

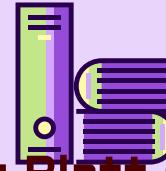
Herzlich willkommen  
zum fünften ERA  
Tutorium des  
Semesters!





## Zusammenfassung

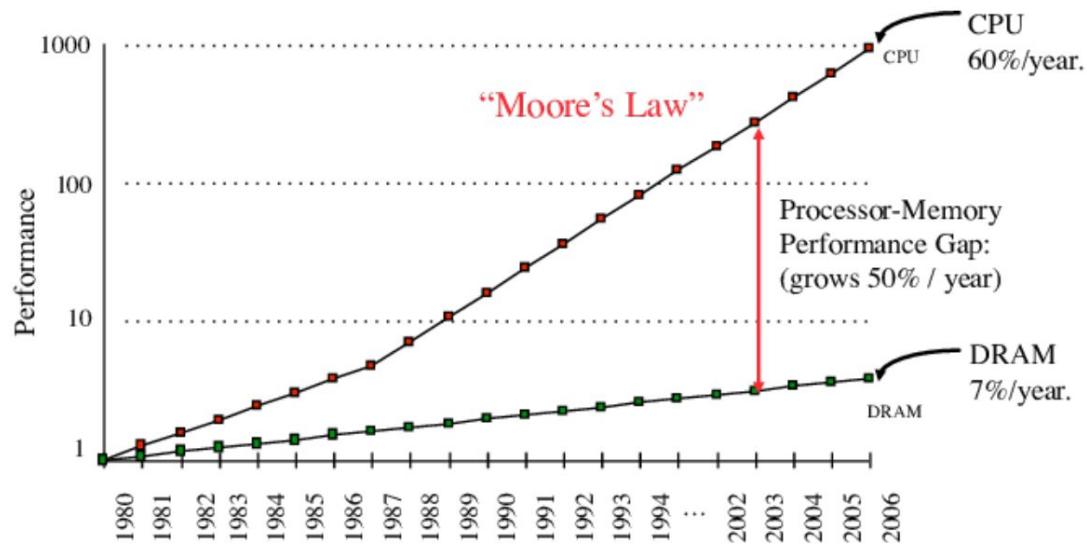
Dies ist kein Ersatz für die VL!  
Keine Garantie auf Korrektheit.  
Nur eine Erinnerung an die  
wichtigsten Themen der Woche.



## Das Blatt gem. lösen

Es ist nicht nötig, das Blatt  
vorher zu lösen. Du  
bekommst für jede Übung  
etwas Zeit, bevor wir sie  
gemeinsam lösen.

# Trend: Speicher vs. CPU Leistung



## Lokalitätsprinzip

```
int n;  
n = read();  
if (n > 0) { return n; }  
if (n < 0) { return n * (-1); }
```

### Zeitliche Lokalität von Speicherzugriffen (temporal locality)

- Wenn ein Programm auf Adresse x zugreift, ist es wahrscheinlich, dass es kurz darauf wieder auf x zugreifen wird
- Caches behalten auf kurzem zugegriffene Daten im Speicher

### Räumliche Lokalität von Speicherzugriffen (spatial locality)

- Wenn ein Programm auf Adresse x zugreift, ist es wahrscheinlich, dass es auch auf Adressen in der Nähe von x zugreift
- Caches behalten benachbarte Daten im Speicher

```
for (int i=0; i < 10; i++) {  
    arr[i] = 100  
}
```

Aufteilung einer Adresse:



Offset: untere Bits der Adresse als Index in die Zeile

Index: bestimmt die Cachezeile

Tag: beschreibt die Daten die abgelegt werden

Arten von Caches:

- Direct mapped: Index beschreibt eindeutige Zeile
- Fully associative: Keine Index-bits, jede Cachezeile möglich
- N-way associative: Index beschreibt Menge aus N Zeilen  
Konsequenz:  $\log_2(N)$  Index Bits

# Assoziative Caches

## Direct Mapped Caches

- Jede Adresse kann auf genau eine Cache Zeile abgebildet werden
- Größte Flexibilität für Hardware, Kleinste für ProgrammiererIn, aber auch geringste Komplexität

## Voll-assoziativer Cache („Fully Associative Cache“)

- Jede Adresse kann auf jede Cache Zeile abgebildet werden
- Nur ein „Cache Set“, keine Index bits
- Größte Flexibilität für ProgrammiererIn, aber auch größte Komplexität (z.B. Anzahl von Vergleichern)

## Kompromiss: Mengenassoziativer Cache („Set-Associative Cache“)

- Jede Adresse kann auf einen Teil der Cache Zeilen abgebildet werden
- Mehrere Cache-Sets / Reduzierte Zahl von Index bits
- Anzahl der Zeilen pro Cache Set = Assoziativität des Caches
  - Beispiel: 4-way associative = 4 Zeilen pro Cacheset
- Mehr Flexibilität
- Handhabbare Komplexität (z.B. Anzahl von Vergleichern)

