# Application Performance Management

## Frühling 2021

## Class Loading

*Zoltán Majó*

# Agenda

**Diskussion Übung und Finalisierung Java NIO**

**Ergänzungen GC**

**Besprechung Evaluation**

**Class Loading**

# Arbeitsblatt

**Neben `position`, `limit` und `capacity` gibt es noch die Variable `mark`. Wozu dient sie und wie wird sie manipuliert? (Klasse Buffer)**

**Antwort: Mark markiert die unterste Position im Buffer die noch bearbeitet werden soll. Es gilt immer:**

```
0 <= mark <= position <= limit <= capacity
```

# Arbeitsblatt

**Frage 2: Was versteht man unter *gathering*? (GatheringByteChannel)**

**Antwort: Die Operation wenn mehrere Buffer auf einen Channel geschrieben werden heisst Gathering. Dabei wird der Inhalt der Buffer in ihrer Reihenfolge auf den Kanal geschrieben. Es ist zum Beachten, dass die einzelnen Buffer unterschiedliche Grössen haben können.**

# Arbeitsblatt

**Frage 3: Was versteht man unter *scattering*? (ScatteringByteChannel)**

**Antwort: Das ist das Gegenteil von gathering. Daten aus einem Kanal werden im mehrere Buffer geschrieben. Ist der erste Buffer voll, so wird der zweite gefüllt, bis dieser voll ist etc.**

# Arbeitsblatt

**Frage 4: Was ist ein *direct buffer*? Wozu dient er? (ByteBuffer)**

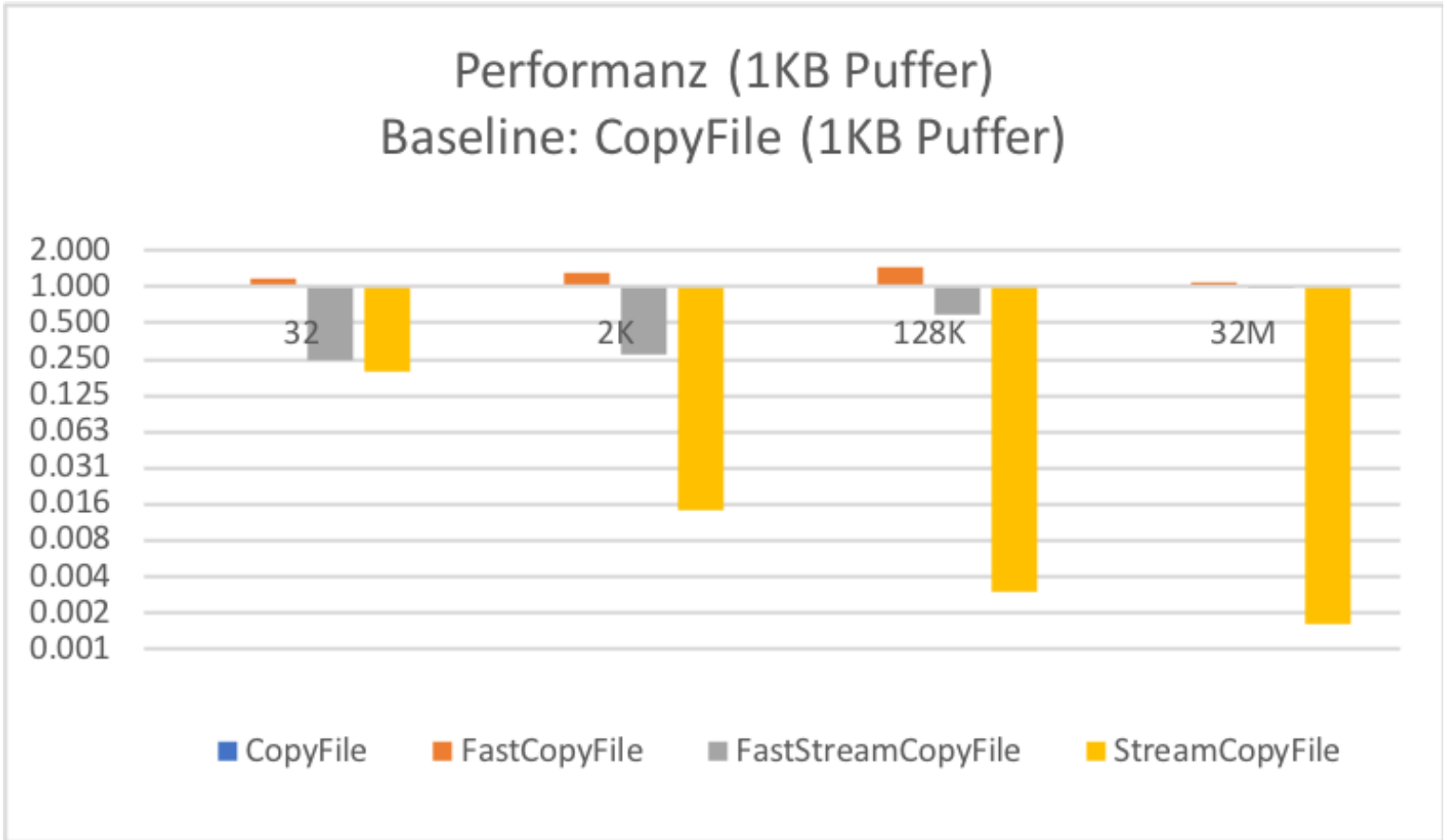
**Antwort: Direct Buffers werden meist vom Betriebssystem verwaltet und deren Speicher kann sich auch ausserhalb der JVM befinden.**

