Rechnerarchitektur I (RAI)

Speicherarchitektur - Cache

Prof. Dr. Akash Kumar

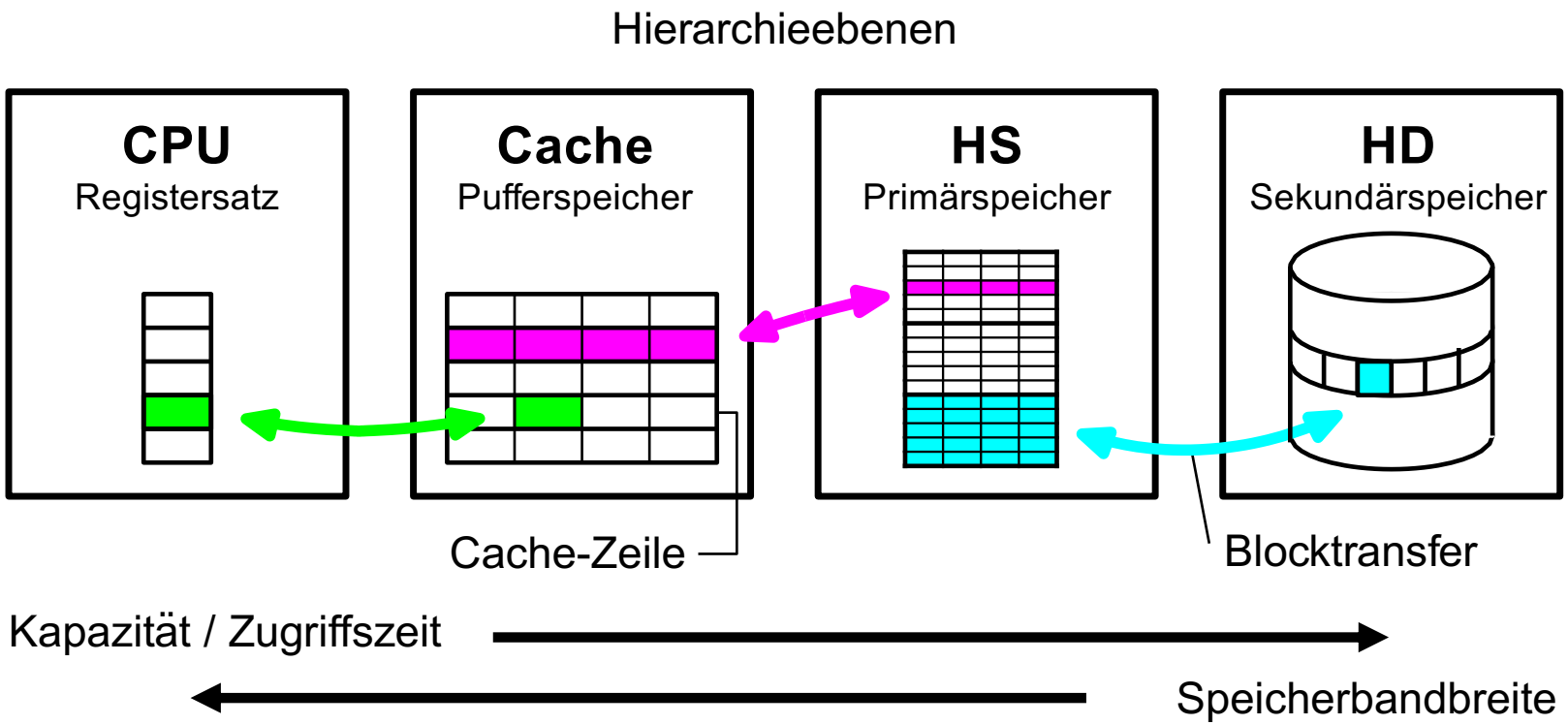
Chair for Processor Design

cfaed.tu-dresden.de

Inhalt

- Cache-Konzept
- Associativity
- Größe Berechnungen
- Read – Write – Replace Strategie

Speicherhierarchie - Blockprinzip



Unterschiedliche Blockgrößen zwischen den Hierarchieebenen sind möglich.

Cache-Speicher

- CAM-Speicher (Content Addressable Memory) als Pufferspeicher zur Überbrückung bzw. Anpassung stark unterschiedlicher Zugriffszeiten (z.B. Prozessorregister – Hauptspeicher – Festplatte).
- Der Cache arbeitet inhaltsorientiert, ist nicht direkt adressierbar. Der Vergleich mit dem Inhalt kann auch maskiert erfolgen (Ausblenden einzelner Bits). Ein Cache-Zugriff ist nicht immer eindeutig.
- Im Cache werden nur Kopien der aktuellen Speicherinhalte der darunter liegenden Hierarchieebene abgebildet. Die aktuellen Daten einer Hierarchieebene befinden sich auch immer in allen darunter liegenden.
- Zur Überbrückung sehr großer Differenzen in der Zugriffszeit bzw. im Datendurchsatz können auch mehrere Caches hintereinander geschaltet werden (Primary Cache, Secondary Cache, übliche Bezeichnung L1, L2, ... L*n*).
- Der Cache-Speicher kann sich mit auf dem CPU-Chip befinden (On-Chip Cache, oft nur L1) oder extern (Off-Chip Cache).
- In Multiprozessorsystemen teilen sich u.U. mehrere CPUs einen Cache (z.B. L3, L4)

Cache-Begriffe

Adressspeicher (Tag-RAM):

Hauptspeicheradresse bzw. Adressteil des Datenblockes (Cache-Line).

Valid-Bit: Cache-Inhalt bzgl. der Adresse ist aktuell, gültig.

Dirty-Bit: Cache- und Hauptspeicherinhalt der Adresse sind nicht konsistent.

Datenspeicher (Data-RAM):

Datenblöcke (Cache-Line): Blöcke des Cache-Speichers ⟹ Blockprinzip.

Datenblockgröße (Cache-Line-Size): Blockgröße in Worten (Byte).

Cache-Treffer (Cache-Hit):

Daten / Befehle aktuell im Cache gefunden.

Trefferrate (Hit-Rate): Maß für die Effizienz des Caches.

Cache-Fehlzugriff (Cache-Miss):

Daten / Befehle nicht im Cache gefunden bzw. nicht aktuell.

Fehlerrate (Miss-Rate): 1 - Trefferrate.

Fehlerzuschlag (Miss-Penalty): Zugriffszeit nach Cache-Fehlzugriff.

Cache-Eigenschaften

Struktur:

Ein Cache-Eintrag besteht aus dem Datenblock (Cache-Line $\Longrightarrow$ Blockprinzip) und dessen Adresse, Adressteil als Tag (Etikett) sowie Statusinformationen.

Sichtbarkeit:

Transparent (nicht sichtbar im Befehlssatz) oder nicht transparent (sichtbar, wird durch den Befehlssatz gesteuert, z.B. Laden, Invalidieren, . . .).

Adressraum:

Realer Adressraum (realer Cache) oder virtueller (virtueller Cache).

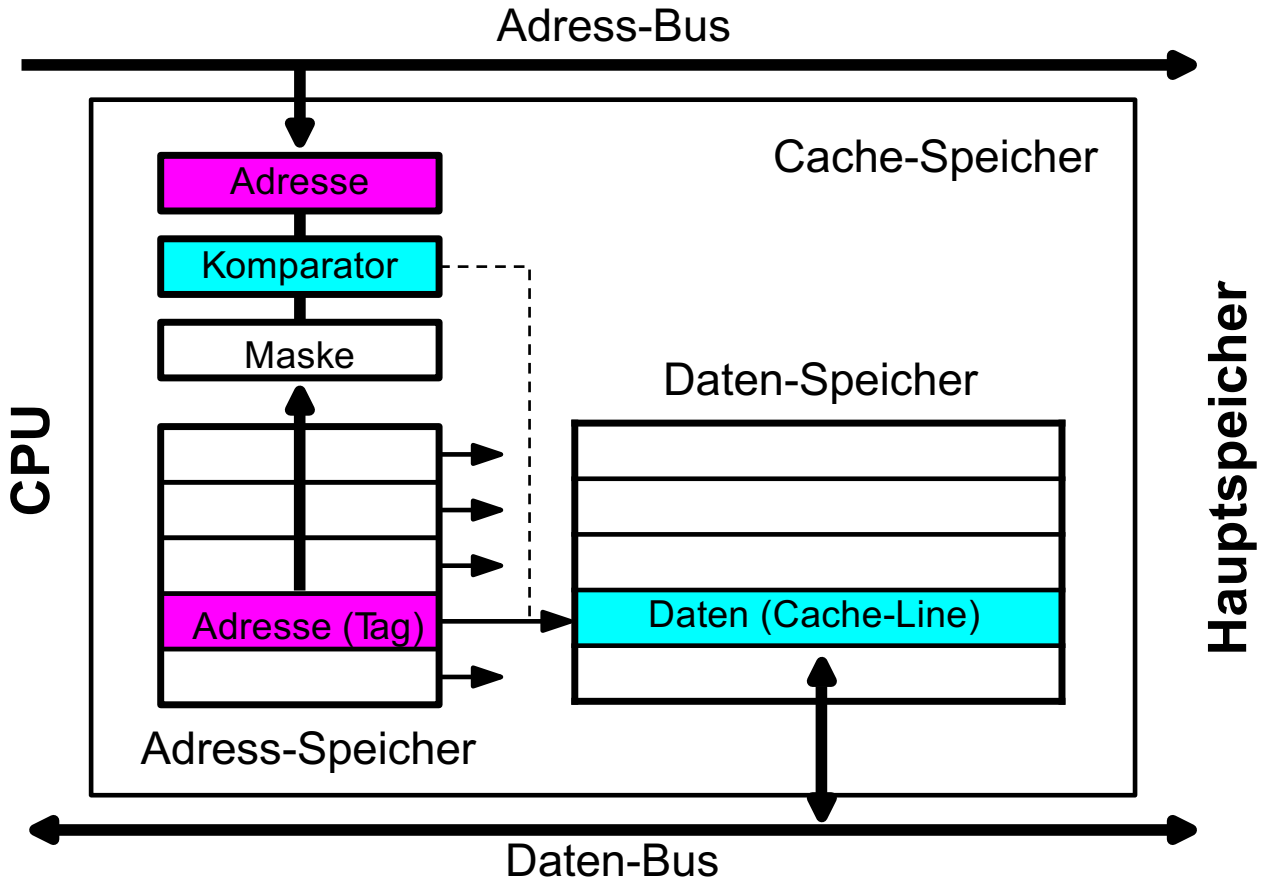
Architektur:

Befehle und Daten gemeinsam (Unified Cache) oder getrennt (Split Cache).

