Fachhochschule Nordwestschweiz
Hochschule für Technik

Application Performance Management

Performance-Optimierung & Profiling

Michael Faes

Übungsbesprechung

Guter Ort für Parallelisierung:

Threads immer wieder verwenden

```
private ExecutorService pool = Executors.newFixedThreadPool(n);
```

```
var results = Collections.synchronizedList(new ArrayList<Result>());

for (var doc : allDocs) {
    pool.submit(() -> {
        var res = findInDoc(searchText, doc);
        if (res.totalHits() > 0) {
            results.add(res);
        }
        return null;
    });
}
```

Zugriff synchronisieren!

«Task»

…

Weiteres «Problem»: Warten, bis alle Tasks fertig sind…

Vorteil von dieser Aufteilung: Tasks beinhalten genügend Arbeit, dass sich Parallelisierung lohnt

- Erstellen von Task-Objekt, Schicken an Threadpool und Synchronisation erzeugen *Overhead*. Falls zu wenig Arbeit, wird *Gewinn von Parallelisierung durch Overhead zunichte gemacht*...

Nachteil: Wenn zu wenig Dokumente vorhanden, gibt es nicht genügend Tasks, um alle CPU-Cores zu nutzen.

Task-Grösse ist ein *Trade-Off*:

Was ist mit Rest der Arbeit?

Hoffnung: Macht vielleicht sehr kleinen Teil aus…

```
var allDocs = collect…();

for (var doc : allDocs) {
    ...
}

results.sort(...);
```

seriell

parallelisierbar

seriell

Aber: Je mehr Parallelisierung, desto wichtiger sind *serielle Teile*!

Amdahlsches Gesetz

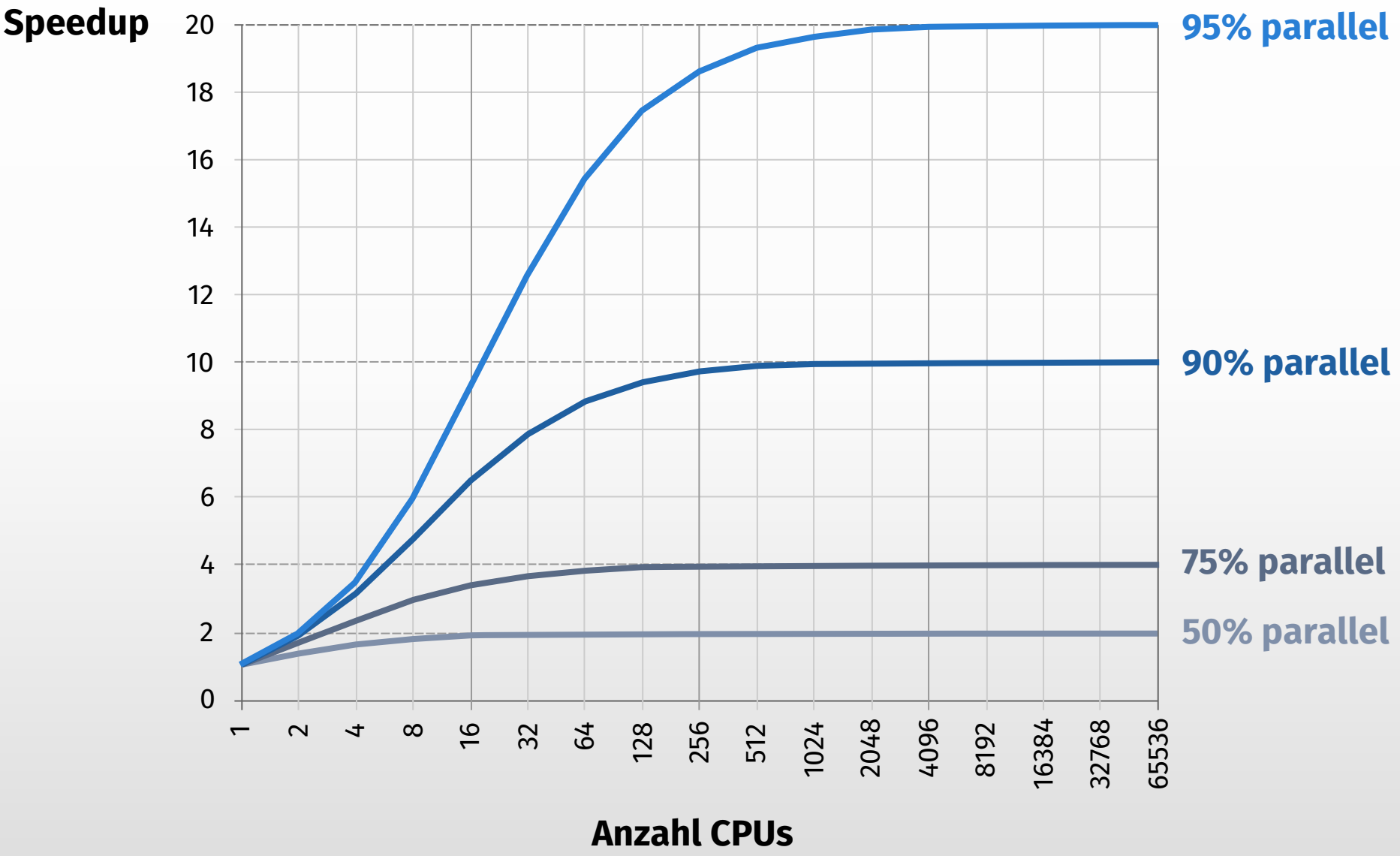
$$T = t_S + t_P$$

$$S = \frac{T}{t_S + \frac{t_P}{n_P}} \leq \frac{T}{t_S}$$

- T Gesamtlaufzeit (mit 1 CPU)
- t_S serieller Anteil
- t_P parallelisierbarer Anteil
- n_P Anzahl CPUs
- S paralleler Speedup (der Antwortzeit)

Beispiel: Code braucht 200 ms mit 1 CPU. `collect` und `sort` brauchen zusammen 10 ms. *Egal wie viele CPUs*, Speedup wird höchstens 20×!

Amdahlsches Gesetz veranschaulicht:



Performance-Optimierung

Performance-Optimierung

Typische Aufgabe: *Ein System «schneller» machen*

Rückblick Metriken: Was heisst «schneller» genau?

- *Antwortzeit* verkürzen?
- *Durchsatz* vergrössern?
- *Effizient* steigern?

