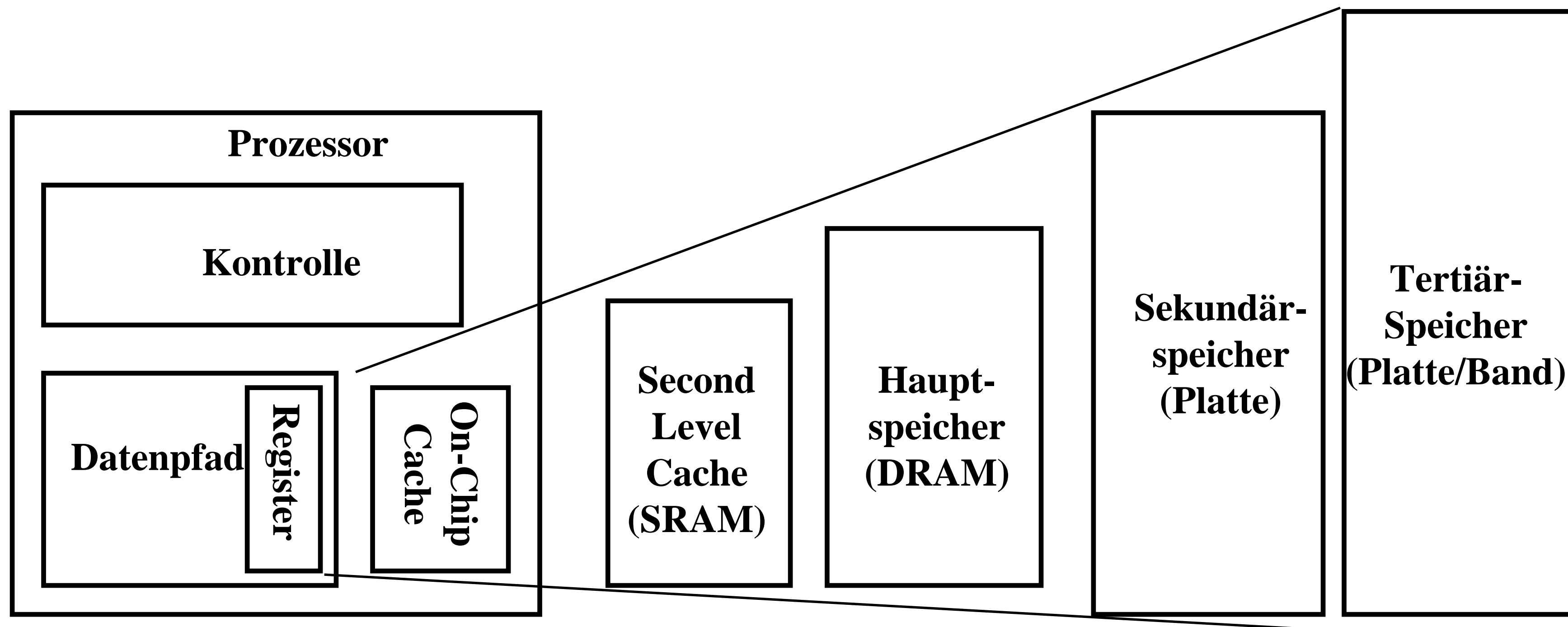
10. Speicherorganisation



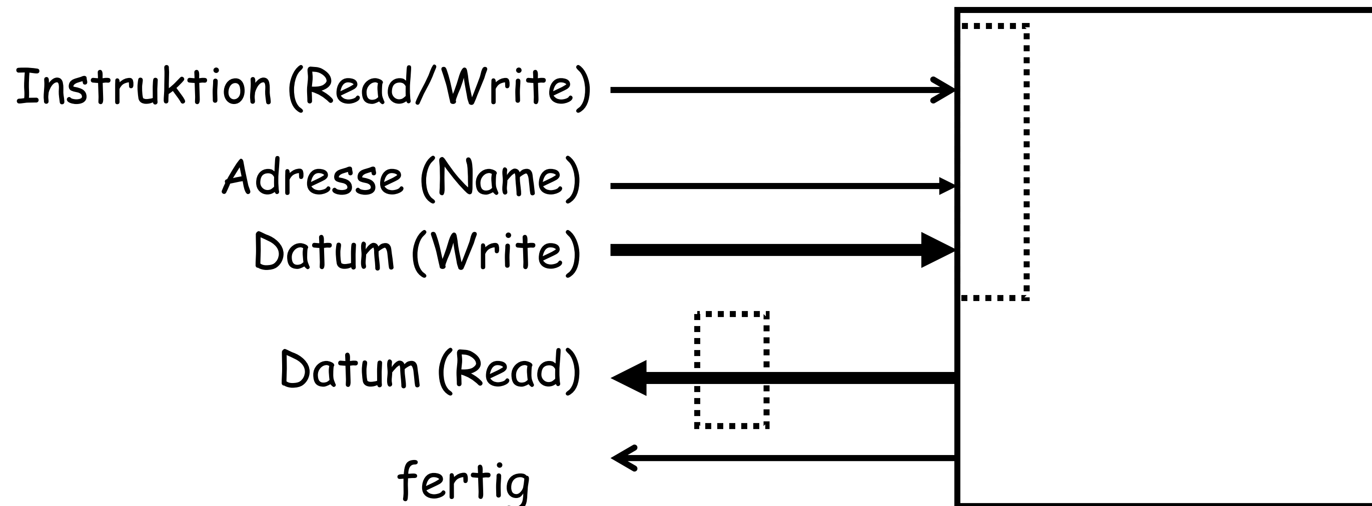
Speicherhierarchien

Cache-Speicher-Organisation

Hauptspeicher-Hintergrundspeicher-Schnittstelle

Abstrakte Sicht auf den Speicher

- Zuordnungen <Name, Wert>
  - Namen entsprechen meist Byteadressen
  - Werte sind häufig auf Byte-Vielfache ausgerichtet
- Folge von Lese- und Schreibinstruktionen
- Schreiben bindet einen Wert an eine Adresse
- Lesen einer Adresse liefert den zuletzt an diese Adresse gebundenen Wert zurück



“Beschleunigung” der Prozessor/Speicherkommunikation (Wh)

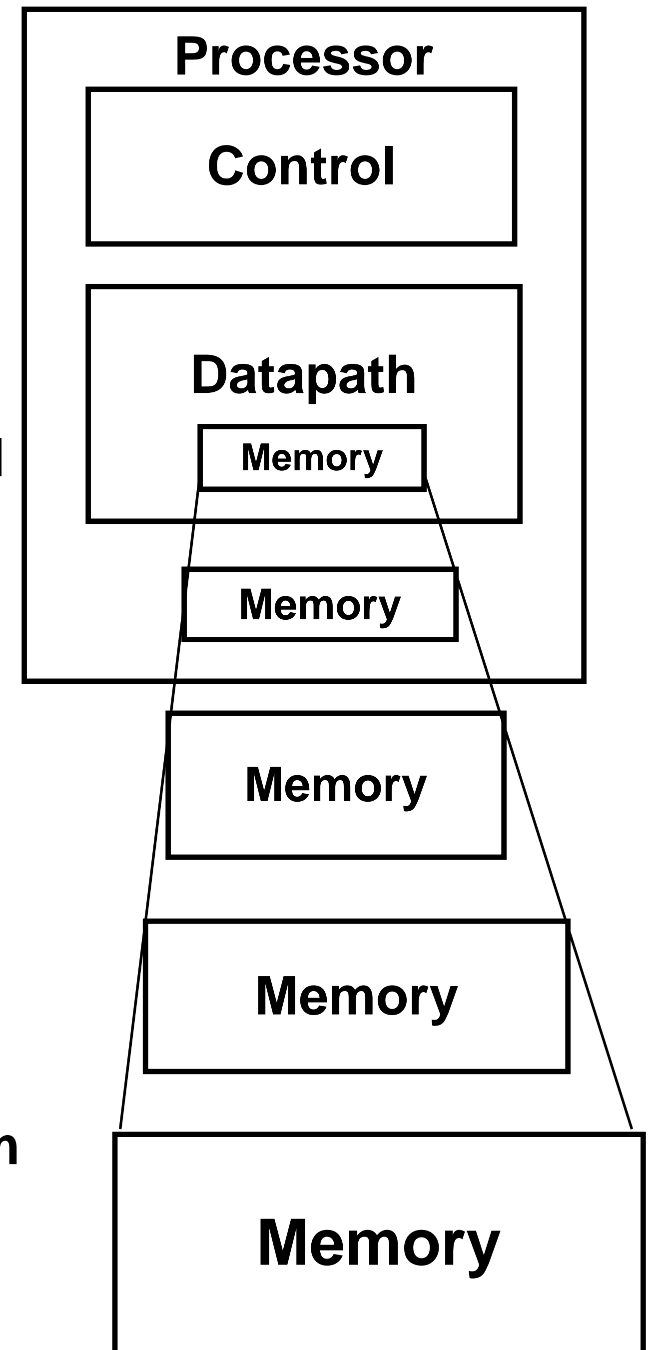
Ziel: Arbeitsspeicher so schnell wie schnellste verfügbare Speicherchips und so groß wie die größten

Lösung:
hierarchische Speicherorganisation

- Prozessor kommuniziert mit top-down organisierter **Folge von Speichern**
- mit wachsender “Entfernung” vom Prozessor
  - steigen Größe und Zugriffszeit
  - fällt der Preis (pro Bit)
- **Ausschnittshierarchie:**
  Alle Daten einer Ebene sind auch in der darunterliegenden Ebene gespeichert.

schnell
klein
teuer

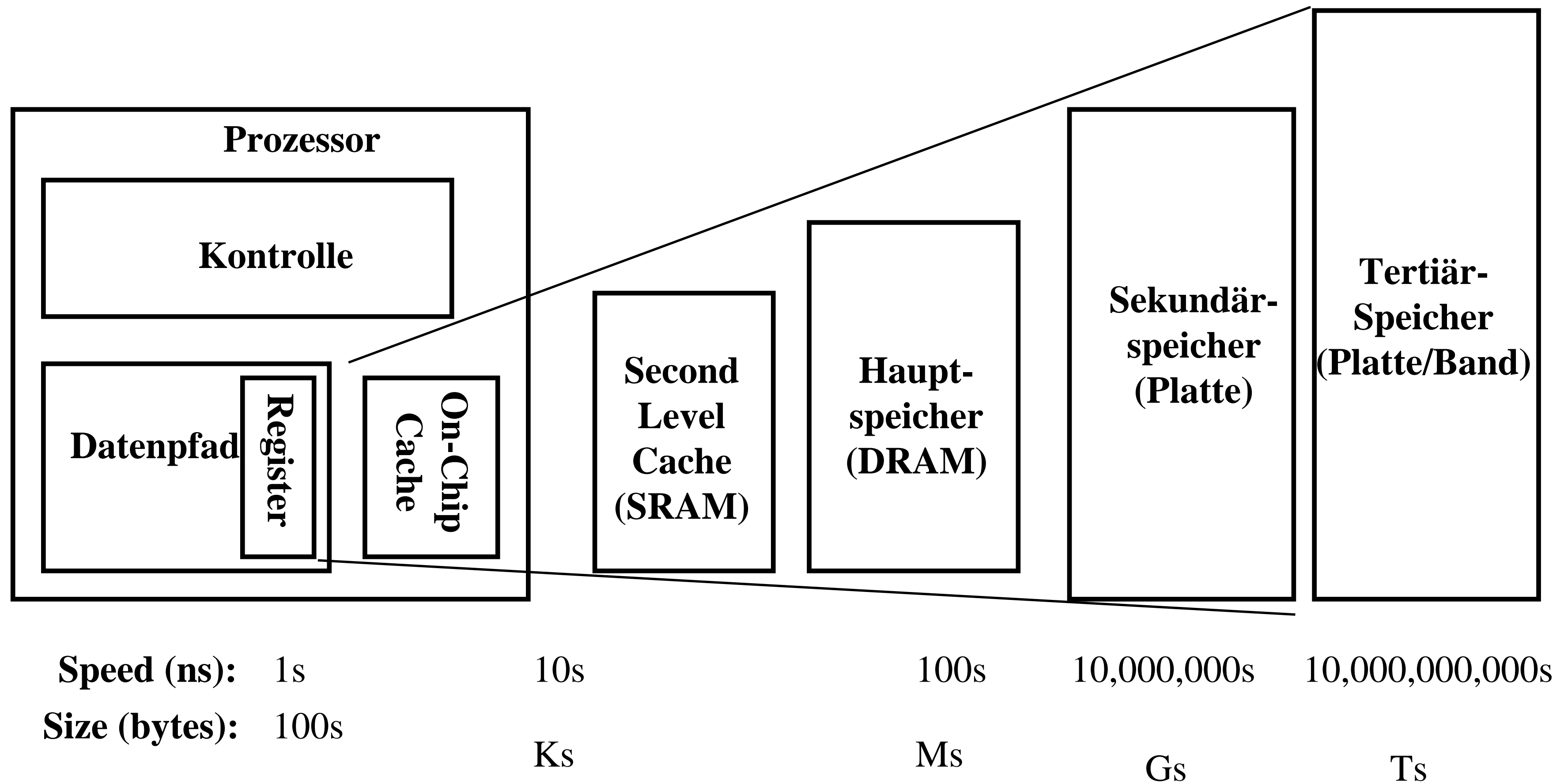
langsam
groß
billig



Eine moderne Speicherhierarchie

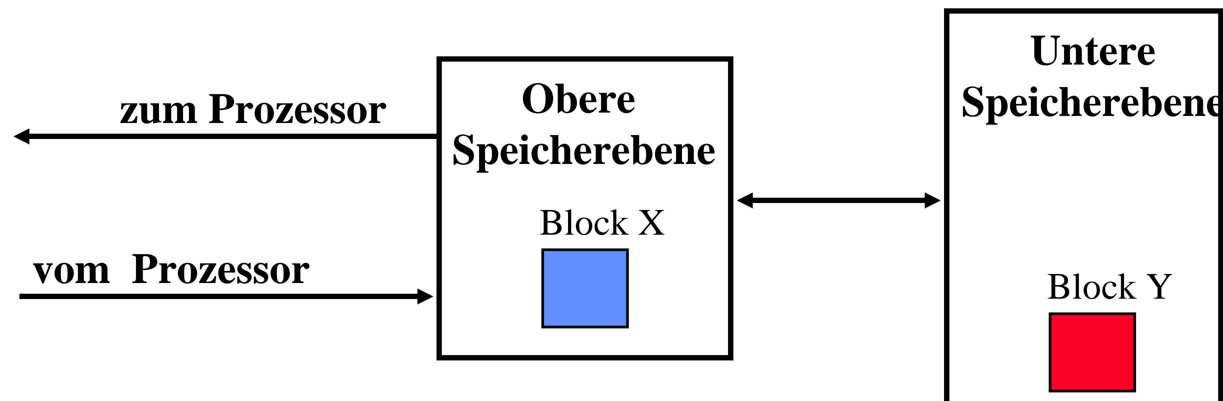
Ausnutzung des Lokalitätsprinzips

- Temporale Lokalität: Halte zuletzt benutzte Daten beim Prozessor
- Räumliche Lokalität: Bewege Blöcke benachbarter Speicherzellen Richtung Prozessor



Grundbegriffe der Speicherhierarchie

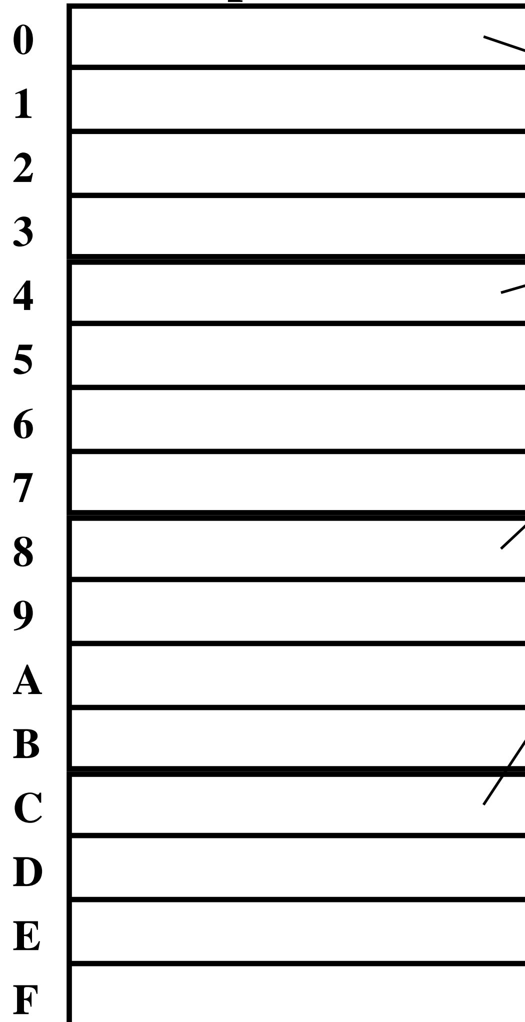
- **Treffer (hit):** Die Daten befinden sich in einem Block der oberen Ebene.
  - **Trefferquote:** Anteil der Treffer an allen Speicherzugriffen
  - **Trefferzeit:** Zugriffszeit auf obere Speicherebene
- **Fehlschlag (miss):** Die Daten müssen von unterer Speicherebene nachgeladen werden.
  - **Fehlschlagsquote** = 1 - Trefferquote
  - **Fehlschlagszeit:** Zeit für das Nachladen eines Blocks + Zeit für den Zugriff auf den Block
- Trefferzeit << Fehlschlagszeit (500 instructions on 21264!)



