

# Application Performance Management

## Caching

Michael Faes

# Rückblick: Performance Testing

**Lasttest:** Analysieren der Leistung unter einer konstanten, definierten Last

**Dauerlasttest:** Untersuchen der Leistung über einen grösseren Zeitraum hinweg

**Failover-Test:** Lasttest bei (manuell verursachtem) Ausfall von System-Komponenten

**Stresstest:** Schrittweises Erhöhen der Last, bis System instabil wird oder ganz ausfällt

# Übersicht

1. Rückblick Performance Testing
2. Caching
  - Beispiele, Speicher-Hierarchie
  - Caching-Grundlagen
  - Cache-Algorithmen
3. Übung

# Nochmals Zeitmassstäbe

Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	3 ns	10 s
Level 3 cache access	10 ns	33 s
Main memory access (DRAM, from CPU)	100 ns	6 min
Solid-state disk I/O (flash memory)	10–100 µs	9–90 hours
Rotational disk I/O	1–10 ms	1–12 months



Quelle:  
Gregg (2020)

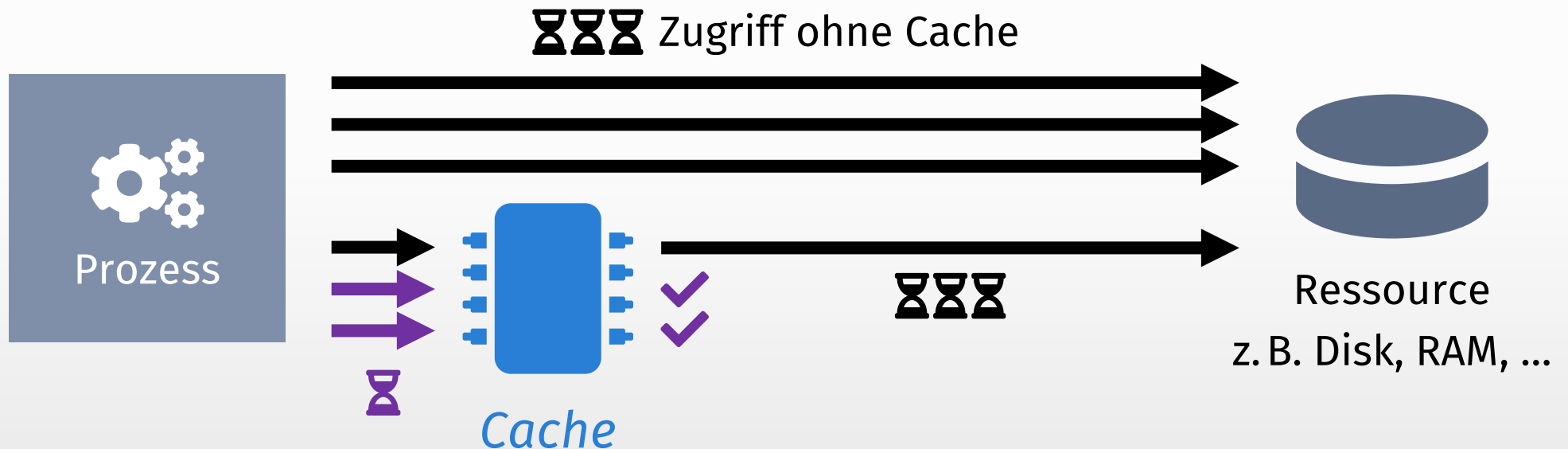
1. Zugriff auf Speicher dauert mehrere 100 Zyklen
2. Die meisten Instruktionen brauchen wohl Daten aus dem Speicher

