Fachhochschule Nordwestschweiz
Hochschule für Technik

Application Performance Management

FS 2021

Caching

Michael Faes

Inhalt

Rückblick

Caching

Basics & Begriffe

Cache-Algorithmen

Verteilte Caches

Übung

Ein Cache für den Key–Value-Store

(Scripting mit JMeter)

Rückblick

Arten von Performance-Tests

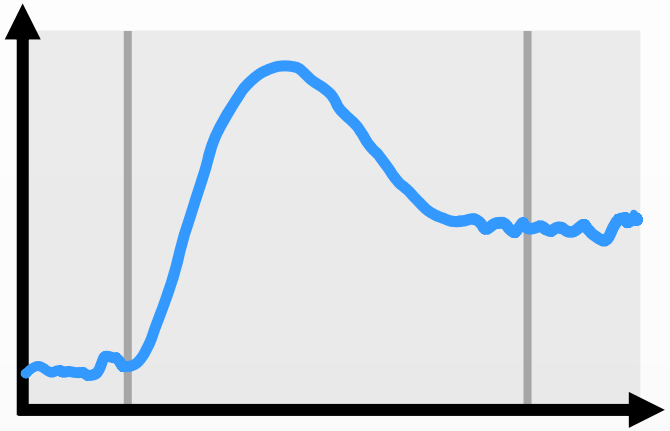
Lasttest: Analysieren der Leistung unter einer *konstanten, definierten* Last

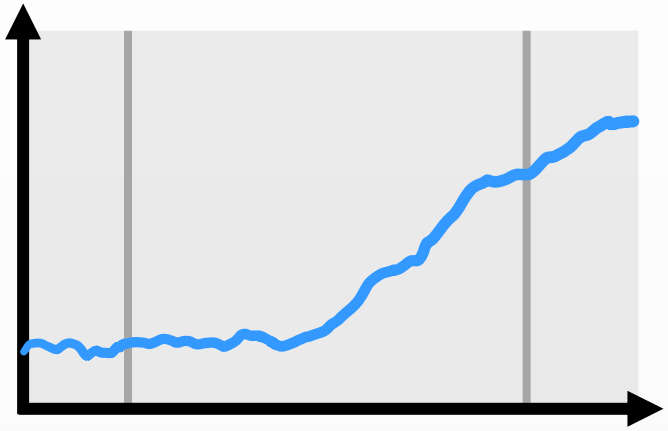
Dauerlasttest: Untersuchen der Leistung über einen *grösseren Zeitraum* hinweg

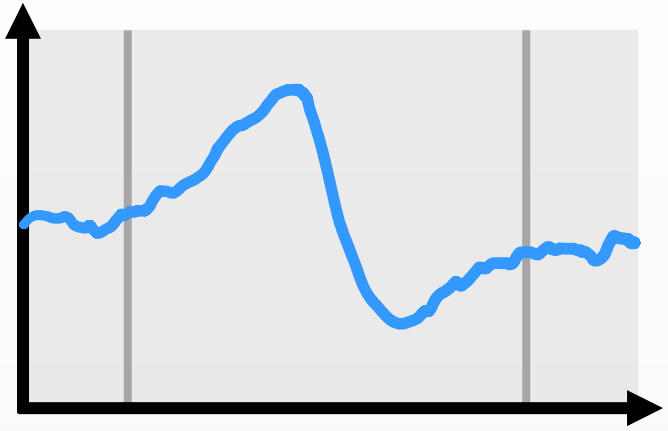
Failover-Test: Lasttest bei (manuell verursachtem) Ausfall von System-Komponenten

