Organisation von Caches

Autor:
Christoph Ersfeld

Seminararbeit zur Veranstaltung
verteilte und parallele Systeme II - Prof. Dr. R. Berrendorf
7. Semester
WS 2002/2003

Inhaltsverzeichnis

Abbildungsverzeichnis **3**

Tabellenverzeichnis **3**

1 Notwendigkeit von Caches **4**
1.1 Speicherzugriffe bei der Programmausführung . . . 4
1.1.1 virtuelle Adressierung . . . 4
1.1.2 Datenorganisation u. Referenz-Lokalität . . . 5
1.2 Kommunikation zwischen Prozessor und Hauptspeicher . . . 5
1.2.1 Speichertypen . . . 6

2 Funktionsprinzip **7**
2.1 Definition und Funktionsweise . . . 7
2.2 Adressierung . . . 8
2.3 Wiederverwendung (reuse) . . . 8
2.4 Vorausgreifendes Laden (read ahead) . . . 9
2.5 Leistungsmaß „Cache-Hit-Rate“ . . . 9

3 Organisation und Verwaltung von Caches **9**
3.1 Chache-Zeile . . . 9
3.2 Organisationsformen . . . 9
3.2.1 Direkte Abbildung . . . 9
3.2.2 n-fach assoziativ abbildender Cache . . . 10
3.3 Identifikation eines Blocks . . . 10
3.4 Speicherkonsistenz u. Aktualisierungsstrategie . . . 11
3.5 Ersetzungsstrategie . . . 12

4 Cache-Typen und Anordnung im Rechnersystem **13**
4.1 Primär-Caches . . . 13
4.2 Sekundär- und Tertiär-Caches . . . 14
4.3 Beispiel: Intel Pentium IV . . . 14

5 Caches in SMP-Systemen **15**
5.1 Speicherbandbreitenbedarf . . . 15
5.2 Kohärenz . . . 15

Literaturverzeichnis **17**

Abbildungsverzeichnis

1 Einfaches Modell der direkten Prozessor-Hauptspeicher-Kommunikation 4
2 Codebeispiel zur Referenzlokalität (I) 5
3 Codebeispiel zur Referenzlokalität (II) 6
4 Erweitertes Modell der Prozessor-Hauptspeicher-Kommunikation mit Cache . . . 7
5 Lebenszyklus eines Datums im Cache, Phase 1+2 8
6 Cache-Miss-Rate in Abhängigkeit von der Cache- u. Cache-Zeilen-Größe 10
7 Direkt abbildend organisierter Cache 11
8 2-fach assoziativ abbildend organisierter Cache 11
9 Identifikation eines Blocks in einem 2-fach assoziativen Cache 12
10 Lebenszyklus eines Datums im Cache, Phase 3+4 12
11 Cache-Typen und Anordnung im Rechnersystem 13
12 Caches im Intel Pentium IV 14
13 Stark vereinfachtes Modell eines Multiprozessor-Systems 15

Tabellenverzeichnis

1 Caches im Intel Pentium IV 15

1 Notwendigkeit von Caches

Der Begriff des „Cache“-Speicher bezeichnet einen schnellen Zwischenspeicher, mit dessen Hilfe die Anzahl verhältnismässig unperformanter Hauptspeicherzugriffe des Prozessors verringert werden soll. Um die Notwendigkeit von Caches und deren Funktionsweise verstehen zu können, müssen die physikalischen Eigenschaften der Prozessor-Hauptspeicher-Kommunikation und Merkmale der Programmausführung bekannt sein.

1.1 Speicherzugriffe bei der Programmausführung

Folgendes stark vereinfachte Modell geht von der direkten Kommunikation zwischen Prozessor und Hauptspeicher aus (Abb. 1). Es soll sich dabei um eine Von-Neumann-Architektur mit gemeinsamen Programm- und Datenspeicher handlen.

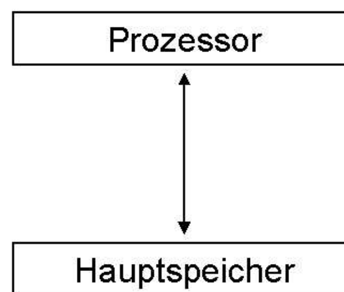

Abbildung 1: Einfaches Modell der direkten Prozessor-Hauptspeicher-Kommunikation

Für jeden Befehl des Programms müssen bei der Programmabarbeitung die nachstehenden Schritte ausgeführt werden:

1. Ggf. Übersetzung der virtuellen Program-Counter Adresse in die physische Adresse
2. Instruktion aus dem Speicher laden
3. Ggf. Übersetzung virtueller Adressen von Parametern in physische Adressen
4. Laden der Parameter in die Register bzw. Schreiben in den Speicher

Es ist ersichtlich, dass die mittlere Anzahl der Speicherzugriffe je Instruktion vom Anteil solcher Befehle abhängt, die lesend oder schreibend auf den Speicher zugreifen.

Das ist bei etwa 25% - 30% aller Instruktionen eines Programms der Fall. Zusammen mit dem Laden der Instruktion selbst erfordert jeder Befehl im Durchschnitt also zunächst etwa 1.25 bis 1.3 Speicherzugriffe [Pfister, 1998].

1.1.1 virtuelle Adressierung

Die gebräuchliche virtuelle Adressierung erfordert bei jedem Speicherzugriff die Übersetzung der virtuellen Adresse in die physische Adresse.