*Müsste CPU sich nicht während 99% der Zeit im Leerlauf befinden??*

**Nein!** → L1–L3-Caches

# Was ist ein Cache?

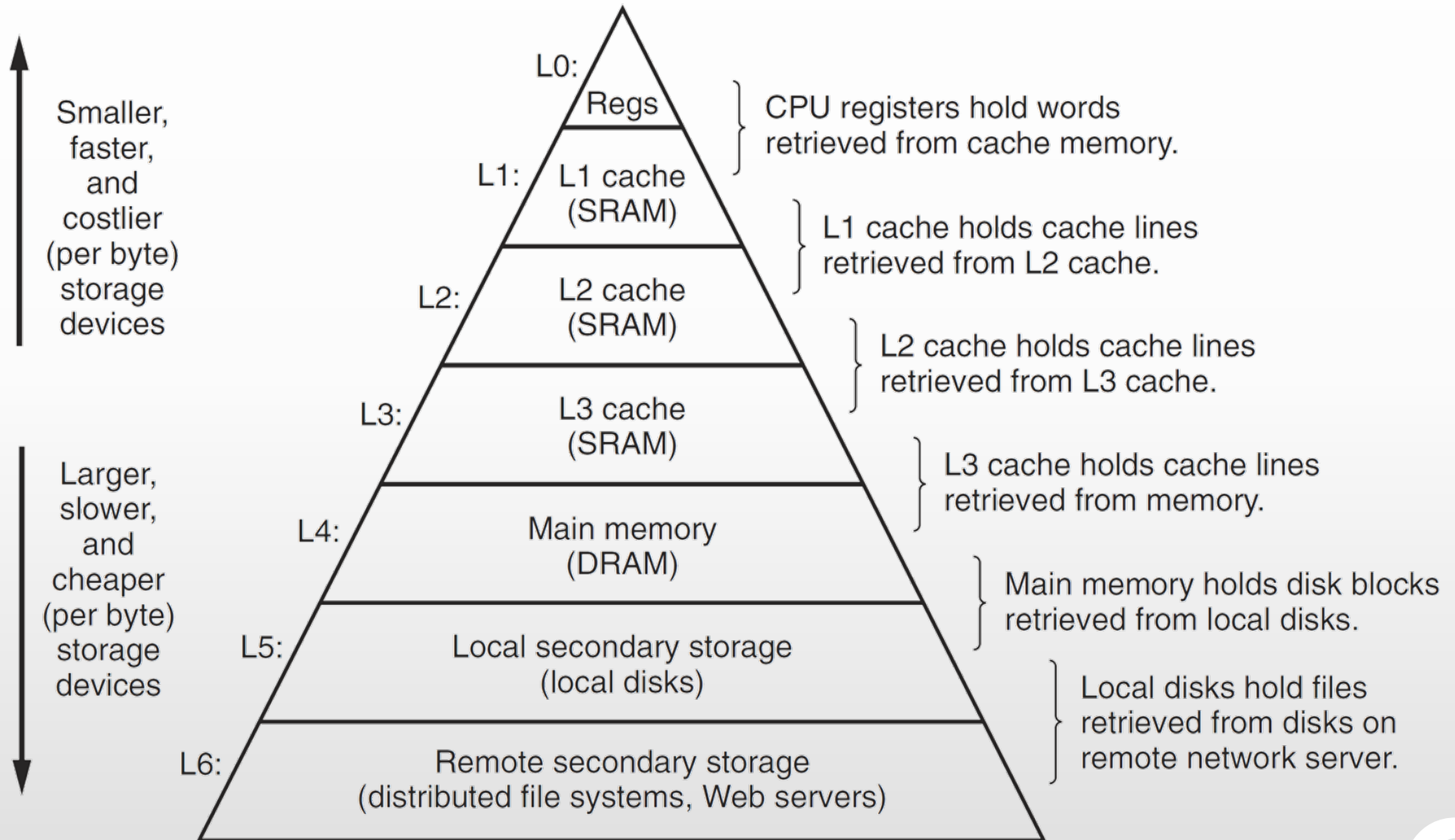
**Cache:** Schneller Speicher, der *wiederholte Zugriffe* auf ein langsame Ressource oder aufwändige Neuberechnungen zu vermeiden hilft.



Daten bleiben im Cache, damit sie bei Bedarf schneller abrufbar sind.

*Bringt nur etwas, wenn Cache deutlich schneller ist!*

# Speicher-Hierarchie



# Beispiele für Caches

## Computersysteme

- CPU-Caches (L1–L3)
- Filesystem-Cache
- (Paging)
- Festplatten mit SSD-Caches

## Applikationen

- *Memoization* (z. B. hashCode)
- Dynamic Programming
- App-spezifische Caches

Übung

## Web

- HTTP (in Browser oder Proxy)
- DNS
- Code Cache für JIT (Java)
- Web-Cache von Google

Übung

# Caching ist (manchmal) schwierig

*“There are only two hard problems in Computer Science: **cache invalidation** and naming things.”*

— Phil Karlton

*“There are two hard problems in Computer Science: cache invalidation, naming things, and off-by-1 errors.”*

*“There are two hard problems in Computer Science: we only have one joke and it's not funny.”*



# Caching kann sehr effektiv sein!

**Beispiel:** *Performance Problems You Can Fix: A Dynamic Analysis of Memoization Opportunities*

[dl.acm.org/citation.cfm?id=2814290](https://dl.acm.org/citation.cfm?id=2814290)

**Memoization:** Spezielle Form von Caching, Zwischenspeichern von berechneten Resultaten

Paper: Automatische Analyse findet Orte in Java-Code, wo Caches sinnvoll sein könnten. Resultate: Bis zu **12.9× Speedup**.

