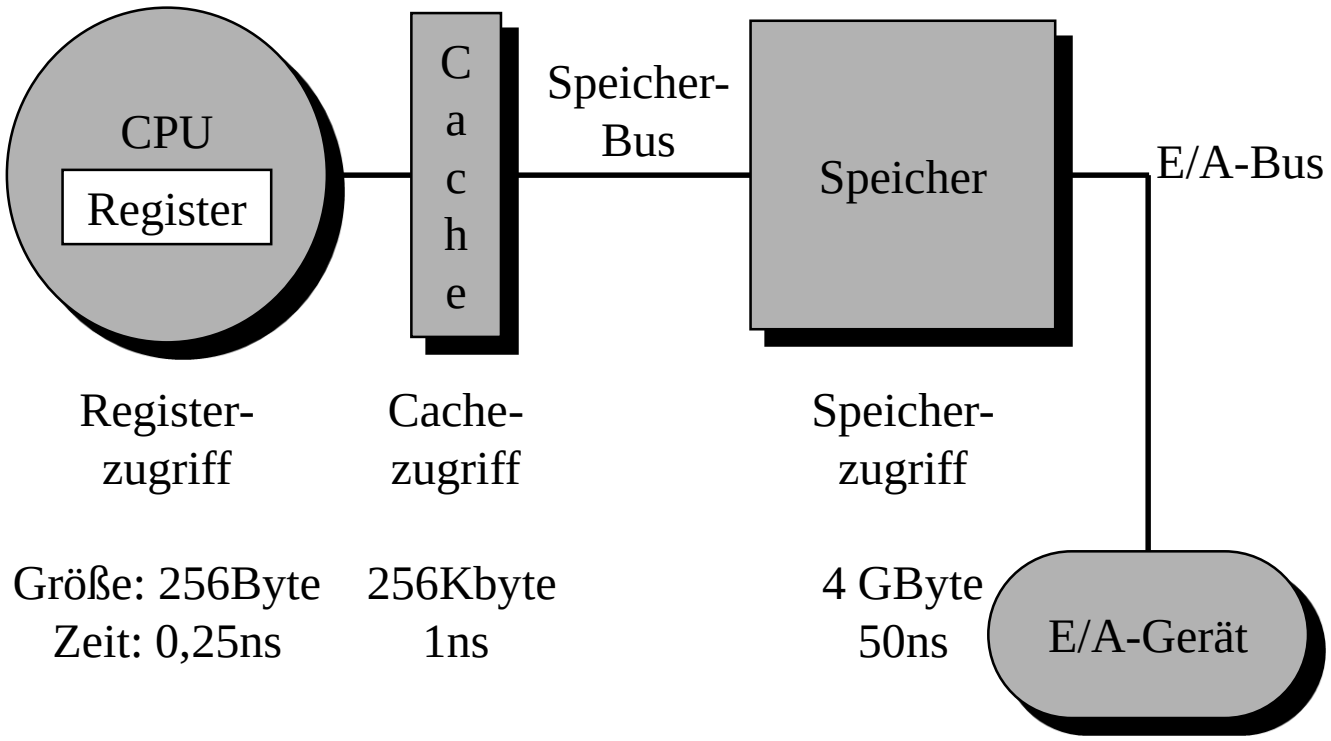
Das Konzept der Speicherhierarchie

Small is fast, daher sind kleine Speicher schneller (und kosten mehr pro Byte).

Vergrößerung von Speichern und schnellerer Zugriff sind aber Schlüsselfunktionen in der Beschleunigung von Rechnern. Daher werden wir uns das, was wir bisher gelernt haben zunutze machen, um das Prinzip der Speicherhierarchie zu verstehen.

Lokalität => kürzlich benutztes Datum wird bald wieder gebraucht. Making the common case fast bedeutet, daß wir den Zugriff hierauf beschleunigen müssen. Smaller is faster bedeutet, daß es sinnvoll ist diese Daten in einem kleinen Speicher nahe der CPU zu halten, während die ansteigend größeren (und langsameren Speicher) immer weiter weg von der CPU anzuordnen sind.
Dies nennt man eine **Speicher-Hierarchie.**

Das folgende Bild zeigt eine typische Speicherhierarchie.


CPU
Register
C
a
c
h
e
Speicher-
Bus
Speicher
E/A-Bus
Register-
zugriff
Cache-
zugriff
Speicher-
zugriff
Größe: 256Byte
Zeit: 0,25ns
256Kbyte
1ns
4 GByte
50ns
E/A-Gerät
Platten-
speicher-
zugriff
600 GByte
5 ms

Def: Ein Cache ist ein kleiner, schneller Speicher, der räumlich nahe der CPU angeordnet ist.

Cash (Cache) heißt wörtlich übersetzt „Bargeld“. Die Analogie ist offensichtlich: Der Hauptspeicher ist das Konto, ich möchte aber schnell auf einen Teil meines Geldes zugreifen können, das ich daher cash bei mir habe.

Wenn auf den Cache zugegriffen wird, und das gesuchte Datum wird gefunden, so bezeichnen wir diesen Zugriff als **Cache-Hit**. Wird das Datum nicht im Cache vorgefunden, so handelt es sich um einen **Cache-Miss**. Wenn Daten in den Cache übertragen werden, so geschieht dies in **Blöcken** fester Größe.
Zeitliche Lokalität lehrt uns, das das eben benutzte Datum sinnvollerweise im Cache aufbewahrt werden soll. Aber räumliche Lokalität sagt, dass mit großer Wahrscheinlichkeit andere Daten aus demselben Block bald zugegriffen werden.

Die Zeit, die ein cache miss braucht, ist bestimmt durch die Zeit, die wir brauchen, um einen Block vom Hauptspeicher in den Cache zu kopieren. Diese hängt ab von der Bandbreite und der Zugriffszeit des Hauptspeichers. In der Regel wartet die CPU, bis der Block übertragen ist. Die Zeit, die die CPU wartet wird als **cache miss penalty** (Strafe) bezeichnet.

Wenn der Computer einen virtuellen Speicher hat, brauchen nicht alle Daten zu jedem Zeitpunkt im Hauptspeicher gehalten zu werden, sondern sie können auf die Platte ausgelagert werden. Wiederum ist der Adressbereich in Blöcke fester Länge unterteilt, die in diesem Falle **Pages** genannt werden. Jede Page ist zu jedem Zeitpunkt entweder im Hauptspeicher oder auf der Platte.
Wenn der Computer einen virtuellen Speicher hat, so benimmt sich dieser zum Hauptspeicher so wie der Hauptspeicher zum Cache. Entsprechend dem Cache miss führt ein Zugriff auf ein Element, das auf der Platte ist zu einem **Page fault**. Dadurch wird ein Einlesen der gesamten Seite von der Platte in den Hauptspeicher initialisiert. In dieser Zeit ist es nicht sinnvoll, die CPU leer zu lassen, da das Einlesen der Seite zu lange dauert. Stattdessen wird die CPU auf eine andere Task geschaltet, während auf einen Page fault reagiert wird.
Die folgende Tabelle zeigt die typischen Größen und Zugriffszeiten jeder Ebene in der Speicherhierarchie eine Maschine im Bereich Desktop PC bis high end Server.

Speicherhierarchie

Level	1	2	3	4
Bezeichnung	Register	Cache	Hauptspeicher	Platte
Typische Größe	<2KB	<16KB	<8GB	>400 GB
Implementierung	Custom CMOS mehrere Ports	On-chip oder off-chip CMOS SRAM	CMOS SDRAM	Magnetplatte
Zugriffszeit	0,25-1ns	0,5-5ns	20-400ns	1-10ms
Bandbreite (MB/s)	4000-32000	800-5000	400-2000	4-32
Verwaltet von	Compiler	Hardware	Betriebssystem	Betriebssystem
Backup Medium	Cache	Hauptspeicher	Platte	Magnetband

Wegen der Lokalität und wegen Smaller is faster können Speicherhierarchien signifikant die Performance erhöhen:

Beispiel:
Cache: 10 mal so schnell wie Hauptspeicher. Annahme, der Cache kann in 90% der Ausführungszeit benutzt werden. Wieviel gewinnen wir an Performance?

Wegen der Lokalität und wegen Smaller is faster können Speicherhierarchien signifikant die Performance erhöhen:

Beispiel:
Cache: 10 mal so schnell wie Hauptspeicher. Annahme, der Cache kann in 90% der Ausführungszeit benutzt werden. Wieviel gewinnen wir an Performance?

Amdahls law:

speedup = 1/(1-0,9+0,9/10) = 1/0,19 = 5,3

Mit Cache steigern wir die Performance um einen Faktor 5,3.

In der Regel können wir solche Aussagen wie *der Cache kann in 90% der Ausführungszeit benutzt werden* nicht treffen. Deshalb müssen wir die quantitative Analyse von Speicherhierarchien auf die CPU-Performance-Gleichung basieren. Dazu wird die Anzahl der Taktzyklen, die die CPU auf Hauptspeicherzugriffe warten muß mit in die Gleichung einbezogen:

Caches

Die Beschleunigung der Speicher hat zunehmende Bedeutung bekommen mit der rapiden Leistungssteigerung der Prozessoren.
Die Technologie der Speicherbausteine hat mit der Entwicklung der Prozessoren jedoch nicht ganz Schritt halten können.
Daher müssen die Rechnerarchitekten an dieser Stelle besonders viel Arbeit leisten, um für die schneller verarbeitenden Pipelines in hinreichender Geschwindigkeit Instruktionen und Daten liefern zu können.

Wir werden in diesem Kapitel eine Reihe von Techniken kennenlernen, die zu einer Leistungssteigerung des Speichersystems beitragen können. Zusätzlich bemühen wir uns wieder um einen Satz von quantitativen Werkzeugen, mit denen wir Verbesserungen gegenüber anderen Vorschlägen evaluieren und bewerten können.

CPU-Zeit = (CPU-Taktzyklen + Memory-stall-Zyklen)* Zykluslänge

Die Gleichung setzt voraus, daß die Cache Hits in den CPU-Taktzyklen enthalten sind, und daß die CPU steht bei einem Cache Miss. Die Anzahl der Memory-stall-Zyklen hängt ab von der Anzahl der Cache Misses und den Kosten pro Cache Miss der Miss-penalty.