# Klassifikation von „Cache Misses“

## Compulsory (cold) Misses

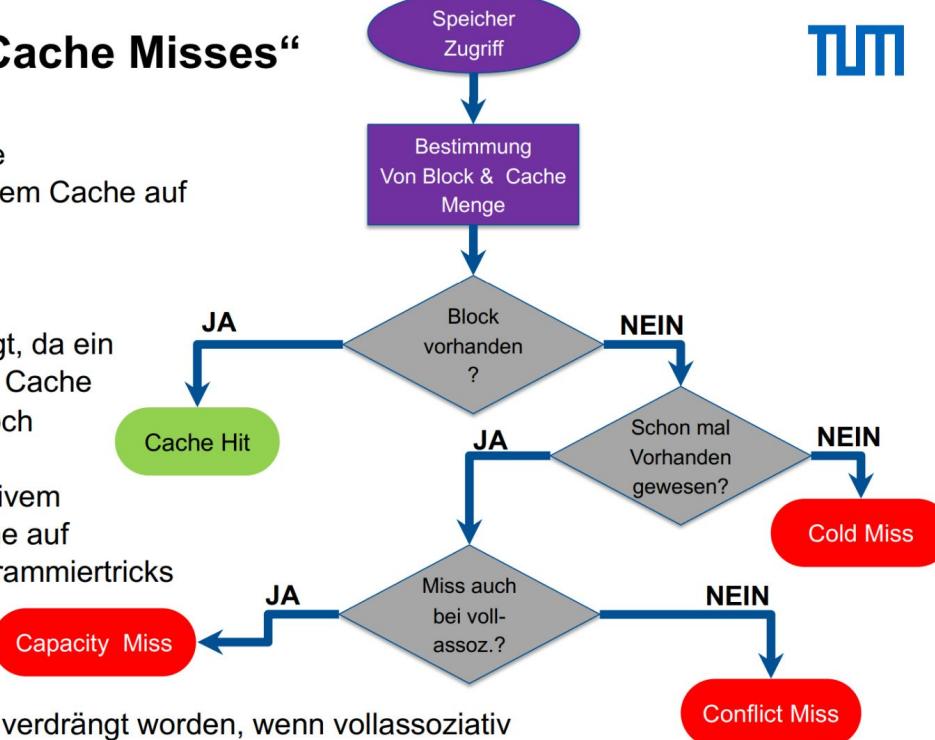
- Erster Zugriff auf eine Adresse
- Treten auch in unendlich großem Cache auf
- Nicht zu vermeiden

## Conflict Misses

- Speicherblock wurde verdrängt, da ein anderer Block auf das gleiche Cache Set abgebildet wurde, aber noch Raum frei war
- Treten nur in mengenassoziativem oder direkt abbildendem Cache auf
- Kann (manchmal) durch Programmiertricks umgangen werden

## Capacity Misses

- Der Speicherblock wäre auch verdrängt worden, wenn vollassoziativ
- Nur zu umgehen durch andere Zugriffsmuster

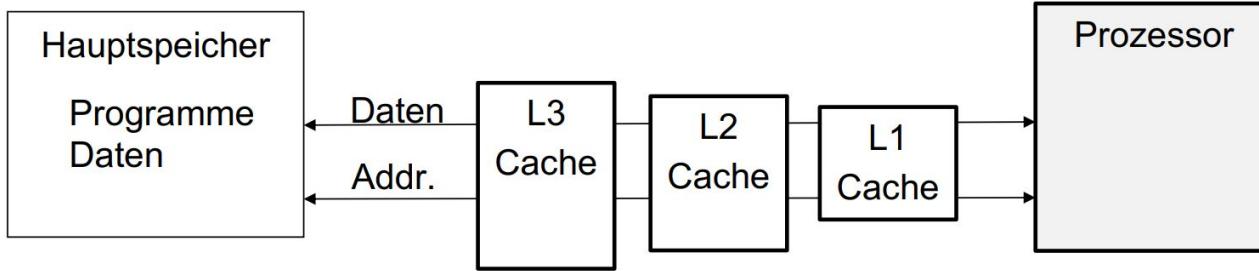


## **Platzierungs- und Ersetzungsstrategie**

- In welche Cache-Zeile wird eine Speicherblock eingetragen?
- Welcher Speicherblock wird dafür (falls nötig) aus dem Cache entfernt?

