringem CPI und hohen Taktgeschwindigkeiten ist also größer und demzufolge ist die Gefahr ebenfalls größer, das Cache-Verhalten bei der Leistungseinschätzung solcher Prozessoren zu vernachlässigen. Wie wir in Abschnitt 7.6 sehen werden, hat die Verwendung von schnellen Pipeline-gestützten Prozessoren in PCs und Workstations zu der Verwendung komplexer Cache-Systeme geführt, selbst in Computern, die für weniger als \$1000 angeboten werden.

Die zuvor gezeigten Beispiele und Gleichungen gehen davon aus, dass die Trefferzeit bei der Ermittlung der Cache-Leistung vernachlässigbar sei. Wenn die Trefferzeit steigt, steigt auch die Gesamtzeit für den Zugriff auf ein Wort aus dem Speichersystem. Wir werden später noch mehr Beispiele dafür sehen, was die Zugriffszeit erhöhen kann. Zunächst jedoch soll eine Vergrößerung des Caches betrachtet werden. Ein großer Cache könnte eine längere Zugriffszeit haben, so als wäre Ihr Schreibtisch in der Bibliothek sehr groß (z.B. 3 m²), so dass es länger dauert, bis Sie ein Buch gefunden haben. Wenn Pipelines mehr als fünf Stufen verwenden, fügt eine langsamere

Trefferzeit wahrscheinlich eine weitere Stufe in die Pipeline ein, weil mehrere Zyklen für einen Cache-Treffer erforderlich sind. Obwohl es komplizierter ist, die Leistung einer tieferen Pipeline zu berechnen, könnte die langsamere Trefferzeit eines großen Caches die Verbesserung der Trefferrate dominieren, was zu einer Verschlechterung der Prozessorleistung führen würde.

Der nächste Unterabschnitt beschreibt alternative Cache-Organisationen, die die Fehlzugriffsraten senken, aber zum Teil die Trefferzeit erhöhen. Weitere Beispiele finden Sie im Abschnitt 7.7.

Reduzierung von Cache-Fehlzugriffen durch eine flexiblere Platzierung von Blöcken

Wenn wir bisher einen Block im Cache platziert haben, haben wir ein einfaches Platzierungsschema verwendet: Ein Block kann an genau einer Stelle im Cache abgelegt werden. Wie bereits erwähnt, spricht man von *direkt abgebildet*, weil jede Blockadresse im Speicher auf eine einzige Position in der oberen Ebene der Hierarchie abgebildet wird. Es gibt zahlreiche Schemata für die Platzierung von Blöcken. Das eine Extrem ist die direkte Abbildung, wobei ein Block an genau einer Position platziert werden kann.

Das andere Extrem ist ein Schema, bei dem ein Block an *jeder* beliebigen Position im Cache platziert werden kann. Ein solches Schema wird als **vollassoziativ** bezeichnet, weil ein Block im Speicher jedem beliebigen Eintrag im Cache zugeordnet werden kann. Um einen bestimmten Block in einem vollassoziativen Cache zu finden, müssen alle Einträge im Cache durchsucht werden. Um die Suche durchführbar zu machen, erfolgt sie parallel mit je einem Vergleicher pro Cache-Eintrag. Diese Vergleicher erhöhen die Hardwarekosten wesentlich, so dass eine vollassoziative Cache-Organisation nur für Caches mit wenigen Blöcken sinnvoll ist.

Zwischen direkt abgebildeten und vollassoziativen Caches gibt es die Organisationsform des **satzassoziativen Caches**. In einem satzassoziativen Cache gibt es eine feste Anzahl von Speicherplätzen (mindestens 2), auf die ein Block gespeichert werden kann. Ein satzassoziativer Cache mit n Positionen für einen Block wird als n -fach satzassoziativer Cache bezeichnet. Ein n -fach satzassoziativer Cache besteht aus einer Menge von Sätzen, die aus jeweils n Blöcken bestehen. Jeder Block im Speicher wird auf einen eindeutigen *Satz* im Cache abgebildet, der durch das Indexfeld bestimmt ist, und ein Block kann in *jedem beliebigen* Element dieses Satzes platziert werden. Eine satzassoziative Platzierung kombiniert also direkt abgebildete Platzierung und vollassoziative Platzierung: Ein Block wird direkt auf einen Satz abgebildet, und dann werden alle Blöcke in dem Satz auf Übereinstimmung durchsucht.

Beachten Sie, dass bei einem direkt abgebildeten Cache die Position eines Speicherplatzes wie folgt festgelegt ist:

$$(\text{Blocknummer}) \bmod (\text{Anzahl der Cache-Blöcke})$$

In einem satzassoziativen Cache ist der Satz, der einen Speicherblock enthält, festgelegt durch

$$(\text{Blocknummer}) \bmod (\text{Anzahl der Sätze im Cache}).$$

Weil der Block in jedem Element des Satzes platziert werden kann, müssen *alle Tags aller Elemente des Satzes* durchsucht werden. In einem vollassoziativen Cache kann der Block überall stehen, und *alle Tags aller Blöcke im Cache* müssen durchsucht werden. Beispielsweise zeigt Abbildung 7.10, wo Block 12 gemäß der Blockplatzierungsstrategie für direkt abgebildete, zweifach satzassoziative und vollassoziative Caches in einem Cache mit insgesamt 8 Blöcken platziert wird.

Vollassoziativer Cache (fully associative cache) Eine Cache-Struktur, wobei ein Block an jeder beliebigen Position im Cache platziert werden kann.

Satzassoziativer Cache (set-associative cache) Ein Cache, bei dem jeder Block auf eine feste Anzahl von mindestens zwei Plätzen gespeichert werden kann.