Organisation:

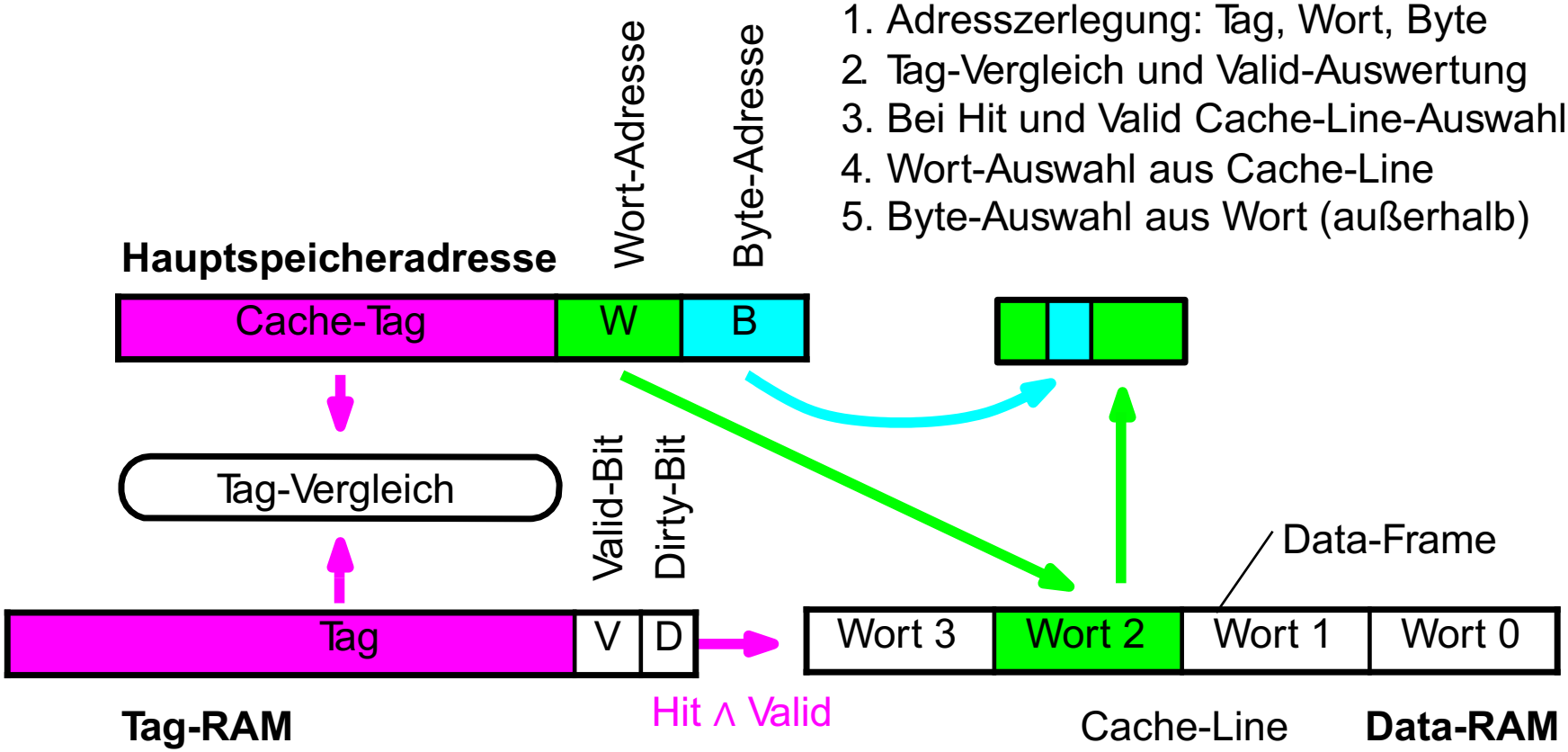
Lösung der Probleme der Speicherhierarchie: Abbildung, Identifikation, Ersetzung, Aktualisierung, Konsistenz, Kohärenz.

Struktur des Cache-Speichers (Übersicht)

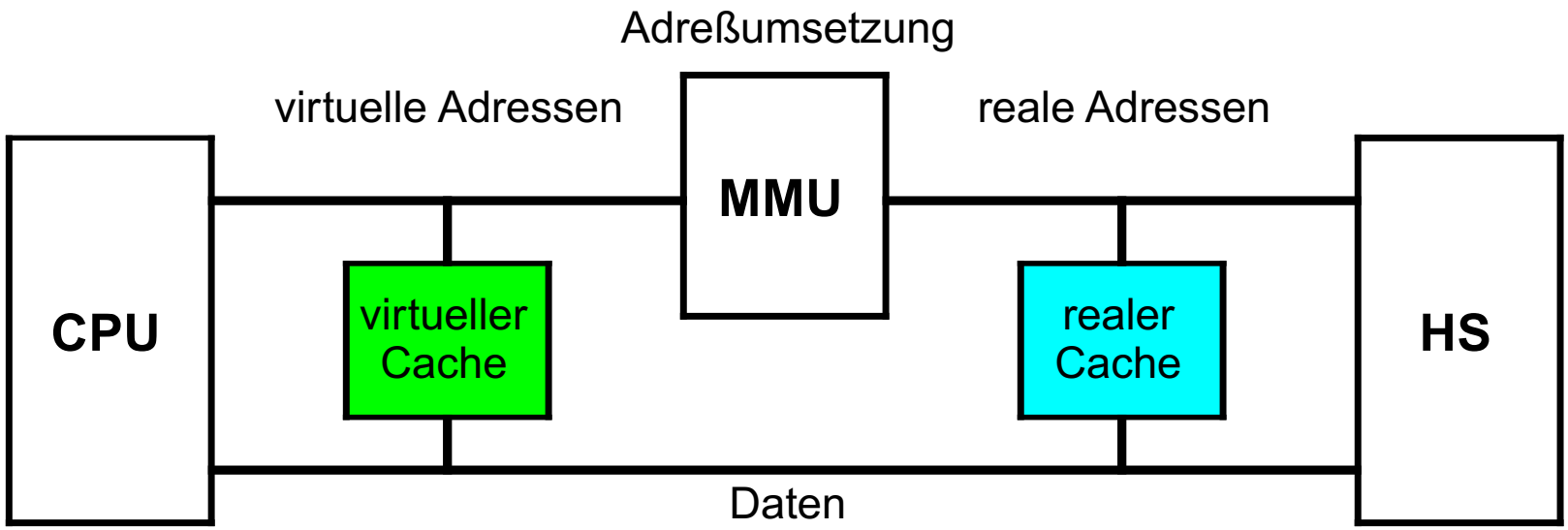


Cache-Struktur

1. Adresszerlegung: Tag, Wort, Byte
2. Tag-Vergleich und Valid-Auswertung
3. Bei Hit und Valid Cache-Line-Auswahl
4. Wort-Auswahl aus Cache-Line
5. Byte-Auswahl aus Wort (außerhalb)

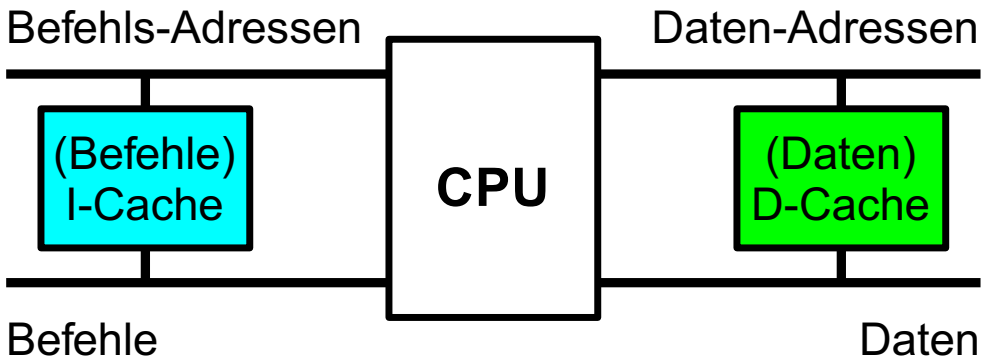


Cache-Adreßraum



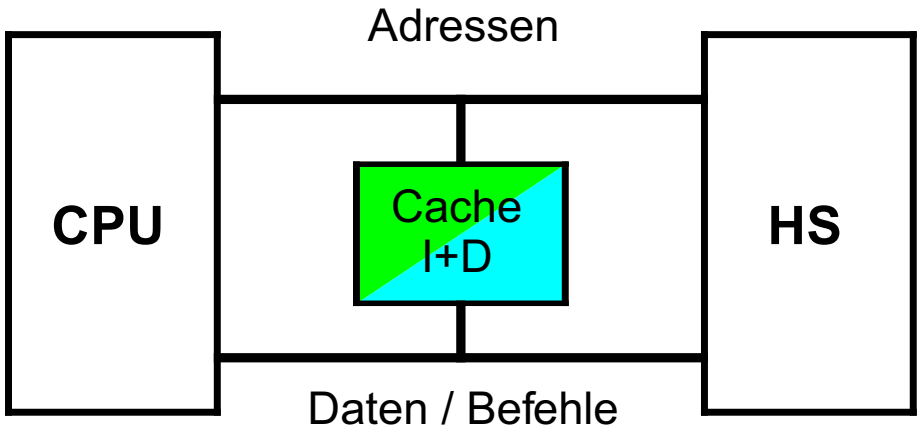
Virtueller Cache	Realer Cache
- schnell, Cache und MMU nebenläufig	- langsam, Adressen durch MMU
- Invalidierung bei Prozeßwechsel, notwendig, da gleiche Adressen möglich	- direkter I/O-Cache-Transfer möglich, da reale Adressen
- besondere Eignung als Befehls-Cache, da Rückschreiben entfällt	- für Snooping geeignet, da 1:1 Adreß-Daten-Abbildung