Memory stall Zyklen = Anzahl Misses * Miss-penalty Zyklen

Wenn man den durchschnittlichen Wert der Misses pro Instruktion kennt, kann man diese Formel umstellen zu

Memory stall Zyklen = IC * Misses pro Instruktion * Miss-penalty Zyklen

Wenn man die durchschnittliche Rate der Misses pro Speicherzugriff kennt, kann man diese Formel umstellen zu

Memory stall Zyklen = IC * Speicherzugriffe pro Instruktion * Missrate * Miss-penalty Zyklen

Entscheidend ist dabei die Formel:

Memory stall Zyklen =
IC * Speicherzugriffe pro Instruktion * Miss rate * Miss penalty

Miss rate ist dabei der Anteil an Speicherzugriffen, die keinen Cache-Treffer erzielen.
Miss penalty ist die Anzahl der zusätzlichen Stauzyklen, die verstreichen, bis der Fehlzugriff behandelt ist.

Im Zusammenhang mit den unterschiedlichen Ebenen der Speicherhierarchie werden wir jeweils vier Fragen behandeln:

1. Wo kann ein Block plaziert werden (in der höheren Ebene)(Block placement)

2. Wie kann ein Block gefunden werden (in der höheren Ebene)(Block identification)

3. Welcher Block sollte ersetzt werden bei einem Miss? (Block replacement)

4. Was passiert beim Schreiben? (write strategie)

Wir werden die Konzepte, die hier vorgestellt werden, an konkreten Beispielen nachvollziehen, und sehen was jeweils dort umgesetzt wurde.

Wir beginnen mit Caches:
1. Frage: Wo kann ein Block im Cache untergebracht werden:

3 Strategien:
Voll assoziativ
Direct mapped
Mengenassoziativ

Die folgende Folie zeigt diese Möglichkeiten:

Voll assoziativ: Der Block kann überall im Cache stehen:

Direct mapped: Wenn der Block die Adresse m hat und der Cache k Blöcke fassen kann, dann kommt der Block an die Stelle m mod k im Cache.

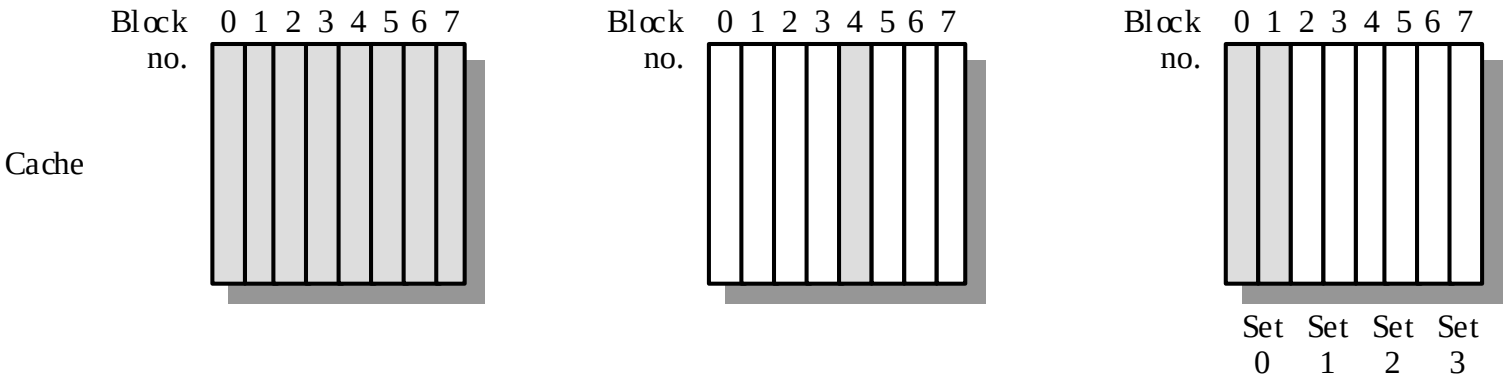
Mengenassoziativ: Wenn der Block die Adesse m hat und der Cache aus k Mengen von jeweils b Blöcken besteht, kann der Block an einer der b Stellen in der Menge mit der Adresse m mod k im Cache stehen. Man bezeichnet diese Situation als b-way-assoziativ.

Diese Typen sind gebräuchlich: direkt=1-Weg, 2-Weg, 4-Weg, 8-Weg, voll assoziativ

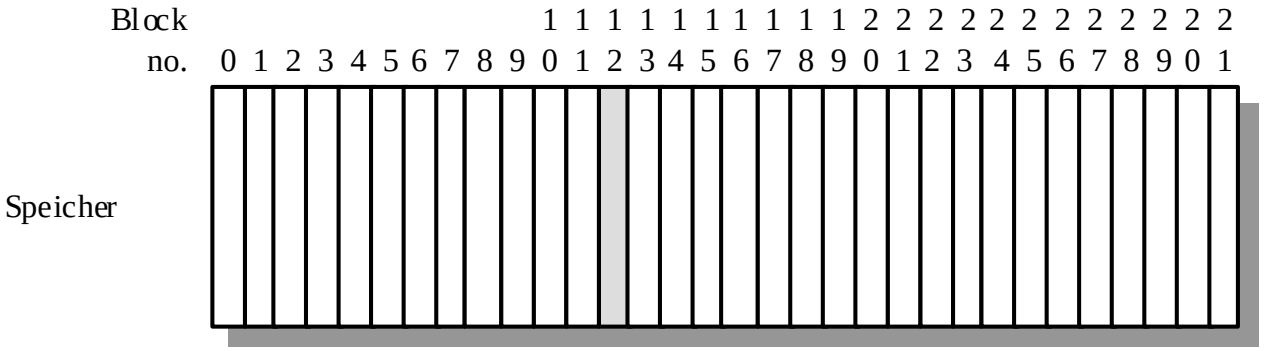
Voll assoziativ
Block 12 kann
überall stehen

Direct mapped:
Block 12 kann nur
an eine Stelle:
(12 mod 8)

2-Weg:
Block 12 kann
überall in Menge
0 stehen
(12 mod 4)

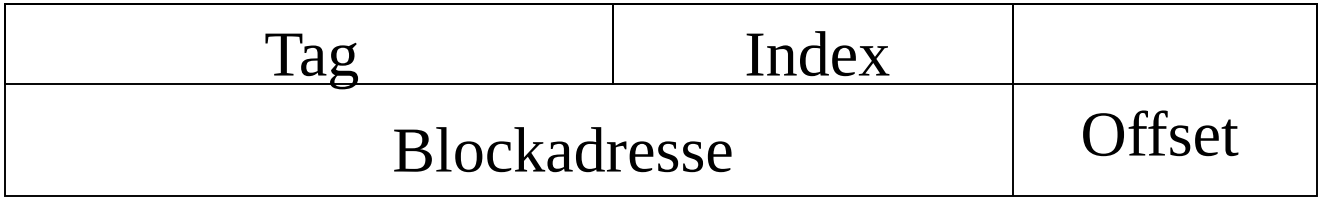


Block Rahmenadresse



2. Frage: Wie kann ein Block gefunden werden im Cache?

Jeder Block hat ein sogenanntes Tag (Erkennungszeichen) im **block frame**, der die Adresse des Blocks enthält. Das Tag wird verglichen mit dem entsprechenden Teil der Adresse desjenigen Datums, das jetzt gerade von der CPU gebraucht wird. Wenn mehrere Daten im Cache in Frage kommen (bei mehrWeg Assoziativität), werden alle Tags gleichzeitig überprüft.



Gesamtadresse

Offset: Die Stelle im Block, an der das gewünschte Datum steht.
Index: Diejenigen Bits, die bestimmen, wo im Cache das Datum stehen kann.
Bestimmt bei direct mapped den genauen Platz, bei set associative die Menge.
Tag: Diejenigen Bits, die mit denen der CPU-Adresse verglichen werden müssen, um zu bestimmen , ob es ein Hit oder ein Miss ist.

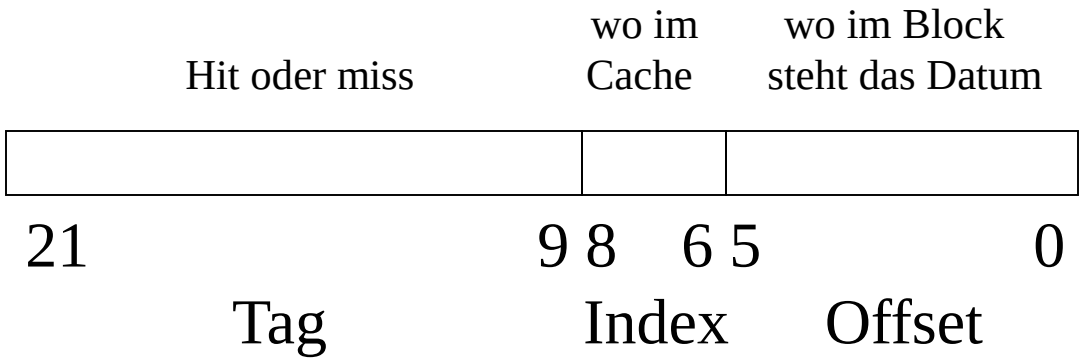
Beispiel:

Wir haben einen 2 Kbyte Cache mit Blöcken von 64 Byte. (insgesamt 32 Blöcke). Der Cache ist 4-Weg-assoziativ, d.h. wir haben acht Mengen: { 0,1,2,3 }, { 4,5,6,7 }, ..., { 28, 29,30,31 }. Unser Hauptspeicher ist 4 Mbyte groß, hat also 22 Adressbits.

Beispiel:

Wir haben einen 2 Kbyte Cache mit Blöcken von 64 Byte. (insgesamt 32 Blöcke). Der Cache ist 4-Weg-assoziativ, d.h. wir haben acht Mengen: { 0,1,2,3 }, { 4,5,6,7 }, ..., { 28, 29,30,31 }. Unser Hauptspeicher ist 4 Mbyte groß, hat also 22 Adressbits.

Dann kann eine Adresse vom Cache folgendermaßen interpretiert werden:



Ein voll assoziativer Cache hat kein Indexfeld. Ein direkt gamappter ein langes. Zusätzlich speichert der Cache noch für jeden Block ein sogenanntes valit-Bit. Dieses sagt aus, ob der Wert im Cache benutzt werden darf oder nicht.

2. Beispiel:

Unser Hauptspeicher ist 4 Gbyte groß.