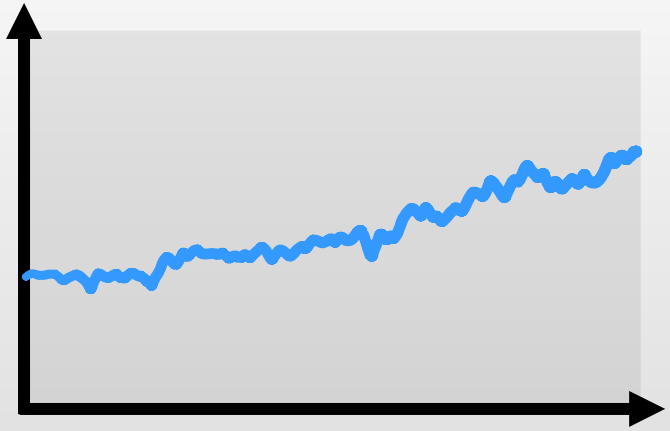
Stresstest: Schrittweises *Erhöhen der Last*, bis System instabil wird oder ganz ausfällt

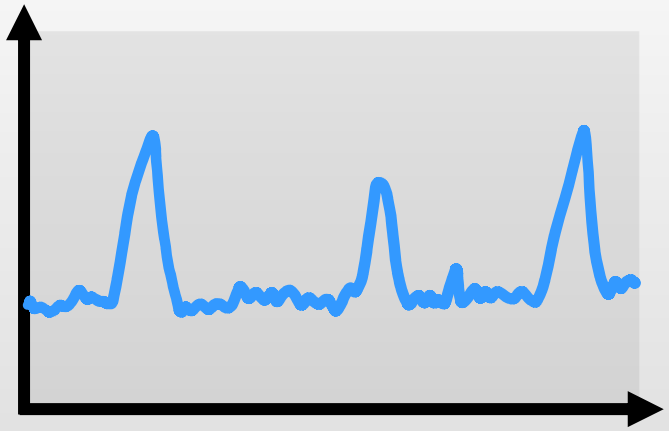
Lastverhalten einer Applikation











Caching

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*

https://dl.acm.org/citation.cfm?id=2814290

Memoization: Spezielle Form von Caching, d.h. Zwischenspeichern von berechneten Resultate

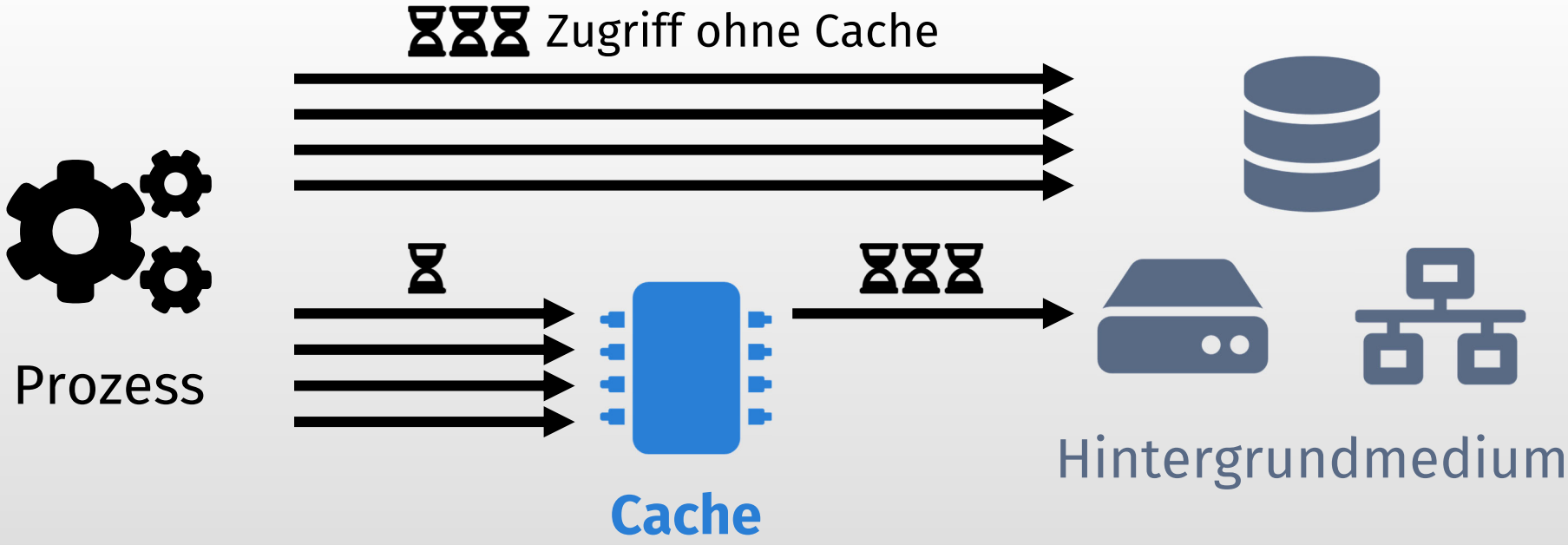
Paper: Automatische Analyse findet Orte in Java-Code, wo Caches sinnvoll sein könnten