Dazu übergibt die CPU die virtuelle Adresse an die Memory Management Unit (MMU). Die MMU liesst aus der sogenannten Seitentabelle einen Offset, mit dem durch Addition die physische Adresse berechnet wird [Tanenbaum, 1992].

Für jede Speicherreferenz muss nun zwei mal auf den Speicher zugegriffen werden, ein mal auf die Seitentabelle und anschließend auf das referenzierte Wort, was die durchschnittliche Anzahl erforderlicher Speicherzugriffe je Instruktion auf 2.5 bis 2.6 verdoppelt.

1.1.2 Datenorganisation u. Referenz-Lokalität

```
void arrayFunc()
{
   char *ptr;

   char array[] = { 10, 11, 12, 13, 14 };

   printf("ptr:          Adresse: %p\n", &ptr);

   for(ptr=array; ptr < array+5; ptr++)
      printf("array[%d]: %d, Adresse: %p\n", ptr-array, *ptr, ptr);
}
```

```
Ausgabe:
ptr:            Adresse: 0x73fd3c
array[0]: 10, Adresse: 0x73fd40
array[1]: 11, Adresse: 0x73fd41
array[2]: 12, Adresse: 0x73fd42
array[3]: 13, Adresse: 0x73fd43
array[4]: 14, Adresse: 0x73fd44
```

Abbildung 2: Codebeispiel zur Referenzlokalität (I)

Typischerweise greifen Programme in zeitlicher Nähe auf relativ begrenzte Speicherbereiche zu, oft dabei auch sequentiell. Dieses Verhalten wird als Referenz-Lokalität bezeichnet und ist vor allem durch die Datenorganisation im Speicher und den Kontrollfluss begründet.

Der Compiler legt beispielsweise Felder (Arrays) und aufeinanderfolgend deklarierte Variablen und Konstanten sequentiell im Speicher ab, was das Codebeispiel aus Abbildung 2 demonstriert. Diese Form der Referenzlokalität wird deshalb als räumliche Lokalität (spatial locality) bezeichnet.

Das bisher betrachtete vereinfachte Modell der Prozessor-Hauptspeicher- Kommunikation (Abb. 1) kann von dieser Eigenschaft nicht profitieren. Es ist aber leicht ersichtlich, dass ein Zwischenspeichern benachbarter Speicherbereiche die Anzahl erforderlicher Hauptspeicherzugriffe erheblich reduzieren kann.

Das zweite Codebeispiel (Abb. 3) zeigt einen Fall, in dem räumliche Referenzlokalität typischerweise nicht gegeben ist. Eine verkettete Liste wird z.B. dann verwendet, wenn die Anzahl zu speichernder Werte nicht zuvor bekannt ist oder eine Sortierordnung auf der Liste angewandt werden soll. Werden die einzelnen Elemente der Liste (bei dieser suboptimalen Implementierung) zur Laufzeit allokatiert, liegen sie im Speicher unter Umständen weit voneinander entfernt auf dem Heap, zudem in beliebiger Reihenfolge. Im wesentlichen ist eine Reduzierung der Hauptspeicherzugriffe hier nur durch Nutzung der sogenannten zeitlichen Lokalität (temporal locality) zu erzielen. Darunter versteht man den zeitlich nahen, wiederholten Zugriff auf Speicherbereiche.

1.2 Kommunikation zwischen Prozessor und Hauptspeicher

Wäre der Hauptspeicher im Modell aus Abschnitt 1.1 genauso schnell wie der Prozessor, würde jede Speicherreferenzierung eine Taktdauer in Anspruch nehmen. Eine Zwischenspeicherung

```
struct listItem* linkedListAppend(char value, struct listItem* tail)
{
   // neues Element auf dem Heap erzeugen
   struct listItem *item = (struct listItem*)malloc(sizeof(struct listItem));

   // An die verkettete Liste anhängen
   item->value = value;
   item->next  = NULL;
   tail->next  = item;

   return item;
}
```

```
// liegt im statischen Datenbereich
struct listItem head0 = { 10, NULL };

void linkedListFunc()
{
   struct listItem *ptr, *item1, *item2;

   // item1 wird zur Laufzeit auf dem Heap erzeugt
   item1 = linkedListAppend(11, &head0);

   // hier macht das Programm irgendwas anderes
   // allokatiert dabei Speicher auf dem Heap
   doSomethingElse();

   // item2 wird zur Laufzeit auf dem Heap erzeugt
   item2 = linkedListAppend(12, item1);

   for(ptr = &head0; ptr != NULL; ptr = ptr->next)
      printf("Wert: %d, Adresse listItem: %p\n", ptr->value, ptr);
}
```

```
struct listItem  {
   char           value;
   struct listItem *next;
};
```

```
Ausgabe:
Wert: 10, Adresse listItem: 0x403010
Wert: 11, Adresse listItem: 0x970758
Wert: 12, Adresse listItem: 0x974770
```

Abbildung 3: Codebeispiel zur Referenzlokalität (II)

könnte keinen Geschwindigkeitsvorteil bewirken.

In der Praxis ist der Hauptspeicher jedoch erheblich langsamer, so dass der Prozessor bei der Ausführung von Speicherzugriffen Wartezyklen einlegen muss. Für einen Prozessor mit einer Taktrate von 2 GHz wäre in unserem vereinfachten Modell ein Hauptspeicher aus SDRAM-Modulen mit einer Zugriffszeit von 7.5 ns etwa um den Faktor 15 zu langsam.