**Allgemein:** *Caching ist eine Technik, die praktisch **orthogonal** zu anderen Technologien und Optimierungen eingesetzt werden kann und oft sehr effektiv ist.*

# Caching-Grundlagen

# Caching-Begriffe

*Cache Access:* Lese-Zugriff auf Cache

*Cache Hit:* Element wurde in Cache gefunden

*Cache Miss:* Element wurde *nicht* in Cache gefunden

*Hit Ratio (Miss Ratio):* Anteil der Zugriffe, die zu Cache Hit (Miss) führen

- Hohe Hit Ratio: Cache ist «hot», sonst «cold»

*Füllstand:* Anteil des verwendeten Platzes im Cache

Ist ein Cache «gut», wenn er voll oder leer ist?

**Falsche Frage!** Leistung des Caches hängt von Hotness ab, nicht von Füllstand! (Wobei komplett leerer Cache natürlich nichts bringt.)

# Warum funktionieren Caches?

## Principle of Locality

- *Temporal locality*: Wenn ein Objekt verwendet wird, wird es *mit grosser Wahrscheinlichkeit bald* wieder verwendet.
- *Spatial locality* (data locality): Ebenso «benachbarte» Objekte (...)

**Konkret:** *Zipf's Law* (80:20-Regel):  
80% aller Zugriffe gehen auf  
die 20% selben Objekte  
(oder auch 90:10, usw.)

