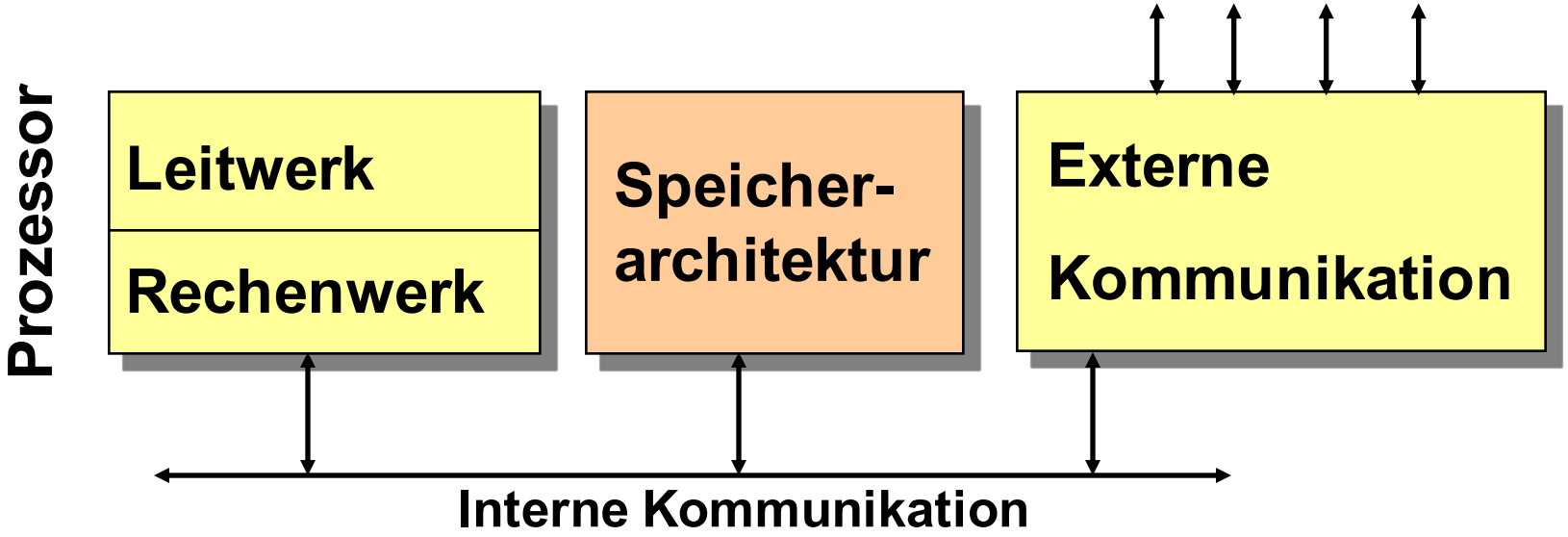
Rechnerarchitektur (RA)

Sommersemester 2020

Speicher

Prof. Dr. Jian-Jia Chen

Kontext



Die Wissenschaft Informatik befasst sich mit der Darstellung, Speicherung, Übertragung und Verarbeitung von Information. [Gesellschaft für Informatik]

Entwurfsziele

- Möglichst große Kapazität
- Möglichst kleine Zugriffszeiten
  - Geringe Zeit bis zur Verfügbarkeit eines Speicherinhalts (kleine *latency*)
  - Kleiner zeitlicher Abstand zwischen Übertragungen, hoher Durchsatz (großer *throughput*)
- Persistente (nicht-flüchtige) Speicherung
- Geringe Energieaufnahme
- Hohe Datensicherheit
- Geringer Preis
- Physikalisch klein

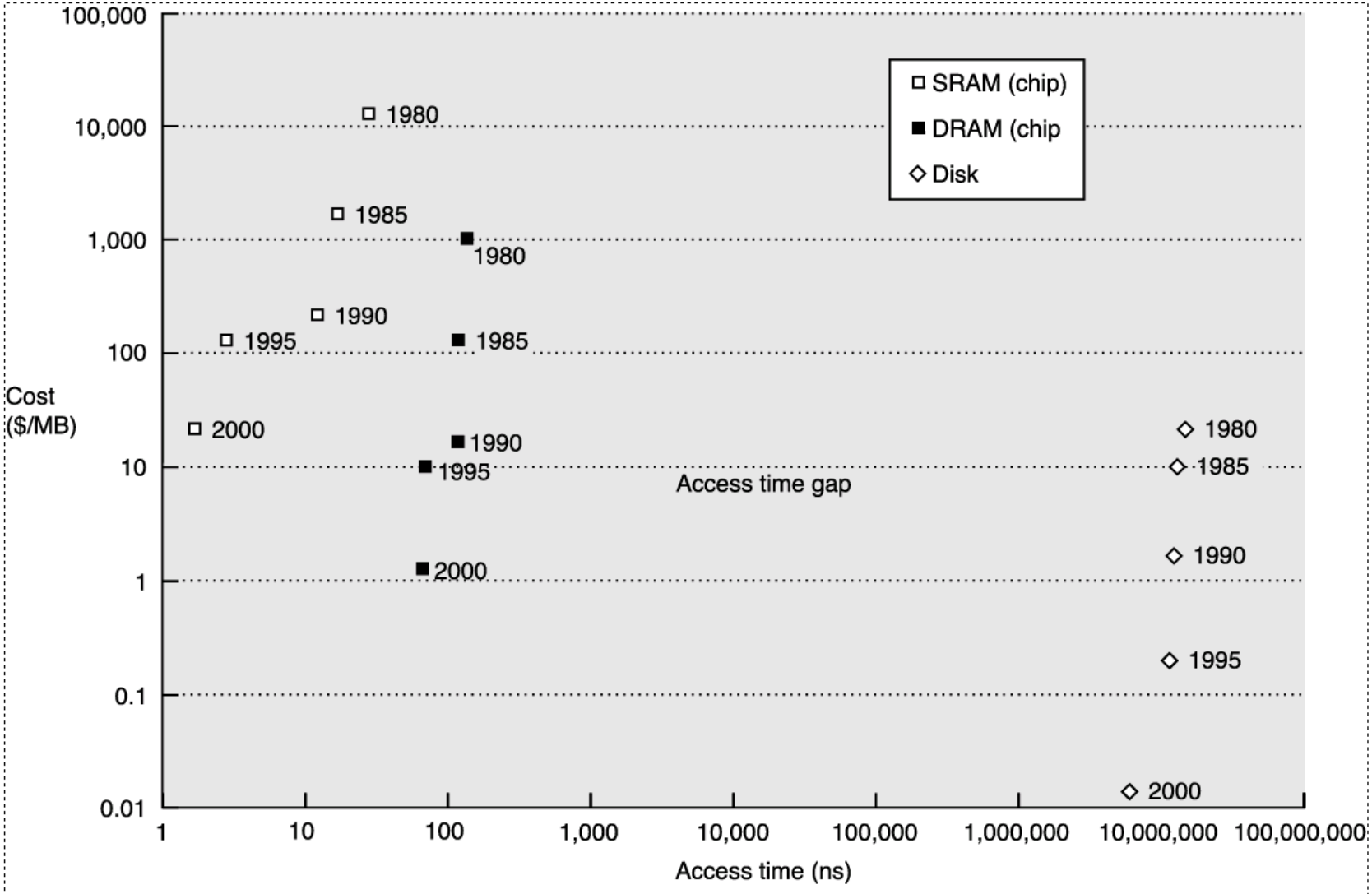
Entwurf von Speicherhierarchien

„Speicherhunger" von Applikationen/Entwicklern/...

"Ideally one would desire an indefinitely large memory capacity such that any particular ... word would be immediately available ..."

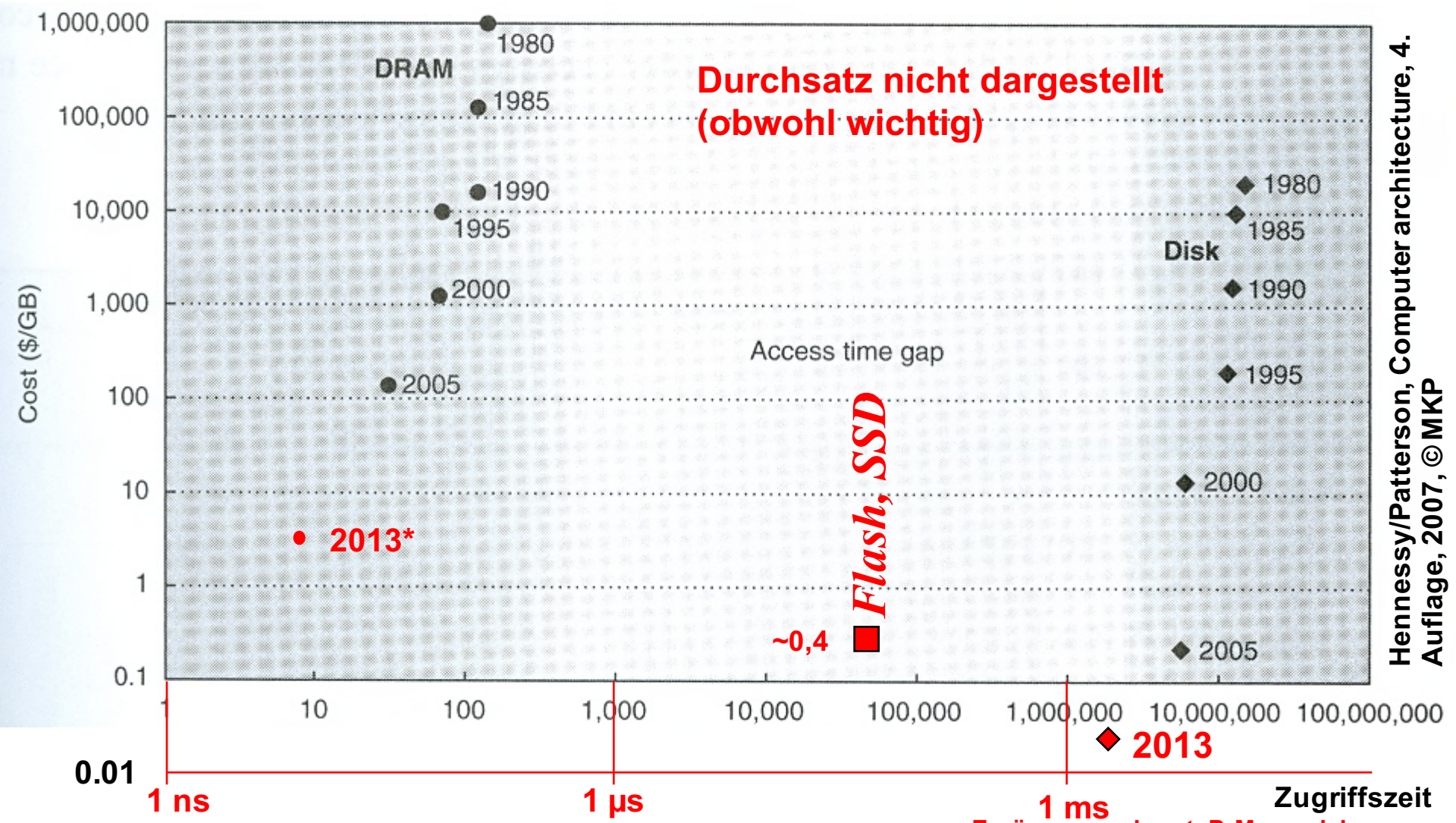
Burks, Goldstine, von Neumann
Preliminary Discussion of the Logical Design of an Electronic Computing Instrument (1946)

Die Realität: Kosten/Mbyte und Zugriffszeiten für verschiedene Speichermedien



[Hennessy/Patterson, *Computer Architecture*, 3. Aufl.]