Kernfragen der Speicherhierarchie

1. **Wo kann ein Block in der oberen Ebene platziert werden?**
   (Blockplatzierung)
2. **Wie kann ein Block in der oberen Ebene gefunden werden?**
   (Blockidentifikation)
3. **Welcher Block sollte bei einem Fehlschlag ersetzt werden?**
   (Blockersetzung)
4. **Was geschieht beim Schreiben?**
   (Schreibstrategie)

Blockplatzierung: Direkte Abbildung (direct mapping)



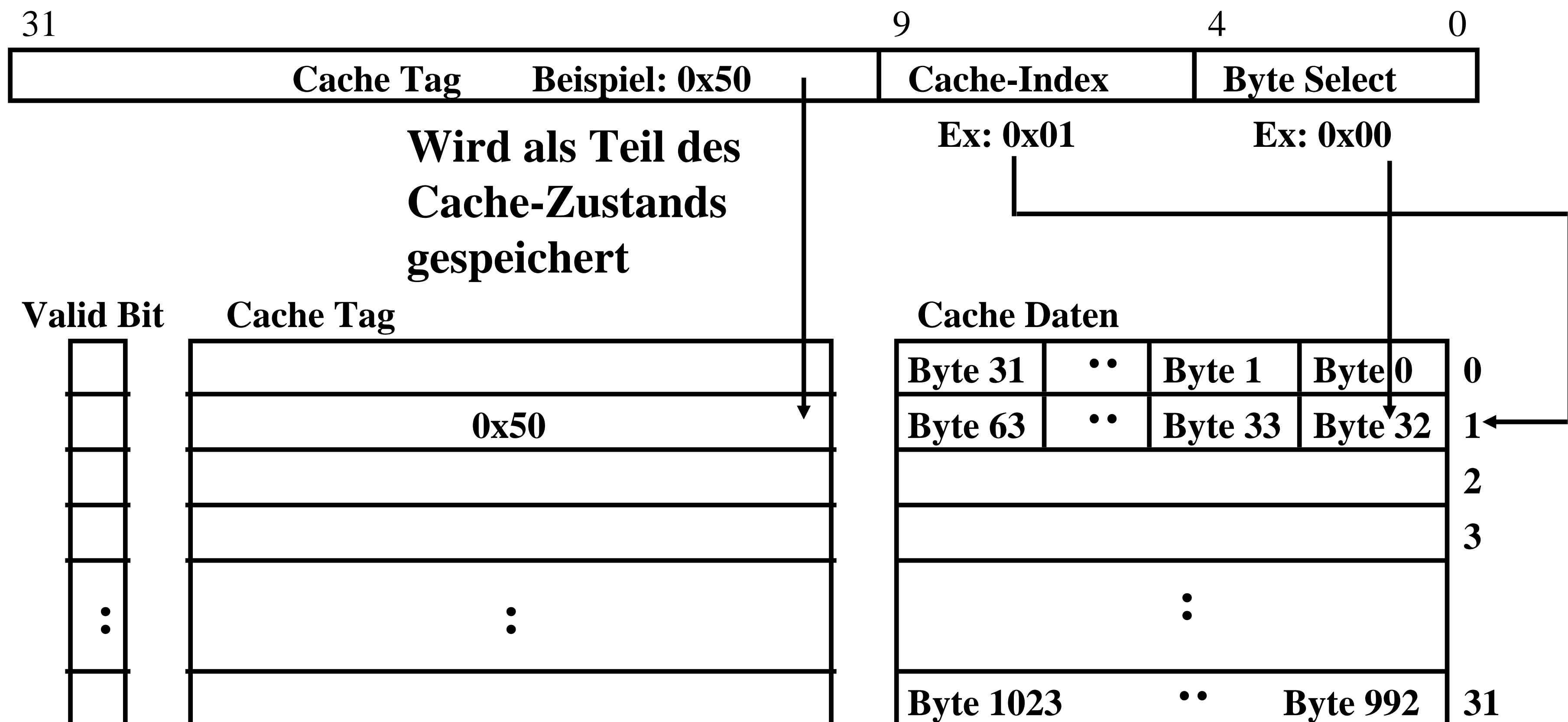
4 Byte direkt abgebildeter Cache

- Cache-Index 0 kann belegt werden mit
  - Speicherblockadressen 0, 4, 8, ... etc.
  - allgemein: Speicherblock mit zwei Nullen in den niedrigwertigsten Bits der Adresse
  - Adresse[1:0] => Cache-Index
- Welcher dieser Blöcke sollte im Cache platziert werden?
- Wie kann man bestimmen, welcher Block sich im Cache befindet?

Beispiel: 1 KB Cache mit direkter Abbildung, 32B Blöcke

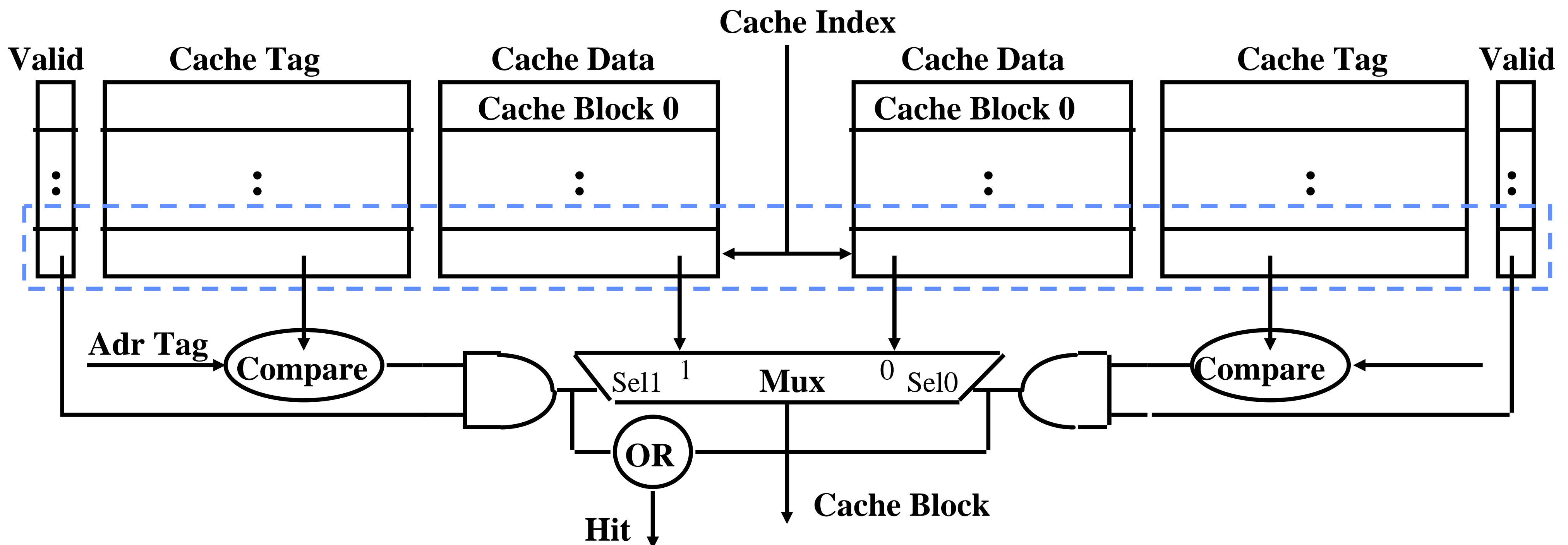
Für einen 2^N Byte-Cache:

- Die oberen (32 - N) Bits sind das sogenannte “Cache Tag”
- Die niedrigsten M Bits dienen der Byte Selektion (Block Größe = 2^M)



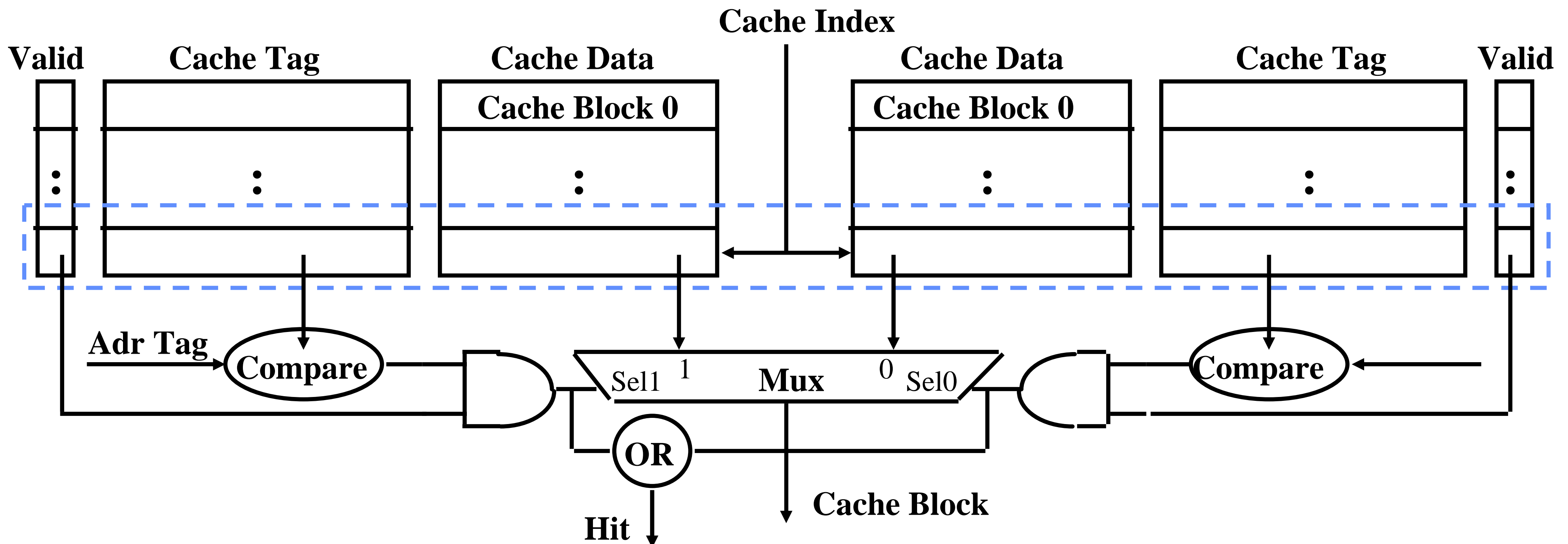
Mengenassoziative Caches (N-way Set Associative Caches)

- N-fache Mengenassoziativität: N Einträge pro Cache-Index
  - N Caches mit direkter Abbildung, die parallel operieren (N liegt meist zwischen 2 und 4)
- Beispiel: zweifach mengenassoziativer Cache
  - Der Cache-Index bestimmt eine “Menge” von Blöcken im Cache
  - Die beiden Tags werden parallel verglichen.
  - Die Daten werden aufgrund des Tagvergleichs selektiert.



Nachteil mengenassoziativer Caches

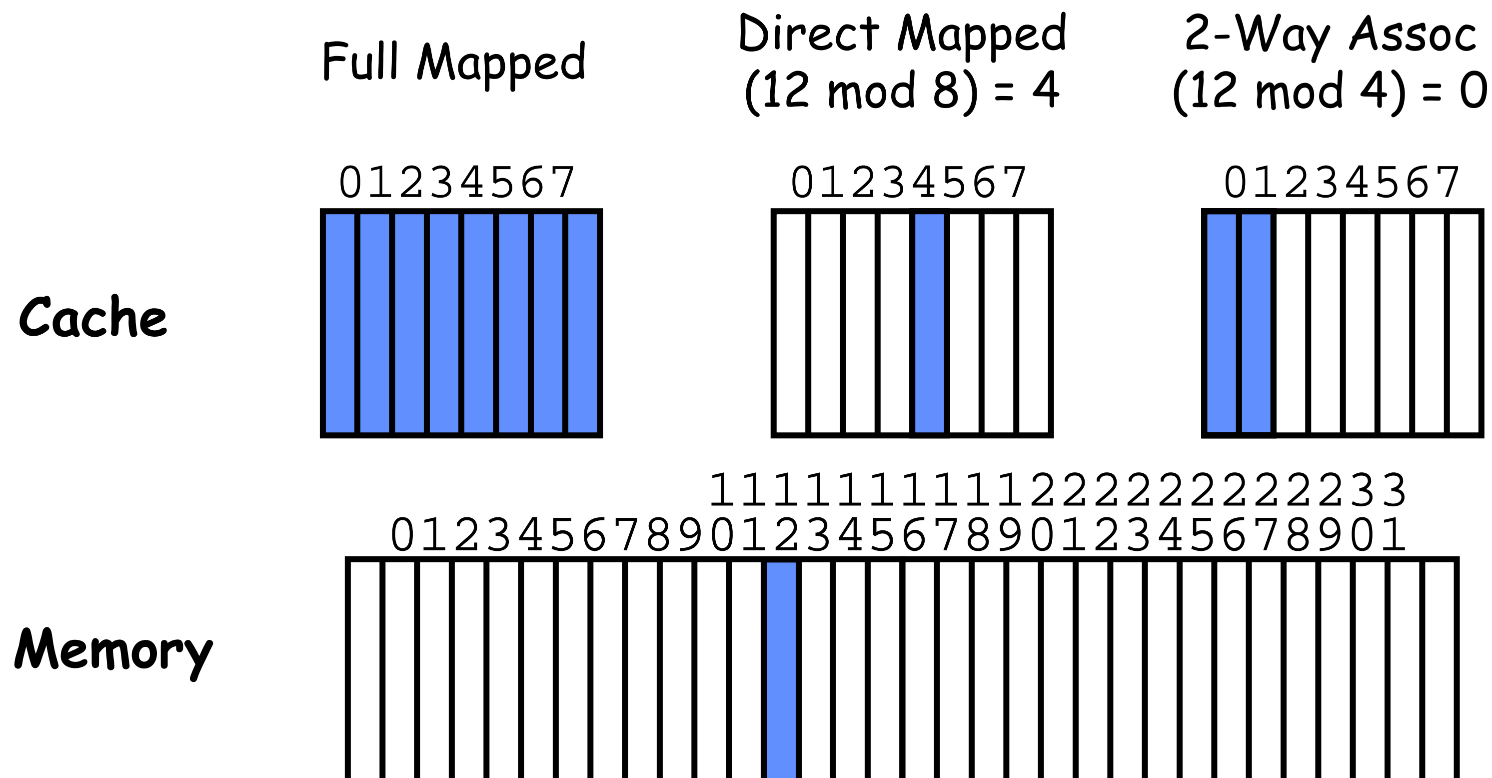
- **N-fach mengenassoziativer Cache vs. direkt abgebildeter Cache:**
  - N Komparatoren vs. 1
  - Verzögerung des Datenflusses durch extra MUX
  - Daten werden NACH Treffer/Fehlschlag bereitgestellt.
- **Bei direkt abgebildetem Cache steht der Block DIREKT zur Verfügung**
  - Unmittelbare Fortsetzung mit Trefferannahme; bei Fehlschlag nachträgliche Korrektur



Zusammenfassung: Blockplatzierungen

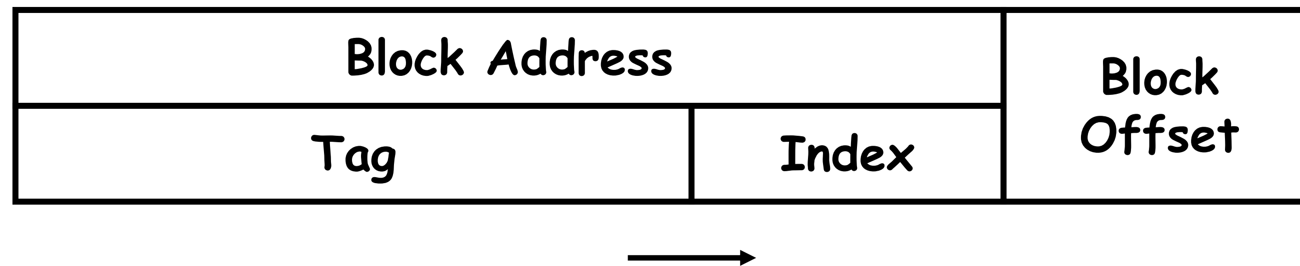
Platziere Block 12 in 8 Block-Cache:

- **voll assoziativ, direkte Abbildung, 2-fache Mengenassoziativität**
- **Mengenassoziative Abbildung = Blockadresse modulo Mengenzahl**



Blockidentifikation

- Überprüfung des Blocktags
  - Index und Blockoffset brauchen nicht geprüft zu werden.
- Wachsende Assoziativität verkleinert den Index und vergrößert das Tag.