Cache-Architektur



Harvard - Architektur

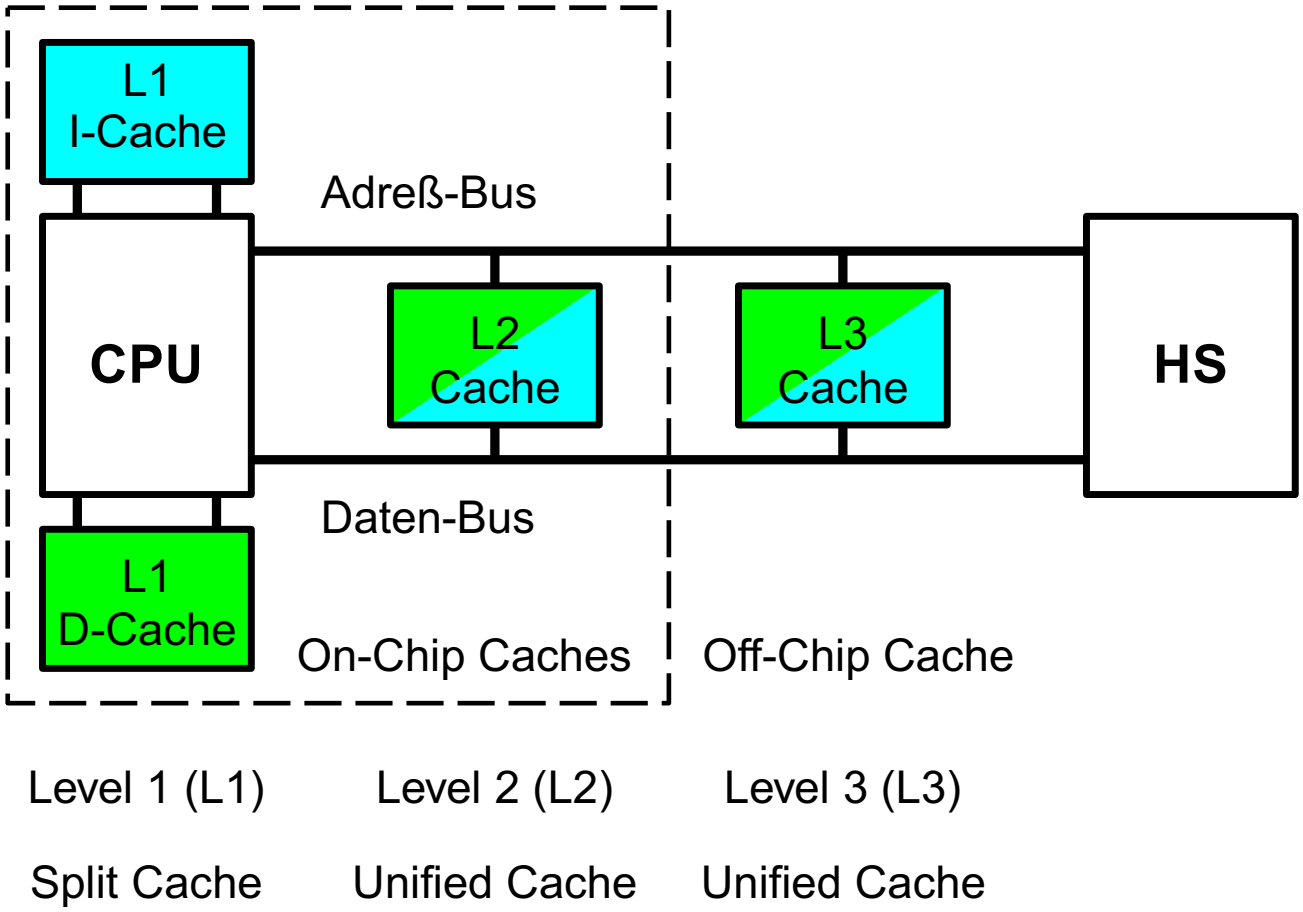
Split Cache, getrennt
für Daten und Befehle
(separate Lokalitäten)



Princeton - Architektur

Unified Cache, gemeinsam
für Daten und Befehle
(überlagerte Lokalitäten)

Cache-Architektur mit drei Ebenen



Cache Assoziativität

Vollassoziativer Cache (fully-associative)

Ein Speicherblock des HS kann sich in einer beliebigen Cache-Zeile des Caches befinden. Der Cache besteht aus einem Satz ($s=1$) mit $k=n_{cl}$ Zeilen

→ k-fach assoziativ (k-Wege).

Direkt abgebildeter Cache (direct-mapped)

Ein Speicherblock des HS kann sich nur in einer bestimmten Zeile des Caches befinden. Die Zeile wird durch den Index in der HS-Adresse festgelegt. Der Cache besteht aus $s=n_{cl}$ Sätzen mit je einer Cache-Zeile ($k=1$)

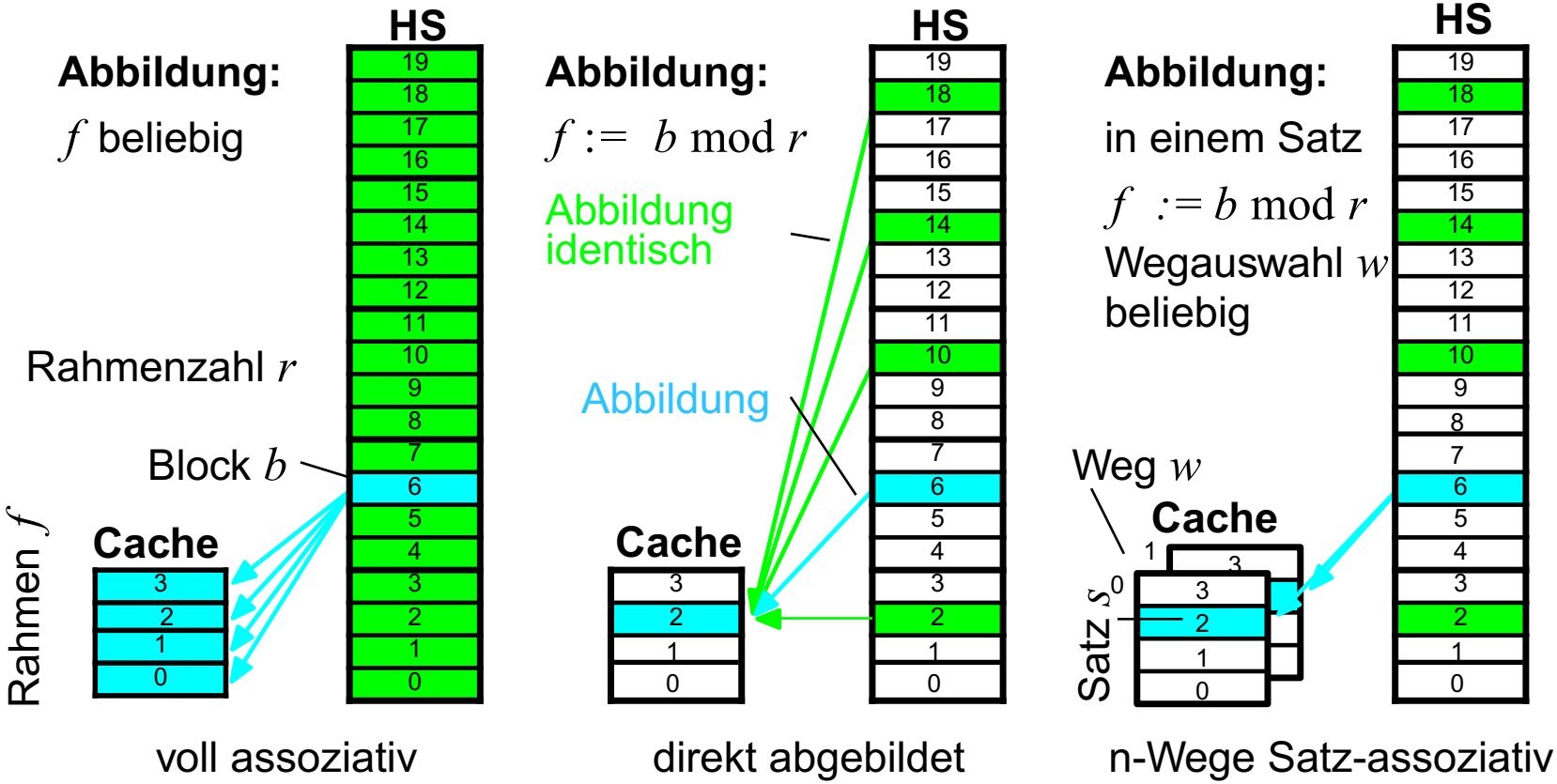
→ 1-fach assoziativ (1-Weg).

