New Java performance developments: compilation and garbage collection

Jeroen Borgers
@jborgers

#jfall17

Part 1: New in Java compilation

Part 2: New in Java garbage collection

Part 1
New in Java compilation

New in Java compilation

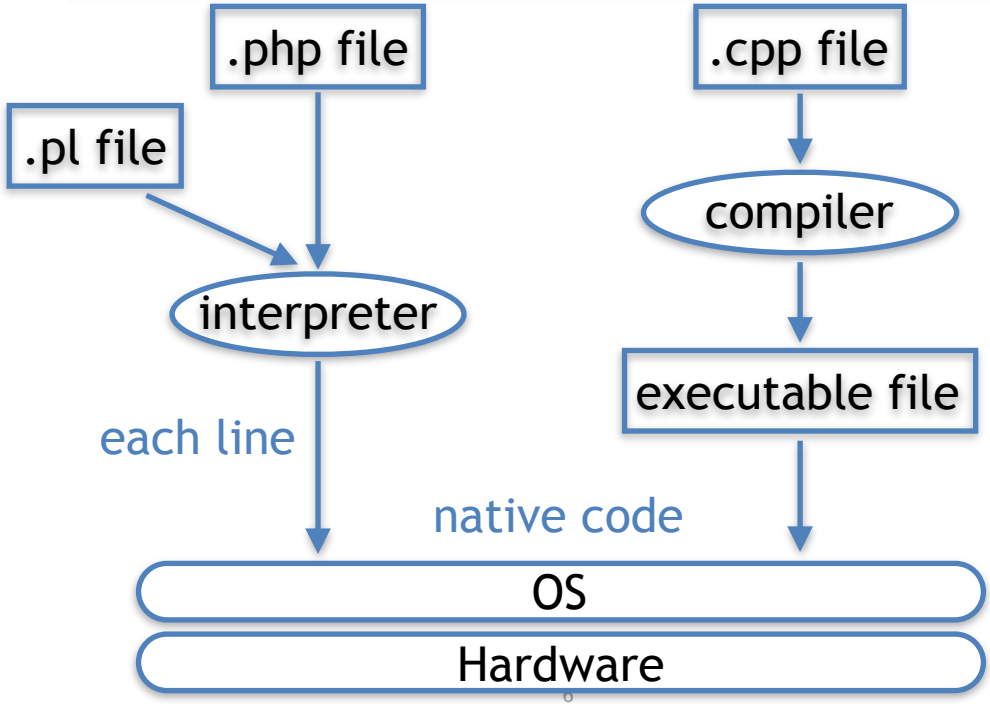
- Compilation basics for the JVM
- AOT-compilation added
- Advantages of AOT-compilation
- Examples using AOT
- Current limitations
- Conclusions

#jfall17

Java compilation basics

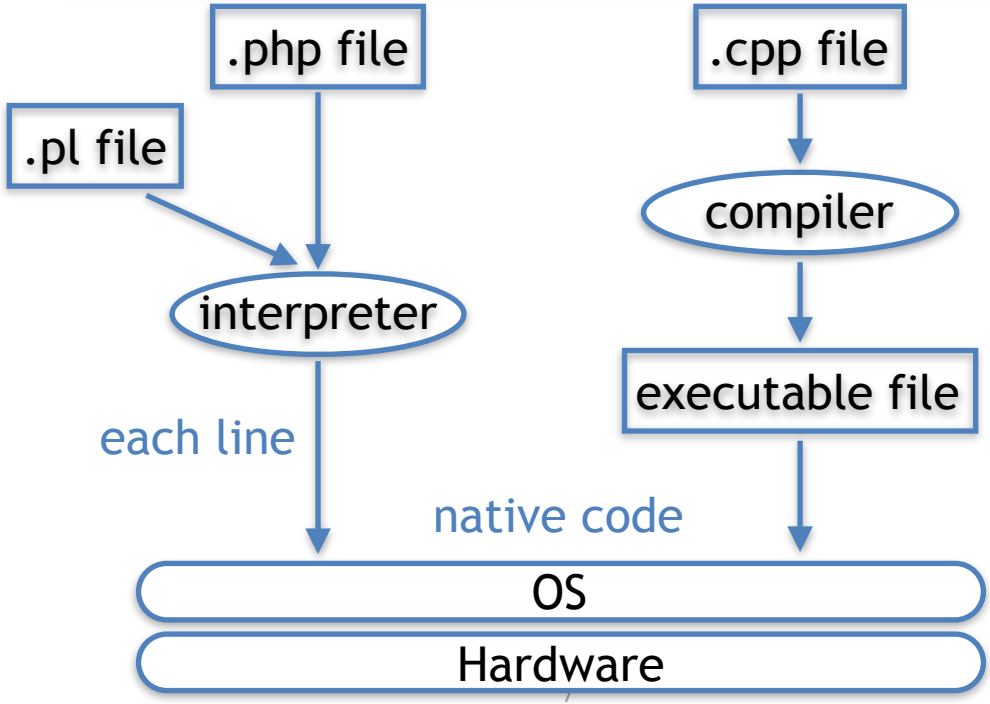
interpretation and JIT-compilation in the JVM

