

Application Performance Management

Frühling 2021

Garbage Collection Tuning

Zoltán Majó

Agenda

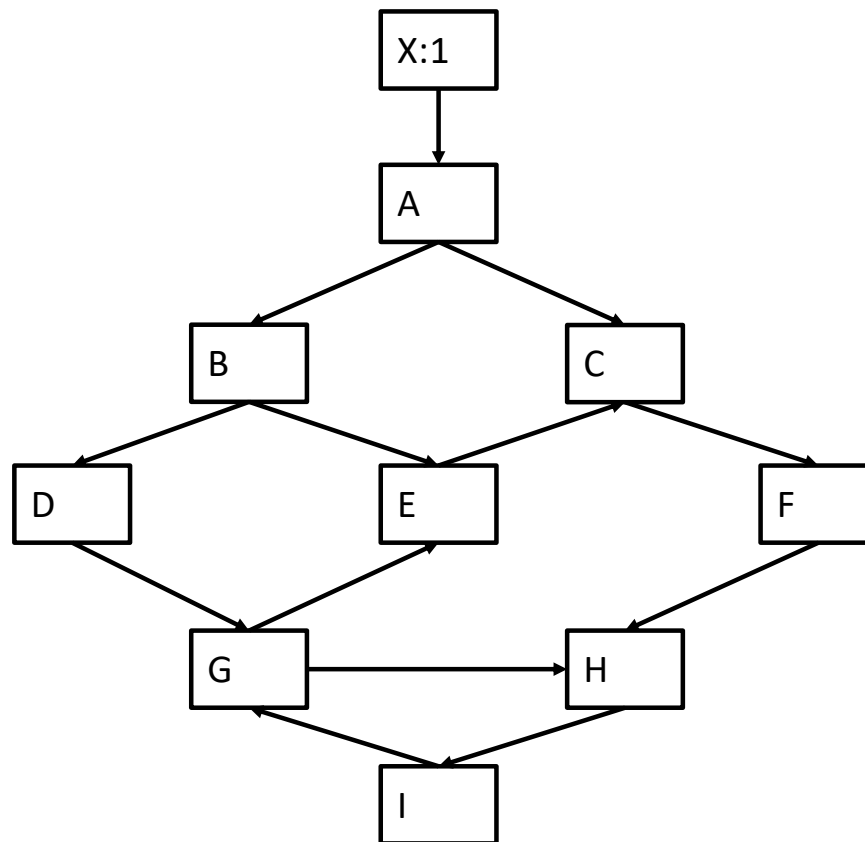
Diskussion Übungen SW8

Generational GC

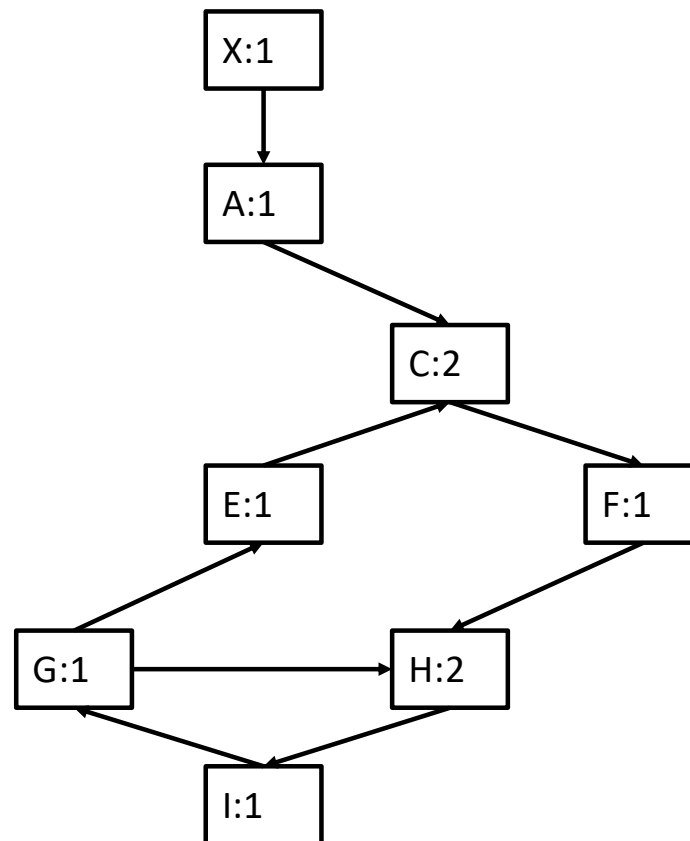
Paralleler und nebenläufiger GC

GC Tuning

Reference Counting




Reference Counting



Vergleich Speicherverwaltungsmethoden

Relevante Kriterien?