# Arbeitsblatt

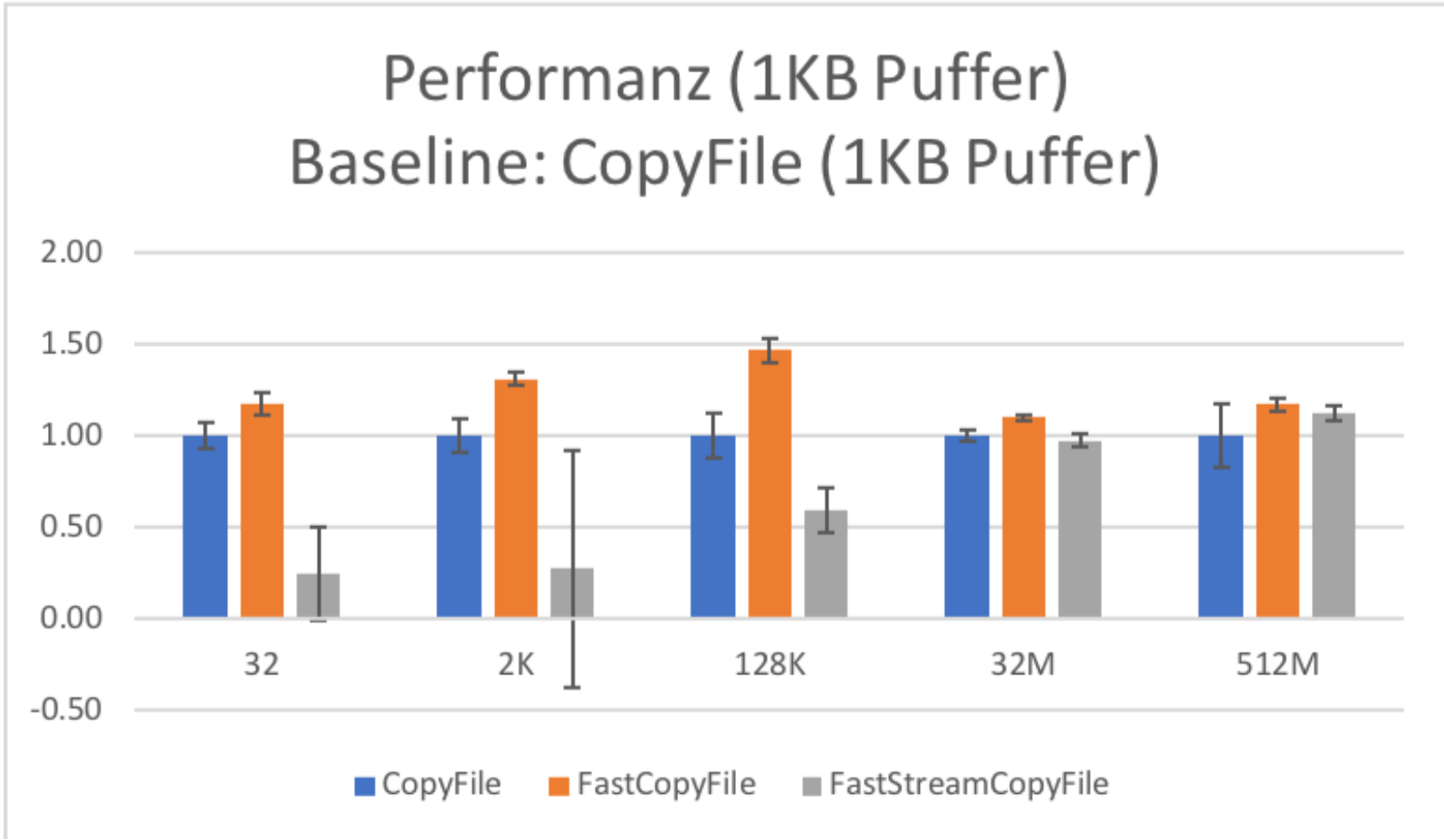
**Frage 5: Wozu dient *buffer slicing*? (slice() in ByteBuffer)**