Interpretation versus compilation



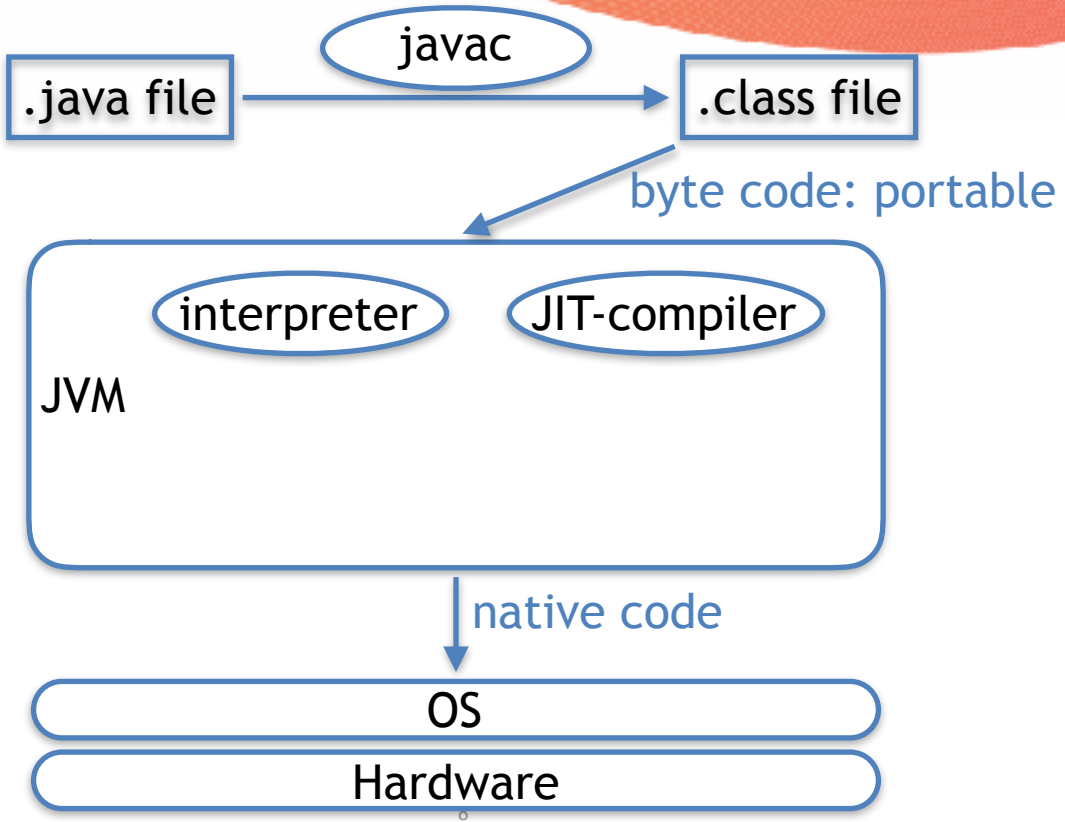
Interpretation versus compilation

- portable source code

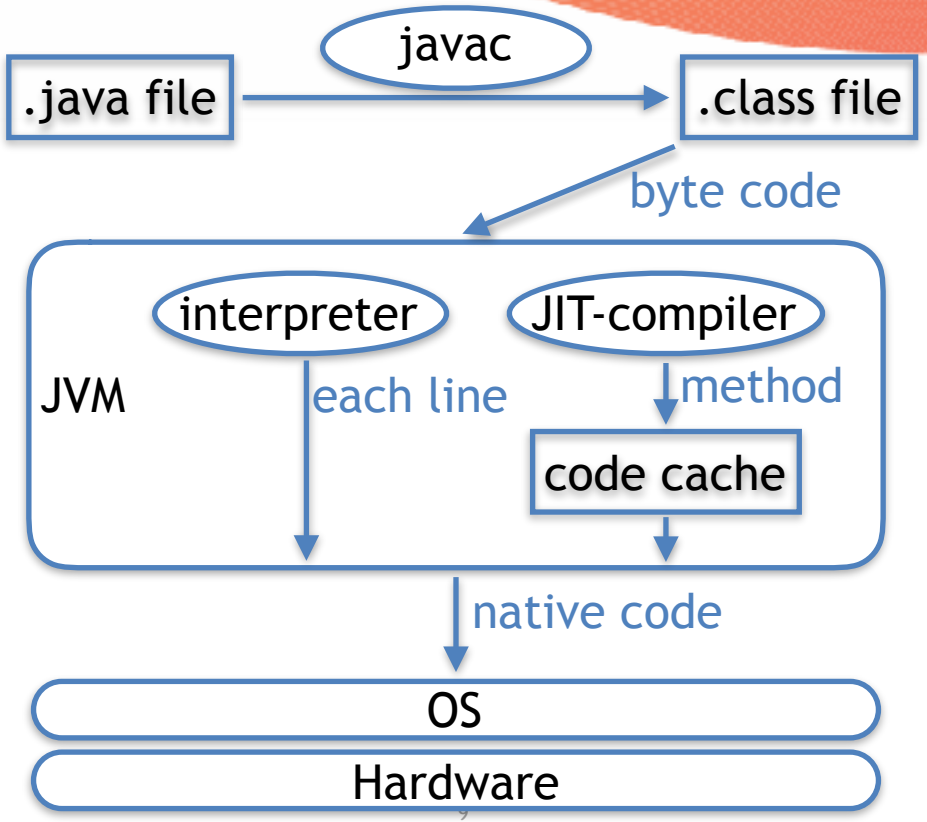


- exe non-portable
- optimizations
- faster execution

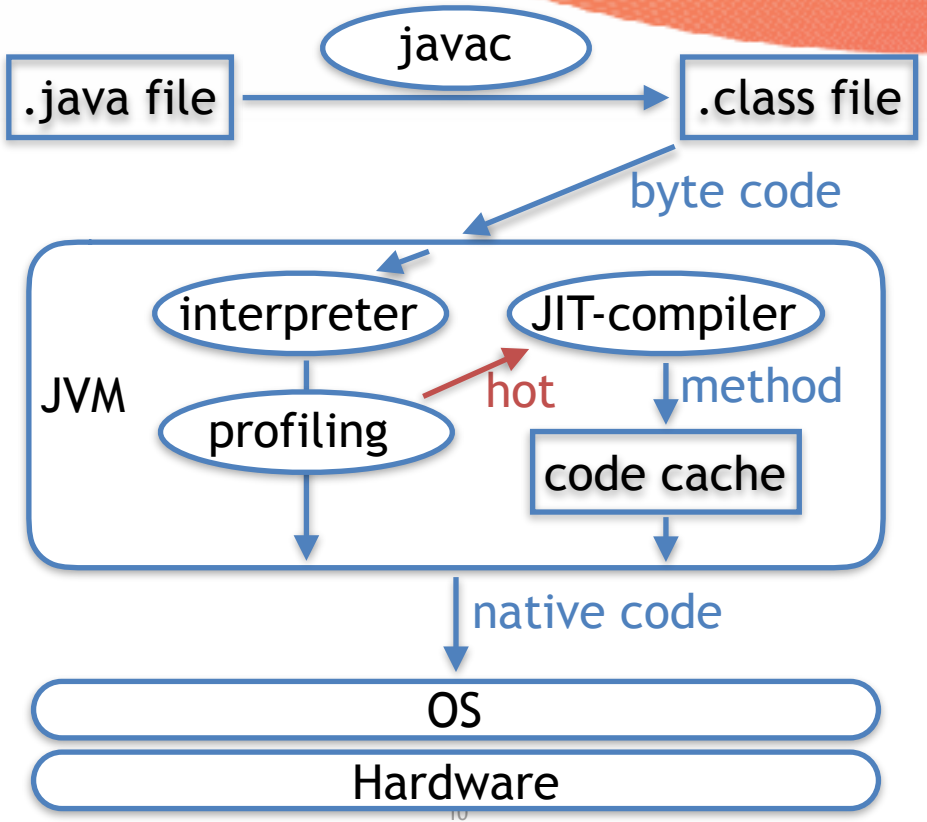
JVM: Interpretation and JIT-compilation



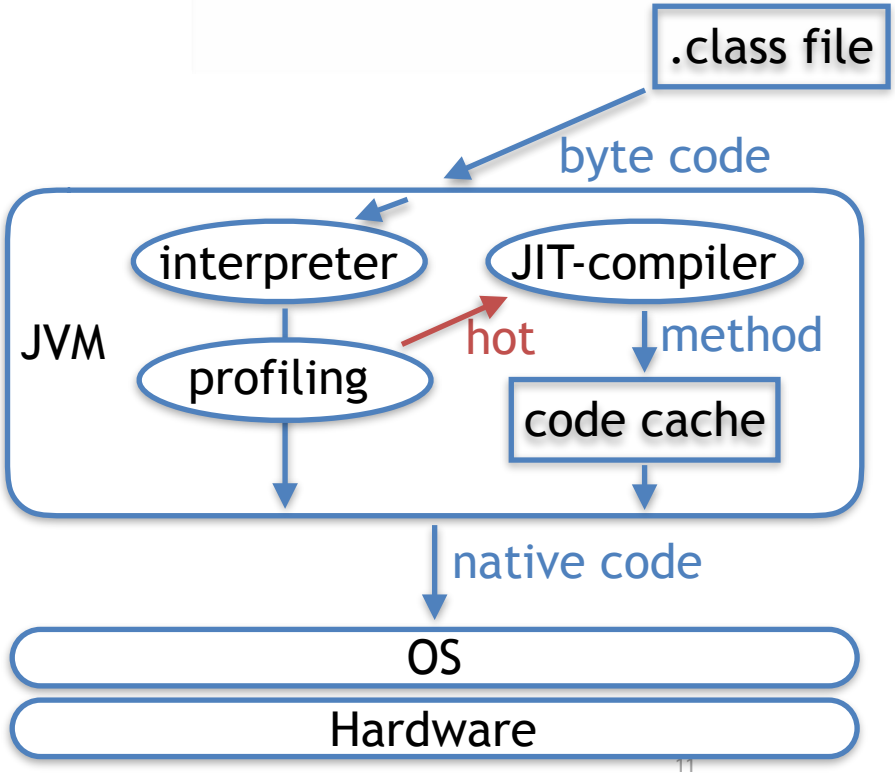
JVM: Interpretation and JIT-compilation



Profile guided JIT-compilation

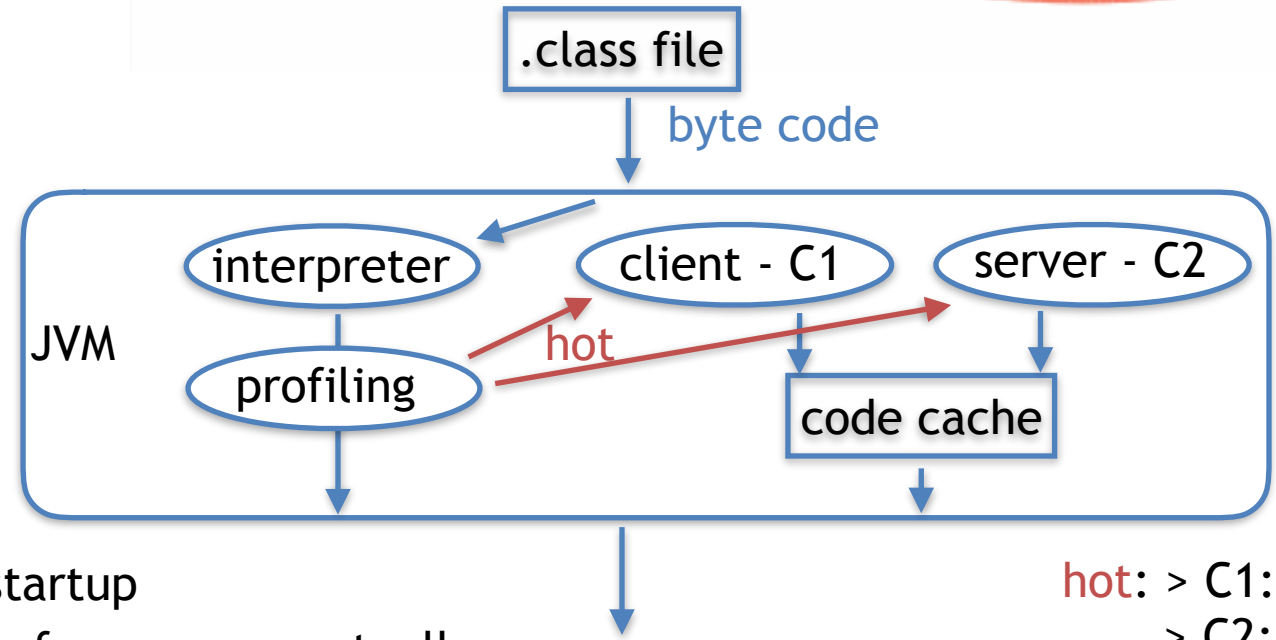


Profile guided JIT-compilation with adaptive optimizations



- focus on hot code:
- speculatively optimize
- method inlining
- branch prediction
- loop unrolling
- de-optimization

C1: client or C2: server compiler

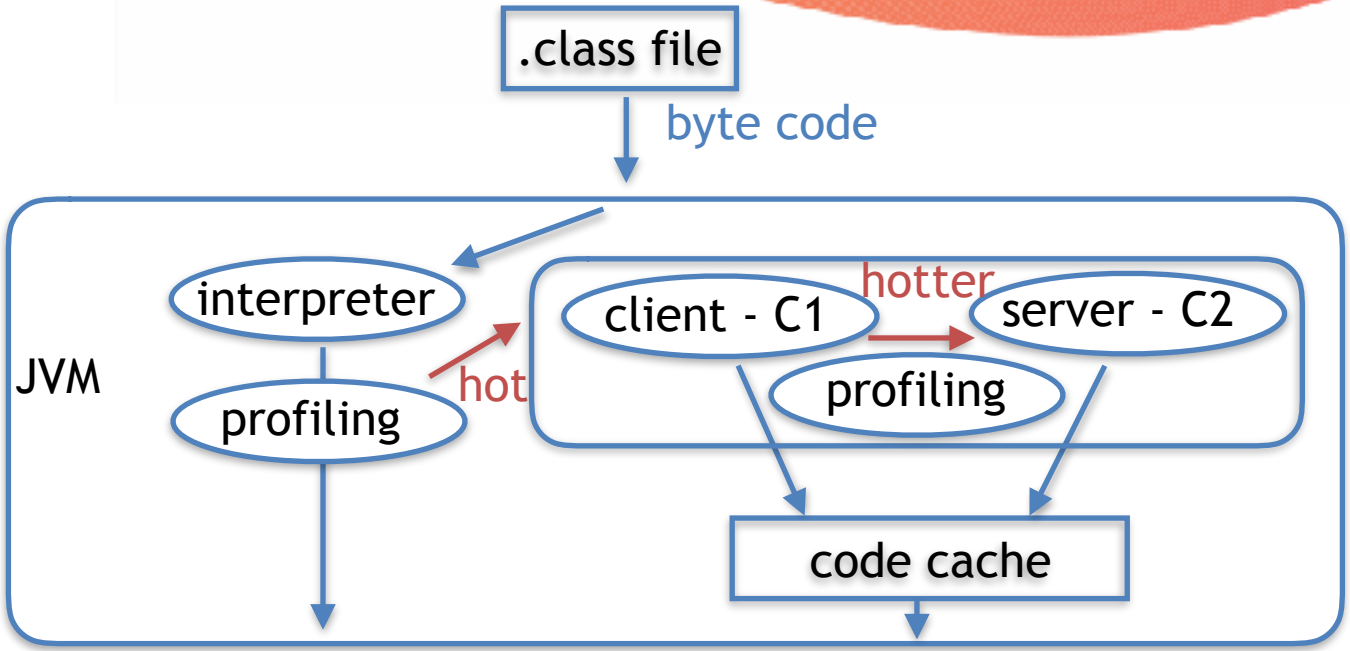


C1: quick startup

C2: best performance eventually by more optimizations

hot: > C1:1500
> C2:10.000 invocations
+ iterations

Tiered compilation: Best of C1 and C2 default in Java 8



C1: quick startup

C2: best performance eventually
by more optimizations

interpretation versus compilation flags

- -Xint
- -Xcomp
- -Xmixed
  - JIT-compilation
  - default

Java 9 test code: HelloJUG

```
package com.jpinpoint.jfall;

import java.util.List;
import java.util.Set;

public class HelloJUG {
    public static void main(String[] args) {
        System.out.println("Hello JUG!");
        System.out.println("Speakers dinner desert: " + List.of( e1: "ice cream",  e2: "chocolate",  e3: "cake"));
        System.out.println("Speakers: " + Set.of("Mark", "Ray", "Arun", "Sander", "Roy", "Jeroen"));
    }
}
```