Die wichtigsten Kenngrößen sind hier die Latenzzeit und die Zykluszeit. Die Latenzzeit bezeichnet die Dauer, nach der ein angefragtes Datum bereitgestellt ist. Die Zykluszeit ist die Mindestdauer, die zwischen zwei aufeinanderfolgenden Zugriffen verstreichen muss [Rechenberg, 1997]. Diese Kenngrößen variieren zwischen den verschiedenen Speichertypen, deren wesentliche Eigenschaften nachfolgend betrachtet werden, erheblich.

1.2.1 Speichertypen

SRAM (static RAM). Die Speicherzellen sind als Matrix angeordnet. Jede 1-Bit Speicherzelle enthält ein Flipflop, das seine Information dauerhaft speichern kann. Ein Refresh wie beim

DRAM und die damit verbundene Verzögerung entfällt, ebenso das Zurückschreiben einer gelesenen Zeile. Die Zykluszeit entspricht daher der Zugriffszeit. Die Zugriffszeit ist u.a. geringer, weil die Adressleitungen für Spalten und Reihen gleichzeitig angelegt werden können.

Nachteile des SRAM-Speichers sind der hohe Fertigungsaufwand und die damit verbundenen hohen Kosten, sowie eine verglichen mit DRAM geringere Speicherdichte.

DRAM (dynamic RAM). Die Speicherzellen sind ebenfalls als Matrix angeordnet. Jede 1-Bit Speicherzelle besteht aus nur einem Transistor und einem Kondensator. Durch Leckströme kommt es zur Entladung des Kondensators, so dass DRAM-Speicher in regelmäßigen Abständen, ca. 8 ms, aufgefrischt werden muss (Refresh). Das Lesen bzw. Schreiben eines Wortes gestaltet sich verglichen mit SRAM aufwendiger: Zuerst wird die adressierte Zeile aus der Matrix gelesen und zwischengespeichert. Der Inhalt der gelesenen Zeile wird dabei zerstört und muss anschließend zurück geschrieben werden. Während des Zurückschreibens kann der Speicher keinen weiteren Zugriff verarbeiten, so dass die Zykluszeit größer ist als die Zugriffszeit. An die zwischengespeicherte Zeile wird dann die Spaltenadresse angelegt, um das gewünschte Datum zu adressieren.

Die Vorteile des DRAM sind der geringe Preis und die hohe Speicherdichte, weshalb er i.d.R. als Hauptspeicher eingesetzt wird.

Es gibt verschiedene Ansätze, die nachteiligen Eigenschaften des DRAM zu reduzieren. Der SDRAM (synchronous DRAM) orientiert sich am Systemtakt und verwaltet intern mehrere Bänke. Beim abwechselnden Zugriff muss deshalb die Erholungszeit nicht abgewartet werden. Zudem sind so Optimierungen durch Pipeline-Verfahren beim Speicherzugriff möglich.

2 Funktionsprinzip

2.1 Definition und Funktionsweise

Wie in Abschnitt 1 vorgreifend erwähnt ist der Cache-Speicher ein schneller Zwischenspeicher aus SRAM. Er nutzt gezielt die in Abschnitt 1.1 aufgezeigten Merkmale der Programmausführung, um durch Wiederverwendung die Anzahl der langsamen Hauptspeicherzugriffe zu minimieren.

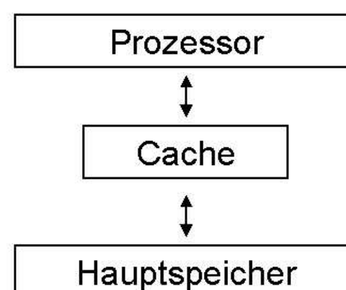

Abbildung 4: Erweitertes Modell der Prozessor-Hauptspeicher-Kommunikation mit Cache

Statt wie bisher direkt auf den Hauptspeicher zuzugreifen, greift der Prozessor im gemäß Abbildung 4 erweiterten Modell nur noch mittelbar über den Cache auf den Hauptspeicher zu.

2.2 Adressierung

Der Prozessor legt die gewünschte Speicheradresse direkt am Cache statt am Hauptspeicher an. Die Steuerlogik des Cache überprüft an Hand der Adresse, ob das entsprechende Wort bereits im Cache vorliegt. Im Falle eines Treffers (hit) kann das Wort unmittelbar in den Prozessor geladen bzw. im Cache verändert werde. Die Zyklusdauer entspricht idealerweise einem Prozessortakt.

Im Falle eines Fehlzugriffs (miss) lädt der Cache einen Block fixer Größe, eine sogenannte Cachezeile, mit dem darin enthaltenen adressierten Wort, das an den Prozessor gegeben bzw. verändert wird.

Der Prozessor hat die Möglichkeit, den Cache entweder mit physischen Adressen oder mit virtuellen Adressen zu adressieren. Virtuell adressierte Caches, sogenannte virtuelle Caches, sind im allgemeinen schneller. Die Adressübersetzung durch die MMU muss nur bei einem Cache-Miss erfolgen. Nachteilig steht dem ein erhöhter Verwaltungsaufwand gegenüber. Virtuelle Adressen sind nur im Kontext eines Prozesses gültig, so dass der Cache-Inhalt nach einen Prozesswechsel gelöscht werden muss oder zusätzliche Verwaltungsinformationen notwendig sind. Desweiteren können gleiche Speicherbereiche unter unterschiedlichen virtuellen Adressen im gleichen Cache bzw. in unterschiedlichen Caches präsent sein. Die Sicherstellung der Kohärenz gestaltet sich dadurch insbesondere in SMP-Systemen aufwendiger [Jacob, 1997] [Rechenberg, 1997].

