# 4.2 Verbesserung der Leistungsfähigkeit von Caches

Peter Marwedel
Informatik 12
TU Dortmund

2014/05/02

# *Cache*-Performanz

Bewertungsmaß für die Leistungsfähigkeit einer Speicherhierarchie:

Mittlere Zugriffszeit = *Hit Time* + *Miss Rate* x *Miss penalty*

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

Maßeinheiten: Zeit oder Taktzyklen möglich

# *Cache*-Performanz (2)

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

# *Cache*-Performanz (3)

Anzahl der Speicher-Wartezyklen abhängig von:

- Anzahl der Fehlzugriffe (*cache misses*) und
- „Kosten" pro *cache miss* (= *miss penalty*)

Speicher-Wartezyklen =

= # *Cache misses* x *miss penalty*

= IC x (misses / Befehl) x *miss penalty*

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

Beachte:

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

# Verbesserung der Leistungsfähigkeit von Caches ($, €, ¥): Übersicht

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

Wir betrachten daher im folgenden Verfahren zur:

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

# *Miss Rate* ↓:

Klassische Methode zur Cache-Leistungssteigerung.

Typen von Cache-Misses:

- *Compulsory*

  Erster Zugriff auf Speicherbereich kann nie im Cache sein.

- *Capacity*

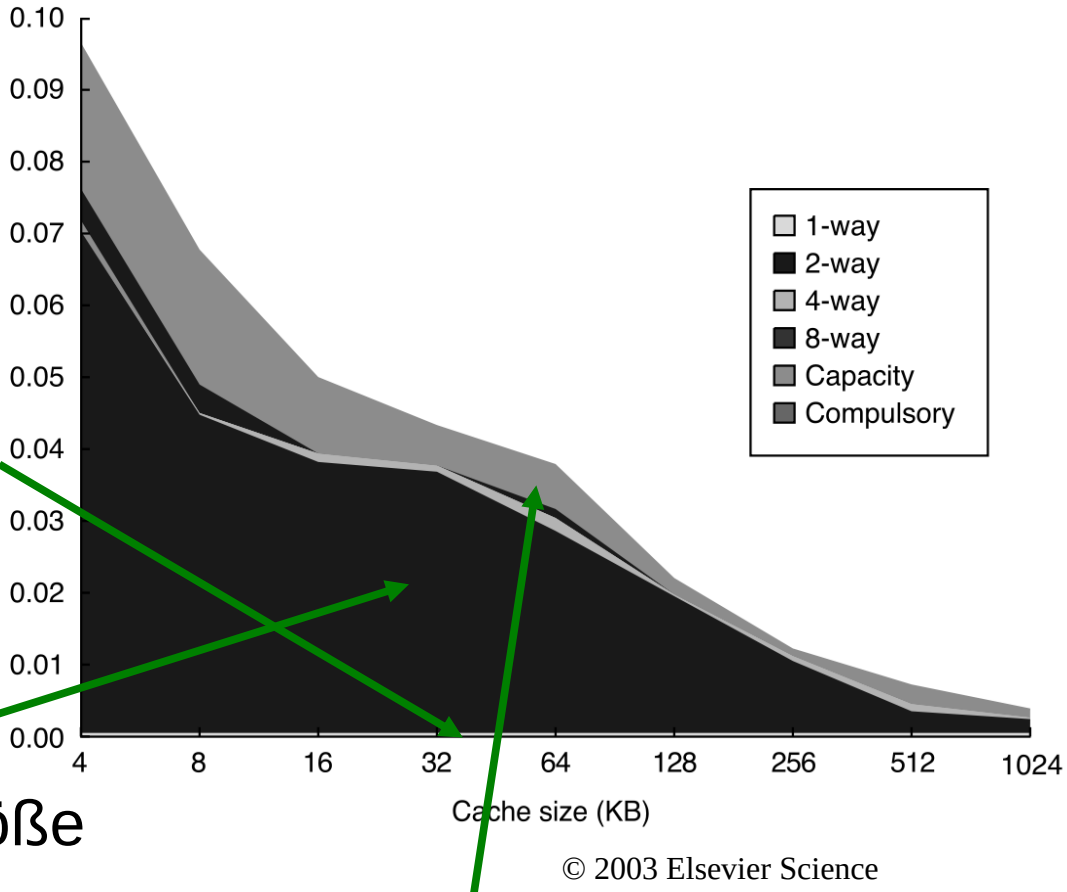
  Falls Cache nicht alle Blöcke enthalten kann, die für Programm-ausführung erforderlich ☞ Kapazitätsüberschreitung: Blöcke müssen verworfen und später wieder geholt werden!
  *Worst Case*: Cache viel zu klein ☞ Blöcke werden ständig ein-/ ausgelagert ☞ "*thrashing*"