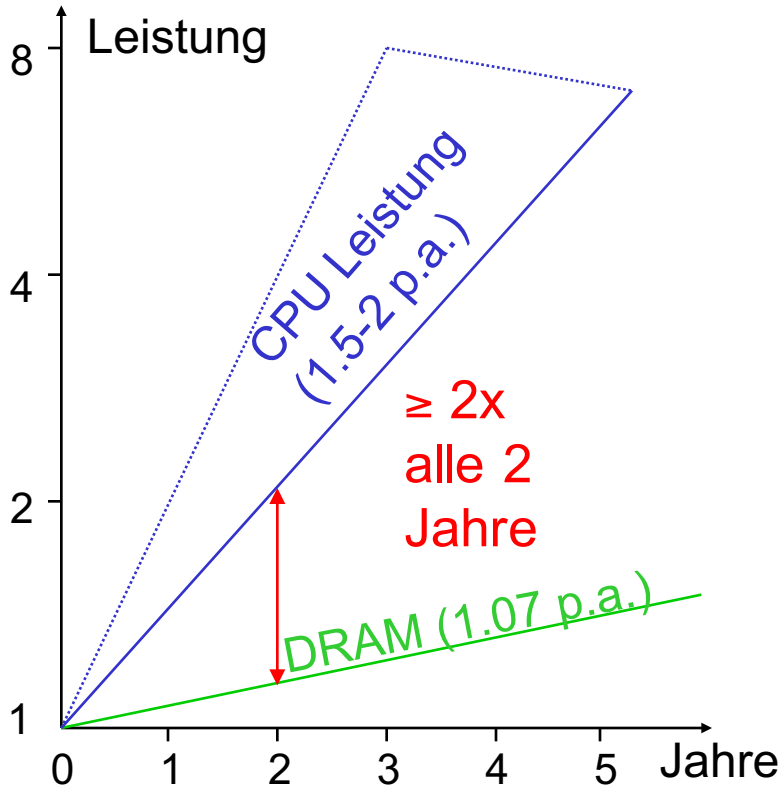
Kosten/Mbyte und Zugriffszeiten für verschiedene verfügbare Technologien



Ergänzungen in rot: P. Marwedel
*:www.jcmit.com/memoryprice.htm
en.wikipedia.org/wiki/CAS_latency

Trend der Leistungen von DRAM

Die Performanzlücke zwischen Prozessoren und DRAM:

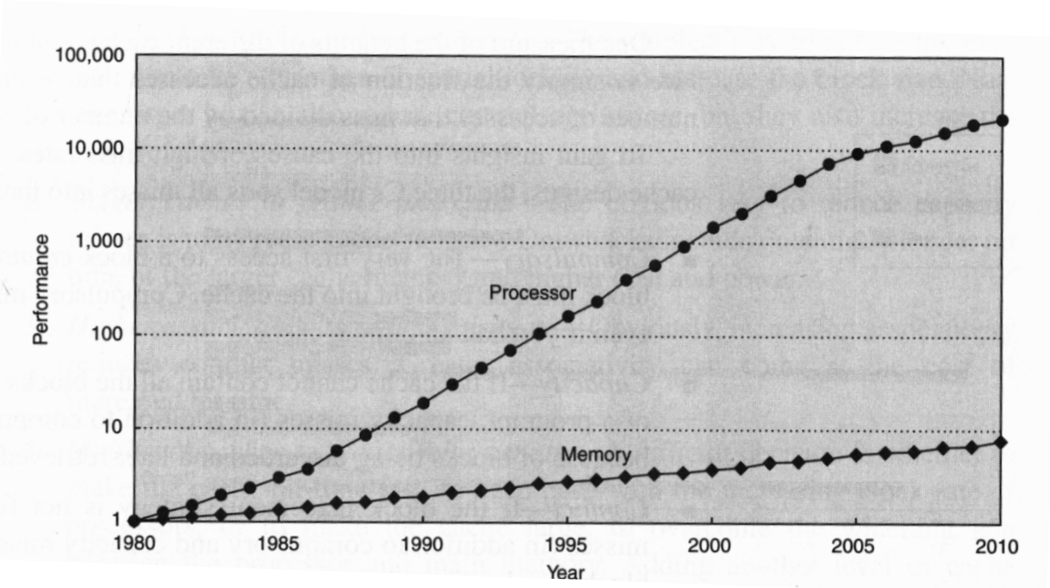


Ähnliche Probleme auch für Eingebettete Systeme

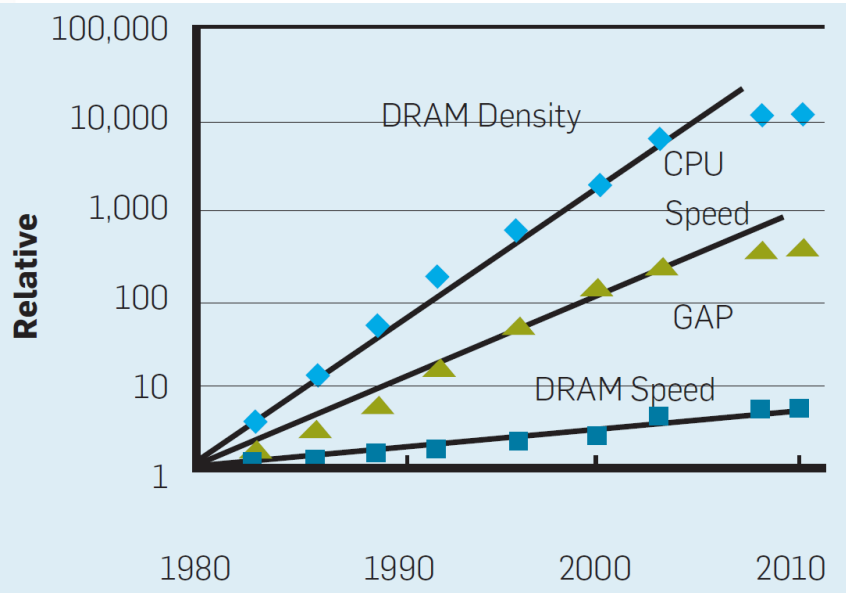
- ☞ Speicherzugriffszeiten ≫ Prozessorzykluszeiten
- ☞ „*Memory wall*" Problem

[P. Machanik: Approaches to Addressing the Memory Wall, TR Nov. 2002, U. Brisbane]

Der Speicher-„Flaschenhals“ (*Memory wall*)



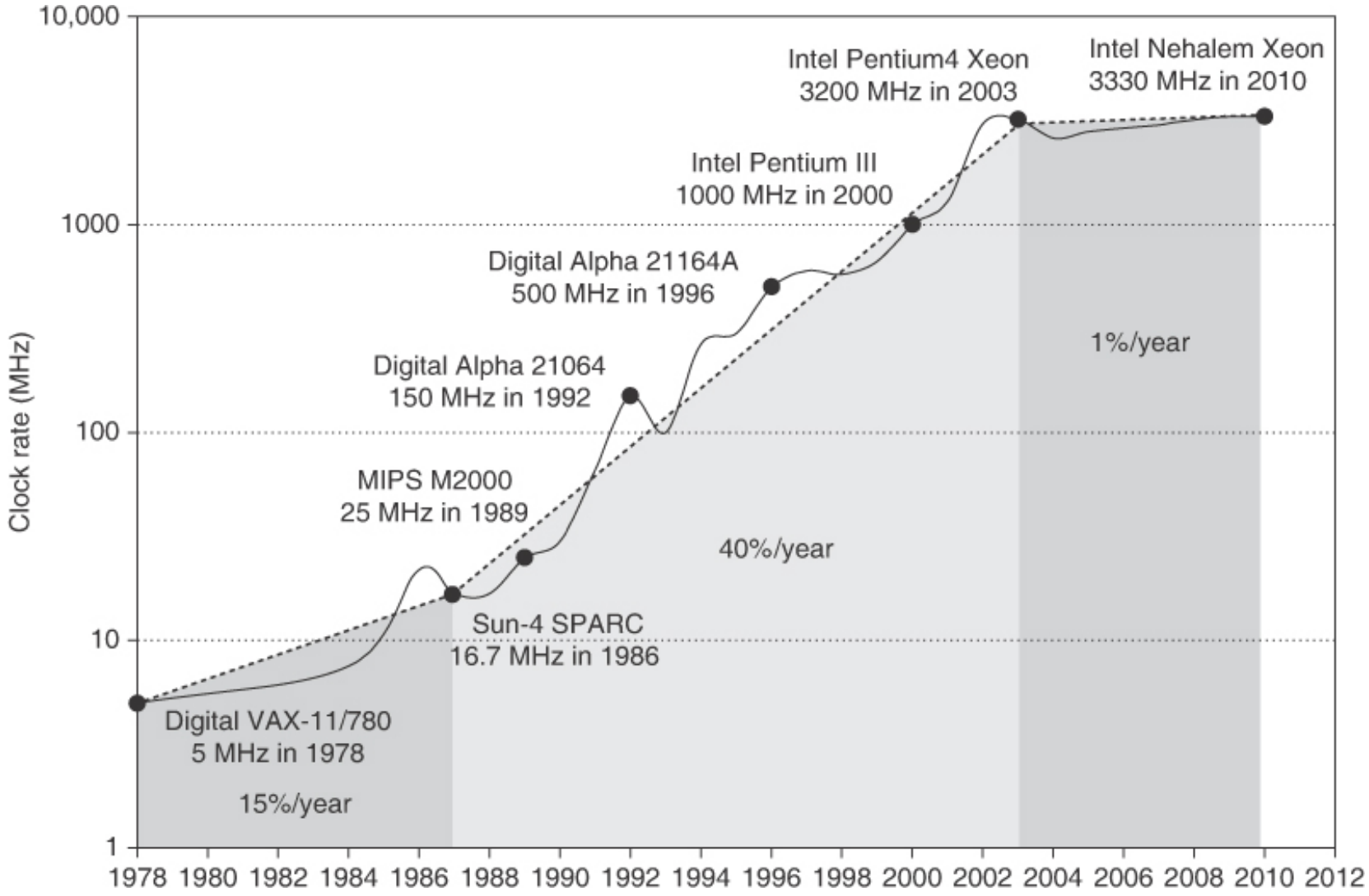
© Elsevier Science



Shekhar Borkar, Andrew A. Chien: The future of Microprocessors, *Communications of the ACM*, Mai 2011, S. 67ff, © ACM