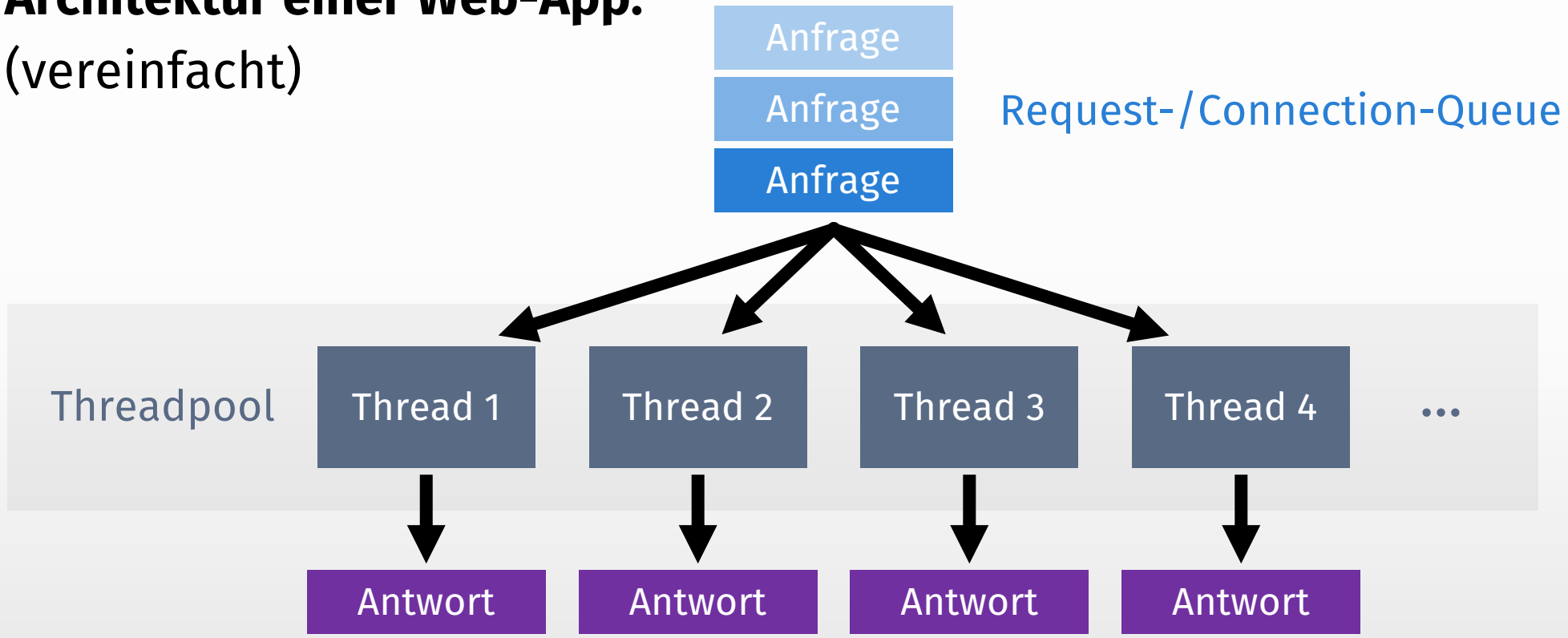
Bei CLI-App wie `DocFinderCli` im Prinzip kein Unterschied zwischen Antwortzeit und Durchsatz: Da immer nur 1 Anfrage aufs Mal, gilt:

$$Durchsatz = \frac{1}{Antwortzeit}$$

Aber bei Mehrbenutzer-Applikationen (z. B. Web-Apps) nicht!

(Inter- / Intra-Request-Parallelismus)

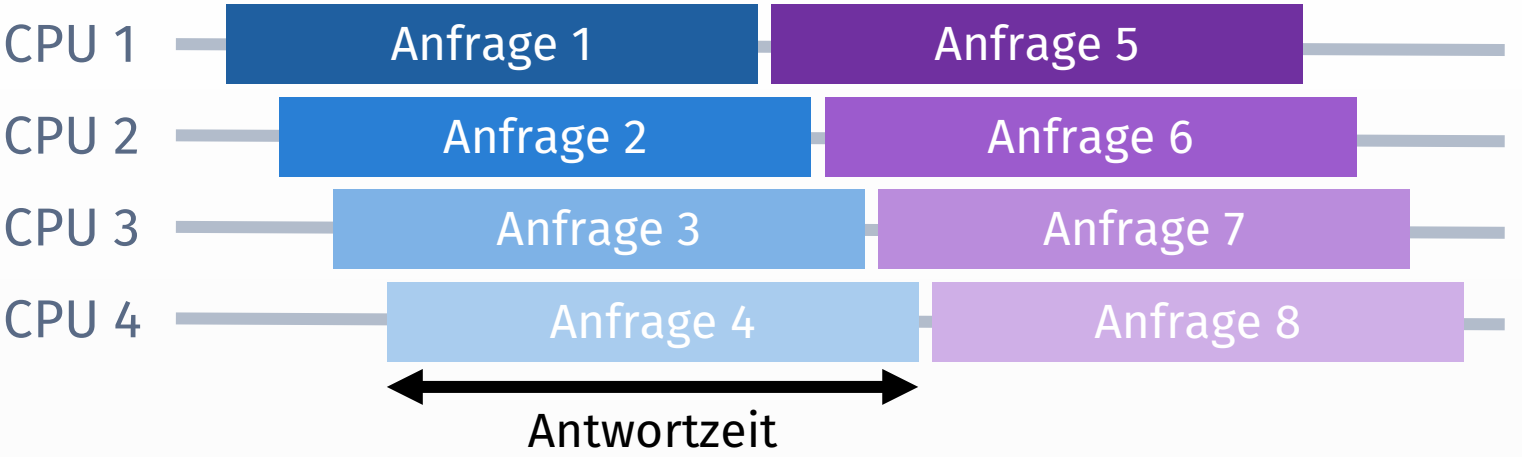
Architektur einer Web-App:
(vereinfacht)