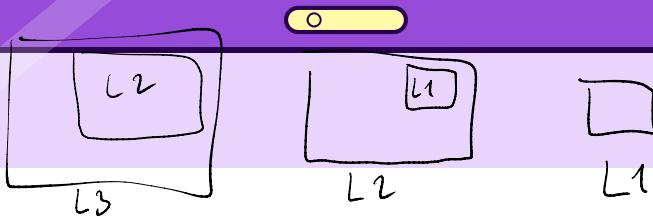
Oft: LRU (Least Recently Used) oder LFU (Least Frequently Used)

- Gut für temporale Lokalität
- Praxis: oft kombiniert für vereinfachter Hardware



Je kleiner ein Cache, desto schneller

- Aber kleine Caches haben wenig Platz
- Heute üblich: 3 stufige Caches (z.T. auch schon mehr)

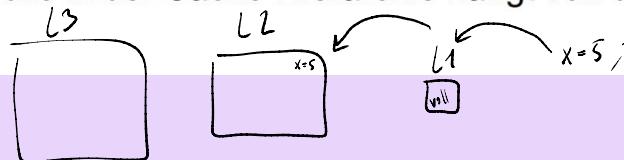


### Inklusiv

- Alle in einem kleineren Cache enthaltenen Speicherblöcke sind auch im darunterliegenden größeren Cache enthalten
- Bei Zugriff auf Hauptspeicher werden Kopien in allen Ebenen abgelegt
- Anzahl der Speicherblöcke, die in der gesamten Cache-Hierarchie vorhanden sein können, hängt von der Größe des größten Caches ab

### Exklusiv

- Jeder Speicherblock ist höchstens einmal als Kopie in der Cachehierarchie enthalten
- Hauptspeicherzugriff lädt in L1, von wo Kopien in L2 verdrängt werden, falls L1 voll. Ebenfalls Verdrängung von L2 in L3
- Anzahl der Speicherblöcke in der Cache-Hierarchie hängt von der Summe der Cachegrößen in der Hierarchie ab



Gegeben sei ein Programm mit  $3 \cdot 10^9$  Instruktionen, das Sie auf einem in-order execution Prozessor mit einer Frequenz von 1.0 GHz ausführen. Dieses Programm läuft auf diesem Prozessor mit durchschnittlich 2 Zyklen pro Instruktion (CPI), exklusive Speicherzugriffe. Jede Instruktion generiere dabei durchschnittlich 1.33 Speicherzugriffe.

- a) Wie groß ist die durchschnittliche Speicherzugriffszeit (Average Memory Access Time), wenn das Programm eine Ausführungszeit von 66 Sekunden hat?

$$\text{CPU Time} = \text{IC} \cdot \left( \frac{\text{CPI}}{f} + \frac{\text{Memory Accesses}}{\text{Instruction}} \cdot \text{Average Memory Access Time} \right)$$

*← Messung der Cache-Gate*

↑  
cycles/s

$$66s = 3 \cdot 10^9 \text{ ins} \cdot \left( \overbrace{\frac{2 \text{ cycles/ins}}{10^9 \text{ cycles/s}}}^{\text{CPI}} + 1,33 \cdot \text{AMAT} \right)$$

Wir beginnen mit der bekannten Formel zur Berechnung der Ausführungszeit:

$$\text{CPU Time} = \text{IC} \cdot \left( \frac{\text{CPI}}{f} + \frac{\text{Memory Accesses}}{\text{Instruction}} \cdot \text{Average Memory Access Time} \right)$$

Hier alle Werte einsetzen:

$$66 \text{ s} = 3 \cdot 10^9 \text{ ins} \cdot \left( \frac{2 \frac{\text{cycles}}{\text{ins}}}{1 \text{ GHz}} + 1.33 \frac{\text{accesses}}{\text{ins}} \cdot \text{Average Memory Access Time} \right)$$

Die Frequenz hat hier nicht wie üblich die Einheit  $\frac{1}{\text{s}}$ , sondern  $\frac{\text{cycles}}{\text{s}}$ :

$$66 \text{ s} = 3 \cdot 10^9 \text{ ins} \cdot \left( \frac{2 \frac{\text{cycles}}{\text{ins}}}{1 \cdot 10^9 \frac{\text{cycles}}{\text{s}}} + 1.33 \frac{\text{accesses}}{\text{ins}} \cdot \text{Average Memory Access Time} \right)$$

Von hier gelingt die Berechnung der durchschnittlichen Speicherzugriffszeit durch einfache Umformungen:

$$66 \text{ s} = 6 \text{ s} + 3 \cdot 10^9 \text{ ins} \cdot 1.33 \frac{\text{accesses}}{\text{ins}} \cdot \text{Average Memory Access Time}$$