Blockersetzung

- Einfach bei direkter Abbildung
- bei Mengenassoziativität oder voller Assoziativität
  - zufällig (random)
  - LRU (Least Recently Used)

Fehlschlag-quoten für VAX Traces

Assoz:	zweifach		vierfach		achtfach	
Größe	LRU	Ran	LRU	Ran	LRU	Ran
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

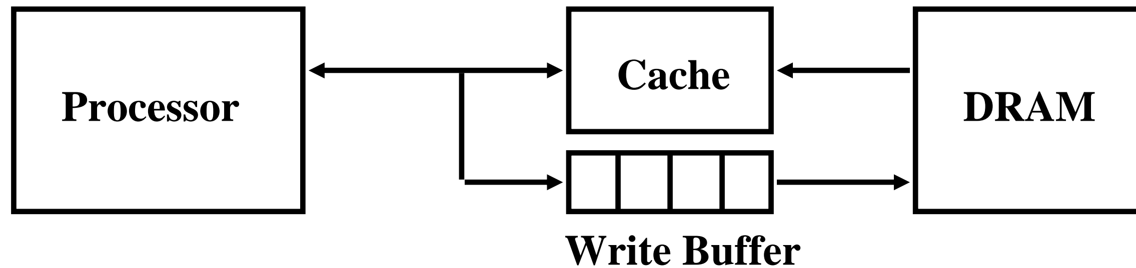
Schreibstrategien

- ***Write through*** —Die Daten werden sowohl in den Cache als auch in die nächsttiefere Ebene geschrieben.
- ***Write back*** — Die Daten werden nur in den Cache geschrieben. Der modifizierte Block wird bei der Ersetzung in den Hauptspeicher zurückgeschrieben.
  - Ist ein Block sauber (unverändert) oder schmutzig (modifiziert)?

- Pros und Cons?
  - WT:
    - » Hauptspeicher und Cache sind immer konsistent
    - » Lesefehlschläge können keine Schreibaktionen auslösen.
  - WB:
    - » keine mehrfachen Speicherschreibzugriffe auf dieselbe Adresse

- WT wird mit Schreibpuffern eingesetzt, damit der Prozessor nicht auf langsame Speicherebenen warten muss.

Schreibpuffer für Write Through



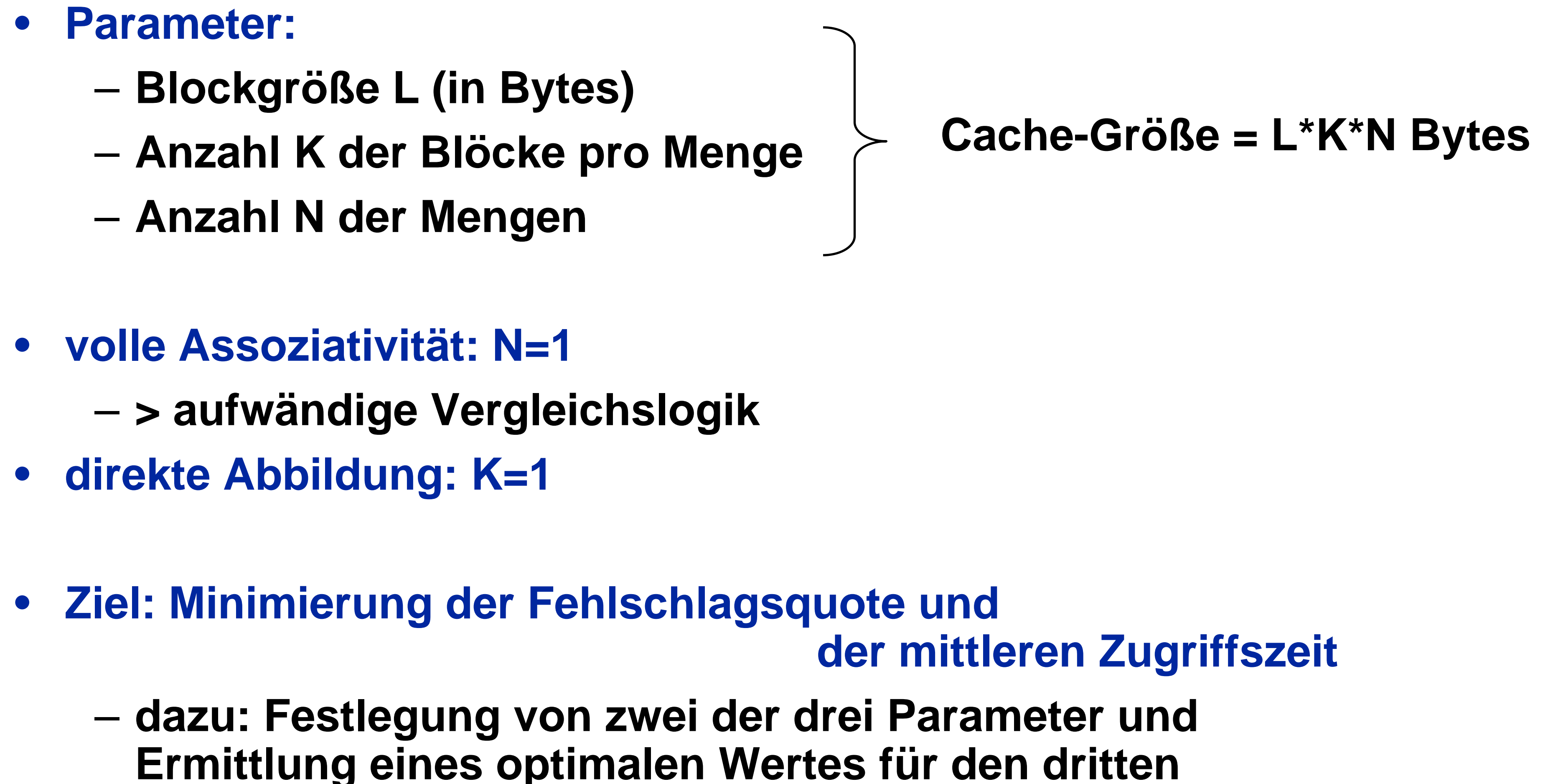
- Schreibpuffer zwischen Cache und Speicher
  - Prozessor schreibt Daten in Cache und Schreibpuffer
  - Speicherkontrolle schreibt Pufferinhalt in den Speicher
- Schreibpuffer arbeitet FIFO (first-in-first-out):
  - Typische Zahl von Einträgen: 4
  - Funktioniert gut, falls: Schreibhäufigkeit << 1 / DRAM write cycle

Strategien bei Schreibfehlschlägen (write misses)

- write allocate oder fetch on write
  - Nachladen des Blocks
- no-write allocate oder write around
  - Schreibzugriff im Hauptspeicher ohne Nachladen des Blocks

- Meist Kombination von
  - write through und write around sowie
  - write back und fetch on write

Cache-Entwurf und Optimierung



Analyse von Caches

- Berechnung der Fehlschlagsquoten anhand von Trace-Analysen
- Trace
  - Protokoll der Adressfolge von Speicherzugriffen
  - hergestellt mithilfe eines Simulationsprogramms oder Hardware-Monitoren
  - Eingabe eines Cache-Simulationsprogramms

- Damit Initialisierungsfehlschläge vernachlässigbar werden, muss ein Trace eine gewisse Mindestlänge haben:

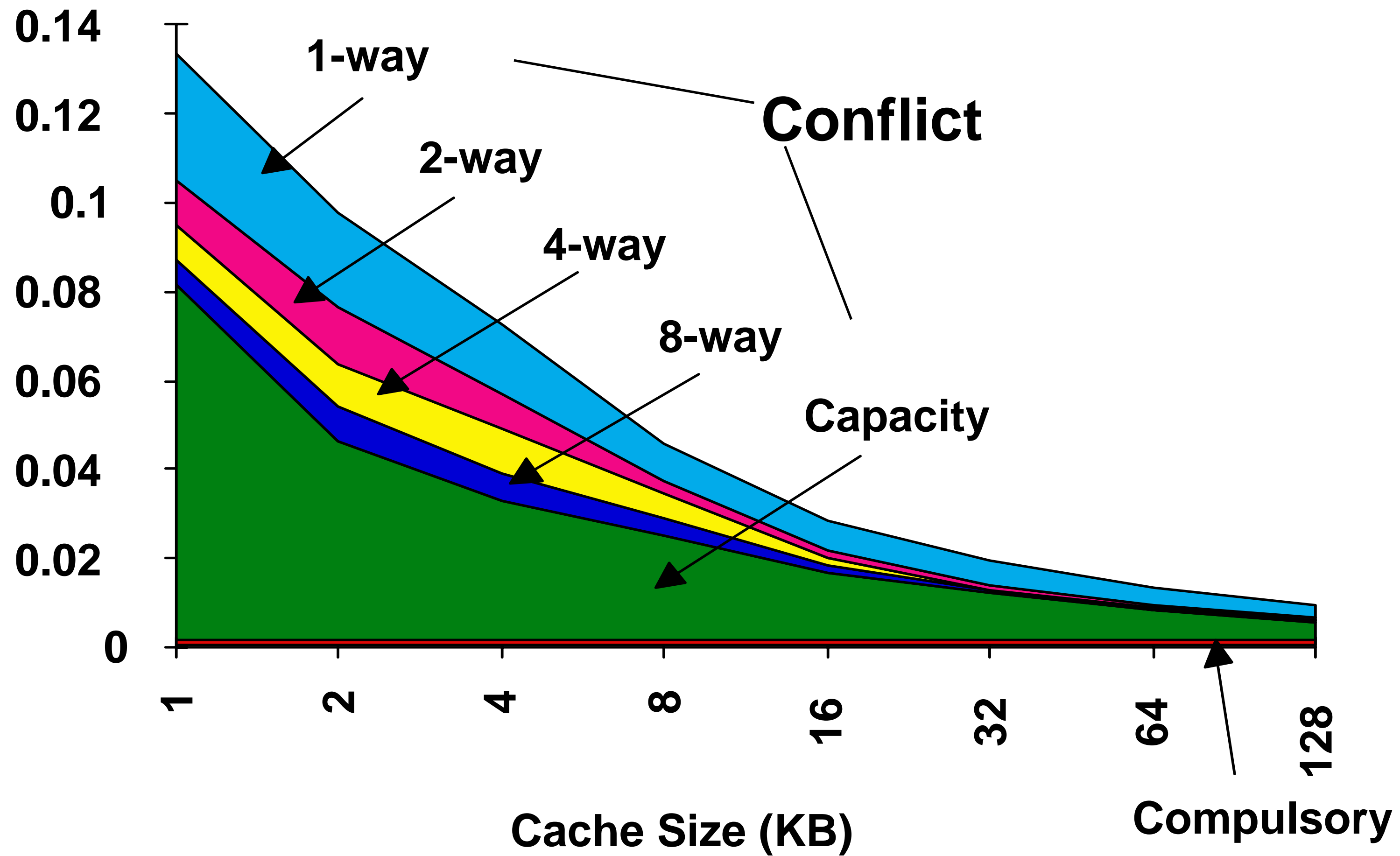
  Bsp: zweifach-mengenassoziativer Cache mit 256 Mengen, d.h. 512 Blöcken
  Annahme: Tracelänge: 100.000, Fehlschlagquote 1% (=1000 Cache-Fehler)
  Da darunter ca. 50% Initialisierungsfehler sind, liefert der Trace kein realistisches Ergebnis.

  Um Verfälschung unter 5% zu halten, müsste der Trace mehr als 1 Mio Adressen enthalten. In der Praxis umfassen Traces meist 5-10 Mio Referenzen.

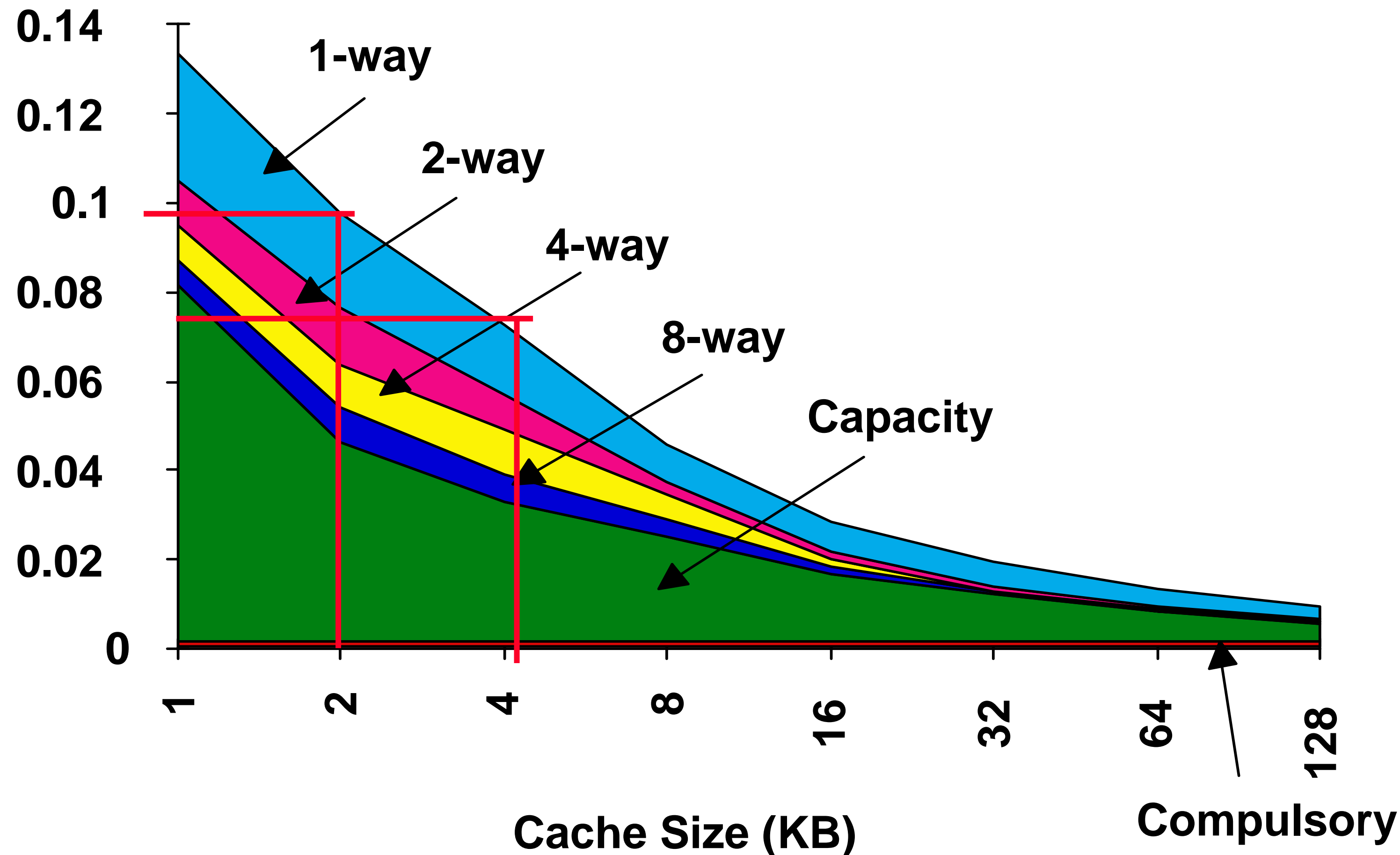
Klassifikation von Fehlschlagursachen – Die 3 Cs

- ***Compulsory*** —> **Unvermeidbar**
  Der erste Zugriff auf einen Block ist immer ein Fehlschlag.
  -> Kaltstart-Fehlschlag oder Erstzugriffsfehlschlag
  (Sogar bei unendlich großem Cache vorhanden)
- ***Capacity*** —> **Kapazitätsbedingt**
  Auslagerung und Neuladen aktueller Blöcke, falls der Arbeitsbereich eines Programms die Cache-Größe übersteigt.
  (Fehlschläge in voll assoziativem endlichem Cache)
- ***Conflict*** —> **Konflikte/Kollisionen**
  Bei direkter Abbildung oder Mengenassoziativität können mehr Blöcke in eine Menge abgebildet werden, als Plätze vorhanden sind. In diesem Fall entstehen (zusätzlich zu kapazitätsbedingten und Kaltstartfehlschlägen) Kollisionsfehlschläge oder Interferenzfehlschläge.
  (Fehlschläge in N-fach assoziativem Cache begrenzter Größe)

3Cs - Absolute Fehlschlagquoten (SPEC92)



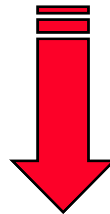
Einfluss der Cache-Größe



- **Faustregel:**
  Verdopplung der Cache-Größe => 25% Verringerung der Fehlschlagquote
- **Was wird dabei reduziert?**

Verbesserung der Leistung eines Cache-Speichers

Mittlere Speicherzugriffszeit = Trefferzeit + Fehlschlagsquote * Nachladezeit



1. Reduziere die Fehlschlagsquote
2. Reduziere die Nachladezeit oder
3. Reduziere die Trefferzeit.

Cache-Organisation?