Wir haben einen 16 Kbyte Cache mit Blöcken von 32 Byte. Der Cache direct mapped. Wie groß sind Tag, Index und Offset? An welcher Stelle im Cache wird das Byte mit der Adresse 0000 0000 0001 1010 1111 1010 0101 1101 im Cache stehen, wenn es sich beim Zugriff um einen Hit handelt? Wie lautet der Tag? Wo im Block steht das Byte?

3. Frage: Welcher Block sollte ersetzt werden?

Drei Strategien:

FIFO

Random

LRU: least recently used

Random ist von der Hardware her das einfachste zu bauen. Random ist besser als FIFO, LRU ist besser als random.

Die folgende Folie zeigt die Miss-Raten der Strategien Random und LRU bei verschiedenen Cache-Größen mit unterschiedlicher Assoziativität.

	Assoziativität					
	Zwei-Weg		Vier-Weg		Acht-Weg	
Größe	**LRU**	**Random**	**LRU**	**Random**	**LRU**	**Random**
16 KB	5.18%	5.69%	4.67%	5.29%	4.39%	4.96%
64 KB	1.88%	2.01%	1.54%	1.66%	1.39%	1.53%
256KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Miss-Raten bei verschiedenen Cache-Größen und unterschiedlicher Assoziativität

(Pentium 4, mix of user programs and os-programs)

4. Frage: Was passiert beim Schreiben?

Lesen passiert öfter als schreiben. Warum?

1. Weil jede arithmetische oder logische Operation 2 Operanden braucht aber nur ein Ergebnis erzeugt aber viel wichtiger

2. Weil jede Instruktion selbst aus dem Speicher gelesen werden muss, aber nur load und stores auf den Datenspeicher zugreifen.

Etwa 26% loads, etwa 9% stores. D.h. 9% / (100% + 26% + 9%) = 7% von allen Speicherzugriffen sind Schreibzugriffe.

Etwa 9% / (26% + 9%) = 25% aller Datenzugriffe sind Schreibzugriffe.

Make the common case fast lehrt uns also, Lesezugriffe besonders schnell zu machen, aber Amdahls Law ermahnt uns, die Schreibzugriffe nicht zu langsam zu machen, weil wir sonst die gewonnene Performance wieder verlieren.

Glücklicherweise ist der common case der einfachere.

Bei direct mapped kann der Block zur gleichen Zeit gelesen werden wie der Vergleich des Tags. Bei heutigen Prozessoren ist der so genannte spekulative Zugriff auf den Cache üblich: Erst erfolgt der Datenzugriff, erst einen Takt später kommt das Tag ok vom Cache. Wenn der Tag-check einen Fehlzugriff findet, wird ein Stau erzeugt.

Beim Schreiben ist das anders. Wir dürfen nicht einfach schreiben, bevor wir wissen, dass das Tag stimmt.
Deswegen dauert Schreiben in der Regel länger als Lesen.
Außerdem spezifiziert der Prozessor, wo im Block geschrieben werden muss. Wie viele Bytes, an welcher Stelle. Beim Lesen macht es nichts, wenn mehr als das erforderliche gelesen wird.

Es gibt zwei wesentlich unterschiedliche Strategien für das Zurückschreiben vom Cache in den Hauptspeicher:

write through: Jeder Schreibvorgang in den Cache wird auch gleich im Hauptspeicher vorgemnommen

write back: erst wenn der Block zurückgeschrieben wird, erfährt der Hauptspeicher von den Veränderungen. Wenn der Block ersetzt wird, wird er in den Speicher geschrieben.

Um Blöcke weniger häufig zurückschreiben zu müssen, wird im Cache jedem Block ein weiteres Bit **(dirty-bit**) hinzugefügt. Dieses wird gesetzt, wenn der block verändert wird. Wenn also der write-back Fall eintritt, aber das dirty-Bit ist nicht gesetzt, braucht nicht zurückgeschrieben zu werden, weil keine Veränderungen vorgenommen worden sind.

Vorteile von *write through*: Hauptspeicher ist immer aktuell, Read Misses bewirken kein zurückschreiben, einfacher zu implementieren.

Vorteile von *write back*: Weniger Speicherverkehr, Mehrere writes erzeugen nur einen Schreibzugriff im Hauptspeicher. Writes gehen mit der Geschwindigkeit des Caches.

Wenn die CPU warten muss auf ein write through, nennen wir das einen Write stall. Um Write-Stalls zu vermindern, benutzt man einen Write-Puffer, in dem der Cache den zu schreibenden Wert ablegt. Dieser schreibt dann in den Hauptspeicher, aber die CPU kann gleichzeitig schon weitermachen. Trotzdem kann es Write-Stalls geben, z.B. wenn der Write-Puffer voll ist.

Neue Daten werden beim Schreiben nicht gebraucht. Daher zwei Strategien:

fetch-on-write: Der Block, in den geschrieben worden ist, wird in den Cache geholt, weil man damit rechnet, daß gleich noch einmal hineingeschrieben wird. (Lokalität)

write-around: Der Block wird nicht geholt.

Beispiel: Ein typischer Level-1-Datencache

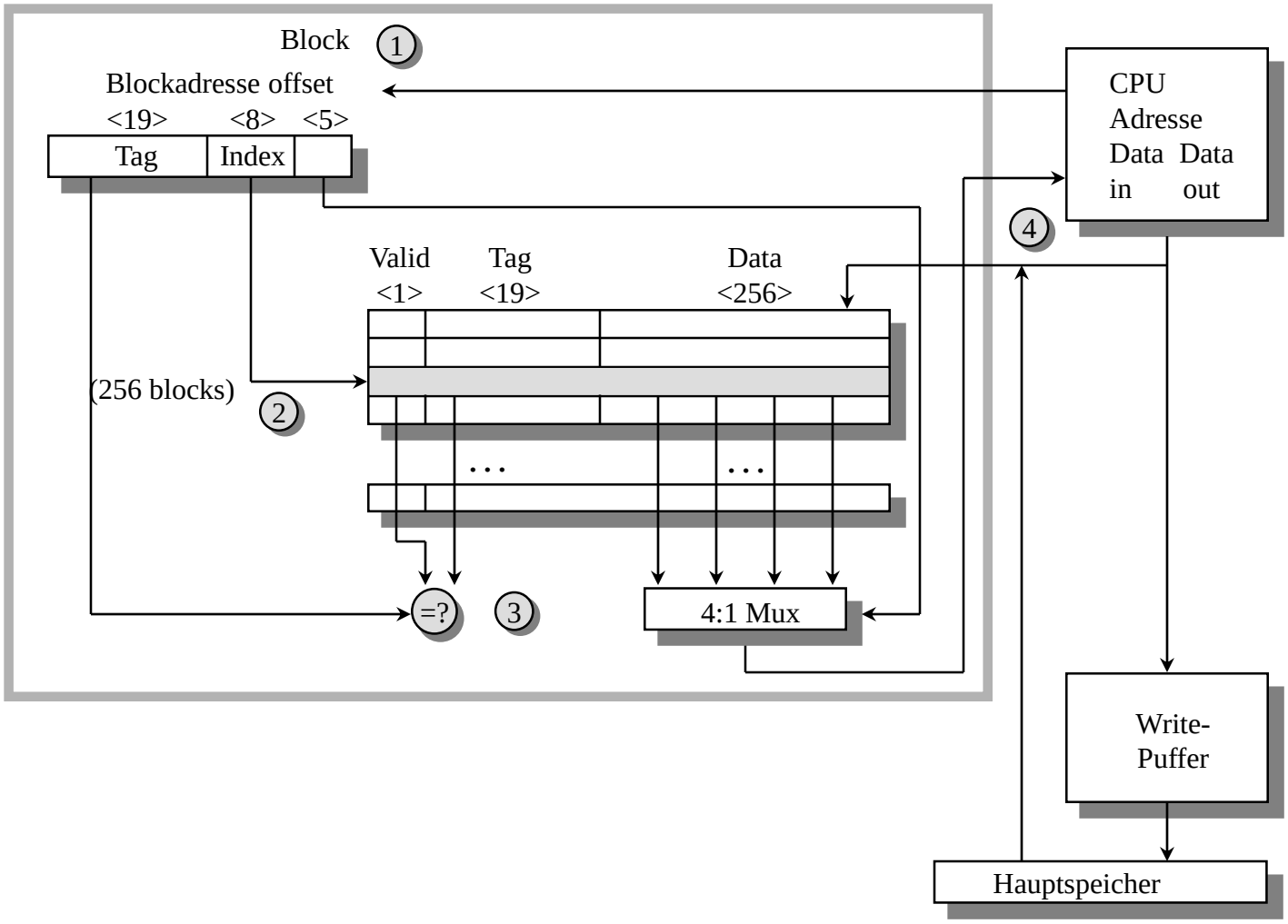
Datencache: 8KByte, in Blocks von jeweile 32 Byte, direct mapped, write through mit einem 4-Block write-Puffer, write-around beim write-miss.

Die folgende Folie zeigt, was gemacht wird für einen Zugriff mit Cache Hit. Die Gesamtadresse ist 32 Bit lang. 27 Bit Blockadresse, 5 Bit Offset. Die 27 Bit sind unterteilt in 19 bit Tag und 8 Bit Index.

Schritt 1 ist die Übermittlung der CPU-Adresse
Schritt 2 greift auf das Datum im Cache zu, das unter dem Index abgelegt ist.
Schritt 3 vergleicht den Tag und überprüft das valid Bit.
Schritt 4 gibt dem Prozessor das Datum und die Hit-Information.

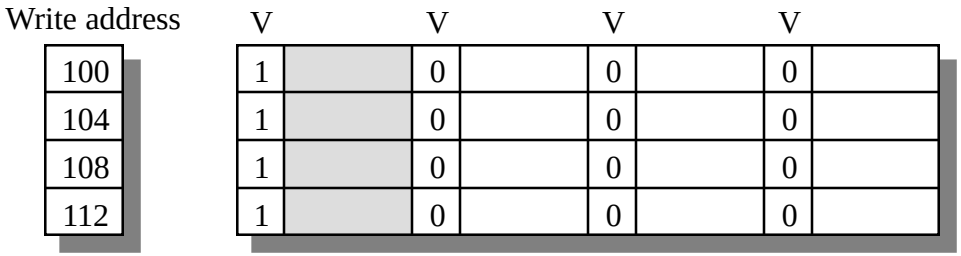
Der Zugriff dauert zwei Taktzyklen.