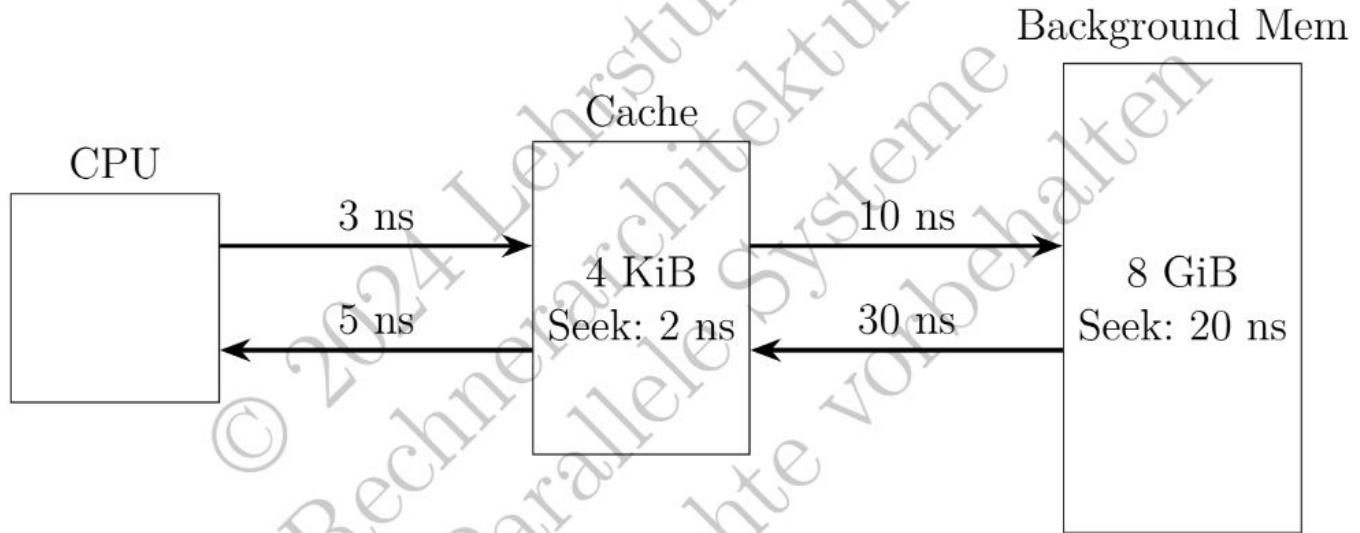
Man beachte, dass von den 66 Sekunden Ausführungszeit gerade einmal 6 Sekunden für die tatsächliche Berechnung verwendet wird. Die restliche Zeit geht durch Speicherzugriffe verloren.

$$\frac{66 \text{ s} - 6 \text{ s}}{3 \cdot 10^9 \text{ ins} \cdot 1.33 \frac{\text{accesses}}{\text{ins}}} = \text{Average Memory Access Time}$$

$$\frac{60 \text{ s}}{4 \cdot 10^9 \text{ accesses}} = \text{Average Memory Access Time}$$

$$15 \cdot 10^{-9} \frac{\text{s}}{\text{access}} = 15 \frac{\text{ns}}{\text{access}} = \text{Average Memory Access Time}$$

Der Speicher eines Computers sei wie folgt aufgebaut:



- b) Was ist die Hit-Latency, Miss Penalty, Miss-Latency und die Miss-Rate?



Die Hit-Latency ist die Antwortzeit des Caches auf einen Speicherzugriff. In diesem Fall besteht diese aus Zeit für die Anfrage (3 ns), Seek-time (2 ns) und Ladezeit (5 ns).

$$\text{Hit Latency} = 3 \text{ ns} + 2 \text{ ns} + 5 \text{ ns} = 10 \text{ ns}$$

Die Miss-Penalty ist einfach die zusätzliche Antwortzeit, die benötigt wird wenn ein Cache-Miss eintritt. Auch hier wieder Anfragezeit (10 ns), Seek-time (20 ns) und Ladezeit (30 ns).

$$\text{Miss Penalty} = 10 \text{ ns} + 20 \text{ ns} + 30 \text{ ns} = 60 \text{ ns}$$

Damit kann nun die Miss-Latency berechnet werden:



$$\text{Miss Latency} = \text{Hit Latency} + \text{Miss Penalty} = 10 \text{ ns} + 60 \text{ ns} = 70 \text{ ns}$$

Um die Miss-Rate zu berechnen muss man nur noch die durchschnittliche Speicherzugriffszeit einsetzen:

$$\text{Average memory access time} = \text{Hit Rate} \cdot \text{Hit Latency} + \text{Miss Rate} \cdot \text{Miss Latency}$$

$$\text{Average memory access time} = (1 - \text{Miss Rate}) \cdot 10 \text{ ns} + \text{Miss Rate} \cdot 70 \text{ ns}$$

$$15 \text{ ns} = (1 - \text{Miss Rate}) \cdot 10 \text{ ns} + \text{Miss Rate} \cdot 70 \text{ ns}$$

$$15 \text{ ns} = 10 \text{ ns} - 10 \text{ ns} \cdot \text{Miss Rate} + \text{Miss Rate} \cdot 70 \text{ ns}$$

$$5 \text{ ns} = 60 \text{ ns} \cdot \text{Miss Rate}$$

$$\frac{5}{60} = \frac{1}{12} \approx 8.3\% = \text{Miss Rate}$$

- c) Ein\*e Programmierer\*in optimiert das Programm um die Hit-Rate im Cache zu maximieren. Dadurch senkt sich die durchschnittliche Speicherzugriffszeit um  $\frac{1}{3} = 33.33\%$ . Dennoch erhöht sich die Ausführungszeit. Woran könnte das liegen?

$$\text{CPU Time} = \text{IC} \cdot \left( \frac{\text{CPI}}{f} + \frac{\text{Memory Accesses}}{\text{Instruction}} \cdot \text{Average Memory Access Time} \right)$$

$$\text{Average Memory Access Time} = \text{Hit Rate} \cdot \text{Hit Latency} + \text{Miss Rate} \cdot \text{Miss Latency}$$