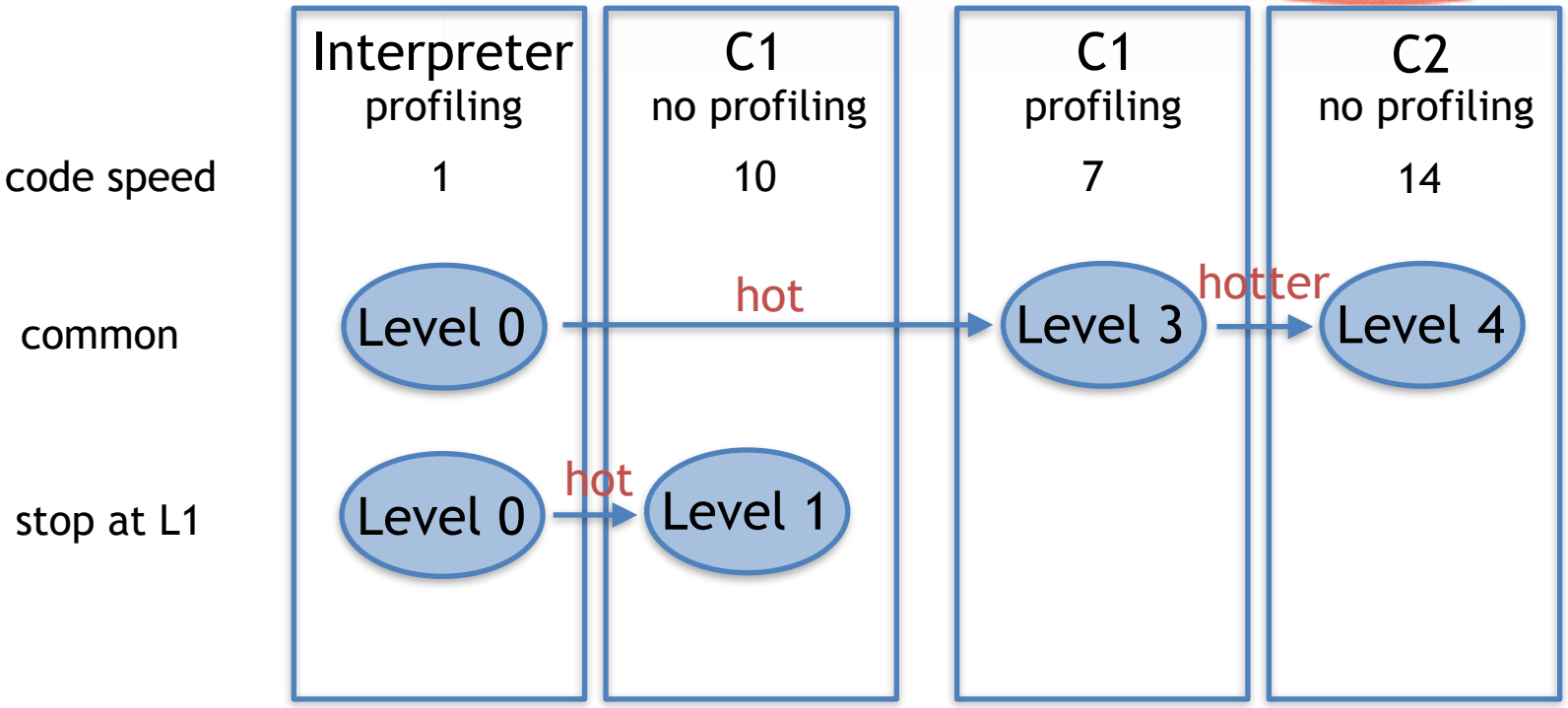
HelloJUG
Which is quickest?

java HelloJUG	
java -Xcomp HelloJUG	
java -Xint HelloJUG	

HelloJUG
Real times, average of 5 runs

java HelloJUG	252 ms
java -Xcomp …	2272 ms
java -Xint …	**214** ms

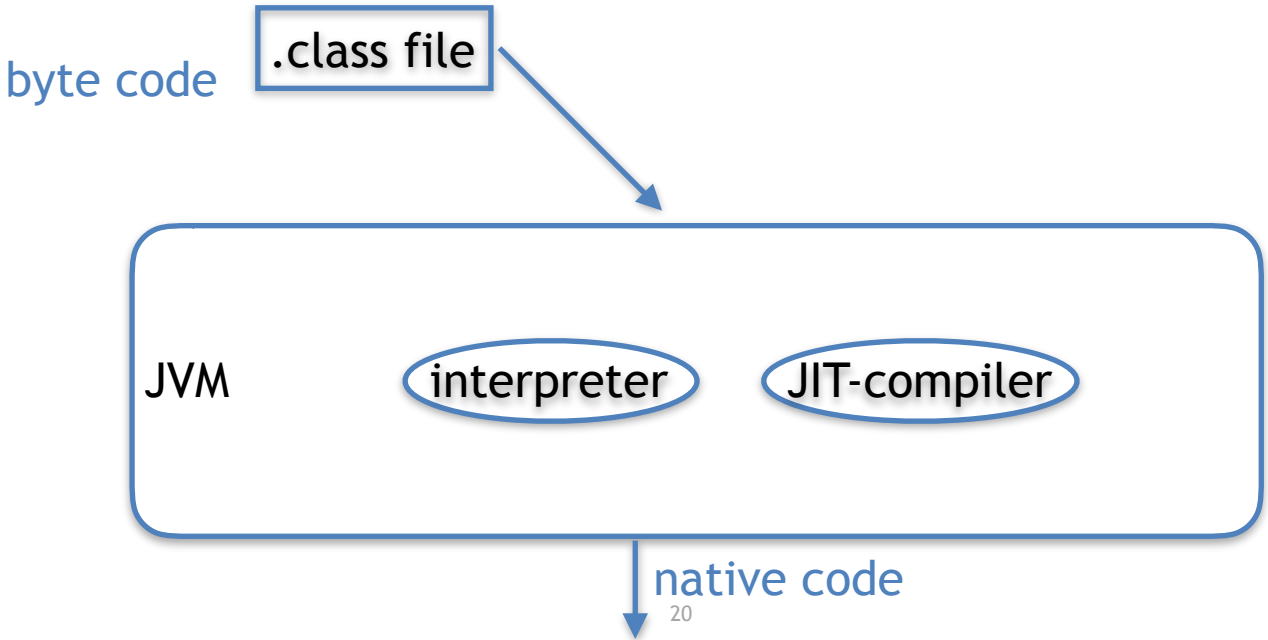
Tiered compilation levels



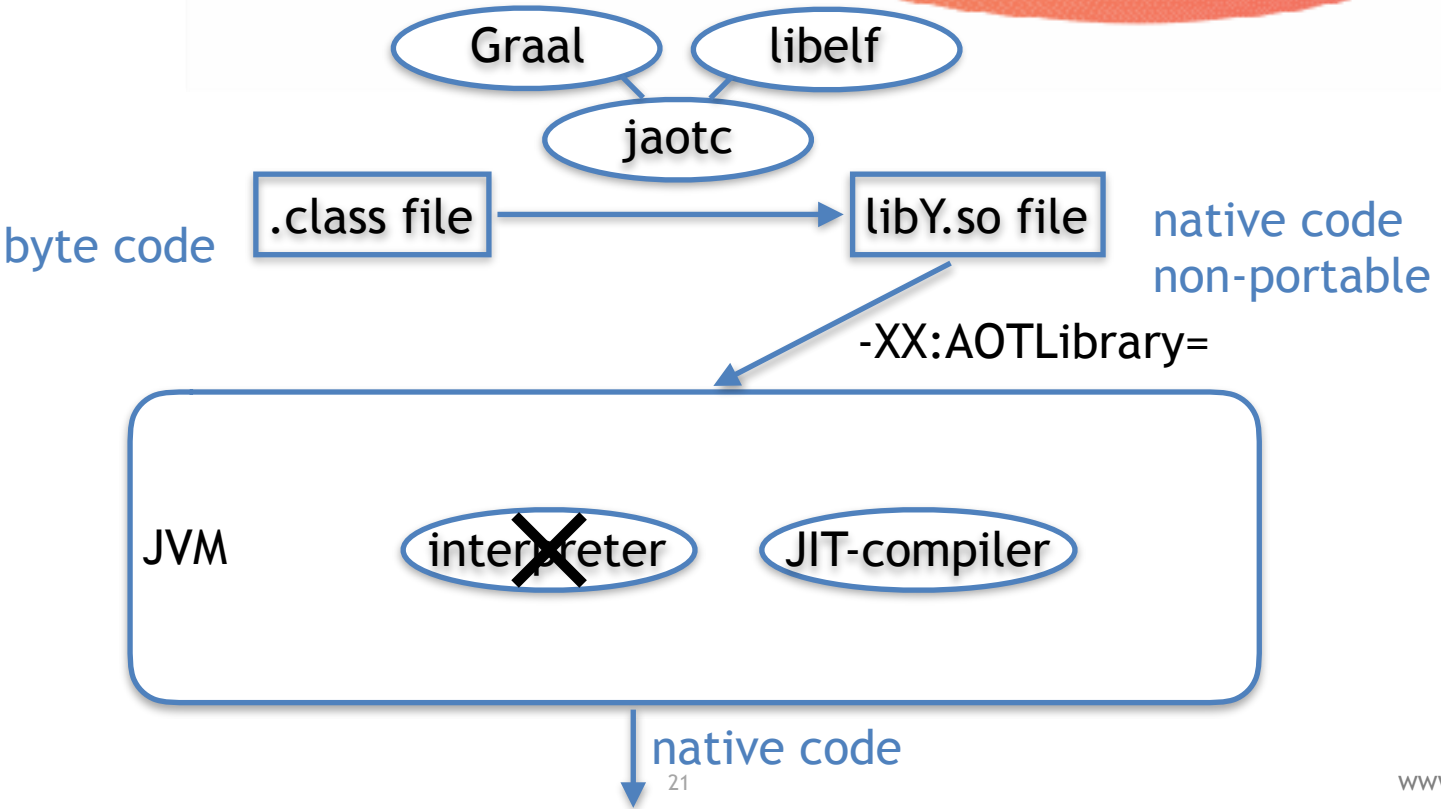
HelloJUG
Real times, average of 5 runs

java HelloJUG	252 ms
java -Xcomp …	2272 ms
java -Xint …	214 ms
java -XX:TieredStopAtLevel=1…	244 ms