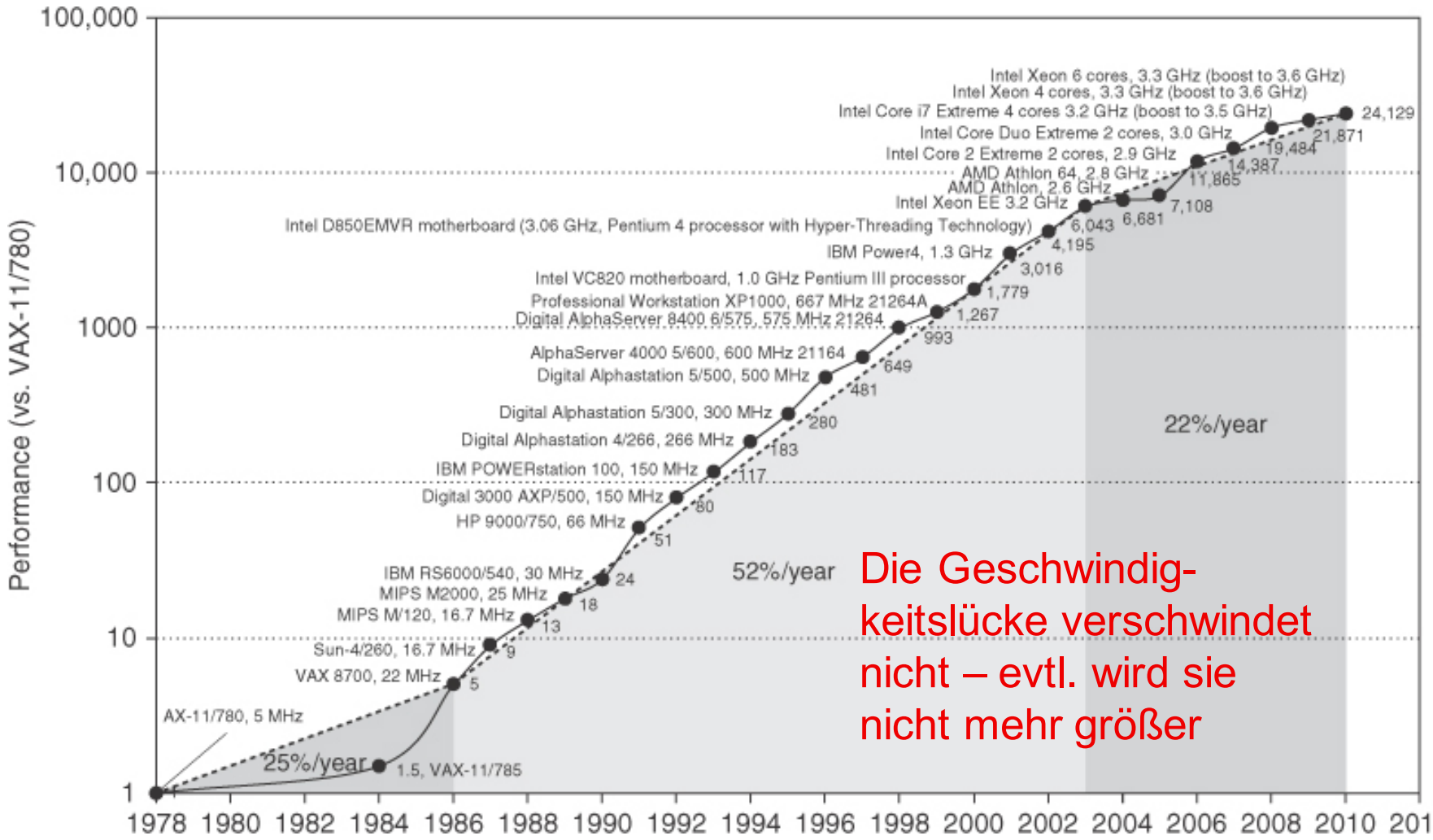
Geringe Steigerungsraten der Speicherzugriffsperformance

Wachstum der Taktraten ist weitgehend beendet

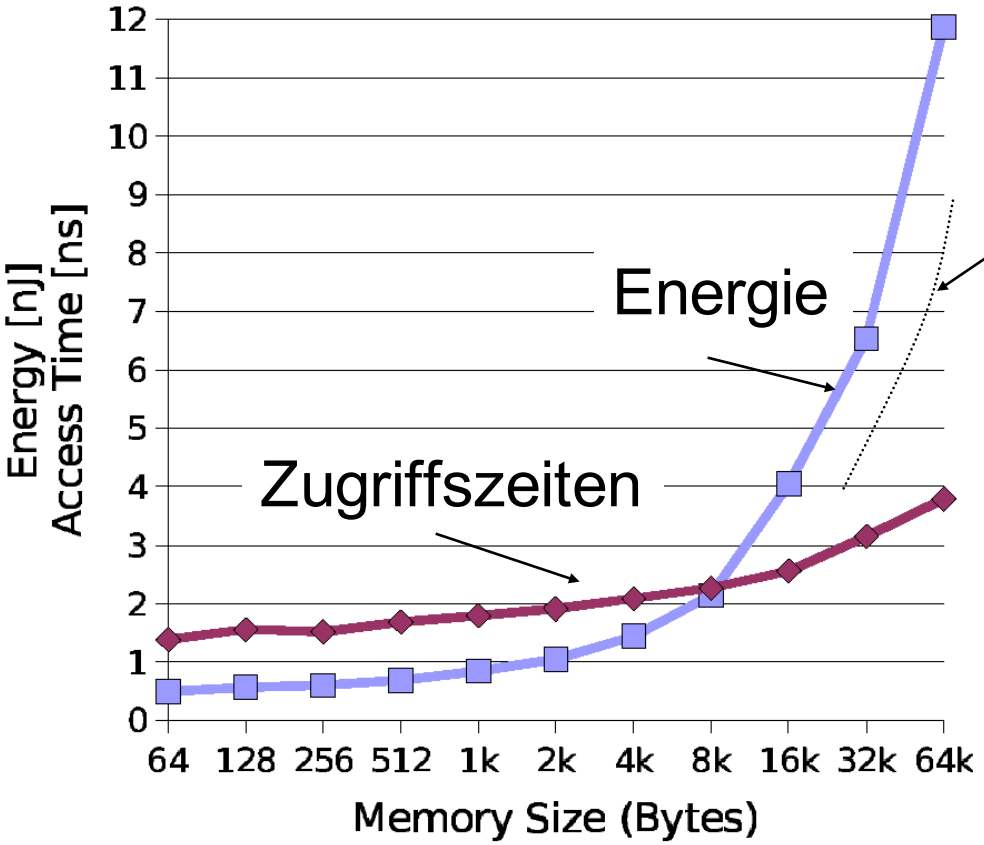


[Hennessy/Patterson: Computer Architecture, 5th ed., 2011]

(Parallele) Performanz wächst weiter



Abhängigkeit von der Speichergröße

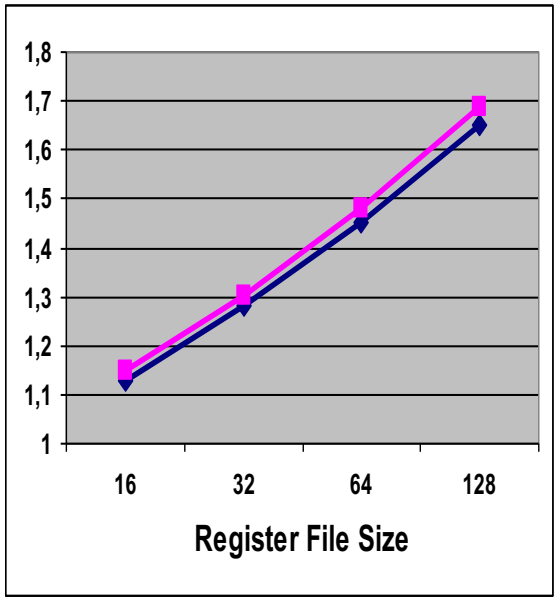


Anwendungen werden größer und größer …

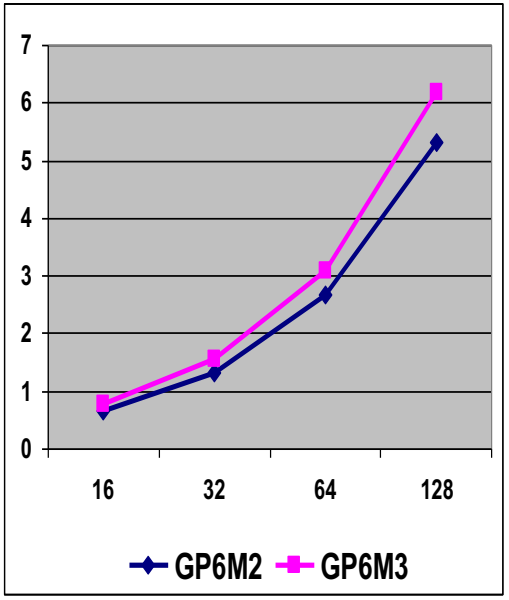
Quelle: CACTI

„Alles“ ist groß für große Speicher

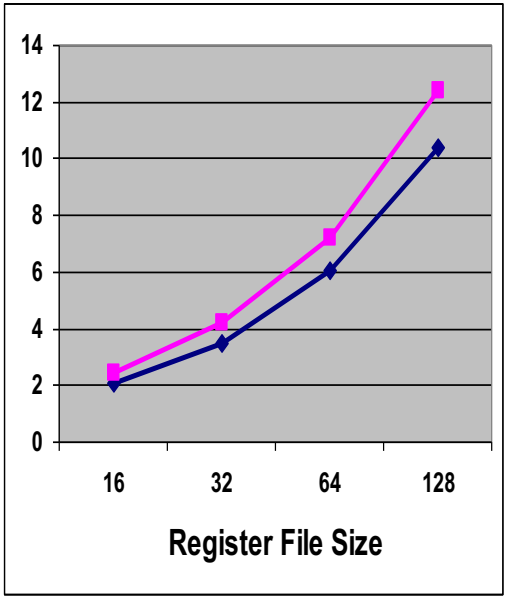
Zykluszeit (ns)*



Fläche ($\lambda^2 x 10^6$)