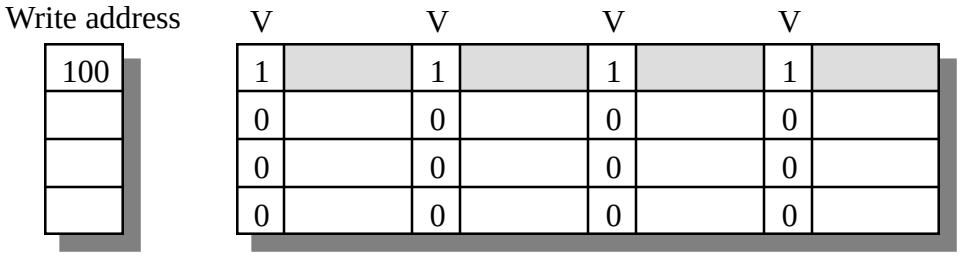
Der Multiplexer ist nur aus folgendem Grund vorhanden: ein maximal 64-Bit breiter Zugriff wird gemacht, wobei diese über zwei Bits des Offsets aus den Blöcken von 256 Bits ausgewählt werden.
Der 8-Bit-Index stellt sicher, dass der Wert immer richtig ist, wenn das tag stimmt.


Block
1
Blockadresse offset
<19>
<8>
<5>
Tag
Index
CPU
Adresse
Data in
Data out
4
Valid
<1>
Tag
<19>
Data
<256>
(256 blocks)
2
=?
3
4:1 Mux
Write-
Puffer
Hauptspeicher

Beim Schreiben sind die ersten drei Schritte dieselben. Wenn es ein Hit ist sowieso.
Dann muss der Wert (write through) in den Write-Puffer geschrieben werden. Der Schreib-Puffer enthält vier Blöcke à vier Worte.
Wenn der Schreibpuffer leer ist, schreibt die CPU Datum und volle Adresse in den Schreibpuffer. Damit kann die CPU weitermachen, während der Speicher den Wert korrigiert.
Wenn Der Puffer aber schon zu schreibende Daten enthält, müssen die Adressen mit dem neuen Datum verglichen werden, da sonst unnötige Schreibvorgänge den Bus und den Speicher verstopfen würden. Es muss geprüft werden, ob ein anderer Wert denselben Block zugreift oder sogar dasselbe Datum. Dann könnte ein Schreibvorgang eingespart werden.
Siehe nächste Folie.
Man spricht von write-merging.

Was passiert beim Cache miss? Beim Lesen bekommt die CPU ein Miss-Signal und die weitere Verarbeitung wird gestaut solange bis der erforderliche Block von 32 Byte im Cache vorhanden ist.
Dies dauert mindestens 10 Taktzyklen (miss penalty). Nachdem der neue Block an die einzig mögliche Stelle im Cache gewandert ist, wird das valid-Bit auf 1 und das dirty-Bit auf 0 gesetzt und es geht weiter.
Beim Schreiben braucht im Cache nichts zu passieren, der Wert wird write-around in den Write-Puffer geschrieben und dann genauso wie beim Hit.


Write address
V
V
V
V
100
104
108
112
1 0 0 0
1 0 0 0
1 0 0 0
1 0 0 0


Write address
V
V
V
V
100
1 1 1 1
0 0 0 0
0 0 0 0
0 0 0 0

Dies für die Daten. Der Prozessor hat außerdem einen fast identischen Instruction Cache. Ebenfalls 8 Kbyte.
Die CPU weiß, ob sie eine Instruktion oder ein Datum zugreifen will, also richtet sie ihre Anfrage an den richtigen Cache.
Manche Architekturen haben einen sogenannten **Unified Cache** (das ist ein in der Regel doppelt so großer Cache für Daten und Instruktionen)

In der folgenden Tabelle sieht man, dass Instruction Caches eine deutlich geringere Miss rate haben als Daten Caches. Das ist klar, weil Programme eben normalerweise Befehl für Befehl ausgeführt werden.

Size	Instruction cache	Data cache	Unified cache
1 KB	3.06%	24.61%	13.34%
2 KB	2.26%	20.57%	9.78%
4 KB	1.78%	15.94%	7.24%
8 KB	1.10%	10.19%	4.57%
16 KB	0.64%	6.47%	2.87%
32 KB	0.39%	4.82%	1.99%
64 KB	0.15%	3.77%	1.35%
128 KB	0.02%	2.88%	0.95%

6.2. Cache Performance

Um zu entscheiden, ob zwei unterschiedliche oder ein unified Cache verwendet werden sollte, braucht man Information über die Häufigkeit der Zugriffe. Die folgende Formel kann uns dann helfen:

Durchschnittliche Zugriffszeit = Hit Zeit + Miss rate * Miss penalty

Beispiel:

Was hat die geringere Miss rate, ein 32 KB unified Cache oder ein 16 KB Daten- und ein 16 KB-Instruction Cache, wenn die Miss Raten der Tabelle eben angenommen werden?
Wir nehmen an, ein Hit braucht 1 Zyklus und ein Miss weitere 50.
Ferner braucht ein load oder store Befehl einen zusätzliche Zyklus für einen Hit beim unified Cache, weil nur ein Speicherport als vorhanden vorausgesetzt wird.

Wie ist die durchschnittliche Speicherzugriffszeit in jedem der Fälle?

Nimm write-trough mit Write-Puffer an, und vernachlässige Staus durch den Write-Puffer.

Wie oben festgestellt, sind 75% der Zugriffe Instruktionsleseops.

Daher ist die durchschnittliche Miss-Rate für die aufgespaltenen Caches:

Durchschnittliche Miss-Rate = (75% * 0,64%) + (25% * 6,47%) = 2,10%

der unified Cache hat einen leicht niedrigere Miss-Rate mit 1,99%, wie wir aus der Tabelle ablesen können.

Für die durchschnittliche Zugriffszeit müssen wir in Instruction und Datenzugriffe unterteilen:

Durchschnittliche Zugriffszeit =
Instruktionsanteil * (Hit Zeit + Instruktions-miss rate * Miss penalty)+
Datenanteil * (Hit Zeit + Daten-miss rate * Miss penalty)=
75% * (1 + 0,64% * 50) + 25 %*(1 + 6,47% *50) =
75% * 1,32 + 25 % * 4,235 = 2,05

Beim unified Cache dagegen:
75% * (1 + 1,99%*50) + 25% * (2 + 1,99% * 50) =
75% * 1,995 + 25 % * 2,995 = 2,24

Wir sehen also: der aufgesplittete Cache ist besser, obwohl er eine geringfügig höhere durchschnittliche Miss-Rate hat.

Kommen wir zurück auf unsere CPU-Performance-Gleichung:

CPU-Zeit = IC * CPI * Zykluszeit

Unter Einbeziehung des Caches ergibt sich

CPU-Zeit = IC * ($CPI_{execution}$ + Speicherstauzyklen/Instruktion) * Zykluszeit

Beispiel:

Wir betrachten folgenden Prozessor: miss-penalty 10 Zyklen, Instruktionen brauchen normalerweise im Durchschnitt 2.0 Zyklen (nicht gerechnet Speicherstaus). Sei die miss rate 2% und die durchschnittliche Anzahl der Zugriffe pro Instruktion 1,33. Wie verändert sich die Performance, wenn der Cache mit berücksichtigt wird?

Beispiel:

Wir betrachten folgenden Prozessor mit unified Cache: miss-penalty 10 Zyklen, Instruktionen brauchen normalerweise im Durchschnitt 2.0 Zyklen (nicht gerechnet Speicherstaus). Sei die miss rate 2% und die durchschnittliche Anzahl der Zugriffe pro Instruktion 1,33. Wie verändert sich die Performance, wenn der Cache mit berücksichtigt wird?

Wir benutzen die obige Form der CPU-Performance-Gleichung:

CPU-Zeit = IC * (2.0 + 1,33 * 2% * 10) * Zykluszeit = IC * 2,266 * Zykuszeit.

Bemerkung: ohne die Speicherhierarchie wäre die CPU-Zeit

= IC * (2.0 + 1,33 * 10) * Zykluszeit = IC * 15,3 * Zykluszeit

also 7 mal schlechter!!!

Je kleiner die $CPI_{execution}$, desto größer der relative Einfluss einer festen Anzahl von Stauzyklen
Wenn wir die CPI berechnen, wird die Miss-Penalty in Zyklen pro Miss gerechnet. D.h. bei gleicher Speicherhierarchie wird ein Rechner mit höherer Taktfrequenz mehr Zyklen pro Miss brauchen.