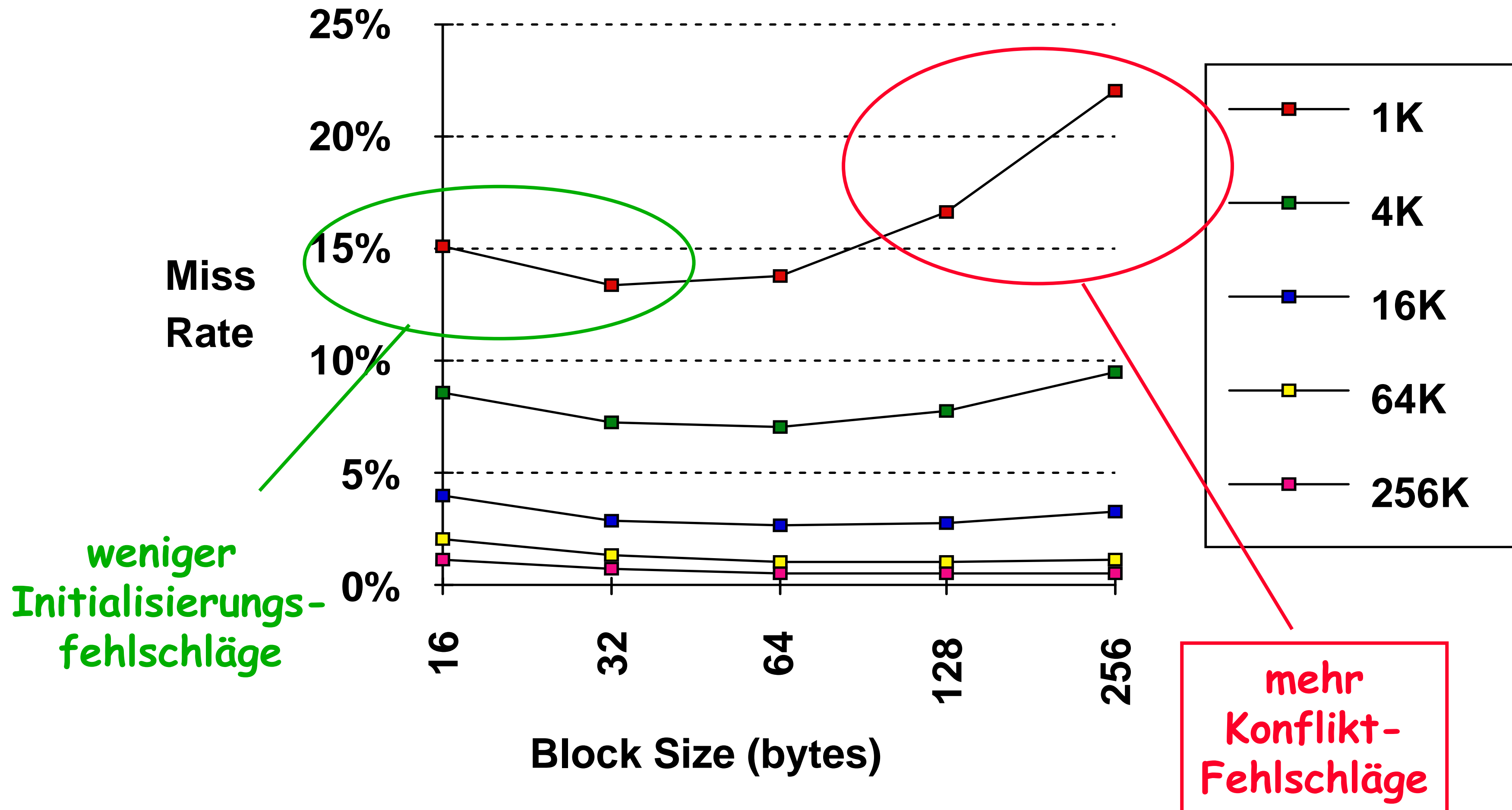
Angenommen, die Cache-Größe werde nicht verändert. Was geschieht, wenn:

1) die Blockgröße
2) die Assoziativität
3) der Compiler

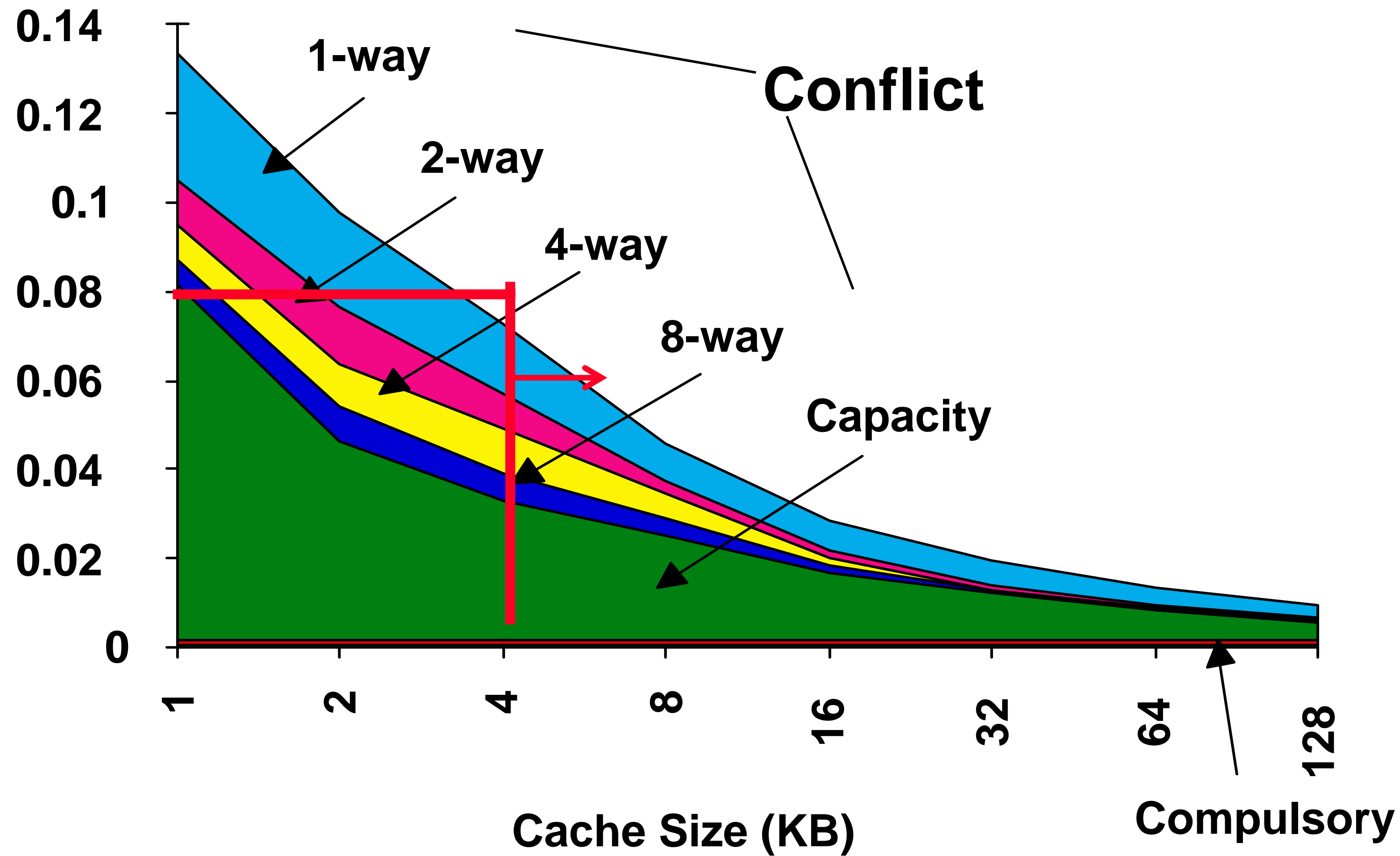
verändert werden?

Welches der 3 Cs ist jeweils betroffen?

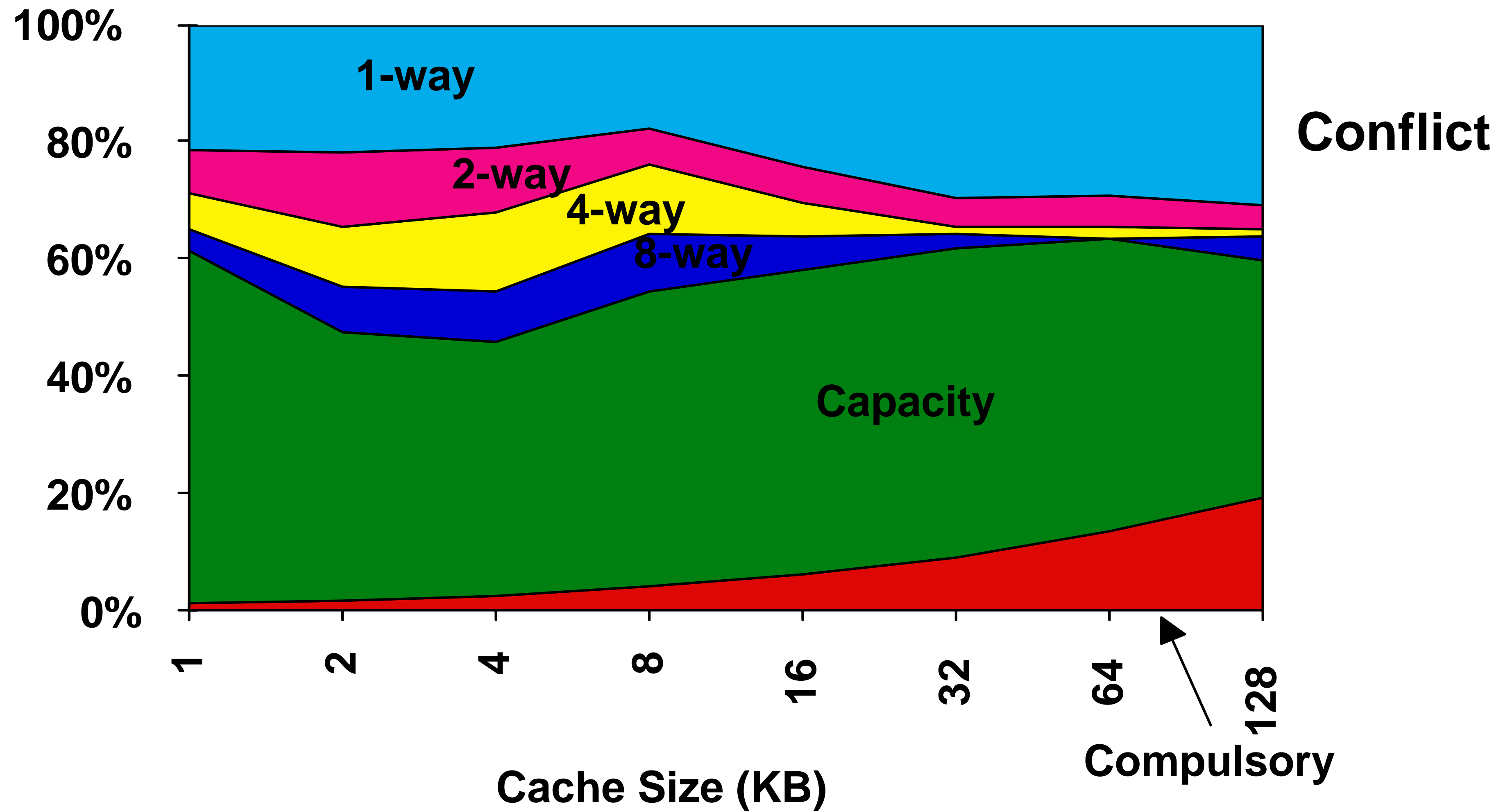
Vergrößerung der Blöcke
(bei fester Cache-Größe und Assoziativität)



Erhöhung der Assoziativität



3Cs Relative Fehlschlagquote



Beispiel: Mittlere Speicherzugriffszeit vs. Assoziativität

Annahme: Taktdauer = 1.10 bei 2-facher, 1.12 bei 4-facher, 1.14 bei 8-facher vs. Taktdauer bei direkter Abbildung, Nachladezeit = 10 Takte

Mittlere Speicherzugriffszeit

Cache-Größe (KB)	Assoziativität 1-fach	2-fach	4-fach	8-fach
1	2.33	2.15	2.07	2.01
2	1.98	1.86	1.76	1.68
4	1.72	1.67	1.61	1.53
8	1.46	1.48	1.47	1.43
16	1.29	1.32	1.32	1.32
32	1.20	1.24	1.25	1.27
64	1.14	1.20	1.21	1.23
128	1.10	1.17	1.18	1.20

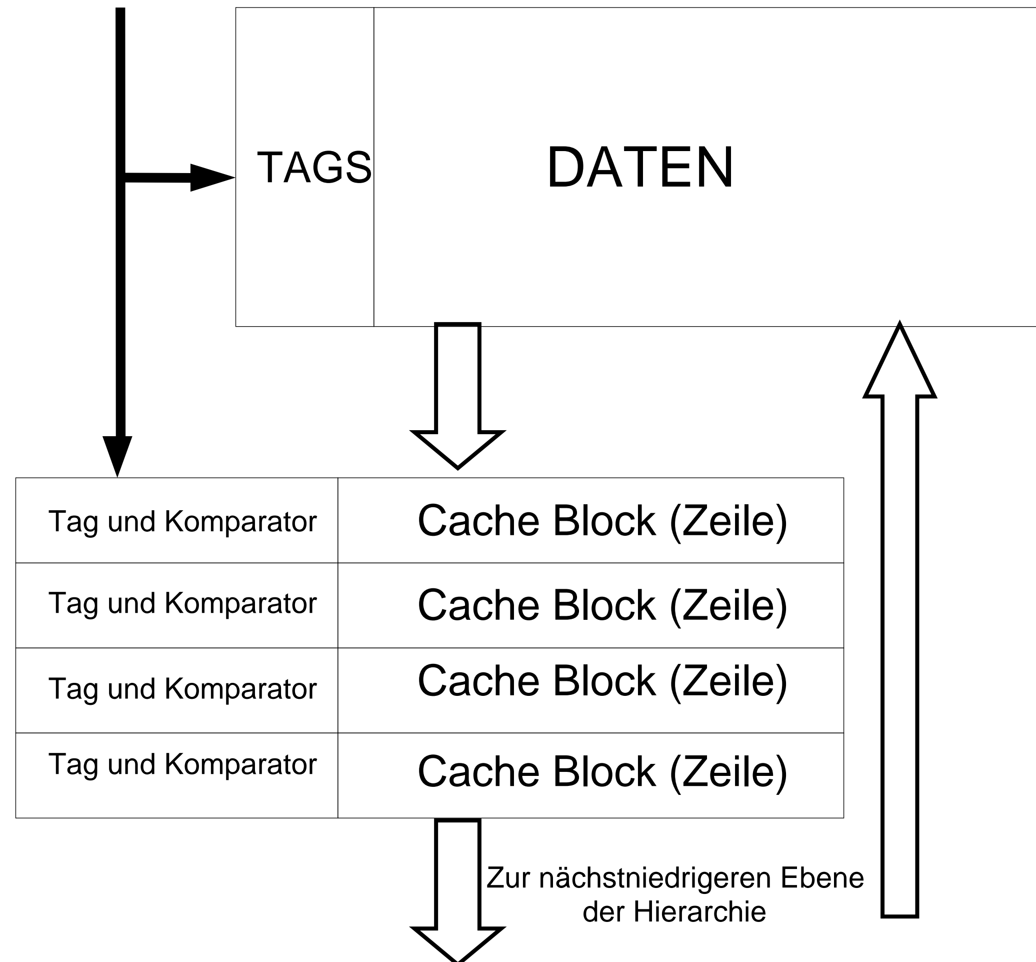
Fehlschlagquoten

Cache-Größe (KB)	Assoziativität 1-fach	2-fach	4-fach	8-fach
1	0,133	0,105	0,095	0,087
2	0,098	0,076	0,064	0,054
4	0,072	0,057	0,049	0,039
8	0,046	0,038	0,035	0,029
16	0,029	0,022	0,020	0,018
32	0,020	0,014	0,013	0,013
64	0,014	0,010	0,009	0,009
128	0,010	0,007	0,006	0,006

Rot bedeutet, dass die mittlere Zugriffszeit trotz wachsender Assoziativität nicht sinkt.