El. Leistung (W)



* Monolithic register file; Rixner’s et al. model [HPCA’00], Technology of 0.18 mm; VLIW configurations for a certain number of ports („GP*x*M*y*REG*z* where: x={6}, y={2, 3} and z={16, 32, 64, 128“}; Based on slide by and ©: Harry Valero, U. Barcelona, 2001

Speicherhierarchie

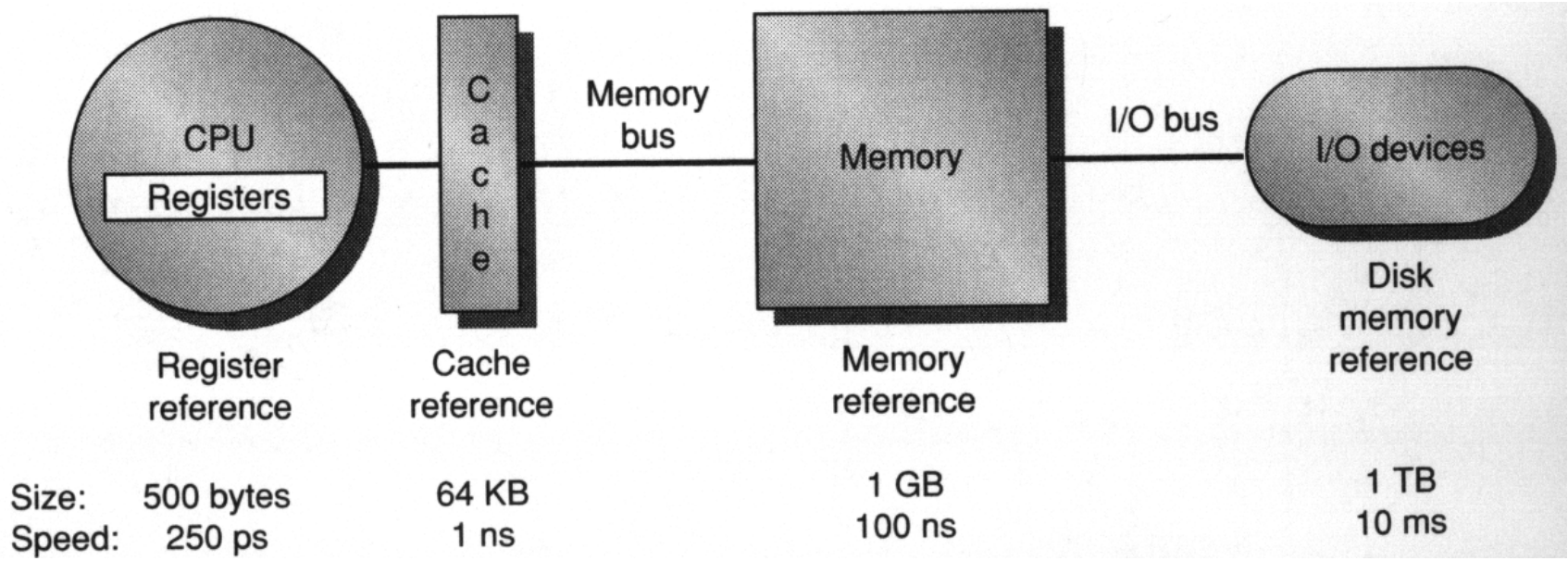
- Große Speicher sind langsam
- Anwendungen verhalten sich üblicherweise lokal
- ☞ Häufig benötigte Speicherinhalte in kleinen Speichern, seltener benötigte Inhalte in großen Speichern ablegen!
- ☞ Einführung einer „Speicherhierarchie“
- ☞ *Illusion* eines **großen Speichers** mit (durchschnittlich) **kleinen Zugriffszeiten**
- Enge Grenze für die Zugriffszeit wird selten garantiert.
- ☞ Spezielle Überlegungen bei Realzeitsystemen

Entwurf von Speicherhierarchien (2)

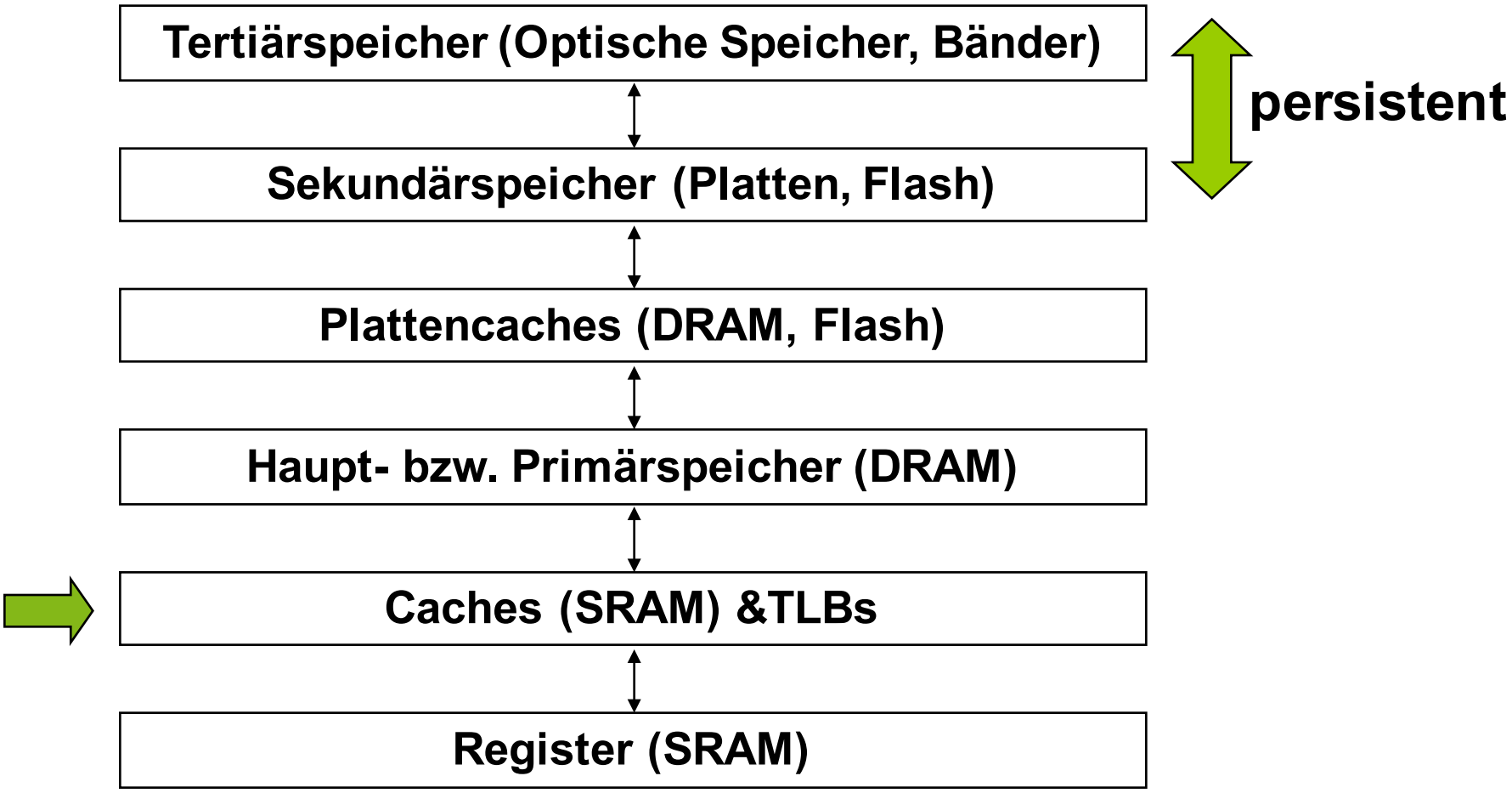
"…. We are … forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible"

Burks, Goldstine, von Neumann
Preliminary Discussion of the Logical Design of an Electronic Computing Instrument (1946)

Größenordnungen von Kapazitäten und Zugriffszeiten (2007)