- *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)

# *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*)

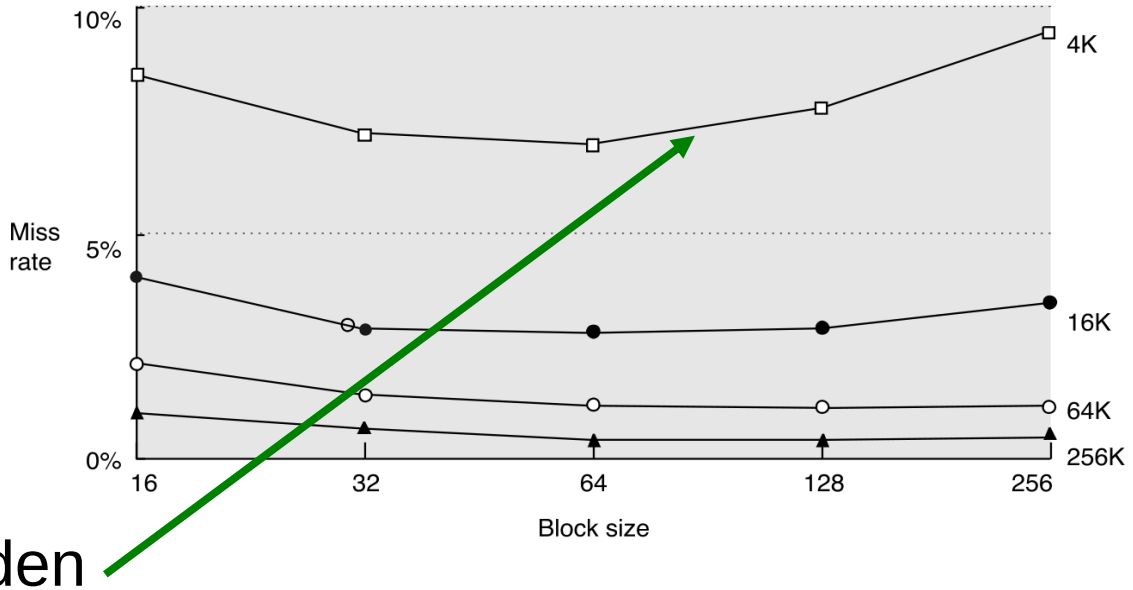
# *Miss Rate* ↓: Größere Cache-Blöcke

Reduziert zwangsweise Fehlzugriffe wegen örtlicher Lokalität

Aber: *Miss penalty* wird erhöht

Auch: Da # Blöcke reduziert evtl. mehr Konflikt- und Kapazitätsfehlzugriffe

☞ Fehlzugriffsrate darf nicht erhöht werden






☞ Kosten des Fehlzugriffs dürfen Vorteile nicht überwiegen

- Abhängig von Latenz und Bandbreite des Speichers (hoch/hoch ☞ große Blöcke, niedrig/niedrig ☞ kleine)

# *Miss Rate* ↓: Größere Caches

Nachteile:

- Längere Cache-Zugriffszeit
- Höhere Kosten

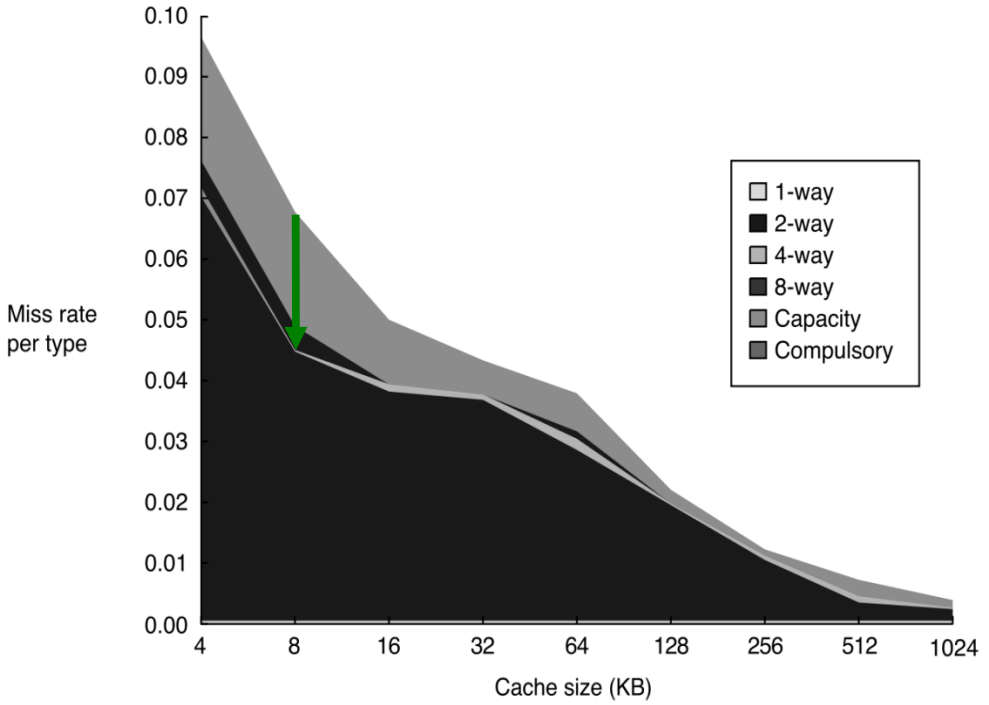
Bei 2-stufigen Caches ist Zugriffszeit der 2. Ebene nicht primär ausschlaggebend
☞ Hier größere Caches verwenden (ggf. langsamer)