2.3 Wiederverwendung (reuse)

Bestimmend für den durch das Caching erzielbaren Nutzen ist die Häufigkeit, mit der auf zwischengespeicherte Daten zugegriffen werden kann. Ein Programm sollte folglich so organisiert sein, dass möglichst viele seiner Programm- und Datenanteile in den Cache geladen, dort gehalten und wiederverwendet werden können (reuse). Jede Wiederverwendung macht einen Zugriff auf den Hauptspeicher bzw. die nächst tiefere Stufe der Cache-Hierarchie überflüssig (Abb. 5).

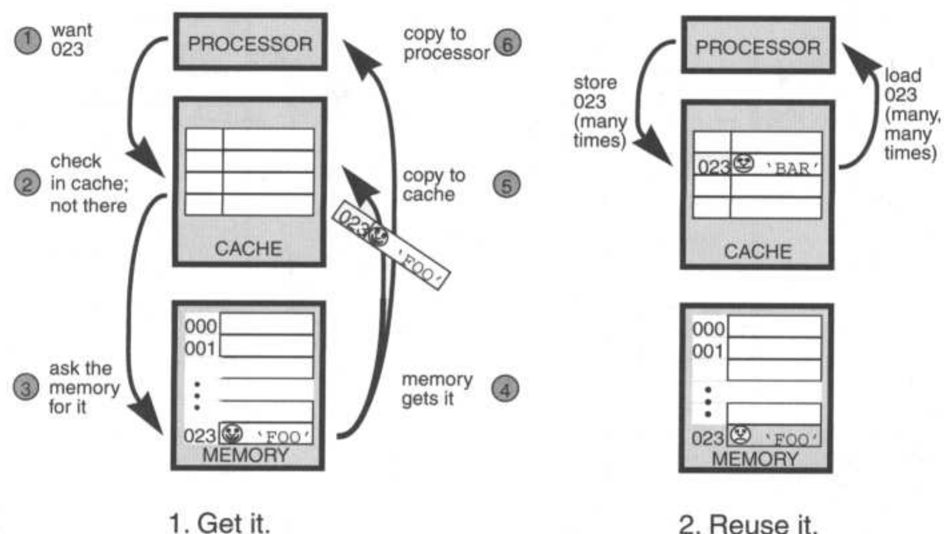

Abbildung 5: Lebenszyklus eines Datums im Cache, Phase 1+2

2.4 Vorausgreifendes Laden (read ahead)

„Read ahead“ bezeichnet das vorausgreifende Laden von Speicherblöcken in den Cache, um die entsprechenden Bereiche bei der ersten Verwendung bereits im Cache präsent zu haben. Auf diese Weise kann ein Fehlzugriff verhindert werden, der den Prozessor zu Wartezyklen zwingt. Dadurch lassen sich signifikante Leistungssteigerungen erzielen.

Die Intel x86-Architektur bietet dazu ab dem Pentium III mehrere „PREFETCH“-Maschinenbefehle, die sich darin unterscheiden, in welche Caches (s. Abschnitt 4) das Datum geladen werden soll [Intel, 2002].

2.5 Leistungsmaß „Cache-Hit-Rate“

Die Effizienz des Cache wird mit der sogenannten „Cache-Hit-Rate“ angegeben. Sie berechnet sich nach folgender Formel:

$$h[\%] = \frac{Anzahl\,hits}{Gesamtzahl\,Zugriffe} \cdot 100$$

Eine Cache-Hit-Rate von 100% wäre das Optimum und würde bedeuten, dass jeder Speicherzugriff aus dem Cache bedient werden konnte. Sie ist neben Unterschieden in der internen Organisation des Cache insbesondere von der Häufigkeit der Wiederverwendung bzw. den in Abschnitt 1.1 erläuterten Programmeigenschaften abhängig. Bei tyischen Anwendungen sollte sie den Wert von 90% nicht unterschreiten.

In der Literatur wird statt der cache hit rate auch häufig der inverse Wert, die „Cache-Miss-Rate“ verwendet.

3 Organisation und Verwaltung von Caches

3.1 Chache-Zeile

Wie bereits in Abschnitt 2.2 erläutert verwaltet ein Cache Speicherblöcke fester Größe. Diese Blöcke werden als Cache-Zeilen (cache-line) bezeichnet werden.

Wegen der räumlichen Lokalität wirkt sich eine höhere Block-Größe tendentiell positiv auf die Cache-Hit-Rate aus (Abb. 6). Demgegenüber wächst der Aufwand bei einem Fehlzugriff, den Block aus dem nierdigeren Cache-Level bzw. dem Hauptspeicher nachzuladen.

3.2 Organisationsformen

Die Organisationsform eines Cache legt fest, in welcher Cache-Line ein Hauptspeicherblock abgelegt wird und wie ein Block im Cache aufzufinden ist [Rechenberg, 1997].

3.2.1 Direkte Abbildung

In einem direkt abbildenden Cache (direct mapped) mit einer Kapatzität von N Cache-Zeilen wird ein Hauptspeicherblock B in der Cache-Zeile $B\,mod\,N$ gespeichert (Abbildung 7).