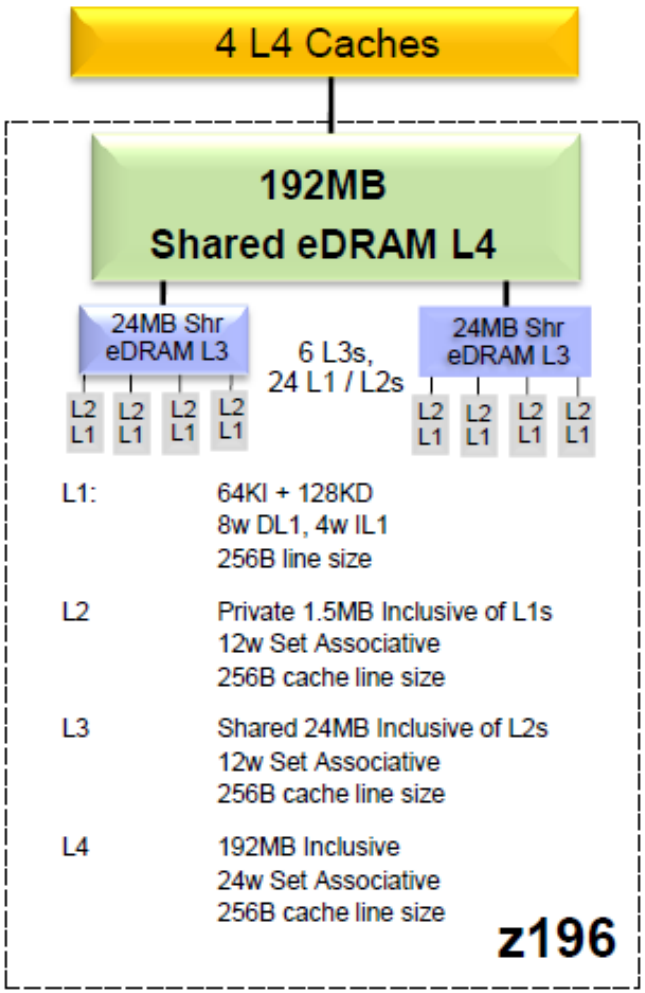


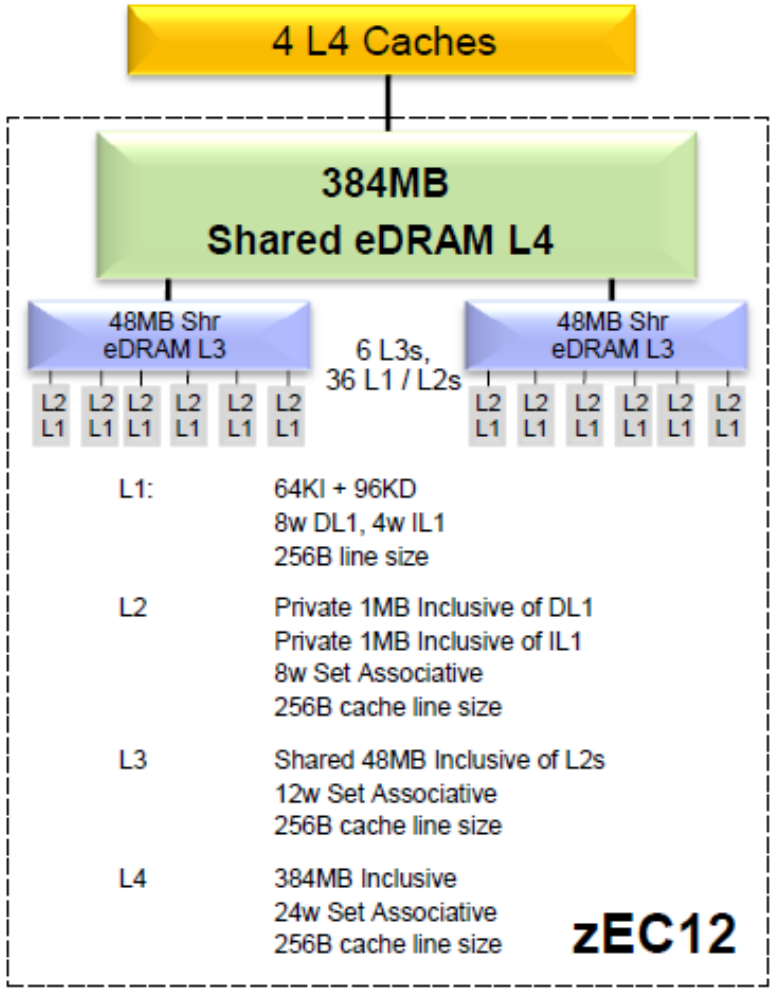
Mögliche Stufen der Speicherhierarchie
und derzeit eingesetzte Technologien



Beispiel: IBM Großrechner

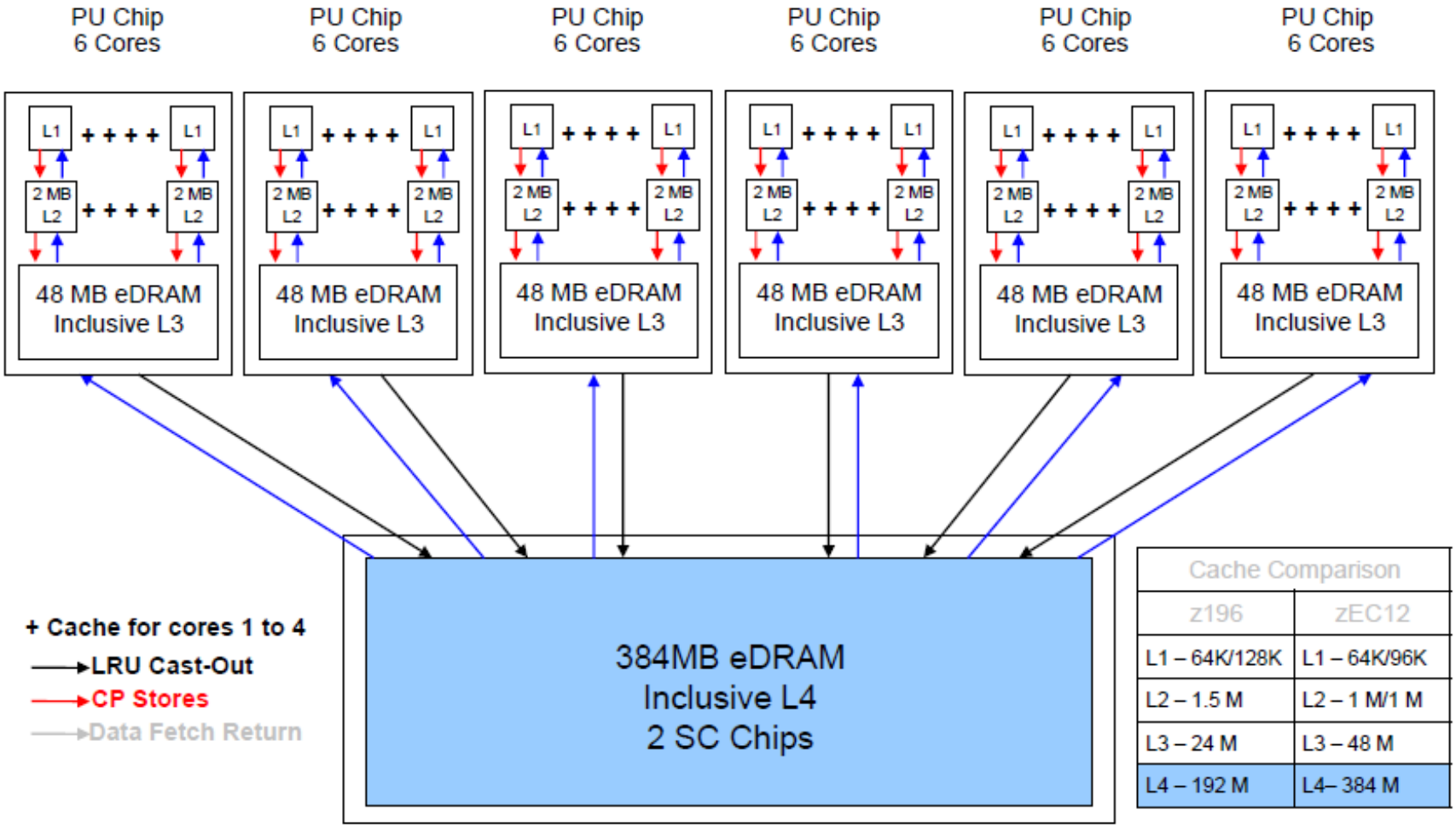
System z Cache Topology – z196 vs. zEC12 Comparison





Beispiel: IBM Großrechner

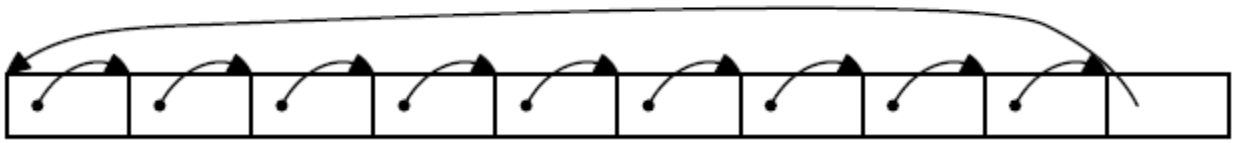
zEC12 Book Level Cache Hierarchy



Cache Comparison	
z196	zEC12
L1 – 64K/128K	L1 – 64K/96K
L2 – 1.5 M	L2 – 1 M/1 M
L3 – 24 M	L3 – 48 M
L4 – 192 M	L4– 384 M

Wie wirkt sich der Cache auf die Laufzeit von Programmen aus?

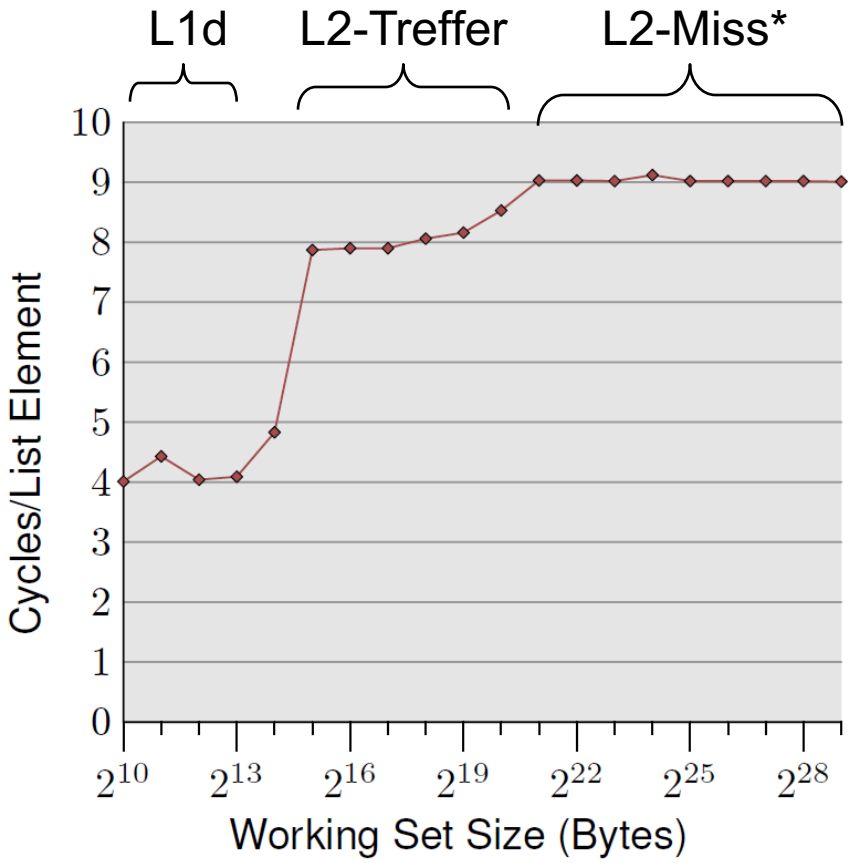
- Untersuchung der Laufzeit beim Durchlauf durch eine im Array abgelegte verkettete Liste mit Einträgen von (NPAD+1)*8 By



- Pentium P4
- 16 kB L1 Daten-Cache mit 4 Zyklen/Zugriff
- 1 MB L2 Cache mit 14 Zyklen/Zugriff
- Hauptspeicher mit 200 Zyklen/Zugriff

U. Drepper: *What every programmer should know about memory**, 2007, http://www.akkadia.org/drepper/cpumemory.pdf; Dank an Prof. Teubner (LS6) für Hinweis auf diese Quelle
* In Anlehnung an das Papier „David Goldberg, *What every programmer should know about floating point arithmetic, ACM Computing Surveys*, 1991 (auch für diesen Kurs benutzt).