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!

# *Hit Time* ↓: Kleine und einfache Caches

Zugriffszeit im Erfolgsfall (*hit time*) 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*

# *Hit Time* ↓: Kleine und einfache Caches

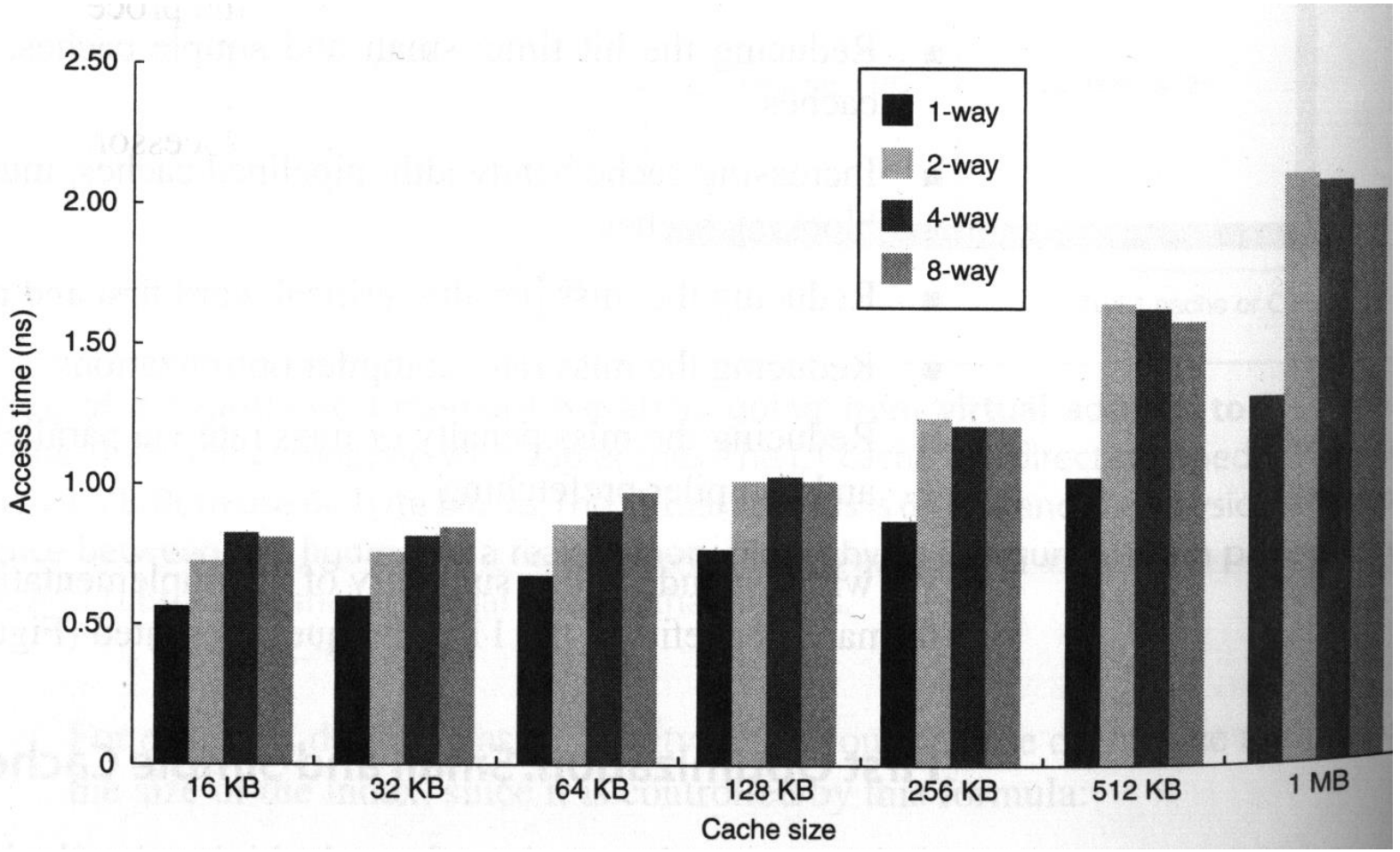



Abb. 5.4. aus HP07, © Elsevier

# *Hit Time* ↓: Keine Adressumsetzung bei Cache-Zugriff

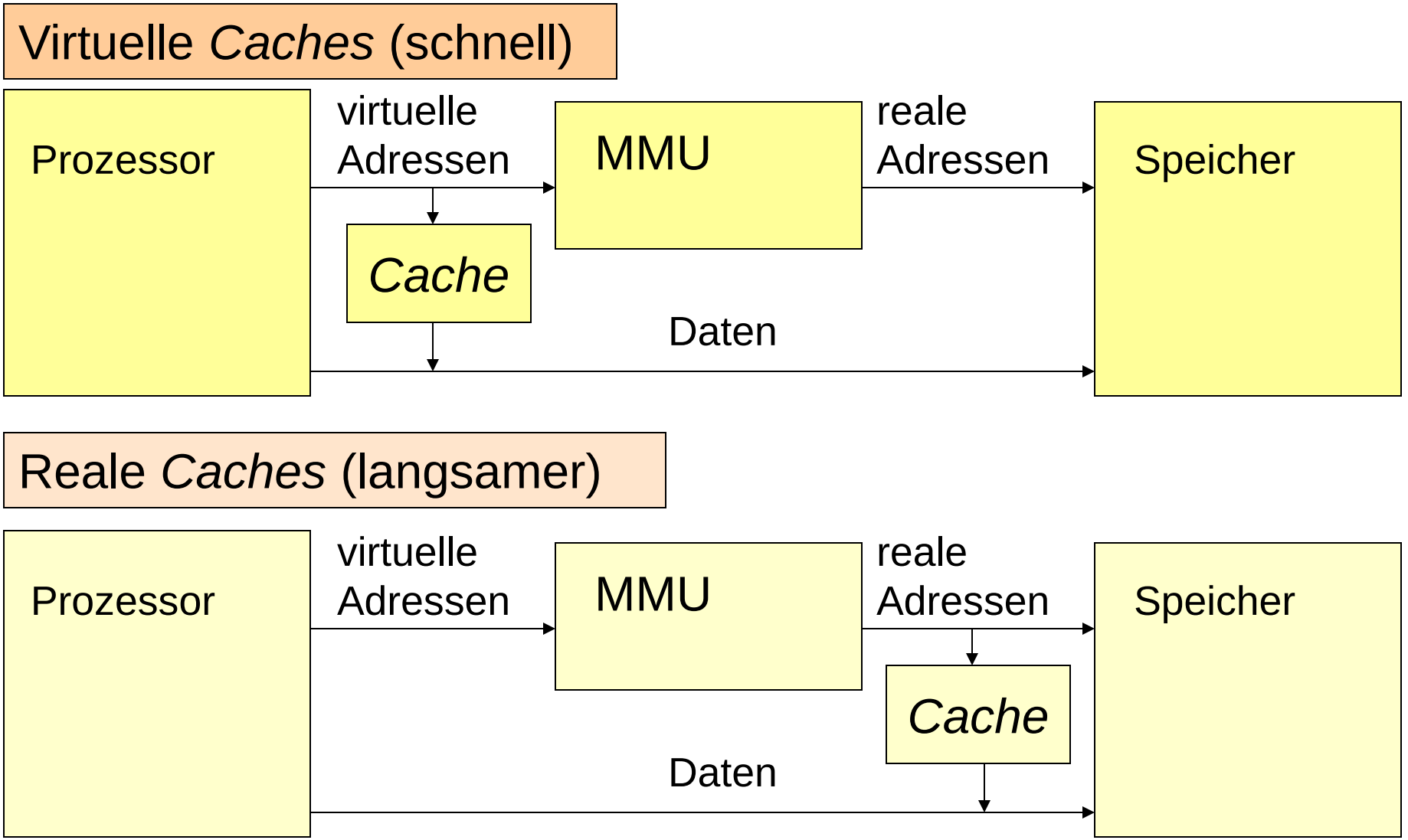
Aktuelle PCs verwenden virtuellen Speicher:

Virtuelle oder reales Caches?