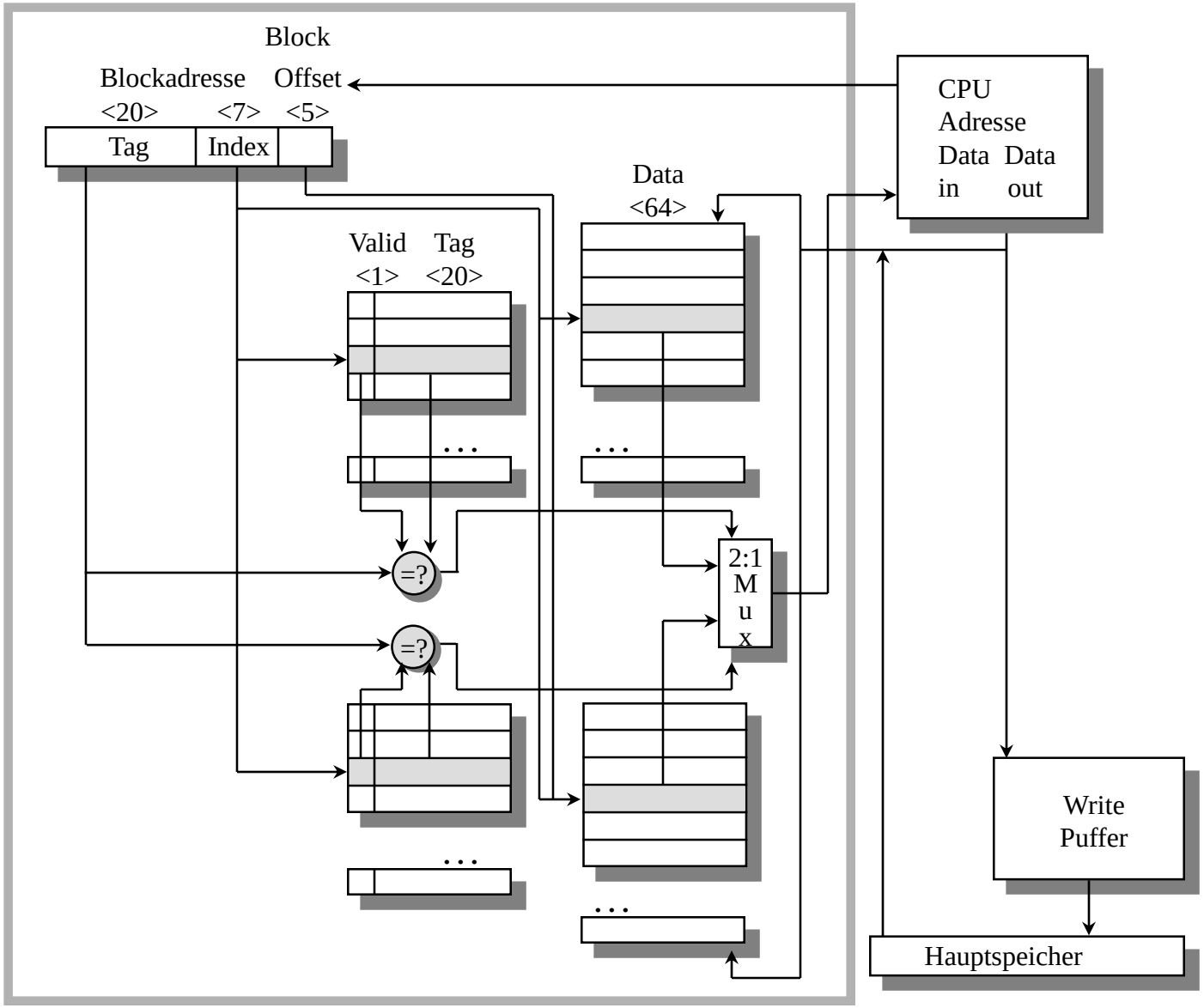
Beispiel:

Vergleichen Sie zwei verschiedene Cache Organisationen über die CPU-Performance:

CPI mit perfektem Cache 2.0. Zykluszeit 2ns. 1,3 Speicherzugriffe pro Instruktion. 64 Kbyte Cache mit 32 Byte Blöcken.
Ein Cache ist direct mapped der andere 2-Weg assoziativ. Auf der nächsten Folie sehen wir, daß der 2-Weg-assoziative Speicher einen zusätzlichen Mux baucht, der durch den Tag-Match gesteuert wird. Über diesen Mux wird das richtige der zwei möglichen Daten ausgewählt. Dadurch muß der Takt um einen Faktor 1.1 langsamer werden. Sei für eine erste Approximation die penalty für beide Caches 70ns (tatsächlich muß es ein ganzzahliges Vielfaches der Zykluszeit der jeweiligen Architektur sein).

Berechnen sie die durchschnittliche Zugriffszeit und die CPU-Performance.

Gehen Sie von einer Miss-Rate für den direkt gemappten Cache von 1,4% und für den Mengen-assoziativen Cache von 1% aus.


Block
Blockadresse
Offset
<20>
<7>
<5>
Tag
Index
CPU
Adresse
Data in
Data out
Data
<64>
Valid
<1>
Tag
<20>
=?
=?
2:1
M
u
x
Write
Puffer
Hauptspeicher

Antwort:

Durchschnittliche Zugriffszeit direct: 2ns + 1,4% * 70ns = 2,98ns
Durchschnittliche Zugriffszeit 2-Weg: 2,2ns + 1,0% * 70ns = 2,9ns

CPU-Performance direkt =
= IC * (2.0 + misses/ instruction * miss penalty) * Zykluszeit
= IC * (2.0 * Zykluszeit+ Zugriffe/instruktion * miss rate * miss penalty * Zykluszeit)
= IC * (2.0 * 2ns + 1,3 * 1,4% * 70ns) = IC * 5,27 ns

CPU-Performance 2-Weg =
= IC * (2.0 + misses/ instruction * miss penalty) * Zykluszeit
= IC * (2.0 * Zykluszeit+ Zugriffe/instruktion * miss rate * miss penalty * Zykluszeit)
= IC * (2.0 * 2,2ns + 1,3 * 1,0% * 70ns) = IC * 5,31 ns

Das heißt der direkt gemappte Cache ist 1.01 mal schneller als der 2-Weg Cache.

Der Grund liegt darin, daß der 2-Weg Cache mit seiner niedrigeren Taktfrequenz Auswirkungen auf alle Befehle hat, nicht nur auf die mit Speicherzugriff.

6.3. Verbesserung der Cache Performance

Durchschnittliche Zugriffszeit = Hit Zeit + Miss-rate * Miss Penalty

Daher drei Stragegien:

Verringerung der Miss Rate

Verringerung der Miss Penalty

Verringerung der Zeit für einen Hit

Cache misses fallen in eine von drei Kategorien:

1. Cold start misses: solche die dadurch auftreten, daß der erste Zugriff ein Miss sein muß. (Compulsory-misses)
2. Capacity misses: Wenn ein Cache nicht alle Blöcke gleichzeitig halten kann, die in einem Programm gebraucht werden, muß hin- und hergeladen werden.
3. Conflict : Wenn die Zuweisungsstrategie direkt oder set-assoziativ ist, kann es vorkommen, daß ein Block ausgelagert werden muß, weil sein Platz gebraucht wird.

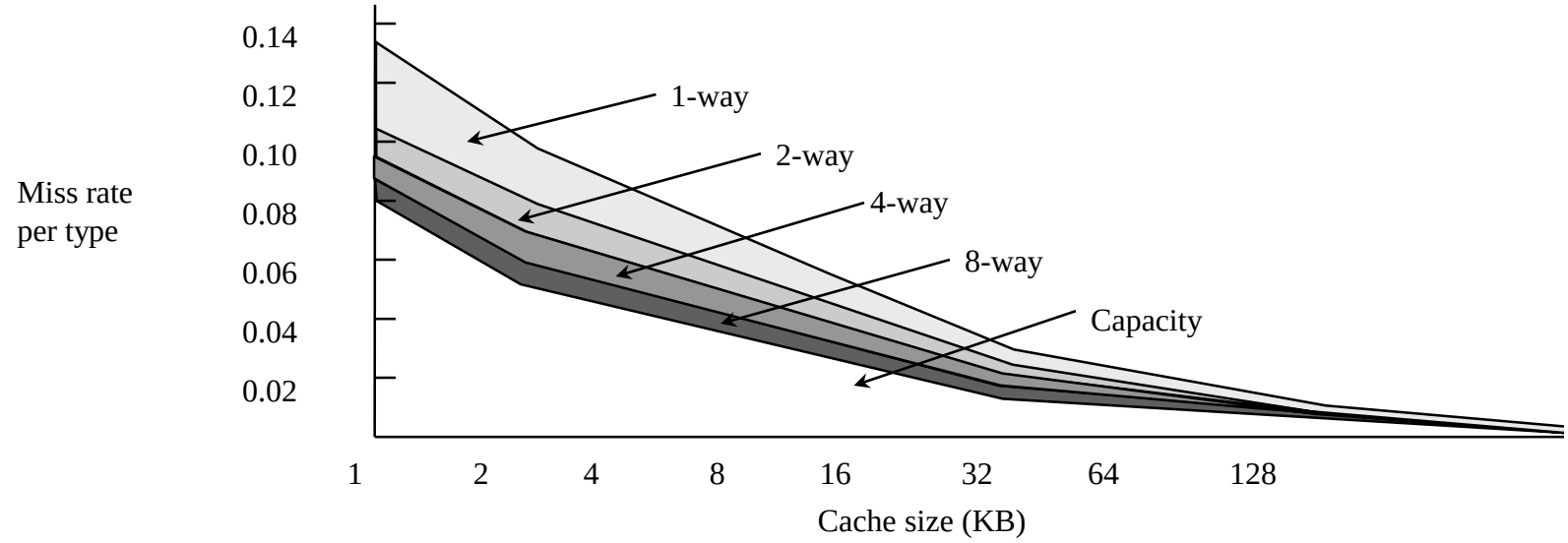
Die folgende Folie zeigt die Häufigkeit von Misses dieser drei Kategorien für unterschiedlich große Caches und unterschiedliche Zugriffsstrategien.

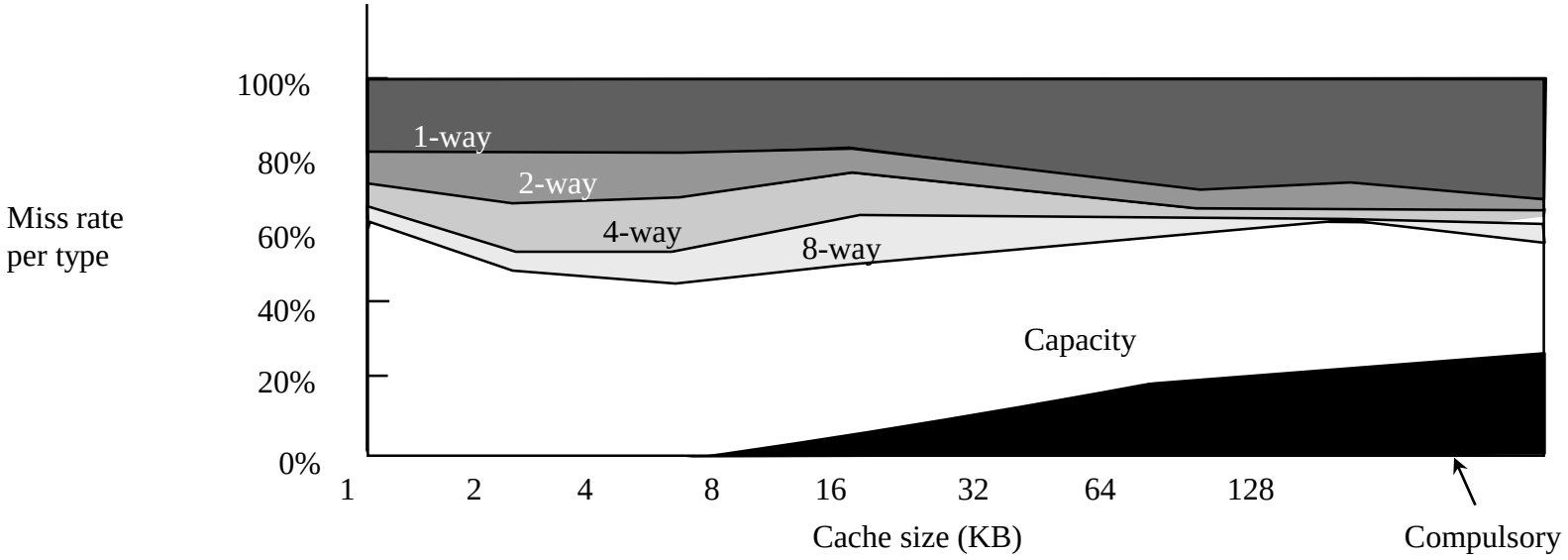
Um zu sehen, wie die Häufigkeit der Konflikt-Misses von der Assoziativität abhängt, werden vier Fälle:
8-Weg bis voll assoziativ,
4-Weg bis 8-Weg,
2-Weg-4-Weg,
direct bis 2-Weg
getrennt dargestellt.