K-Wege satzassoziativer Cache (k-way set-associativ)

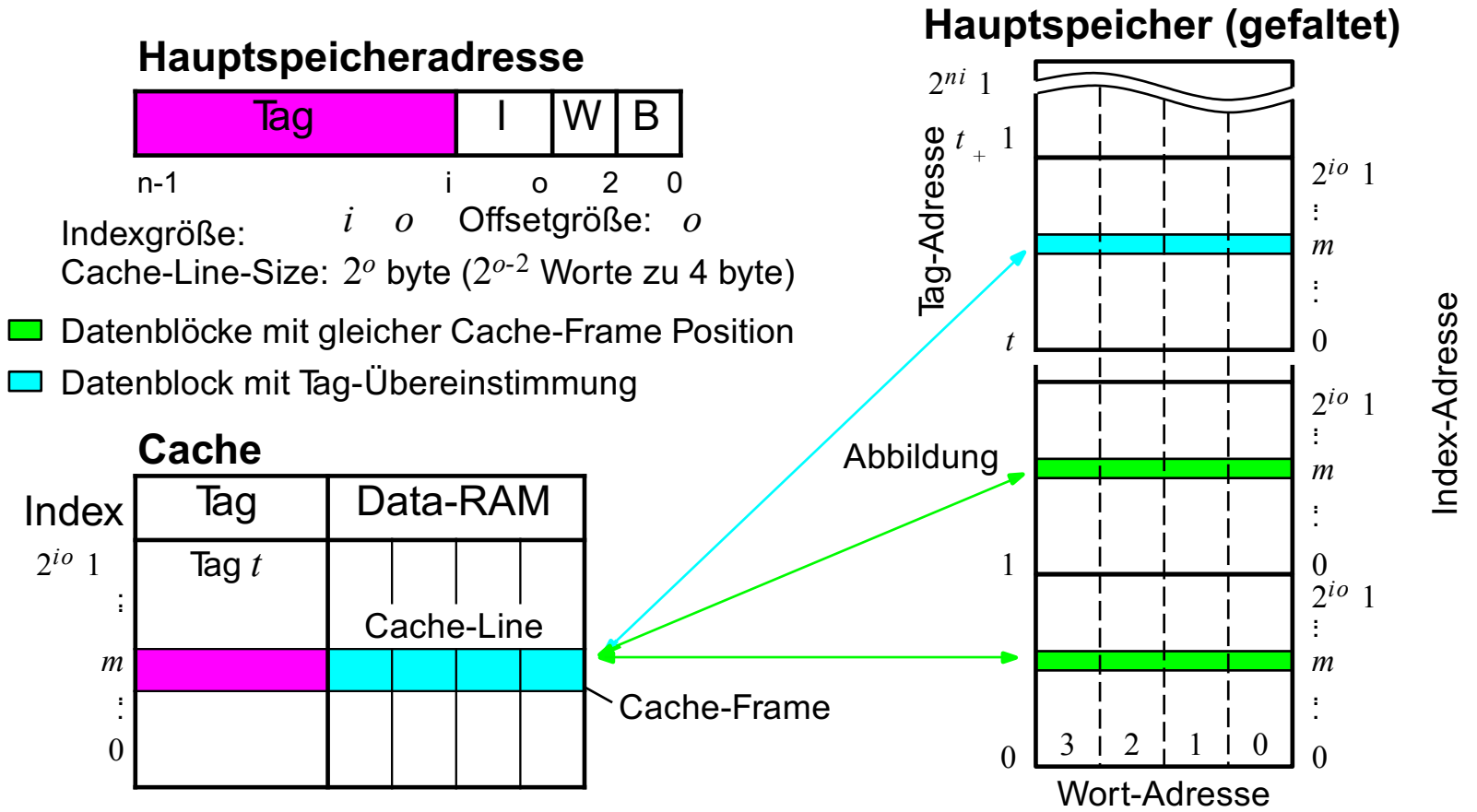
Ein Speicherblock des HS kann sich nur in einem bestimmten Satz des Caches befinden. Der Satz wird durch den Index in der HS-Adresse festgelegt. Der Cache besteht aus $s=n_{cl}/k$ Sätzen mit je k Cache-Zeilen

→ k-fach satzassoziativ (k-Wege).