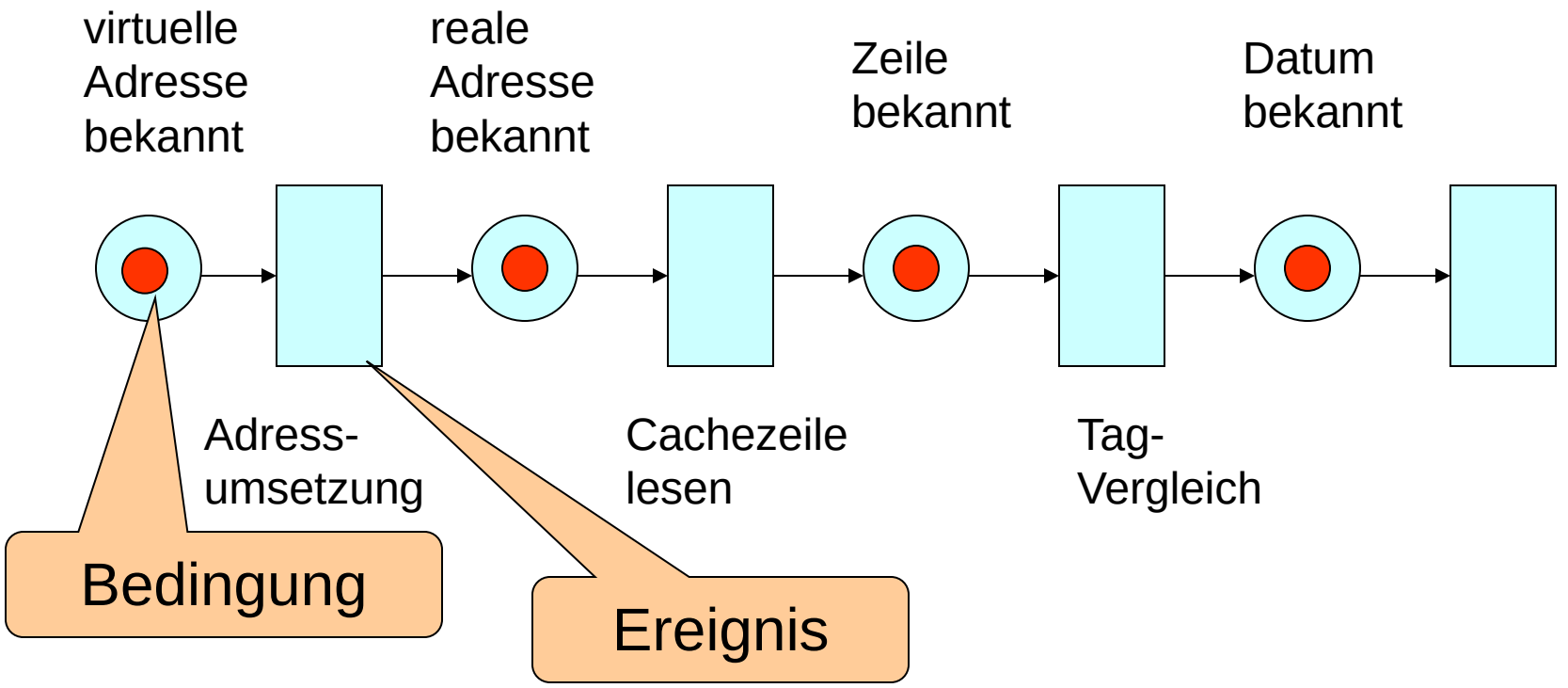
Siehe Kurs „Rechnerstrukturen“

Hier: Zusätzliche Betrachtung von Mischformen

# Zur Erinnerung aus RS

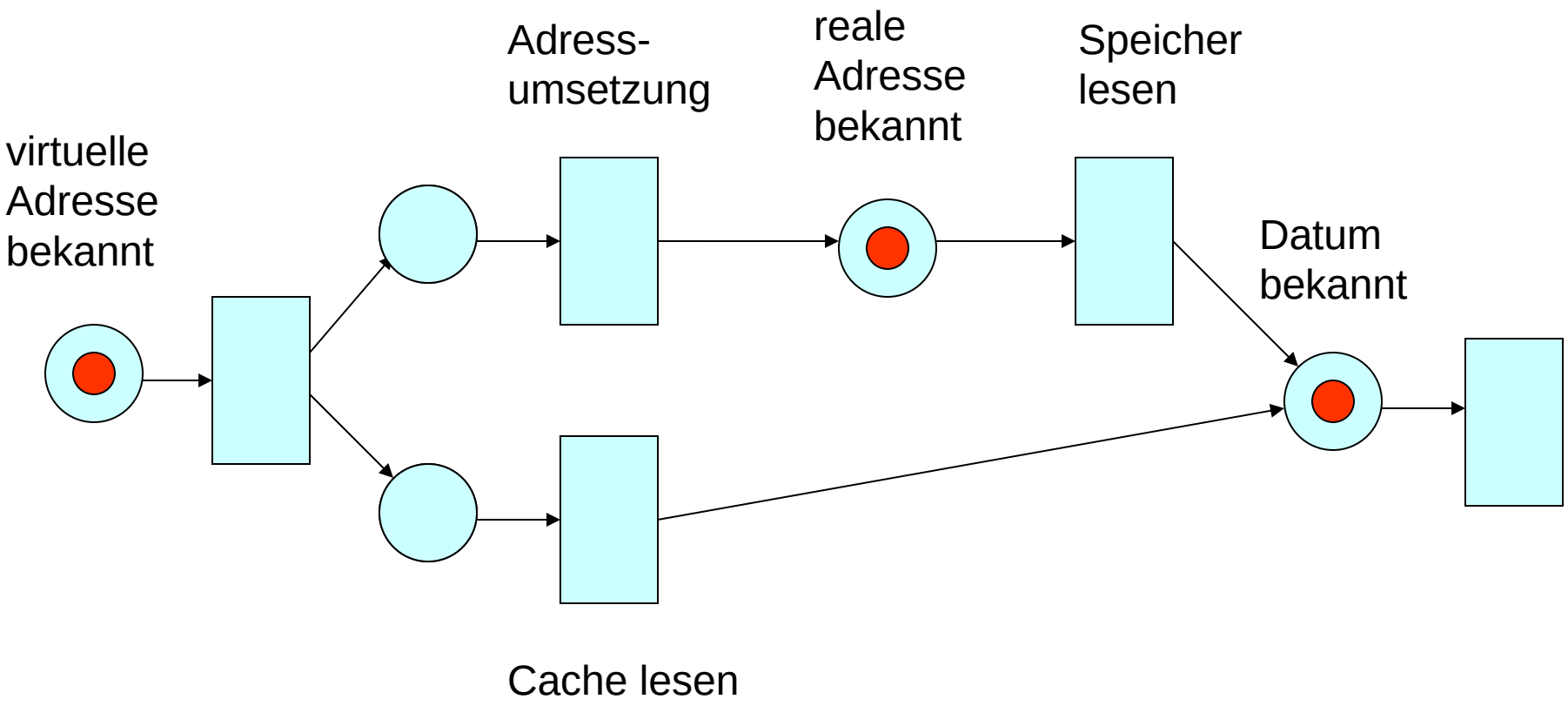