**Antwort: Damit kann ein Teil des Buffers wiederum als Buffer verwendet werden. Die Daten – der darunterliegende Byte-Array – werden aber von beiden Buffer geteilt.**

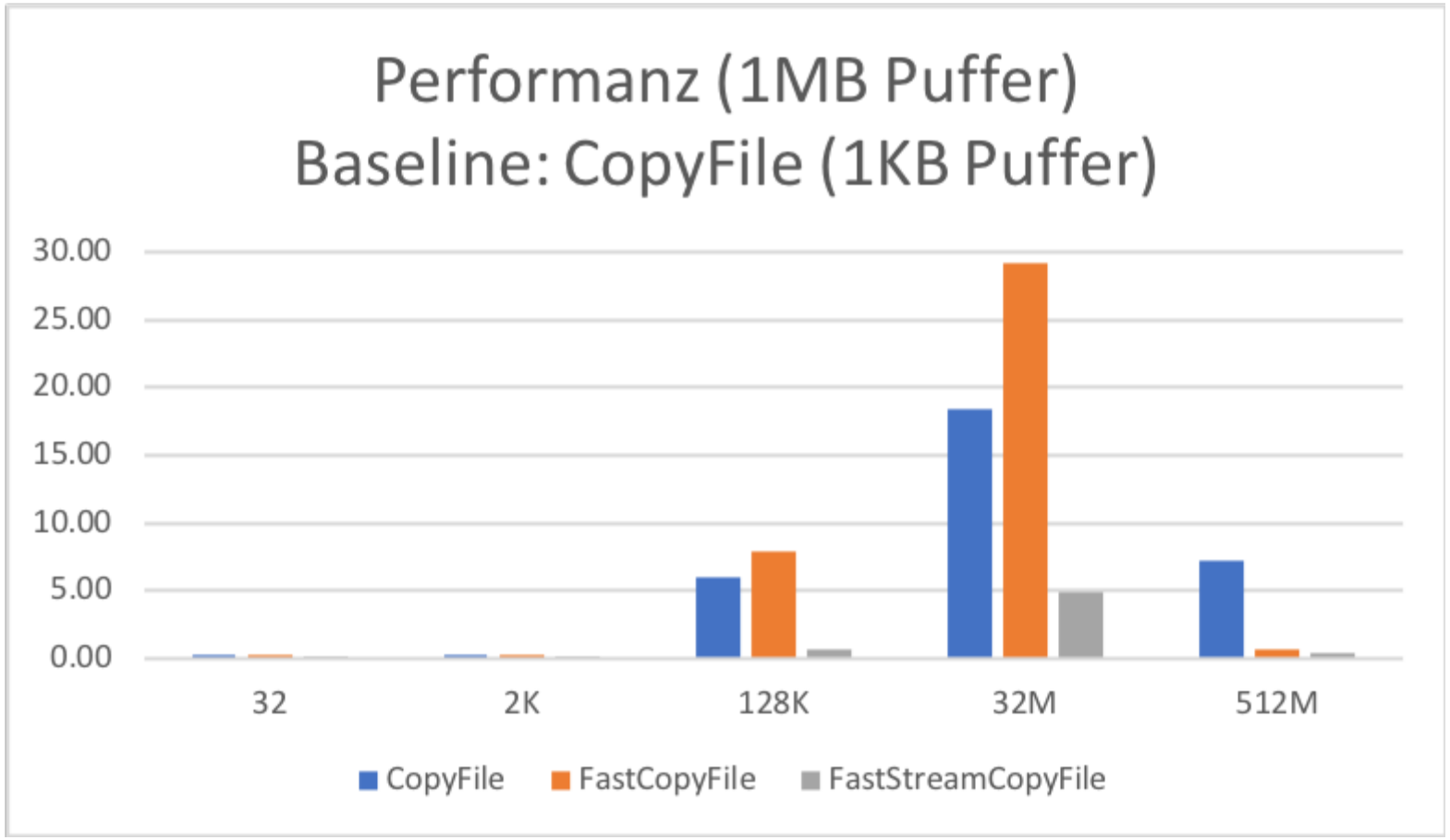
# Frage 7


Performanz (1KB Puffer)
Baseline: CopyFile (1KB Puffer)
2.000
1.000
0.500
0.250
0.125
0.063
0.031
0.016
0.008
0.004
0.002
0.001
32
2K
128K
32M
CopyFile
FastCopyFile
FastStreamCopyFile
StreamCopyFile