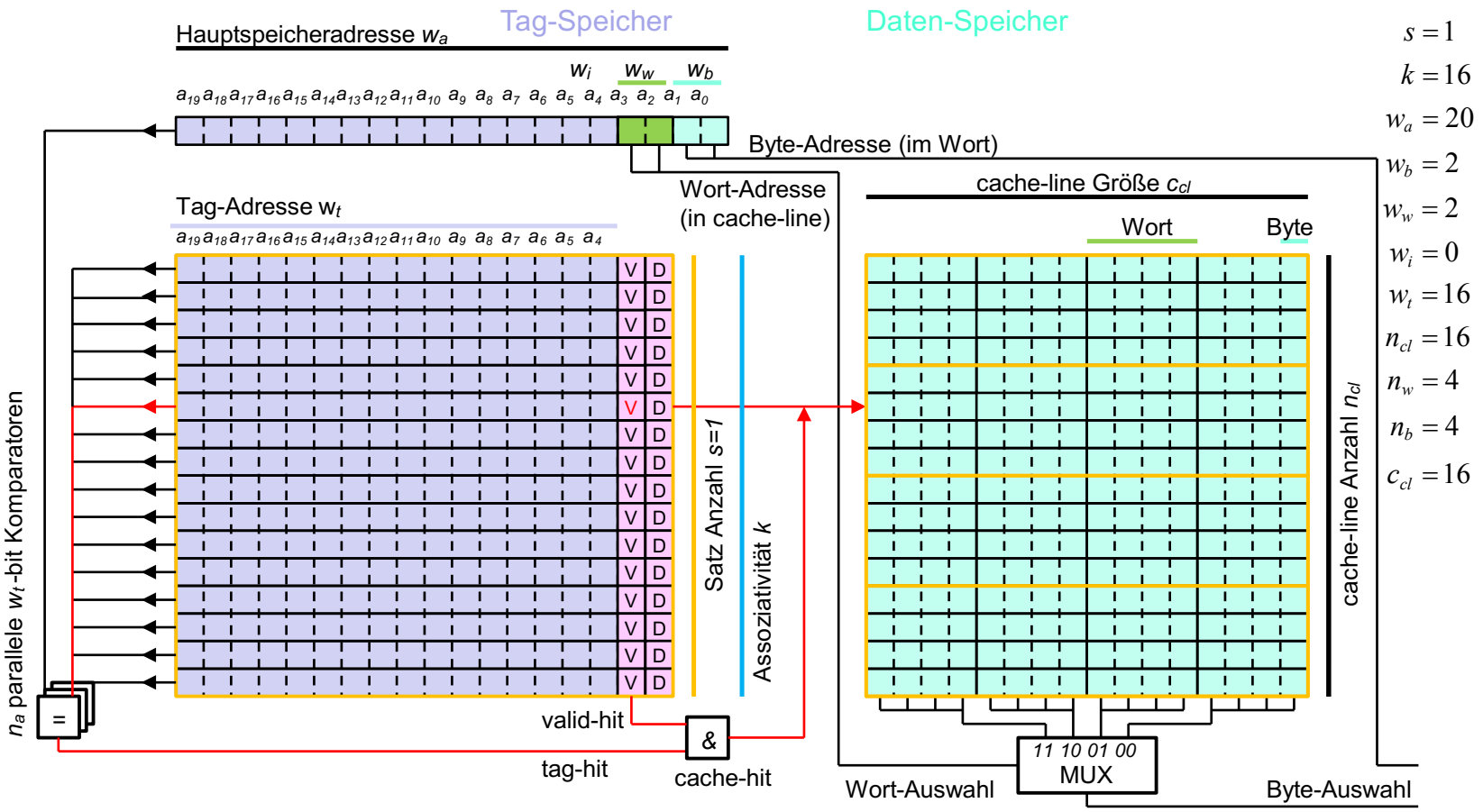
Cache-Organisation – Abbildungsproblem



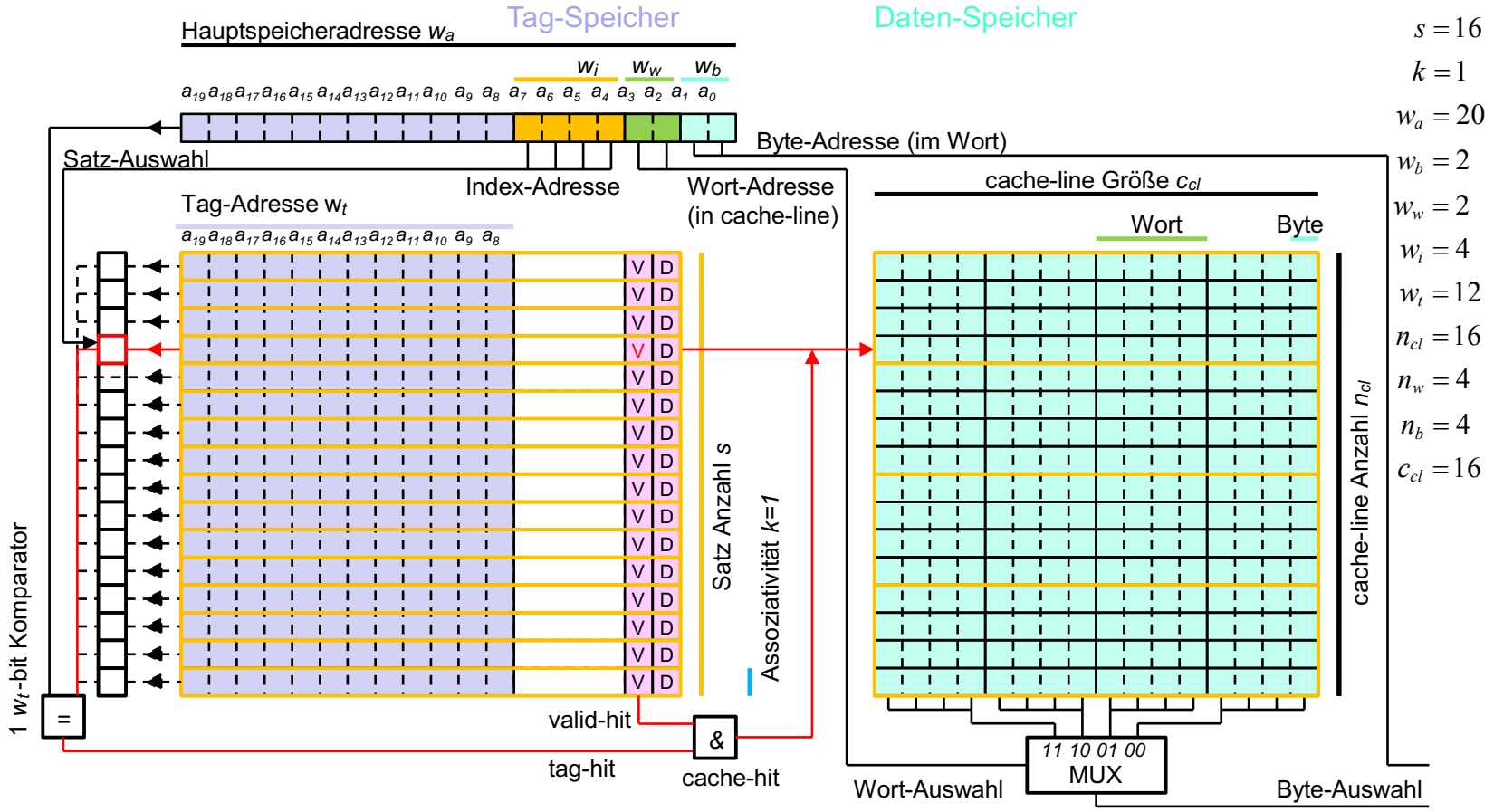
Beispiel – Abbildung: Cache - Speicher



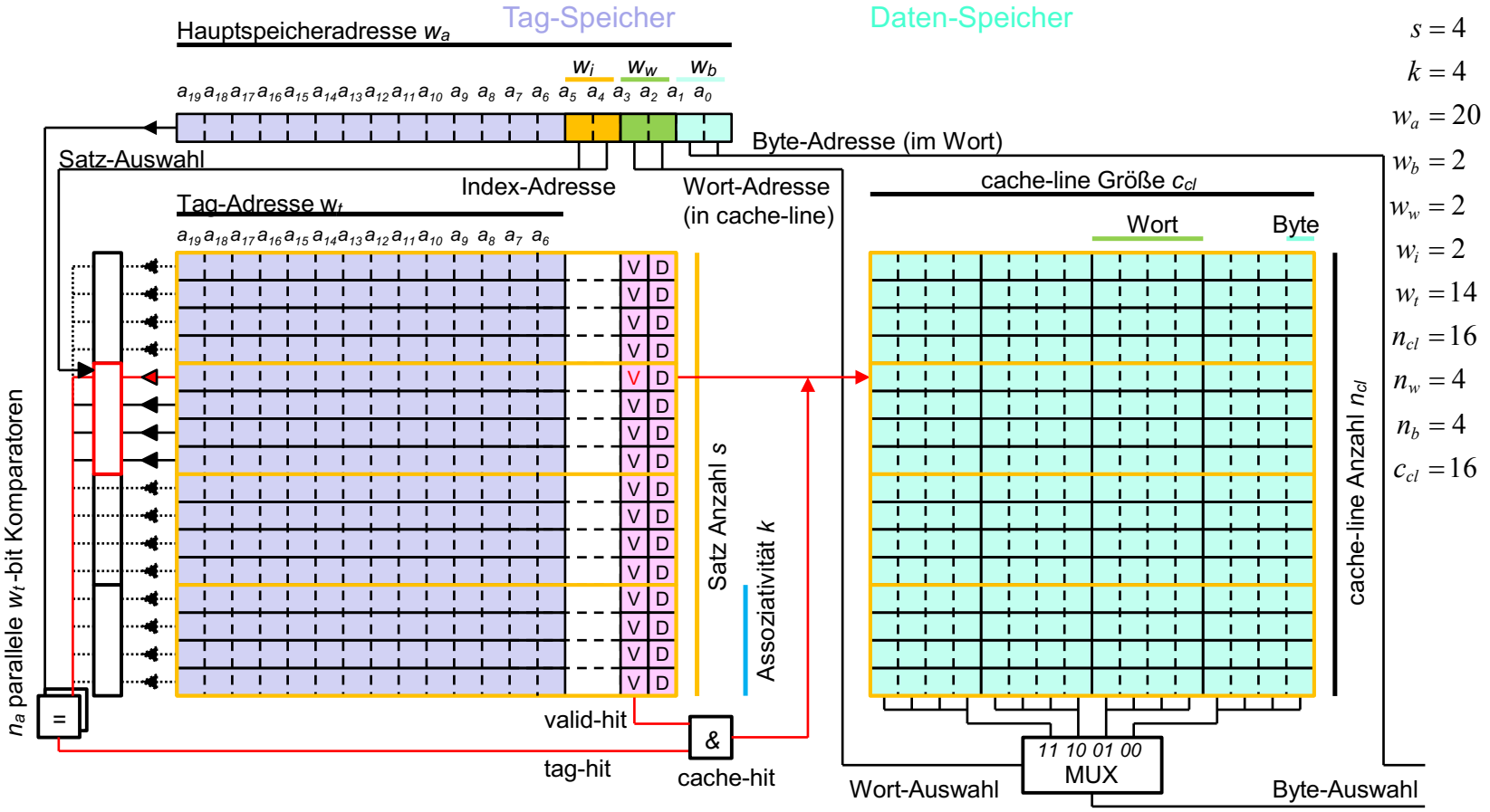
Vollassoziativer Cache (fully – k-way – associative)



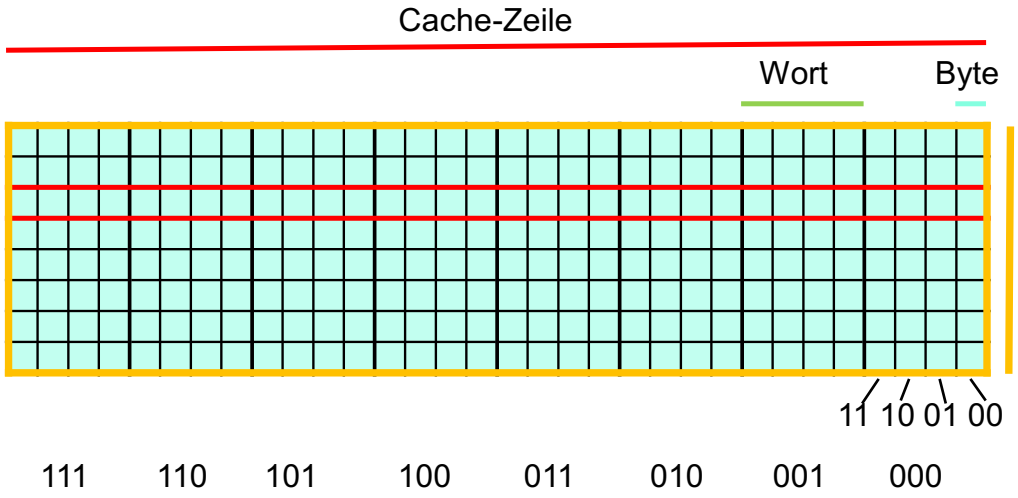
Direkt abgebildeter Cache (1-way associative: direct mapped)



K-Wege satzassoziativer Cache (k-way set associative)



Aufbau eines Cache-Satzes, Cache-Zeile



Cache-Zeilenkapazität c_{cl} Byte

$$c_{cl} = 2^{w_w} \cdot 2^{w_b} = n_w \cdot n_b$$

Satzkapazität c_s Byte

$$c_s = k \cdot c_{cl}$$

Byteadresse w_b bit

Wortadresse w_w bit

$k = 8$

$w_b = 2$

$w_w = 3$

$n_w = 8$

$n_b = 4$

$c_{cl} = 32$

vollassoziativ: $s = 1, \quad k = n_{cl}, \quad c_s = n_{cl} \cdot c_{cl}$

direkt abgebildet: $s = n_{cl}, \quad k = 1, \quad c_s = c_{cl}$

k-Wege satzassoziativ: $s = n_{cl} / k, \quad c_s = k \cdot c_{cl}$

Daten-Speicher: $c_d = n_{cl} \cdot c_{cl}$

Cache-Berechnungen

w_a Bitanzahl der Hauptspeicheradresse

w_t Bitanzahl der Tag-Adresse

w_i Bitanzahl der Index-Adresse

w_w Bitanzahl der Wort-Adresse

w_b Bitanzahl der Byte-Adresse

n_{cl} Anzahl der Cache-Zeilen

s Anzahl der Sätze (Mengen)

k Assoziativität im Satz

c_{cl} Kapazität der Cache-Zeile (Byte)

c_d Kapazität des Daten-Speichers (Byte)

c_t Kapazität des Tag-Speichers, ohne V,D

$n_w = 2^{w_w}$ Anzahl der Wörter je Cache-Zeile

$n_b = 2^{w_b}$ Anzahl der Byte je Wort

$s = 2^{w_i}$

$c_s = k \cdot c_{cl}$ Kapazität eines Satzes (Byte)

Cache-Berechnungen

Hauptspeicheradresse (Aufteilung)

$w_a = w_t + w_i + w_w + w_b$ (Summe der Bitanzahlen)

Cache-Zeilengröße (cache line size)

$c_{cl} = 2^{w_w} \cdot 2^{w_b}$ Byte

$= n_w \cdot n_b$ Byte (Wort-Anzahl je Cache-Zeile * Byte-Anzahl je Wort)

Cache-Zeilenanzahl

$n_{cl} = 2^{w_i} \cdot k = s \cdot k$ (Satz-Anzahl je Cache * Assoziativität je Satz)

Daten-Speichergröße

$c_d = n_{cl} \cdot c_{cl}$ (Cache-Zeilenanzahl * Cache-Zeilengröße)

Tag-Speichergröße (ohne V,D)

$c_t = w_t \cdot n_{cl}$ Bit (Bitanzahl der Tagadresse * Cache-Zeilenanzahl)

Cache-Analyse

Beispielaufgabe: Ein 4-fach satzassoziativer Cache hat eine Daten-Speichergröße von 4 MiByte. Die Cache-Zeile enthält 16 Worte zu je 4 Byte. Bestimmen Sie die Aufteilung der Hauptspeicheradresse (32 bit) sowie die Größe des Tag-Speichers, ohne V- und D-Bit!

1. Bitanzahl der Wort-, Byteadresse

$$w_w = ld16 = 4, \quad w_b = ld4 \quad = 2$$

2. Cache-Zeilengröße

$$c_{cl} = n_w \cdot n_b = 64 \text{ Byte}$$

3. Daten-Speichergröße

$$c_d = n_{cl} \cdot c_{cl}, \quad n_{cl} = \frac{c_d}{c_{cl}} = \frac{4 \cdot 2^{20}}{64} = 2^{16}$$

4. Cache-Zeilenanzahl

$$n_{cl} = 2^{w_i} \cdot k, \quad w_i = ld\,\frac{n_{cl}}{k} = ld\,\frac{2^{16}}{4} = 14$$

5. Bitanzahl der Tag-Adresse

$$w_a = w_t + w_i + w_w + w_b$$

$$w_t = w_a - (w_i + w_w + w_b) = 12$$

6. Tag-Speichergröße

$$c_t = w_t \cdot n_{cl} = 3 \cdot 2^{18} \text{ Bit}$$

$$w_a = 32, \quad w_t = 12, \quad w_i = 14$$

$$w_w = 4, \quad w_b = 2, \quad ct = 3 \cdot 2^{18} \text{ Bit}$$

Cache Assoziativität – Vorteile/ Nachteile

1. **Vollassoziativer Cache (fully-associative)**

- sehr gute Trefferrate, universell einsetzbar
- aufwändige HW-Realisierung, n_{cl} voll-parallele Komparatoren für die gesamte Tag-Breite, langsam

2. **Direkt abgebildeter Cache (direct-maped)**

- einfache HW-Realisierung, direkte Zeilen-Adressierung mit Index, nur ein Vergleicher mit einem um den Index reduzierten Tag, sehr schnell
- schlechte Trefferrate, Ping-Pong Effekt (ständiges Ein- und Auslagern der Daten, cache trashing)

3. **K-Wege satzassoziativer Cache (k-way set-associativ)**

- Kompromiss aus vollassoziativem und direkt abgebildetem Cache, 8-fach satzassoziativer Cache ist nahezu so gut wie ein vollassoziativer
- gute Trefferrate bei hoher Geschwindigkeit – Kompromiss aus 1. und 2.