# Modell des Ablaufs bei realen Caches




Adressumsetzung und Lesen des Caches nacheinander

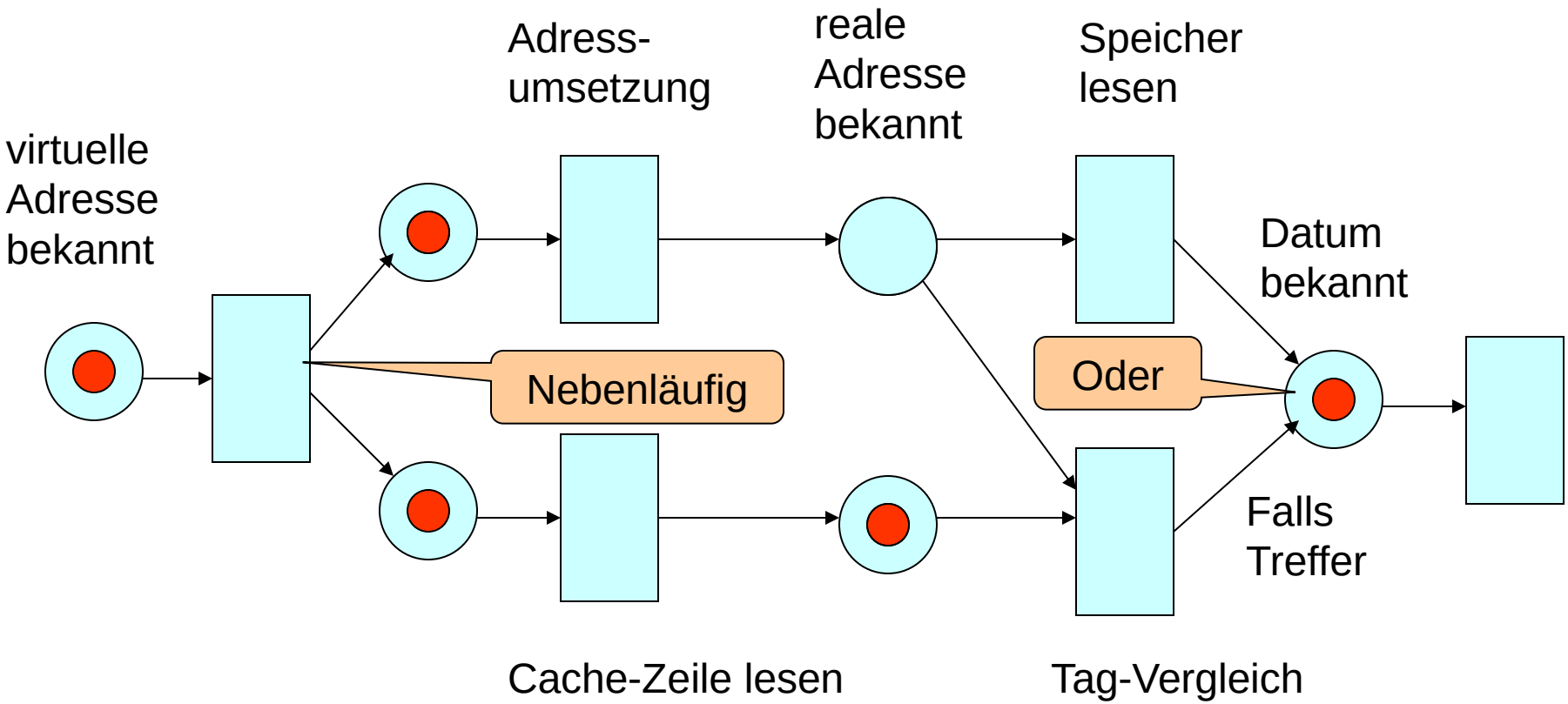
# Modell des Ablaufs bei virtuellen Caches




