Rechnerarchitektur (RA)

Sommersemester 2020

Verbesserung der Leistungsfähigkeit von Caches

Jian-Jia Chen
Informatik 12
jian-jia.chen@tu-..
http://ls12-www.cs.tu-dortmund.de/daes/
Tel.: 0231 755 6078

Cache-Performanz (1)

- *CPU-Ausführungszeit* =
  (# *CPU-Takte* + #*Speicher-Wartezyklen*) x *Taktzeit*
- Einbeziehung von *Speicher-Wartezyklen* (*memory stall cycles*) = Takte, in denen CPU auf Speicher wartet
- Annahmen:
  - *CPU-Takte* enthalten Zeit für erfolgreichen Cache-Zugriff
  - *Speicher-Wartezyklen* treten nur bei Fehlzugriffen auf

Cache-Performanz (2)

Speicher-Wartezyklen sind abhängig von:

- Anzahl der Fehlzugriffe (*cache misses*) und
- „Kosten" (Zeitverlust) pro *cache miss* (= *miss penalty*)
- Anzahl der Befehle *(Instruction count) IC*

Speicher-Wartezyklen

= # *Cache misses* x *miss penalty*
= *IC* x (*misses / Befehl*) x *miss penalty*
= *IC* x (# *Speicherzugriffe / Befehl*) x
x *Fehlzugriffsrate* x *miss penalty*

- # *Speicherzugriffe / Befehl* > 1, da immer 1x Befehlsholen
- *miss penalty* ist eigentlich Mittelwert
- Alle Größen sind messbar (ggf. durch Simulation)

Cache-Performanz (3)

Bewertungsmaß für die Leistungsfähigkeit einer Speicherhierarchie:

$$\textit{Mittlere Zugriffszeit} = \textit{Hit Time} + \textit{Miss Rate} \times \textit{Miss penalty}$$

- *Hit Time* = Zeit für erfolgreichen Zugriff (in CPU-Leistungsgleichung Teil der „normalen" Ausführungszeit)
- *Miss Rate* = Anteil der Fehlzugriffe
- *Miss Penalty* = Wartezeit der CPU auf Speicher, die pro Fehlzugriff entsteht

Maßeinheiten: physikalische Zeit oder Taktzyklen möglich

Verbesserung der Leistungsfähigkeit von Caches: Übersicht

- Beeinflussende Größen (Kapitel 5.2, Hennessy, 3./4. Aufl.):
  - *Hit Time*
  - *Miss Rate*
  - *Miss Penalty*
  - *Cache Bandwidth (Bandbreite)*

Wir betrachten daher im folgenden Verfahren zur:

- Reduktion der Zugriffszeit im Erfolgsfall,
- Reduktion der Rate fehlgeschlagener *Cache*-Zugriffe,
- Reduktion der Wartezyklen, die ein fehlgeschlagener Zugriff erzeugt,
- Erhöhung der Cache-Bandbreite.

Miss Rate ↓:

Klassische Methode zur Cache-Leistungssteigerung.

Typen von *Cache-Misses*:

- *Compulsory*

  Der erste Zugriff auf Speicherbereich kann nie im Cache sein.

- *Conflict*

  Bei *direct mapped* und n-Wege assoziativem $ können Konflikte bzgl. des Ablegens eines Blocks auftreten
  (d.h. zu viele Blöcke werden auf gleiches „Set" abgebildet)

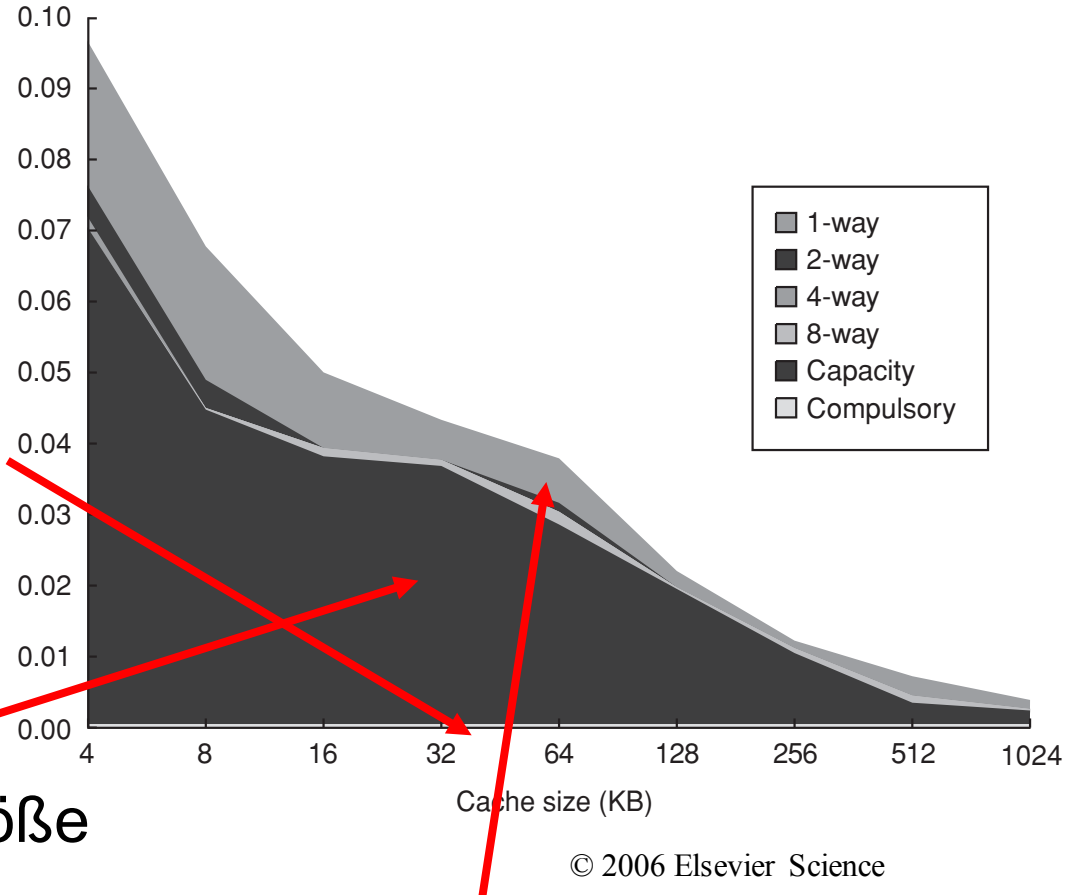
- *Capacity*

  Misses, die bei Assoziativität auftreten würden

  Bei Kapazitätsüberschreitung: Blöcke verwerfen, später wieder holen
  Worst Case: Cache viel zu klein

Miss Rate ↓: (2)

Anteil der Fehlzugriffstypen (SPEC2000)

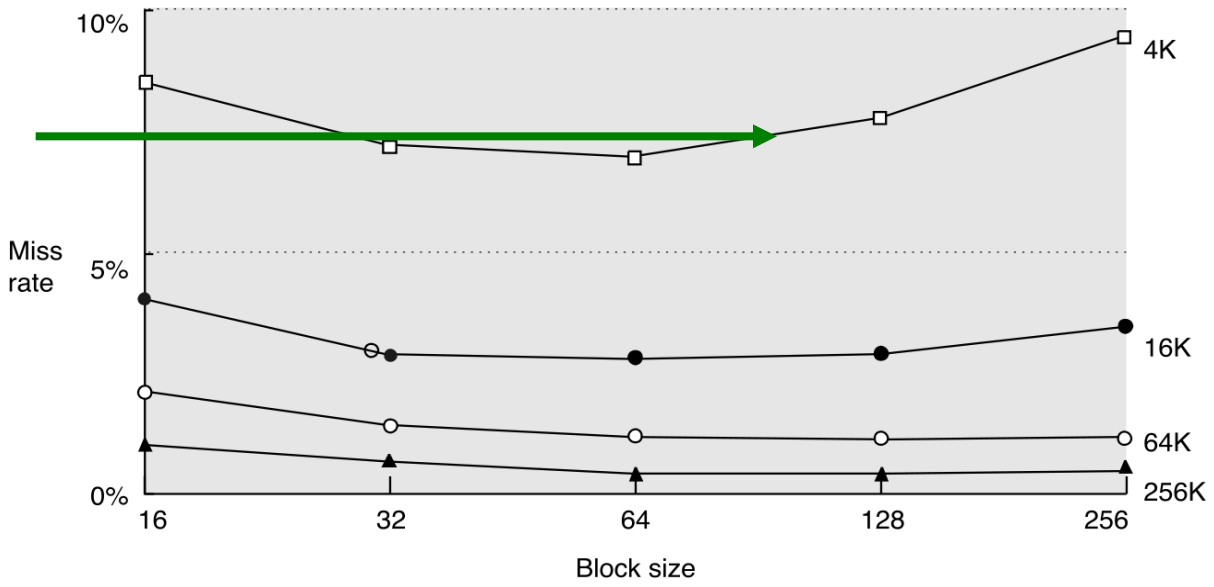


- Extrem wenige zwangsweise „*misses*" (normal bei lang laufenden Programmen)
- *Capacity misses* reduziert mit wachsender Cache-Größe
- Weitere Fehlzugriffe durch fehlende Assoziativität (Konflikte; bzgl. voll assoziativem *Cache*)

Miss Rate ↓: Größere Cache-Blöcke

Reduziert zwangsweise Fehlzugriffe wegen örtlicher Lokalität

- längere Ladezeiten
- # Blöcke reduziert, evtl. mehr Konflikt- und Kapazitätsfehl-zugriffe



- Kosten des Fehlzugriffs dürfen Vorteile nicht überwiegen
- Abhängig von Latenz und Bandbreite des Speichers

- *Miss Rate*
- *Hit Time*
- *Cache Bandwidth*
- *Miss Penalty*

Übung

	Cache Size			
Block Size (bytes)	4K	16K	64K	256K
16	8.57%	3.94%	2.04%	1.09%
32	7.24%	2.87%	1.35%	0.70%
64	7.00%	2.64%	1.06%	0.51%
128	7.78%	2.77%	1.02%	0.49%
256	9.51%	3.29%	1.15%	0.49%

Abb. C.11. aus HP07, © Elsevier

Nehmen wir an, dass das Speichersystem anfänglich 80 Zyklen zwischen Hauptspeicher und Cache für Cache Misses benötigt. Dann liefert es jede 2 Zyklen 16 Bytes. Deshalb liefert es 16 Bytes in 82 Zyklen, 32 Bytes in 84 Zyklen, 48 Bytes in 86 Zyklen, usw. Welche Blockgröße hat die kleinste durchschnittliche Zugriffszeit für jede Cachegröße oben?