Algorithmus			
Reference Counting			
Mark & Sweep GC			
Mark & Copy GC			
Mark & Compact GC			

Kriterien (Input aus Individualarbeit)

Komplexität

- Reference Counting: Mehr Aufwand immer wieder
- Mark and Sweep: Etwas weniger Komplexität, ähnlicher Speicherverbrauch
- Mark and Copy: Mehr Speicherverbrauch
- Mark and Compact: Höchste Komplexität (Heap wird dreimal traversiert, kein Extraverbrauch)

Speicherverbrauch

Stop-the-world

Implementationsschwierigkeit

- Mark and Compact ist wahrscheinlich am schwierigsten zu realisieren

Fragmentierung

Geschwindigkeit

- Kopieren kann Bus-Kapazität in Anspruch nehmen

Vergleich Speicherverwaltungsmethoden

Algorithmus	Fragmentierung	Footprint	Stop-the-world	Geschwindigkeit Allokierung	Geschwindigkeit GC	Cycles
Reference Counting	✗	✓	✗	✗	N/A	✗
Mark & Sweep GC	✗	✓	✗	✗	✗	✓
Mark & Copy GC	✓	✗	✗	✓	✓	✓
Mark & Compact GC	✓	✓	✗	✓	✗	✓

Vergleich Speicherverwaltungsmethoden

Bemerkung: Alle Algorithmen brauchen das Benutzerprogramm stoppen

Algorithmus	Fragmentierung	Footprint	Stop-the-world	Geschwindigkeit Allokation	Geschwindigkeit GC	Cycles
Reference Counting	✗	✓	✗	✗	N/A	✗
Mark & Sweep GC	✗	✓	✗	✗	✗	✓
Mark & Copy GC	✓	✗	✗	✓	✓	✓
Mark & Compact GC	✓	✓	✗	✓	✗	✓

Bemerkungen

Kein Algorithmus hat gutes Resultat bei allen Kriterien

- Stop-the-world ist problematisch bei allen

Frage: Könnte man verschiedene Algorithmen kombinieren?

- «Best-of-both-worlds» Lösung

Generational GC

Empirische Beobachtung 1

Wenn ein Objekt eine lange Zeit erreichbar war, wird es wahrscheinlich erreichbar bleiben

Empirische Beobachtung 2

In vielen Programmen sterben die meisten Objekte jung

Idee 1: Arbeit wird erspart, wenn junge Objekte häufig und alte Objekte selten gescanned werden.

Idee 2: Für alte Objekte kann mehr Aufwand gewidmet werden.

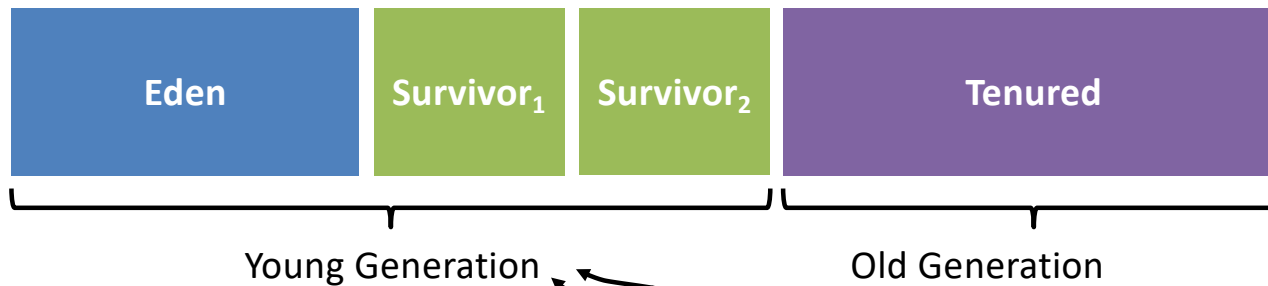
Generational GC in der HotSpot JVM

Bemerkung: Bisher wurde GC unabhängig von einer konkreten Implementierung betrachtet