Lesen des Caches und Adressumsetzung „parallel“

# Mischform „*virtually indexed, real tagged*“

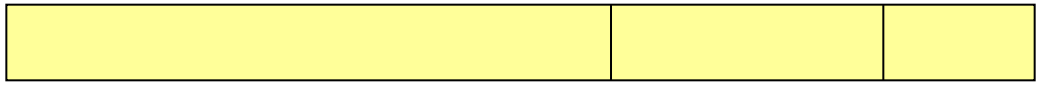
Indexwerte für virtuelle und reale Adressen identisch
☞ Zugriff auf realen Cache kann bereits während der Umrechung auf reale Adressen erfolgen.



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

Nachteil:

- 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.
- De facto bedeutet das, dass wir statt Seiten nunmehr „*Meta-Pages*“ als Einheiten der Speicherzuteilung nutzen. Damit wird aber die Speicherzuteilung sehr grob-granular.
- Beispiel: *direct mapped* $ mit 256 kB = $2^{18}$ B, mit 64 Bytes pro $-Eintrag ☞ 12 Bits für den Index, Meta-Pages von 256 kB.



# Unterschiedliche Kombinationsmöglichkeiten

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

# *Cache Bandwidth ↑: Multibanked Caches*

Blöcke werden über mehrere Speicherbänke verteilt

| Block-adresse | Bank 0 |
|---|---|
| 0 | |
| 4 | |
| 8 | |
| 12 | |

| Block-adresse | Bank 1 |
|---|---|
| 1 | |
| 5 | |
| 9 | |
| 13 | |

| Block-adresse | Bank 2 |
|---|---|
| 2 | |
| 6 | |
| 10 | |
| 14 | |

| Block-adresse | Bank 3 |
|---|---|
| 3 | |
| 7 | |
| 9 | |
| 15 | |

# *Cache Bandwidth* ↑: *Pipelined Cache Access*

*Cache* Zugriff über mehrere Stufen der Pipeline verteilen

☞ 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!

Beachte:

- *Cache*-Latenz ändert sich nicht, aber sie wird versteckt.

# *Miss Penalty ↓: Nonblocking Caches*

Prinzip: Überlappen der Ausführung von Befehlen mit Operationen auf der Speicherhierarchie