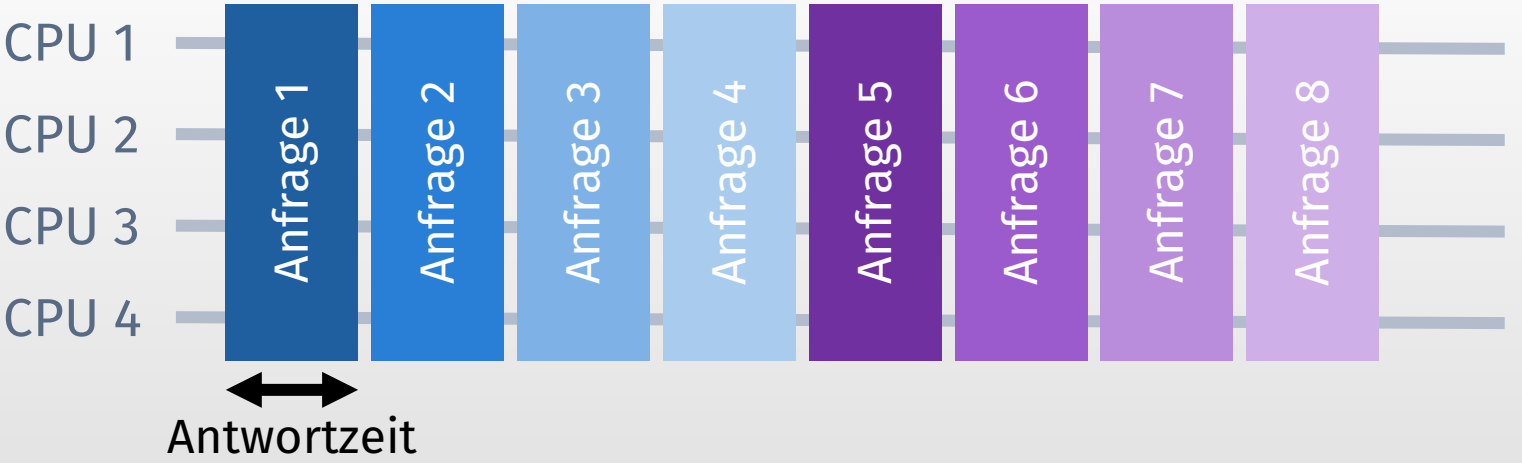
Anfragen werden auf Threads verteilt: ***Inter**-Request-Parallelismus*

Arbeit *innerhalb einer Anfrage* aufteilen: ***Intra**-Request-Parallelismus*

Ohne (Intra-Request-)Parallelismus:



Mit (Intra-Request-)Parallelismus:



Antwortzeit wird verbessert, aber Durchsatz nicht!

Performance-«Optimierung»

Achtung: Begriff «Optimierung» irreführend.

«*Unter einem Optimum versteht man das* ***beste erreichbare Resultat*** *im Sinne eines Kompromisses zwischen verschiedenen Parametern oder Eigenschaften* […].»

— Wikipedia

In vielen Fällen versucht man nicht, Performance-*Optimum* zu finden, sondern möchte einfach «*bessere Performance*»…

Anderer Begriff: *Performance-Tuning*

Übersicht Woche 2

1. Übungsbesprechung
2. Grundlagen Performance-Optimierung
3. Methodologien
4. Performance beobachten: Counters, Profiling & Tracing
5. Übung: *Profiling mit VisualVM*

Grundlagen Performance-Optimierung

Warum optimieren & wie viel?

Performance-Verbesserungen *müssen sich lohnen*. Wirtschaftliches Konzept: *Return of Investment (ROI)*

- **Kosten sparen.** Typisch für Organisationen mit grossen Datenzentren (Google, Amazon, Netflix, usw.)
- **Bessere User-Experience.** Wichtig (auch) für kleine Firmen/Startups. *Glückliche Kunden statt Ex-Kunden!*
- **Performance ist kritisch für Anwendung.** Echtzeitsysteme, High-Frequency-Trading, …
- **Produktivitätseinbusse.** Wichtig für interne Applikationen

ROI kann auch nützlich sein, um zu entscheiden, wann Performance-Optimierung «abgeschlossen» ist.

Perspektiven

Optimieren von Performance oder *Finden von Performance-Problem*?

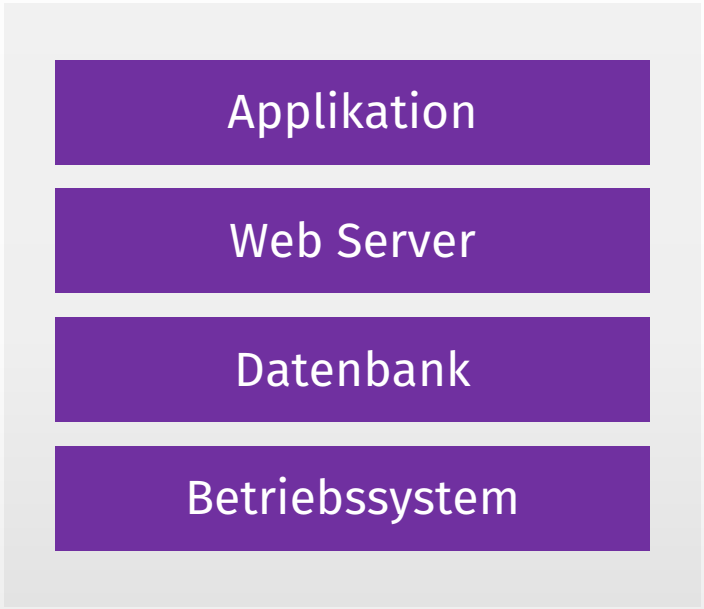
Eigentlich das selbe, nur Frage der Perspektive, bzw. Erwartung.

Erwartung	Resultat: Performance wird besser	Resultat: Performance bleibt gleich
Performance *müsste* höher sein!	Problem wurde behoben	Problem besteht weiterhin!
***Könnte* Performance höher sein?**	Performance wurde optimiert!	Performance konnte nicht optimiert werden

Grundprobleme

Bei Mehrbenutzer-Applikationen (z. B. Web-Apps):

Probleme der Last



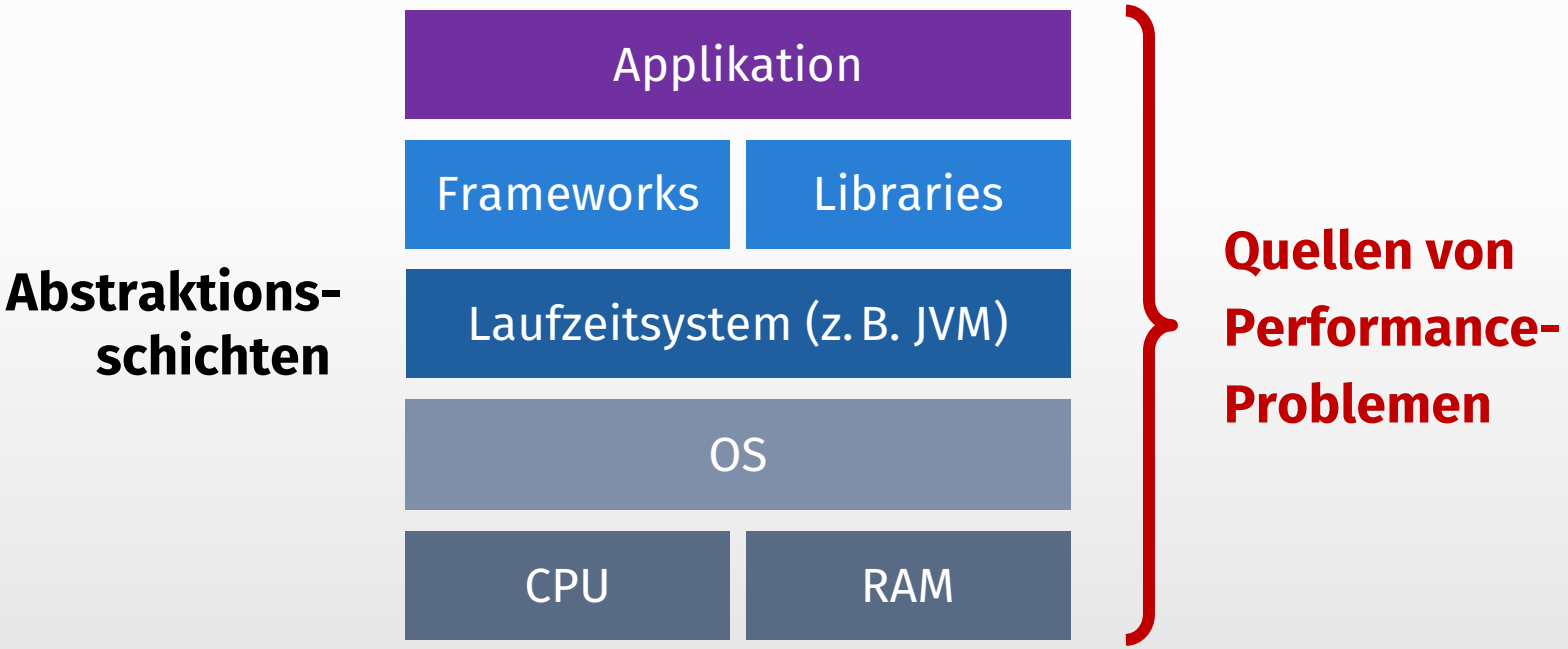
Probleme der Architektur/Implement.