Zyklen/Zugriff als Funktion der Listengröße



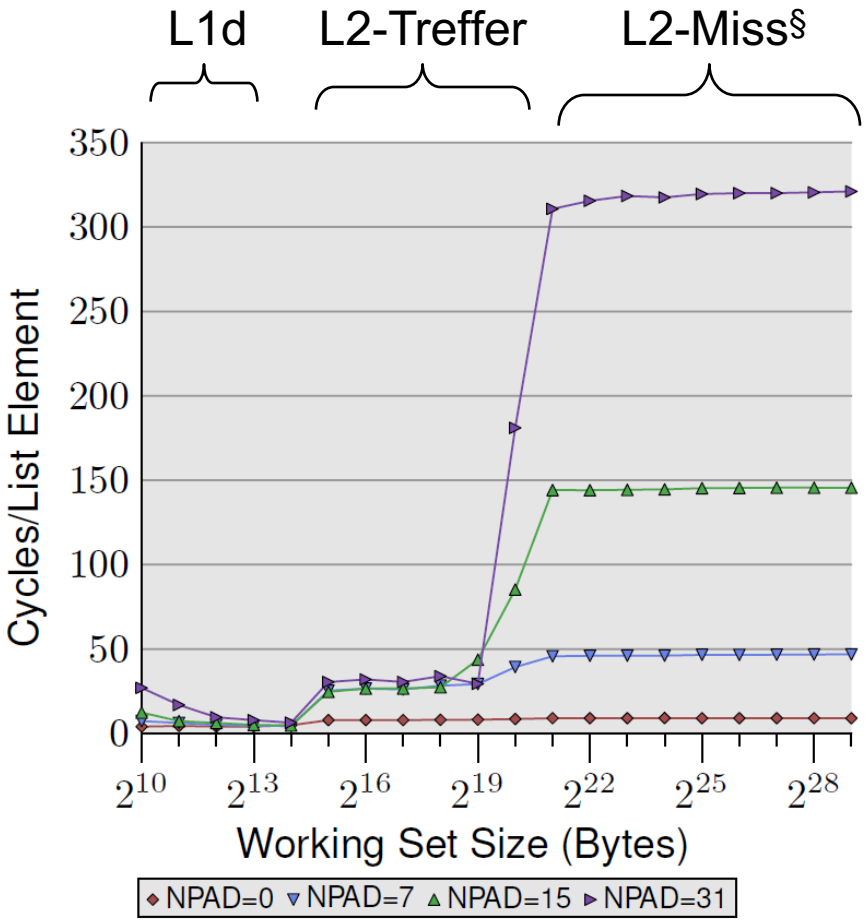


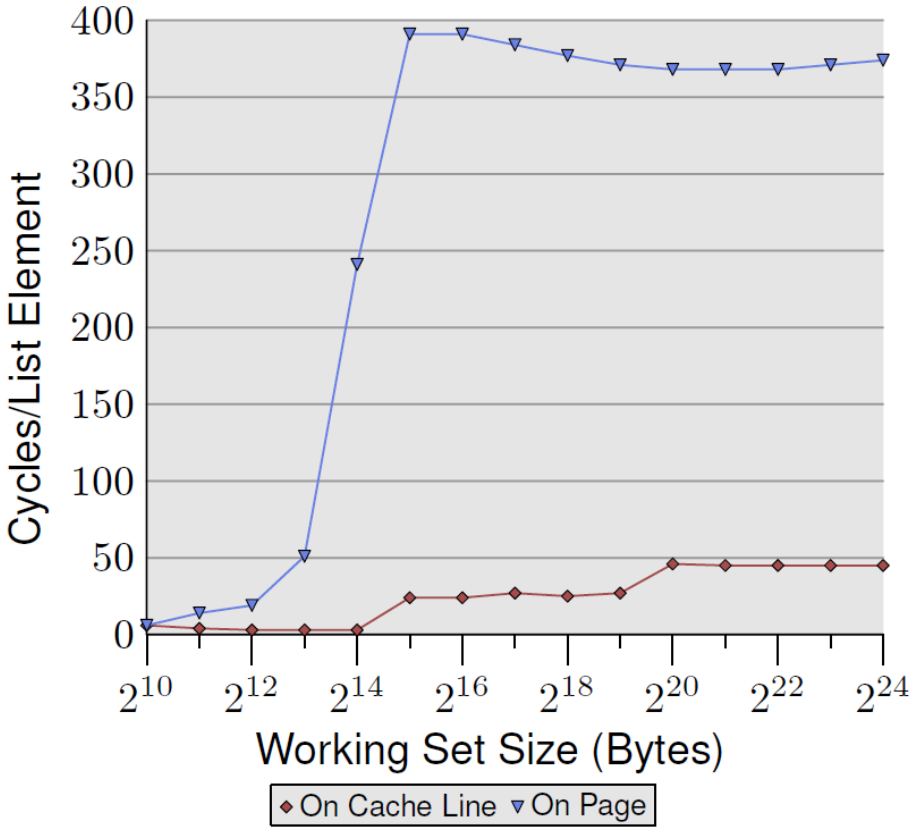
Figure 3.10: Sequential Read Access, NPAD=0

* Funktionierendes Prefetching

§ Prefetching schlägt fehl

Einfluss von TLB-Misses und größeren Caches

Elemente auf verschiedenen Seiten; Anwachsen der Laufzeit, wenn TLB-Größe überschritten wird



Größere Caches verschieben die Stufen

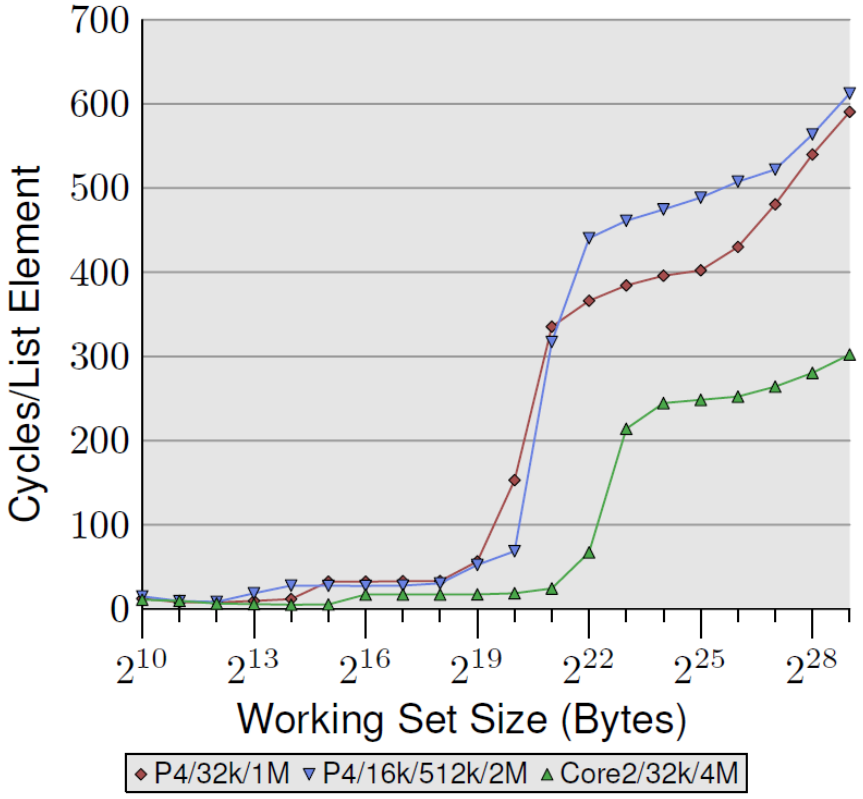


Figure 3.14: Advantage of Larger L2/L3 Caches

Caches

- Daten im *Cache* i.d.R. in größeren Einheiten organisiert
  ☞ *Cache*-Block (oder *cache line*)
- Bei erstem Zugriff (temporale Lokalität) in *Cache* geladen
- Örtliche Lokalität: Auch andere Daten im *Cache*-Block mit großer Wahrscheinlichkeit bald gebraucht
- Zugriff bei *cache miss* bestimmt durch
  - Latenzzeit des Speichers
    (Dauer des Zugriffs auf 1. Element im Cache Block) und
  - Bandbreite (Dauer des Transfers der weiteren Daten)
- Bei virtuellem Speicher weitere Hierarchieebene

Vier Fragen für Caches

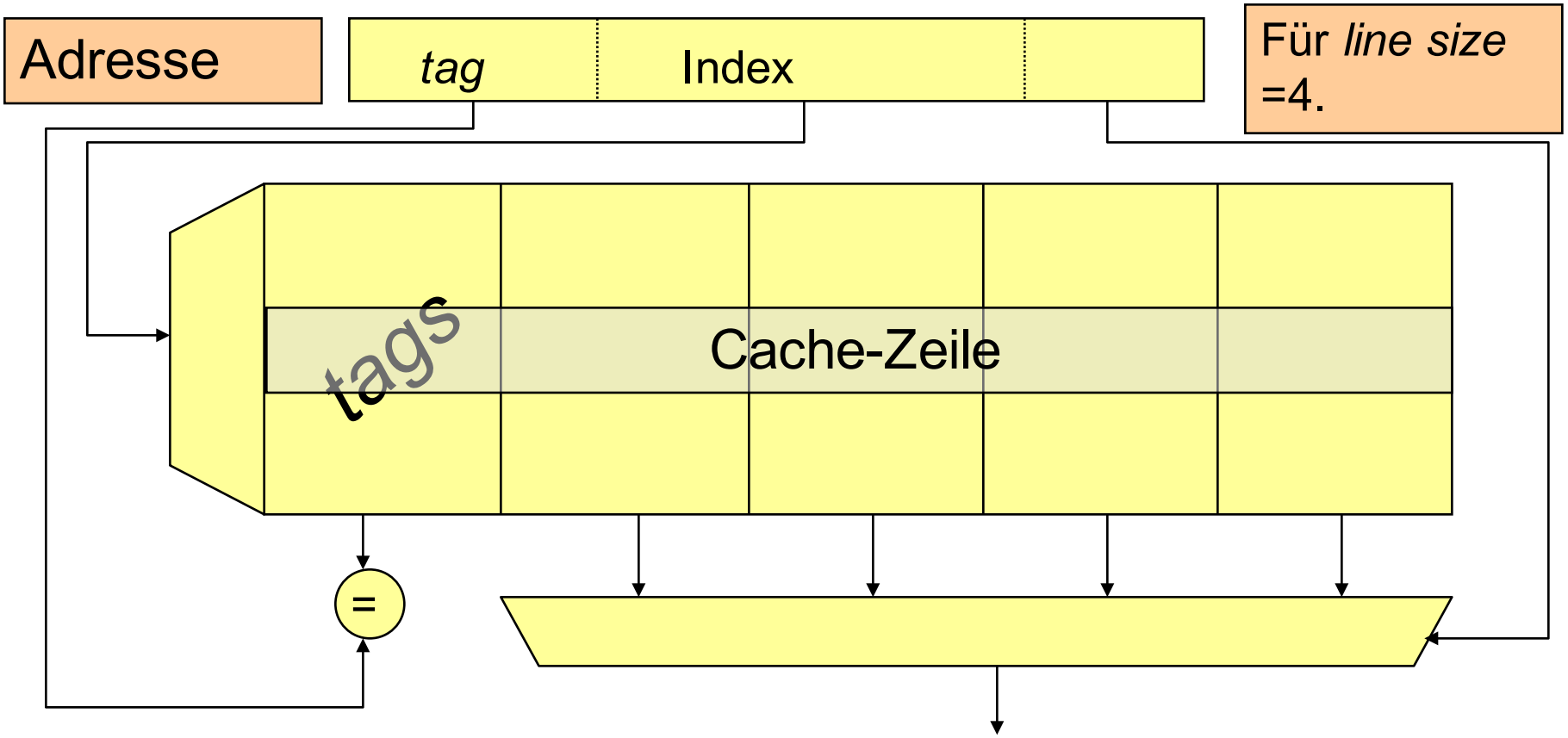
1. Wo kann ein Speicherblock im Cache abgelegt werden (*block placement*)
2. Wie wird ein Speicherblock gefunden (*block identification*)
3. Welcher Block sollte bei einem Fehlzugriff (*cache miss*) ersetzt werden? (*block replacement*)
4. Was passiert beim Schreiben von Daten in den Cache? (*write strategy*)