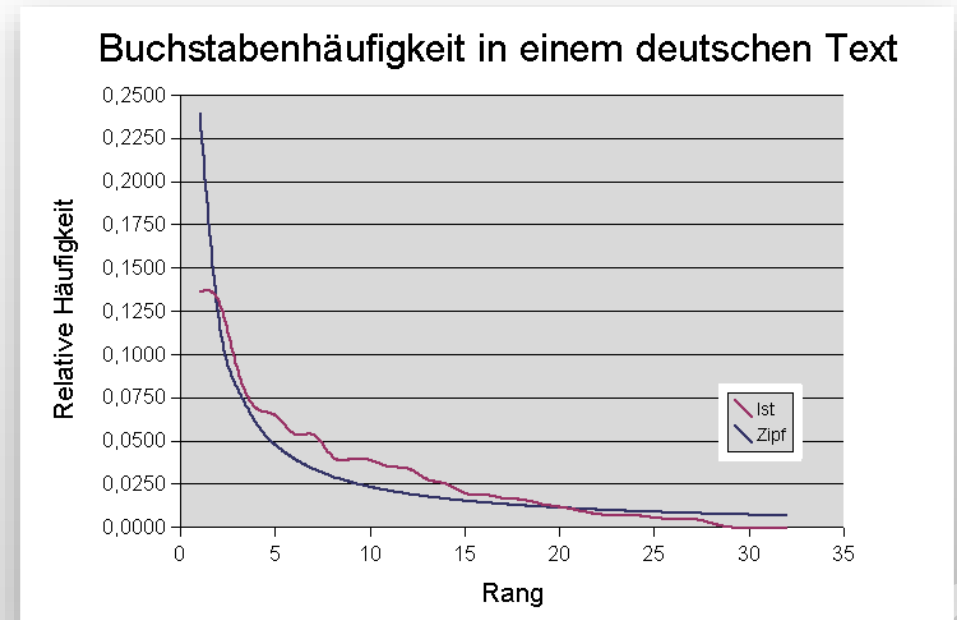


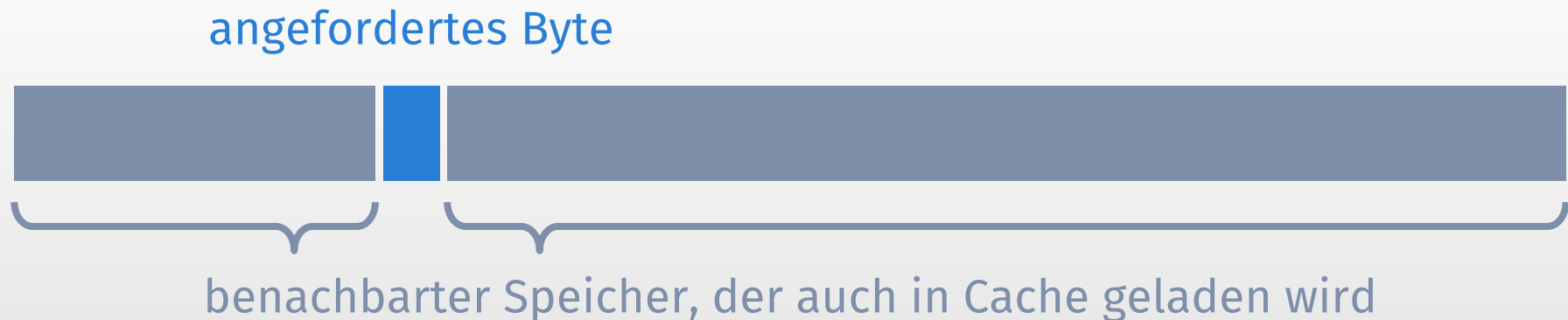
Bild: «Anton» (deutsche Wikipedia)

# Spatial Locality & Cache Lines

CPU-Caches speichern keine einzelnen Bits oder Bytes, sondern grössere Speicher-Stücke: *Cache Lines*.

- Grösse typischerweise 64 oder 128 Bytes

Wenn CPU ein Byte vom Speicher anfragt, wird ganze Cache Line geladen:



Grund: *Spatial Locality*. Daten, die kurz nacheinander verwendet werden, befinden sich oft nahe beieinander im Speicher. **Oder sollten!**

# Spatial Locality: Beispiele

1

```
for (int i = 0; i < m; i++) {  
    for (int j = 0; j < n; j++) {  
        sum += array[i][j];  
    }  
}
```

Wie verhalten sich die Code-Stücke im Bezug auf Locality?

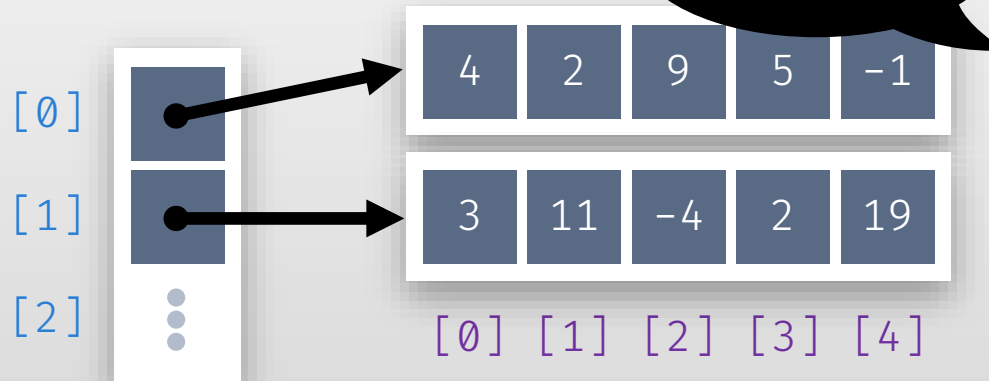
1: gute Locality 👍

2: schlechte Locality! 👎

2

```
for (int j = 0; j < n; j++) {  
    for (int i = 0; i < m; i++) {  
        sum += array[i][j];  
    }  
}
```

Werte innerhalb von (1D-)Array sind im Speicher benachbart!



**Performance-Unterschiede:**

1000 × 1000 **ints**: 4.6×

4000 × 4000 **bytes**: 22.0×



# Wann lohnt sich ein Cache?

Parameter:

$h$  Hit Ratio

$m = 1 - h$  Miss Ratio

$T_R$  Zugriffs-Zeit auf Ressource

$T_C$  Zugriffs-Zeit auf Cache

Zeit ohne Cache:  $T_{\text{ohne}} = T_R$

Zeit mit Cache:  $T_{\text{mit}} = m \times T_R + T_C$

plus sekundäre  
Effekte!

**Lohnt sich, wenn:**  $T_C / T_R < h$

# Caching-Effektivität erhöhen

**Ziel:** Zugriffszeit mit Cache reduzieren:  $T_{\text{mit}} = m \times T_R + T_C$

$T_R$  und  $T_C$  sind typischerweise nicht einfach änderbar. Bleibt  $m$ ...