Jetzt diskutieren wir eine konkrete Implementierung (Java HotSpot 8)

Generational GC in der HotSpot JVM

Heap Layout



Minor GC

- Nur young Generation wird collected
- Mark and Copy Algorithmus

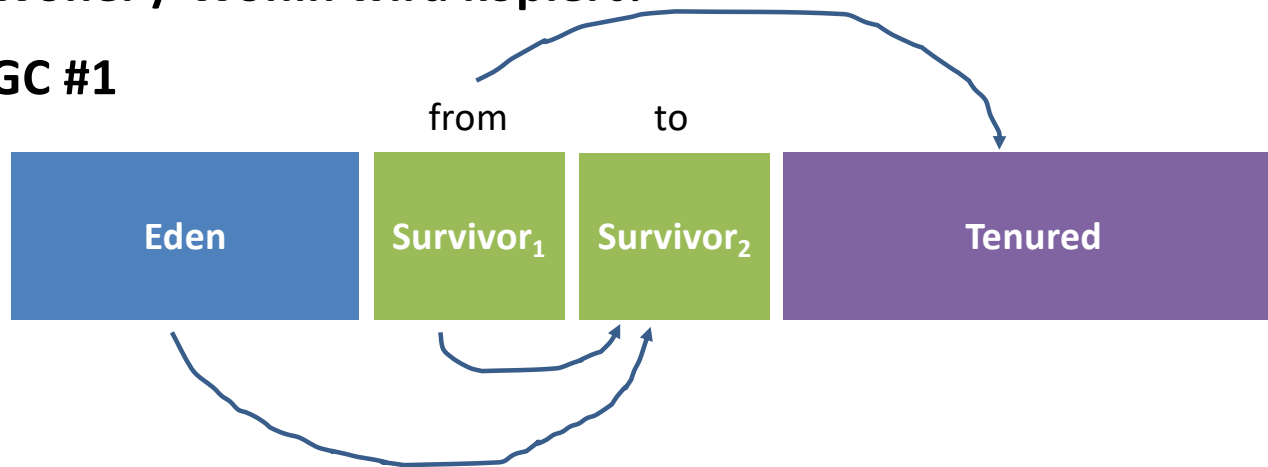
Major GC

- Young und old Generation wird collected
- Mark and Compact Algorithmus

Minor GC

Woher / Wohin wird kopiert?