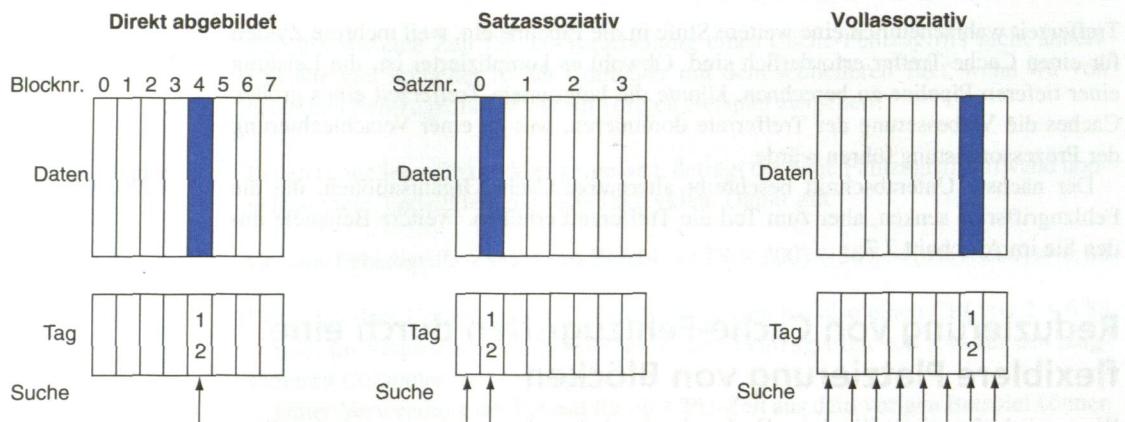


Abb. 7.10 Die Position eines Speicherblocks mit der Adresse 12 unterscheidet sich in einem Cache mit 8 Blöcken bei direkt abgebildeter, satzassoziativer und vollassoziativer Platzierung. Bei der direkt abgebildeten Platzierung gibt es nur einen Cache-Block, in dem Speicherblock 12 gefunden werden kann, und dieser Cache-Block ist angegeben durch $(12 \bmod 8) = 4$. In einem zweifach satzassoziativen Cache mit 8 Cache-Blöcken gibt es vier Sätze, und der Speicherblock 12 muss sich in Satz $(12 \bmod 4) = 0$ befinden; der Speicherblock kann sich in jedem Element des Sitzes befinden. Bei einer vollassoziativen Platzierung kann der Speicherblock mit der Blockadresse 12 in jedem der acht Cache-Blöcke erscheinen.

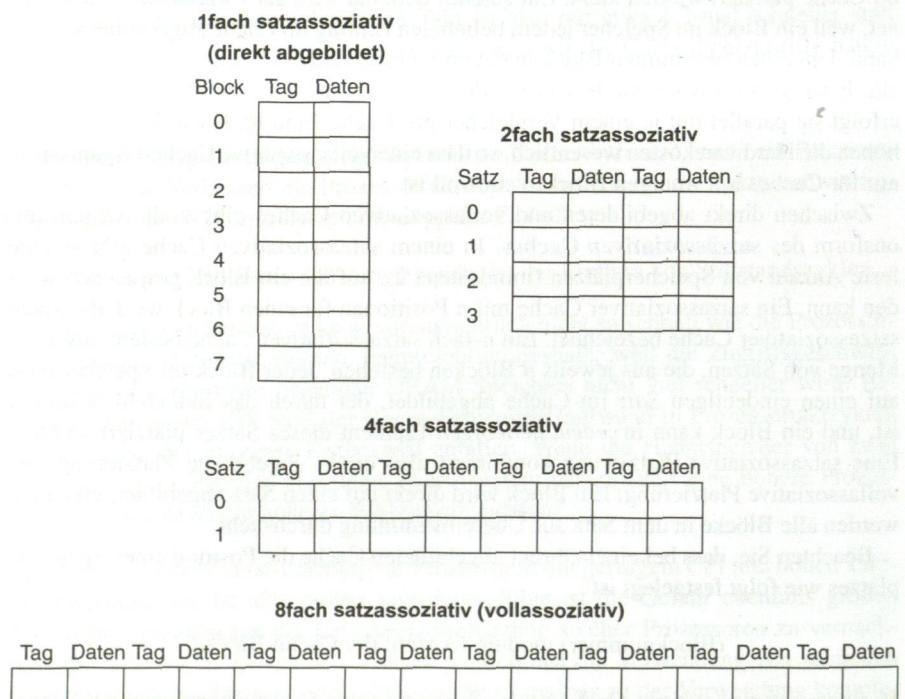


Abb. 7.11 Ein Cache mit 8 Blöcken, konfiguriert als direkt abgebildet, zweifach satzassoziativ, vierfach satzassoziativ und vollassoziativ. Die Gesamtgröße des Caches in Blöcken ist gleich der Anzahl der Sätze mal der Assoziativität. Für eine feste Cache-Größe verringert also eine Vergrößerung der Cache-Assoziativität die Anzahl der Sätze, während sie die Anzahl der Elemente pro Satz erhöht. Mit acht Blöcken ist ein achtfach satzassoziativer Cache dasselbe wie ein vollassoziativer Cache.

Wir können uns jede Blockplatzierungsstrategie als Variante der Satzassoziativität vorstellen. Abbildung 7.11 zeigt die möglichen Cache-Organisationsstrukturen für

einen 8 Blöcke großen Cache. Ein direkt abgebildeter Cache entspricht einem einfach satzassoziativen Cache: Jeder Cache-Eintrag enthält einen Block und jeder Satz besitzt ein Element. Ein vollassoziativer Cache der Größe m entspricht einem m -fach satzassoziativen Cache; er enthält einen Satz mit m Blöcken, und ein Eintrag kann sich in jedem Block innerhalb dieses Satzes befinden.

Vorteil einer erhöhten Assoziativität ist normalerweise eine Verringerung der Fehlzugriffsrate, wie das nächste Beispiel zeigt. Der wichtigste Nachteil, den wir gleich noch genauer betrachten werden, ist eine langsamere Trefferzeit.

Fehlzugriffe und Assoziativität in Caches

Angenommen, es gibt drei kleine Caches, die jeweils aus vier 1-Wort-Blöcken bestehen. Ein Cache ist vollassoziativ, ein zweiter ist zweifach satzassoziativ, und der dritte ist direkt abgebildet. Ermitteln Sie die Anzahl der Fehlzugriffe für jede Cache-Organisation für die Blockadressfolge 0, 8, 0, 6, 8.