1. *Block Placement*

- ***Direct mapping***
  Für *caching* von Befehlen besonders sinnvoll, weil bei Befehlen *aliasing* sehr unwahrscheinlich ist.
- ***Set associative mapping***
  Sehr häufige Organisationsform, mit Set-Größe=2 oder 4, selten 8.
- ***Associative mapping***
  Wegen der Größe eines Caches kommt diese Organisationsform kaum in Frage.

2. *Block Identification* bei *direct mapping*

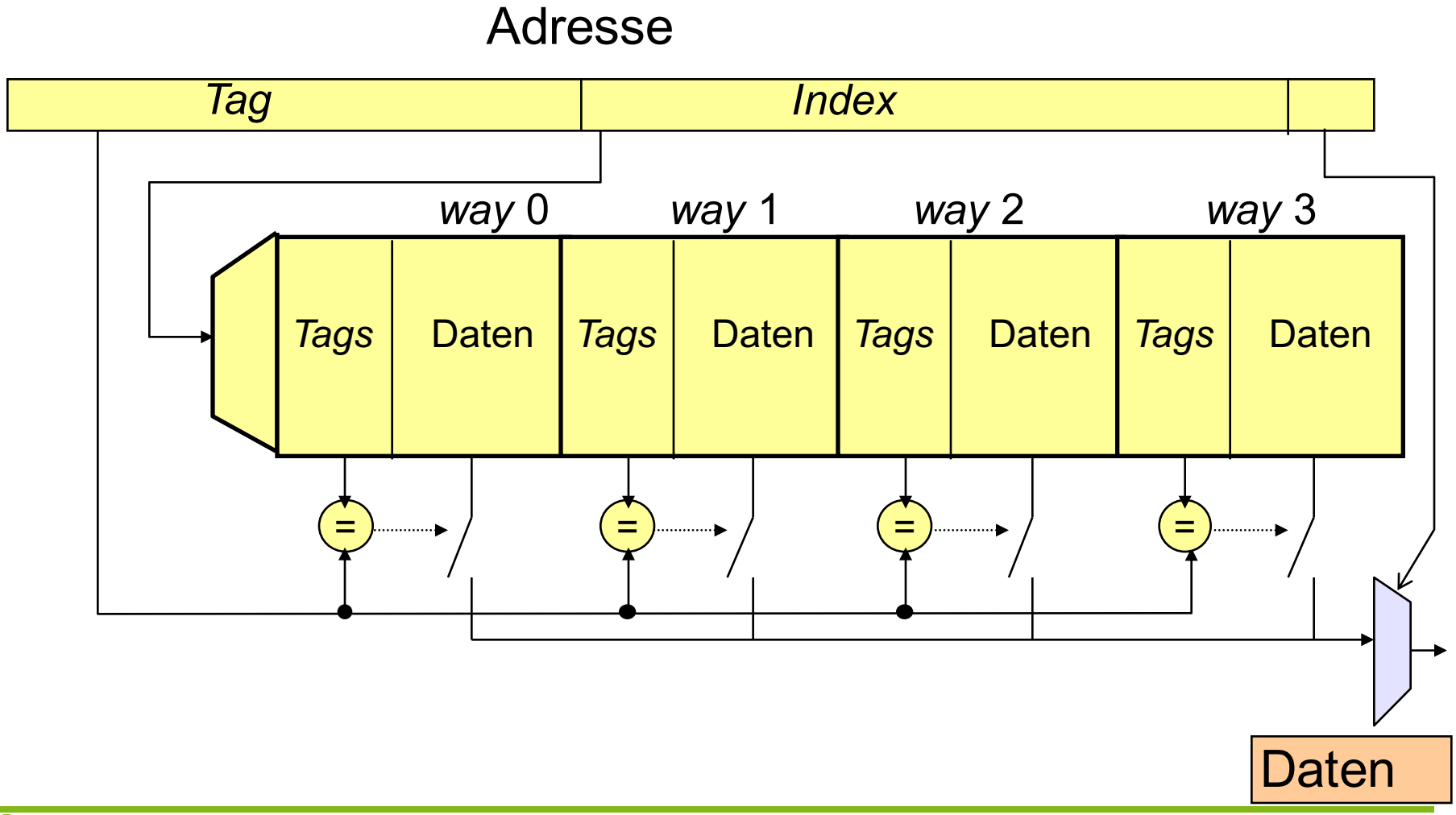
Such-Einheit im Cache: ***Cache*-Zeile** (***cache line***).



Weniger *tag bits*, als wenn man jedem Wort *tag bits* zuordnen würde.

2. *Block Identification* beim *Set-associative cache*
n-way cache, *block size = line size*

|Set| = 4



3. Ersetzung von Cache-Blöcken

Bei fehlgeschlagenem Cache-Zugriff (*cache miss*) muss Cache-Block ausgewählt und Inhalt mit angeforderten Daten ersetzt werden

Bei *direct mapped*: trivial

- Nur ein Block auf *hit* geprüft
- Nur dieser kann ersetzt werden
- ☞ extrem einfache / schnelle Hardware-Realisierung

Bei voll/n-Wege assoziativem Cache:

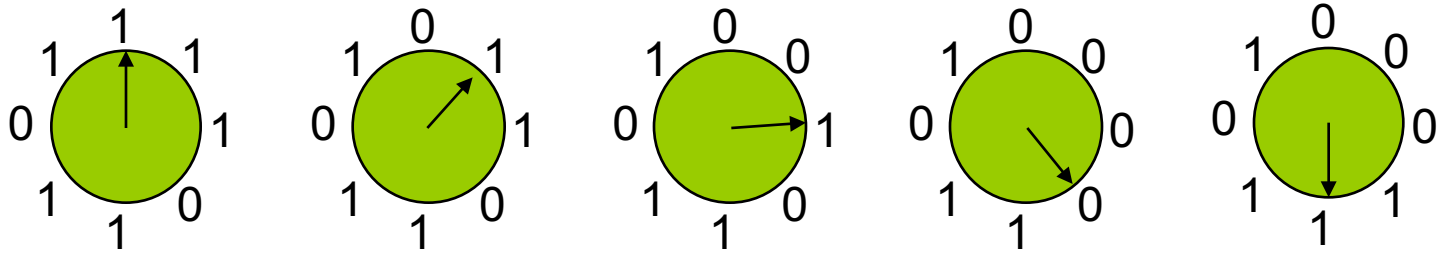
- Viele mögliche Blöcke für Ersetzung zur Auswahl
- Welcher Block soll ersetzt (d.h. entfernt) werden?
- Gefahr, dass diese Daten bald wieder gebraucht werden

3: Ersetzungsstrategien für Böcke

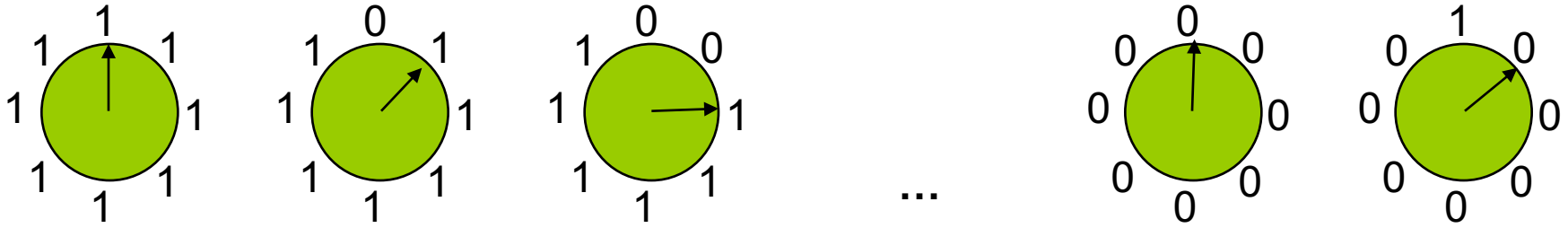
- **Zufällig (*random*)**
  - Ziel: Belegung des Caches möglichst uniform verteilen
  - Ggf. werden Pseudozufallszahlen(folgen) verwendet für reproduzierbares Verhalten (*debugging*!)
  - Für Realzeitsysteme eine Katastrophe ☹
- ***Least-recently used* (LRU)**
  - Der am längsten nicht benutzte Block wird ersetzt
  - Realisierung erfordert einen gewissen Aufwand
  - Für Realzeitsysteme die beste Lösung ☺
- ***First in, first out* (FIFO)**
  - Ältester Block wird ersetzt (auch wenn gerade benutzt)
- ***Clock***
  - Suche nach Block mit *Used-bit*='0' gleichzeitigem Rücksetzen des Bits. Stop nach max. 1 Runde.