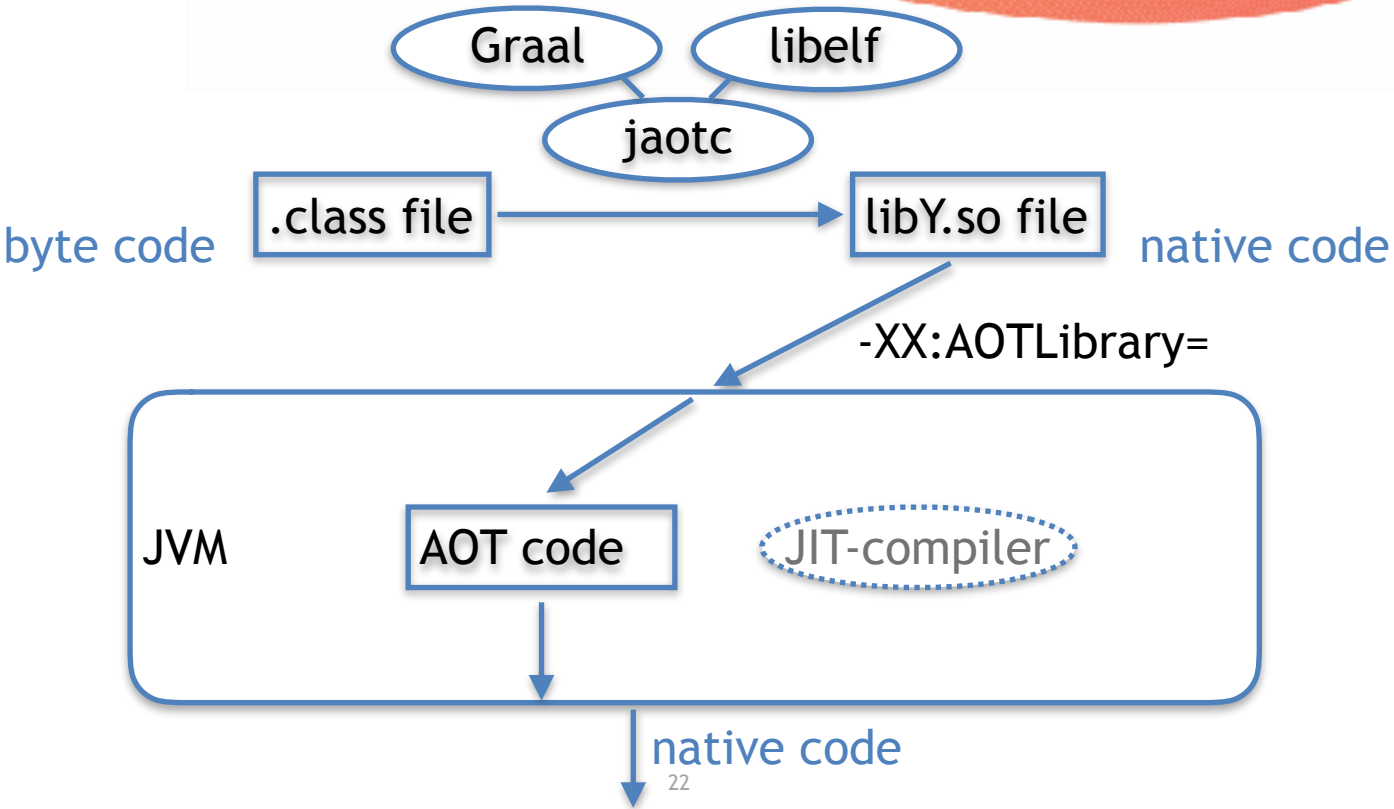
AOT-compilation added



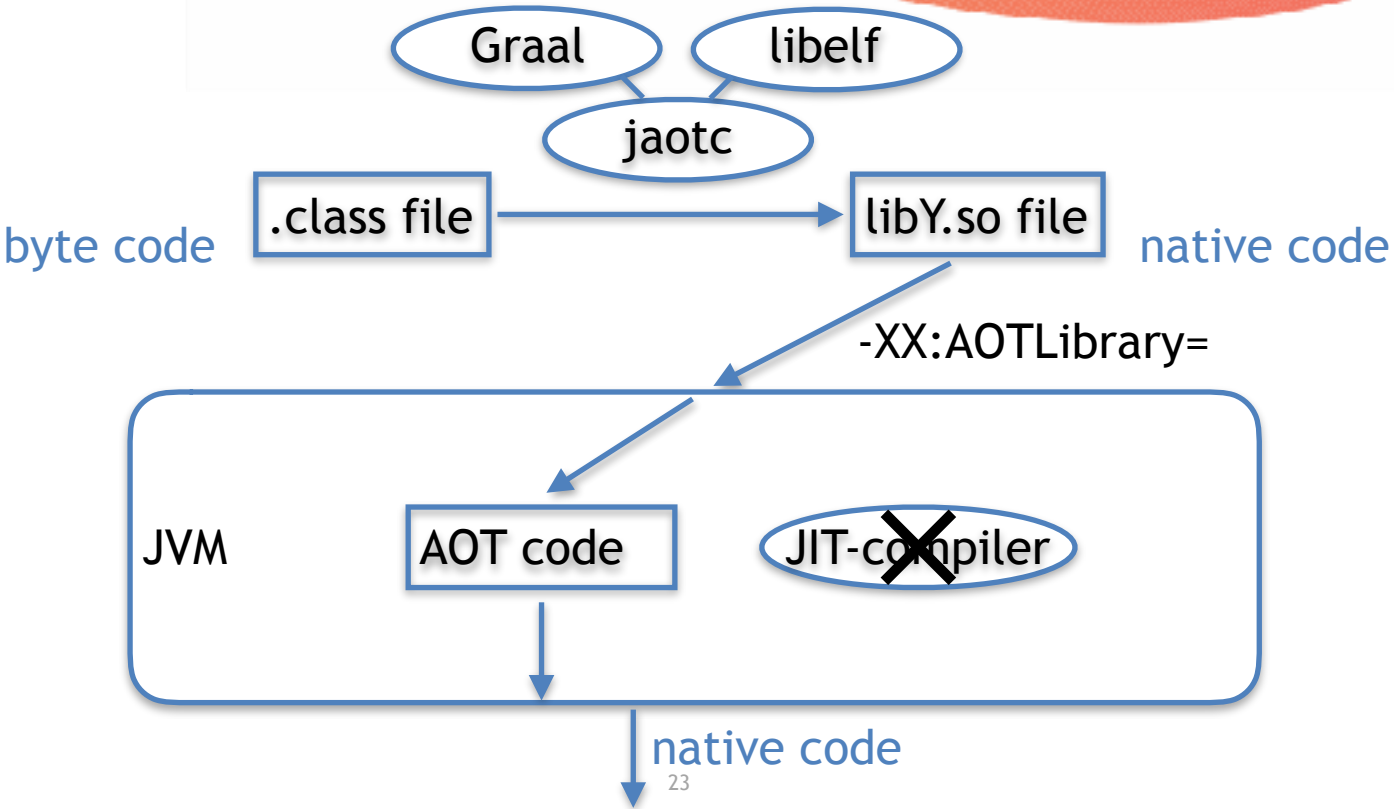
AOT-compilation added



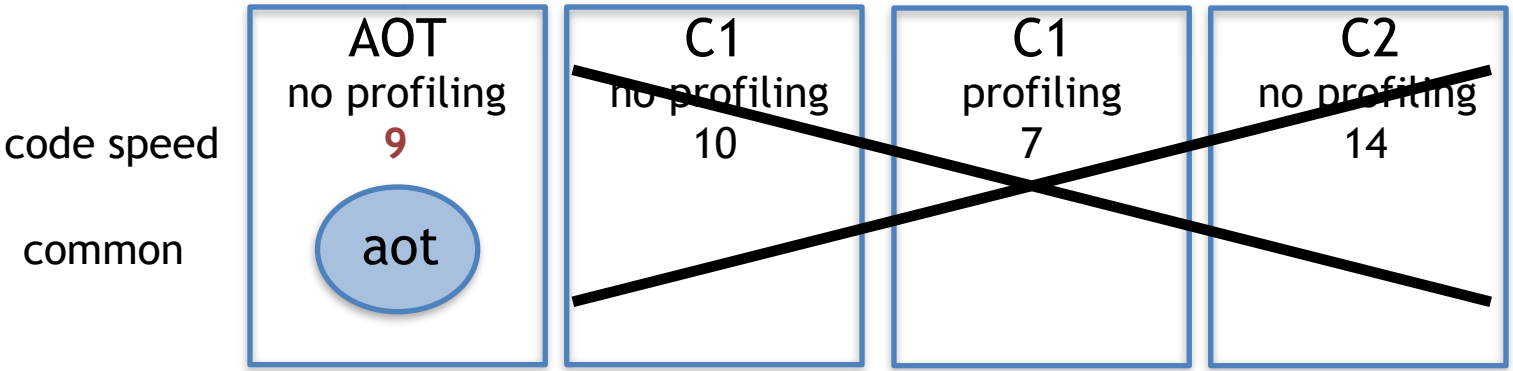
AOT in 2 flavors: tiered (with JIT) and non-tiered (no JIT)



Non-tiered AOT-compilation: no JIT



Non-tiered compilation with AOT

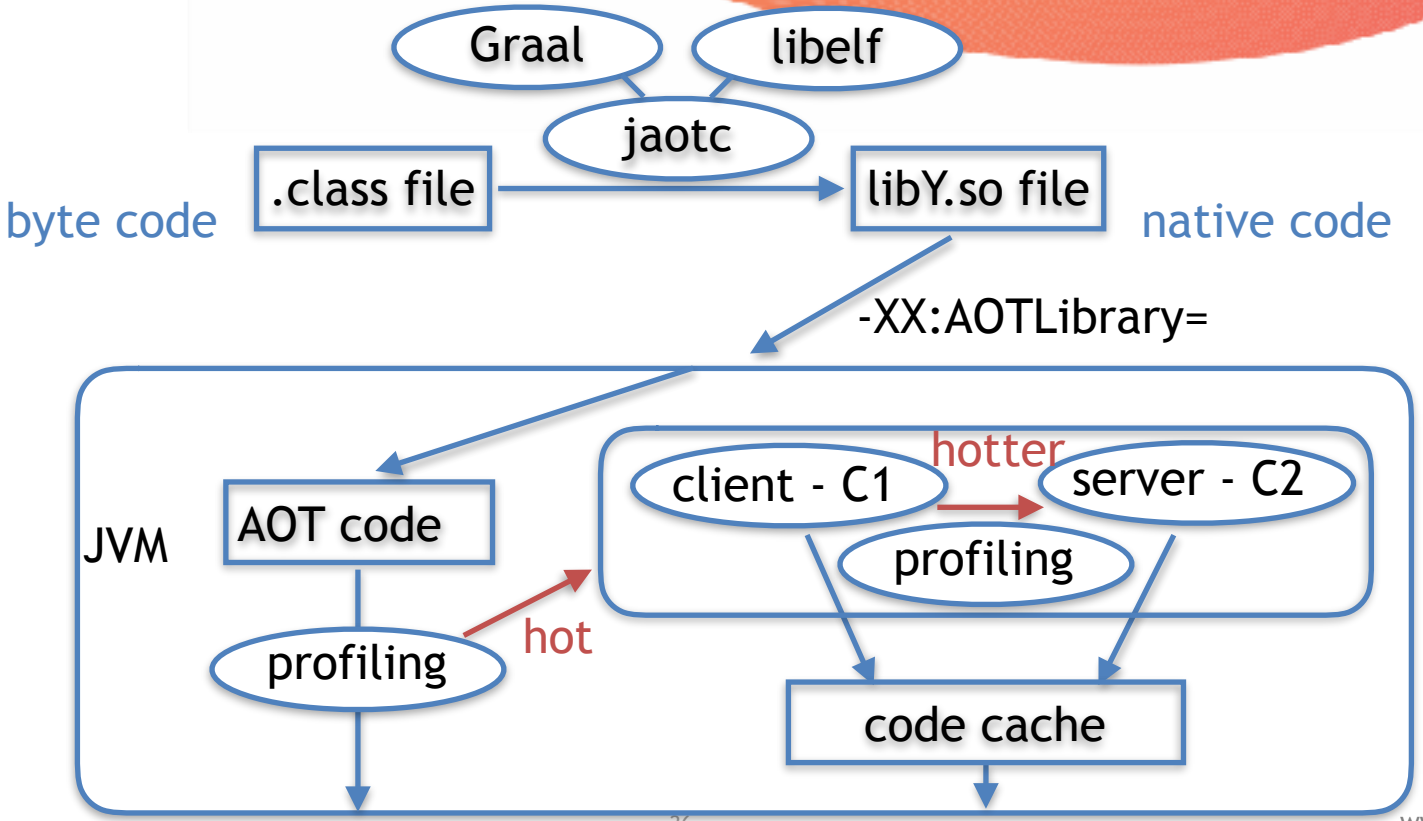


- footprint more important than peak performance
  - no JIT-code cache, no compiler threads
- more predictable behavior
  - constant speed of code, no JIT-compilation taking CPU