Das gleiche wird auf der darauffolgenden Folie graphisch dargestellt:

Das obere Diagramm gibt die absoluten Miss-raten an, das untere die Miss-raten relativ zu einander.

Cache Größe	Assoziativität	Gesamt Miss-Rate	Miss-Raten Anteile					
			Compulsory		Capacity		Conflict	
1KB	1-Weg	0,133	0,002	1%	0,080	60%	0,052	39%
1KB	2-Weg	0,105	0,002	2%	0,080	76%	0,023	22%
1KB	4-Weg	0,095	0,002	2%	0,080	84%	0,013	14%
1KB	8-Weg	0,087	0,002	2%	0,080	92%	0,005	6%
2KB	1-Weg	0,098	0,002	2%	0,044	45%	0,052	53%
2KB	2-Weg	0,076	0,002	2%	0,044	58%	0,030	39%
2KB	4-Weg	0,064	0,002	3%	0,044	69%	0,018	28%
2KB	8-Weg	0,054	0,002	4%	0,044	82%	0,008	14%
4KB	1-Weg	0,072	0,002	3%	0,031	43%	0,039	54%
4KB	2-Weg	0,057	0,002	3%	0,031	55%	0,024	42%
4KB	4-Weg	0,049	0,002	4%	0,031	64%	0,016	32%
4KB	8-Weg	0,039	0,002	5%	0,031	80%	0,006	15%
8KB	1-Weg	0,046	0,002	4%	0,023	51%	0,021	45%
8KB	2-Weg	0,038	0,002	5%	0,023	61%	0,013	34%
8KB	4-Weg	0,035	0,002	5%	0,023	66%	0,010	28%
8KB	8-Weg	0,029	0,002	6%	0,023	79%	0,004	15%
16KB	1-Weg	0,029	0,002	7%	0,015	52%	0,012	42%
16KB	2-Weg	0,022	0,002	9%	0,015	68%	0,005	23%
16KB	4-Weg	0,020	0,002	10%	0,015	74%	0,003	17%
16KB	8-Weg	0,018	0,002	10%	0,015	80%	0,002	9%
32KB	1-Weg	0,020	0,002	10%	0,010	52%	0,008	38%
32KB	2-Weg	0,014	0,002	14%	0,010	74%	0,002	12%
32KB	4-Weg	0,013	0,002	15%	0,010	79%	0,001	6%
32KB	8-Weg	0,013	0,002	15%	0,010	81%	0,001	4%
64KB	1-Weg	0,014	0,002	14%	0,007	50%	0,005	36%
64KB	2-Weg	0,010	0,002	20%	0,007	70%	0,001	10%
64KB	4-Weg	0,009	0,002	21%	0,007	75%	0,000	3%
64KB	8-Weg	0,009	0,002	22%	0,007	78%	0,000	0%
128KB	1-Weg	0,010	0,002	20%	0,004	48%	0,004	40%
128KB	2-Weg	0,007	0,002	29%	0,004	58%	0,001	14%
128KB	4-Weg	0,006	0,002	31%	0,004	61%	0,001	8%
128KB	8-Weg	0,006	0,002	31%	0,004	62%	0,000	7%


Miss rate per type
0.14
0.12
0.10
0.08
0.06
0.04
0.02
1-way
2-way
4-way
8-way
Capacity
1
2
4
8
16
32
64
128
Cache size (KB)


Miss rate per type
100%
80%
60%
40%
20%
0%
1-way
2-way
4-way
8-way
Capacity
1
2
4
8
16
32
64
128
Cache size (KB)
Compulsory

Man sieht, die Kaltstartmisses stellen bei allen langen Programmen nur einen kleinen Anteil.

Was kann man nun tun, um die Miss-Raten zu verringern?
Offenbar verringert man Konflikt-Misses durch höhere Assoziativität, die aber von der Hardwarerealisierung her teuer ist.
Capacity-Misses können durch größere Caches verringert werden.

Wir werden uns jetzt einige Techniken ansehen, die an der Verbesserung der Miss-Raten arbeiten.

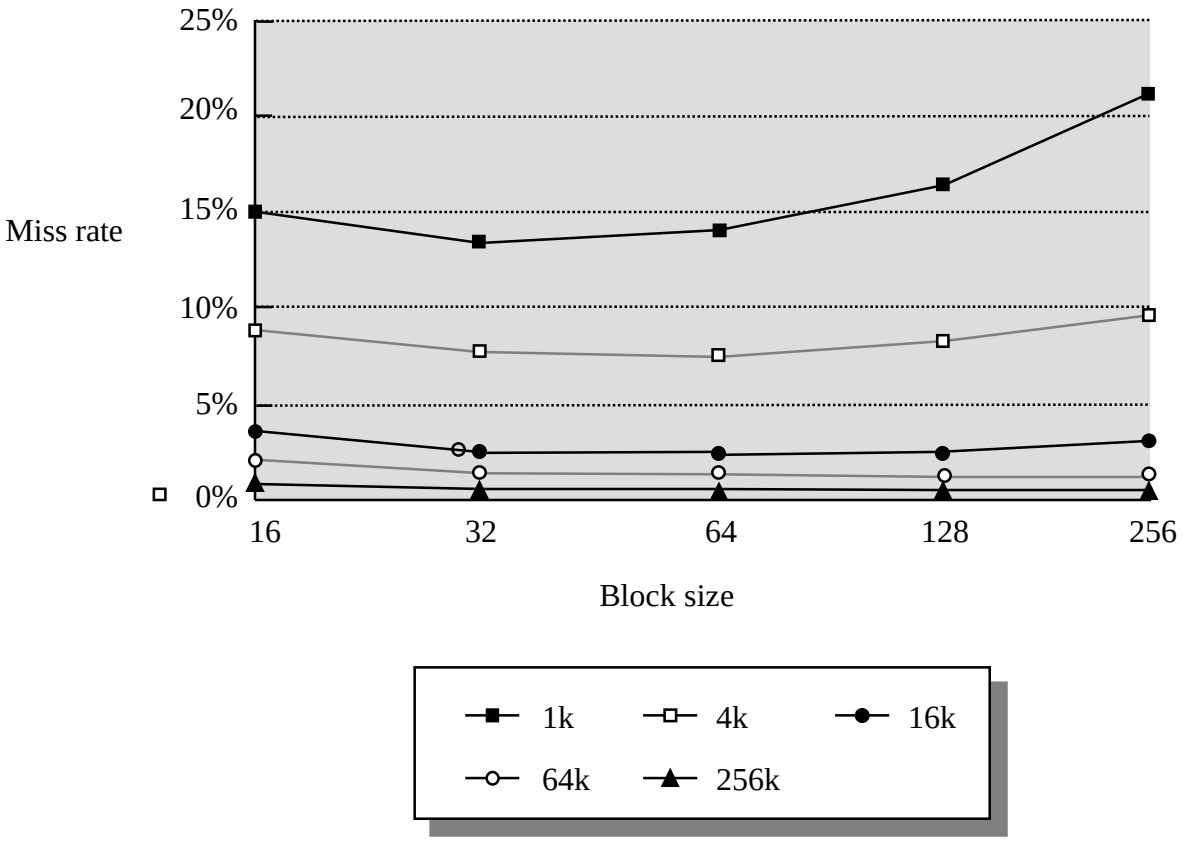
Erste Technik:
Größere Blockgrößen

Die einfachste Technik der Verringerung der Anzahl der Misses ist die Vergrößerung der Blöcke. Man profitiert dabei vom Prinzip der **räumlichen Lokalität**.

Andererseits **erhöht** die vergrößerte Blockgröße die **Miss penalty**, denn es müssen mehr Bytes pro Miss aus dem Speicher geholt werden.

Ferner können größere Blöcke die Häufigkeit der Konflikt-Misses und letztlich sogar die der Kapazitäts-Misses erhöhen, wenn der Cache klein ist. Es macht keinen Sinn, die Blockgrößen zu erhöhen, wenn die Miss-Rate dadurch steigt.

Die folgende Folie zeigt die Wechselwirkung von Blockgröße zu Miss-Rate. Jede Kurve repräsentiert eine Cachegröße. Grundlage sind wieder die SPEC92 Programme.


25%
20%
15%
10%
5%
0%
Miss rate
16
32
64
128
256
Block size
1k
4k
16k
64k
256k

	Cache size				
Block size	**1K**	**4K**	**16K**	**64K**	**256K**
16	15.05%	8.57%	3.94%	2.04%	1.09%
32	13.34%	7.24%	2.87%	1.35%	0.70%
64	13.76%	7.00%	2.64%	1.06%	0.51%
128	16.64%	7.78%	2.77%	1.02%	0.49%
256	22.01%	9.51%	3.29%	1.15%	0.49%

Beispiel:

Wir beziehen uns auf die Daten in obiger Tabelle.
Angenommen, der Speicher braucht 40 Zyklen an Overhead und liefert dann 16 Bytes pro 2 Zyklen, d.h. die ersten 16 Bytes brauchen 42 Zyklen, 32 Bytes 44 Zyklen, ...,256 Bytes 72 Zyklen.
Welche Blockgröße liefert die minimale durchschnittliche Zugriffszeit für die betrachteten Cache-Größen?