Resultate: Bis zu **12.9× Speedup** für gesamte App!

Was ist ein Cache?

Schneller Puffer-Speicher, der wiederholte Zugriffe auf ein langsames Hintergrundmedium oder aufwändige Neuberechnungen zu vermeiden hilft.

— ungefähr Wikipedia



“Cache” vs. “Buffer”

Gemeinsamkeiten: Speicher, der…

- … eine limitierte Grösse hat
- … Datenobjekte desselben Typs speichert
- … die Leistung eines Systems optimiert

Buffer:

- Reduziert Transfers zu Medium durch Zusammenfassung (“Schreib-Cache”)
- Zwischenspeicher für Komponenten, die nicht direkt kommunizieren können

Cache:

- Speichert Daten, die ***in Zukunft*** wertvoll sein ***könnten*** (“Lese-Cache”)

Caching-Begriffe

Cache Access: Lese-Zugriff auf Cache-Speicher

Cache Hit (Cache Miss): Element wurde (nicht) gefunden

Hit Ratio (Miss Ratio): Anteil der Zugriffe, die zu einem Cache Hit (Miss) führten

- Hohe Hit Ratio: Cache ist "**hot**", sonst "**cold**"

Füllstand: Anteil des verwendeten Platz gegenüber Gesamtplatz im Cache

- Hotness und Füllstand grundsätzlich unabhängig!

Beispiele für Caches

Hardware:

- CPU Cache (L1, L2, L3, …)
- Filesystem Cache
- (Paging)
- Festplatten mit SSD-Caches

Software:

- Web (Browser oder Server)
- DNS
- Memoization
- Dynamic Programming
- Code Cache für JIT (Java)
- Website-Cache von Google
- Google selbst?!
- …

Nochmals: Warum Caching?

Caching ist eine Technik, die praktisch orthogonal zu anderen Technologien und Optimierungen eingesetzt werden kann und u.U. sehr effektiv ist.

Stellen Sie sich vor:

- Bei jedem Aufruf einer Website werden alle Grafiken, Schriftarten, Stylesheets, Scripts, usw. Neu geladen
- Und für jeden dieser Aufrufe noch eine DNS-Abfrage!
- …

Warum funktionieren Caches?

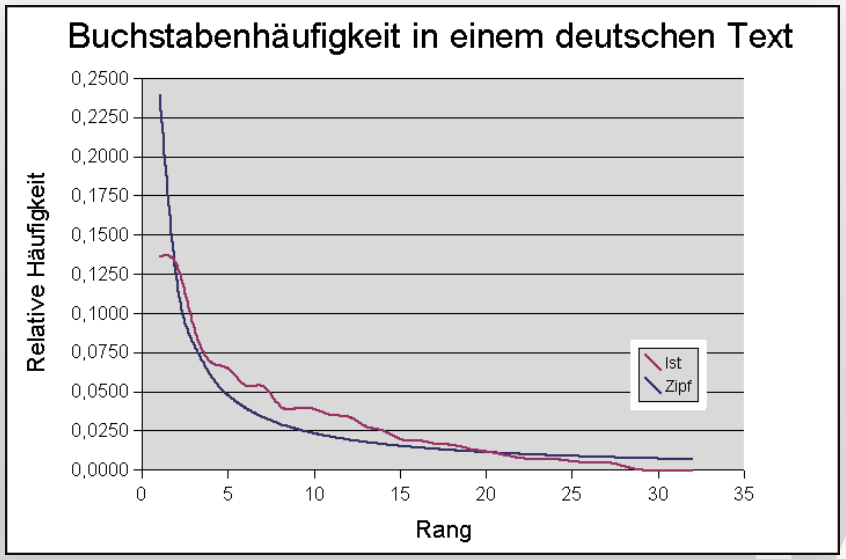
Principle of Locality

- **Locality by time:** wenn ein Objekt verwendet wird, wird es mit grosser Wahrsch. bald wieder verwendet
- **Locality by space:** ebenso “benachbarte” Objekte

Konkret: Zipf’s Law (80:20-Regel):

80% aller Zugriffe gehen auf die 20% selben Objekte

(oder auch 90:10, usw.)