Hit-/Miss-Ratio sind abhängig von:



- Cache-Grösse (Gesamtgrösse minus Metadaten)
- Zugriffsmuster
- *Cache-Algorithmus* (Cache Replacement Policy)

Möglichkeiten zum Erhöhen der Cache-Effektivität:

- Cache vergrössern (natürlich nicht bei Hardware-Caches...)
- Locality verbessern (Zugriffsmuster)
- **Cache-Algorithmus ändern!**



# Cache-Algorithmen

*(Cache Replacement Policies)*

# Cache-Algorithmen

Cache-Grösse ist begrenzt. Wenn Cache voll ist und neues Element gespeichert werden soll, muss mindestens 1 Element aus Cache entfernt werden. (Oder neues Element wird verworfen.)

*Cache-Algorithmus:* Bestimmt, welche Elemente entfernt werden

Theoretisch optimaler Cache-Algorithmus?

***Bélády-Algorithmus:** Immer das Element entfernen, das in Zukunft am längsten nicht verwendet wird.*

- Unmöglich in der Praxis, aber kann verwendet werden, um **theoretisches Minimum von Cache Misses** zu bestimmen
- Erlaubt Vergleich mit echten Algorithmen

# FIFO-Algorithmus

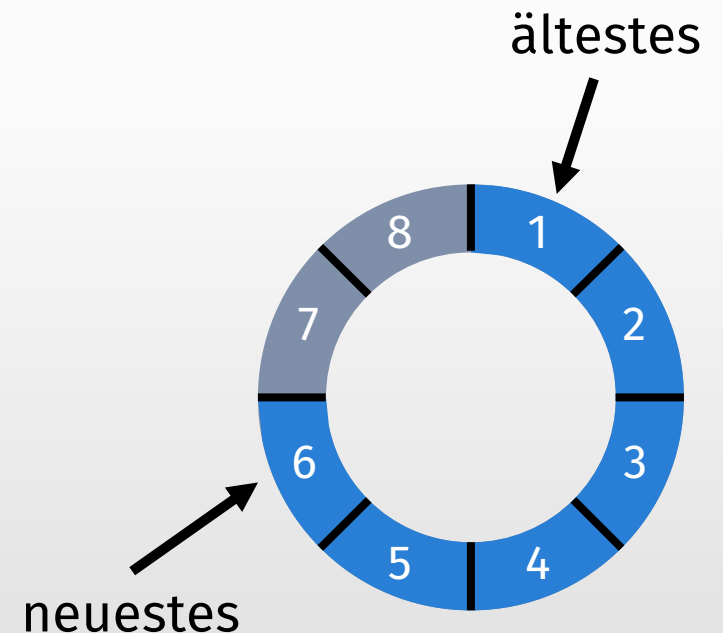
Gute **praktische** Algorithmen?

## *First-in-First-out*

Entfernt Element, das sich **am längsten im Cache** befindet

### Warum?

- Einfache Heuristik für Locality by time
- Effiziente Implementation mit Ringbuffer:  
Braucht nur zwei Zeiger, für ältestes und  
neuestes Element



# LRU-Algorithmus

## *Least Recently Used*

Entfernt Element, das am ***längsten nicht gelesen*** wurde

### **Warum?**

- «Echte» Locality by time (was aber selbst auch nur Heuristik ist!)

Muss aber für jedes Cache-Element Metadaten speichern, was nutzbare Cache-Grösse reduziert (Anzahl Elemente)

# MRU-Algorithmus

## *Most Recently Used*

Entfernt Element, das **als letztes** gelesen wurde

### **Warum???**

- In gewissen Situation gilt Temporal Locality nicht!  
(Dafür aber vielleicht Spatial Locality)

**Beispiel:** Mehrmaliges Iterieren über Datensatz, der knapp nicht in Cache passt. LRU würde keinen einzigen Cache Hit landen, MRU viele!

# LFU-Algorithmus

## *Least Frequently Used*

Entfernt Element, das **am wenigsten oft** gelesen wurde

### Warum?

- Allgemeine Heuristik: Vergangenheit ist aussagekräftig für Zukunft

Braucht Metadaten pro Element, wie LRU und MRU

Weitere: [en.wikipedia.org/wiki/Cache\\_replacement\\_policies](https://en.wikipedia.org/wiki/Cache_replacement_policies)

# RR-Algorithmus

## *Random Replacement*

Entfernt ein **zufälliges** Element

### Warum?

- Super-effizient und einfache Implementation ohne Metadaten
- Egal in welchem Muster auf Daten zugegriffen wird: Wahrscheinlichkeit, dass «wichtigstes» Element entfernt wird, ist klein. 🍏

Wurde wegen Einfachheit in CPU-Caches von ARM-Prozessoren verwendet

**Fragen?**