Beispiel:

Wir beziehen uns auf die Daten in obiger Tabelle.
Angenommen, der Speicher braucht 40 Zyklen an Overhead und liefert dann 16 Bytes pro 2 Zyklen, d.h. die ersten 16 Bytes brauchen 42 Zyklen, 32 Bytes 44 Zyklen, ...,256 Bytes 72 Zyklen.
Welche Blockgröße liefert die minimale durchschnittliche Zugriffszeit für die betrachteten Cache-Größen?

Durchschittliche Zugriffszeit = Hit Zeit + Miss-rate * Miss penalty

Also z.B. Blockgröße 16, 1K-Cache
DZZ = 1 + (0,1505 * 42) = 7,321 Zyklen

oder
Blockgröße 256, 16 K-Cache
DZZ = 1 + (0,0049 * 72) = 1,353 Zyklen

Es ergibt sich folgende Tabelle:

		Cache size				
Block size	**Miss penalty**	**1K**	**4K**	**16K**	**64K**	**256K**
16	42	7.321	4.599	2.655	1.857	1.458
32	44	**6.870**	**4.186**	**2.263**	1.594	1.308
64	48	7.605	4.360	2.267	**1.509**	**1.245**
128	56	10.318	5.357	2.551	1.571	1.274
256	72	16.847	7.847	3.369	1.828	1.353

Wie bei all diesen Techniken versucht der Architekt die Gesamtwirkung zu optimieren:

Blockgrößenauswahl hängt ab von der Speicherbandbreite und der Zugriffszeit des Hauptspeichers.

Große Bandbreite und schnelle Zugriffszeit ermöglicht größere Blöcke (da die penalty klein ist im Vergleich zur gewonnenen Miss-Rate)

Umgekehrt: kleine Bandbreite oder langsamer Zugriff führen zu kleineren Blöcken.

Zweite Technik:
Höhere Assoziativität

Wir haben auf den Diagrammen gesehen, daß höhere Assoziativität zur Verringerung der Miss-Rate beiträgt.

Dabei können zwei Beobachtungen zu nützlichen Daumenregeln werden:
1. 8-Weg Assoziativität ist in der Performance fast genau so gut wie volle Assoziativität.
2. Direkt mapped Cache der Größe N ist etwa so gut wie ein 2-Weg-Cache der Größe N/2

Wiederum gibt es eine Wechselwirkung zwischen Miss-Rate und Miss-Penalty, wie das folgende Beispiel zeigt:

Beispiel:

Nehmen wir an, bei Erhöhung der Assoziativität steigt die Zykluszeit in der folgenden Weise:

$$\textbf{Zykluszeit}_{\textbf{2-Weg}} = 1{,}10 * \textbf{Zykluszeit}_{\textbf{direkt}}$$
$$\textbf{Zykluszeit}_{\textbf{4-Weg}} = 1{,}12 * \textbf{Zykluszeit}_{\textbf{direkt}}$$
$$\textbf{Zykluszeit}_{\textbf{8-Weg}} = 1{,}14 * \textbf{Zykluszeit}_{\textbf{direkt}}$$

nehmen wir ferner einen Zyklus für einen Hit an und eine Miss-penalty von 50 Zyklen für den direkt gemappten Fall. Ferner fordern wir der Einfachheit halber nicht, dass die Miss-Penalty auf ein ganzzahliges Vielfaches der Zykuszeit gerundet wird.

Mit den Werten aus der folgenden Tabelle (die wir bereits aus dem Abschnitt über die drei Arten von Cache-Misses gesehen haben): für welche Cache-Größen gelten folgende drei Ungleichungen?

$$\textbf{Durchschnittliche Zugriffszeit}_{\textbf{8-Weg}} < \textbf{Durchschnittliche Zugriffszeit}_{\textbf{4-Weg}}$$
$$\textbf{Durchschnittliche Zugriffszeit}_{\textbf{4-Weg}} < \textbf{Durchschnittliche Zugriffszeit}_{\textbf{2-Weg}}$$
$$\textbf{Durchschnittliche Zugriffszeit}_{\textbf{2-Weg}} < \textbf{Durchschnittliche Zugriffszeit}_{\textbf{1-Weg}}$$

Durchschnittliche Zugriffszeit$_{8\text{-Weg}}$ < Durchschnittliche Zugriffszeit$_{4\text{-Weg}}$
Durchschnittliche Zugriffszeit$_{4\text{-Weg}}$ < Durchschnittliche Zugriffszeit$_{2\text{-Weg}}$
Durchschnittliche Zugriffszeit$_{2\text{-Weg}}$ < Durchschnittliche Zugriffszeit$_{1\text{-Weg}}$

Antwort:

Durchschnittliche Zugriffszeit = (1 + miss rate * miss penalty) * Zykluszeit

z.B. 1 KB Cache:

$$\text{Durchschnittliche Zugriffszeit}_{1\text{-Weg}} = (1 + 0{,}133 * 50) * \text{Zykluszeit}_{1\text{-Weg}}$$
$$= 7{,}65 * \text{Zykluszeit}_{1\text{-Weg}}$$

Da der Hauptspeicher für höhere Assoziativität genauso schnell ist, müssen wir die Formel auf folgende Weise benutzen:

$$\text{Durchschnittliche Zugriffszeit}_{2\text{-Weg}} = 1*\text{Zykluszeit}_{2\text{-Weg}} + 0{,}105 * (50 * \text{Zykluszeit}_{1\text{-Weg}})$$
$$= 1{,}10 * \text{Zykluszeit}_{1\text{-Weg}} + 5{,}25 * \text{Zykluszeit}_{1\text{-Weg}}$$
$$= 6{,}35 * \text{Zykluszeit}_{1\text{-Weg}}$$

	Associativity			
Cache size (KB)	**One-way**	**Two-way**	**Four-way**	**Eight-way**
1	7.65	6.35	5.87	5.49
2	5.90	4.90	4.32	3.84
4	4.60	3.95	3.57	3.09
8	3.30	3.00	2.87	2.59
16	2.45	2.20	2.12	2.04
32	2.00	1.80	1.77	**1.79**
64	1.70	1.60	1.57	**1.59**
128	1.50	1.45	1.42	**1.44**

Durchschnittliche Zugriffszeiten für unterschiedliche Assoziativität und unterschiedliche Cache-Größen

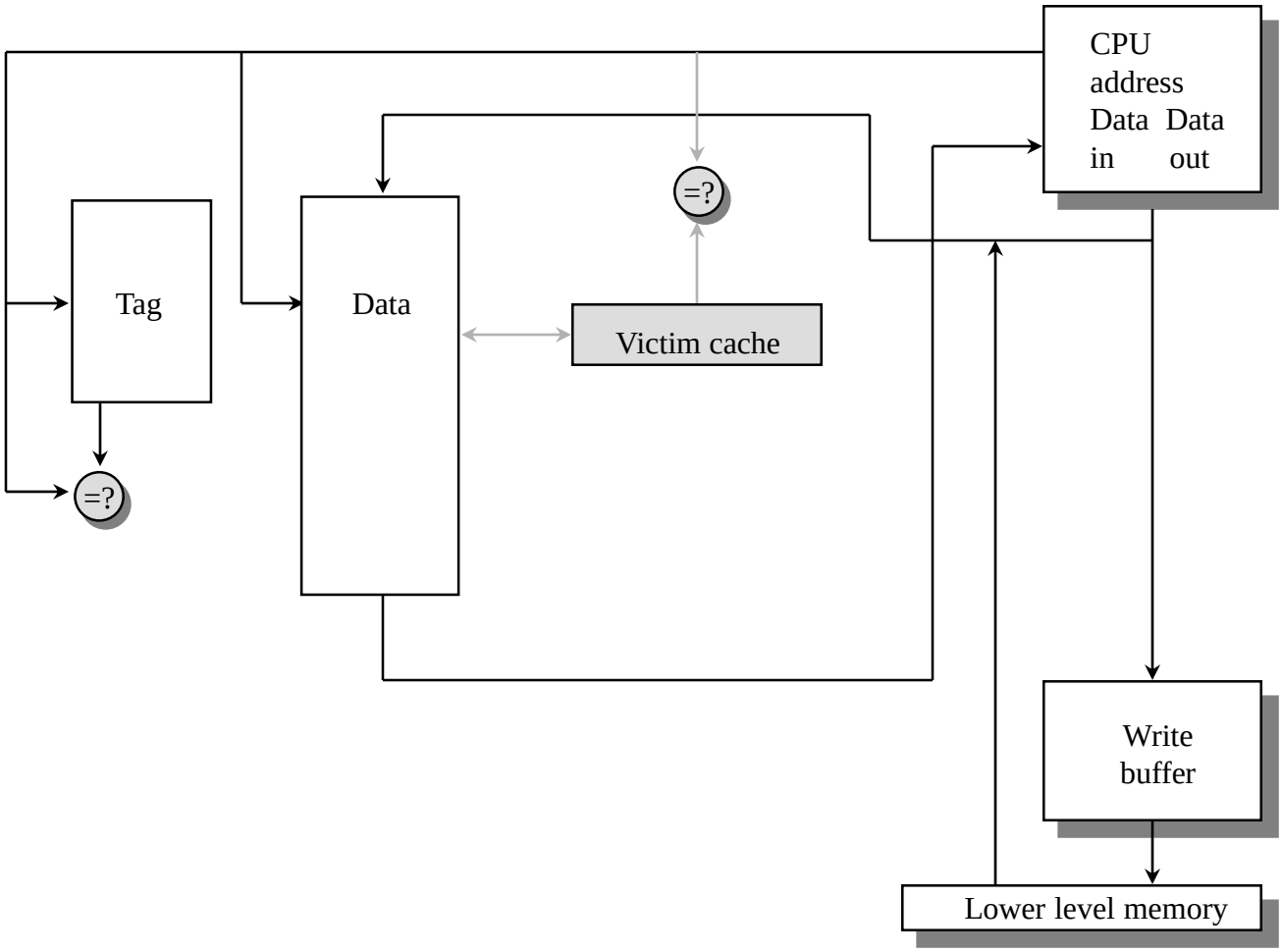
Dritte Technik:

Victim Caches

Auf den Weg zwischen Cache und Hauptspeicher wird ein kleiner voll assoziativer Cache eingebaut, der nur diejenigen Blöcke aufnimmt, die gerade hinausgeflogen sind. Bei einem Zugriff wird dieser Victim Cache parallel mit überprüft, und wenn das Datum gefunden wird, wird es mit dem Datum an der entsprechenden Stelle im Cache vertauscht.
Architektur sie folgende Folie

Victim Caches eignen sich zur Verringerung der Miss-Rate bei direkt gemappten und kleinen Caches.