Der direkt abgebildete Cache ist am einfachsten. Zuerst ermitteln wir, auf welchen Cache-Block jede Speicherblockadresse abgebildet wird:

Blockadresse	Cache-Block
0	(0 modulo 4) = 0
6	(6 modulo 4) = 2
8	(8 modulo 4) = 0

Jetzt können wir den Cache-Inhalt nach jedem Zugriff eintragen, wobei ein leerer Eintrag bedeutet, dass der Block ungültig ist. Farbiger Text steht für einen neuen Eintrag, der dem Cache für den zugehörigen Zugriff hinzugefügt wurde, und normaler Text steht für einen alten Eintrag im Cache. Es zeigt sich, dass der direkt abgebildete Cache bei jedem der fünf Zugriffe einen Fehlzugriff erzeugt:

Adresse des Speicherblocks, auf den zugegriffen wird	Treffer oder Fehlzugriff	Inhalt des Cache-Blocks nach dem Zugriff			
		0	1	2	3
0	Fehlzugriff	Speicher[0]			
8	Fehlzugriff	Speicher[8]			
0	Fehlzugriff	Speicher[0]			
6	Fehlzugriff	Speicher[0]		Speicher[6]	
8	Fehlzugriff	Speicher[8]		Speicher[6]	

Der satzassoziative Cache hat zwei Sätze (mit den Indizes 0 und 1) mit zwei Elementen pro Satz. Zuerst stellen wir fest, auf welchen Satz die Blockadressen abgebildet werden:

Blockadresse	Cache-Satz
0	(0 modulo 2) = 0
6	(6 modulo 2) = 0
8	(8 modulo 2) = 0

Weil wir die Wahl haben, welcher Eintrag in einem Satz bei einem Fehlzugriff ersetzt werden soll, brauchen wir eine Ersetzungsregel. Satzassoziative Caches er-

BEISPIEL

ANTWORT

setzen im Allgemeinen den Block, auf den am längsten nicht mehr zugegriffen wurde (least recently used), d.h. der Block wird ersetzt, dessen Zugriff am weitesten in der Vergangenheit liegt. (Wir werden später noch genauer auf die Ersetzungsregeln eingehen.) Unter Verwendung dieser Ersetzungsregel sieht der Inhalt des satzassoziativen Caches nach jedem Verweis wie folgt aus:

Adresse des Speicherblocks, auf den zugegriffen wird	Treffer oder Fehlzugriff	Inhalt des Cache-Blocks nach dem Zugriff			
		Satz 0	Satz 0	Satz 1	Satz 1
0	Fehlzugriff	Speicher[0]			
8	Fehlzugriff	Speicher[0]	Speicher[8]		
0	Treffer	Speicher[0]	Speicher[8]		
6	Fehlzugriff	Speicher[0]	Speicher[6]		
8	Fehlzugriff	Speicher[8]	Speicher[6]		

Beachten Sie, dass beim Zugriff auf Block 6 dieser den Block 8 ersetzt, weil auf Block 8 vor längerer Zeit als auf Block 0 zugegriffen wurde. Der zweifach satzassoziative Cache erzeugt vier Fehlzugriffe, einen weniger als der direkt abgebildete Cache.

Der vollassoziative Cache umfasst vier Cache-Blöcke (in einem einzigen Satz); jeder Speicherblock kann in jedem Cache-Block abgelegt werden. Der vollassoziative Cache zeigt mit nur drei Fehlzugriffen die beste Leistung:

Adresse des Speicherblocks, auf den zugegriffen wird	Treffer oder Fehlzugriff	Inhalt des Cache-Blocks nach dem Zugriff			
		Block 0	Block 1	Block 2	Block 3
0	Fehlzugriff	Speicher[0]			
8	Fehlzugriff	Speicher[0]	Speicher[8]		
0	Treffer	Speicher[0]	Speicher[8]		
6	Fehlzugriff	Speicher[0]	Speicher[8]	Speicher[6]	
8	Treffer	Speicher[8]	Speicher[8]	Speicher[6]	

Für diese Zugriffsfolge sind drei Fehlzugriffe das Beste, was wir erreichen können, weil auf drei unterschiedliche Blockadressen zugegriffen wird. Hätten wir acht Blöcke im Cache gehabt, hätte es im zweifach satzassoziativen Cache keine Ersetzungen gegeben (überprüfen Sie dies selbst!), und es hätte dieselbe Anzahl von Fehlzugriffen wie im vollassoziativen Cache stattgefunden. Hätten wir analog dazu 16 Blöcke gehabt, würden alle drei Caches dieselbe Anzahl an Fehlzugriffen aufweisen. Diese Änderung der Fehlzugriffssrate zeigt uns, dass Cache-Größe und Assoziativität für die Bestimmung der Cache-Leistung nicht unabhängig voneinander zu betrachten sind.

Um wie viel wird die Fehlzugriffssrate durch die Assoziativität reduziert? Tabelle 7.4 zeigt die Verbesserung für die SPEC2000-Benchmarks für einen 64 KB großen Daten-Cache mit 16-Wort-Blöcken und einer Assoziativität von direkter Abbildung bis hin zur achtfachen Assoziativität. Von der einfachen zur zweifachen Assoziativität sinkt die Fehlzugriffssrate um etwa 15 %, aber es gibt kaum weitere Verbesserungen auf dem Weg zu höheren Assoziativitäten.

Tab. 7.4 Die Fehlzugriffsraten bei einem Daten-Cache mit einem Aufbau wie etwa im Prozessor Intrinsity FastMATH für SPEC2000-Benchmarks mit einer ein- bis achtfachen Assoziativität. Diese Ergebnisse für 10 SPEC2000-Programme stammen aus Hennessy und Patterson [2003].

Assoziativität	Datenfehlzugriffsraten
1	10,3 %
2	8,6 %
4	8,3 %
8	8,1 %

Einen Block im Cache finden

Jetzt betrachten wir die Aufgabe, einen Block in einem satzassoziativen Cache zu finden. Wie in einem direkt abgebildeten Cache enthält jeder Block in einem satzassoziativen Cache ein Adress-Tag, das zusammen mit dem Index die Blockadresse angibt. Abbildung 7.12 zeigt, wie sich eine Speicheradresse zusammensetzt, auf die von dem Prozessor zugegriffen wird. Der Index wird verwendet, um den Satz auszuwählen, der die betreffende Adresse enthält, und die Tags aller Blöcke im Satz müssen daraufhin durchsucht werden, ob ein Tag dabei ist, das mit dem Tag-Teil der angelegten Adresse übereinstimmt. Der Block-Offset-Teil der Adresse selektiert das Wort innerhalb des Blocks. Weil die Geschwindigkeit eine wesentliche Rolle spielt, werden alle Tags im Satz parallel durchsucht. Wie bei einem vollassoziativen Cache würde eine sequenzielle Suche die Trefferzeit für einen satzassoziativen Cache zu langsam machen.