Victim Caches: Schnelle Treffer + Wenige Konflikte

- Wie kann man die schnelle Zeit eines Caches mit direkter Abbildung erhalten und trotzdem Kollisionshazards vermeiden?
- Stelle Puffer für aus dem Cache geladene Daten zur Verfügung.
- Jouppi [1990]: Ein Victim-Cache mit nur vier Einträgen entfernte zwischen 20% und 95% aller Konflikte in einem 4 KB Datencache mit direkter Abbildung.
- Verwendet in Alpha und HP Maschinen



Pseudo-Assoziativität

- **Ziel:** **Trefferzeit direkt-abgebildeter Caches und Fehlschlagquote zweifach-assoziativer**
- **Methode: Teile Cache in zwei Hälften und teste bei Fehlschlag, ob Block in zweiter Hälfte liegt => Pseudo-Treffer (langsamer Treffer)**

- **Nachteil:**
  Fließbandverarbeitung schwieriger bei Trefferzeiten von einem oder zwei Takten
  - **besser für Caches, die nicht unmittelbar den Prozessor bedienen (ab Level 2)**
  - **Verwendet in MIPS R1000 L2 cache, ähnlich auch in UltraSPARC**

Vorausladen (Prefetching) von Instruktionen und Daten per Hardware

- Instruktionen
  - Alpha 21064 lädt z.B. bei einem Fehlschlag zwei Blöcke.
  - Der zusätzliche Block wird in einem Strompuffer (stream buffer) platziert.
  - Bei einem Fehlschlag wird zunächst der Strompuffer überprüft.

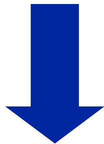
- Daten
  - Jouppi [1990] 1 Datenstrompuffer fing 25% der Fehlschläge eines 4KB Cache auf, 4 Puffer sogar 43%.
  - Palacharla & Kessler [1994] konnten für wissenschaftliche Programme mit 8 Strompuffern zwischen 50% und 70% der Fehlschläge von zwei 64KB, 4-fach mengenassoziativen Caches auffangen.
- Prefetching baut darauf, zusätzliche Speicherbandbreite ohne Nachteile nutzen zu können.

Compiler Optimierungen

- **Instruktionen**
  - Umordnung von Prozeduren im Speicher zur Reduktion von Kollisionsfehlschlägen
  - Analyse mit Profiling Tools um Konflikte zu analysieren und zu umgehen
- **Daten**
  - *Arrayverschmelzung (Merging Arrays)*:
    - » Verbesserung räumlicher Lokalität
    - » einzelnes Array mit zusammengesetzten Elementen statt zwei Arrays
  - *Schleifenumordnung (Loop Interchange)*:
    - » geschachtelte Schleifen so umordnen, dass auf die Daten in der Anordnung im Speicher zugegriffen wird
  - *Schleifenverschmelzung (Loop Fusion)*:
    - » Zusammenfügen von Schleifen mit dem selben Indexbereich und gemeinsamen Variablen
  - *Blockbildung (Blocking)*:
    - » Verbesserung der zeitlichen Lokalität durch Zugriff auf Datenblöcke statt langer Zeilen oder Spalten

Beispiel zur Array Verschmelzung

```
/* Vorher: 2 sequentielle Arrays */
int val[SIZE];
int key[SIZE];
```



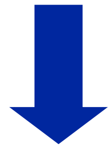
```
/* Nachher: 1 Array mit
   strukturierten Daten */
struct merge {
  int val;
  int key;
};
struct merge merged_array[SIZE];
```

Reduktion von Konflikten zwischen val und key;

Verbesserung der räumlichen Lokalität

Beispiel zur Schleifenumordnung

```
/* Vorher */
  for (j = 0; j < 100; j = j+1)
     for (i = 0; i < 5000; i = i+1)
          x[i][j] = 2 * x[i][j];
```



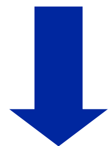
```
/* Nachher */
  for (i = 0; i < 5000; i = i+1)
     for (j = 0; j < 100; j = j+1)
          x[i][j] = 2 * x[i][j];
```

Serielle Zugriffe statt sprungweise Zugriffe auf jedes 100ste Wort

Verbesserung der räumlichen Lokalität

Beispiel zur Schleifenverschmelzung

```
/* Vorher */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
     a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
     d[i][j] = a[i][j] + c[i][j];
```

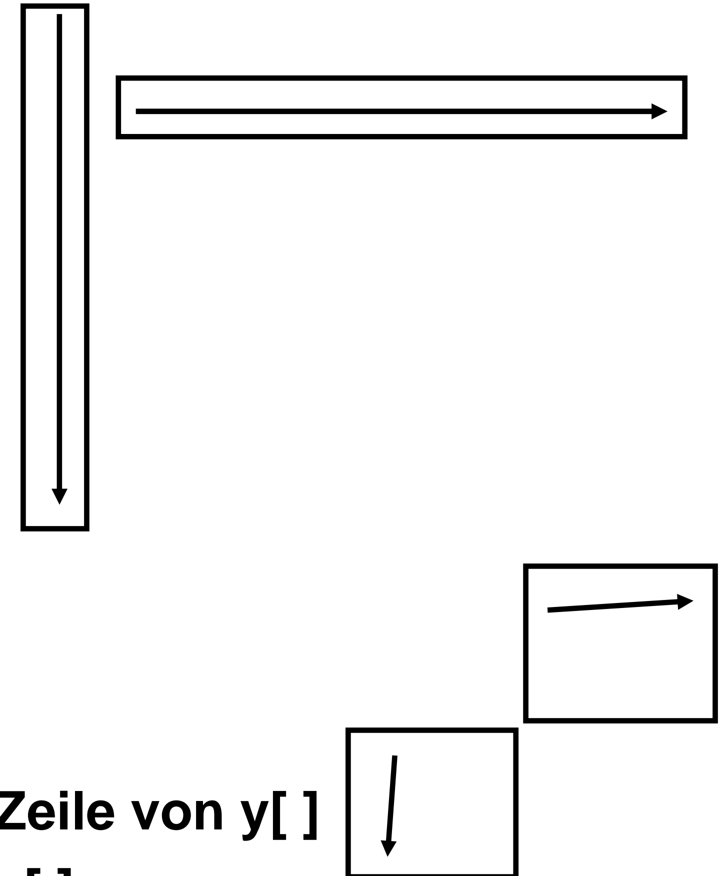


```
/* Nachher */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
  {  a[i][j] = 1/b[i][j] * c[i][j];
     d[i][j] = a[i][j] + c[i][j];}
```

2 Fehlschläge pro Zugriff auf a und c vs. 1 Fehlschlag
Verbesserung der räumlichen Lokalität

Beispiel zur Blockbildung

```
/* Vorher */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
     {r = 0;
      for (k = 0; k < N; k = k+1){
        r = r + y[i][k]*z[k][j];};
      x[i][j] = r;
     };
```



- Innere Schleifen:
  - Lesen aller NxN Elemente von z[ ]
  - Wiederholtes Lesen aller N Elemente einer Zeile von y[ ]
  - Schreiben aller N Elemente einer Zeile von x[ ]
- Idee: Rechne mit BxB Teilmatrix, die in Cache passt.

Beispiel zur Blockbildung (Fortsetzung)

```
/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i+1)
   for (j = jj; j < min(jj+B-1,N); j = j+1)
     {r = 0;
      for (k = kk; k < min(kk+B-1,N); k = k+1) {
        r = r + y[i][k]*z[k][j];};
      x[i][j] = x[i][j] + r;
     };
```

- **B heißt Blockfaktor.**

Zusammenfassung: Reduktion der Fehlschlagsquote

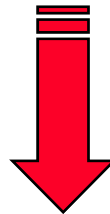
Mittlere Speicherzugriffszeit = Trefferzeit + Fehlschlagsquote * Nachladezeit

3 Cs: Compulsory, Capacity, Conflict

0. Vergrößerung des Caches
1. Vergrößerung der Blockgröße
2. Erhöhung der Assoziativität
3. Victim Cache
4. Pseudo-Assoziativität
5. Vorausladen von Instruktionen und Daten
6. Compiler Optimierungen

Verbesserung der Leistung eines Cache-Speichers

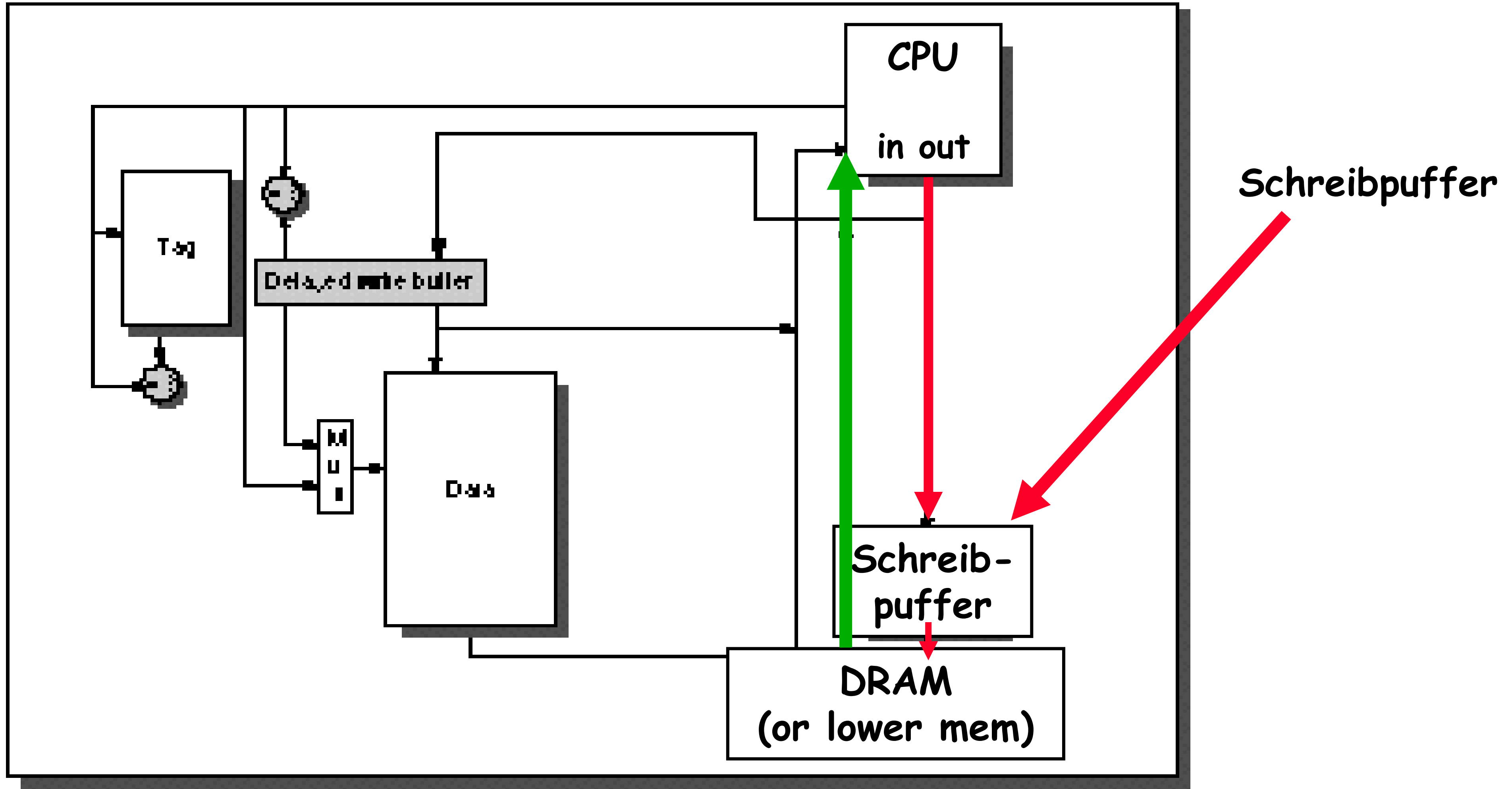
Mittlere Speicherzugriffszeit = Trefferzeit + Fehlschlagsquote * Nachladezeit



1. Reduziere die Fehlschlagquote
2. Reduziere die Nachladezeit oder
3. Reduziere die Trefferzeit.

Lesezugriffe vor Schreibzugriffen

Make the common case fast!



Lesen mit Priorität vor Schreibfehlschlägen

- Write-through/write around mit Schreibpuffer
  - CPU braucht nicht auf Abschluss der Schreiboperation zu warten
  - Aber: RAW Konflikt mit Hauptspeicherlesezugriff bei Cache-Fehlschlag
    - » Nachladezeit kann erhöht werden, wenn auf das Leeren des Schreibpuffers gewartet werden muss
    - » besser: Überprüfe Schreibpufferinhalte vor Lesezugriff auf Hauptspeicher
- Write-back verwendet Puffer zum Zurückschreiben ausgelagerter Blöcke
  - Lesefehlschlag ersetzt modifizierten Block
  - Kopiere modifizierten Block in Schreibpuffer und lese neuen Block, dann schreibe modifizierten Block zurück
  - CPU braucht nicht auf das Zurückschreiben des Blocks zu warten

Teilblockplatzierung

- Unterteilung der Cache-Blöcke in Teilblöcke,
  - die einzeln geladen werden können und
  - jeweils mit einem Gültigkeitsbit versehen sind.