Miss Rate ↓: Größere Caches

Nachteile:

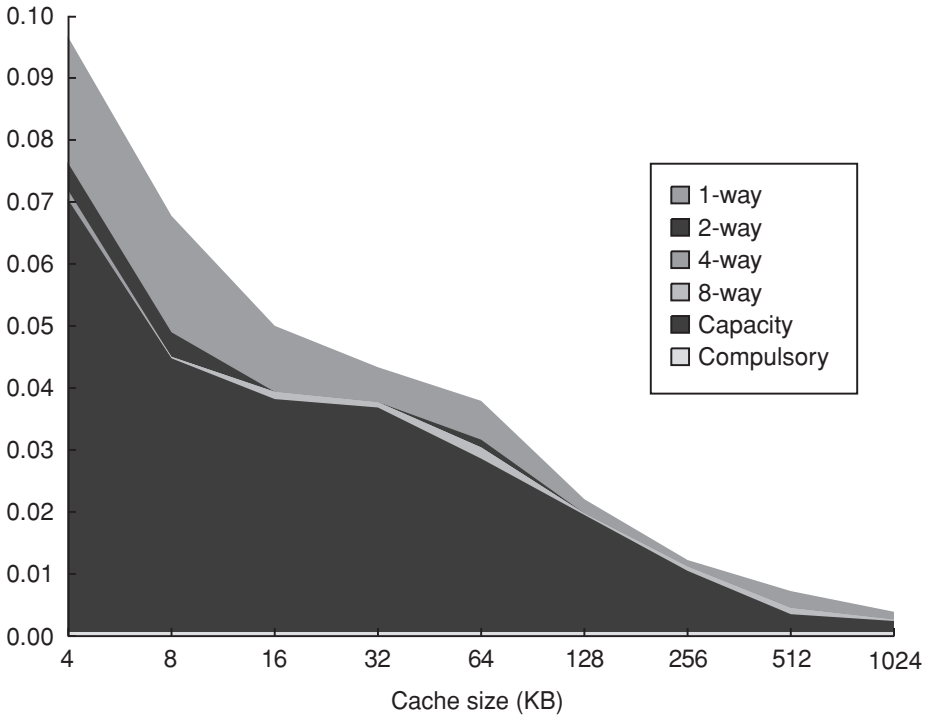
- Höhere Kosten
- Längere Cache-Zugriffszeit

Bei 2-stufigen Caches ist Zugriffszeit der 2. Ebene nicht primär ausschlaggebend
☞ Hier größere Caches verwenden (ggf. langsamer)

Verbreitete Lösung: Große Caches als 2nd- oder 3rd-level Caches haben heute Kapazitäten wie Hauptspeicher früher!

Miss Rate ↓: Höhere Assoziativität

Assoziativität reduziert Fehlzugriffsrate



- In der Praxis 8-Wege assoziativ ≅ voll assoziativem Cache (aber: deutlich geringere Komplexität!)!
- „2:1-*Cache*-Daumenregel" *direct mapped* Cache der Größe N hat ca. Fehlzugriffsrate von 2-Wege assoziativem Cache halber Größe
- Hohe Assoziativität erhöht Zykluszeit, da Cache (1. Ebene) direkt mit CPU gekoppelt!

- *Miss Rate*
- ***Hit Time***
- *Cache Bandwidth*
- *Miss Penalty*

Hit Time ↓: Kleine und einfache Caches

Zugriffszeit im Erfolgsfall (*hit time*) ist kritisch für Leistung von Caches, da sie CPU-Takt begrenzt

- Indizierung des *Tag*-Speichers (via Index) und Vergleich mit angefragter Adresse ☞ schnell, wenn *Cache* klein
- Kleiner Cache kann *on-chip* (der CPU) realisiert werden ☞ kürzere Signallaufzeiten, effizienterer Datentransfer
- Einfacher, d.h. *direct mapped*: ☞ Überlappung von Datentransfer und *Tag*-Prüfung möglich

☞ Größe des L1-*Caches* ggf. nicht erhöht, evtl. sogar reduziert (Pentium III: 16KB, Pentium 4: 8KB)

Schwerpunkt auf höherer Taktrate, Kombination mit L2 *Cache*

- *Miss Rate*
- ***Hit Time***
- *Cache Bandwidth*
- *Miss Penalty*

Hit Time ↓: Kleine und einfache Caches

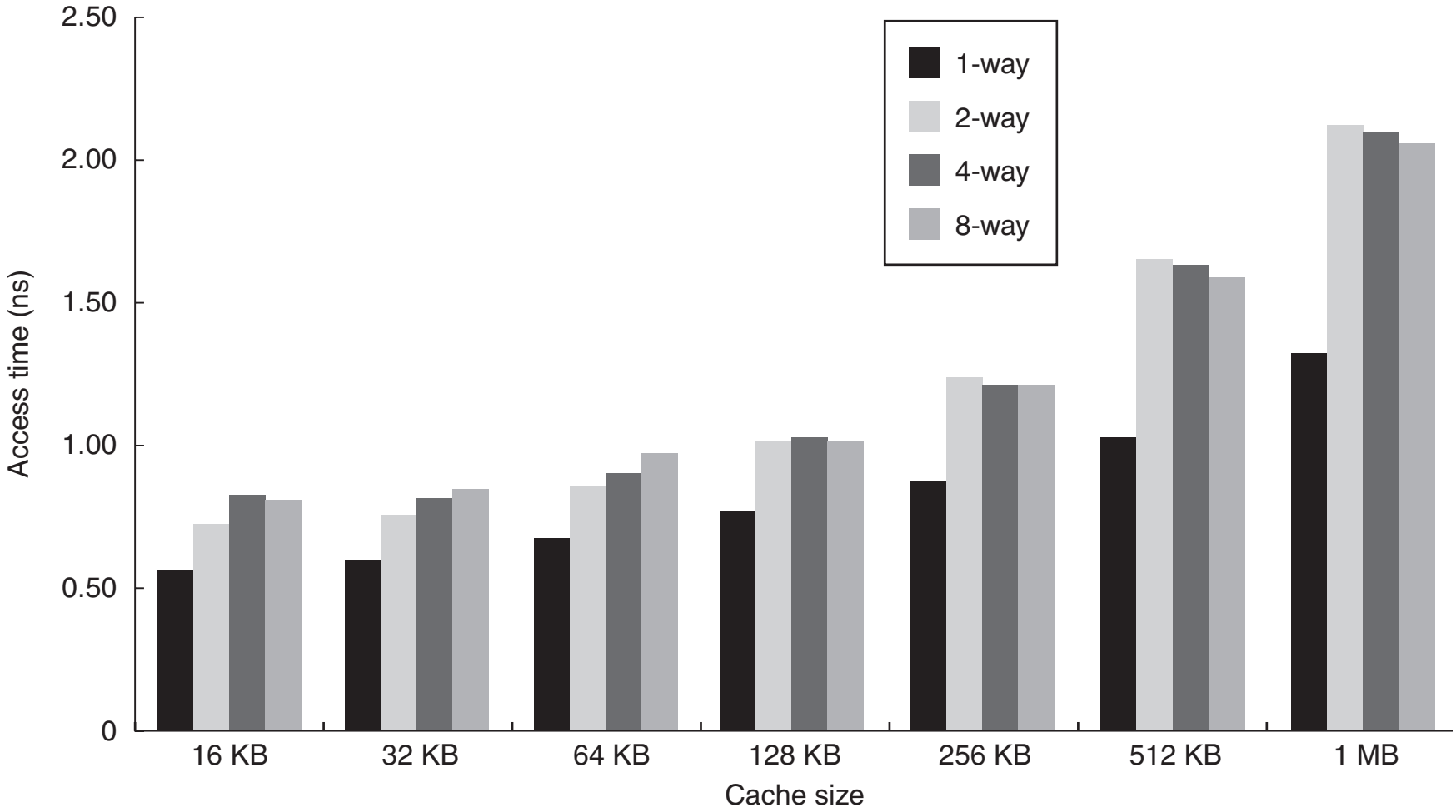


Abb. 5.4. aus HP07, © Elsevier

- *Miss Rate*
- ***Hit Time***
- *Cache Bandwidth*
- *Miss Penalty*

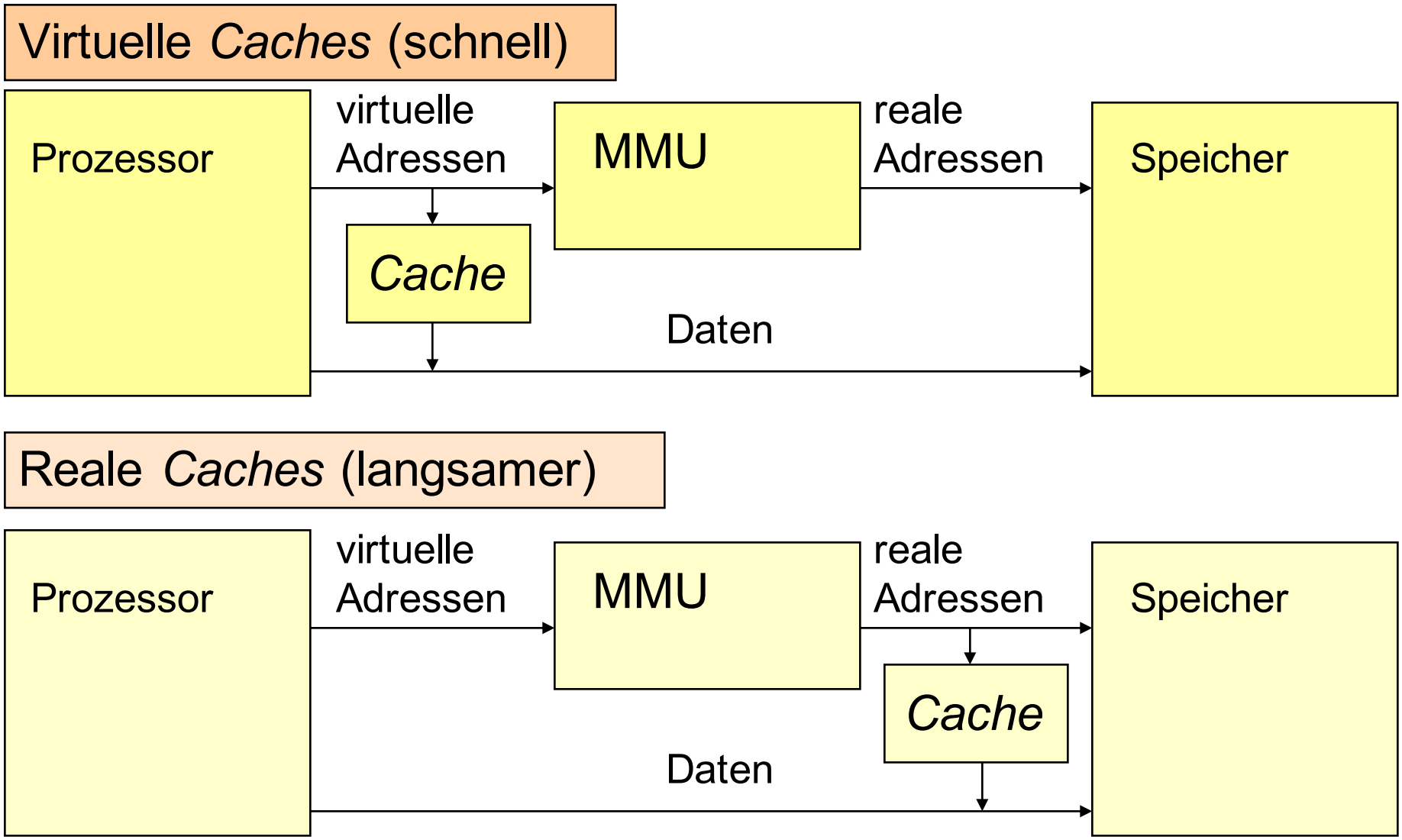
Hit Time ↓: Keine Adress-umrechnung bei Cache-Zugriff