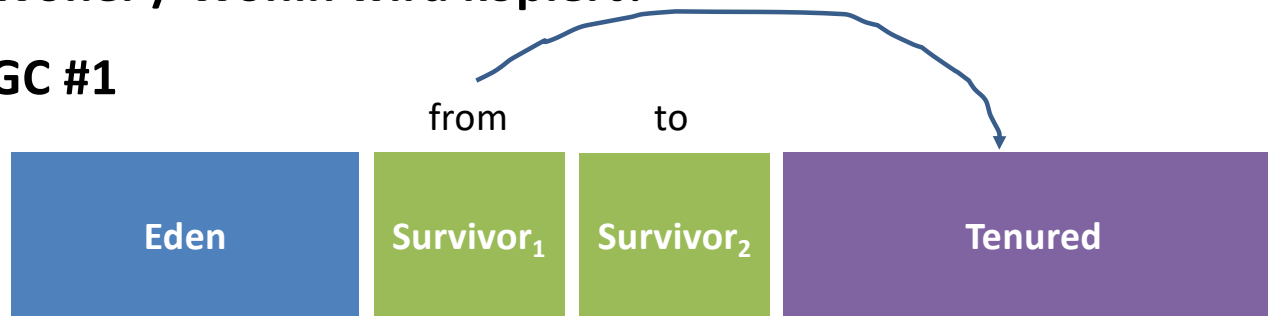
GC #1



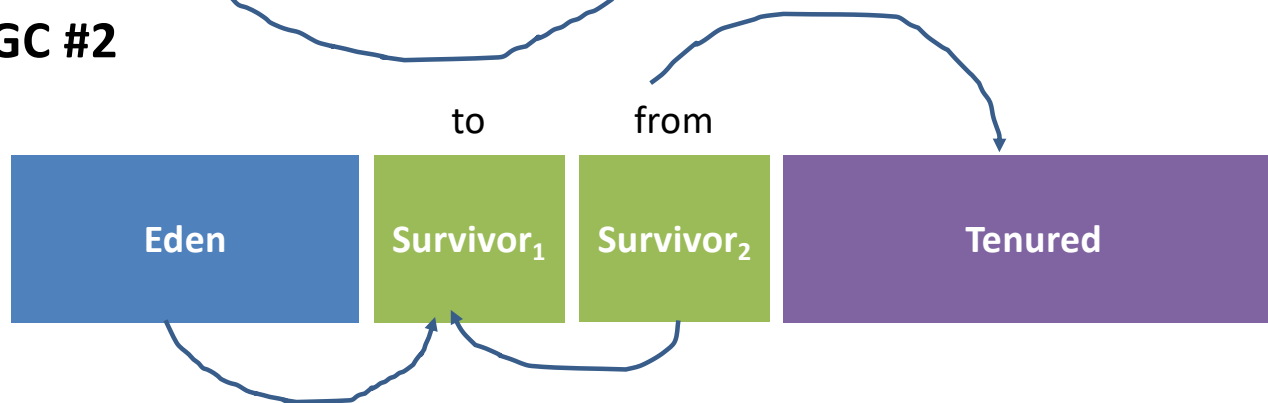
Minor GC

Woher / Wohin wird kopiert?

GC #1



GC #2

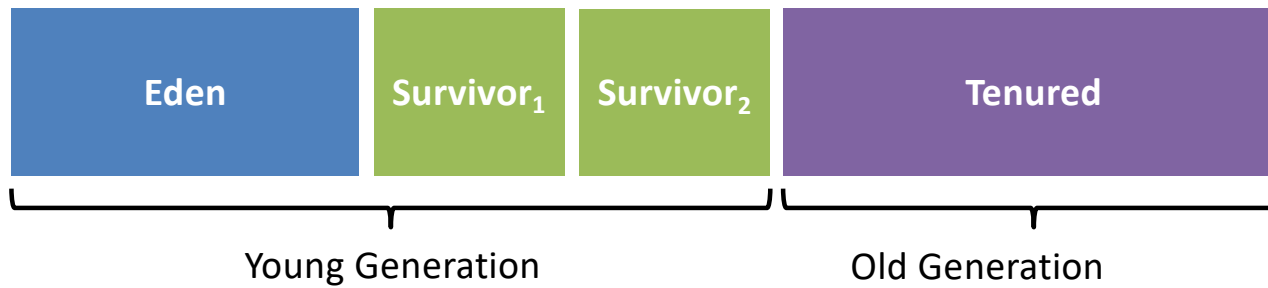


Generational GC in der HotSpot JVM

Fortsetzung: Kapitel 2 – 3.9 des Skriptes

Frage

Was sind die Roots für einen Minor GC?



Remembered Sets

Wie können wir vermeiden, dass bei einem Minor GC nicht der ganze Tenured Generation traversiert wird?

- Bemerkung: Alte Objekte referenzieren selten junge Objekte
- Remembered Set: Liste mit allen Objekten im Tenured Generation, welche Young Generation Objekte referenzieren
- Compiler inseriert Code, der beim Schreiben eines Feldes von einem Objekt ausgeführt wird → neue Objekt wird in den «Remembered Set» aufgenommen

Optimierung: Card Table

- Nicht einzelne Objekte sondern grössere Teile des Tenured Generation werden gemerkt

Beispiel: Skript Kapitel 3.8

Agenda

Diskussion Übungen

Generational GC

Paralleler und nebenläufiger GC

GC Tuning

Verbesserung Performanz

Was sind aus Applikationssicht wichtige Performanzmerkmale?