Cache-Organisation – Ersetzungsproblem

Welche Cache-Line wird beim Nachladen eines mehrfach assoziativen Caches verdrängt, ersetzt (beim direkt abgebildeten Cache kein Ersetzungsproblem)?

LRU (Least Recently Used):
Ersetze am längsten nicht mehr benutzte Cache-Line. Für die Entscheidung ist eine Referenzstatistik erforderlich (UC – Utilization Counter).

Zufallsprinzip (RANDOM):
Ersetze eine Cache-Line nach dem Zufallsprinzip (Pseudo-Zufallszahlen).

FIFO (First In First Out):
Ersetze älteste geladene Cache-Line (erfordert Ladestatistik).

LFU (Least Frequently Used):
Ersetze am wenigsten benutzte Cache-Line (erfordert Benutzungsstatistik).

Round Robbin: Ersetze Cache-Line in vorher festgelegter Reihenfolge.

Optimal (Perfect): Ersetze in Zukunft nicht mehr benötigte Cache-Line.

Cache-Organisation – Aktualisierungsproblem

Datenkonsistenz und Datenkohärenz

Wann werden welche Daten im Cache oder und im Hauptspeicher aktualisiert?

Konsistenz:

Sowohl der Cache als auch der Hauptspeicher enthalten stets die identischen aktuellen Daten. Alle untergeordneten Hierarchieebenen enthalten stets auch die Daten aller übergeordneten Ebenen in identischer Form.

Kohärenz:

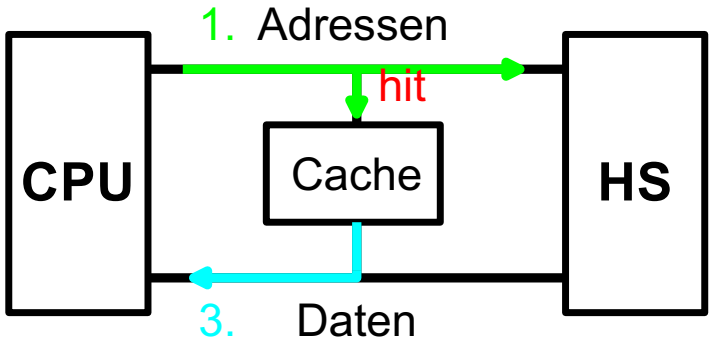
Der CPU werden immer die aktuellen Daten bereitgestellt, unabhängig davon, ob sie im Cache oder im Hauptspeicher stehen. Auf veraltete Daten (Stale Data) darf nicht zugegriffen werden (⟹ Konsistenzabschwächung).

Problemfälle:

- Auslagern, Rückschreiben in untergeordnete Hierarchieebenen (Castoff).
- Mehrere Master im System (CPU und DMAC, Direct Memory Access).

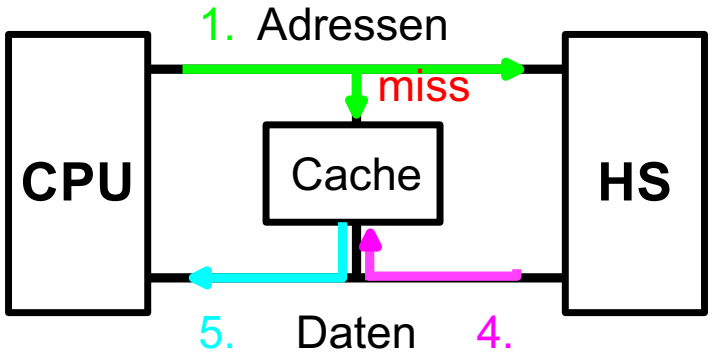
Cache Lese-Strategie (Read Strategy)

Treffer (Read-Hit)



1. Cache und HS adressieren
2. Cache-Hit, HS-Zugriff abbrechen
3. Lesen der Daten aus dem Cache

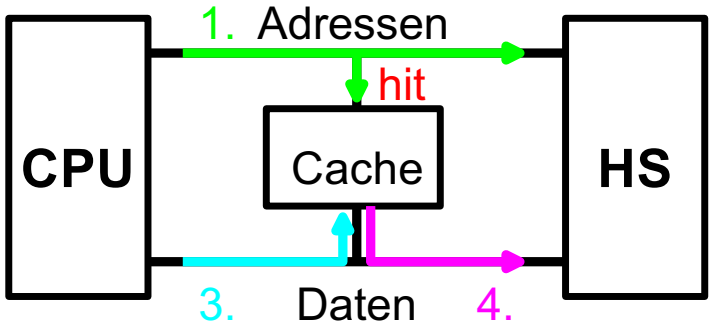
Fehlzugriff (Read-Miss)



1. Cache und HS adressieren
2. Cache-Miss, HS-Zugriff fortsetzen
3. Verdrängung einer Cache-Line (Ersetzungsproblem, Castoff)
4. Cache-Line aus HS in Cache laden
5. Lesen der Daten aus dem Cache

Treffer (Write-Hit) Cache Schreib-Strategie (Write Strategy)

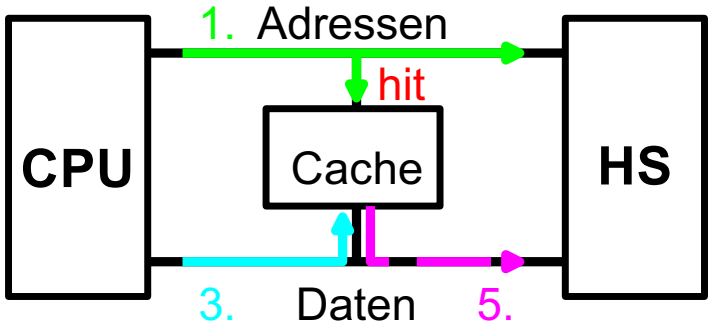
Write-Through (WT)



1. Cache und HS adressieren
2. Cache-Hit
3. Schreiben der Daten in den Cache
4. HS unverzüglich aktualisieren (Schreibpuffer, Write-Buffer)

⇨ Zeitaufwendig aber konsistent

Write-Back, Copy-Back (WB)

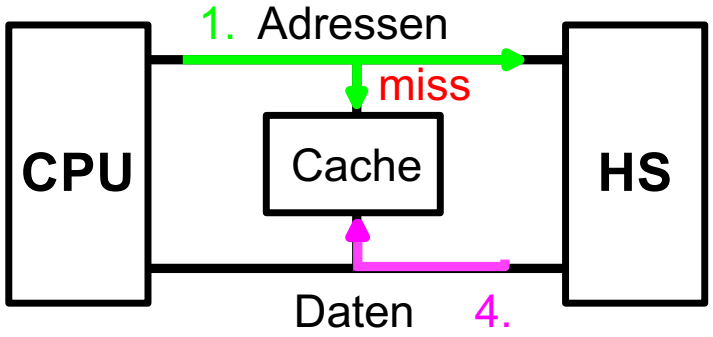


1. Cache und HS adressieren
2. Cache-Hit
3. Schreiben der Daten in den Cache
4. Setzen des Dirty-Bit
5. HS bei Verdrängung aktualisieren

⇨ Probleme mit Daten-Konsistenz

Fehler (Write-Miss) Cache Schreib-Strategie (Write Strategy)

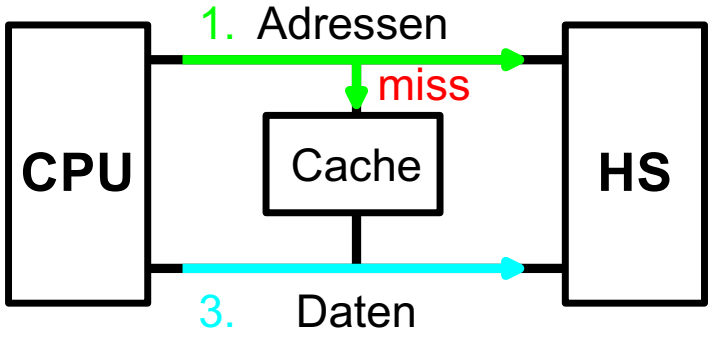
Write-Allocate



1. Cache und HS adressieren
2. Cache-Miss, HS-Zugriff fortsetzen
3. Verdrängen einer Cache-Line (Ersetzungsproblem, Castoff)
4. Cache-Line aus HS in Cache laden

Weiter analog Treffer (Write-Hit), typisch:

Write-Around (No-Write-Allocate)



1. Cache und HS adressieren
2. Cache-Miss, HS-Zugriff fortsetzen
3. Schreiben der Daten in den HS (Cache wird nicht geladen)

Write-Allocate mit Write-Back
Write-Around mit Write-Through

Hennessy/ Patterson Chapter 5

Cache Misses

- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
  - Instruction cache miss
    - Restart instruction fetch
  - Data cache miss
    - Complete data access

Write-Through

- On data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- Write through: also update memory
- But makes writes take longer
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI = 1 + 0.1×100 = 11
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full

Write-Back

- Alternative: On data-write hit, just update the block in cache
  - Keep track of whether each block is dirty
- When a dirty block is replaced
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first