Ein Nachteil dieser Organisationsform besteht darin, dass sich zwei oder mehr Speicherblöcke, die auf diese Cache-Zeile abgebildet werden, bei gleichzeitger Nutzung wechselseitig aus dem Cache verdrängen. Das häufige Umladen der Inhalte verringert die Effizienz, wenn auch direkt abbildende Caches durch ihren verhältnismäßig einfachen Aufbau am schnellsten arbeiten.

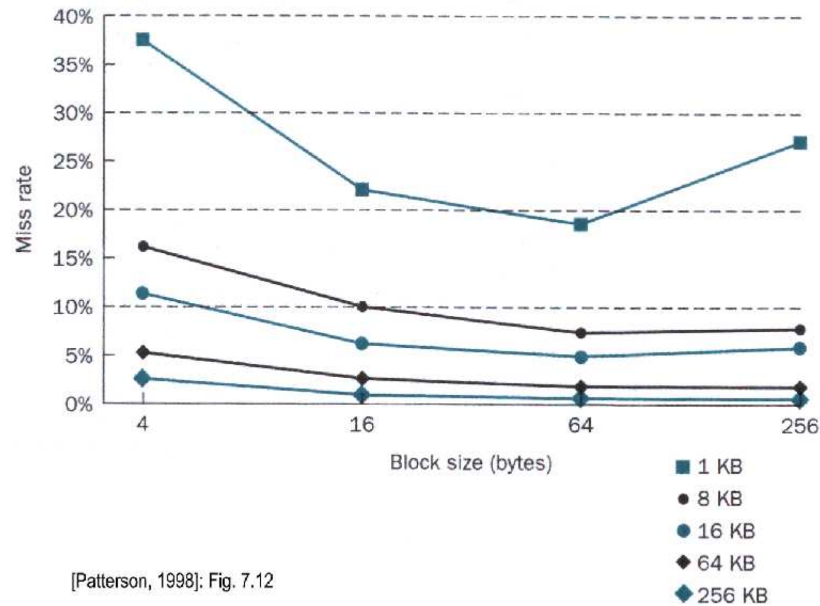

Abbildung 6: Cache-Miss-Rate in Abhängigkeit von der Cache- u. Cache-Zeilen-Größe

3.2.2 n-fach assoziativ abbildender Cache

Ein Cache heisst n-fach assoziativ abbildend (n-way set associative), wenn ein Speicherblock B in n Cache-Lines, einem sogenannten Satz (Set), abgelegt werden kann. Der Cache ist dazu unterteilt in $M = N/n$ Sätze. Der einem Speicherblock zugeordnete Satz bestimmt sich durch den Ausdruck $B \, mod \, M$. Abbildung 8 zeigt einen 2-fach assoziativ abbildenden Cache.

Der Spezialfall $n = 1$ beschreibt einen direkt abbildenden Cache, $n = N$ einen voll-assoziativen Cache, in dem ein Speicherblock an beliebiger Stelle abgelegt werden kann.

Für jeden Assoziativitäts-Grad n werden intern n Komparatoren benötigt, so dass der Hardwareaufwand mit zunehmender Assoziativität wächst, zugleich die Geschwindigkeit sinkt. Dennoch wächst die Effizienz in Abhängigkeit vom Grad der Assoziativität. Als Kompromiss zwischen Realisierungsaufwand, Geschwindigkeit und Effizienz kommen besonders häufig 2, 4, oder 8-fach assoziative Caches zum Einsatz.

3.3 Identifikation eines Blocks

Jeder Cache-Zeile ist eine große Anzahl von Hauptspeicherblöcken zugeordnet. Es bedarf folglich eines Mechanismus, mit dem ein im Cache gespeicherter Block identifiziert werden kann. Jede Cache-Zeile ist deshalb mit zusätzlichen Verwaltungsinformationen versehen, einer Kennung (tag) und einem oder mehreren Flag-Bits, darunter ein Bit für die Gültigkeit der Zeile (valid bit). Die Kennung bestimmt sich aus der Adresse des Speicherblocks.

Abbildung 9 zeigt die Funktionsweise des Identifikationsmechanismus am Beispiel eines Adressierungsvorgangs bei einem 2-fach assoziativen Cache mit 128 Zeilen (64 Sätzen) der Länge 16 Bytes.

In Schritt 1 wird die Adresse dekodiert. Die niederwertigsten 4 Bit der 32-Bit Adresse bezeichnen eines der 16 möglichen Byte einer Cache-Zeile. Der dem Speicherblock zugeordnete Satz bestimmt sich aus den nächst höherwertigen 6 Bits. Die verbliebenen Adressbits bilden die Kennung und identifizieren den Block eindeutig innerhalb des Satzes.

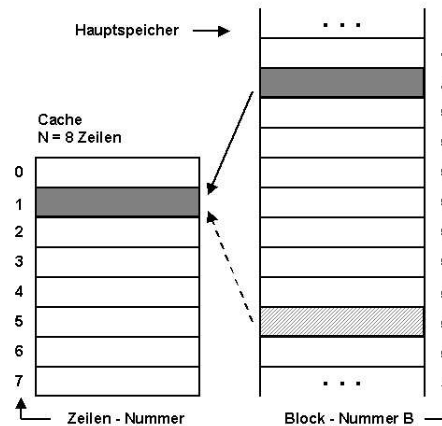

Abbildung 7: Direkt abbildend organisierter Cache

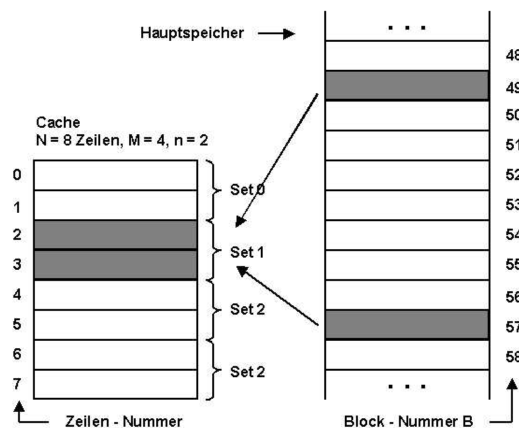

Abbildung 8: 2-fach assoziativ abbildend organisierter Cache

In Schritt 2 wird der Satz ausgewählt, in Schritt 3 ausgelesen und die Kennung jeder Zeile mit der Kennung aus der angelegten Adresse verglichen, sofern das Valid-Bit die Gültigkeit des Blocks anzeigt.