- Durchsatz
- Reaktionsfähigkeit

Ansätze

Für Steigerung des Durchsatzes: **Paralleler GC**

- GC Algorithmus parallelisiert

Für Reduzierung der Pausenzeiten : **Nebenläufiger GC**

- GC Algorithmus (oder Teile davon) läuft gleichzeitig mit dem Benutzerprogramm

Annahmen für Beispiel

Applikation

- Parallel mit zwei Threads
- Perfekt parallelisiert

Hardware

- Zwei Prozessorkernen

Serieller vs. Paralleler vs. Nebenläufiger GC

Serieller GC



Paralleler GC



Nebenläufiger GC



Zeit

Fragen

Bei welcher GC-Variante ist der Durchsatz am besten?

Bei welcher GC-Variante sind die Pausenzeiten am niedrigsten?

Individualarbeit 1: Selbststudium GC-Implementierungen

Selbststudium Paralleler und Nebenläufiger GC in Java HotSpot

- Kapitel 4 des Skriptes

HotSpot GCs: Übersicht

Serial GC

- Young Generation: Serieller Mark & Copy
- Old Generation: Serieller Mark & Compact

Parallel GC

- Young Generation: Paralleler Mark & Copy
- Old Generation: Paralleler Mark & Compact

Concurrent Mark and Sweep (CMS)

- Young Generation: Paralleler Mark & Copy
- Old Generation: Mostly Concurrent Mark & Sweep

Agenda

Diskussion Übungen SW8

Generational GC

Paralleler und nebenläufiger GC

GC Tuning

GC Tuning

Wiederholung: Was sind aus Applikationssicht wichtige Performanzmerkmale?

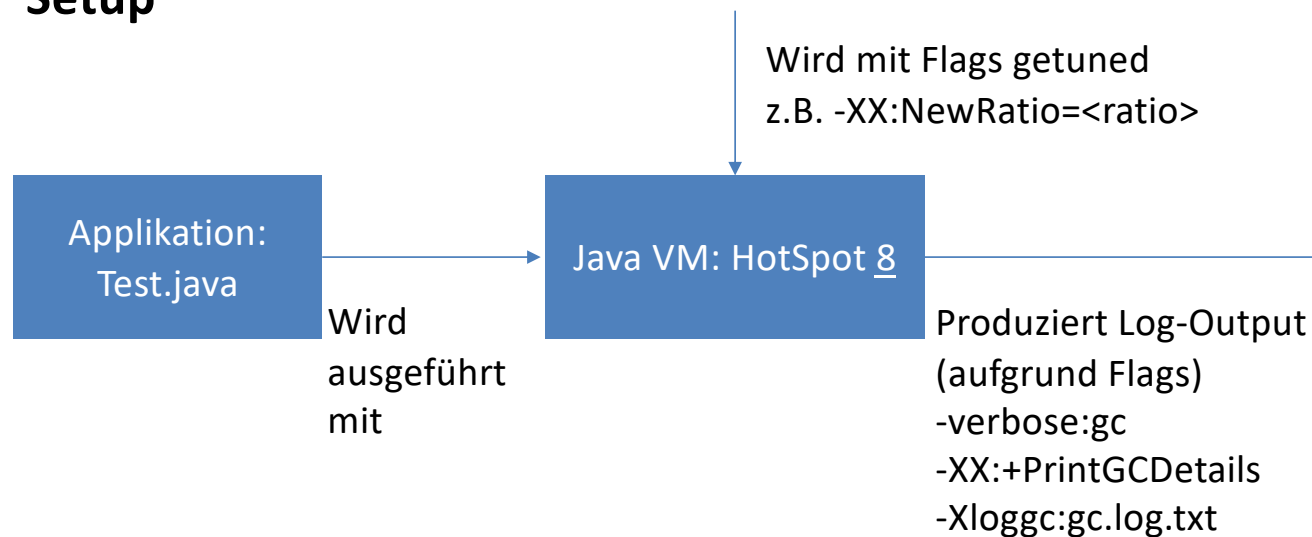
- Durchsatz
- Reaktionsfähigkeit

Auch wenn der für das Ziel entsprechende GC-Implementierung eingeschaltet wurde, ist die Performanz nicht gut genug