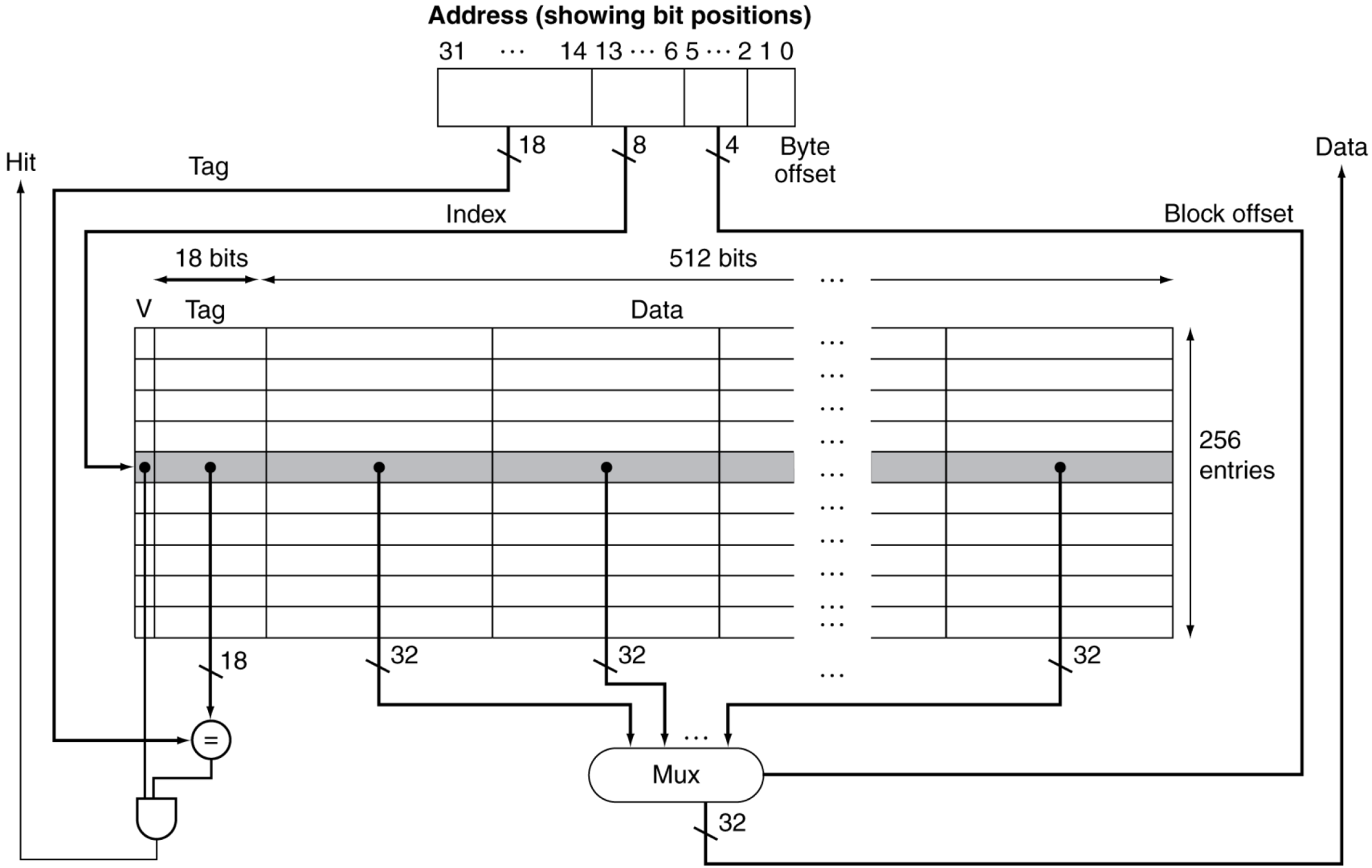
Write Allocation

- What should happen on a write miss?
- Alternatives for write-through
  - Allocate on miss: fetch the block
  - Write around: don't fetch the block
    - Since programs often write a whole block before reading it (e.g., initialization)
- For write-back
  - Usually fetch the block

Example: Intrinsity FastMATH

- Embedded MIPS processor
  - 12-stage pipeline
  - Instruction and data access on each cycle
- Split cache: separate I-cache and D-cache
  - Each 16KB: 256 blocks × 16 words/block
  - D-cache: write-through or write-back
- SPEC2000 miss rates
  - I-cache: 0.4%
  - D-cache: 11.4%
  - Weighted average: 3.2%

Example: Intrinsity FastMATH



Measuring Cache Performance

- Components of CPU time
  - Program execution cycles
    - Includes cache hit time
  - Memory stall cycles
    - Mainly from cache misses
- With simplifying assumptions:

$$\text{Memory stall cycles}$$

$$= \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

Cache Performance Example

- Given
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions
- Miss cycles per instruction
  - I-cache: $0.02 \times 100 = 2$
  - D-cache: $0.36 \times 0.04 \times 100 = 1.44$
- Actual CPI = 2 + 2 + 1.44 = 5.44
  - Ideal CPU is 5.44/2 =2.72 times faster

Average Access Time

- Hit time is also important for performance
- Average memory access time (AMAT)
  - AMAT = Hit time + Miss rate × Miss penalty
- Example
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
  - AMAT = 1 + 0.05 × 20 = 2ns
    - 2 cycles per instruction

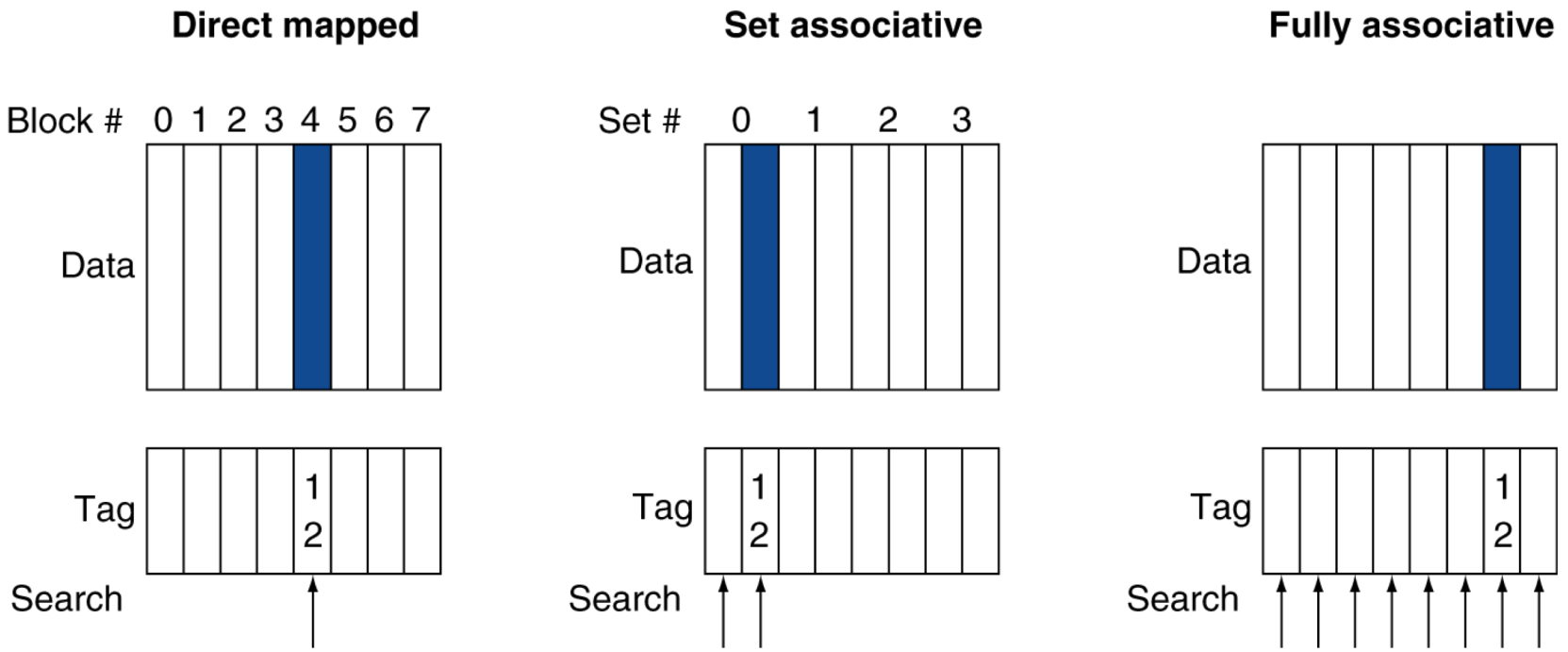
Performance Summary

- When CPU performance increases
  - Miss penalty becomes more significant
- Decreasing base CPI
  - Greater proportion of time spent on memory stalls
- Increasing clock rate
  - Memory stalls account for more CPU cycles
- Can't neglect cache behavior when evaluating system performance