Aktuelle PCs verwenden virtuellen Speicher:

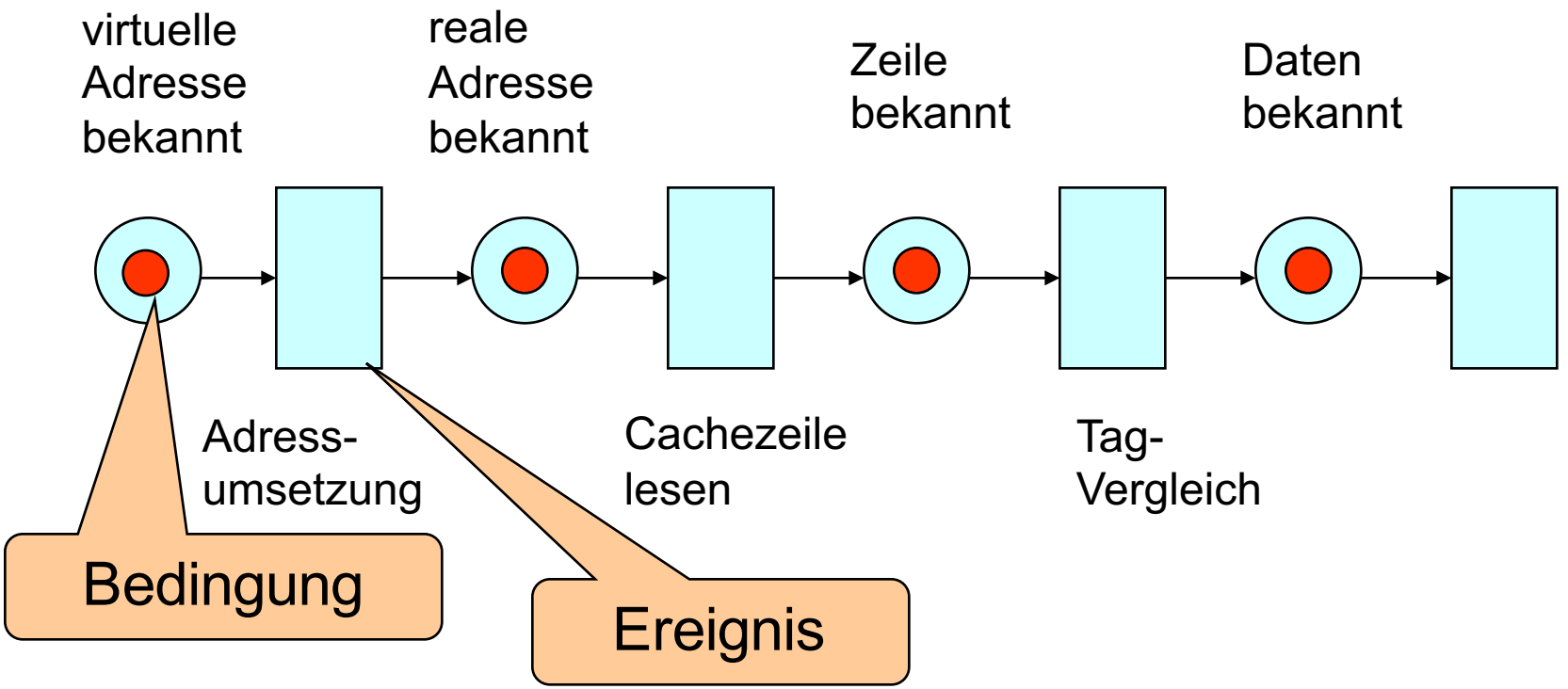
Virtuelle oder reale Caches?

Hier: Zusätzliche Betrachtung von Mischformen

Zur Erinnerung aus RS

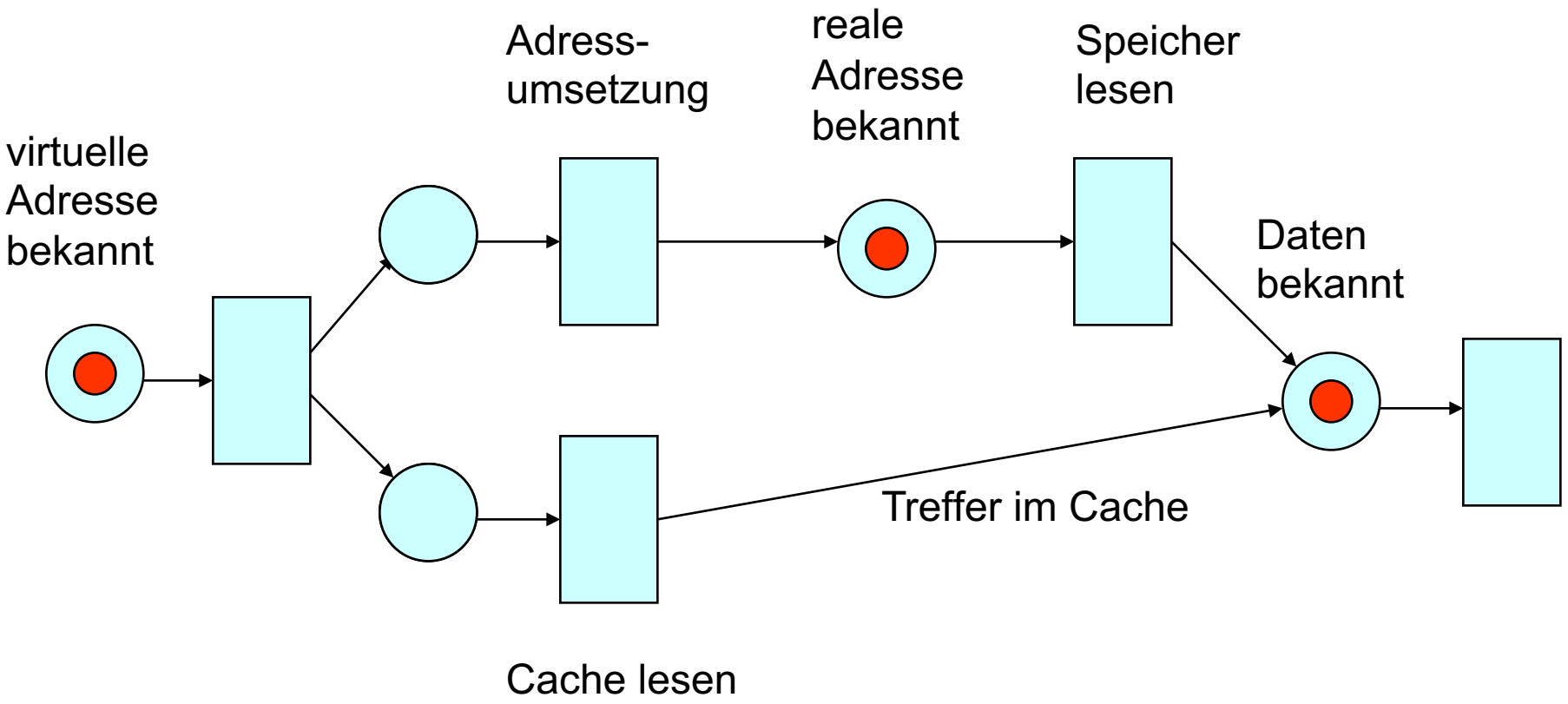


Modell des Ablaufs bei realen Caches



Adressumsetzung und Lesen des Caches nacheinander

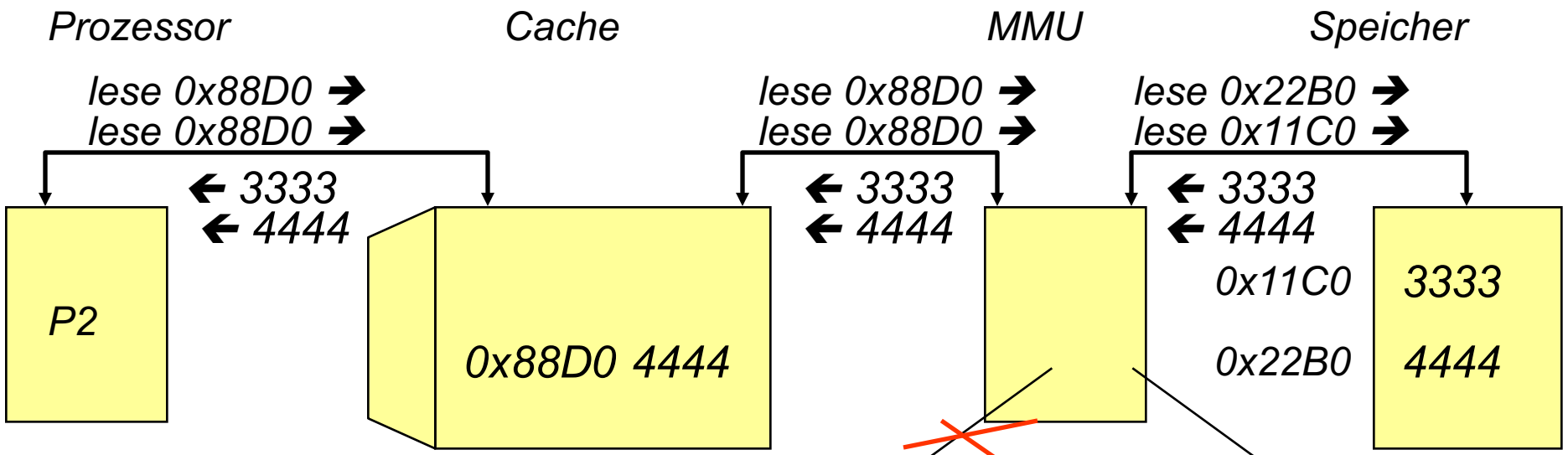
Modell des Ablaufs bei virtuellen Caches