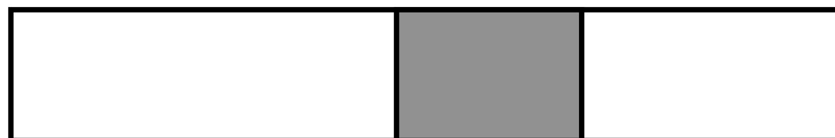
- zweistufige Adressierung
  - Tagüberprüfung
  - Gültigkeitsbitüberprüfung

- Vorteile:
  - Nachladen von Teilblöcken erfordert weniger Zeit
  - kürzere Tags als bei separatem Tag pro Teilblock

Schnelle Reaktivierung der CPU

- **Warte nicht auf das Laden des vollständigen Blocks, bevor CPU weiterarbeiten kann**
  - ***Early restart*— sobald gesuchtes Wort geladen wurde, wird dieses an die CPU weitergeleitet und diese kann weiterarbeiten**
  - ***Critical Word First*—lade zuerst das gesuchte Wort und leite dieses unmittelbar an die CPU weiter, lade anschließend unabhängig den Rest des Blocks.**

- **vor allem bei großen Blöcken**
- **räumliche Lokalität => Oft erfolgt Zugriff auf das nächste Wort, so dass der Vorteil des early restart verloren gehen kann**



block

Nicht-blockierende Caches

- ***Non-blocking cache*** oder ***lockup-free cache*** erlaubt die Verarbeitung weiterer Anfragen während des Nachladens von Blöcken

- "***hit under miss***" reduziert die effektive Nachladezeit durch Verarbeitung weiterer Anfragen
- "***hit under multiple miss***" or "***miss under miss***" => Überlagerung mehrerer Nachladevorgänge
  - komplexer Cache Controller
  - mehrere Speicherbänke erforderlich
  - Pentium Pro verarbeitet bis zu 4 Cache-Fehlschläge gleichzeitig

Second-level Caches

- Optimierung der Schnittstelle Cache-Hauptspeicher durch weiteren Pufferspeicher
- Vorteil:
  - erster Cache schnell und klein mit guter Trefferzeit
  - zweiter Cache größer mit geringer Fehlschlagsquote
- L2 Gleichungen (t_M = Average Memory Access Time)

$$t_M = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}$$

$$\text{Miss Penalty}_{L1} = \text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2}$$

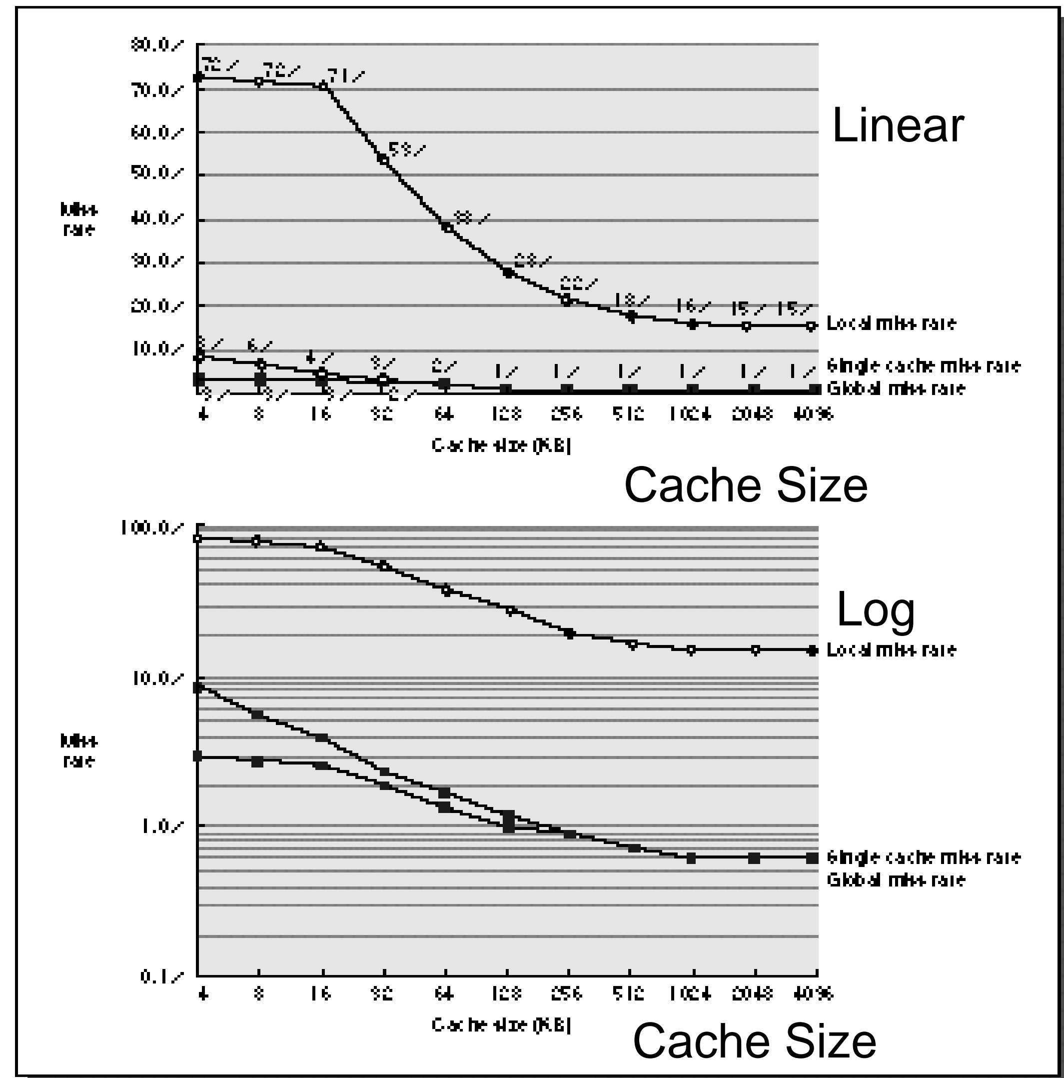
$$\Rightarrow t_M = \text{Hit Time}_{L1} + \underline{\text{Miss Rate}_{L1}} \times (\text{Hit Time}_{L2} + \underline{\text{Miss Rate}_{L2}} + \text{Miss Penalty}_{L2})$$

- Definitionen:
  - *Lokale Fehlschlagsquote*
    = # Fehlschläge des Caches / # Zugriffe auf diesen Cache (Miss rate_{L2})
  - *Globale Fehlschlagsquote*
    = # Fehlschläge des Caches / # Speicherzugriffe der CPU
  - Was zählt, ist die globale Fehlschlagsquote (= Prozentsatz der Speicherzugriffe, die zum Hauptspeicher gehen).

Vergleich lokaler und globaler Fehlschlagsquoten

- **Beispiel:**
  - **32 KByte 1st level cache**
  - **variabler 2nd level cache**
- **Globale Fehlschlagsquote entspricht Fehlschlagsquote einzelnen Caches, falls L2 >> L1**
- **Lokale Fehlschlagsquote wenig aussagekräftig**
- **L2 unabhängig von CPU Taktzeit**

- **höhere Assoziativität sinnvoll, um Fehlschlagsquote zu senken**
- **größere Blöcke**
- **schnelle Treffer und wenige Fehlschläge**



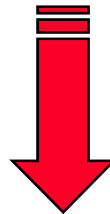
Zusammenfassung: Reduktion der Nachladezeit

5 Techniken

- **Lesepriorität vor Zurückschreiben bei Fehlschlägen**
- **Teilblockplatzierung**
- **Schnelle CPU-Reaktivierung:
  Early Restart und Critical Word First on miss**
- **Nicht-blockierende Caches (Hit under Miss, Miss under Miss)**
- **Second Level Caches**

Verbesserung der Leistung eines Cache-Speichers

Mittlere Speicherzugriffszeit = Trefferzeit + Fehlschlagsquote * Nachladezeit



1. **Reduziere die Fehlschlagquote**
2. **Reduziere die Nachladezeit oder**
3. **Reduziere die Trefferzeit.**

Reduktion der Trefferzeit

- schnelle und einfache first-level Caches
  - schnell -> klein, on-chip -> schneller Tagvergleich
  - einfach -> direkte Abbildung
    -> Überlappung von Datenübertragung und Tagvergleich

- virtuelle Adressierung in Cache
  - keine Adressumsetzung vor Cache-Zugriff

- Fließbandverarbeitung von Schreibzugriffen
  - Überlappung von Tagvergleich nachfolgenden Schreibbefehls mit Schreiben der Daten von vorherigem Schreibbefehl

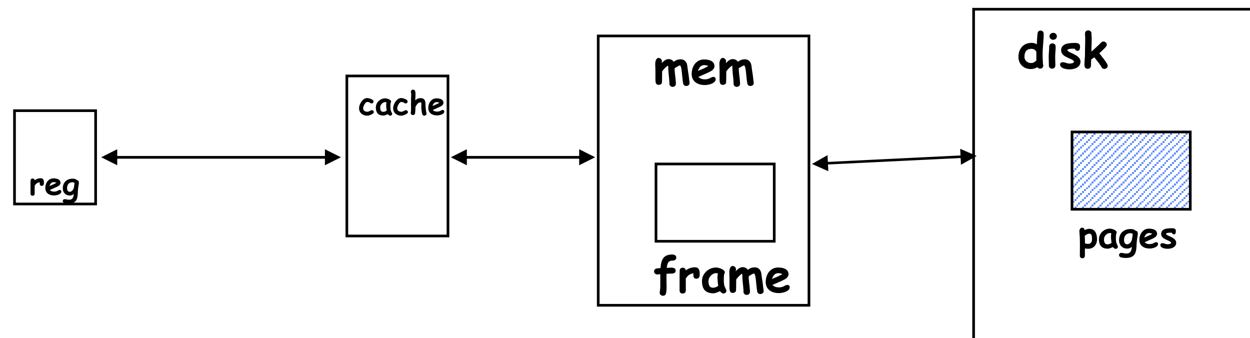
Zusammenfassung: Cache-Optimierungstechniken

	Technik	*MR*	*MP*	*HT*	*Complexity*
Fehlschlagsquote	Größere Blöcke	+	−		0
	Höhere Assoziativität	+		−	1
	Victim Caches	+			2
	Pseudo-Assoziativität	+			2
	HW Prefetching of Instr/Data	+			2
	Compilertechniken	+			0
Nachladezeit	Priority to Read Misses		+		1
	Teilblockplatzierung		+		1
	Early Restart & Critical Word 1st		+		2
	Nicht-blockierende Caches		+		3
	Second Level Caches		+		2
Trefferzeit	Kleine, einfache Caches	−		+	0
	virtuelle Adressierung			+	2
	Pipelining von Schreibtreffern			+	1

Entwurf eines virtuellen Speichersystems

Charakteristika:

- Größe der Blöcke (Seiten), die zwischen Hintergrundspeicher und Hauptspeicher ausgetauscht werden (typisch 4 KB)
- Ersetzungsstrategie (Reduktion der Fehlschlagquote wichtig-> LRU Strategie)
- Schreibstrategie (write back, da write through zu teuer)
- Blockplatzierung und -identifikation
- bedarfsgesteuertes Nachladen von Blöcken (-> durch Betriebssystem)



<u>Seitenverwaltung (Paging Organization)</u>

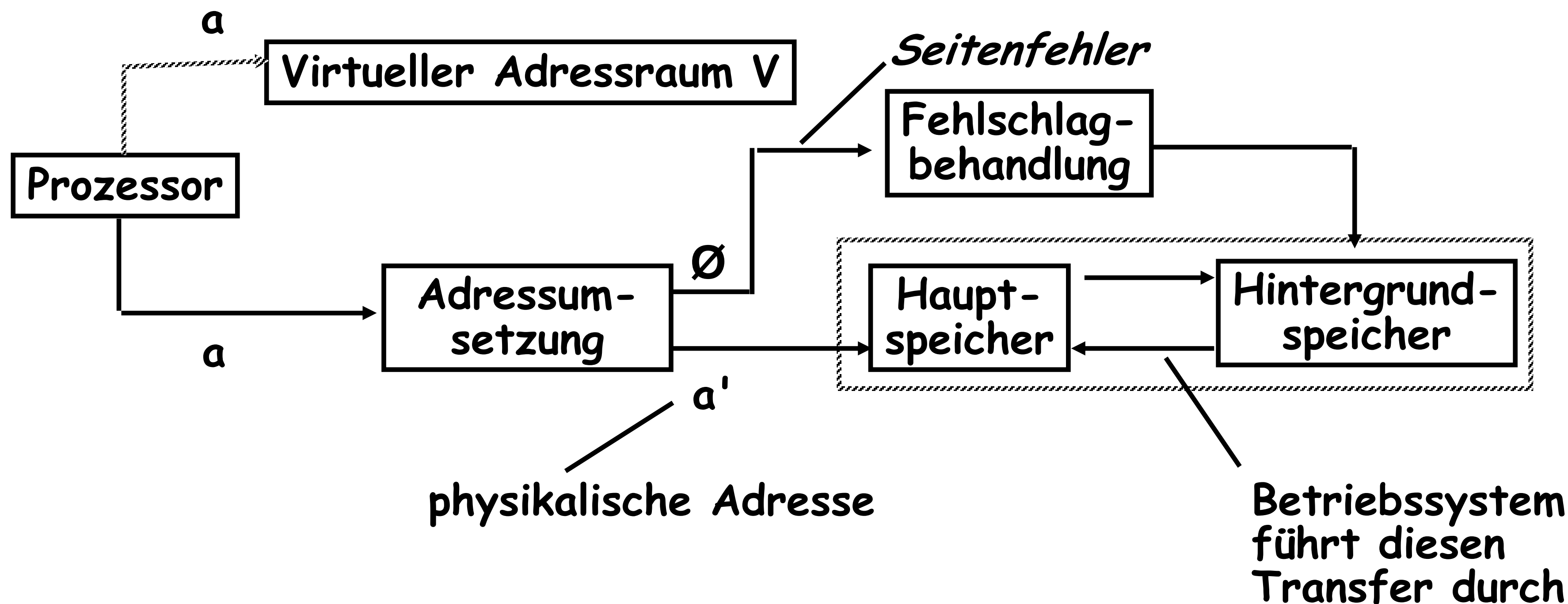
virtueller and physikalischer Adressbereich wird in Blöcke gleicher Größe partitioniert

Adressabbildung

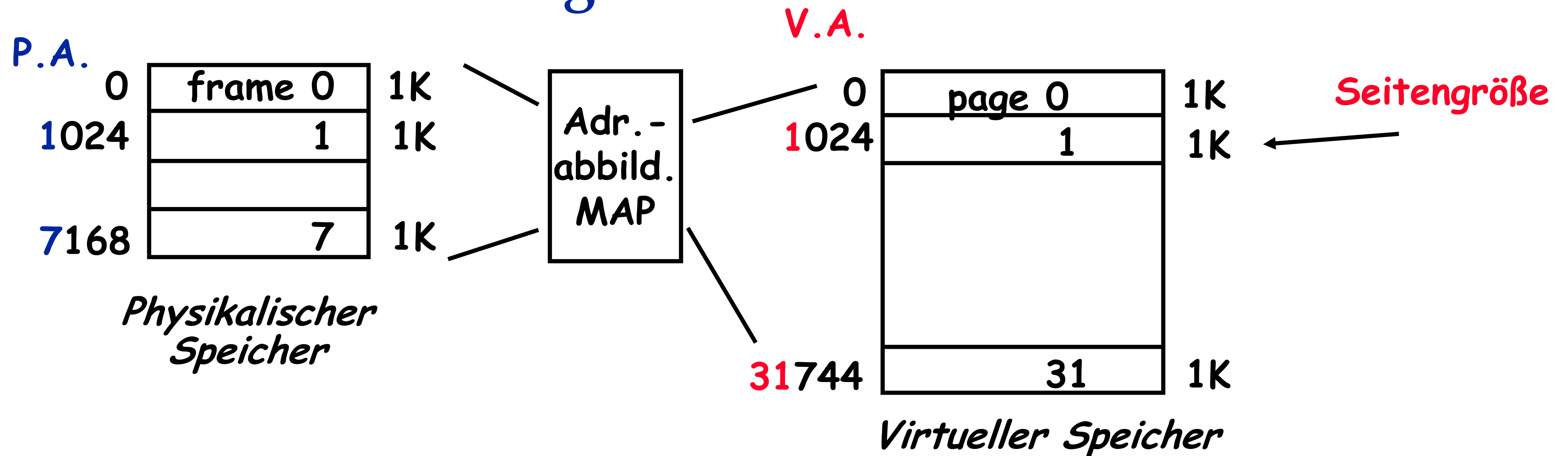
$V = \{0, 1, \ldots, n - 1\}$ virtueller Adressbereich
$M = \{0, 1, \ldots, m - 1\}$ physikalischer Adressbereich
MAP: $V \dashrightarrow M \cup \{\emptyset\}$ Adressabbildungsfunktion