Associative Caches

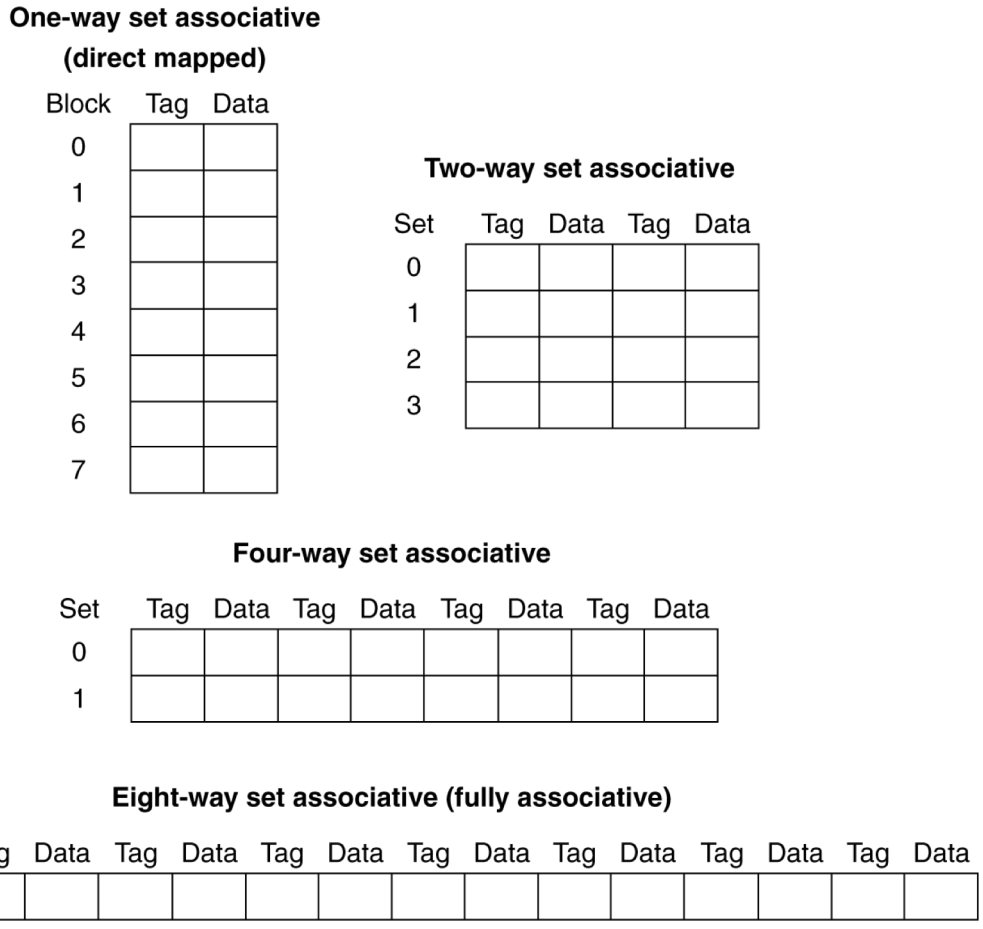
- Fully associative
  - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once
  - Comparator per entry (expensive)
- *n*-way set associative
  - Each set contains *n* entries
  - Block number determines which set
    - (Block number) modulo (#Sets in cache)
  - Search all entries in a given set at once
  - *n* comparators (less expensive)

Associative Cache Example



Spectrum of Associativity

- For a cache with 8 entries



Associativity Example

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence: 0, 8, 0, 6, 8
- Direct mapped

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	**Mem[0]**			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		**Mem[6]**	
8	0	miss	**Mem[8]**		Mem[6]	

Associativity Example

- 2-way set associative

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	**Mem[0]**			
8	0	miss	Mem[0]	**Mem[8]**		
0	0	hit	**Mem[0]**	Mem[8]		
6	0	miss	Mem[0]	**Mem[6]**		
8	0	miss	**Mem[8]**	Mem[6]		

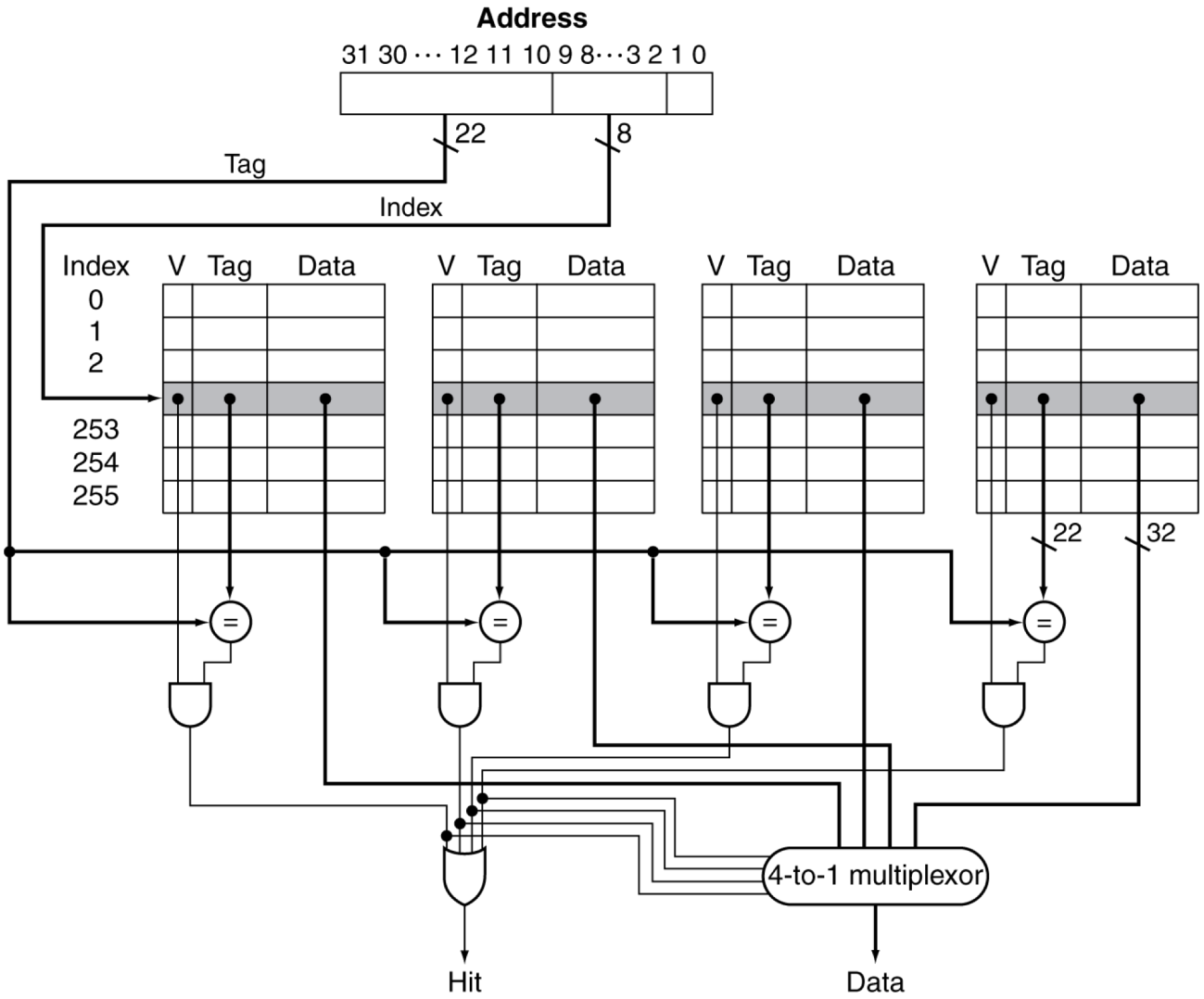
- Fully associative

Block address		Hit/miss	Cache content after access			
0		miss	**Mem[0]**			
8		miss	Mem[0]	**Mem[8]**		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	**Mem[6]**	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	

How Much Associativity

- Increased associativity decreases miss rate
  - But with diminishing returns
- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%

Set Associative Cache Organization



Replacement Policy

- Direct mapped: no choice
- Set associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
- Least-recently used (LRU)
  - Choose the one unused for the longest time
    - Simple for 2-way, manageable for 4-way, too hard beyond that
- Random
  - Gives approximately the same performance as LRU for high associativity

Multilevel Caches

- Primary cache attached to CPU
  - Small, but fast
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache

Multilevel Cache Example

- Given
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns
- With just primary cache
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - Effective CPI = 1 + 0.02 × 400 = 9

Example (cont.)

- Now add L-2 cache
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- Primary miss with L-2 hit
  - Penalty = 5ns/0.25ns = 20 cycles
- Primary miss with L-2 miss
  - Extra penalty = 500 cycles
- CPI = 1 + 0.02 × 20 + 0.005 × 400 = 3.4
- Performance ratio = 9/3.4 = 2.6

Multilevel Cache Considerations

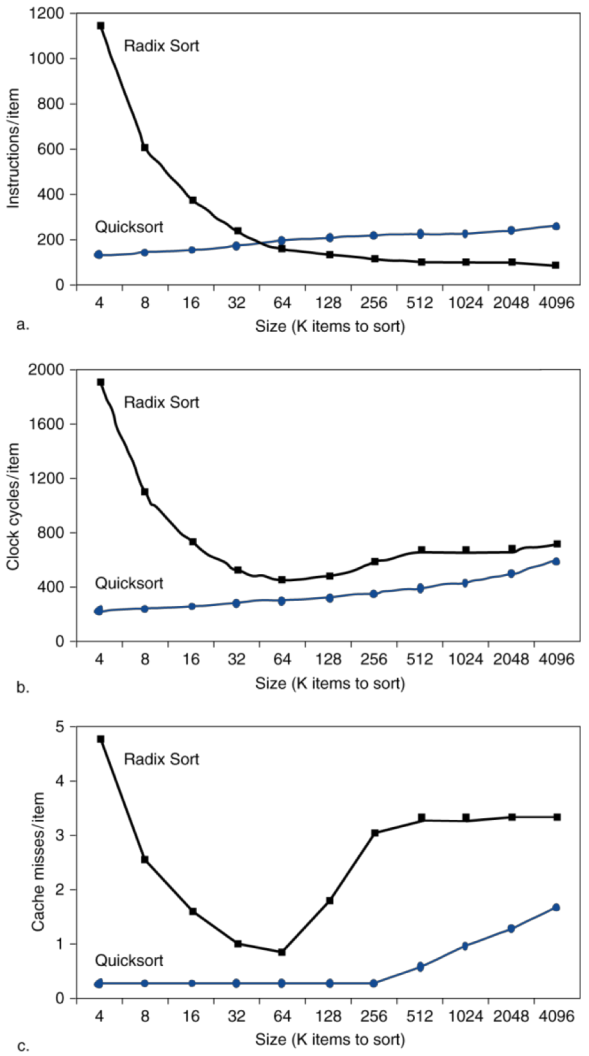
- Primary cache
  - Focus on minimal hit time
- L-2 cache
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact
- Results
  - L-1 cache usually smaller than a single cache
  - L-1 block size smaller than L-2 block size

Interactions with Advanced CPUs

- Out-of-order CPUs can execute instructions during cache miss
  - Pending store stays in load/store unit
  - Dependent instructions wait in reservation stations
    - Independent instructions continue
- Effect of miss depends on program data flow
  - Much harder to analyse
  - Use system simulation

Interactions with Software

- Misses depend on memory access patterns
  - Algorithm behavior
  - Compiler optimization for memory access



Software Optimization via Blocking

- Goal: maximize accesses to data before it is replaced
- Consider inner loops of DGEMM:

```
for (int j = 0; j < n; ++j)
{
  double cij = C[i+j*n];
  for( int k = 0; k < n; k++ )
    cij += A[i+k*n] * B[k+j*n];
  C[i+j*n] = cij;
}
```