Durch die Veränderungen am Programm könnten sich neben der reduzierten durchschnittlichen Speicherzugriffszeit auch die Anzahl an Instruktionen, die Anzahl an Zyklen/Instruktion, sowie die Anzahl an Speicherzugriffen/Instruktion verändert haben. Wenn sich diese zum Schlechten verändern, können sie die Zugewinne durch die reduzierte durchschnittliche Speicherzugriffszeit auslöschen.

$$\frac{IC_1}{IC_2} = \frac{\frac{CPU\ Time}{\left(\frac{CPI}{f} + \frac{\text{Mem Acc}}{\text{Ins}} \cdot \text{AMAT}_1\right)}}{\frac{CPU\ Time}{\left(\frac{CPI}{f} + \frac{\text{Mem Acc}}{\text{Ins}} \cdot \text{AMAT}_2\right)}}$$

- d) Angenommen neben der um  $\frac{1}{3}$  reduzierten durchschnittlichen Speicherzugriffszeit verändert sich durch die Optimierung nur die Anzahl an Instruktionen. Wie sehr muss sich die Anzahl an Instruktionen relativ verändern, damit sich die Laufzeit nicht verbessert?

Ansatz: Stellen Sie Formel für CPU Time nach IC um und setzen Sie einmal die reguläre Average Memory Access Time und dann die verkürzte Average Memory Access Time ein. Um die relative Abweichung zu berechnen, teilen Sie dann die beiden Instruction Counts durcheinander.

$$CPU\ Time = IC \cdot \left( \frac{CPI}{f} + \frac{\text{Memory\ Accesses}}{\text{Instruction}} \cdot \text{Average\ Memory\ Access\ Time} \right)$$

$$\text{Average\ Memory\ Access\ Time} = \text{Hit\ Rate} \cdot \text{Hit\ Latency} + \text{Miss\ Rate} \cdot \text{Miss\ Latency}$$

Da sich nur die Average Memory Access Time und der Instruction Count verändert, kann man das Verhältnis zwischen neuem und der altem Instruction Count wie folgt notieren:

$$\frac{IC_2}{IC_1} = \frac{\frac{\text{CPU Time}}{\left( \frac{\text{CPI}}{f} + \frac{\text{Memory Accesses}}{\text{Instruction}} \cdot \text{Average Memory Access Time}_2 \right)}}{\frac{\text{CPU Time}}{\left( \frac{\text{CPI}}{f} + \frac{\text{Memory Accesses}}{\text{Instruction}} \cdot \text{Average Memory Access Time}_1 \right)}}$$

Durch Kürzen erhält man:

$$\frac{IC_2}{IC_1} = \frac{\left( \frac{\text{CPI}}{f} + \frac{\text{Memory Accesses}}{\text{Instruction}} \cdot \text{Average Memory Access Time}_1 \right)}{\left( \frac{\text{CPI}}{f} + \frac{\text{Memory Accesses}}{\text{Instruction}} \cdot \text{Average Memory Access Time}_2 \right)}$$

Durch Einsetzen der Zahlen:

$$\frac{IC_2}{IC_1} = \frac{\left( \frac{2 \frac{\text{cycles}}{\text{ins}}}{10^9 \frac{\text{cycles}}{\text{s}}} + 1.33 \frac{\text{accesses}}{\text{ins}} \cdot 15 \frac{\text{ns}}{\text{access}} \right)}{\left( \frac{2 \frac{\text{cycles}}{\text{ins}}}{10^9 \frac{\text{cycles}}{\text{s}}} + 1.33 \frac{\text{accesses}}{\text{ins}} \cdot \left( 15 + \frac{1}{3} \cdot 15 \right) \frac{\text{ns}}{\text{access}} \right)}$$

Durch Umformung somit:

$$\frac{IC_2}{IC_1} = \frac{\left(2 \cdot 10^{-9} \frac{s}{ins} + 1.33 \frac{accesses}{ins} \cdot 15 \cdot 10^{-9} \frac{s}{access}\right)}{\left(2 \cdot 10^{-9} \frac{s}{ins} + 1.33 \frac{accesses}{ins} \cdot 10 \cdot 10^{-9} \frac{s}{access}\right)}$$

Kürzen durch den Faktor  $10^{-9}$ :

$$\frac{IC_2}{IC_1} = \frac{\left(2 \frac{s}{ins} + 1.33 \cdot 15 \frac{s}{ins}\right)}{\left(2 \frac{s}{ins} + 1.33 \cdot 10 \frac{s}{ins}\right)}$$

Das Verhältnis von altem und neuem Instruction Count ist einheitenlos:

$$\frac{IC_2}{IC_1} = \frac{\left(2 \frac{s}{ins} + 1.33 \cdot 15 \frac{s}{ins}\right)}{\left(2 \frac{s}{ins} + 1.33 \cdot 10 \frac{s}{ins}\right)} \approx 143\%$$

Die Anzahl der Instruktionen muss sich also um circa 43% erhöhen, um eine Senkung der Average Memory Access Time um  $\frac{1}{3}$  auszugleichen.