3.4 Speicherkonsistenz u. Aktualisierungsstrategie

Das Problem, wie und wann der korrespondierende Hauptspeicherbereich nach einer Veränderung im Cache aktualisiert werden soll, wird allgemein als Datenkonsistenzproblem (consistency problem) bezeichnet. Die zwei grundsätzlichen Lösungsstrategien sind das „Durchschreiben“ (write through) und das „Zurückschreiben“ (write back).

write back Änderungen werden zunächst nur im Cache ausgeführt. Erst das Verdrängen der Cachezeile durch einen normalen Ersetzungsvorgang oder der Zugriff eines anderen Prozessors veranlassen das Rückschreiben in den Hauptspeicher. Um das Zurückschreiben nicht modifizierter Blöcke zu vermeiden gibt es zu jeder Cachezeile ein sogenanntes „dirty bit“, das nach einer Modifikation gesetzt wird. Besondere Beachtung erfordert die vorübergehende Dateninkonsistenz beim Multiprozessorsystemen.

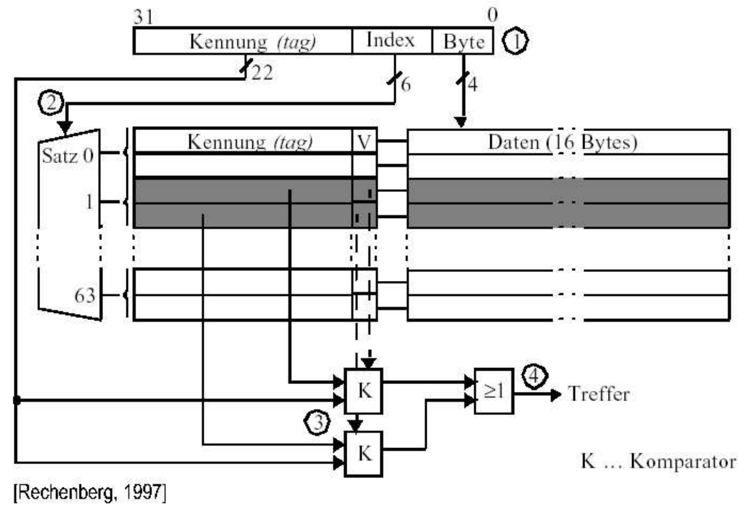

Abbildung 9: Identifikation eines Blocks in einem 2-fach assoziativen Cache

write through Jede Änderung im Cache wird sofort in den Hauptspeicher zurückgeschrieben. So bleibt zwar die Datenkonsistenz im Hauptspeicher erhalten, es kommt jedoch zu einer hohen Belastung des Speicherbusses.

3.5 Ersetzungsstrategie

Die Frage, welche Cache-Zeile beim Laden eines neuen Speicherblocks in den Cache überschrieben wird (Abb. 10), wenn keine der in Frage kommenden Zeilen mehr frei ist, wird als Ersetzungsproblem (replacement problem) bezeichnet.

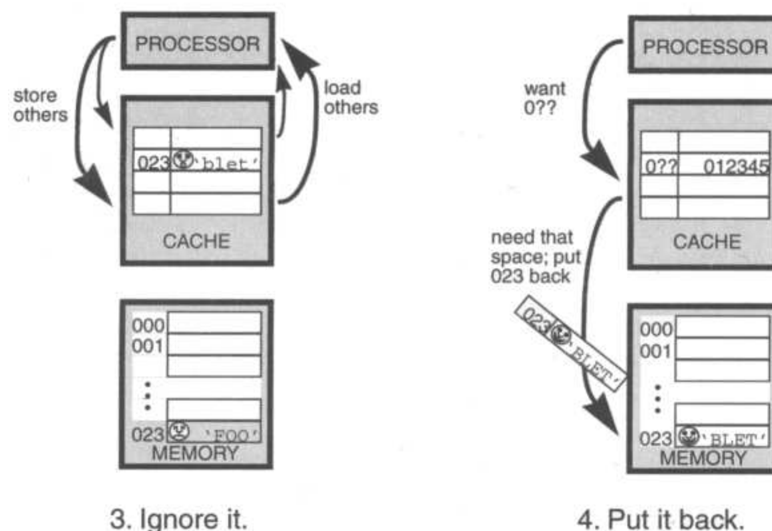

Abbildung 10: Lebenszyklus eines Datums im Cache, Phase 3+4

In einem direkt abbildenden Cache ist der zu verdrängende Block eindeutig bestimmt (s. Abschnitt 3.2.1), so dass dieses Problem nur für assoziative Caches gelöst werden muss. Die beiden

gebräuchlisten Strategien sind „last recently used“ (LRU) und „random“.