Wenn die Gesamtgröße des Caches gleich bleibt, erhöht eine Steigerung der Assoziativität die Anzahl der Blöcke pro Satz, d.h. die Anzahl der gleichzeitigen Vergleiche, die für eine parallele Suche erforderlich sind: Jede Steigerung um einen Faktor von 2 in der Assoziativität verdoppelt die Anzahl der Blöcke pro Satz und halbiert die Anzahl der Sätze. Analog dazu senkt jede Minderung der Assoziativität um einen Faktor von 2 die Größe des Index um 1 Bit und erhöht die Größe des Tags um 1 Bit. In einem vollassoziativen Cache gibt es effektiv nur einen Satz, und alle Blöcke müssen parallel überprüft werden. Es gibt also keinen Index, und die gesamte Adresse (ohne Block-Offset) wird mit dem Tag jedes Blocks verglichen. Mit anderen Worten, wir durchsuchen den gesamten Cache ohne Indizierung.

In einem direkt abgebildeten Cache, wie in Abbildung 7.6 gezeigt, braucht man nur einen einzigen Vergleicher, weil der Eintrag sich nur in einem Block befinden kann, und wir können einfach durch Indizierung auf den Cache zugreifen. Abbildung 7.13 zeigt, dass in einem vierfach satzassoziativen Cache vier Vergleicher benötigt werden, ebenso wie ein 4:1-Multiplexer, um aus den vier potenziellen Blöcken des ausgewählten Satzes den richtigen Block auszuwählen. Der Cache-Zugriff besteht in der Indizierung des entsprechenden Satzes und der anschließenden Durchsuchung der Tags dieses Satzes. Als Kosten eines assoziativen Caches fallen die zusätzlichen Vergleicher an, beziehungsweise etwaige Verzögerungen, die entstehen, weil die Vergleiche durchgeführt und eine Auswahl zwischen den Blöcken des Satzes getroffen werden müssen.

Tag	Index	Block-Offset
-----	-------	--------------

Abb. 7.12 Die drei Komponenten einer Speicheradresse, die an einen satzassoziativen oder direkt abgebildeten Cache angelegt wird. Der Index wird verwendet, um den Satz auszuwählen, und anschließend wird das Tag verwendet, um den Block durch Vergleich mit den Blöcken im ausgewählten Satz auszuwählen. Der Block-Offset ist die Adresse der gewünschten Daten in dem Block.

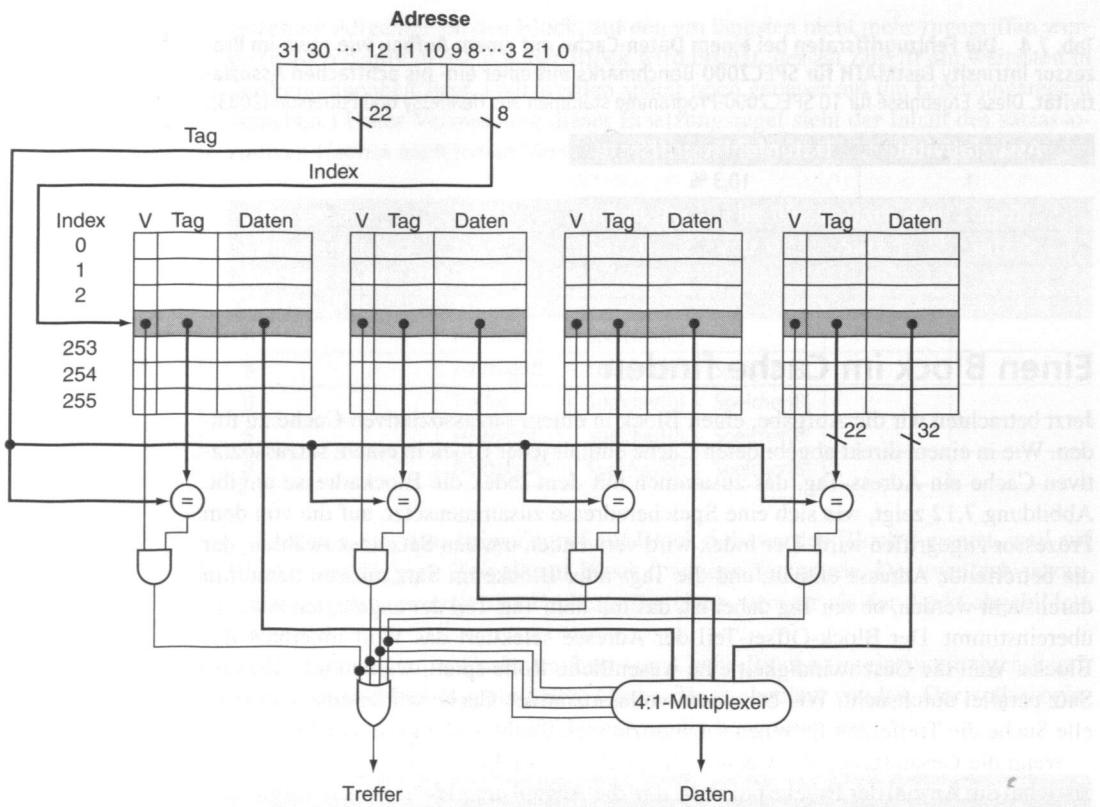


Abb. 7.13 Die Implementierung eines vierfach satzassoziativen Caches benötigt vier Vergleicher und einen 4:1-Multiplexer. Die Vergleicher stellen fest, welcher Block des ausgewählten Satzes mit dem Tag übereinstimmt (falls vorhanden). Anhand der Ausgabe des Vergleichers werden die Daten aus einem der vier Blöcke des indizierten Satzes ausgewählt, wofür ein Multiplexer mit decodiertem Auswahlsignal verwendet wird. In einigen Implementierungen können die Output-enable-Signale für den Datenteil der Cache-RAMs verwendet werden, um den Eintrag in dem Satz auszuwählen, der die Ausgabe veranlasst. Das Output-enable-Signal stammt von den Vergleichern, die das übereinstimmende Element zu einer Datenausgabe veranlassen. Dieser Aufbau macht den Multiplexer überflüssig. (V = valid kennzeichnet in der Abbildung die Gültigkeits-Bits.)

Ob in einer Speicherhierarchie ein direkt abgebildeter, satzassoziativer oder vollassoziativer Cache verwendet wird, ist von den Kosten für einen Fehlzugriff im Vergleich zu den Kosten für die Implementierung der Assoziativität sowohl in Hinblick auf die Zeit als auch auf die zusätzliche Hardware abhängig.