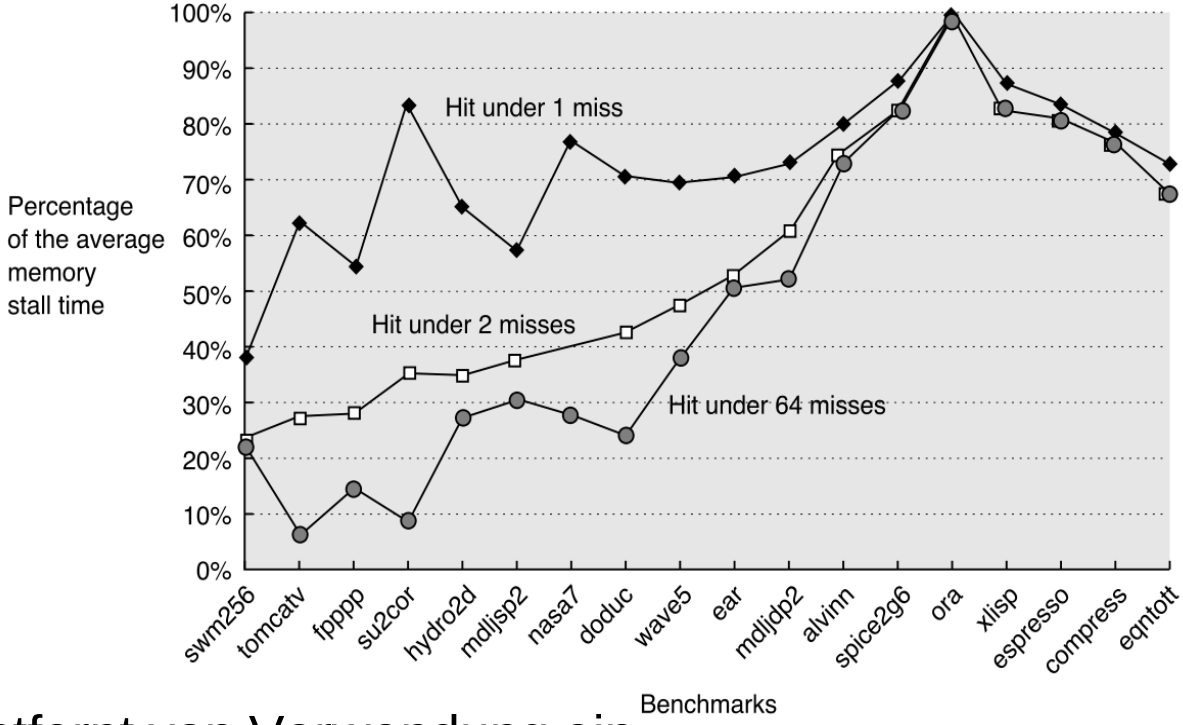
Betrachten CPUs mit *out-of-order completion/execution*
☞ Anhalten der CPU bei Fehlzugriff auf Cache nicht erforderlich (z.B. weiteren Befehl während Wartezeit holen)

Notwendig: *Cache* kann trotz vorangegangem (und ausstehenden) Fehlzugriffs 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 (SPEC92)

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




Verwendeter Compiler plant Ladeoperationen entfernt von Verwendung ein.
Jetzt mehrere ausstehende Zugriffe auf den gleichen Cache-Block möglich ☞ muss geprüft werden

# *Miss Penalty* ↓: Zusammenfassung von Schreibpuffern

Betrachte *write through Cache*

Schreibzugriffe in (kleinem) Puffer gespeichert und bei Verfügbarkeit des Speicherinterfaces ausgeführt
Hier: Daten Wort-weise 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

| Write address | V | | V | | V | | V | |
|---|---|---|---|---|---|---|---|---|
| 100 | 1 | Mem[100] | 0 | | 0 | | 0 | |
| 108 | 1 | Mem[108] | 0 | | 0 | | 0 | |
| 116 | 1 | Mem[116] | 0 | | 0 | | 0 | |
| 124 | 1 | Mem[124] | 0 | | 0 | | 0 | |

| Write address | V | | V | | V | | V | |
|---|---|---|---|---|---|---|---|---|
| 100 | 1 | Mem[100] | 1 | Mem[108] | 1 | Mem[116] | 1 | Mem[124] |
| | 0 | | 0 | | 0 | | 0 | |
| | 0 | | 0 | | 0 | | 0 | |
| | 0 | | 0 | | 0 | | 0 | |

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

Beachte:

- *Miss Rate*(L2) gemessen auf von L1 „übrigen" Zugriffen
- 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!

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* ↓: Priorisierung von Lese- über Schreibzugriffen

Idee: Lesezugriffe behandeln,
bevor Schreiboperation abgeschlossen

**Problem bei *write through*:**

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

Lösungsmöglichkeiten:

- Warten bis Schreibpuffer leer (kontraproduktiv)
- Inhalte des Schreibpuffers prüfen und mit Lesezugriff fortfahren, falls keine Konflikte (üblich in *Desktop/Server*)

**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 (Daten/Code) in *Cache* laden, bevor sie gebraucht werden

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

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*
- Beachte:
  - *Prefetch*-Instruktionen 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)
- In Alpha 21264 auch *prefetch* für Schreibzugriffe (falls angeforderter Block komplett geschrieben wird: *write hint* Befehl alloziert diesen im *Cache*, aber Daten nicht gelesen!)

# 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