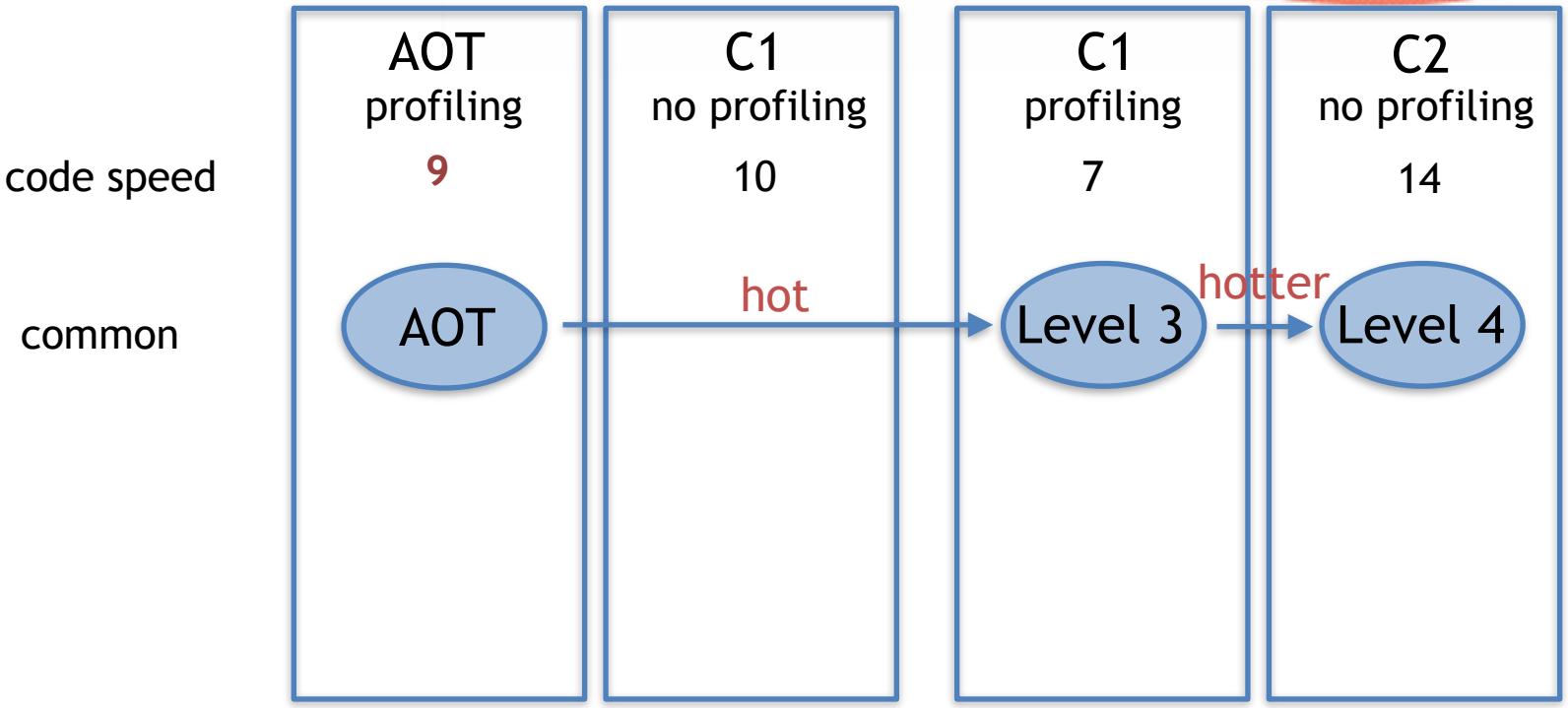
HelloJUG
Real times, average of 5 runs

java HelloJUG	252 ms
java -Xcomp …	2272 ms
java -Xint …	214 ms
java -XX:TieredStopAtLevel=1…	244 ms
java -XX:AOTLibrary=lib..nont.so…	214 ms

Tiered AOT-compilation



Tiered compilation levels with AOT



Why AOT-compilation?

- Faster startup time
  - compiled/optimized methods immediately available
  - JIT optional, for best peak-performance
- Reach peak-performance quicker
- Potentially less memory usage
  - Class data sharing (CDS)
  - No JIT-compiler overhead
- Less CPU usage
  - less interpreting, profiling, compiling

#jfall17

AOT-compiling HelloJUG

```
jeroen@jeroen-VirtualBox:~/Proj/HelloJUG/out/HelloJUG$ jaotc --info --output libHelloJUG-t.so --compile-for-tiered --cl
ass-name com.jpinpoint.jfall.HelloJUG
Compiling libHelloJUG-t...
1 classes found (39 ms)
2 methods total, 2 methods to compile (7 ms)
Compiling with 2 threads
.
2 methods compiled, 0 methods failed (2039 ms)
Parsing compiled code (8 ms)
Processing metadata (20 ms)
Preparing stubs binary (1 ms)
Preparing compiled binary (0 ms)
Creating binary: libHelloJUG-t.o (47 ms)
Creating shared library: libHelloJUG-t.so (86 ms)
Total time: 3665 ms
```

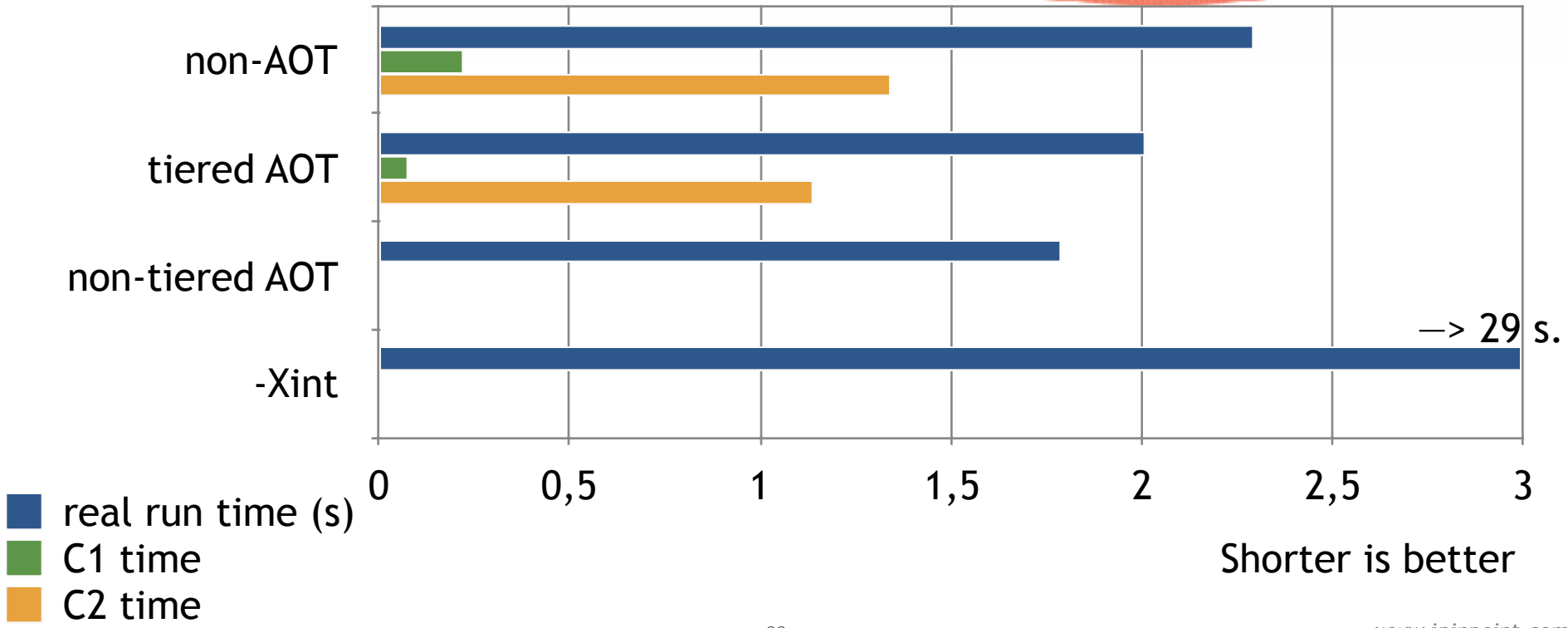
Run HelloJUG using AOT with -XX:AOTLibrary=

```
jeroen@jeroen-VirtualBox:~/Proj/HelloJUG/out/HelloJUG$ java -XX:AOTLibrary=./libHelloJUG-t.so,./libjava.base-coop-t.so
-cp . com.jpinpoint.jfall.HelloJUG
Hello JUG!
Speaker dinner desert: [ice cream, chocolate, cake]
Speakers: [Mark, Sander, Arun, Roy, Ray, Jeroen]
```

Run HelloJUG using AOT with -XX:AOTLibrary and -XX:PrintAOT

```
jeroen@jeroen-VirtualBox:~/Proj/HelloJUG/out/HelloJUG$ java -XX:AOTLibrary=./libHelloJUG-t.so -XX:+PrintAOT -cp . com.j
pinpoint.jfall.HelloJUG
      5    1     loaded    ./libHelloJUG-t.so  aot library
    143    1     aot[ 1]   com.jpinpoint.jfall.HelloJUG.<init>()V
    143    2     aot[ 1]   com.jpinpoint.jfall.HelloJUG.main([Ljava/lang/String;)V
Hello JUG!
Speaker dinner desert: [ice cream, chocolate, cake]
Speakers: [Ray, Sander, Roy, Jeroen, Arun, Mark]
```

CreateCalendars micro-benchmark
Average of 5 runs, 2 CPU's



Current limitations

- For JDK 9 only supported on Linux x64
  - JDK 10 (18.3) also on MacOS and Windows
- No use of profiling data during AOT (yet)
- Only supported for G1 GC and Parallel GC