- e) Was sind Gründe, weshalb sich die Laufzeiten eines Programms heutzutage nicht mehr so einfach wie oben berechnen lassen?

Es gibt diverse Gründe, wieso Laufzeiten nicht mehr vorhersehbar sind. Hier eine Auswahl:

- Moderne Prozessoren sind in der Regel out-of-order Prozessoren. Diese können Instruktionen umsortieren, damit Speicherzugriffe sich überschneiden können. Damit kann die Anzahl an Wartezyklen (memory stall cycles) teils deutlich reduziert werden.
- Auf Multikernrechnern gibt es in der Regel mehrere Cache-Ebenen, die privat oder geteilt sein können und über Coherence-Protokolle verfügen. Hit/Miss-rates sowie die Latenzen können somit von anderen Prozessorkernen beeinflusst werden.
- Heutzutage laufen Programme oft nicht mehr alleine auf einem System. Interferenzen durch das Betriebssystem oder andere Programme können starke Auswirkungen auf den Cache und die Laufzeit haben.

Gegeben sei eine 16-Bit Architektur mit byte-adressiebarem Speicher. Zudem ist ein 256 Byte großer Cache mit 2 Byte pro Cachezeile gegeben. Vergleichen sie in den folgenden Teilaufgaben direct mapped Caches, 4-fach assoziative Caches und voll-assoziative Caches, indem sie die Aufgaben jeweils für die 3 unterschiedlichen Arten beantworten.

- In wie viele unterschiedliche Cachezeilen kann ein Byte aus dem Speicher potentiell getragen werden.

- Voll-assoziativ: So viele Cachezeilen im Cache. Insgesamt gibt es  $256/2 = 128$  Cachezeilen
- 4-fach assoziativ: 4
- Direct mapped: 1

- b) Es wird auf das Byte an der Adresse 0xABCD zugegriffen. Wie viele unterschiedliche Cachezeilen müssen überprüft werden um festzustellen ob das Datum an der Adresse bereits gecached ist? Geben Sie zudem an welcher Vergleich stattfindet um festzustellen ob der Wert bereits gecached ist?

$0xABCD = 0b\ 1010\ 1011\ 1100\ 1101$

$0xAB$

D → 1 Bit Offset, 128 "Cachesets" → 7 Bits Index → 8 Bit Tag

V → 1 Bit Offset → 15 Bit Tag :  $0b\ 101\ 0101\ 110\ 0110$   
 $= 0x55E6$

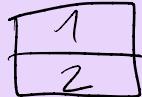
L → 1 Bit Offset,  $128/4 = 32$  Sets →  $\log_2 32 = 5$  Bits Index

10 Bit Tag =  $0b\ 10\ 010\ 1111-02AF$

- Voll-assoziativ: Jede Cachzeile muss auf den Tag überprüft werden. Bei Voll-assoziativen Caches haben wir keinen Index. Für den Offset benötigen wir 1 Bit. Somit sind die restlichen 15 Bit Tag. Somit muss jede Cachzeile mit dem Tag  $0b101.0101.1110.0110 = 0x5E6$  verglichen werden.
- 4-fach assoziativ: 1 Bit für Offset,  $128/4 = 32$  Cachesets. Somit benötigen wir 5 Bit für den Index und haben  $16 - 1 - 5 = 10$  Bit Tag. Somit ist der Tag  $0b10.1010.1111 = 0x2AF$ .
- Direct mapped: 1 Bit Offset; 128 Cachesets. Somit benötigen wir 8 Bit für den Index und haben  $16 - 1 - 8 = 7$  Bit Tag. Somit ist der Tag  $0b101.0101 = 0x55$

Für die folgenden Fragen wird ein System betrachtet, das nur einen Rechenkern mit einem Cache besitzt. Dieser Cache sei 2-fach assoziativ, habe 4 Cachezeilenmengen (Cache Sets) und eine Cachezeilenlänge von 32 Byte. Die Architektur sei eine reine 32-Bit Architektur und alle Zugriffe 32 Bit breit. Der Speicher sei wie üblich Byte-adressierbar.

- a) Wie viele Cachezeilen hat der Cache?



8 Cachezeilen	
0	Set 0
1	
2	Set 1
3	
4	Set 2
5	
6	Set 3
7	

$$2 \cdot 4 = 8$$

Index

00

01

[0

] )

$$\text{Tag} = \text{Adresse} - \text{Index} - \text{Offset}$$

Für die folgenden Fragen wird ein System betrachtet, das nur einen Rechenkern mit einem Cache besitzt. Dieser Cache sei 2-fach assoziativ, habe 4 Cachezeilenmengen (Cache Sets) und eine Cachezeilenlänge von 32 Byte. Die Architektur sei eine reine 32-Bit Architektur und alle Zugriffe 32 Bit breit. Der Speicher sei wie üblich Byte-adressierbar.

b) Wie viele Bits benötigt man für Offset, Index und Tag?

$$\text{Offset} = \log_2 32 = 5 \text{ Bit}$$

$$\text{Index} = \log_2 4 = 2 \text{ Bit}$$

$$\text{Tag} = 32 - 5 - 2 = 25 \text{ Bit}$$

$$\left\lceil \frac{5}{2} \right\rceil = 3 \quad \left\lfloor \frac{5}{2} \right\rfloor = 2$$

**Offset Bits** – bezeichnen das Byte der Cachezeile, auf das zugegriffen werden soll.

Die Cachelinegröße ist in unserem Beispiel 32 Bytes, deswegen braucht man  $\lceil \log_2 32 \rceil = 5$  Bits an Offset.