Lesen des Caches und Adressumsetzung „parallel“

Virtueller *Cache* bei Prozesswechsel (*context switch*)

Zu einer bestimmten Adresse im Cache gehört vor und nach dem Prozesswechsel ***eine andere*** *Zelle im Speicher.*



☞ *Bei Prozesswechsel muss der Cache ungültig erklärt werden.*

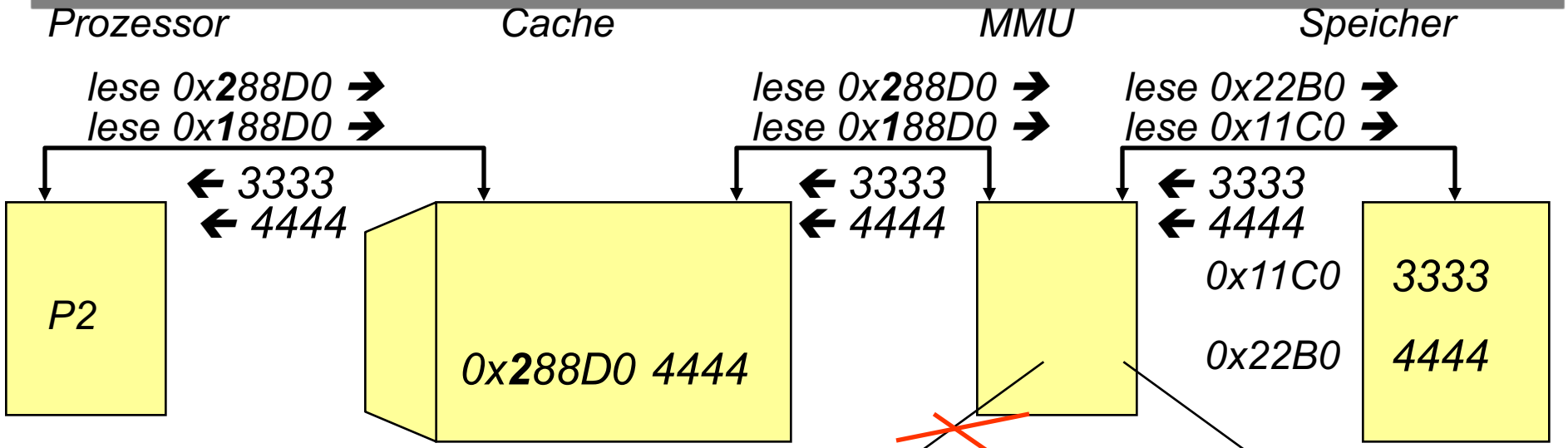
Prozess P1	
Seite	**Kachel**
0x88D.	0x11C.

Prozess P2	
Seite	**Kachel**
0x88D.	0x22B.

Bei jedem Prozesswechsel gehen alle Informationen im Cache verloren.
☞ *Schlecht für große Caches!*

Virtueller *Cache* mit PIDs bei Prozesswechsel (*context switch*)

Virtuelle Adresse wird um Prozessnummer (process identifier, PID) erweitert. Zu einer bestimmten Adresse im Cache gehört vor und nach dem Prozesswechsel wieder dieselbe Zelle.



☞ *Bei Prozesswechsel keine Aktion im Cache erforderlich.*

Set associative mapping ☞ Eintrag für P1 würde erhalten.
☞ *Gut für große Caches!*

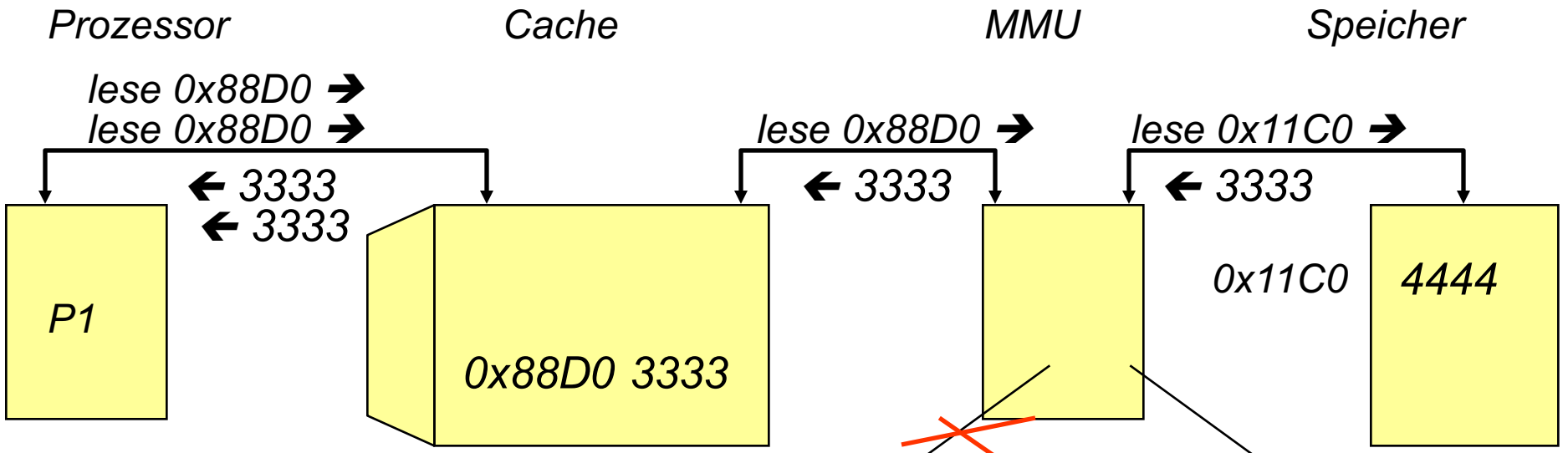
Prozess P1	
Seite	**Kachel**
0x88D.	0x11C.

Prozess P2	
Seite	**Kachel**
0x88D.	0x22B.

TLBs in der MMU könnten PIDs nutzen.

Virtueller *Cache* bei Seitenfehler

Zu einer bestimmten Adresse im Cache gehört vor und nach dem Seitenfehler dieselbe Information.



☞ *Bei Seitenfehlern keine Aktion im Cache.*

Vor Aus-/Einlagern	
Seite	**Kachel**
0x88D.	0x11C.

Nach Aus-/Einlagern	
Seite	**Kachel**
0x77F.	0x11C.

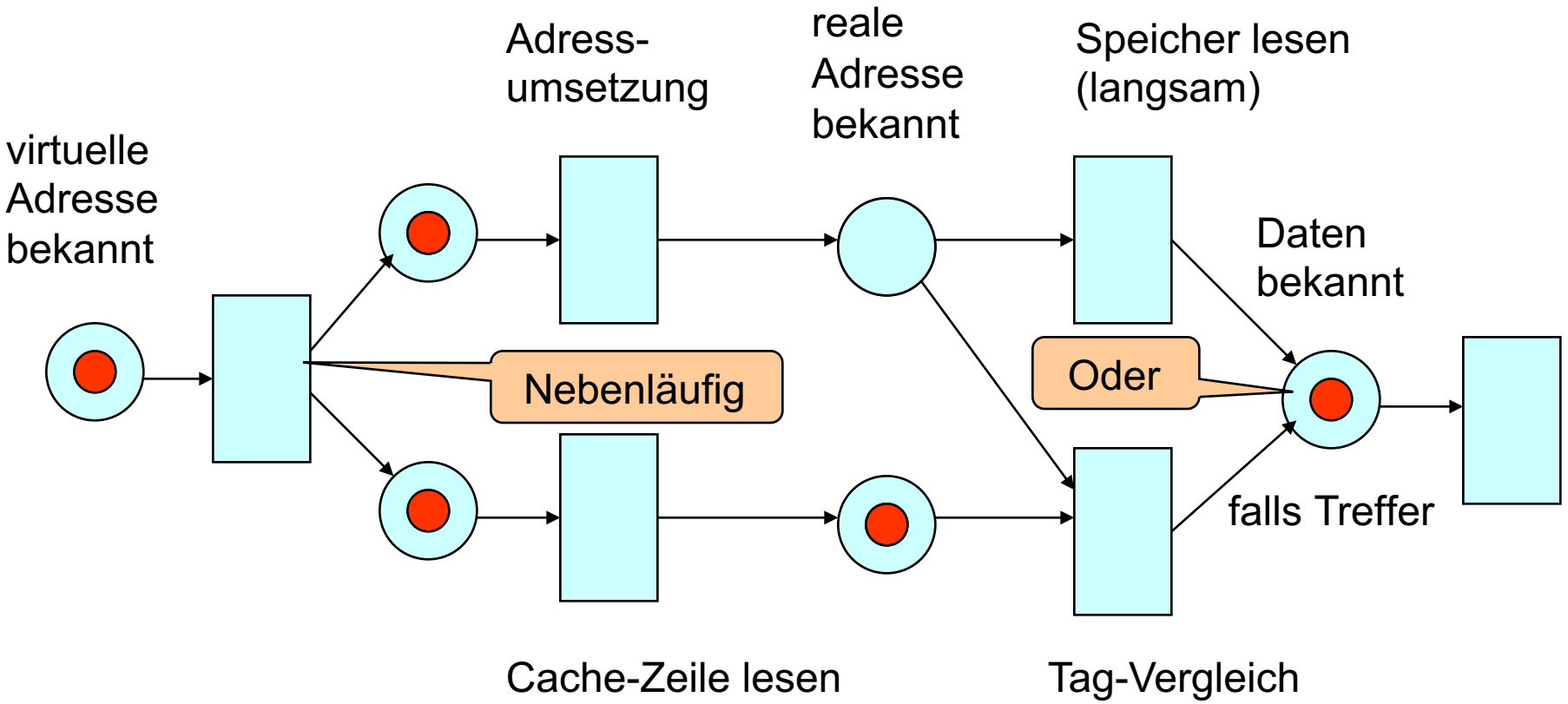
Virtually indexed, virtually tagged

Schnell, aber es gibt Probleme

- Speicherschutz: die MMU muss überprüfen, ob die virtuelle Adresse gültig für den aktuellen Prozess ist. MMU wird immer für jeden Zugriff benutzt
- Prozesswechsel: die Daten im Cache müssen gelöscht werden oder mit zusätzlichen Bits für die Prozessnummer *(engl. process identifier)* getagt (engl. *tagged*) werden
- Eingabe/Ausgabe: Reale Adressen werden typischerweise benutzt um die Eingabegeräte und Ausgabegeräte (I/O) zu lesen bzw. schreiben. Deshalb braucht das System Adressabbildung zwischen der virtuellen Adresse und der realen Adresse.