Selbst ohne Probleme mit Architektur, Implementation, Konfiguration, usw., kann Performance leiden, wenn Last zu gross ist!

Quellen von Performance-Problemen

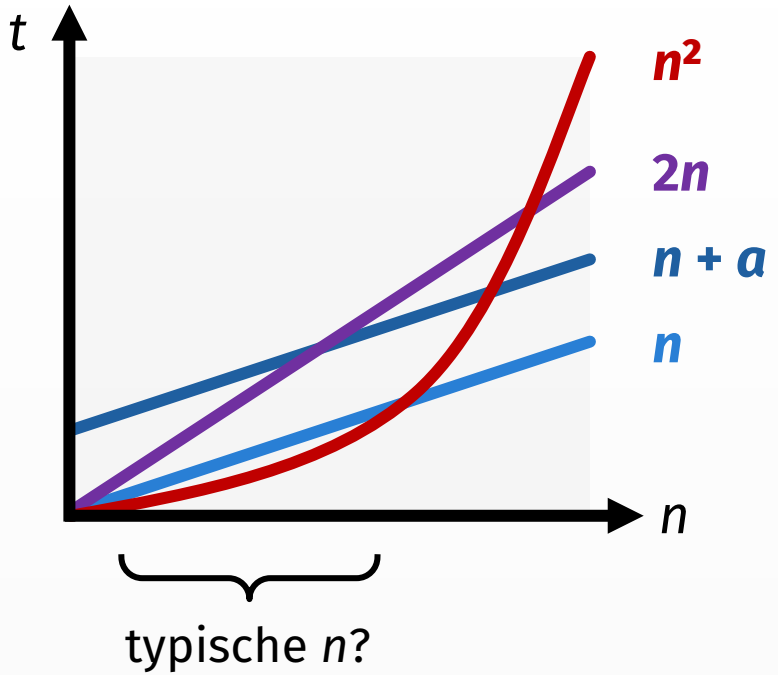
Performance-Probleme können von überall her kommen…

… und lassen sich nicht «weg-abstrahieren»:



Heisst: Übliche Informatik-Techniken helfen bei Performence wenig.

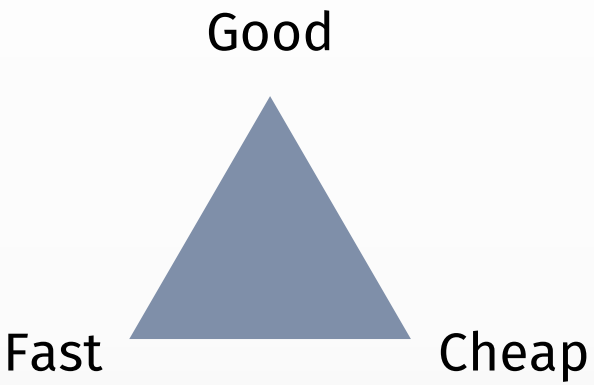
Performance-Optimierung kann bedeuten:

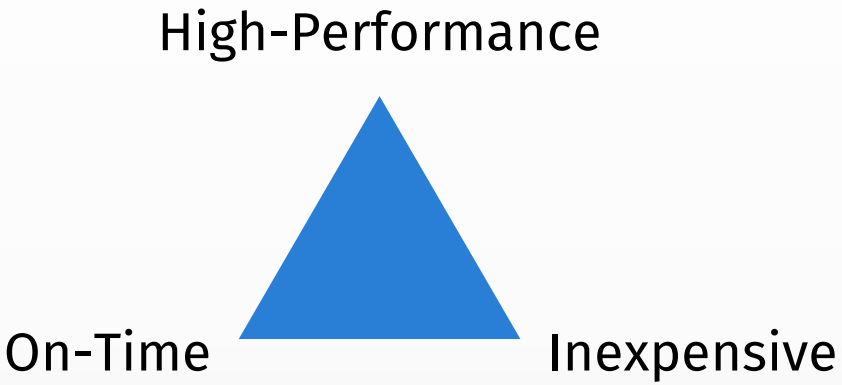


- Wechseln zu Algorithmus mit niedrigerer Zeitkomplexität
- Wechseln zu Algorithmus mit niedrigerem Overhead *für gegebene Datenmenge* (!)
- Effizientere Implementation von Algorithmus
- Verwenden von effizienterer Datenstruktur
- Intra-Request-Parallelisierung
- Einbauen von Cache an geeigneter Stelle im System
- Vermeiden von sonstigen Mehrfachberechnungen (Hint: Übung)
- Tunen von Parametern, z. B. Grösse von Threadpool, Cache-Grösse, …
- Hinzufügen eines Index in einer DB
- Wechseln zu effizienterem Compiler/zu effizienteren Sprache
- …

Trade-Offs

Good, fast, cheap: Pick any two.





CPU–Speicher-Tradeoff

- Wenn Arbeit rechenintensiv ist, kann Speicher verwendet werden, um Resultate zu cachen
- Auf Systemen mit vielen CPUs auch umgekehrt: CPU-Zeit verwenden, um Daten zu komprimieren und Speicher(-Zugriffe) zu sparen.