“Anton” (deutsche Wikipedia)

Wann lohnt sich ein Cache?

Parameter:

h	hit ratio
$m = 1 - h$	miss ratio
T_m	Zugriffs-Zeit auf Medium
T_c	Zugriffs-Zeit auf Cache

Zeit ohne Cache: $T_{ohne} = T_m$

Zeit mit Cache: $T_{mit} = m \times T_m + T_c$

plus sekundäre Effekte!

Lohnt sich, wenn: $T_c \,/\, T_m < h$

Cache-Algorithmen

Cache-Grösse ist begrenzt. Wenn Cache voll ist, muss mind. ein Element aus Cache entfernt werden

- (Oder neues Element wird wieder verworfen)

Cache-Algorithmus (engl. Cache Replacement Policy) bestimmt, ***welche Elemente entfernt werden***

Bélády-Algorithmus

Theoretisch effizientester Cache-Algorithmus:

Entfernt immer das Element, welches in der Zukunft am längsten nicht benötigt wird.

Normalerweise unmöglich in der Praxis

- Aber kann im Nachhinein verwendet werden, um theoretisches Minimum von Cache Misses zu bestimmen
- Erlaubt Vergleich mit echten Algorithmen

Was wäre ein guter *praktischer* Algorithmus?

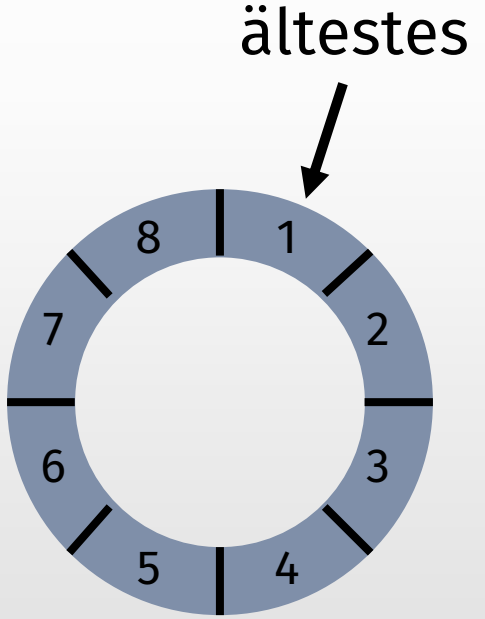
FIFO-Algorithmus

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 einen Zeiger auf ältestes 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 Zusatz-Info speichern (was Cache-Grösse reduziert)

MRU-Algorithmus

Most Recently Used!

Entfernt Element, das als letztes gelesen wurde

Warum???

- In gewissen Situation gilt temporal locality nicht! (Dafür vielleicht spatial locality)
- Beispiel: Mehrmaliges Iterieren über Datensatz, der knapp nicht in Cache past
- 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 Zusatz-Info pro Element, wie LRU und MRU

Weitere: en.wikipedia.org/wiki/Cache_replacement_policies

RR-Algorithmus

Random Replacement:

Entfernt ein zufälliges Element

Warum?