Mischform „*virtually indexed, real tagged*“

Indexwerte für virtuelle und reale Adressen identisch
☞ Zugriff auf realen Cache kann bereits während der Umrechung auf reale Adressen erfolgen.

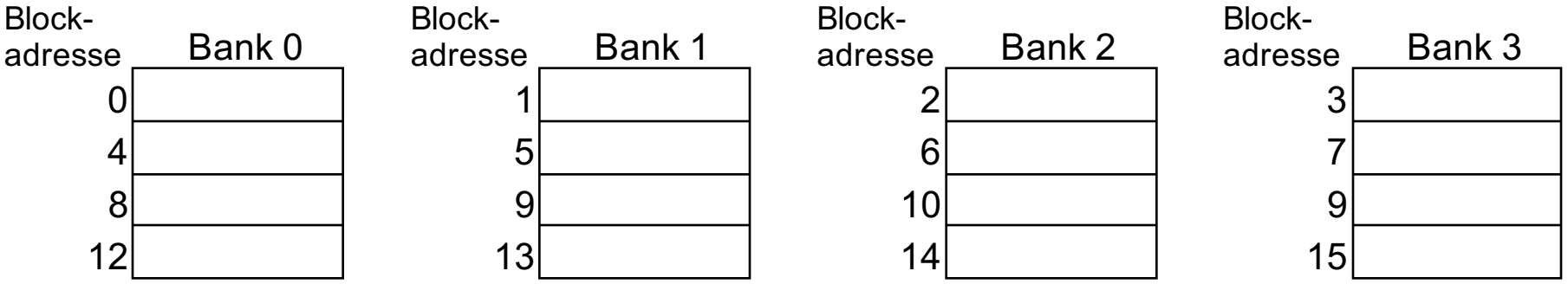


Unterschiedliche Kombinationsmöglichkeiten

Index/ Tag	**Virtuell**	**Real**
Virtuell	*Virtually indexed, virtually tagged* (VIVT) Schnell, Kohärenz-probleme	*Physically (real)-indexed, virtually tagged* Theoretisch möglich, aber sinnlos
Real	*Virtually indexed/ physically (real)- tagged* (VIPT) Mittelschnell	*Physically (real)-indexed*, *physically (real)-tagged* (PIPT) Langsam, keine Kohärenzprobleme

Cache Bandwidth ↑: Multibanked Caches

Blöcke werden über mehrere Speicherbänke verteilt



- *Miss Rate*
- *Hit Time*
- ***Cache Bandwidth***
- *Miss Penalty*

Cache Bandwidth ↑: Pipelined Cache Access

Cache Zugriff wird über mehrere Stufen der Pipeline verteilt

☞ Latenzzeiten des L1-Caches können größer sein als bei Zugriff in einem Takt (Pentium 4: 4 Zyklen)

Konsequenz mehrerer Pipelinestufen

- Größere Verzögerung für Sprungbefehle (*branch penalty*) bei falscher Sprungvorhersage
- Längere Verzögerung zwischen Laden und Verwenden von Operanden (*load delay*), siehe MIPS R4000!
- *Cache*-Latenz ändert sich nicht, aber sie wird versteckt.

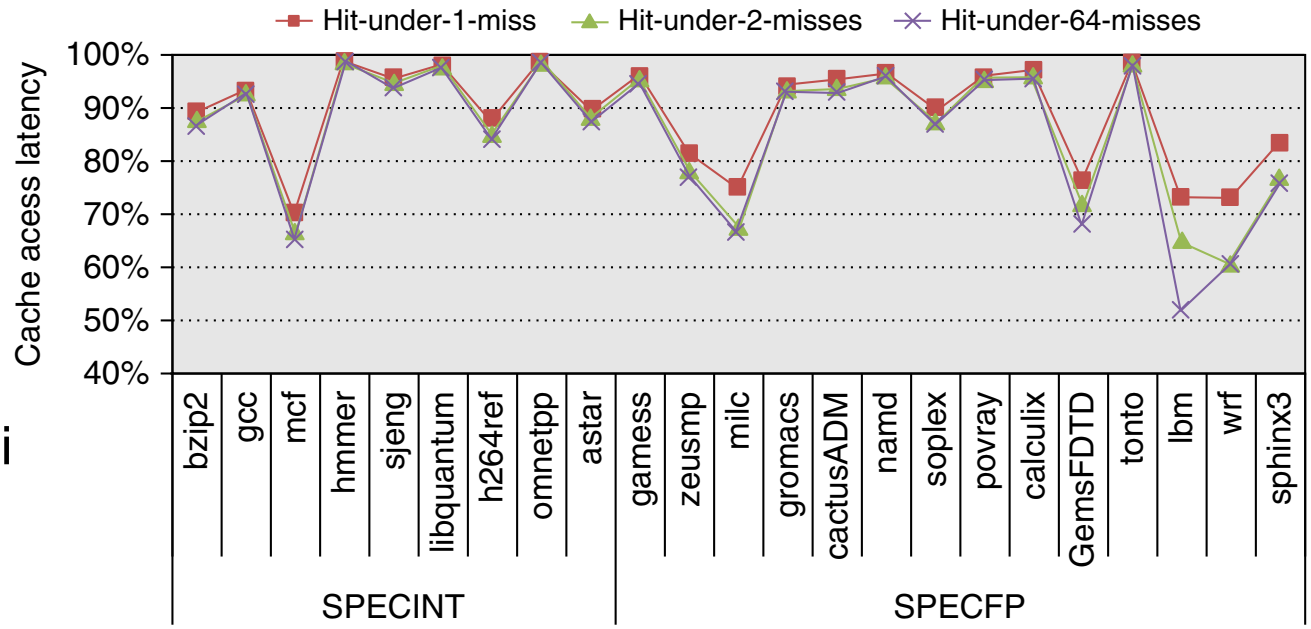
Miss Penalty ↓: *Nonblocking Caches*

- Die Ausführung von Befehlen sollte sich mit Operationen auf der Speicherhierarchie überlappen
- CPUs mit *out-of-order completion/execution*
  ☞ Anhalten der CPU bei Fehlzugriff auf Cache nicht erforderlich (z.B. weiteren Befehl während Wartezeit holen)
- Notwendig: *Cache* kann trotz vorangegangenem (und ausstehendem) Fehlzugriff nachfolgende erfolgreiche Zugriffe (*hits*) befriedigen
  ☞ bezeichnet als "*hit under miss*"
- Auch: "*hit under multiple misses*" bzw. "*miss under miss*"

Miss Penalty ↓: *Nonblocking Caches* (2)

Nicht-blockierende Caches „*hit under (multiple) misses*”:
Vorteil gegen blockierenden Cache (SPECINT/SPECFP)

- *hit under 64 misses* = 1 miss pro 64 Registern
- Verbesserung durch *hit under multiple misses* hauptsächlich bei FP-Benchmarks



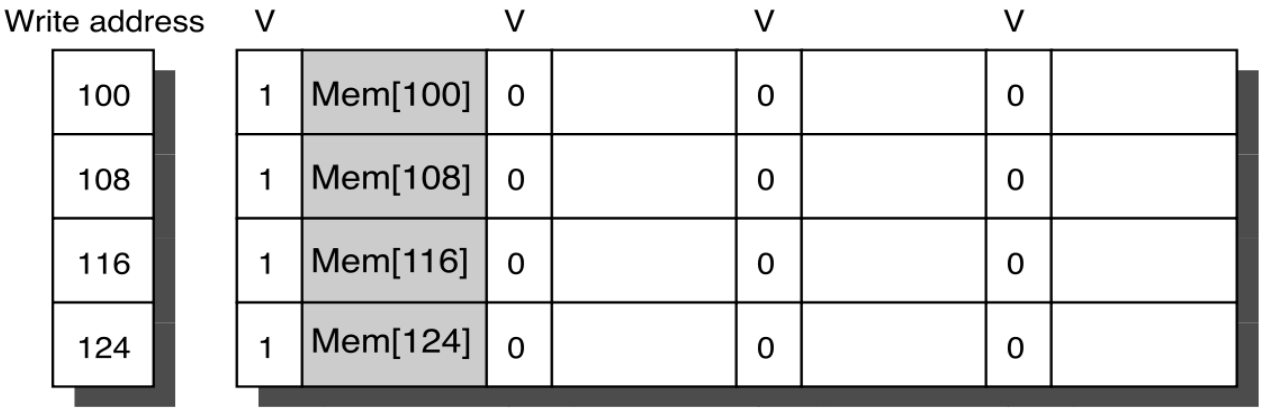
Miss Penalty ↓: Zusammenfassung von Schreibpuffern

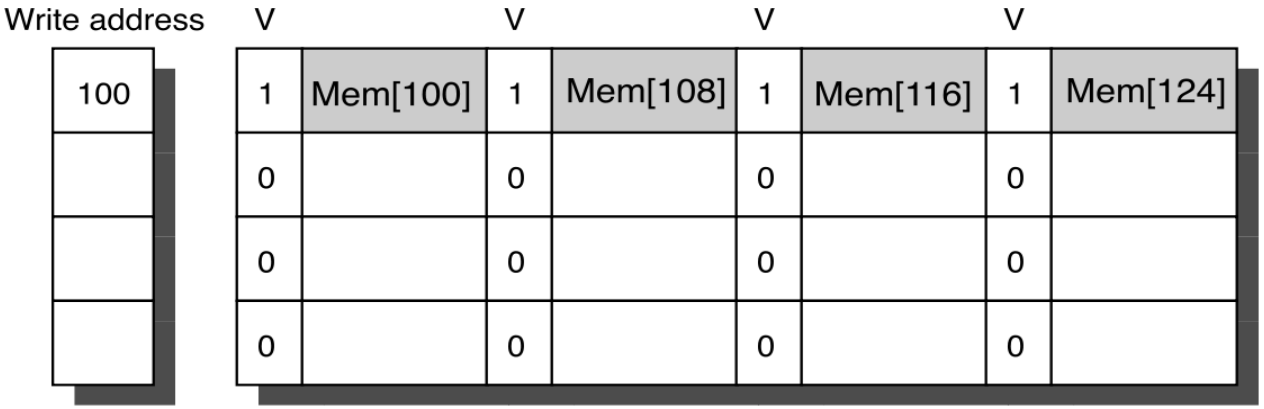
- Betrachte *write through Cache*
- Schreibzugriffe in (kleinem) Puffer gespeichert und bei Verfügbarkeit des Speicherinterfaces ausgeführt
  Hier: Daten wortweise geschrieben!
- Annahme: Schreiben größerer Datenblöcke effizienter möglich
  (trifft auf heutige Speicher in der Regel zu!)
- ☞ Schreibpuffer im *Cache* gruppieren, falls zusammenhängender Transferbereich entsteht

Miss Penalty ↓: Zusammenfassung von Schreibpuffern (2)

Write buffer merging:

- Oben: ohne
- Unten: mit





Beachte: Beispiel extrem optimistisch!