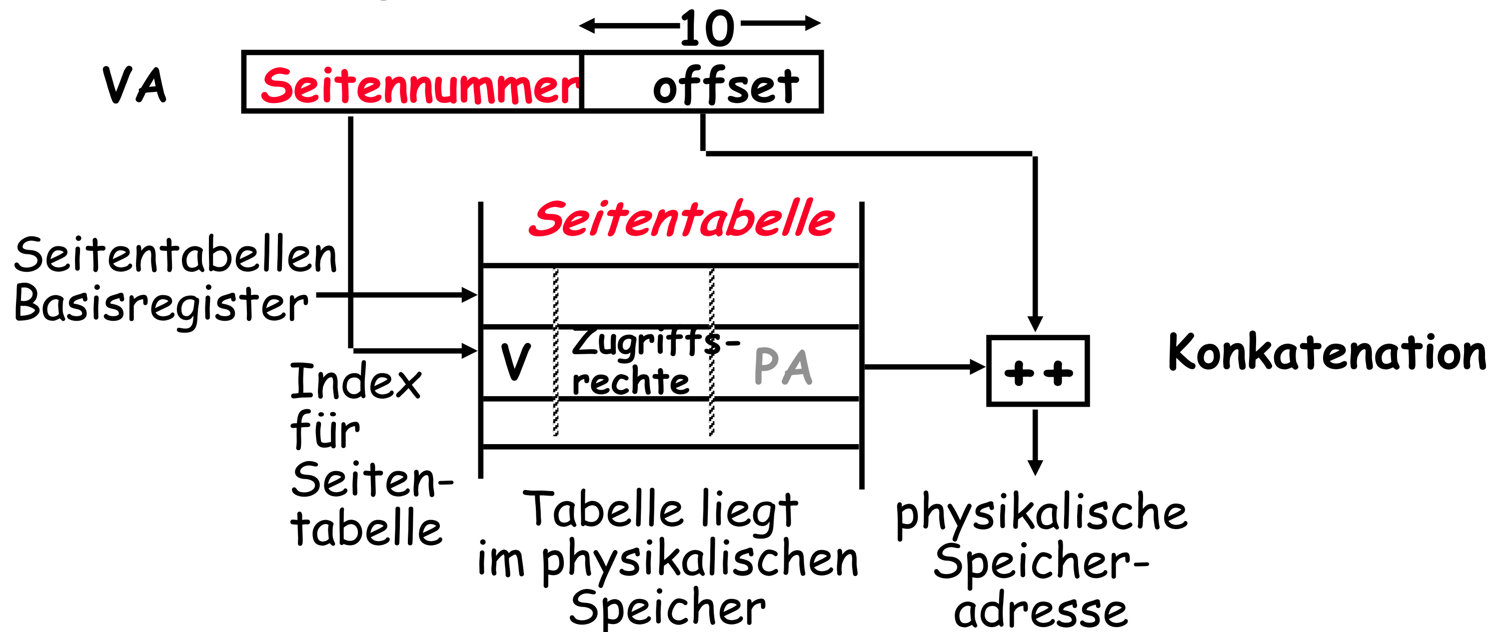
$n >> m$

MAP(a) =
- a' falls Daten mit virtueller Adresse <u>a</u> an physikalischer Adresse <u>a'</u> liegen und <u>a'</u> in M
- Ø falls Daten mit virtueller Adresse a nicht in M liegen

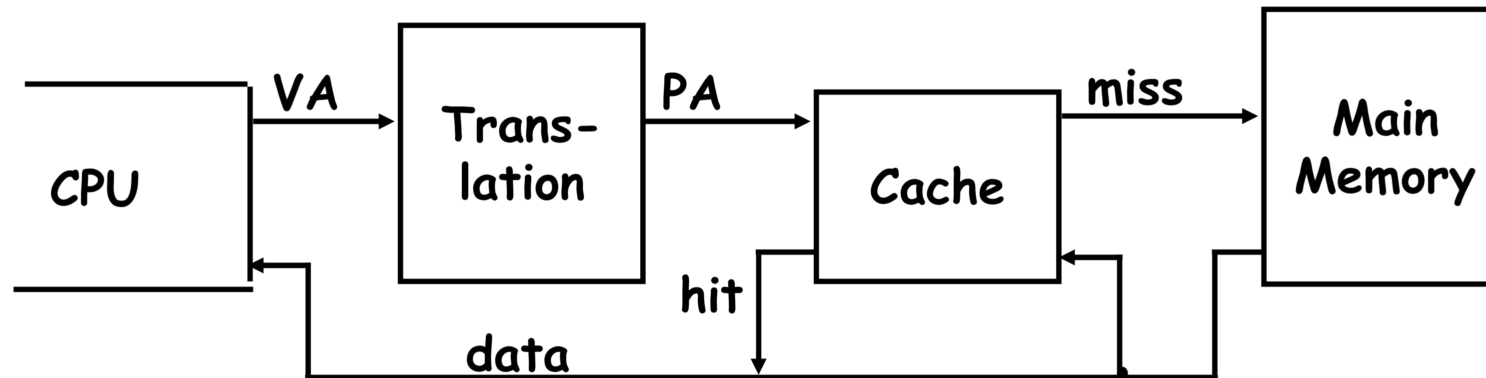


Seitenverwaltung





Realisierung der Adressumsetzung



- Die Seitentabelle ist eine große im Speicher liegende Datenstruktur.
- Zwei Speicherzugriffe für jedes Laden, Speichern, Instruktionsholen!!!

- Verwendung von virtuellen Adressen im Cache?
  - Synonymproblem

- Puffern der Adressumsetzungen?

Translation Lookaside Buffers (TLBs)

Spezieller Pufferspeicher für bereits vorgenommene Adressübersetzungen

Virtuelle Adresse	Phys. Adresse	Dirty	Ref	Valid	Access

Cache für Seitentabellen-Adressabbildungen

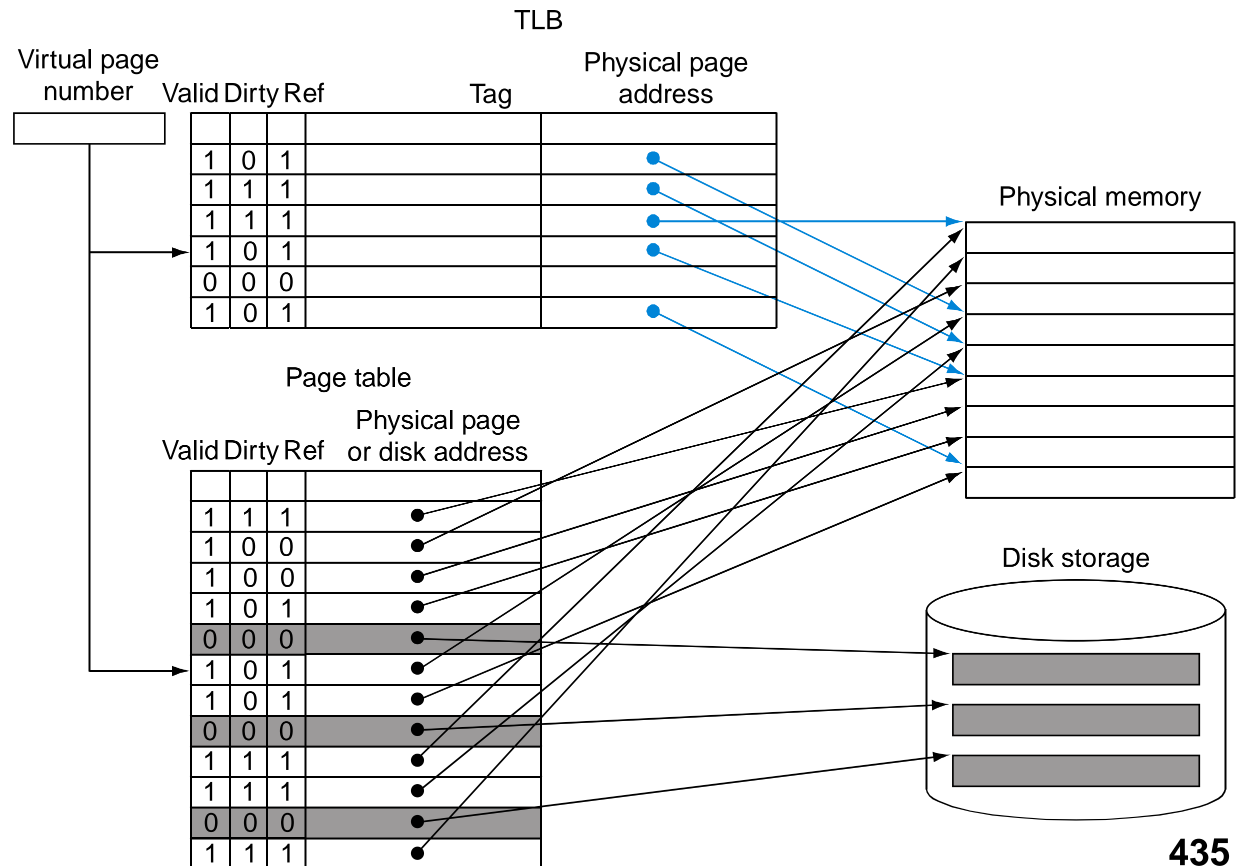
TLB Zugriffszeit ist vergleichbar zur Cachezugriffszeit (viel geringer als die Hauptspeicherzugriffszeit)

Typische Werte:

16-512 Einträge,

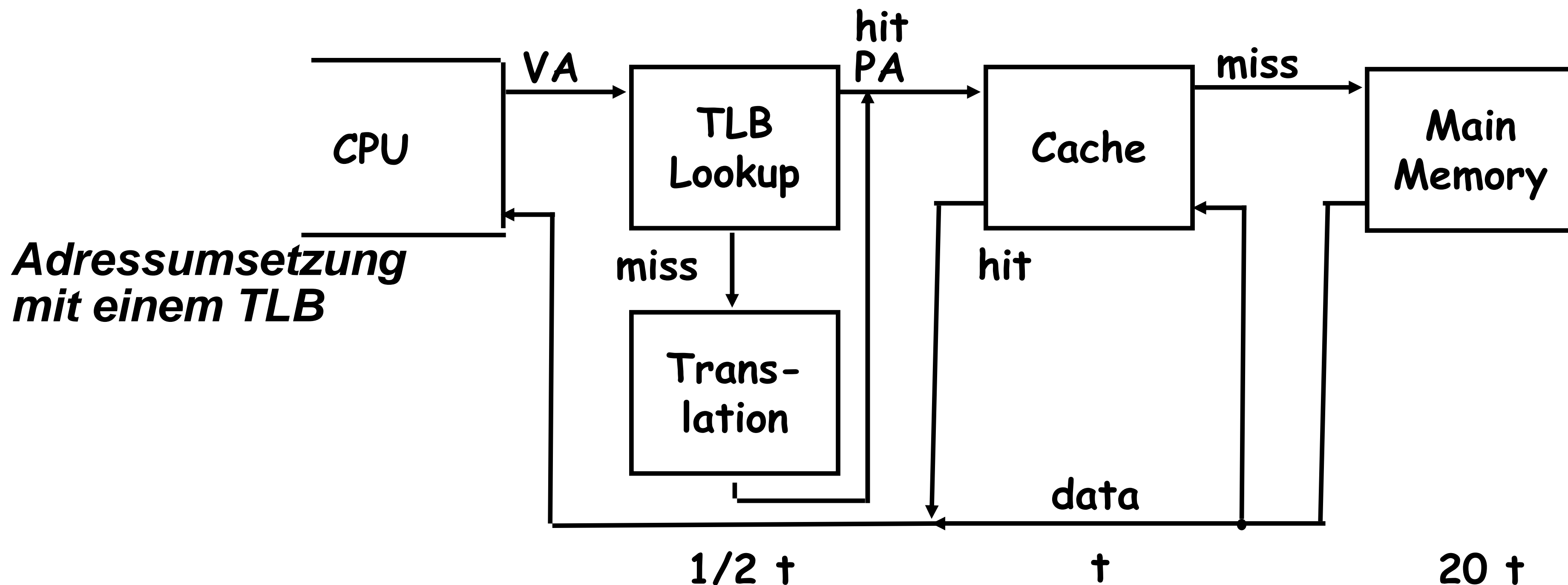
miss-rate: .01% - 1%

miss-penalty: 10 – 100 cycles



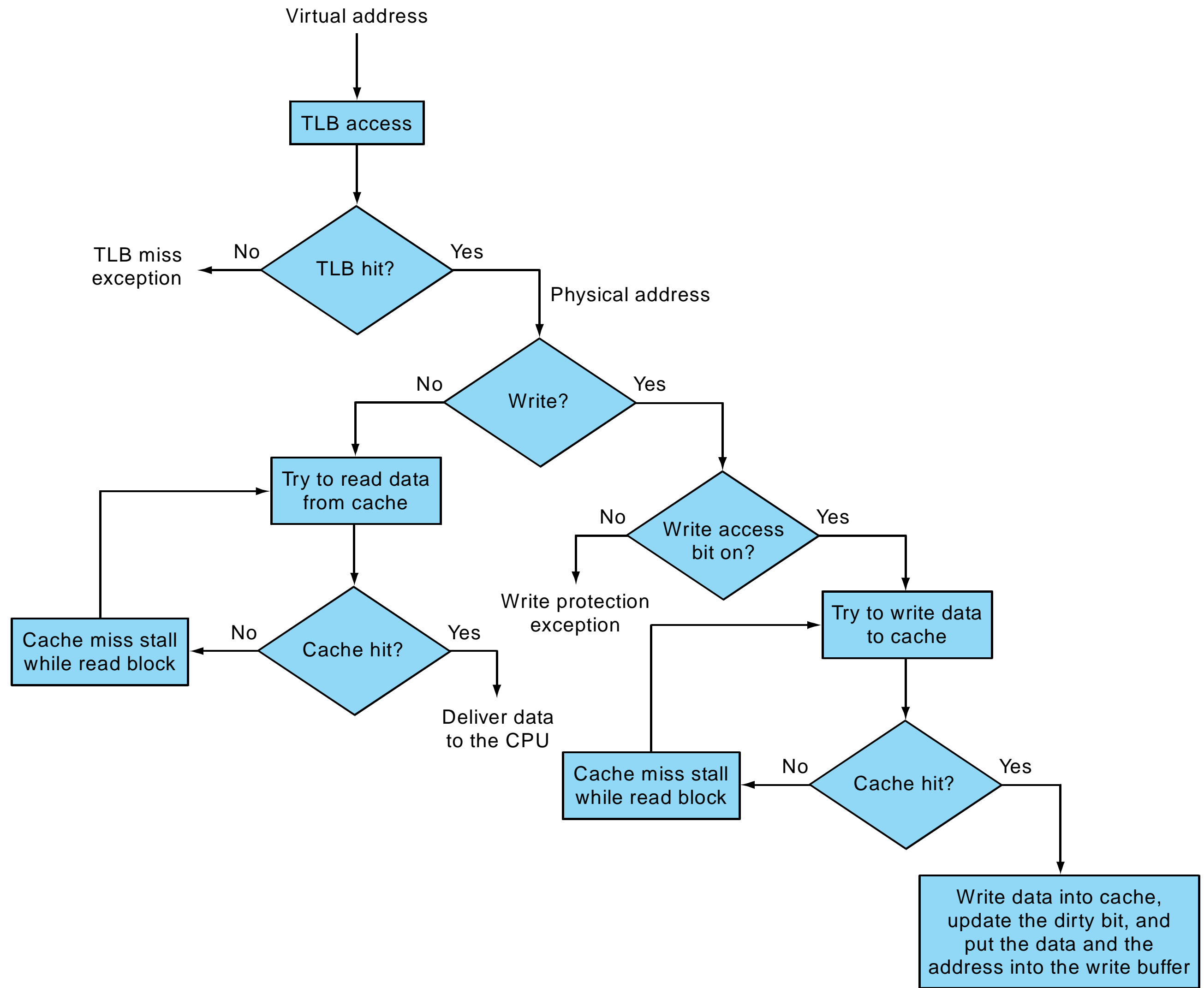
Translation Look-Aside Buffers II

- **Wie andere Caches, können TLBs voll assoziativ, mengenassoziativ oder direkt abgebildet organisiert sein.**
- **TLBs sind normalerweise klein, mit nicht mehr als 128 - 256 Einträgen, sogar auf Maschinen am oberen Ende des Leistungsspektrums**
- **Dies erlaubt voll assoziativen Zugriff auf solchen Maschinen. Mittelklassige Maschinen verwenden kleine n-fach mengenassoziative Organisationen.**