Bei vielen frei wählbaren Parametern gibt es Trade-Offs.

Beispiele

Grösse von Netzwerkbuffer:

kleinerer Buffer
weniger Overhead pro Verbindung → skalierbar

grösserer Buffer
besserer Netzwerk-Durchsatz

Grösse von Threadpool:

weniger Threads
weniger Scheduling-Overhead in OS

mehr Threads
bessere Nutzung von CPUs

Schon bekannt: Grösse von parallelen Tasks

Performance–Correctness-Tradeoff

Performance-Gewinne durch *«Reduktion» von Korrektheit*?

In gewissen Fällen schon:

- *Heuristiken* bei Optimierungsproblemen
  Beispiel: Schnellste Route bei Navi-Apps. Vereinfachung: Schnellste Route führt *meistens* über Autobahn. Reduziert Komplexität.
- *Präzision* bei Optimierungsproblemen
  Beispiel: Maximum einer Nutzenfunktion finden. Nach 2 signifikanten Stellen abbrechen und Rechenzeit sparen.
- *Genauigkeit (accuracy)* bei unkritischen Features
  Beispiel: Für Kaufstatistiken bei Web-Shop niedrigeres DB-Isolation-Level konfigurieren. Ein paar verlorene Updates sind egal (?)

Effektivität von Optimierung

Performance-Optimierung ist am effektivsten, wenn sie «nahe» bei der wirklichen Arbeit passiert. Also im Applikations-Code selbst.

Beispiele für Optimierungsmöglichkeiten

Ebene	Optimierungsmöglichkeiten	Gewinne
Applikation	Applikationslogik, DB-Anfragen, Caches	gross, z. B. 20×
Datenbank	Tabellen-Layout, Indizes	…
System calls	Lesen/Schreiben vs. Memory-mapped I/O	…
Dateisystem	Clustergrösse, Cachegrösse, Journaling	..
Speichermedium	RAID-Level, Anzahl und Typen von Medien	klein, z. B. 20%

Gesparte Arbeit «weiter unten» bringt wenig, da Arbeit «weiter oben» trotzdem gemacht wurde.

Methodologien

Methodologien

Methodologie (Vorgehensweise): nicht planlos Dinge ausprobieren, sondern systematisch vorgehen (und schneller ans Ziel kommen).

Anti-Methodologie: Strassenlaternen-Methode
Verwenden von bereits bekannten, im Internet gefundenen oder zufälligen Werkzeugen und schauen, ob etwas auffällt.

Analogie:
Eines Nachts sieht ein Polizist einen Betrunkenen den Boden unter einer Strassenlaterne absuchen und fragt ihn, was er sucht. Der Betrunkene antwortet, dass er seinen Schlüsselbund verloren habe. Der Polizist findet die Schlüssel auch nicht und fragt ihn: «Sind Sie sicher, dass Sie sie hier unter der Laterne verloren haben?» Der Betrunkene antwortet: ***«Nein, aber das Licht ist hier am besten.»***

Findet vielleicht *ein Problem*, aber wahrscheinlich nicht *das Problem*.

1. Problemstellung klären

Erster Schritt bei jeder Performance-Optimierung. Fragen:

1. Warum denkst du, dass ein Performance-Problem vorhanden ist?
2. Hatte dieses System überhaupt mal bessere Performance?
3. Was hat sich kürzlich geändert? Software? Hardware? Last?
4. Kann das Problem durch Antwortzeit/Laufzeit ausgedrückt werden?
5. Betrifft das Problem andere Personen/Apps oder nur dich/deine?
6. In welcher Umgebung tritt das Problem auf? Benützte Software und Hardware? Versionen, Konfiguration?

Ziel: Klare Problemstellung für folgende Analyse. Aber Antworten auf diese Fragen können Problem teilweise schon alleine lösen!

2. Scientific Method

Allgemeingültige Methode: Studieren von Unbekanntem durch *Aufstellen und Testen von Hypothesen*.

Schritte:

1. Frage
2. Hypothese
3. Vorhersage
4. Test (*Beobachtung* oder *Experiment*)
5. Auswertung

Beispiel: Frage: App ist langsamer auf System mit weniger Speicher. Hypothese: Grund ist kleinerer Dateisystem-Cache. Vorhersage: höhere Anzahl Cache-Misses. Test: Messen der Cache-Misses.

3. USE-Methode

Von Brendan Gregg. Fokus auf Ressourcen-Auslastung. Ziel: *Bottleneck* in einem System finden.

Zusammenfassung:

Für jede Ressource, prüfe Auslastung, Sättigung und Fehler.

Utilization **S**aturation **E**rrors

USE

Idee: Komplette Liste von Ressourcen führt dazu, dass nichts wichtiges übersehen wird.

Auch Ressourcen, die nicht (einfach) analysiert werden können, gehören in Liste. Zumindest *weiss man, dass man etwas nicht weiss!*

(Metriken für Ressourcennutzung)

Auslastung (utilization)

Definition 1: *Verhältnis zwischen Zeit, in der Ressource verwendet wird, und Gesamtzeit*

Definition 2: *Durchschnittlicher Anteil von verwendeter Kapazität*

Beispiele

- CPU: wurde *während 80% der Zeit* verwendet
- Speicher: war *durchschnittlich zu 60%* ausgelastet

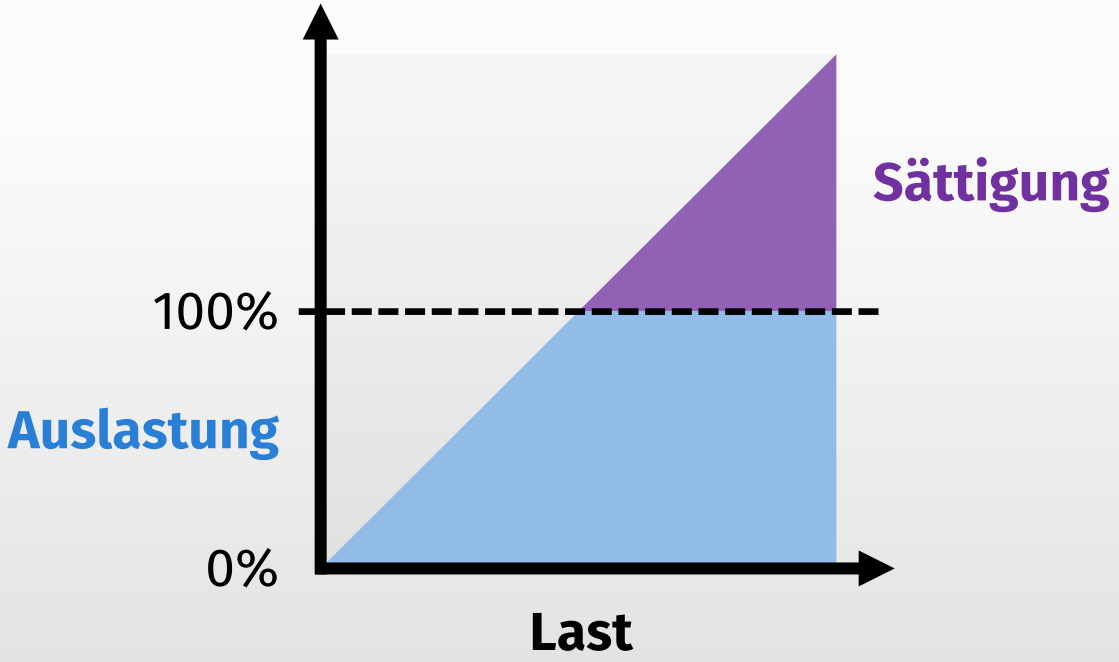
Unterschied zwischen *verwendet* und *ausgelastet*!