Clock

Annahme: Manche Einträge unbenutzt, Platz benötigt



Annahme: Alle Einträge benutzt, Platz benötigt



Realisierung von LRU mittels verketteter Liste

Datenstruktur: verkettete Liste von Einträgen, sortiert nach der Reihenfolge des Zugriffs.

Operationen:

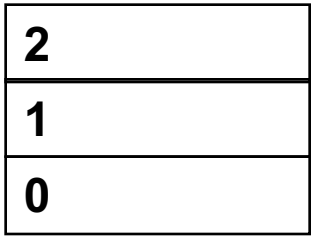
- Zugriff auf Eintrag i:
  - i wird am Anfang der Liste eingefügt und,
  - falls i sonst noch in der Liste vorkommt, dort entfernt.
- Verdrängen eines Eintrags:
  - Der Eintrag, der zum letzten Element der Liste gehört, wird überschrieben.

Beispiel: Dynamischer Ablauf

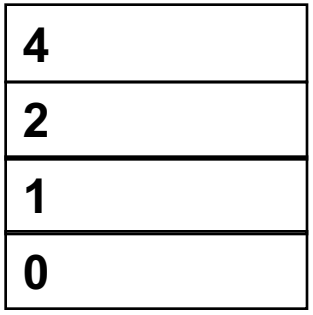
Zugriff auf *tag*=0

Zugriff auf *tag*=1

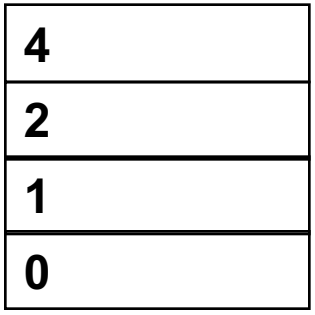
Zugriff auf *tag*=2

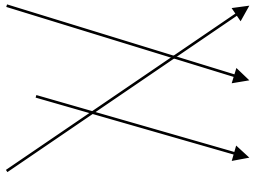


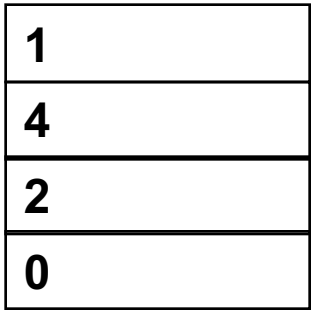
Zugriff auf *tag*=4



Zugriff auf *tag*=4



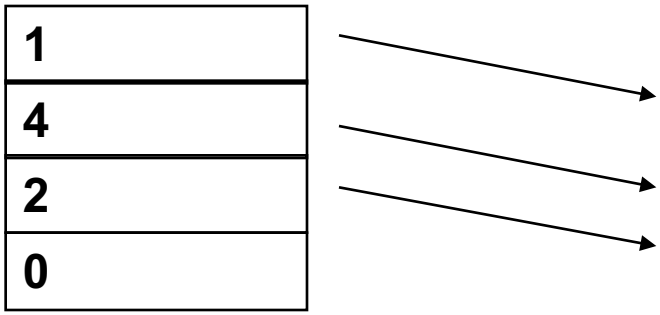




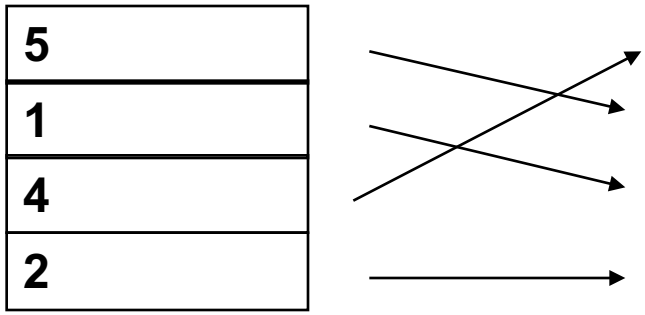
Zugriff auf *tag*=1

Beispiel: Dynamischer Ablauf (2)

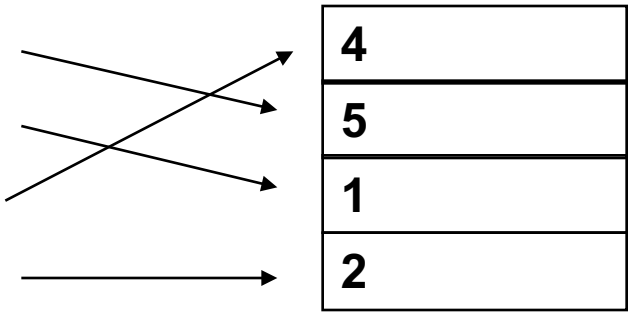
lt. letzter Folie



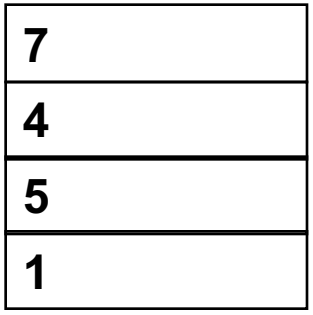
Zugriff auf *tag*=5



Zugriff auf *tag*=4

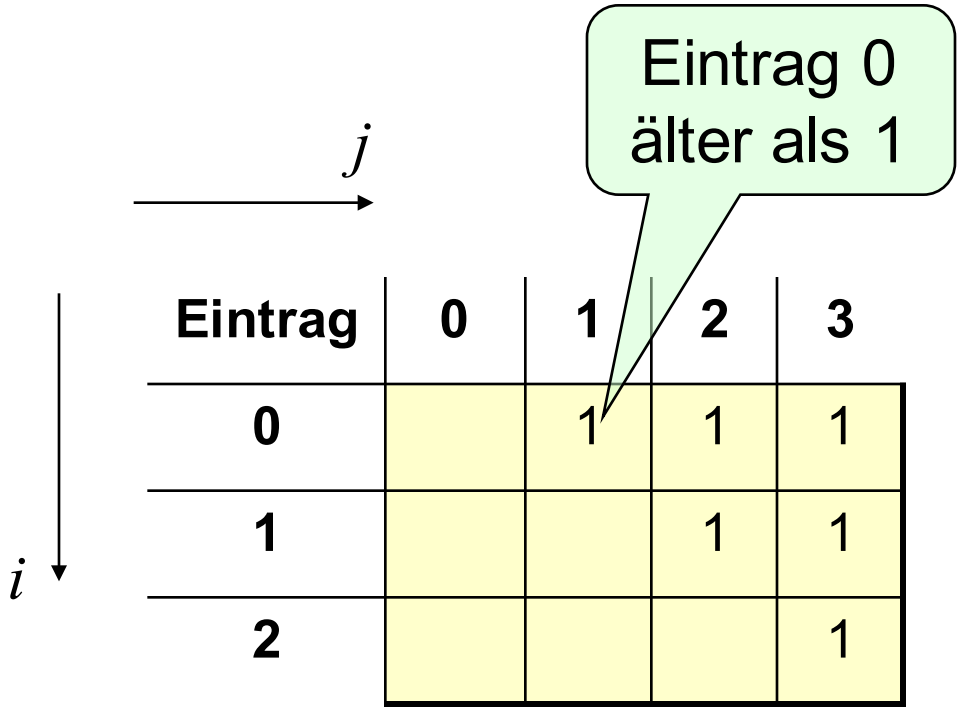


Zugriff auf *tag*=7



Realisierung von LRU mittels Dreiecksmatrix (1)

Def: $f[i, j] = 1$, falls Zugriff auf i älter ist als der auf j und 0 sonst.



Wegen Antisymmetrie nur Dreiecksmatrix $f[i, j]$, $i < j$ (d.h. Zeilenindex < Spaltenindex) zu speichern.

Realisierung von LRU mittels Dreiecksmatrix (2)

Operationen:

1. Zugriff auf k: k wird jüngster Eintrag:
   $\forall j > k : f[k, j] = 0; \forall i < k : f[i, k] = 1$
2. *Miss* im *Cache*: gesucht wird ältester Eintrag, d.h. Eintrag k, für den gilt: $\forall j > k : f[k, j] = 1 \wedge \forall i < k : f[i, k] = 0$

Annahme: Initialisierung mit 1

Eintrag	0	1	2	3
0		1	1	1
1			1	1
2				1

Cache miss ☞ Überschreiben von *way* 0 →

Eintrag	0	1	2	3
0		0	0	0
1			1	1
2				1

Cache miss ☞ Überschreiben von *way* 1 →

Jüngster Eintrag jeweils gestrichelt

Eintrag	0	1	2	3
0		1	0	0
1			0	0
2				1

Realisierung von LRU mittels Dreiecksmatrix (3)

Cache miss ☞ Überschreiben von *way* 2

Ein-trag	0	1	2	3
0		1	0	0
1			0	0
2				1

→

Cache miss ☞ Überschreiben von *way* 3

Ein-trag	0	1	2	3
0		1	1	0
1			1	0
2				0

→

Ein-trag	0	1	2	3
0		1	1	1
1			1	1
2				1

Cache miss ☞ Überschreiben von *way* 0

→

Ein-trag	0	1	2	3
0		0	0	0
1			1	1
2				1

Hit on way 2 ☞ way 2 wird jüngster →

Ein-trag	0	1	2	3
0		0	1	0
1			1	1
2				0

Realisierung von LRU mittels Dreiecksmatrix (4)

Hit on way 0 ☞ way
0 wird jüngster →

Ein-trag	0	1	2	3
0		0	1	0
1			1	1
2				0

Hit on way 3 ☞ way
3 wird jüngster →

Ein-trag	0	1	2	3
0		0	0	0
1			1	1
2				0

Ein-trag	0	1	2	3
0		0	0	1
1			1	1
2				1

Write allocate vs. *No-write allocate*

2 Varianten hinsichtlich des Schreibens:

Write allocate

- Block wird bei fehlgeschlagenem Schreibzugriff im Cache alloziert (entspricht Verhalten bei *read miss*)

No-write allocate

- Block bei *write miss* nur auf nächster Hierarchiestufe aktualisiert

Beliebig kombinierbar mit *write back/through*, i.d.R. aber:

- *Write back* + *write allocate*
- *Write through* + *no-write allocate*

Zusammenfassung

- Speicherhierarchien
  - Motivation
  - Caches
    - Durchschnittliche Laufzeiten
    1. Wo kann Speicherblock abgelegt werden (*block placement*)
    2. Wie wird ein Speicherblock gefunden (*block identification*)?
    3. Welcher Block sollte bei einem Fehlzugriff (*cache miss*) ersetzt werden? (*block replacement*)
       - Clock
       - LRU
    4. Was passiert beim Schreiben von Daten? (*write strategy*)