LRU Nach dem LRU-Prinzip wird die Cache-Zeile ersetzt, auf die am längsten nicht mehr zugegriffen wurde. Dazu muss für jede Cache-Zeile eine Alterungsinformation verwaltet werden, die bei jedem Zugriff fortgeschrieben wird. Deshalb sind Caches nach dem LRU-Prinzip hardwareseitig aufwendiger als solche nach dem random-Verfahren.

random Bei diesem Verfahren wird die zu ersetzende Cachezeile auf Basis eines Pseudozufallsgenerators ausgewählt. Auch wenn dieses Verfahren keinerlei Rücksicht auf die Aktualität der ausgewählten Cachezeile nimmt, ist es sehr effizient.

4 Cache-Typen und Anordnung im Rechnersystem

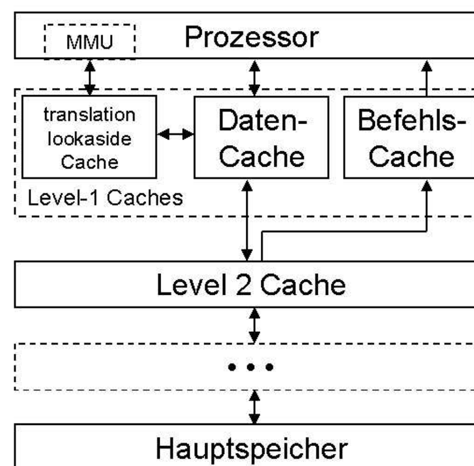

Abbildung 11: Cache-Typen und Anordnung im Rechnersystem

In der Speicherhierarchie heutiger Systeme finden sich Cache-Speicher mit verschiedenen Aufgaben und Eigenschaften. Abbildung 11 soll hier einen Überblick über die wichtigsten Typen verschaffen, bevor im folgenden detailliert auf sie eingegangen wird.

Es sei erwähnt, dass es eine Vielzahl weiterer spezialisierter Caches in aktuellen Prozessoren gibt, auf die hier nicht eingegangen wird.

4.1 Primär-Caches

Die sogenannten Primär- oder Level-1 Caches sind üblicherweise im Prozessor integriert und arbeiten mit dessen Taktgeschindigkeit. Die Größe der Level-1 Caches beträgt nur etwa 1kB bis 256kB. In erster Linie wird die Größe des Level-1 Cache durch die Herstellungskosten und Leistungsaufnahme begrenzt. Die Cache-Lines (s. Abschnitt 3.1) sind i.d.R. nicht länger als 32 Bytes und damit in wenigen Zyklen nachladbar.

Auf Ebene der Primär-Caches sind Daten- und Befehls-Cache meist getrennt (split cache), oft gibt es zusätzlich einen sogenannten Translation Lookaside Buffer (TLB). Er ist ein spezieller Daten-Cache in der MMU, der die Anzahl der Zugriffe auf die Seitentabellen minimieren soll, indem er eine geringe Anzahl von Seitendiskriptoren (Einträge einer Seitentabelle) zwischenspeichert.

Die Unterteilung der Primär-Caches erhöht die absolute Bandbreite, mit der auf den Cache zugegriffen werden kann. Zusätzlich ermöglicht die Trennung spezialisierte Leistungsoptimierungen.

Die Adressierung der Primärcaches erfolgt häufig virtuell.

4.2 Sekundär- und Tertiär-Caches

Zwischen dem Hauptspeicher und den Primär-Caches finden sich häufig weitere Caches in der Speicherhierarchie, die Sekundär-Caches (Level-2) und Tertiär-Caches (Level-3). Es handelt sich um SRAM-Speicher, dessen Geschwindigkeit zwischen der des Hauptspeichers und des Level-1 Cache einzuordnen ist. Die reduzierte Geschwindigkeit ermöglicht größere Kapazitäten, die typischerweise ca. 256 KB bis 8 MB betragen. Zusätzlich sind die Cachezeilen oft ein Vielfaches größer als bei den Primär-Caches.

Davon profitieren Anwendungen, die auf grossen Datenbeständen arbeiten und wenig räumliche Lokalität aufweisen, z.B. Datenbanken (s. Abschnitt 1.1).

Sekundär- und Tertiär-Caches unterscheiden i.d.R. nicht zwischen Daten und Instruktionen (unified Cache).

4.3 Beispiel: Intel Pentium IV

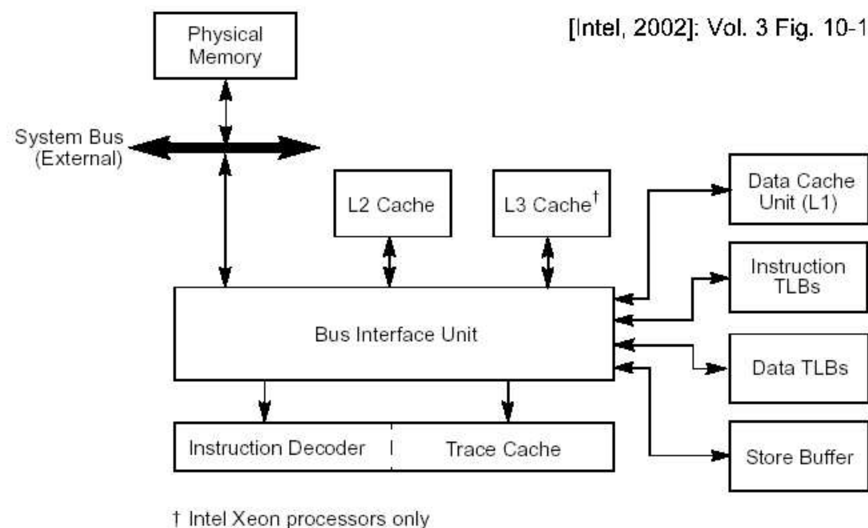

Abbildung 12: Caches im Intel Pentium IV

Der Intel Pentium IV verfügt über Level-1 und Level-2 Caches sowie einige Spezial-Caches, die bislang nicht betrachtet wurden (Abb. 12).