- Problem kann manchmal durch einen manuellen Eingriff gelöst werden = **GC Tuning**

Individualarbeit 2: GC Tuning

Setup



Beispieloutput

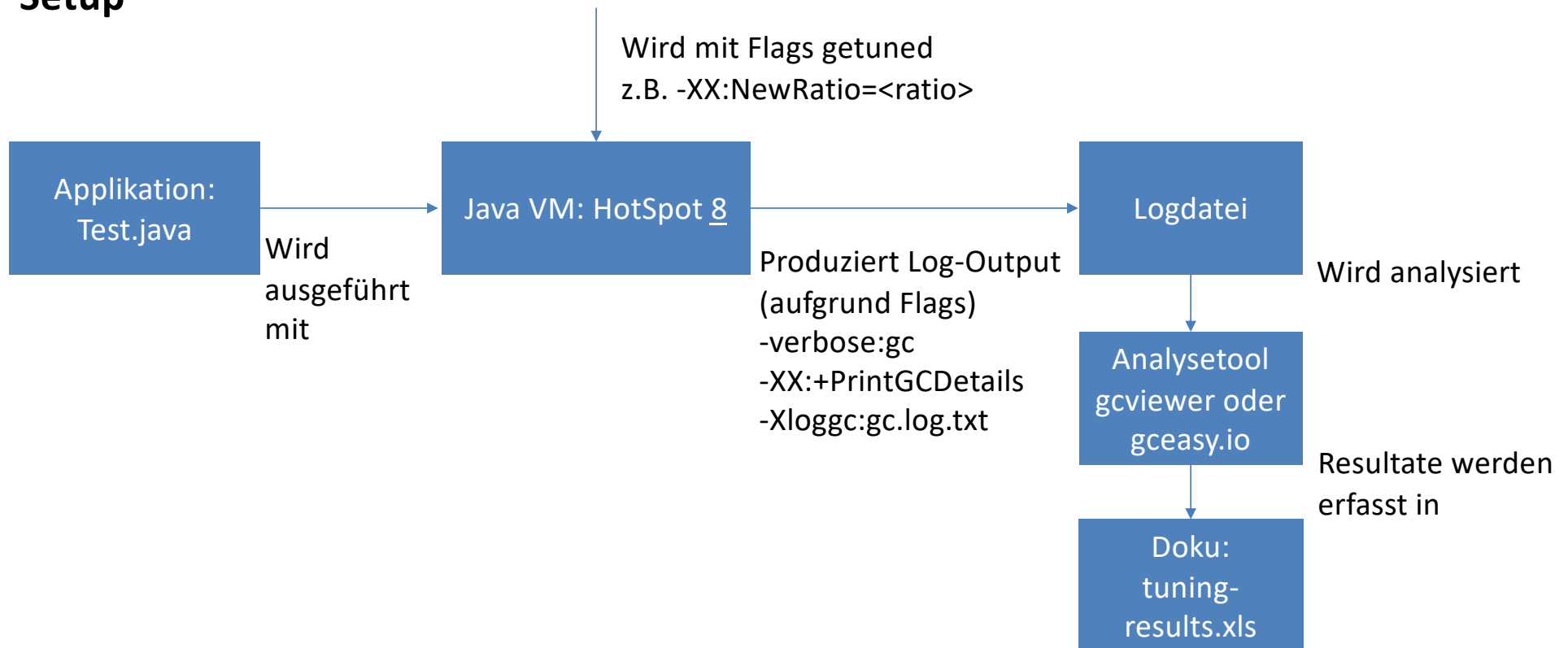
```
Java HotSpot(TM) 64-Bit Server VM (25.141-b15) for bsd-amd64 JRE (1.8.0_141-b15), built on Jul 12 2017 04:35:23 by "java_re" with gcc 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2336.11.00)
Memory: 4k page, physical 16777216k(1248276k free)
```

```
/proc/meminfo:
```

```
CommandLine flags: -XX:InitialHeapSize=33554432 -XX:MaxHeapSize=33554432 -XX:NewRatio=1 -XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:SurvivorRatio=1 -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseParallelGC
0.120: [GC (Allocation Failure) [PSYoungGen: 6143K->5106K(11264K)] 6143K->5794K(27648K), 0.0030349 secs] [Times: user=0.01 sys=0.01, real=0.00 secs]
0.167: [GC (Allocation Failure) [PSYoungGen: 11239K->3729K(11264K)] 11927K->4425K(27648K), 0.0032820 secs] [Times: user=0.00 sys=0.01, real=0.00 secs]
0.171: [GC (Allocation Failure) [PSYoungGen: 9865K->1648K(11264K)] 10561K->2344K(27648K), 0.0006056 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
0.172: [GC (Allocation Failure) [PSYoungGen: 7786K->5090K(11264K)] 8482K->8475K(27648K), 0.0022283 secs] [Times: user=0.00 sys=0.01, real=0.00 secs]
0.218: [GC (Allocation Failure) [PSYoungGen: 11229K->5106K(11264K)] 14615K->9020K(27648K), 0.0012928 secs] [Times: user=0.01 sys=0.01, real=0.00 secs]
0.267: [GC (Allocation Failure) [PSYoungGen: 11246K->3553K(11264K)] 15160K->7467K(27648K), 0.0011604 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
0.270: [GC (Allocation Failure) [PSYoungGen: 9693K->1152K(11264K)] 13607K->5346K(27648K), 0.0009220 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
0.272: [GC (Allocation Failure) [PSYoungGen: 7293K->5090K(11264K)] 11487K->11469K(27648K), 0.0019212 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
0.316: [GC (Allocation Failure) [PSYoungGen: 11231K->5090K(11264K)] 17610K->11621K(27648K), 0.0013460 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