**Index Bits** – bestimmen die Cachezeile, in der sich die gewählte Adresse befinden kann.

In unserem Beispiel gibt es 4 Sets.

Daher benötigt man hier  $\lceil \log_2 4 \rceil = 2$  Bits.

**Tag Bits** – der Rest:  $32 - 5 - 2 = 25$  Bits. Werden bei Cache Request mit der Adresse abgeglichen.

c) Gegeben sei folgender Ausschnitt eines C-Programms:

```
int data[1000];
```

```
int sum = 0;
```

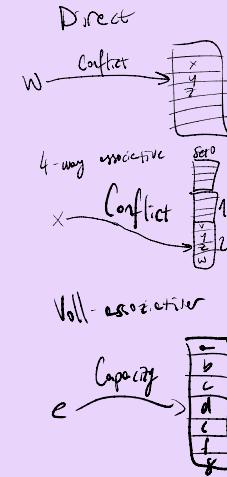
```
// ABSCHNITT 1
```

```
for (int i = 0; i < 1000; i += 64) {  
    sum = sum + data[i];  
}
```

```
// ABSCHNITT 2
```

```
for (int i = 0; i < 1000; i += 32) {  
    sum = sum + data[i];  
}
```

WV viele Zugriffe?



Nehmen Sie an, dass nur das Array im Speicher liegt und alle anderen Werte in Registern gehalten werden. Der Cache sei zu Beginn leer. Gehen Sie davon aus, dass ein int 32-Bit groß ist und stets vollständig in einer Cachezeile liegt.

i) Wie viele „Capacity Misses“ sind in den beiden Abschnitten zu erwarten?

Jeder **int** ist 4 Byte groß und es liegt ein Stride von 32 bzw. 64 **ints** vor. Damit erhöht sich die Speicheradresse bei jedem Zugriff um  $4 \cdot 32 = 128$  bzw. 256. Ausgehend von der Speicheradresse 0 (die tatsächliche Startadresse spielt hier keine Rolle), kann man die ersten Iterationen in Abschnitt 1 berechnen:

Iteration	Tag	Index	Offset
1	0000 0000 0	00	0 0000
2	0000 0001 0	00	0 0000
3	0000 0010 0	00	0 0000
4	0000 0011 0	00	0 0000
...	...	...	...

Es ist festzustellen, sich lediglich der Tag-Bereich der Speicheradresse verändert. Dementsprechend wird immer auf dieselbe Cachezeilenmenge zugegriffen. Diese enthält zwei Cachezeilen, die in den ersten beiden Iterationen ohne andere Daten zu verdrängen aufgefüllt werden. Anschließend wird in jeder Iteration ein Datum verdrängt.

Deshalb #Iterationen<sub>Abschnitt 1</sub> =  $\left\lceil \frac{1000}{64} \right\rceil = 16$  und  
#Iterationen<sub>Abschnitt 2</sub> =  $\left\lceil \frac{1000}{32} \right\rceil = 32.$

- i) Im ersten Abschnitt treten keine Capacity Misses auf, da auf alle Datenblöcke das erste Mal zugegriffen wird.

Im zweiten Abschnitt tritt bei jedem Zugriff auf ein Element, auf das im ersten Abschnitt schon zugegriffen wurde, ein Capacity Miss auf, da diese auch in einem vollassoziativen Cache verdrängt worden wären. D.h. es treten 16 Capacity Misses auf.



c) Gegeben sei folgender Ausschnitt eines C-Programms:

```
int data[1000];
int sum = 0;

// ABSCHNITT 1
for (int i = 0; i < 1000; i += 64) {
    sum = sum + data[i];
}

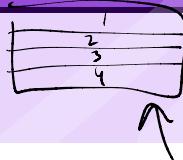
// ABSCHNITT 2
for (int i = 0; i < 1000; i += 32) {
    sum = sum + data[i];
```

ii) Wie viele „Cold Misses“ sind zu erwarten?

Ansatz: Betrachten Sie die Adressen der einzelnen Array-Elemente.