Sie **reduzieren die Konflikt-Misses**.


CPU
address
Data in
Data out
=?
Tag
Data
Victim cache
=?
Write
buffer
Lower level memory

Vierte Technik

Pseudo-assoziative Caches

Ein direkt mapped Cache wird auf eine andere Weise betrieben: erst normal, bei einem Cache-Miss wird ein zweiter Platz möglich, z. B. der mit dem invertierten signifikantesten Bit des Index.

Beispiel:

	Tag	**Index**	**Offset**
Adresse	**10110101**	**1101**	**1110110**

Ich suche zuerst unter 1101 und vergleiche das Tag. Wenn ich fündig werde: Hit erster Klasse
Wenn nicht, suche ich unter 0101 und vergleiche das Tag. Fündig -> Hit zweiter Klasse.

Also gibt es einen schnellen und einen langsamen Hit, aber der langsame ist viel schneller als die Miss-Penalty.

Wir profitieren dabei davon, daß die Miss-Penalty nicht steigt gegenüber direkt, aber dagegen die Hit Zeit beim pseudo-Hit länger ist.

Beispiel:

Angenommen, es kostet zwei zusätzliche Zyklen, den Eintrag in der alternativen Position zu suchen. Mit den Werten des letzten Beispiels (DZZ für unterschiedliche Assoziativität und unterschiedliche Cache-Größen), welcher der drei Typen direkt, 2-Weg, pseudo-assoziativ ist der beste im Falle von 2KByte Cachegröße bzw. 128 KByte?

Antwort:

Durchschnittliche Zugriffszeit$_{direkt}$ = 5,9
Durchschnittliche Zugriffszeit$_{2\text{-}Weg}$ = 4,9
Durchschnittliche Zugriffszeit$_{pseudo}$ = Hit Zeit$_{pseudo}$ +miss rate$_{pseudo}$ *miss penalty$_{pseudo}$
Beginnen wir mit dem letzten Teil der Summe:

miss rate$_{pseudo}$ = miss rate$_{2\text{-}Weg}$

miss penalty$_{pseudo}$ = miss penalty$_{direkt}$

Hit Zeit$_{pseudo}$ = Hit Zeit$_{direkt}$ + Alternativhit rate$_{pseudo}$* 2

Die Hit rate für den Zweiten Versuch ist die Differenz zwischen der Hit-rate 2-Weg und der Hit-rate direkt:

Alternativhit rate$_{pseudo}$ = Hit rate$_{2\text{-}weg}$ - Hit rate$_{direkt}$

= (1 - miss rate$_{2\text{-}Weg}$) - (1 - miss rate$_{direkt}$)

= miss rate$_{direkt}$ - miss rate$_{2\text{-}Weg}$

Das ergibt zusammen:

Durchschnittliche Zugriffszeit$_{pseudo}$ = Hit Zeit$_{direkt}$ + (miss rate$_{direkt}$ - miss rate$_{2\text{-}Weg}$) * 2 + miss rate$_{2\text{-}Weg}$ * miss penalty$_{direkt}$

Mit der Tabelle auf der folgenden Folie (die wir bereits kennen) können wir einsetzen:

Durchschnittliche Zugriffszeit$_{pseudo\ 2K}$ = 1+(0,098-0,076) *2 + 0,076*50
= 1 + 0,022*2 + 3,8 = 4,844 Zyklen

Durchschnittliche Zugriffszeit$_{pseudo\ 128K}$ = 1+(0,010-0,007) *2 + 0,007*50
= 1 + 0,003*2 + 0,35 = 1,356 Zyklen

Durchschnittliche Zugriffszeit$_{direkt2K}$ = 5,9 Zyklen
Durchschnittliche Zugriffszeit$_{2\text{-}Weg2K}$ = 4,9 Zyklen

Durchschnittliche Zugriffszeit$_{direkt128K}$ = 1,5 Zyklen
Durchschnittliche Zugriffszeit$_{2\text{-}Weg128K}$ = 1,45 Zyklen

Das waren die Werte aus dem letzten Beispiel. Also Pseudo ist der beste in beiden Fällen

Cache Größe	Assoziativität	Gesamt Miss-Rate	Miss-Raten Anteile					
			Compulsory		Capacity		Conflict	
1KB	1-Weg	0,133	0,002	1%	0,080	60%	0,052	39%
1KB	2-Weg	0,105	0,002	2%	0,080	76%	0,023	22%
1KB	4-Weg	0,095	0,002	2%	0,080	84%	0,013	14%
1KB	8-Weg	0,087	0,002	2%	0,080	92%	0,005	6%
2KB	1-Weg	0,098	0,002	2%	0,044	45%	0,052	53%
2KB	2-Weg	0,076	0,002	2%	0,044	58%	0,030	39%
2KB	4-Weg	0,064	0,002	3%	0,044	69%	0,018	28%
2KB	8-Weg	0,054	0,002	4%	0,044	82%	0,008	14%
4KB	1-Weg	0,072	0,002	3%	0,031	43%	0,039	54%
4KB	2-Weg	0,057	0,002	3%	0,031	55%	0,024	42%
4KB	4-Weg	0,049	0,002	4%	0,031	64%	0,016	32%
4KB	8-Weg	0,039	0,002	5%	0,031	80%	0,006	15%
8KB	1-Weg	0,046	0,002	4%	0,023	51%	0,021	45%
8KB	2-Weg	0,038	0,002	5%	0,023	61%	0,013	34%
8KB	4-Weg	0,035	0,002	5%	0,023	66%	0,010	28%
8KB	8-Weg	0,029	0,002	6%	0,023	79%	0,004	15%
16KB	1-Weg	0,029	0,002	7%	0,015	52%	0,012	42%
16KB	2-Weg	0,022	0,002	9%	0,015	68%	0,005	23%
16KB	4-Weg	0,020	0,002	10%	0,015	74%	0,003	17%
16KB	8-Weg	0,018	0,002	10%	0,015	80%	0,002	9%
32KB	1-Weg	0,020	0,002	10%	0,010	52%	0,008	38%
32KB	2-Weg	0,014	0,002	14%	0,010	74%	0,002	12%
32KB	4-Weg	0,013	0,002	15%	0,010	79%	0,001	6%
32KB	8-Weg	0,013	0,002	15%	0,010	81%	0,001	4%
64KB	1-Weg	0,014	0,002	14%	0,007	50%	0,005	36%
64KB	2-Weg	0,010	0,002	20%	0,007	70%	0,001	10%
64KB	4-Weg	0,009	0,002	21%	0,007	75%	0,000	3%
64KB	8-Weg	0,009	0,002	22%	0,007	78%	0,000	0%
128KB	1-Weg	0,010	0,002	20%	0,004	48%	0,004	40%
128KB	2-Weg	0,007	0,002	29%	0,004	58%	0,001	14%
128KB	4-Weg	0,006	0,002	31%	0,004	61%	0,001	8%
128KB	8-Weg	0,006	0,002	31%	0,004	62%	0,000	7%

Fünfte Technik

Hardware prefetching von Daten und Instruktionen

Dies ist eine Technik, die sich besonders für Instruktions-Caches auszahlt.
Sie wird allen modernen Prozessoren benutzt.

Bei einem Miss werden zwei Blöcke geholt: der erforderliche und der nächste. Der richtige kommt in den Cache, der andere in den **Instruction Stream Puffer**: Dieser ist dem Cache angegliedert. Bei einem erneuten Miss wird zuerst der Instruction Stream Puffer überprüft und beim Treffer dieser in den Cache übertragen und ein neuer Block (der Nächste) geprefetcht.

Ein einzelner Block in einem Instruction Stream Puffer kann bereits zu einer Verringerung der Misses um 15-25% führen.

Die Idee ist ähnlich den victim caches.

Beispiel:

Was ist die Miss-Rate des Prozessors von oben mit Instruction-Prefetching?
Dabei sei die prefetch hit rate 25%.
Wieviel größer müsste der Instruction-Cache sein, um die gleiche Performance zu erzielen, wenn Prefetching nicht genutzt würde?

Antwort:

Nehmen wir einen zusätzlichen Takt an für den Prefetch-Treffer, wenn der Cache die Instruktion nicht enthält.

Durchschnittliche Zugriffszeit$_{prefetch}$ = Hit Zeit + Miss-Rate * prefetch Hit-Rate * 1 + Miss-Rate * (1 - Prefetch-Hit-Rate) * Miss-Penalty

Die folgende Folie (die wir bereits kennen) gibt uns als Miss-Rate für einen 8KByte Cache 1,10%. Mit 2 Zyklen Hit Zeit und 50 Zyklen Penalty ergibt sich:

Durchschnittliche Zugriffszeit$_{prefetch}$ = 2 + 1,1 * 25% * 1 + 1,1 * (1 - 25%) * 50
= 2,415

Um die effektive miss rate zu ermitteln, lösen wir die Ursprungsformel:

Durchschnittliche Zugriffszeit = Hit Zeit + miss rate * miss penalty

nach miss rate auf:

miss rate = (Durchschnittliche Zugriffszeit -Hit Zeit) / miss penalty

= (2,415 - 2) / 50 = 0,83%

Also ist die effektive miss rate mit prefetching 0,83%.
Jetzt schauen wir in der Tabelle nach (nächst Folie, Beziehung von Cache Größen und miss raten, kennen wir schon).

Dort liegt 0,83% zwischen 0,64% für einen 16 Kbyte Cache und 1,10% für einen 8 KB Cache.

Size	Instruction cache	Data cache	Unified cache
1 KB	3.06%	24.61%	13.34%
2 KB	2.26%	20.57%	9.78%
4 KB	1.78%	15.94%	7.24%
8 KB	1.10%	10.19%	4.57%
16 KB	0.64%	6.47%	2.87%
32 KB	0.39%	4.82%	1.99%
64 KB	0.15%	3.77%	1.35%
128 KB	0.02%	2.88%	0.95%