0.370: [GC (Allocation Failure) [PSYoungGen: 11216K->2080K(11264K)] 17747K->8628K(27648K), 0.0006940 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
0.372: [GC (Allocation Failure) [PSYoungGen: 8222K->5090K(11264K)] 14769K->14671K(27648K), 0.0026775 secs] [Times: user=0.01 sys=0.01, real=0.00 secs]
0.375: [GC (Allocation Failure) [PSYoungGen: 11231K->5090K(11264K)] 20812K->15663K(27648K), 0.0014235 secs] [Times: user=0.01 sys=0.01, real=0.00 secs]
0.419: [GC (Allocation Failure) [PSYoungGen: 11231K->4001K(11264K)] 21804K->14590K(27648K), 0.0009612 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
0.467: [GC (Allocation Failure) [PSYoungGen: 10143K->1920K(11264K)] 20732K->12525K(27648K), 0.0006870 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
0.468: [GC (Allocation Failure) [PSYoungGen: 8062K->5090K(11264K)] 18667K->18592K(27648K), 0.0026388 secs] [Times: user=0.00 sys=0.01, real=0.00 secs]
0.471: [Full GC (Ergonomics) [PSYoungGen: 5090K->0K(11264K)] [ParOldGen: 13502K->8264K(16384K)] 18592K->8264K(27648K), [Metaspace: 2682K->2682K(1056768K)], 0.0040828 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]
```

Individualarbeit 2: GC Tuning (Forts.)

Setup



Bemerkungen

JVisualVM ist für die Übung nicht nötig

- Analyse der generierten Logdateien ist ausreichend

Ziele des Tunings

- Durchsatz → Übung 1
- Reaktionsfähigkeit → Übung 2

Empfohlene Bibliografie (zum Thema GC)

1. Oracle Corp.:

HotSpot Virtual Machine Garbage Collection Tuning Guide

- Praktische Aspekte der GCs in der Oracle JVM

2. Aho et al.: **Compilers: Prinzipien, Techniken und Werkzeuge ('08)**

- Für ein besseres Verständnis der GC Mechanismen allgemein
- Kapitel 7.4-7.8
- In der FHNW-Bibliothek in Brugg verfügbar
- **Englische Version:** Compilers: Principles, techniques and tools ('07)

Zusammenfassung

Generational GC

Paralleler und nebenläufiger GC

GC Tuning

Individualarbeit

Selbststudium Paralleler und Nebenläufiger GC in Java HotSpot

- Kapitel 4 des Skriptes

GC Tuning

- Siehe Übungsblatt