AOT-compilation conclusions

- Promising technique for quicker startup times
- Most noticeable when user waiting for startup / execution
  - Short run: non-tiered - without JIT-compiler
  - Small devices with little resources
  - IDE's, unit tests, code generation, coding checks, ..
  - Can be 10% to 2x better for short execution or startup
- AOT-compile and bundle only touched methods of java.base etc. should help more
  - training run in future versions

Part 1: New in Java compilation

Part 2: New in Java garbage collection

#jfall17

Part 2:
New garbage collectors

Shenandoah and
Epsilon GC

New garbage collectors

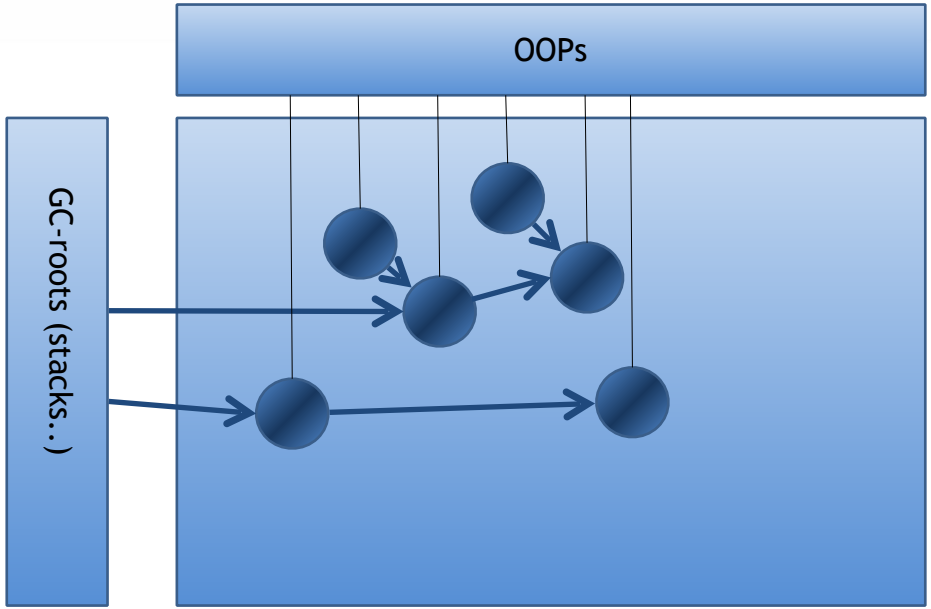
- GC basics
- Serial, Parallel, Concurrent GC
- Region based: G1 GC
- Shenandoah GC
- Epsilon GC
- Conclusions

#jfall17

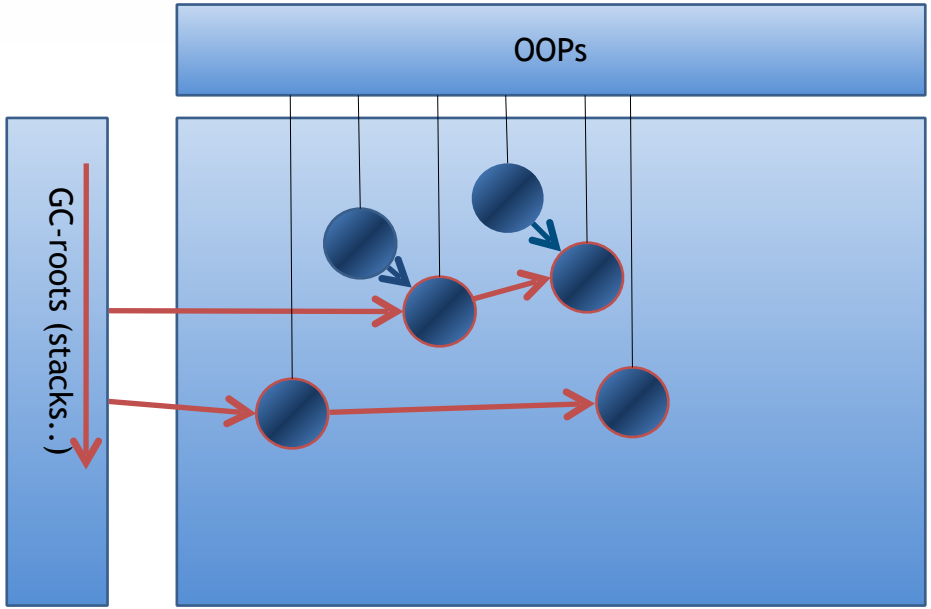
Garbage collection basics

Mark&Sweep, Compaction and Mark&Copy

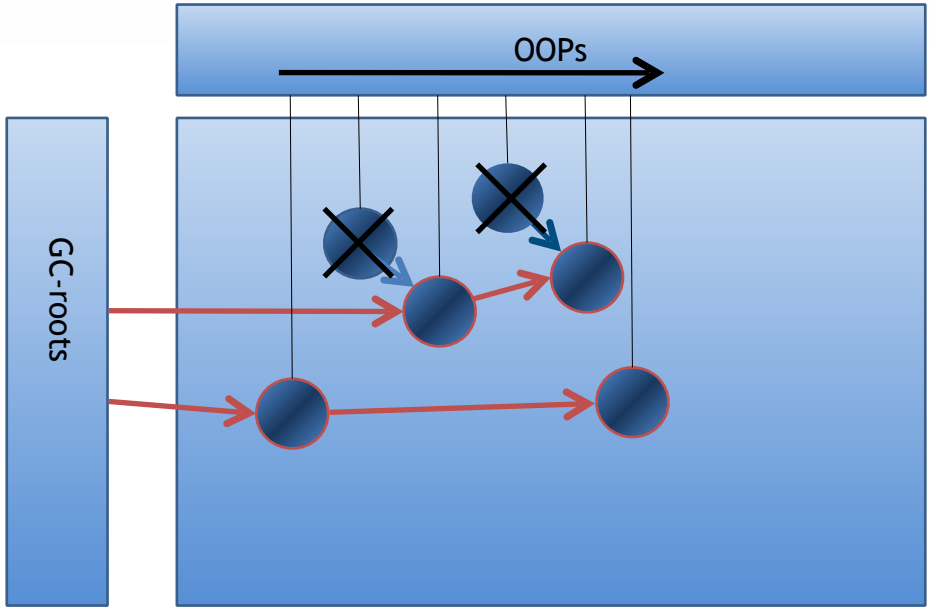
Mark and Sweep



Mark



Sweep

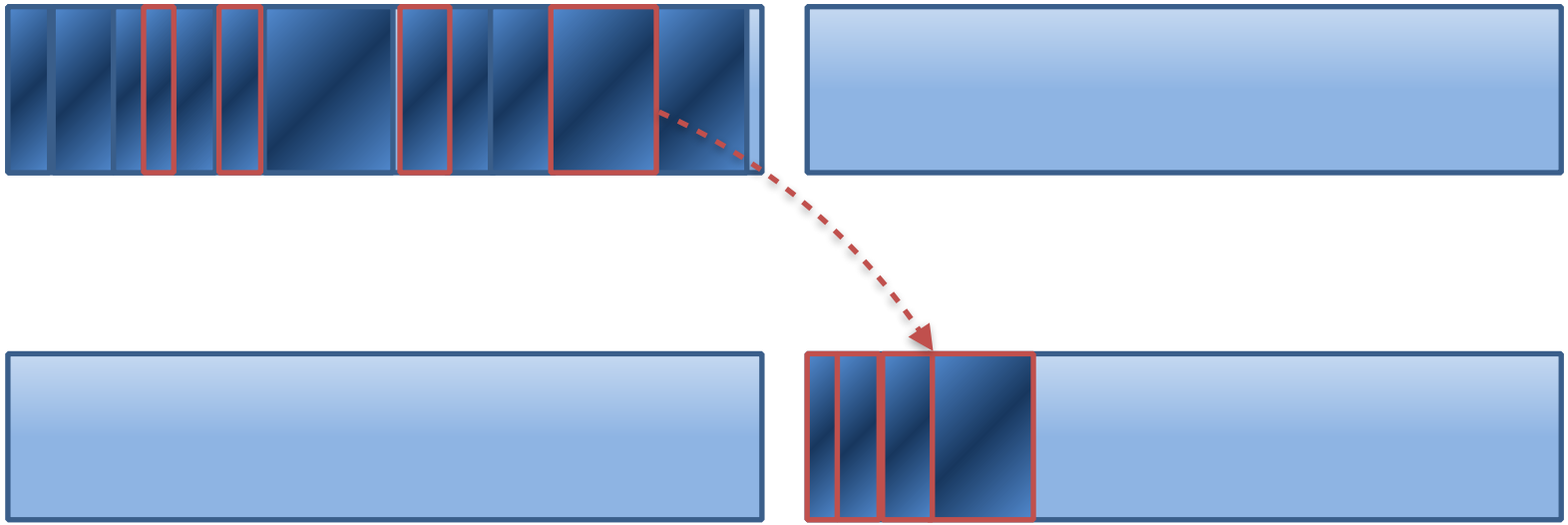


Mark-Sweep: Fragmentation

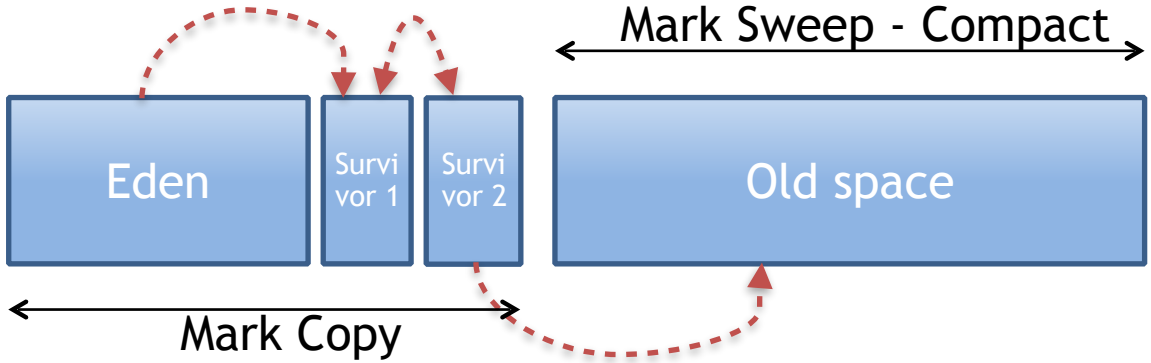
- After compaction:

- Compaction is expensive

Mark-Copy: no fragmentation

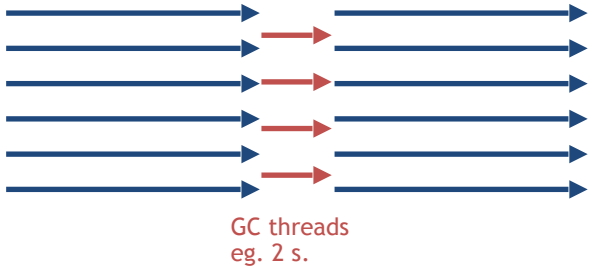


Generational GC: Young and Old

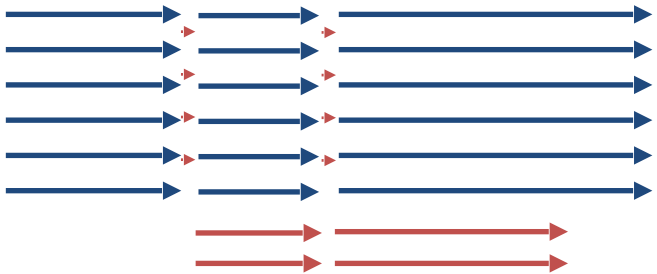


Serial GC: stop-the-world pauzes

Serial vs Parallel GC

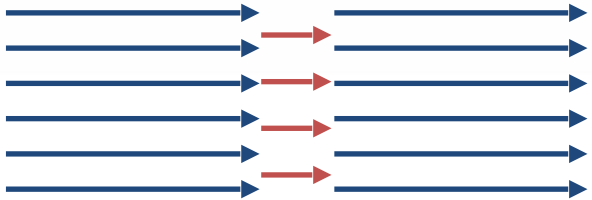


Concurrent Mark Sweep GC - mostly concurrent to app

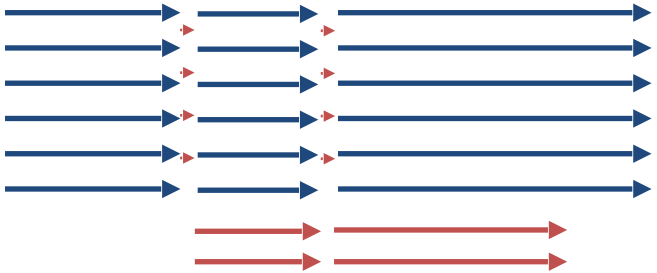


GC threads
eg. 2x100 ms. STW
10 s. concurrent

Parallel GC vs CMS
throughput vs responsiveness

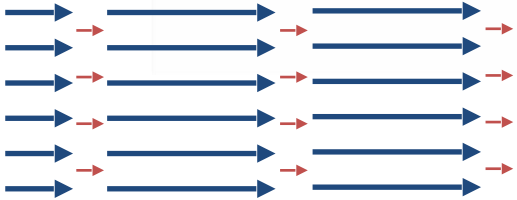


GC threads
eg. 2 s.

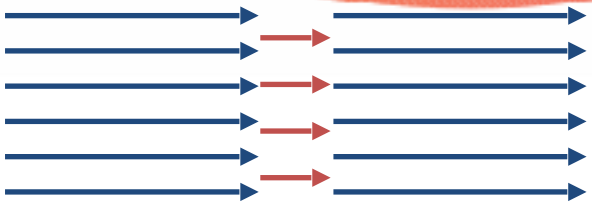


GC threads
eg. 2x100 ms. STW
10 s. concurrent

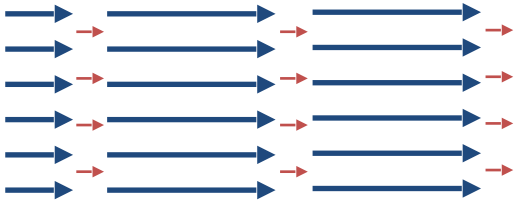
Parallel GC vs CMS Young and Old



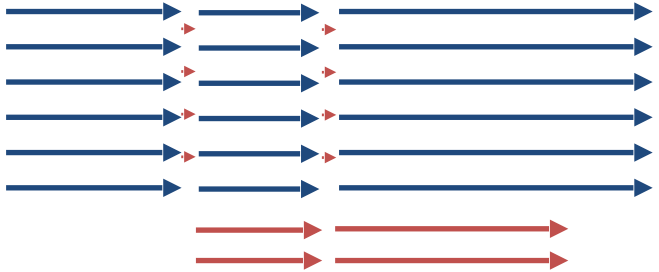
GC threads
300 ms.



GC threads
2 s.

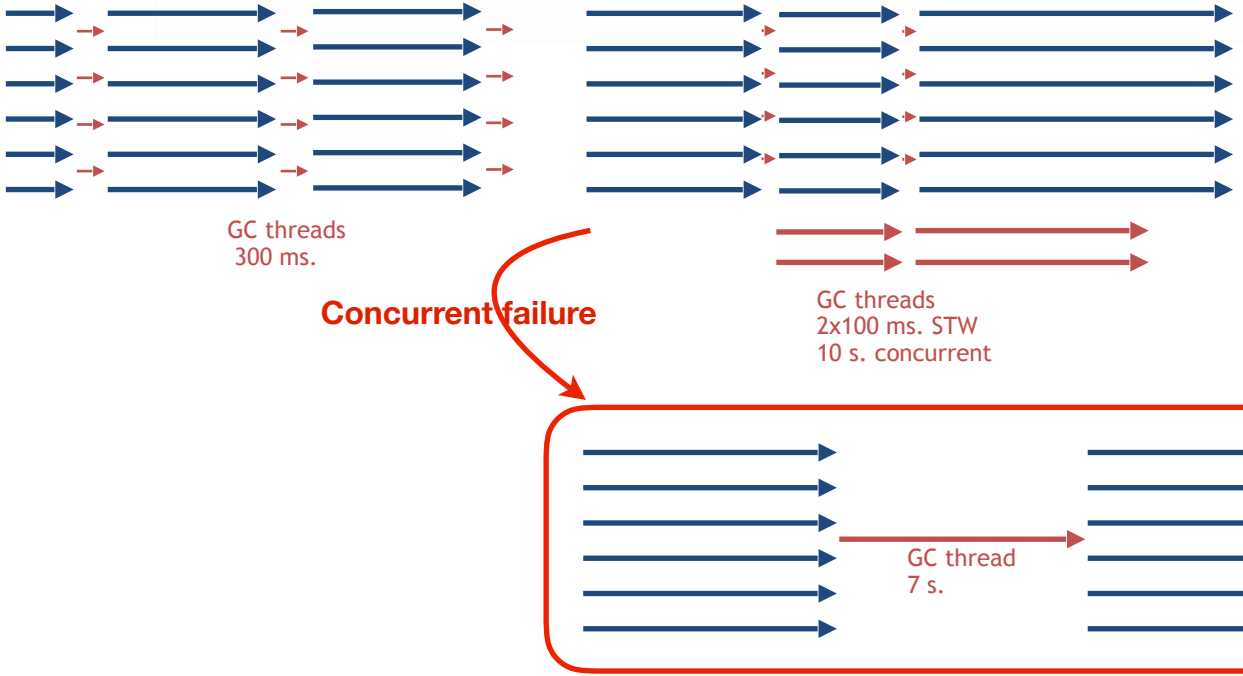


GC threads
300 ms.



GC threads
2x100 ms. STW
10 s. concurrent

CMS problem 1: failures

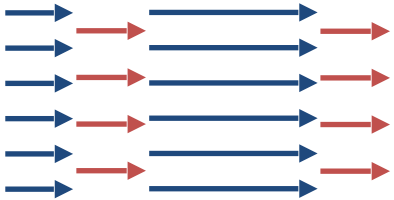


CMS problem 2: not scalable

4 GB heap

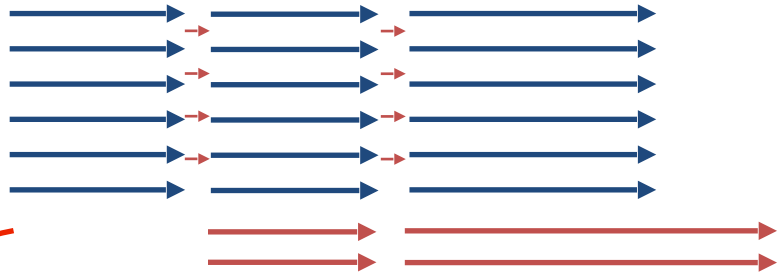
x4

16 GB heap



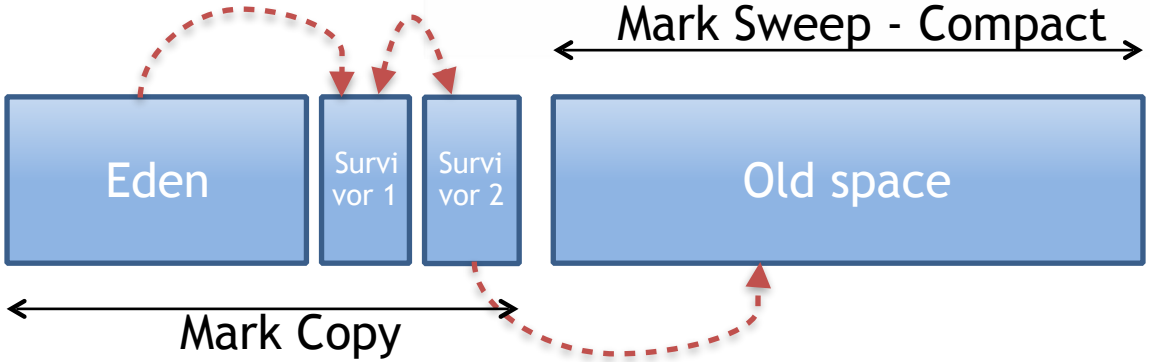
GC threads
1200 ms.

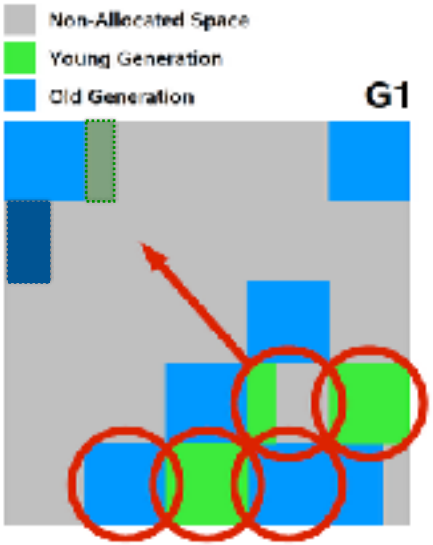
Concurrent failure



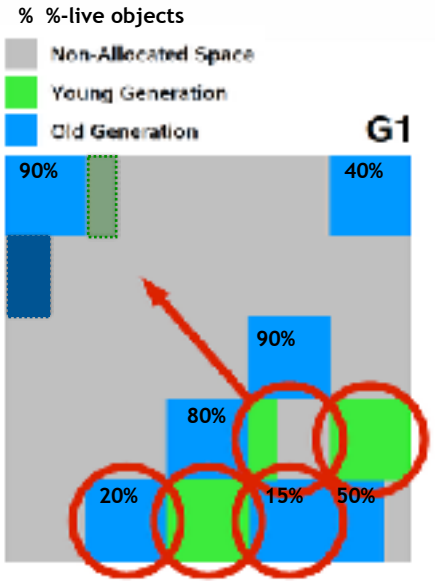
GC threads
2x400 ms. STW
40 s. concurrent