Ab dem Pentium IV kommt erstmals der „Trace-Cache“ statt eines L1-Befehls-Cache in der x86-Familie zu Einsatz. Er speichert bereits dekodierte Instruktionen (μops). Der „Store-Buffer“ puffert alle Schreibvorgänge, so dass die CPU das Schreiben in den Hauptspeicher und/oder Cache nicht abwarten muss.

	Größe	**Zeilengröße**	**Assoziativtätsgrad**
L1-Data	8 kB	64 Bytes	4
L1-Instruction	nicht vorhanden		

	Größe	Zeilengröße	Assoziativtätsgrad
Store-Buffer	24 Einträge		
Instruction-Trace-Cache	12 μops		8
Data-TLB	64 Einträge		voll
Instruction-TLB	128 Einträge		4
L2-Cache	256 o. 512 kB (on die)	64 Bytes	8
L3-Cache (nur XEON)	512 kB o. 1 MB	64 Bytes	8

Tabelle 1: Caches im Intel Pentium IV

Wie aus Tabelle 1 hervorgeht, sind die Level-1 Caches verhältnismäßig klein, demgegenüber ist die Zeilenlänge groß. Die Vorgängermodelle aus der Intel P6-Familie besaßen einen mit 16 KB doppelt so großen L1-Daten-Cache mit einer Zeilengröße von 32 Bytes, die Größe der Level-2-Caches reichte bis zu 2 MB bei 4-facher Assoziativität.

5 Caches in SMP-Systemen

5.1 Speicherbandbreitenbedarf

Abbildung 13 zeigt eine sehr vereinfachte Abstraktion eines Multiprozessor-Systems mit einem gemeinsamen Hauptspeicher.

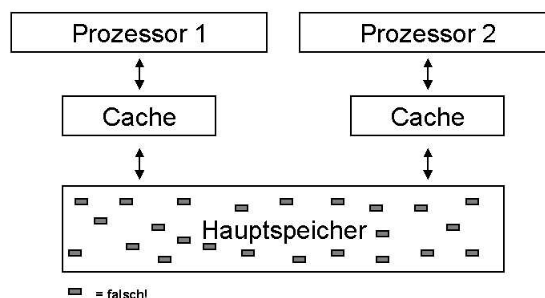

Abbildung 13: Stark vereinfachtes Modell eines Multiprozessor-Systems

Es ist leicht ersichtlich, dass sich die Probleme, die bei Einprozessorsystemen Caches zur Leistungssteigerung notwendig machen (Abschnitt 1.1), bei Multiprozessor-Systemen mit zunehmender Prozessorenzahl besonders auswirken.

Die Bandbreite des Hauptspeichers skaliert nicht. Ist sie in einem Einprozessorsystem bei einer Cache-Miss-Rate von 10% ausgelastet, so sollte sie in einem Dualprozessorsystem nicht mehr als 5% je CPU betragen. Deshalb finden in SMP-Systemen tendentiell größere Caches Anwendung. Eine weitere Verbesserungsmöglichkeit ist das Interleaving. Der Hauptspeicher wird in mehrere physikalische Bänke aufgeteilt, auf die gleichzeitig zugegriffen werden kann.

5.2 Kohärenz

Die Prozessoren eines SMP-Systems kommunizieren über den gemeinsamen Speicher. Die Korrektheit erfordert, dass jeder Prozessor nur mit dem aktuell gültigen Datum arbeitet. Wie Abbildung 13 veranschaulicht, enthält der Hauptspeicher jedoch ungültige Bereiche, nachdem sie im Cache geändert aber noch nicht zurück geschrieben wurden.

Auf die Verfahren zur Sicherstellung der Kohärenz wird in anderen Seminararbeiten eingegangen.

Literatur

[Intel, 2002] Intel Corp.: IA-32 Intel Architecture Software Developerś Manual. 2002 http://developer.intel.com/design/pentium4/manuals/

[Jacob, 1997] Jacob, B.: Segmented Addressing Solves the Virtual Cache Synonym Problem. University of Maryland, 1997 http://www.ee.umd.edu/ blj/papers/UMD-SCA-97-01.pdf

[Pfister, 1998] Pfister, G. F.: In Search of Clusters. Prentice-Hall, New Jersey 1998

[Przybylski, 1990] Przybylski, S. A.: Cache and Memory Hierarchy Design - A Performance-Directed Approach. Morgan Kaufmann Publishers, San Mateo (CA) 1990

[Rechenberg, 1997] Rechenberg, P., Pomberger, G.: (Hrsg.): Informatik Handbuch. Hanser, München, Wien 1997

[Patterson, 1998] Patterson, D. A., Hennessy, J. L.: Computer Organization and Design, 2nd Edition. Morgan Kaufmann Publishers, San Francisco 1998

[Tanenbaum, 1992] Tanenbaum A.: Moderne Betriebssysteme. Hanser, München, Wien 1992