Falls *write buffer merging* nicht vorgesehen, ist Schreibpuffer auch nicht mehrere Worte breit!

Miss Penalty ↓: Einführung mehrerer *Cache*-Ebenen

- Unabhängig von der CPU
- Fokus auf Schnittstelle *Cache*/Hauptspeicher

Dilemma: Leistungsfähigkeit des *Caches* steigern durch

- Beschleunigung des *Caches* (um mit CPU „mitzuhalten") [☞ *Cache* muss kleiner werden] oder
- Vergrößerung des *Caches* (um viele Anfragen an Speicher „befriedigen" zu können) [☞ Cache langsamer]

☞ Eigentlich beide notwendig, aber unvereinbar!

Lösung: 2 *Cache*-Ebenen

- 1st-*level Cache* (L1): schnell, klein, angepasst an CPU-Takt
- 2nd-*level Cache* (L2): groß genug, viele Zugriffe zu puffern, kann langsamer sein

Miss Penalty ↓: Einführung mehrerer *Cache*-Ebenen (2)

Leistungsbewertung jetzt komplexer:

Interaktion von L1 und L2-Cache muss berücksichtigt werden!

(1) *Mittl. Zugriffszeit* = *Hit Time*(L1) + *Miss Rate*(L1) x *Miss Penalty* (L1)
mit Charakteristik des L2-Caches

(2) *Miss Penalty*(L1) = *Hit Time*(L2) + *Miss Rate*(L2) x *Miss Penalty*(L2)

(3) *Mittl. Zugriffszeit* = *Hit Time*(L1) + *Miss Rate*(L1) x
(*Hit Time*(L2) + *Miss Rate*(L2) x *Miss Penalty*(L2))

Beachte:

- Unterscheidung von lokaler und globaler Fehlzugriffsrate
(lokal groß für L2-*Cache*, da nur auf Fehlzugriffen von L1)

Miss Penalty ↓: Einführung mehrerer *Cache*-Ebenen (3)

Parameter von 2-Ebenen-Cache-Systemen:

- Geschwindigkeit von L1-Cache beeinflusst CPU-Takt
- Geschwindigkeit von L2-Cache beeinflusst nur Verzögerung bei Fehlzugriff auf Daten in L1-Cache ☞ (relativ?) selten

☞ Viele Alternativen für L2, die bei L1 nicht anwendbar sind

Ziel: L2-Cache (wesentlich) größer als L1-Cache!

Miss Penalty ↓: Einführung mehrerer *Cache*-Ebenen (4)

Verhältnis von L1- zu L2-Cache-Inhalten:

- *Multilevel* ***inclusion*** (Daten in L1 immer in L2 vorhanden)
  ☞ Konsistenz(prüfung) einfach!
  Beachte: Setzt deutlich größeren L2-Cache voraus als s.u.

- *Multilevel* ***exclusion*** (Daten von L1 nie in L2 vorhanden)
  L1-Fehlzugriff bewirkt Vertauschung von Blöcken zw. L1 u. L2
  ☞ speicherökonomisch, falls L2 nur gering größer als L1
  (z.B. AMD Athlon: L1 = 64kB, L2 = 256kB)

Miss Penalty ↓: Critical Word First & Early Restart

CPU benötigt häufig nur einen Wert aus dem *Cache*-Block

☞ nicht warten, bis kompletter Block nachgeladen ist

2 Realisierungsmöglichkeiten:

- *Critical Word First*
  - Zuerst angefragtes Datenwort aus Speicher holen
  - CPU-Ausführung sofort fortsetzen lassen
  - Weitere Inhalte dann parallel zur CPU holen
- *Early Restart*
  - Daten in normaler Reihenfolge in Cache-Block laden
  - CPU-Ausführung fortsetzen, sobald angefragtes Wort geladen

Es ist vorteilhaft nur bei großen Cache-Blöcken

Es ist vorteilhaft mit örtlicher Lokalität im aktuellen Block!

Miss Penalty ↓: Priorisierung von Lese- über Schreibzugriffen

Problem bei *write through*:

Wichtigste Optimierung: Schreibpuffer
☞ kann Wert enthalten, der von Lesezugriff angefragt ist

Idee: Wir wollen Lesezugriffe behandeln, bevor Schreiboperation abgeschlossen ist

Lösungsmöglichkeiten:

- Warten bis Schreibpuffer leer (kontraproduktiv)
- Inhalte des Schreibpuffers prüfen und mit Lesezugriff fortfahren, falls keine Konflikte

Vorgehen bei *write back*:

- „*dirty*" Block in (kleinen) Puffer übernehmen
- Zuerst neue Daten lesen, dann „*dirty*" Block schreiben

Miss Penalty ↓: *Prefetching* (durch HW oder SW)

(deutsch etwa „Vorabruf", „vorher abholen")

Elemente in *Cache* laden, bevor sie gebraucht werden

- *Prefetching* versucht Speicherbandbreite zu verwenden, die anderweitig ungenutzt bliebe
- Kann mit „regulären" Anfragen/Zugriffen interferieren!

Miss Penalty ↓: *Prefetching* (durch HW oder SW)

a) Hardware-*Prefetching* für Instruktionen/Daten

- Bei *cache miss*: angefragter Block → *Cache*, sequentieller Nachfolger → "*stream buffer*"
- Falls bei späterer Anfrage Block im *stream buffer*: ursprüngliche Cache-Anfrage verwerfen, aus *stream buffer* nachladen und neuen *prefetch* starten

Miss Penalty ↓: *Prefetching* (durch HW oder SW) (2)

b) *Prefetching* kontrolliert durch Compiler

- Verwendet spezielle Befehle moderner Prozessoren
- Möglich für Register und *Cache*
- *Prefetch*-Befehle dürfen keine Ausnahmen (insbes. Seitenfehler) erzeugen ☞ *non-faulting instructions*
- Cache darf nicht blockierend sein (damit Prefetch-Lade-vorgänge und weitere Arbeit der CPU parallel möglich)
- *Prefetching* erzeugt *Overhead* ☞ Kompromiss erforderlich (nur für die Daten *prefetch* Instruktionen erzeugen, die mit großer Wahrscheinlichkeit Fehlzugriffe erzeugen)

- *Miss Rate*
- *Hit Time*
- *Cache Bandwidth*
- *Miss Penalty*

Zusammenfassung

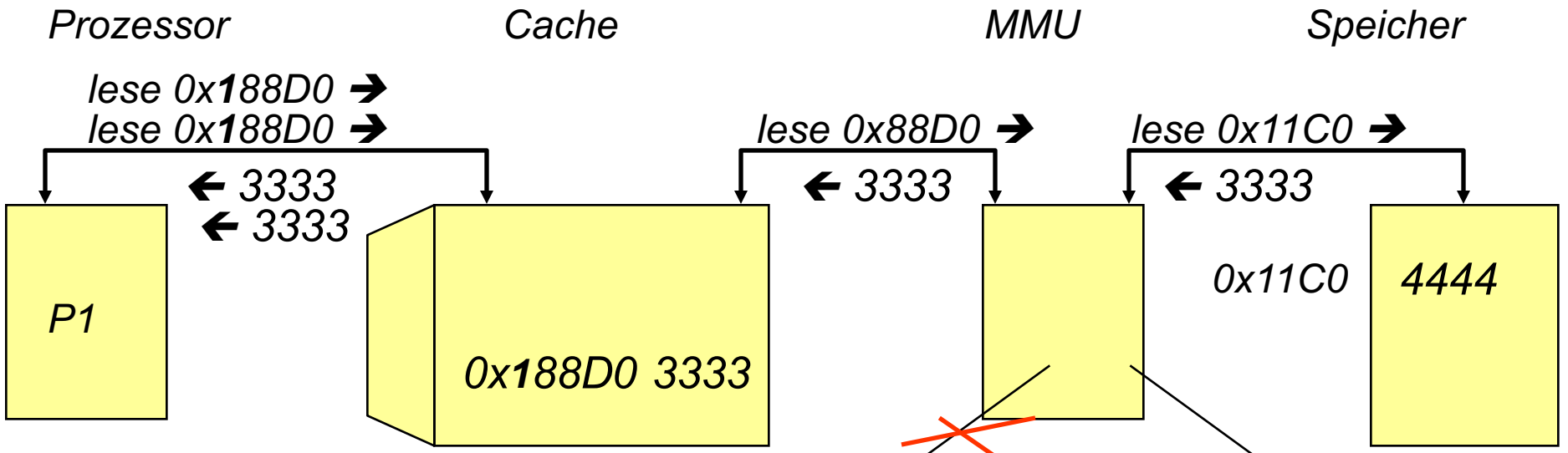
Betrachten von Verfahren zur

- Reduktion der Rate fehlgeschlagener *Cache*-Zugriffe
- Reduktion der Zugriffszeit im Erfolgsfall
- Erhöhung der Speicherbandbreite
- Reduktion der Wartezyklen, die ein fehlgeschlagener Zugriff erzeugt

The rest is omitted in the lecture

Virtueller *Cache* mit PIDs bei Seitenfehler

Zu einer bestimmten Adresse im Cache gehört vor und nach dem Seitenfehler dieselbe Information.