BEISPIEL

Größe der Tags versus Satzassoziativität

Eine erhöhte Assoziativität benötigt mehr Vergleicher und mehr Tag-Bits pro Cache-Block. Ermitteln Sie für einen Cache mit 4K-Blöcken, einer Blockgröße von 4 Wörtern und einer 32-Bit-Adresse die Gesamtzahl der Sätze sowie die Gesamtzahl der Tag-Bits für direkt abgebildete, zweifach und vierfach satzassoziative und vollassoziative Cache-Organisationen.

ANTWORT

Weil es $16 (= 2^4)$ Bytes pro Block gibt, müssen bei 32-Bit-Adressen $32 - 4 = 28$ Bits für Index und Tag verwendet werden. Der direkt abgebildete Cache hat dieselbe Anzahl an Sätzen wie Blöcke und damit 12 Bit breiten Index, weil $\log_2(4K) = 12$. Die Gesamtzahl der Tag-Bits ist also $(28 - 12) \times 4K = 16 \times 4K = 64$ KBit.

Jeder Assoziativitätsgrad senkt die Anzahl der Sätze um einen Faktor von 2 und damit die Anzahl der Bits für die Indizierung des Caches um 1. Andererseits erhöht sich die Anzahl der Bits im Tag um 1. Ein zweifach satzassoziativer Cache hat also 2K Sätze, und die Gesamtzahl der Tag-Bits beträgt $(28 - 11) \times 2 \times 2K = 34 \times 2K = 68$ KBit. Für einen vierfach satzassoziativen Cache ist die Anzahl der Sätze gleich 1K, und die Gesamtzahl der Tag-Bits beträgt $(28 - 10) \times 4 \times 1K = 72 \times 1K = 72$ KBit.

Für einen vollassoziativen Cache gibt es nur einen Satz mit 4K-Blöcken, und das Tag ist 28 Bit groß, was zu $28 \times 4K \times 1 = 112K$ Tag-Bits führt.

Auswahl, welcher Block ersetzt werden soll

Wenn in einem direkt abgebildeten Cache ein Fehlzugriff erfolgt, kann der angeforderte Block an genau einer Position abgelegt werden, und der Block, der diese Position belegt, muss ersetzt werden. In einem assoziativen Cache haben wir die Wahl, wo der angeforderte Block platziert wird und entsprechend welcher Block ersetzt werden soll. In einem vollassoziativen Cache kommen alle Blöcke für das Ersetzen in Frage. In einem satzassoziativen Cache müssen wir zwischen den Blöcken im betroffenen Satz auswählen.

Das am häufigsten verwendete Schema ist **LRU** (*least recently used*), das auch im vorigen Beispiel verwendet wurde. In einem LRU-Schema wird der Block ersetzt, auf den am längsten nicht mehr zugegriffen wurde. LRU wird implementiert, indem protokolliert wird, wann die einzelnen Blöcke in einem Satz im Vergleich zu den anderen Blöcken benutzt wurden. Für einen zweifach satzassoziativen Cache genügt ein einziges Bit in jedem Satz, das beim Zugriff auf einen Block gesetzt wird, um zu kennzeichnen, auf welchen der beiden Blöcke als letztes zugegriffen wurde. Mit steigender Assoziativität wird die Implementierung von LRU schwieriger. In Abschnitt 7.5 werden wir ein alternatives Ersetzungsschema vorstellen.

LRU (*least recently used*) Ein Ersetzungsschema, bei dem der Block ersetzt wird, auf den am längsten nicht mehr zugegriffen wurde.

Reduzierung des Fehlzugriffsaufwands durch Cache-Speicherhierarchien

Alle modernen Rechner nutzen Caches. Größtenteils werden diese Caches auf demselben Chip wie der Mikroprozessor implementiert. Um die Lücke zwischen den schnellen Taktraten moderner Prozessoren und der relativ langen Zugriffszeit auf DRAMs weiter zu schließen, unterstützen viele Mikroprozessoren eine zusätzliche Cache-Ebene. Dieser Cache auf zweiter Ebene, der sich auf demselben Chip oder außerhalb des Chips in einem separaten SRAM-Chip-Satz befinden kann, wird herangezogen, wenn im primären Cache ein Fehlzugriff erfolgt. Wenn dieser sekundäre Cache die gewünschten Daten enthält, ist der Fehlzugriffsaufwand für den primären Cache die Zugriffszeit des sekundären Caches, was sehr viel kleiner sein kann als die Zugriffszeit auf den Hauptspeicher. Enthalten weder der primäre noch der sekundäre Cache die Daten, ist ein Hauptspeicherzugriff erforderlich und es entsteht ein höherer Fehlzugriffsaufwand.

Aber wie signifikant ist die Leistungsverbesserung durch Verwendung eines sekundären Cache? Dies soll am nächsten Beispiel verdeutlicht werden.

Die Leistung von Cache-Speicherhierarchien

Angenommen, wir haben einen Prozessor mit einem Grund-CPI von 1,0, alle Zugriffe treffen im primären Cache und wir haben eine Taktgeschwindigkeit von 5 GHz.

BEISPIEL

Wir gehen von einer Hauptspeicherzugriffszeit von 100 ns aus, einschließlich der Fehlzugriffsverarbeitung. Angenommen, die Fehlzugriffssrate pro Befehl im primären Cache beträgt 2 %. Wie viel schneller wird der Prozessor, wenn wir einen zweiten Cache mit einer Zugriffszeit von 5 ns für Treffer und Fehlzugriff hinzufügen, der groß genug ist, um die Fehlzugriffssrate auf den Hauptspeicher auf 0,5 % zu reduzieren?