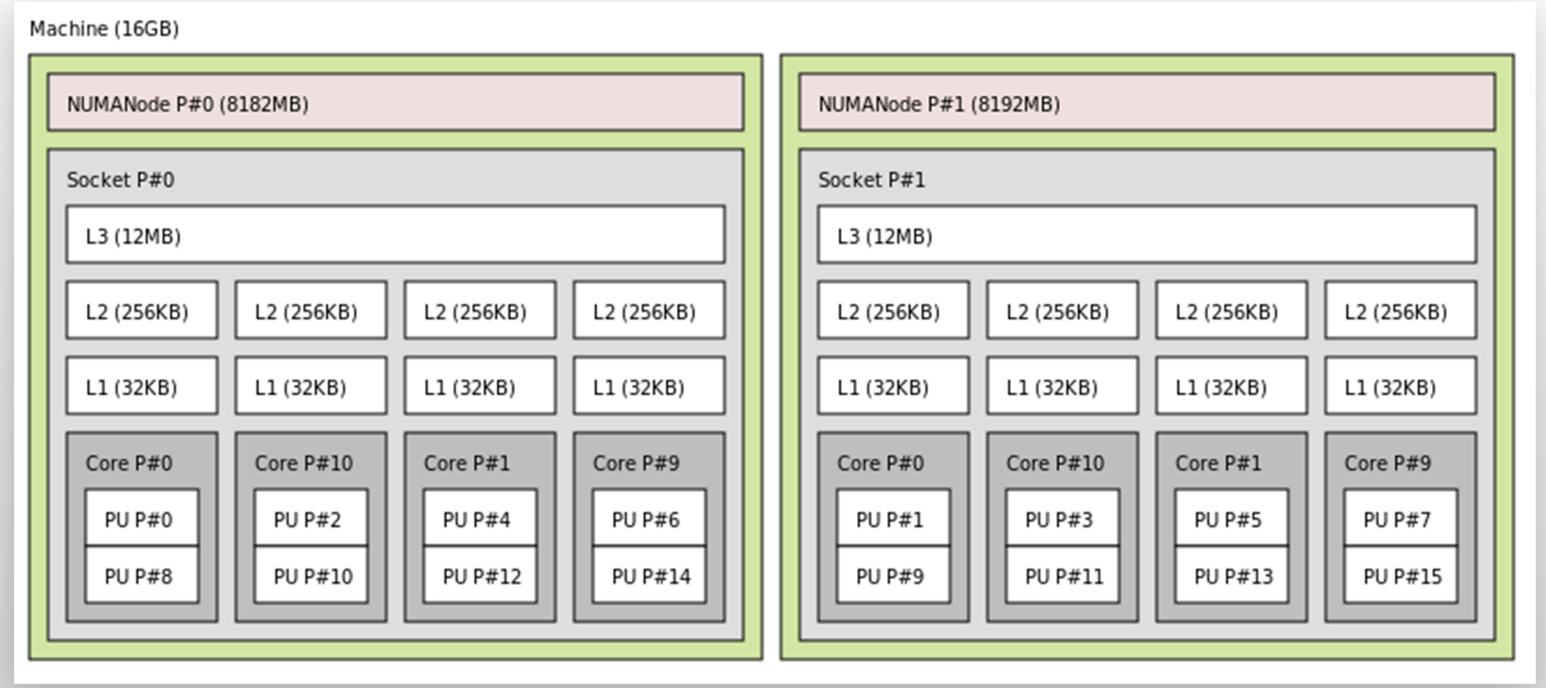
- Super-effizient und einfache Implementation ohne jeglichen Zusatz-Speicher
- Wahrscheinlichkeit, dass “wichtigstes” Element entfernt wird ist klein, unabhängig von Zugriff-Muster!

Wurde wegen Einfachheit in ARM-Prozessoren verwendet

Verteilte Caches

Mehrere, durch hohe Latenz getrennte Caches optimieren Zugriff auf das selbe Medium

Beispiel: CPU-Caches



xmodulo.com

Cache-Kohärenz

Cache-Koheränz: ***Ein System von Caches liefert konsistente Daten an alle verarbeitenden Einheiten***

- Temporäre Inkonsistenz ok, solange nicht gelesen wird

Write-through: Beim Schreiben werden Daten immer sofort auch auf Medium geschrieben

Write-back: Daten werden vorest im Cache gehalten und erst später (...) auf Medium geschrieben

- Inkonsistenz wird mit zusätzlichen Mechanismen behandelt
- Z.B. MESI-Protokoll in Intel CPUs

Übung:
Caching für den Key–Value-Store