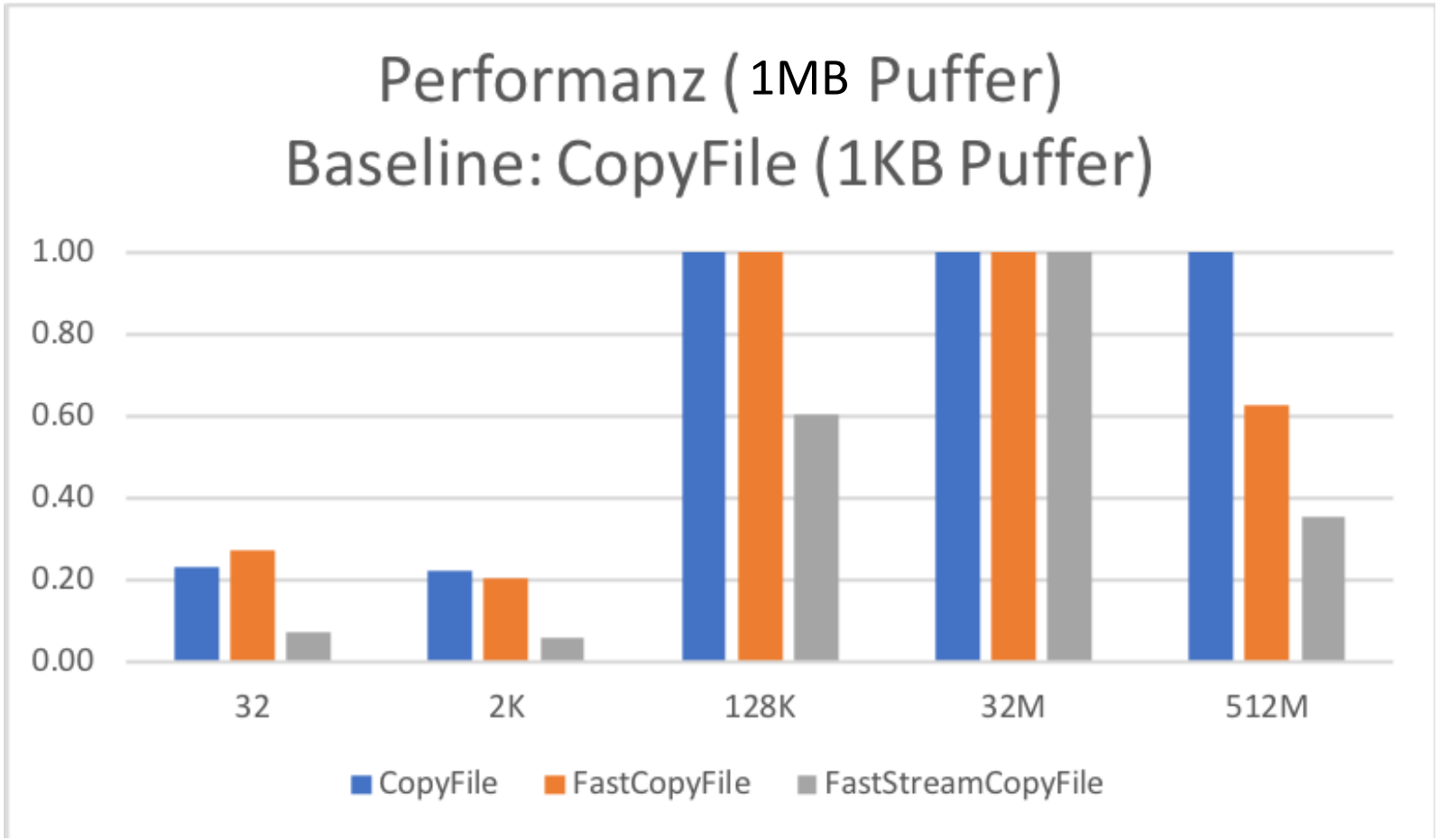
# Frage 7


Performanz (1KB Puffer)
Baseline: CopyFile (1KB Puffer)
2.00
1.50
1.00
0.50
0.00
-0.50
32
2K
128K
32M
512M
CopyFile
FastCopyFile
FastStreamCopyFile