Beispiel Personenlift: Selbst wenn Lift während 100% der Zeit *verwendet* wird (in Bewegung), ist er nicht unbedingt voll *ausgelastet*.

Sättigung (saturation)

Ausmass von «Überbelastung», d. h. Arbeit, die wegen Vollauslastung nicht sofort erledigt werden kann

Viele Ressourcen «akzeptieren» immer noch Arbeit, wenn sie 100% ausgelastet sind. Arbeit landet in Warteschlange.



Demo: CPU «load average» unter Linux

USE-Methode, Schritt 1: Liste von Ressourcen

Mögliche Hardware-Ressourcen

- CPU (Sockets, Cores, Hardware Threads)
- Speicher
- Netzwerkschnittstellen (Ethernet, WLAN, …)
- Speichermedien (Festplatten, SSDs)
- …

Mögliche Software-Ressourcen

- Locks (App)
- Threadpools (App)
- Heap-Grösse & Garbage Collection (JVM)
- Max. Anzahl Prozesse/Threads (OS)
- Max. Anzahl offener Dateien (OS)
- …

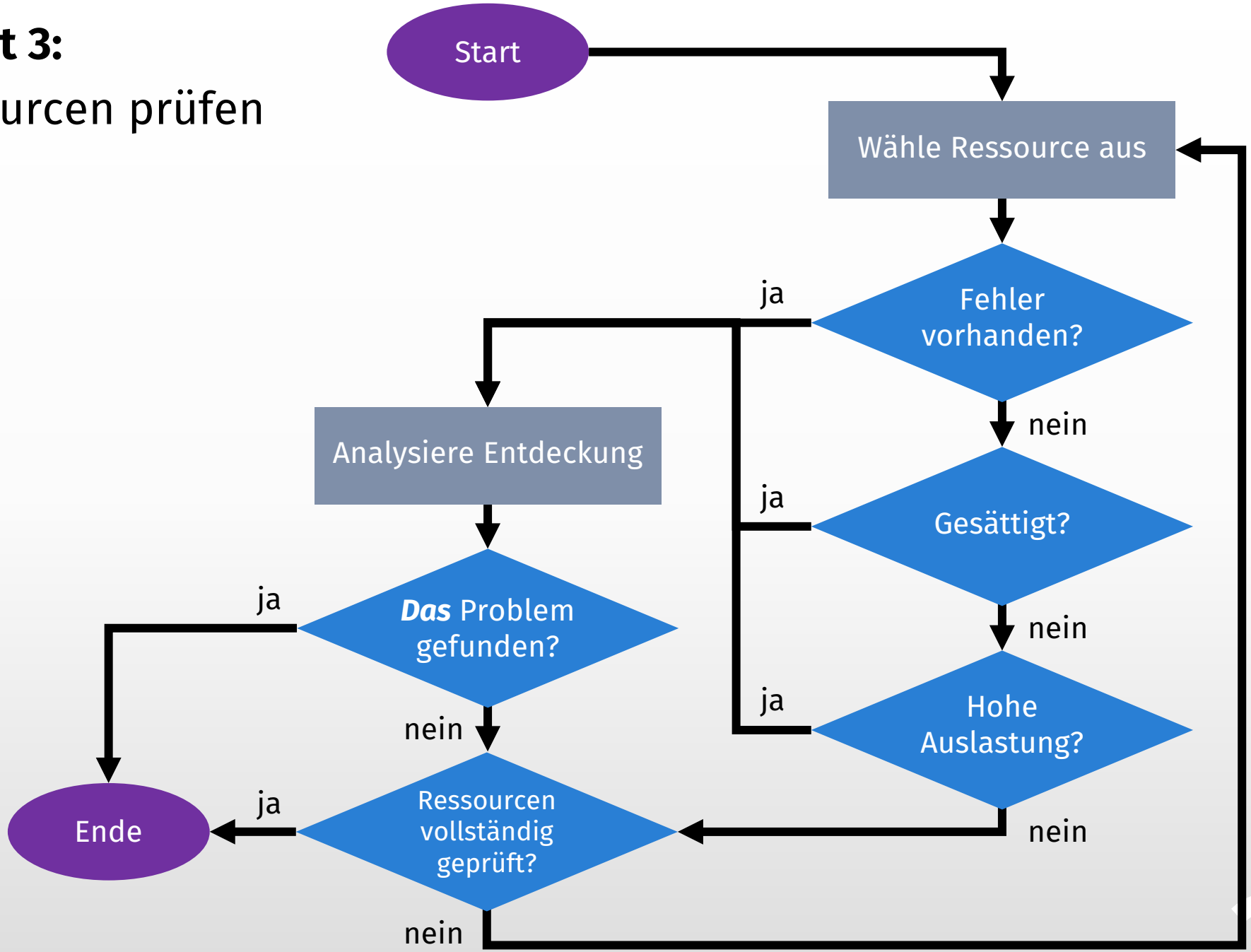
Schritt 2: Metriken für jede Ressource

Beispiele

Ressource	Art	Metriken
CPU	Auslastung	Pro CPU-Auslastung, Gesamtauslastung
CPU	Sättigung	«run queue length»
Speicher	Auslastung	verfügbarer Speicher (systemweit), verfügbarer Heap-Platz (JVM)
Speicher	Sättigung	Swapping
Speicher	Fehler	`OutOfMemoryError` (JVM)
Garbage Collection	Auslastung	CPU-Anteil von GC-Threads
Netzwerkschnittst.	Auslastung	Empfang-Durchsatz, Sende-Durchsatz
Speichermedium	Auslastung	% verwendete Zeit, Lese-Durchsatz, Schreib-Durchsatz
Speichermedium	Fehler	Gerätefehler, z. B. S.M.A.R.T.