- ii) Im ersten Abschnitt tritt bei jedem Zugriff ein Cold Miss auf. D.h.  $\#\text{Cold Misses}_{\text{Abschnitt } 1} = \#\text{Iterationen}_{\text{Abschnitt } 1} = 16$

Im zweiten Abschnitt hingegen sind tritt ein Cold Miss nur dann auf, wenn die jeweiligen Daten nicht im ersten Abschnitt schon benutzt wurden. Dies trifft auf die Indizes  $0, 32, 64, 96, \dots, 960, 992$  zu. Also treten auch zweiten Abschnitt 16 Cold Misses auf.



ein Set

## Hilfreich für die HA!

Gegeben sei ein 4-fach assoziativer Cache mit 16 Cachezeilen. Pro Cachezeile werden 4 Byte abgespeichert. In den folgenden Tabellen ist jeweils das Cache Set 0 angegeben. Cachen sie die folgenden Werte jeweils einmal mit LFU und LRU als Ersetzungsstrategie und füllen sie in den Tabellen aus. Als Vereinfachung wird von jeder Adresse nur der Tag angegeben und Index und Offset werden in dieser Aufgabe ignoriert.

- Cachen sie die Werte mit LRU als Ersetzungsstrategie

*Beispiel: In Zeitschritt 1 wird die Adresse mit dem Tag 12 gecached. Da das Datum an der Adresse noch nicht im Cache vorhanden ist handelt es sich um ein Miss. Der Wert wird in Cachezeile 1 und der letzte Access in Last Use von Cachezeile 1 gespeichert.*

LRU

Time	Tag	Line 0		Line 1		Line 2		Line 3		Hit/Miss
		Tag	LU	Tag	LU	Tag	LU	Tag	LU	
1	12	12	1	-	-	-	-	-	-	Miss
2	20	12	1	20	2	-	-	-	-	Miss
3	12	12	3	20	2	-	-	-	-	Hit
4	12	12	4	20	2	-	-	-	-	Hit
5	28	12	4	20	2	28	5	-	-	Miss
6	32	12	4	20	2	28	5	32	6	Miss
7	38	12	4	38	7	28	5	32	6	Miss
8	25	25	8	38	7	28	5	32	6	Miss
9	12	25	8	38	7	12	3	32	6	Miss
10	32	25	8	38	7	12	3	32	10	Hit

Time	Tag	Line 0		Line 1		Line 2		Line 3		Hit/Miss
		Tag	LU	Tag	LU	Tag	LU	Tag	LU	
1	12	12	1		-	-	-	-	-	Miss
2	20	12	1	20	2	-	-	-	-	Miss
3	12	12	3	20	2	-	-	-	-	Hit
4	12	12	4	20	2	-	-	-	-	Hit
5	28	12	3	20	2	28	5	-	-	Miss
6	32	12	3	20	2	28	5	32	6	Miss
7	38	12	3	38	7	28	5	32	6	Miss
8	25	25	8	38	7	28	5	32	6	Miss
9	12	25	8	38	7	12	9	32	6	Miss
10	32	25	8	38	7	12	9	32	10	Hit



- b) Cachen sie die Werte mit LFU als Ersetzungsstrategie

*Beispiel: In Zeitschritt 1 wird die Adresse mit dem Tag 12 gecached. Da das Datum an der Adresse noch nicht im Cache vorhanden ist handelt es sich um ein Miss. Der Wert wird in Cachezeile 1 gespeichert und der Counter auf 1 gesetzt.*

Time	Tag	Line 0		Line 1		Line 2		Line 3		Hit/Miss
		Tag	Count	Tag	Count	Tag	Count	Tag	Count	
1	12	12	1	-	-	-	-	-	-	Miss
2	20	12	1	20	1	-	-	-	-	Miss
3	12	12	2	20	1	-	-	-	-	Hit
4	12	12	3	20	1	-	-	-	-	Hit
5	28	12	3	20	1	28	1	-	-	Miss
6	32	12	3	20	1	28	1	32	1	Miss
7	38	12	3	38	1	28	1	32	1	Miss
8	25	12	3	25	1	28	1	32	1	Miss
9	12	12	4	25	1	28	1	32	1	Hit
10	32	12	4	25	1	28	1	32	2	Hit

Time	Tag	Line 0		Line 1		Line 2		Line 3		Hit/Miss
		Tag	Count	Tag	Count	Tag	Count	Tag	Count	
1	12	12	1	-	-	-	-	-	-	Miss
2	20	12	1	20	1	-	-	-	-	Miss
3	12	12	2	20	1	-	-	-	-	Hit
4	12	12	3	20	1	-	-	-	-	Hit
5	28	12	3	20	1	28	1	-	-	Miss
6	32	12	3	20	1	28	1	32	1	Miss
7	38	12	3	38	1	28	1	32	1	Miss
8	25	12	3	25	1	28	1	32	1	Miss
9	12	12	4	25	1	28	1	32	1	Hit
10	32	12	4	25	1	28	1	32	2	Hit



- c) Vergleichen Sie die Ersetzungsstrategien. Was sind Vor- und Nachteile der jeweiligen Strategien?

- LFU: Gut bei repetitiven Zugriffsmustern, relativ einfach zu implementieren; kann schlecht bei unregelmäßigen Zugriffsmustern sein
- LRU: Schwierig zu implementieren; schlecht bei zufälligen oder verstreuten Zugriffen