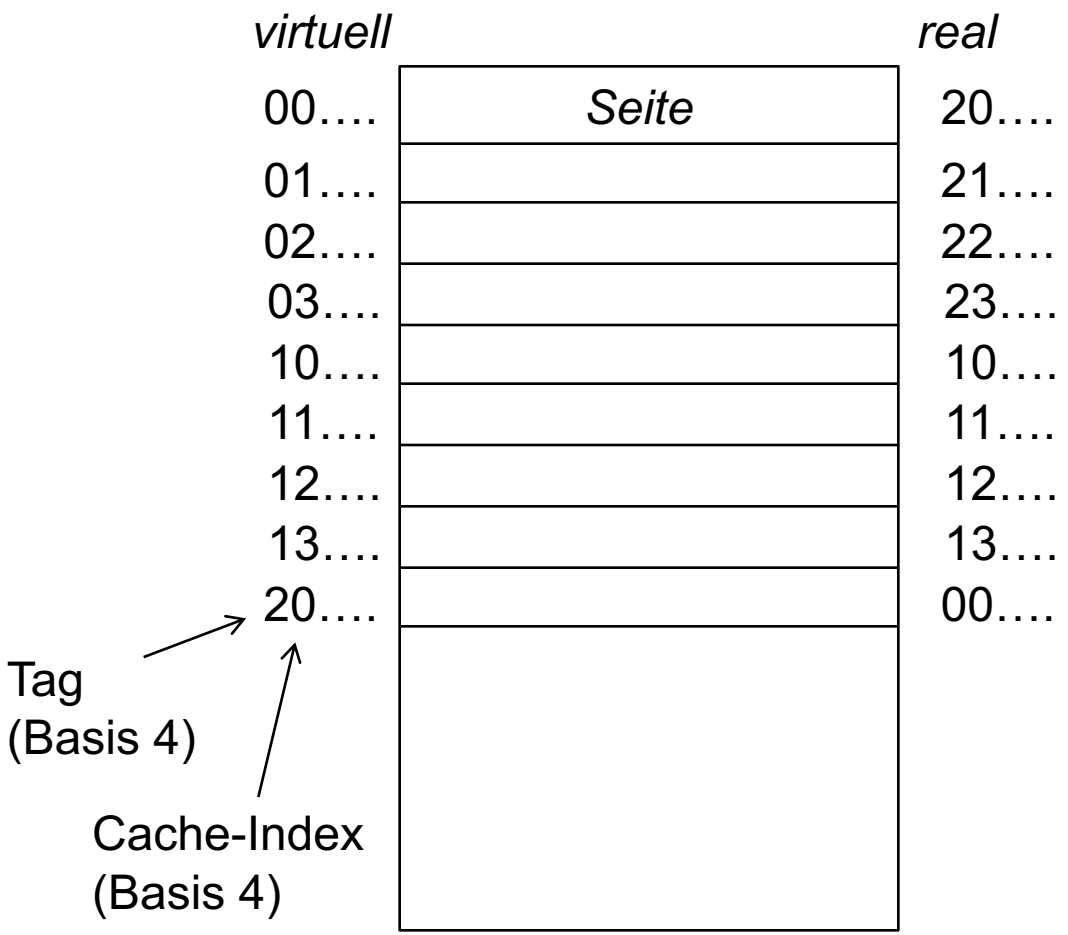
☞ *Bei Seitenfehlern keine Aktion im Cache.*

Vor Aus-/Einlagern	
Seite	**Kachel**
0x88D.	0x11C.

Nach Aus-/Einlagern	
Seite	**Kachel**
0x77F.	0x11C.

Mischform „*virtually indexed, real tagged*“ (2)

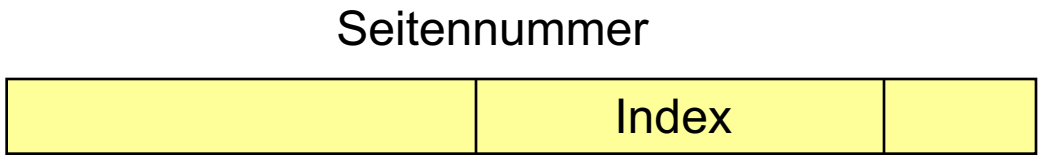
- Sei eine „*Meta-Page*“ der Speicherbereich, der zu einem bestimmten Tag-Feld gehört (unsere Bezeichnung)
- Innerhalb jeder „*Meta-Page*“ müssen virtueller und realer Index übereinstimmen.
- Meta-Page-Zuordnung beliebig



Mischform „*virtually indexed, real tagged*“ (3)

Nachteil:

- De facto: statt Seiten „*Meta-Pages*“ als Einheiten der Speicherzuteilung.
  Sehr grob-granulare Speicherzuteilung.
- Beispiel: *direct mapped* $, 256 kB=$2^{18}$ B, 64 Bytes /Eintrag
  ☞ 12 Bits für den Index, Meta-Pages von 256 kB.



Miss Rate ↓
Compiler-Optimierungen

Fortschritt der Compiler-Technik ermöglicht Optimierungen, die Wissen über Speicherhierarchie einbeziehen

Verschiedene Bereiche unterscheidbar:

- Örtliche und zeitliche Lokalität der Zugriffe verbessern!
- Verbesserung des Zugriffs auf Code
  - Umordnung (z.B. Von Prozeduren) zur Reduktion von Fehlzugriffen durch Konflikte
  - Ausrichtung von Basisblöcken an Cache-Block-Grenzen
    ☞ kaum Fehlzugriffe mehr auf weiteren seq. Code
- Zahlreiche Verfahren Bestandteil der ES- und SES-Kurse

Miss Rate ↓
Compiler Optimierungen (2)

Umordnung von Schleifen

Feld (x[.][.]) zeilenweise gespeichert (*row major order*)

Problem: Zugriffe erfolgen nicht sequentiell
(Hier: verschachtelte Schleife ☞ Zugriff mit großer Schrittweite [100])

```
/* Vorher */
for (j = 0; j < 100; j++)
    for (i = 0; i < 5000; i++)
        x[i][j] = 2 * x[i][j]

/* Nachher */
for (i = 0; i < 5000; i++)
    for (j = 0; j < 100; j++)
        x[i][j] = 2 * x[i][j]
```

Falls Array nicht komplett in Cache ☞ viele Fehlzugriffe
(extrem, falls Cache-Größe gleich Zeilengröße des Arrays!)

Modifizierter Code greift sequentiell zu (sofern *row major*!)
☞ Effizienzverbesserung ohne Änderung der Anzahl ausgeführter Befehle;

Miss Rate ↓
Compiler Optimierungen (3)

Blockweise Bearbeitung

(*tiling*)

Idee: Fehlzugriffe reduzieren durch verbesserte temporale Lokalität

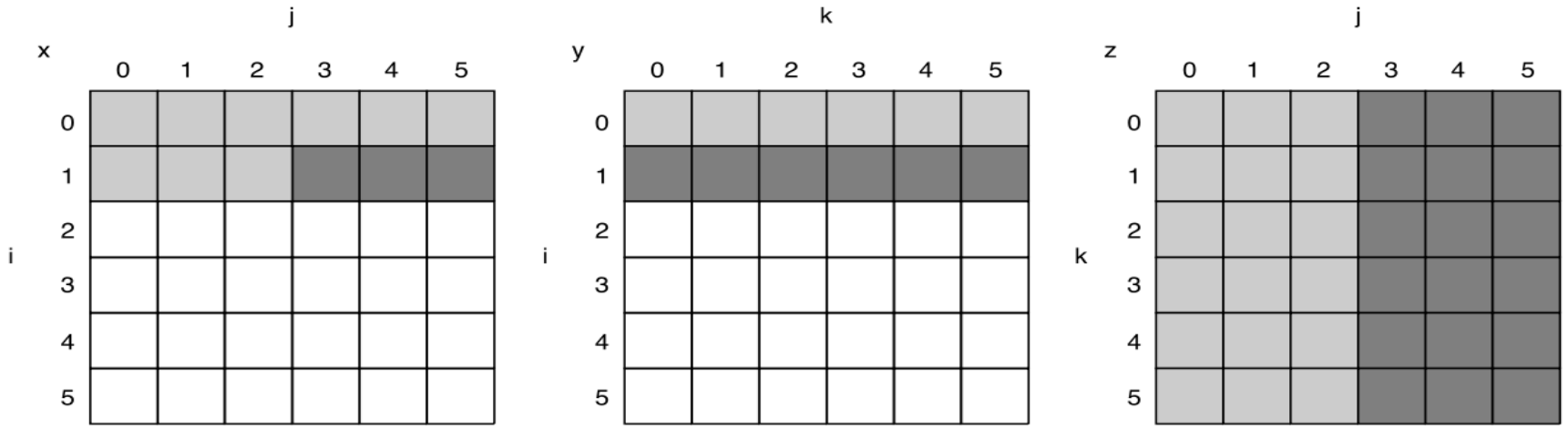
```
/* Vorher */
for (i = 0; i < N; i++)
for (j = 0; j < N; j++) {
      r = 0;
      for (k = 0; k < N; k++)
         r = r + y[i][k]*z[k][j];
      x[i][j] = r;
      }
```

Betrachte Matrixmultiplikation

Problem: Zugriff sowohl zeilen- als auch spaltenweise
☞ Umordnung von Schleifen nicht hilfreich

Miss Rate ↓: Compiler Optimierungen (4)

Zugriffschema bei „naiver” Matrixmultiplikation
(„Alter” des Zugriffs über Einfärbung: dunkel = aktuell,
mittel = länger zurückliegend, weiß = nicht angefragt)



☞ erfordert $2N^3 + N^2$ Datenzugriffe

Lösungsprinzip: Bearbeitung kleiner Teilblöcke der Daten (hier: Teilmatrizen)

Miss Rate ↓: Compiler Optimierungen (5)

Matrixmultiplikation mit „*blocking factor*" B:

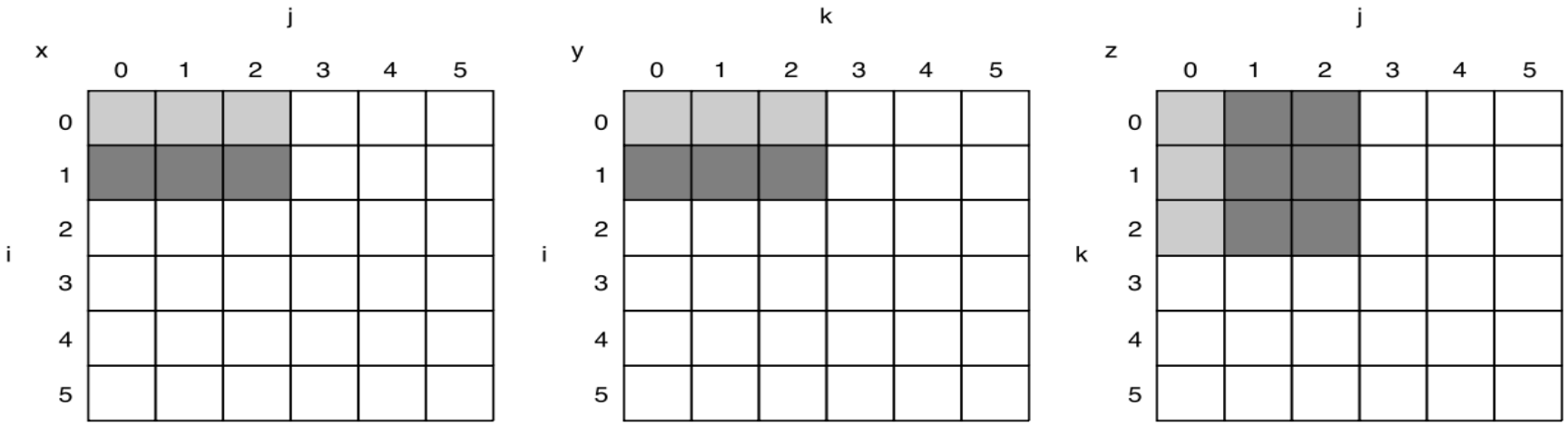
```
/* Nachher */
for (jj = 0; jj < N; jj += B)
for (kk = 0; kk < N; kk += B)
for (i = 0; i < N; i++)
for (j = jj; j < min(jj+B,N); j++) {
      r = 0;
      for (k = kk; k < min(jj+B,N); k++)
         r = r + y[i][k]*z[k][j];
      x[i][j] = x[i][j] + r;
      }
```