Schritt 3:

Ressourcen prüfen



Weitere Details & Methodologien: *Systems Performance* (Gregg 2020)

Ausschnitte im AD:

2.5 Methodology

When faced with an underperforming and complicated system environment, the first challenge can be knowing where to begin your analysis and how to proceed. As I said in Chapter 1, performance issues can arise from anywhere, including software, hardware, and any component along the data path. Methodologies can help you approach these complex systems by showing where to start your analysis and suggesting an effective procedure to follow.

This section describes many performance methodologies and procedures for system performance and tuning, some of which I developed. These methodologies help beginners get started and serve as reminders for experts. Some *anti-methodologies* have also been included.

To help summarize their role, these methodologies have been categorized as different types, such as observational analysis and experimental analysis, as shown in Table 2.4.

Table 2.4 **Generic system performance methodologies**

Section	Methodology	Type
2.5.1	Streetlight anti-method	Observational analysis
2.5.2	Random change anti-method	Experimental analysis
2.5.3	Blame-someone-else anti-method	Hypothetical analysis
2.5.4	Ad hoc checklist method	Observational and experimental analysis
2.5.5	Problem statement	Information gathering
2.5.6	Scientific method	Observational analysis
2.5.7	Diagnosis cycle	Analysis life cycle
2.5.8	Tools method	Observational analysis
2.5.9	USE method	Observational analysis
2.5.10	RED method	Observational analysis
2.5.11	Workload characterization	Observational analysis, capacity planning
2.5.12	Drill-down analysis	Observational analysis
2.5.13	Latency analysis	Observational analysis
2.5.14	Method R	Observational analysis
2.5.15	Event tracing	Observational analysis
2.5.16	Baseline statistics	Observational analysis
2.5.17	Static performance tuning	Observational analysis, capacity planning

Appendix A

USE Method: Linux

This appendix contains a checklist for Linux derived from the USE method [Gregg 13d]. This is a method for checking system health, and identifying common resource bottlenecks and errors, introduced in Chapter 2, Methodologies, Section 2.5.9, The USE Method. Later chapters (5, 6, 7, 9, 10) described it in specific contexts and introduced tools to support its use.

Performance tools are often enhanced, and new ones are developed, so you should treat this as a starting point that will need updates. New observability frameworks and tools can also be developed to specifically make following the USE method easier.

Physical Resources

Component	Type	Metric
CPU	Utilization	Per CPU: `mpstat -P ALL 1`, sum of CPU-consuming columns (`%usr`, `%nice`, `%sys`, `%irq`, `%soft`, `%guest`, `%gnice`) or inverse of idle columns (`%iowait`, `%steal`, `%idle`); `sar -P ALL`, sum of CPU-consuming columns (`%user`, `%nice`, `%system`) or inverse of idle columns (`%iowait`, `%steal`, `%idle`) System-wide: `vmstat 1`, `us` + `sy`; `sar -u`, `%user` + `%nice` + `%system` Per process: `top`, `%CPU`; `htop`, `CPU%`; `ps -o pcpu`; `pidstat 1`, `%CPU` Per kernel thread: `top`/`htop` (`K` to toggle), where VIRT == 0 (heuristic)
CPU	Saturation	System-wide: `vmstat 1`, `r` > CPU count[1]; `sar -q`, `runq-sz` > CPU count; `runqlat`; `runqlen` Per process: /proc/PID/schedstat 2nd field (`sched_info.run_delay`); `getdelays.c`, `CPU`[2]; `perf sched latency` (shows average and maximum delay per schedule)[3]

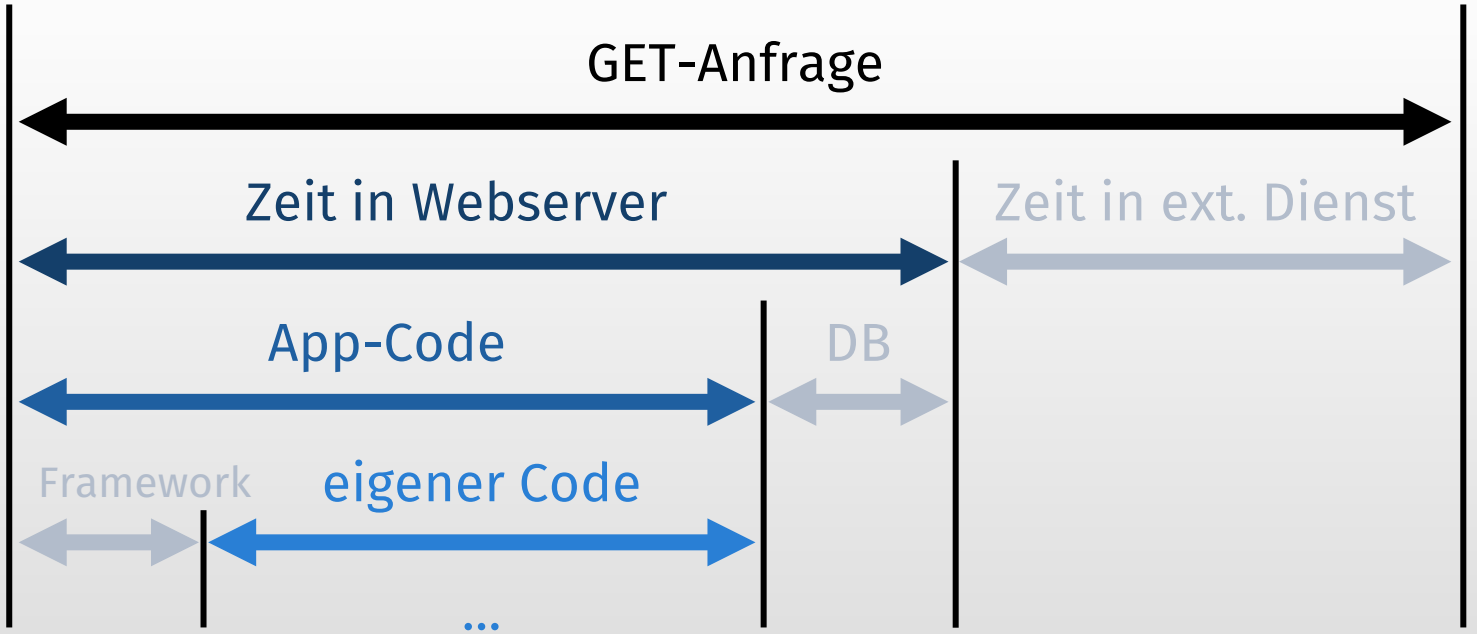
[1] The `r` column reports those threads that are waiting *and* threads that are running on-CPU. See the vmstat(1) description in Chapter 6, CPUs.
[2] Uses delay accounting; see Chapter 4, Observability Tools.
[3] There is also the sched:sched_process_wait tracepoint for perf(1); be careful about overheads when tracing, as scheduler events are frequent.