# Frage 7


Performanz (1MB Puffer)
Baseline: CopyFile (1KB Puffer)
30.00
25.00
20.00
15.00
10.00
5.00
0.00
32
2K
128K
32M
512M
CopyFile
FastCopyFile
FastStreamCopyFile

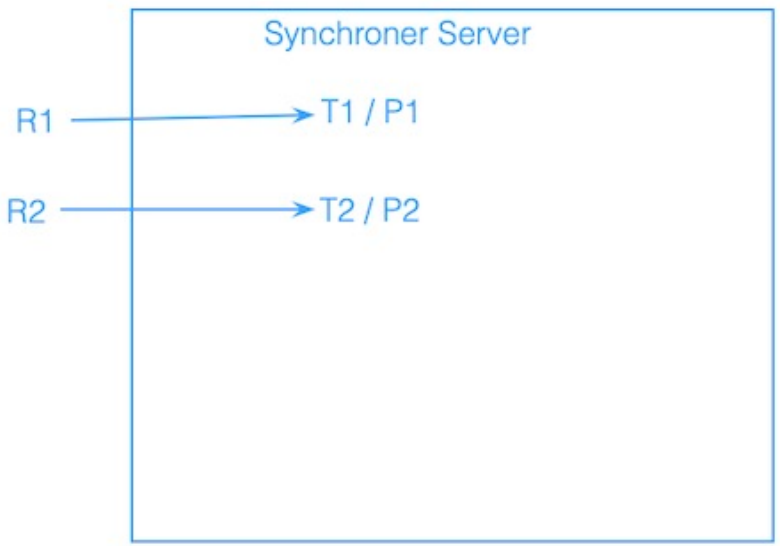
# Frage 7

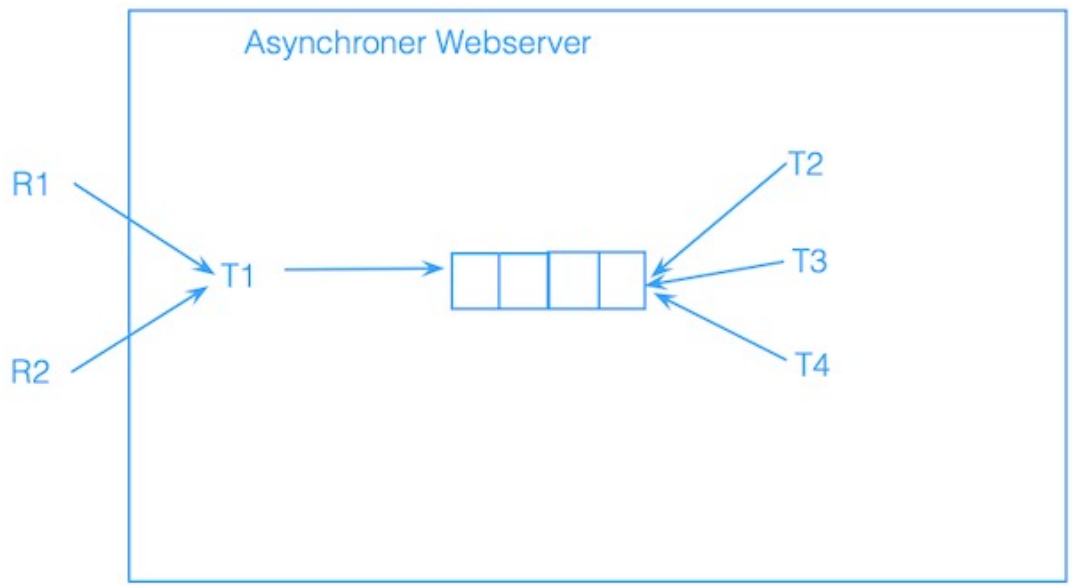

Performanz ( 1MB Puffer)
Baseline: CopyFile (1KB Puffer)
1.00
0.80
0.60
0.40
0.20
0.00
32
2K
128K
32M
512M
CopyFile
FastCopyFile
FastStreamCopyFile

Tradeoff zwischen Schnelligkeit und Skalierbarkeit bzw. zwischen Code-Komplexität