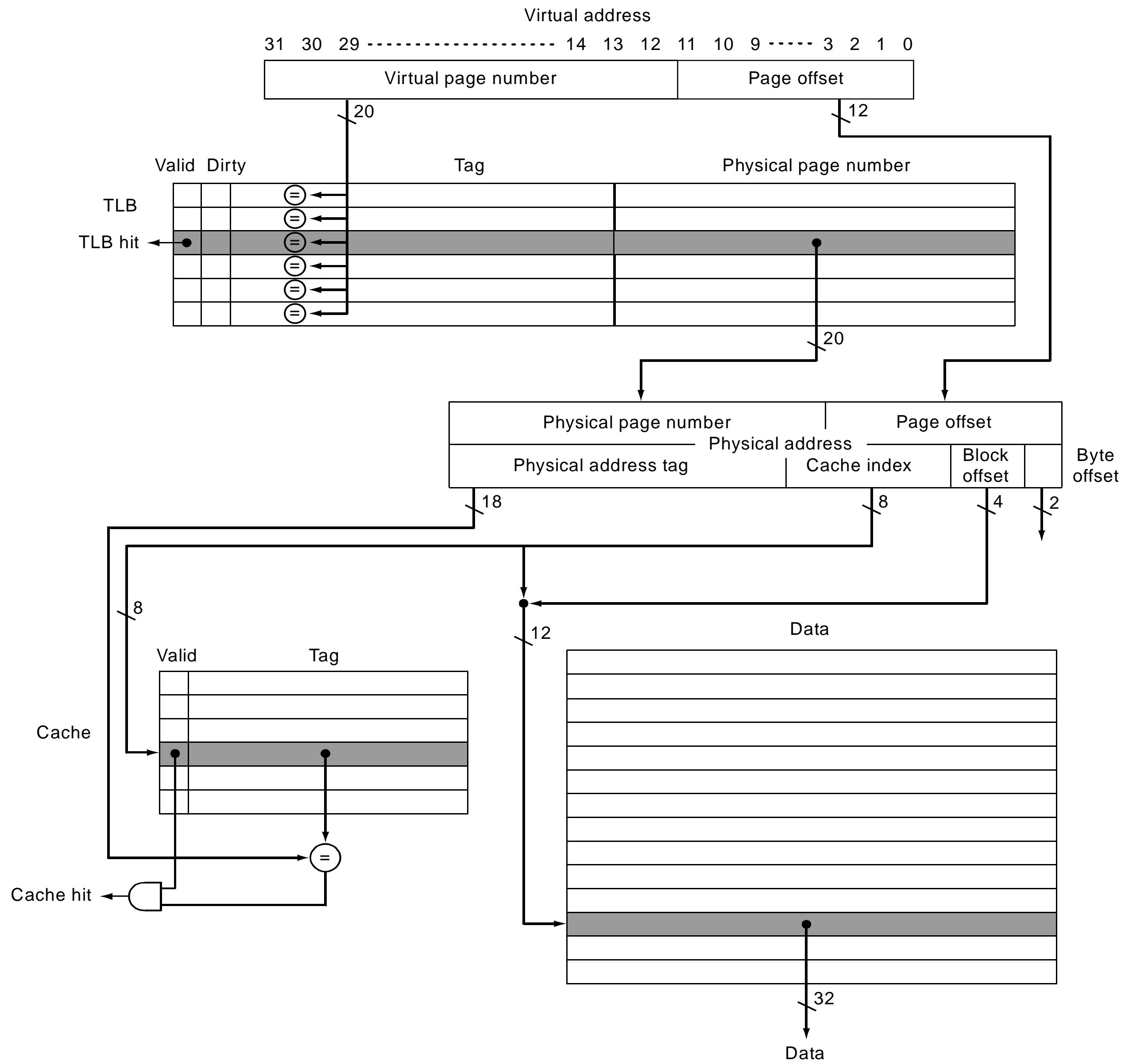


Adressumsetzung mit einem TLB

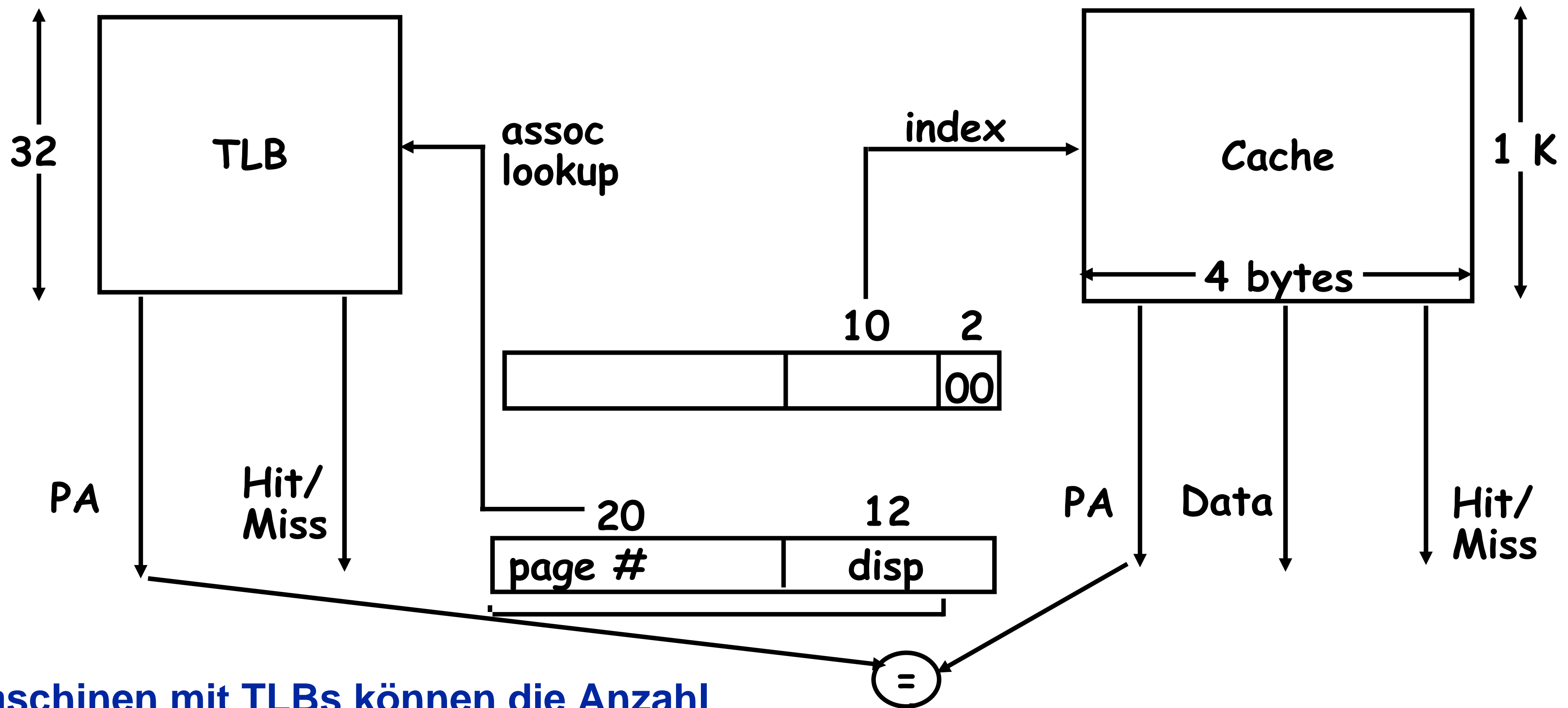
TLBs und Caches



TLBs und Caches



Überlappende Cache- und TLB-Zugriffe



Maschinen mit TLBs können die Anzahl der Takte pro Cachezugriff durch eine Überlappung der Cache- und TLB-Zugriffe reduzieren.

Die oberen Bits der virtuellen Adresse werden zum Zugriff auf den TLB verwendet, die unteren Bits als Cache-Index.

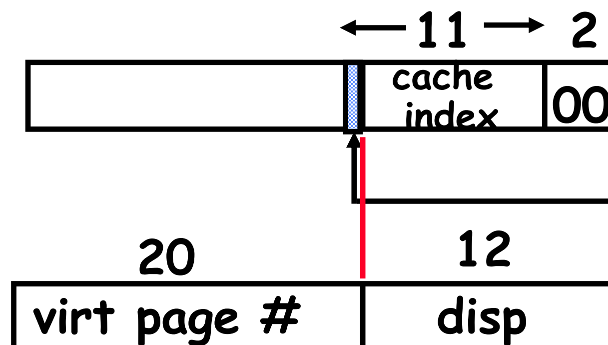
IF	cache hit AND (cache tag = PA)
THEN	deliver data to CPU
ELSE	
IF	[cache miss OR (cache tag /=PA)] and TLB hit
THEN	access memory with the PA from the TLB
ELSE	do standard VA translation

Probleme mit überlappendem TLB Zugriff

Der überlappende Zugriff funktioniert nur, solange die Adressbits, die als Index in den Cache verwendet werden, sich durch die Adressumsetzung nicht ändern.

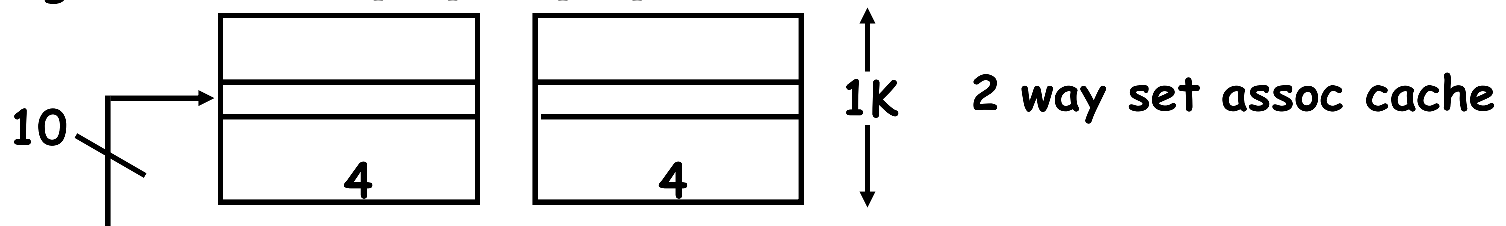
Dies bedeutet eine Einschränkung auf kleine Caches, große Seitengrößen oder n-fache Mengenassoziativität bei größeren Caches.

Beispiel: wie zuvor mit doppelt so großem Cache, also 8 K bytes statt 4 K:



Dieses Bit ändert sich bei der VA Übersetzung, aber es wird für den Cachezugriff verwendet.

Mögliche Lösungen:
- wähle 8K byte Seitengröße;
- wähle 2-fach mengenassoziativen Cache oder
- Software garantiert VA[13]=PA[13]



Moderne Systeme

Characteristic	Intel Pentium P4	AMD Opteron
Virtual address	32 bits	48 bits
Physical address	36 bits	40 bits
Page size	4 KB, 2/4 MB	4 KB, 2/4 MB
TLB organization	1 TLB for instructions and 1 TLB for data Both are four-way set associative Both use pseudo-LRU replacement Both have 128 entries TLB misses handled in hardware	2 TLBs for instructions and 2 TLBs for data Both L1 TLBs fully associative, LRU replacement Both L2 TLBs are four-way set associativity, round-robin LRU Both L1 TLBs have 40 entries Both L2 TLBs have 512 entries TLB misses handled in hardware

FIGURE 7.34 Address translation and TLB hardware for the Intel Pentium P4 and AMD Opteron. The word size sets the maximum size of the virtual address, but a processor need not use all bits. The physical address size is independent of word size. The P4 has one TLB for instructions and a separate identical TLB for data, while the Opteron has both an L1 TLB and an L2 TLB for instructions and identical L1 and L2 TLBs for data. Both processors provide support for large pages, which are used for things like the operating system or mapping a frame buffer. The large-page scheme avoids using a large number of entries to map a single object that is always present.

Characteristic	Intel Pentium P4	AMD Opteron
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	8 KB for data, 96 KB trace cache for RISC instructions (12K RISC operations)	64 KB each for instructions/data
L1 cache associativity	4-way set associative	2-way set associative
L1 replacement	Approximated LRU replacement	LRU replacement
L1 block size	64 bytes	64 bytes
L1 write policy	Write-through	Write-back
L2 cache organization	Unified (instruction and data)	Unified (instruction and data)
L2 cache size	512 KB	1024 KB (1 MB)
L2 cache associativity	8-way set associative	16-way set associative
L2 replacement	Approximated LRU replacement	Approximated LRU replacement
L2 block size	128 bytes	64 bytes
L2 write policy	Write-back	Write-back

FIGURE 7.35 First-level and second-level caches in the Intel Pentium P4 and AMD Opteron. The primary caches in the P4 are physically indexed and tagged; for a discussion of the alternatives, see the Elaboration on page 527.

Zusammenfassung: Cache-Grundprinzipien

- **Lokalitätsprinzip**
  - Programme greifen zu jedem Zeitpunkt auf einen relativ kleinen Bereich ihres Adressraumes zu.
    - » Zeitliche Lokalität
    - » Räumliche Lokalität

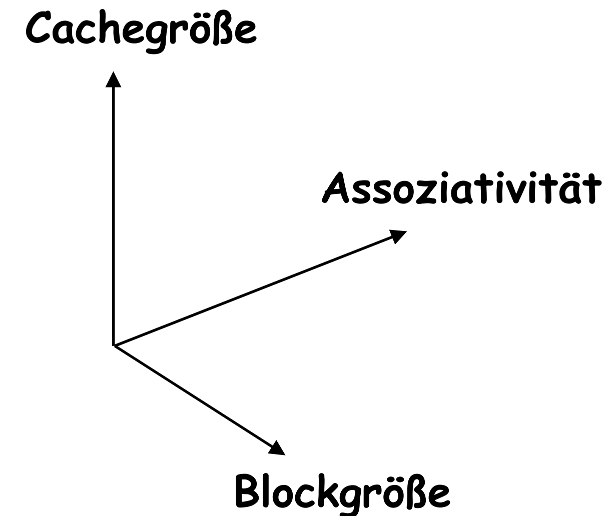
- **Drei Kategorien von Fehlschlägen**
  - Compulsory Misses: unvermeidbar, z.B. Kaltstartfehlschläge
  - Capacity Misses: begrenzte Cachegröße
  - Conflict Misses: begrenzte Cachegröße und/oder Assoziativität

- **Schreibstrategien:**
  - Write Through: benötigt Schreibpuffer
  - Write Back: komplexe Kontrolle

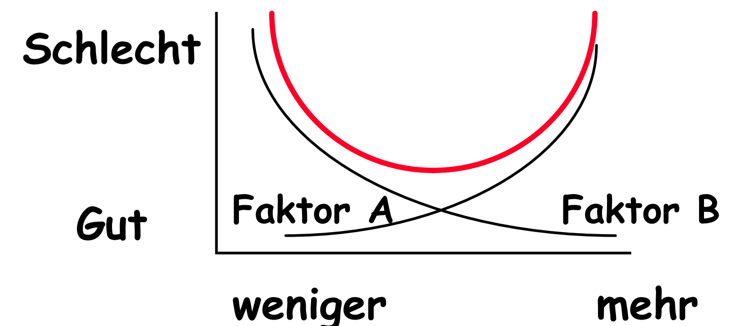
- **CPU Zeit kann durch Cache-Fehlschläge stark beeinflusst werden. Was bedeutet dies für Compiler, Datenstrukturen, Algorithmen?**

Zusammenfassung: Cache-Entwurf

- Interagierende Dimensionen
  - Cachegröße
  - Blockgröße
  - Assoziativität
  - Ersetzungsstrategie
  - write-through vs write-back



- Optimale Wahl ist Kompromiss
  - abhängig von Zugriffscharakteristika
    - » Arbeitslast
    - » Gebrauch (I-Cache, D-Cache, TLB)
  - abhängig von Technologie / Kosten



- Einfachheit ist meist am besten.

Zusammenfassung: TLB, Virtueller Speicher

- Caches, TLBs, Virtuelle Speicher funktionieren nach demselben Prinzip.
- Kernfragen:
  - 1) Wo kann ein Block platziert werden?
  - 2) Wie wird ein Block identifiziert?
  - 3) Welcher Block wird bei einem Fehlschlag ausgelagert?
  - 4) Wie ist das Schreiben organisiert?

- Seitentabellen bilden virtuelle Adressen auf physikalische Adressen ab.

- TLBs machen virtuelle Speicher praktikabler.
  - Datenlokalität => Lokalität bei Adressdaten (zeitlich und räumlich)
- TLB Fehlschläge sind signifikant für die Prozessorleistung

- Virtueller Speicher ermöglicht, dass mehrere Prozesse einen einzelnen physikalischen Speicher gemeinsam nutzen können, ohne dass alle Prozesse auf die Platte ausgelagert werden müssen.
  Schutzmechanismen sind dabei wichtiger als die Speicherhierarchie.