ANTWORT

Der Fehlzugriffsaufwand für den Hauptspeicher beträgt

$$\frac{100 \text{ ns}}{0,2 \frac{\text{ns}}{\text{Taktzyklus}}} = 500 \text{ Taktzyklen}$$

Der effektive CPI mit einer Cache-Ebene ist gegeben durch

$$\text{Gesamt-CPI} = \text{Grund-CPI} + \text{Speicherstillstandszyklen pro Befehl}$$

Für den Prozessor mit einer Cache-Ebene erhalten wir

$$\text{Gesamt-CPI} = 1,0 + \text{Speicherstillstandszyklen pro Befehl} = 1,0 + 2\% \times 500 = 11,0$$

Mit zwei Cache-Ebenen kann ein Fehlzugriff im primären Cache (dem Cache der ersten Ebene) entweder durch den sekundären Cache oder durch den Hauptspeicher bedient werden. Der Fehlzugriffsaufwand für einen Zugriff auf den Cache auf zweiter Ebene ist

$$\frac{5 \text{ ns}}{0,2 \frac{\text{ns}}{\text{Taktzyklus}}} = 25 \text{ Taktzyklen}$$

Wenn der Fehlzugriff im sekundären Cache bedient wird, ist dies der ganze Fehlzugriffsaufwand. Wenn der Fehlzugriff den Hauptspeicher braucht, ist der gesamte Fehlzugriffsaufwand die Summe aus den Zugriffszeiten auf den sekundären Cache sowie auf den Hauptspeicher.

Für einen Cache mit zwei Ebenen ist der Gesamt-CPI also die Summe der Stillstandszyklen aus beiden Cache-Ebenen sowie der grundlegende CPI:

$$\begin{aligned} \text{Gesamt-CPI} &= 1 + \text{Primäre Stillstände pro Befehl} \\ &\quad + \text{Sekundäre Stillstände pro Befehl} \\ &= 1 + 2\% \times 25 + 0,5\% \times 500 = 1 + 0,5 + 2,5 = 4,0 \end{aligned}$$

Der Prozessor mit dem sekundären Cache ist also um

$$\frac{11,0}{4,0} \approx 2,8$$

schneller.

Alternativ hätten wir die Stillstandszeiten auch berechnen können, indem wir die Summe der Stillstandszyklen berechnen, die im sekundären Cache treffen ($(2\% - 0,5\%) \times 25 = 0,4$), und der Zugriffe, die auf den Hauptspeicher weitergeleitet werden. Die letzteren umfassen die Kosten für den Zugriff auf den sekundären Cache sowie die Hauptspeicherzugriffszeit ($0,5\% \times (25 + 500) = 2,6$). Die Summe, $1,0 + 0,4 + 2,6$, ist ebenfalls 4,0.

Die Designüberlegungen für den primären und den sekundären Cache unterscheiden sich ganz wesentlich. Insbesondere erlaubt eine Cache-Struktur mit zwei Ebenen, dass sich der Entwurf des primären Caches auf die Minimierung der Trefferzeit konzentriert, um einen kürzeren Taktzyklus zu erzielen, während sich der Entwurf des sekundären

Caches auf die Fehlzugriffsrate konzentriert, um den Fehlzugriffsaufwand für lange Speicherzugriffszeiten zu reduzieren.

Diese Fokussierung auf unterschiedliche Aufgaben ist durch das Zusammenspiel der beiden Caches möglich. Der Fehlzugriffsaufwand des primären Caches wird durch das Vorhandensein des sekundären Caches wesentlich reduziert, so dass der primäre Cache kleiner sein kann und eine höhere Fehlzugriffsrate haben darf. Für den sekundären Cache wird die Zugriffszeit durch das Vorhandensein des primären Caches weniger wichtig, weil die Zugriffszeit des sekundären Caches zwar den Fehlzugriffsaufwand des primären Caches beeinflusst, jedoch nicht direkt die Trefferzeit des primären Caches oder die Prozessorzykluszeit.

Die Auswirkung dieser Änderungen auf die beiden Caches erkennt man durch einen Vergleich der einzelnen Caches mit dem optimalen Design eines primären Caches. Im Vergleich mit einem Cache mit einer einzigen Ebene ist der primäre Cache einer **Cache-Speicherhierarchie (multilevel cache)** häufig kleiner. Darüber hinaus verwendet der primäre Cache häufig eine kleinere Blockgröße, um der kleineren Cache-Größe gerecht zu werden, und weist einen reduzierten Fehlzugriffsaufwand auf. Im Vergleich dazu ist der sekundäre Cache häufig größer als ein Cache mit einer einzigen Ebene, weil die Zugriffszeit des sekundären Caches weniger kritisch ist. Abgesehen von einer größeren Gesamtgröße verwendet der sekundäre Cache häufig auch größere Blöcke als ein Cache mit einer einzigen Ebene.

Cache-Speicherhierarchie (multilevel cache) Eine Speicherhierarchie mit mehreren Cache-Ebenen (statt der Verwendung nur eines Caches und eines Hauptspeichers).

In Kapitel 2 haben wir gesehen, dass Quicksort einen algorithmischen Vorteil gegenüber Bubble-Sort hatte, der durch Sprache oder Compileroptimierung nicht ausgeglichen werden konnte. Abbildung 7.14(a) zeigt die Anzahl ausgeführter Befehle pro zu sortierendem Element für Radix-Sort bzw. Quicksort. Für sehr große Arrays hat Radix-Sort einen algorithmischen Vorteil gegenüber dem Quicksort in Hinblick auf die Anzahl der Operationen. Abbildung 7.14(b) zeigt die Zeit pro zu sortierendem Element statt ausgeführte Befehle pro Element. Wir sehen, dass die Linien mit demselben Verlauf beginnen wie in Abbildung 7.14(a), aber dann divergiert die Linie für den Radix-Sort mit steigender Anzahl der zu sortierenden Daten. Was ist hier passiert? Abbildung 7.14(c) beantwortet diese Frage, indem sie die Cache-Fehlzugriffe pro sortiertem Element betrachtet. Quicksort hat durchgängig weniger Fehlzugriffe pro zu sortierendem Element. Leider ignoriert die standardmäßige Algorithmenanalyse den Einfluss der Speicherhierarchie. Nachdem schnellere Taktgeschwindigkeiten und das Gesetz von Moore den Architekten erlauben, auch noch die letztmögliche Leistung aus einem Befehlsstrom herauszuholen, ist die Analyse der Speicherhierarchie unabdingbar für eine gute Verarbeitungsleistung. Wie wir in der Einleitung gesagt haben, ist ein Verständnis für das Verhalten der Speicherhierarchie wichtig, wenn man auf den heutigen Rechnern leistungsfähige Programme entwickeln will.