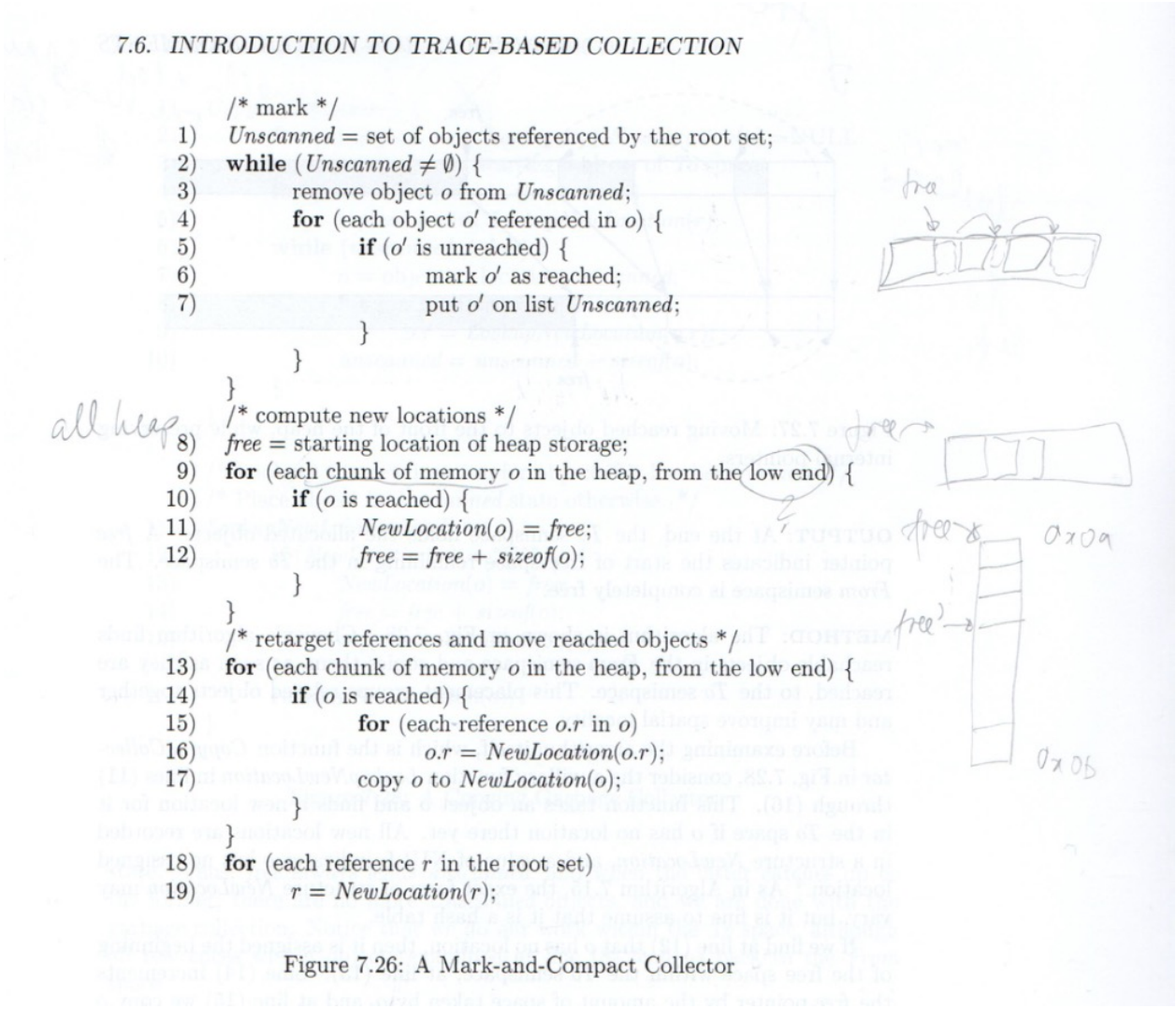
Wie schnell wird ein Request abgearbeitet?
- Synchrone Variante wahrscheinlich besser, denn es gibt keinen Kontextwechsel zwischen Threads / Prozesse, d.h. jede Anfrage hat ihren eigenen Thread / Prozess
- hier müsste man gute Datenstrukturen verwenden, die nicht blockieren (bei Operationen auf Shared Memory) -> Auswahl und Verständnis nicht trivial

New I/O skaliert gut, denn er asynchron ist, d.h. man muss nicht pro Verbindung einen neuen Thread / Prozess kreieren