Annahme: x[.][.] initial == 0, Block $B \times B$ passt in Cache

☞ Erfordert $2N^3/B + N^2$ Zugriffe ☞ Verbesserung ca. Faktor B

Miss Rate ↓: Compiler Optimierungen (6)

Zugriffschema bei blockweiser Matrixmultiplikation:



☞ Nützt Kombination von temporaler Lokalität (z[.][.]) und örtlicher Lokalität (y[.][.]) aus

Extremfall: Kleine Blockgröße ☞ Daten können in Registern gehalten werden

Miss Penalty ↓: *Victim Caches*

Problem: Verwerfen von *Cache*-Inhalten ungeschickt, falls bald wieder entsprechender Zugriff erfolgt

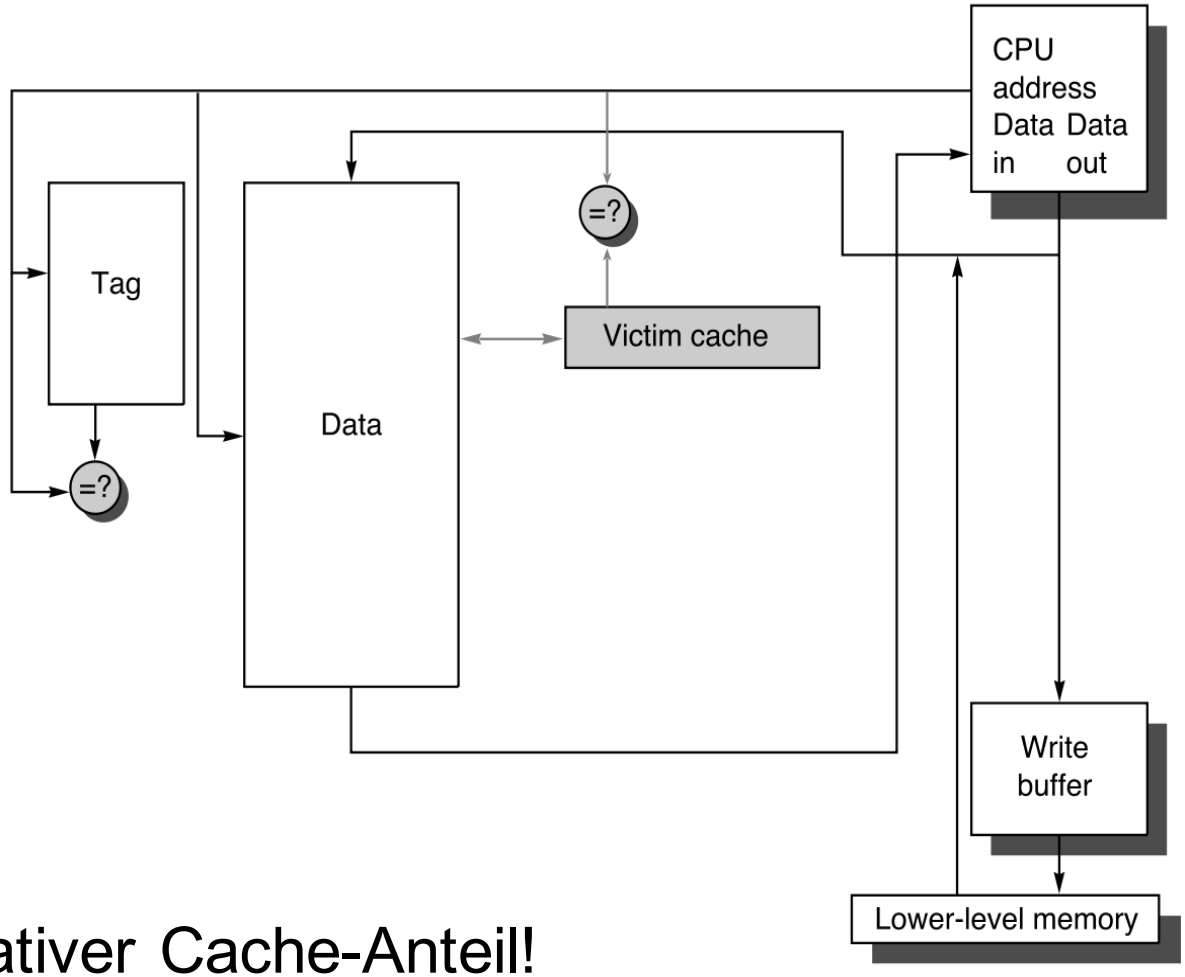
Idee: Verworfene Daten (kurz) puffern (= "Unter"-*Cache*)

- Erfordert kleinen, voll-assoziativen *Cache* zwischen eigentlichem Cache und der nächsten Hierarchieebene
- Bei Fehlzugriffen werden zuerst *victims* (= gerade verworfene Inhalte) geprüft, bevor Zugriff auf die nächste Hierarchieebene erfolgt

Verbesserung der Cache-Leistungsfähigkeit insbesondere bei kleinen, *direct mapped Caches* (bis zu 25% Zugriffe bei 4K-Cache)
AMD Athlon: *victim cache* mit 8 Einträgen: Ist das plausibel?
Was realisiert man mit dem "*Cache* im *Cache*"?

Miss Penalty ↓: *Victim Caches* (2)

Struktur eines *Caches* mit *Victim Cache*



☞ kleiner voll-assoziativer Cache-Anteil!

Hit Time ↓: Way Prediction

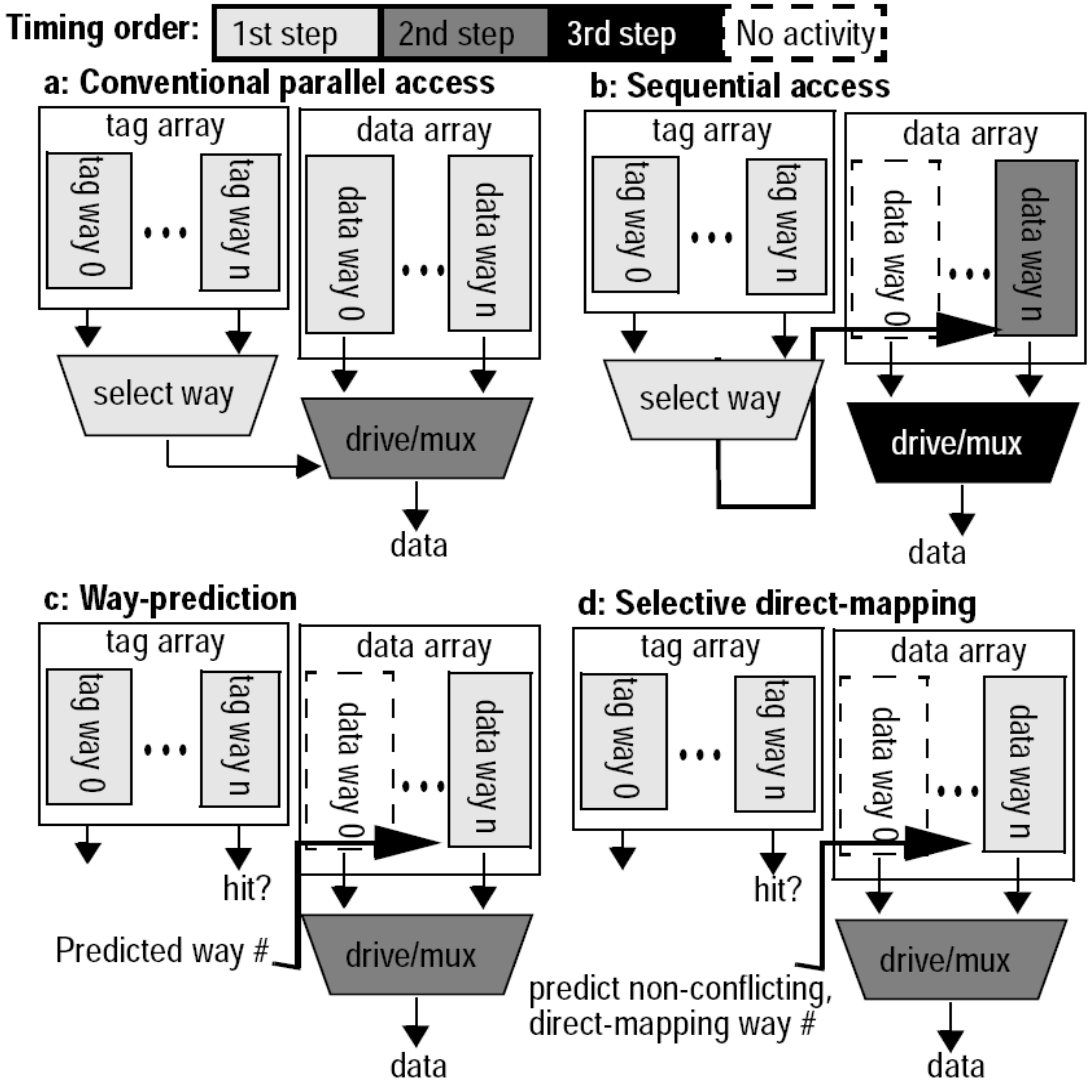
Idee: Zugriffsgeschwindigkeit eines *direct mapped* Caches erhalten und trotzdem Konflikte reduzieren

Methode: Zusätzliche Bits im *Cache*-Set zur Vorhersage des nächsten Zugriffs (in diesem Set)

- Multiplexer zum Auslesen der Daten kann früh gesetzt werden
- Nur ein *Tag*-Vergleich erforderlich (im Erfolgsfall)
- Weitere Tags nur geprüft bei (potentiellem!) *cache miss* (erfordert dann weitere Taktzyklen)

Way-prediction macht Caches energieeffizienter!

Hit Time ↓: Way prediction (2)



[M. D. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, K. Roy: Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct-Mapping, *MICRO-34*, 2001]

Beispiel: Alpha 21264:
1 Bit für *way prediction* in 2-Wege assoziativem (Befehls-) Cache
☞ Latenz 1 Takt bei korrekter (85% für SPEC95) und
3 Takte bei falscher Vorhersage

Hit Time ↓: Trace Caches

Prinzip:

- Befehle werden nur entlang möglicher Ausführungsfolgen gespeichert; *Cache* bestimmt dynamische Befehlssequenz, die in *Cache* geladen wird
- Einträge beginnen an beliebiger Adresse
- Blöcke gehen über Sprungbefehle hinaus

☞ Sprungvorhersage wird Teil des *Cache*-Systems!

- *Traces* werden ggf. mehrfach in *Cache* geladen als Folge von bedingten Sprungbefehlen mit variierendem Verhalten

Beispiel: Intel NetBurst Mikroarchitektur (Pentium 4)