Grundlegendes zur Leistungsfähigkeit von Programmen



Vertiefung: Cache-Hierarchien führen zu verschiedenen Komplikationen. Als erstes gibt es jetzt unterschiedliche Arten von Fehlzugriffen und entsprechende Fehlzugriffsraten. In dem Beispiel auf Seite 407 haben wir die Fehlzugriffsrate des primären Caches gesehen, ebenso wie die **globale Fehlzugriffsrate** – den Anteil der Zugriffe, die in allen Cache-Ebenen einen Fehlzugriff erzeugt haben. Es gibt auch eine Fehlzugriffsrate für den sekundären Cache, nämlich das Verhältnis aller Fehlzugriffe im sekundären Cache dividiert durch die Anzahl aller Zugriffe. Diese Fehlzugriffsrate wird als die **lokale Fehlzugriffsrate** des sekundären Caches bezeichnet. Weil der primäre Cache die Zugriffe filtert, insbesondere diejenigen mit guter räumlicher und temporaler Lokalität, ist die lokale Fehlzugriffsrate des sekundären Caches sehr viel höher als die globale Fehlzugriffsrate. Für das Beispiel

Globale Fehlzugriffsrate (global miss rate) Der Anteil der Zugriffe, die in allen Ebenen einer Cache-Speicherhierarchie zu einem Fehlzugriff führen.

Lokale Fehlzugriffsrate (local miss rate) Der Anteil der Zugriffe auf eine Ebene eines Caches, die Fehlzugriffe verursachen. Wird in Hierarchien aus mehreren Ebenen verwendet.

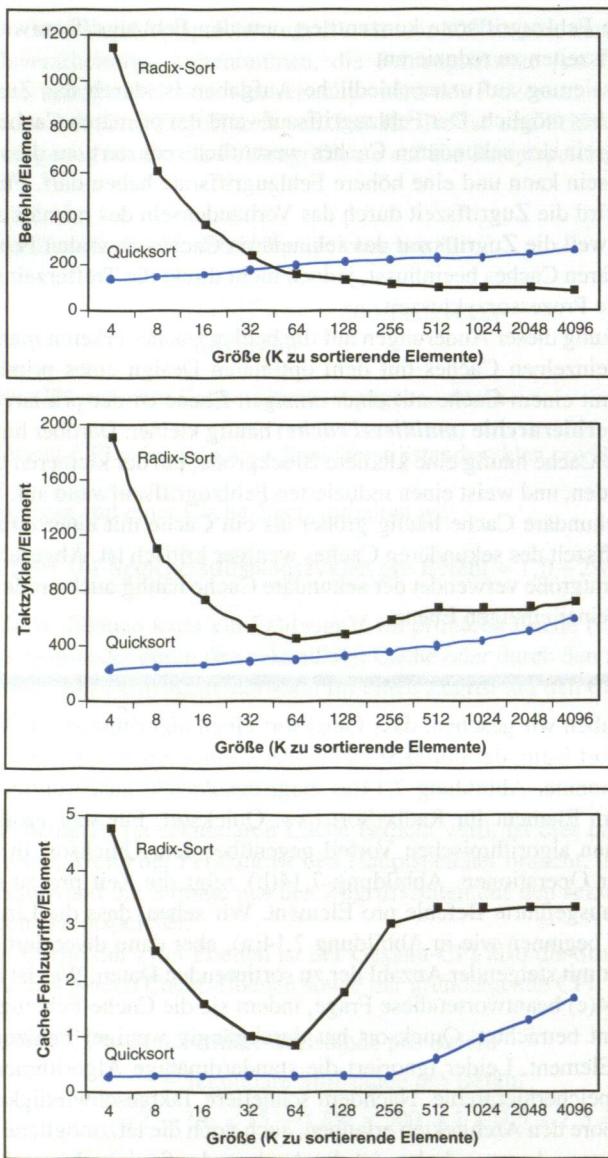


Abb. 7.14 Vergleich von Quicksort und Radix-Sort nach (a) pro sortiertem Element ausgeführten Befehlen, (b) Zeit pro sortiertem Element und (c) Cache-Fehlzugriffe pro sortiertem Element. Diese Daten stammen aus einer Arbeit von LaMarca und Ladner [1996]. Obwohl sich die Zahlen für neuere Rechner von den hier gezeigten sicher unterscheiden, gilt das Konzept weiterhin. Aufgrund solcher Ergebnisse wurden neue Versionen von Radix-Sort entwickelt, die die Speicherhierarchie berücksichtigen, um diese algorithmischen Vorteile nutzen zu können (siehe Abschnitt 7.7). Die grundlegende Idee der Cache-Optimierungen ist, alle Daten aus einem Block wiederholt zu nutzen, bevor der Block aufgrund eines Fehlzugriffs ersetzt wird.

auf Seite 407 können wir die lokale Fehlzugriffsraten berechnen als: $0,5\% / 2\% = 25\%$! Glücklicherweise bestimmt häufig die globale Fehlzugriffsraten, wie oft wir auf den Hauptspeicher zugreifen müssen.

Weitere Probleme entstehen, weil die Caches möglicherweise unterschiedliche Blockgrößen haben, um eine Übereinstimmung mit der größeren oder kleineren Cache-Gesamtgröße zu erzielen. Analog dazu kann sich die Assoziativität des Caches ändern. On-Chip-Caches werden häufig mit einer Assoziativität von vier

oder höher eingerichtet, während Off-Chip-Caches selten eine Assoziativität von mehr als 2 haben. On-Chip L1-Caches (d.h. Caches auf erster Ebene) haben im Allgemeinen eine niedrigere Assoziativität als On-Chip L2-Caches, weil eine schnelle Trefferzeit für L1-Caches wichtig ist. Diese Änderungen in der Blockgröße und Assoziativität führen zu Problemen bei der Modellierung der Caches, was häufig bedeutet, dass alle Ebenen in Kombination simuliert werden müssen, um das Verhalten zu verstehen.

Vertiefung: Bei Out-of-order-Prozessoren ist die Leistungabschätzung komplexer, weil sie Befehle während des Fehlzugriffsaufwands ausführen. Statt der Befehls- und der Daten-Fehlzugriffsrate verwenden wir Fehlzugriffe pro Befehl als Maß, und damit diese Formel:

$$\frac{\text{Speicherstillstandzyklen}}{\text{Befehl}} = \frac{\text{Fehlzugriffe}}{\text{Befehl}} \times (\text{Gesamtfehlzugriffslatenz} - \text{überlappende Fehlzugriffslatenz})$$



Es gibt keine allgemeine Methode, die überlappende Fehlzugriffslatenz zu berechnen, deshalb braucht man für Simulationen von Speicherhierarchien für Out-of-order-Prozessoren unbedingt eine Simulation des Prozessors und der Speicherhierarchie. Nur wenn man die Ausführung des Prozessors bei einem Fehlzugriff beobachten kann, erkennt man, ob der Prozessor stillsteht, weil er auf Daten wartet, oder ob er einfach in der Zwischenzeit andere Arbeiten erledigt. Als Richtschnur gilt, dass der Prozessor häufig den Fehlzugriffsaufwand für einen L1-Cache-Fehlzugriff, der im L2-Cache trifft, durch nützliche Arbeit überbrücken kann, jedoch selten einen Fehlzugriff auf den L2-Cache.