# Ergänzung 1: Algorithmus Mark and Compact



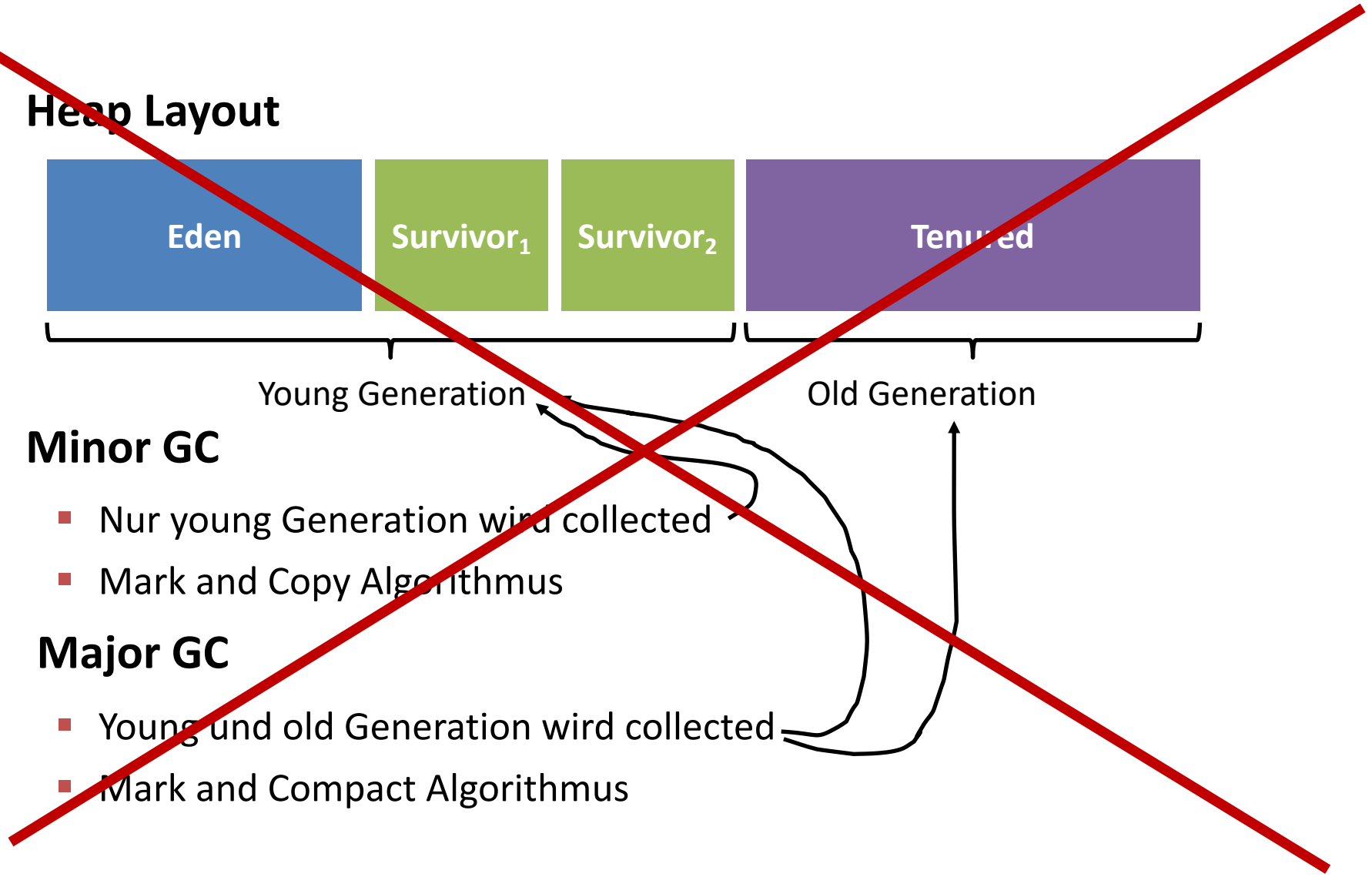
7.6. INTRODUCTION TO TRACE-BASED COLLECTION

```
     /* mark */
1)   Unscanned = set of objects referenced by the root set;
2)   while (Unscanned ≠ ∅) {
3)       remove object o from Unscanned;
4)       for (each object o' referenced in o) {
5)           if (o' is unreached) {
6)               mark o' as reached;
7)               put o' on list Unscanned;
             }
         }
     }
     /* compute new locations */
8)   free = starting location of heap storage;
9)   for (each chunk of memory o in the heap, from the low end) {
10)      if (o is reached) {
11)          NewLocation(o) = free;
12)          free = free + sizeof(o);
         }
     }
     /* retarget references and move reached objects */
13)  for (each chunk of memory o in the heap, from the low end) {
14)      if (o is reached) {
15)          for (each reference o.r in o)
16)              o.r = NewLocation(o.r);
17)          copy o to NewLocation(o);
         }
     }
18)  for (each reference r in the root set)
19)      r = NewLocation(r);
```