4. Antwortzeit-Analyse

Nützliche Methodologie für Probleme/Verbesserung der Antwortzeit

Idee: Antwortzeit einer Operation in kleinere Teile aufteilen, Zeit für Teile messen und dann *für längeren Teil wiederholen*.

Beispiel: GET-Anfrage



5. Performance-Mantras

Möglichkeiten bei Performance-Optimierungen
(sortiert nach Effektivität):

1. Mach es nicht.
2. Mach es, aber mach es nicht *nochmals*.
3. Mach es seltener.
4. Mach es später.
5. Mach es, wenn niemand schaut.
6. Mach es nebenläufig.
7. Mach es billiger (effizienter).

Profiling

Profiling

Konkrete Anwendung der Antwortzeit-Analyse: *Profiling*

Beantwortet Frage:

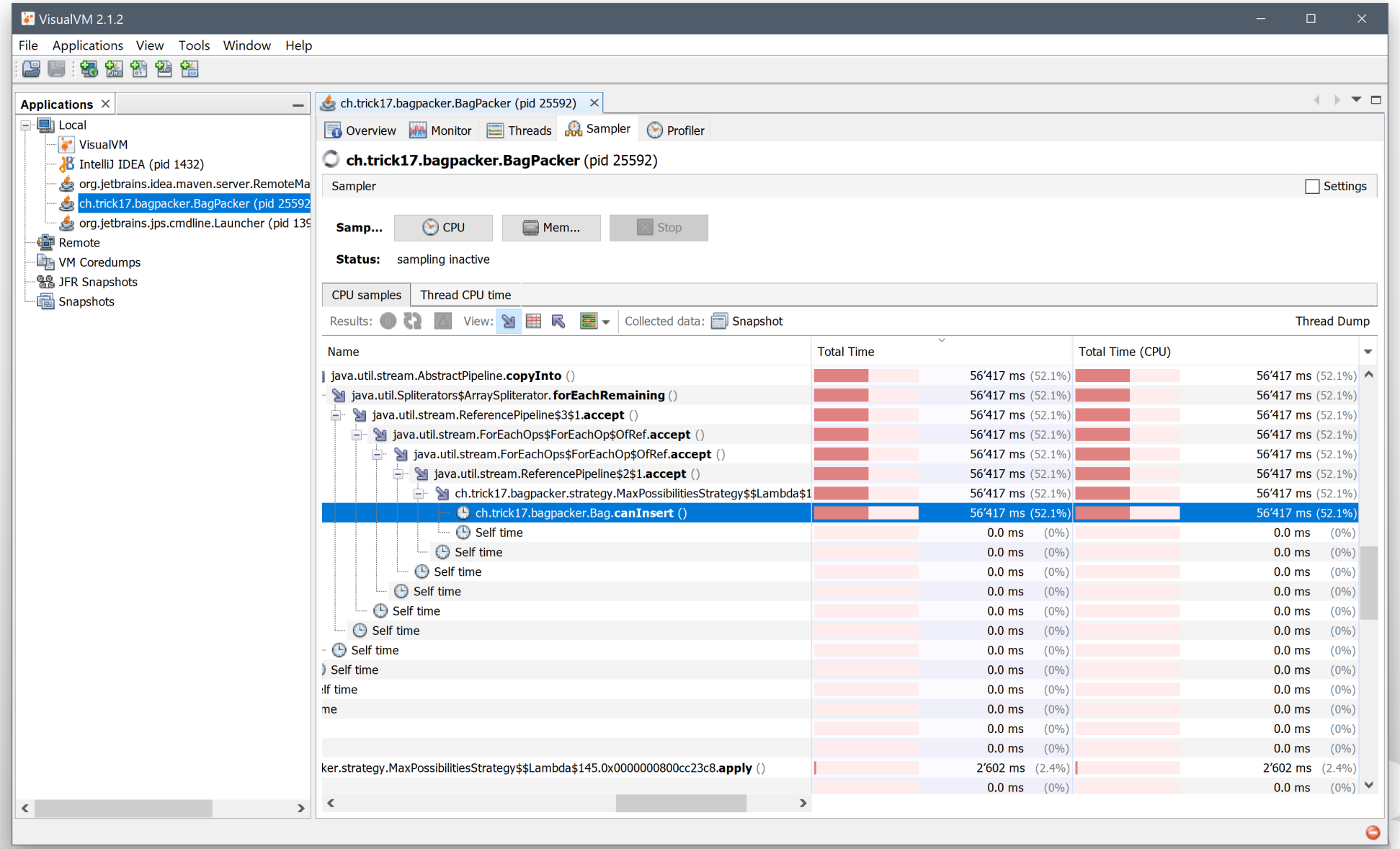
Welche Operationen eines Programms dauern am längsten?

- Oder: *Welche Objekte brauchen am meisten Speicher?*
- Oder: *Welche Operationen verwenden am häufigsten Locks?*
- Oder: *Welche Datenbank-Anfragen dauern am längsten?*

Typischerweise: *Welche **Methoden/Funktionen** dauern am längsten?*

Produziert ein *Profil*: grobes Bild von Programmablauf, das Performance-Optimierungen lenken kann.

VisualVM


VisualVM 2.1.2
File Applications View Tools Window Help
Applications
Local
VisualVM
IntelliJ IDEA (pid 1432)
org.jetbrains.idea.maven.server.RemoteMa
ch.trick17.bagpacker.BagPacker (pid 25592
org.jetbrains.jps.cmdline.Launcher (pid 139
Remote
VM Coredumps
JFR Snapshots
Snapshots
ch.trick17.bagpacker.BagPacker (pid 25592)
Overview Monitor Threads Sampler Profiler
ch.trick17.bagpacker.BagPacker (pid 25592)
Sampler
Settings
Samp... CPU Mem... Stop
Status: sampling inactive
CPU samples Thread CPU time
Results: View: Collected data: Snapshot
Thread Dump
Name
Total Time
Total Time (CPU)
java.util.stream.AbstractPipeline.copyInto ()
java.util.Spliterators$ArraySpliterator.forEachRemaining ()
java.util.stream.ReferencePipeline$3$1.accept ()
java.util.stream.ForEachOps$ForEachOp$OfRef.accept ()
java.util.stream.ForEachOps$ForEachOp$OfRef.accept ()
java.util.stream.ReferencePipeline$2$1.accept ()
ch.trick17.bagpacker.strategy.MaxPossibilitiesStrategy$$Lambda$1
ch.trick17.bagpacker.Bag.canInsert ()
Self time
ker.strategy.MaxPossibilitiesStrategy$$Lambda$145.0x0000000800cc23c8.apply ()
56'417 ms (52.1%)
0.0 ms (0%)
2'602 ms (2.4%)

Fragen?