Vertiefung: Bei der Optimierung der Algorithmen besteht die Herausforderung darin, dass sich die Speicherhierarchie für unterschiedliche Implementierungen derselben Architektur in der Cache-Größe, Assoziativität, Block-Größe und Cache-Anzahl unterscheidet. Um mit dieser Variabilität zurechtzukommen, parametrisieren einige neuere numerische Bibliotheken ihre Algorithmen und durchsuchen den Parameterraum zur Laufzeit, um die beste Kombination für einen bestimmten Computer zu finden.



Welche der folgenden Aussagen ist in Hinblick auf das Design mit mehreren Cache-Ebenen richtig?



1. Für Caches auf erster Ebene ist die Trefferzeit am wichtigsten, für Caches auf zweiter Ebene die Fehlzugriffsrate.
2. Für Caches auf erster Ebene ist die Fehlzugriffsrate am wichtigsten, für Caches auf zweiter Ebene die Trefferzeit.

Zusammenfassung

In diesem Abschnitt haben wir uns auf drei Themen konzentriert: Cache-Leistung, Ausnutzung der Assoziativität zur Reduzierung von Fehlzugriffsralten sowie die Verwendung von Cache-Speicherhierarchien mit mehreren Ebenen, um den Fehlzugriffsaufwand zu reduzieren.

Weil die Gesamtzahl der Zyklen, die für ein Programm aufgewendet werden, gleich der Summe der Prozessorzyklen und Speicherstillstandzyklen ist, kann das verwendete Speichersystem einen wesentlichen Einfluss auf die Programmausführungszeit haben. Wenn ein Prozessor schneller wird (indem CPI verringert und Taktrate gesteigert werden oder beides), steigt die relative Auswirkung der Speicherstillstandzyklen, so dass für eine hohe Leistung ein gutes Speichersystem unabdingbar ist. Die Anzahl der Speicherstillstandzyklen ist von der Fehlzugriffsrate und dem Fehlzugriffsaufwand abhängig. Die Herausforderung ist also, wie wir in Abschnitt 7.5 sehen werden, eine Reduzierung dieser Faktoren, ohne andere kritische Faktoren in der Speicherhierarchie negativ zu beeinflussen.

Um die Fehlzugriffsrate zu reduzieren, haben wir die Verwendung assoziativer Platzierungsschemata betrachtet. Solche Schemata können die Fehlzugriffsrate eines Caches reduzieren, indem eine flexiblere Platzierung der Blöcke innerhalb des Caches erlaubt wird. Vollassoziative Schemata erlauben, dass Blöcke irgendwo platziert werden, machen es aber auch erforderlich, dass jeder Block im Cache durchsucht wird, um eine Anforderung zu bedienen. Diese Suche wird im Allgemeinen durch einen Vergleicher pro Cache-Block implementiert, wobei die Tags parallel durchsucht werden. Die Kosten dieser Vergleicher machen große vollassoziative Caches ungeeignet für die Praxis. Satzassoziativ Caches sind eine praktische Alternative, weil wir nur Vergleiche mit den Tags der Blöcke eines einzigen Satzes durchführen müssen, der wiederum durch die Indizierung ausgewählt wurde. Satzassoziativ Caches haben höhere Fehlzugriffsraten, sind aber schneller im Zugriff. Welcher Assoziativitätsgrad die beste Leistung erbringt, ist sowohl von der Technologie als auch von den Implementierungsdetails abhängig.

Schließlich haben wir noch Cache-Speicherhierarchien betrachtet – eine Technik, den Fehlzugriffsaufwand durch Einführung eines größeren sekundären Caches zu reduzieren, der Fehlzugriffe auf den primären Cache bedient. Caches auf zweiter Ebene sind zu einem allgemeinen Instrument geworden, weil die Designer erkannt haben, dass die begrenzte Siliziumfläche und das Ziel hoher Taktraten verhindern, dass primäre Caches größer werden. Der sekundäre Cache, der häufig um einen Faktor 10 oder mehr größer als der primäre Cache ist, verarbeitet viele Zugriffe, die im primären Cache einen Fehlzugriff verursachen. In solchen Fällen ist der Fehlzugriffsaufwand die Zugriffszeit auf den sekundären Cache (in der Regel < 10 Prozessorzyklen) im Gegensatz zur Zugriffszeit auf den Speicher (in der Regel > 100 Prozessorzyklen). Wie bei der Assoziativität sind die Designentscheidungen bezüglich der Größe des sekundären Caches und seiner Zugriffszeit von verschiedenen Implementierungsaspekten abhängig.

... a system has been devised to make the core drum combination appear to the programmer as a single level store, the requisite transfers taking place automatically.

Kilburn et al., „One-level storage system,“ 1962

Virtueller Speicher (virtual memory) Eine Technik, die den Hauptspeicher als „Cache“ für den Sekundärspeicher verwendet.

7.4

Virtueller Speicher

Im vorigen Abschnitt haben wir gezeigt, wie Caches schnellen Zugriff auf zuvor genutzte Teile des Codes und der Daten eines Programms bieten. Analog dazu kann der Hauptspeicher als „Cache“ für den Sekundärspeicher dienen, der normalerweise unter Verwendung von Festplatten implementiert wird. Diese Technik wird auch als **virtueller Speicher (virtual memory)** bezeichnet. Ursprünglich gab es zwei Hauptgründe für die Verwendung von virtuellem Speicher: Er erlaubt eine effiziente und sichere gemeinsame Nutzung des Speichers durch mehrere Programme, und er befreit die Programmierer von der Last, mit einer kleinen, begrenzten Menge an Hauptspeicher auskommen zu müssen. Vier Jahrzehnte nach seiner Erfindung gilt der erste der genannten Gründe immer noch.