Figure 7.26: A Mark-and-Compact Collector

Quelle: Aho et al. Compilers: Principels, Techniques, and Tools, 2nd Edition

# Ergänzung 2: Minor / Major / Full GC



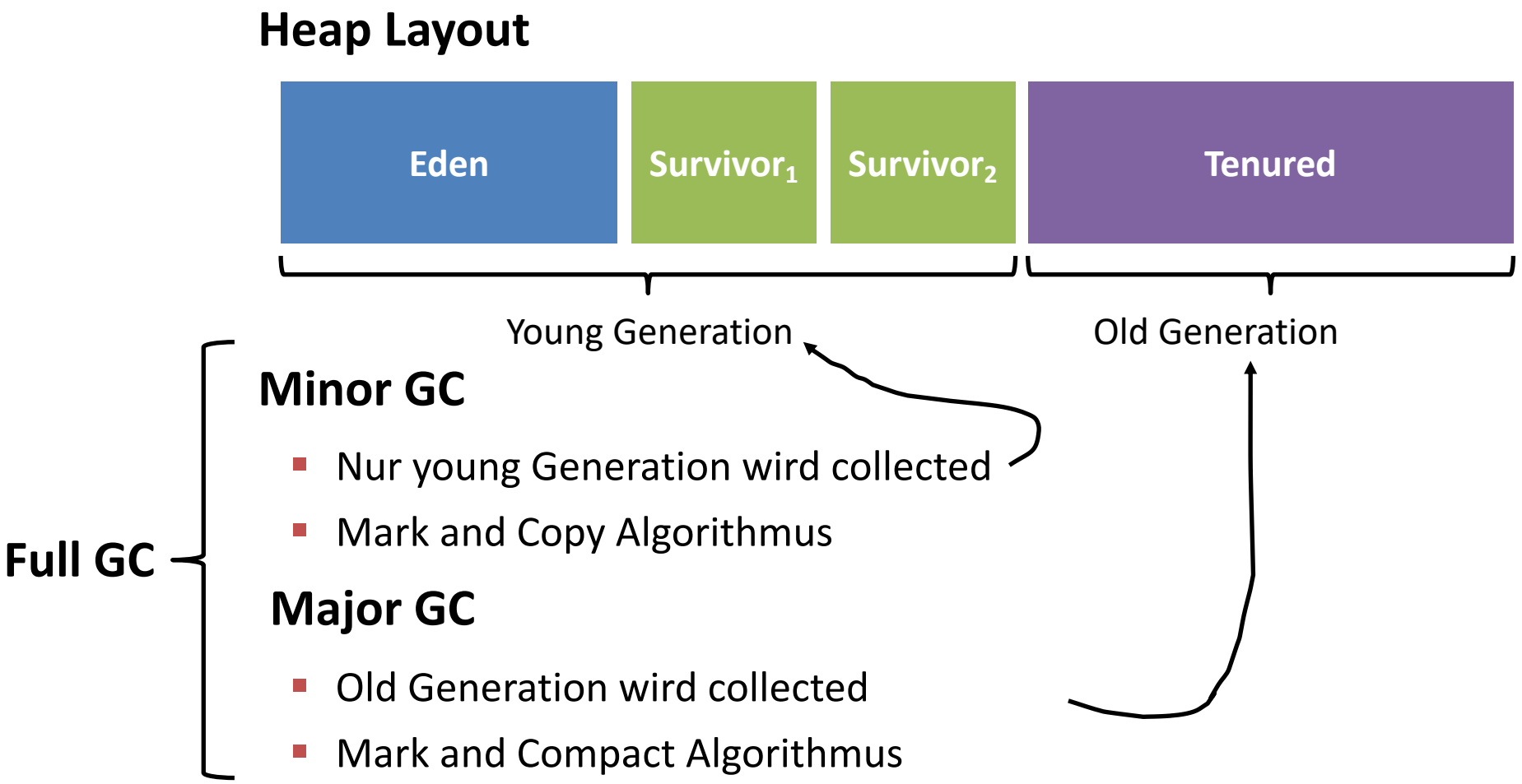

## Minor GC

- Nur young Generation wird collected
- Mark and Copy Algorithmus

## Major GC

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

# Ergänzung 2: Minor / Major / Full GC



# Besprechung Evaluation

# Class Loading

**Script**