Solution: regionalize



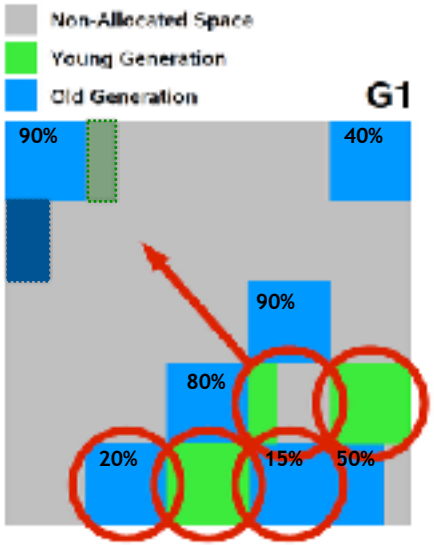


G1: Garbage first: young + old with most garbage

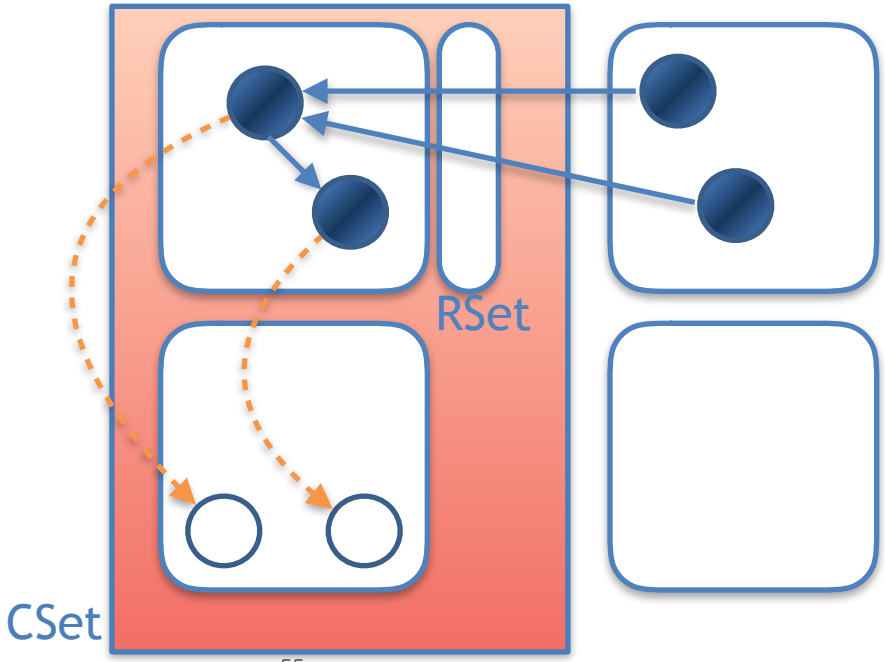


G1: Garbage first: young + old regions with most garbage

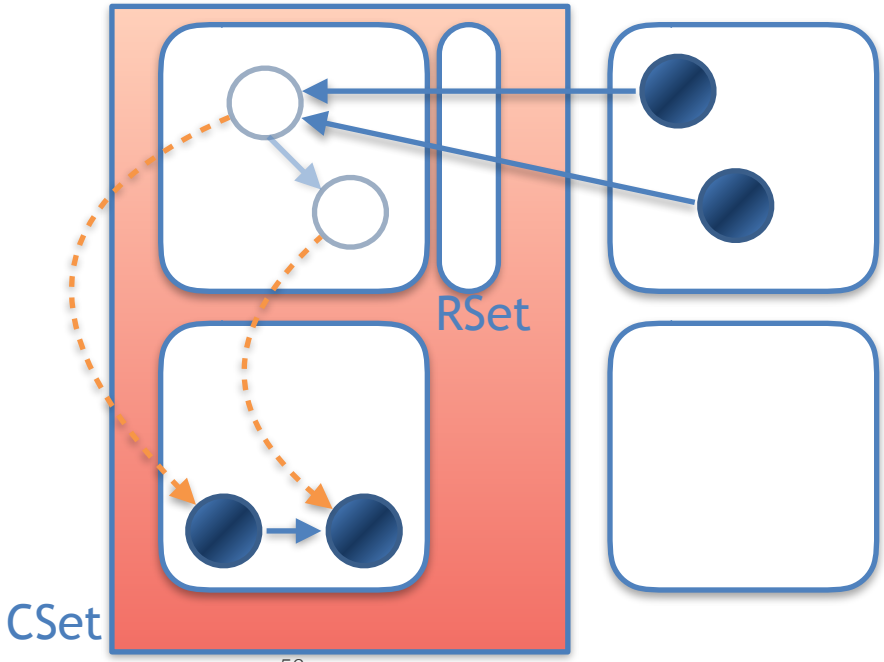
- Concurrent mark in Old
- Mark-copy: no fragmentation
- Limit copy #regions to meet -XX:MaxPauseTimeMillis
- Solves scalability and failures
- Still stw pauses



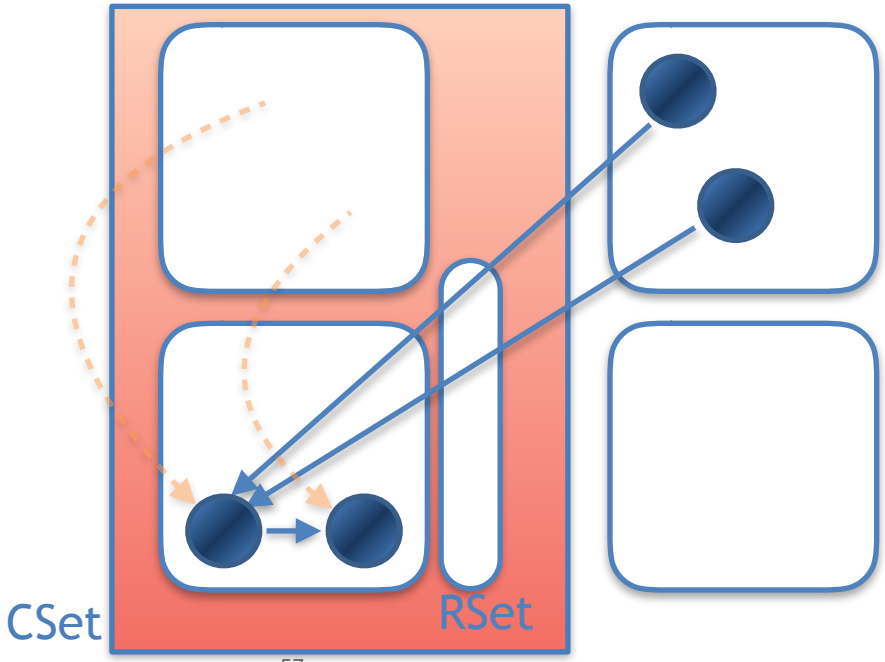
G1 stw-pauses: Mark young + Copy Y&O



G1 stw-pauses: Mark young + Copy Y&O + Update refs



G1 pauses: Mark young + Copy Y&O + Update refs



#jfall17

Shenandoah GC

Ultra low pause times

Stw-pause times in G1 how to beat them?

- Mark Young
- Copy Young & Old
- Update refs

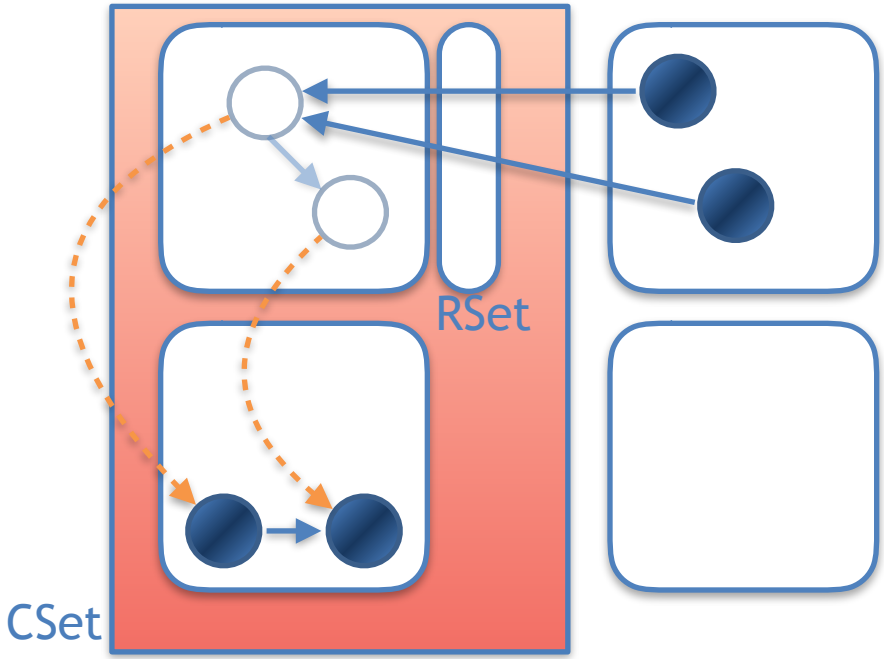
pause times in G1 vs Shenandoah

- ~~Mark Young~~
- Copy ~~Young~~ & Old
- Update refs

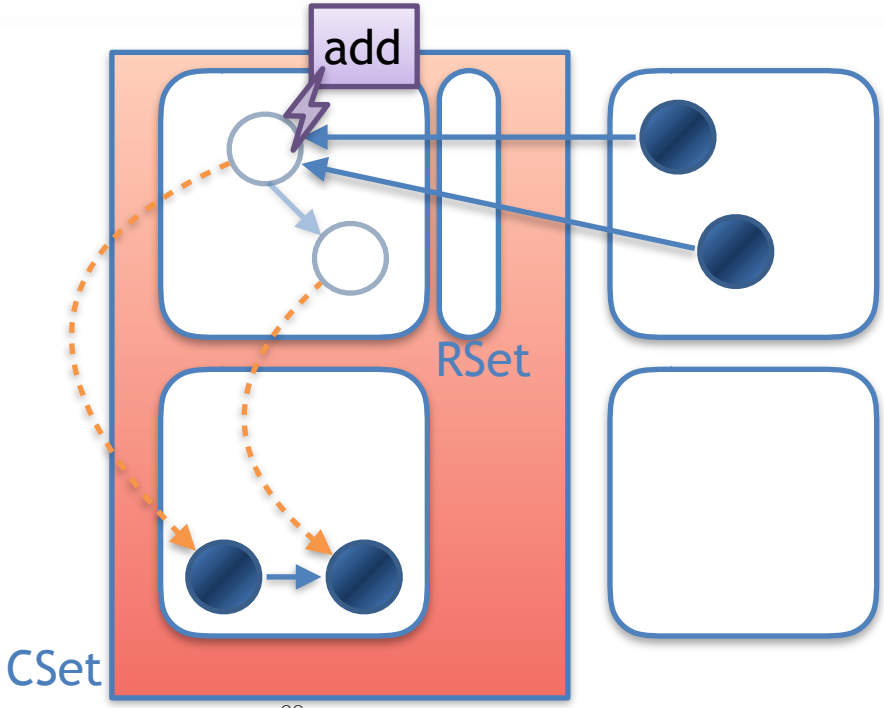
removed pause times in Shenandoah

- Mark Young
- Copy Young & Old ⟵ concurrent
- Update refs ⟵ concurrent

How to copy Old and update refs concurrently?



How to copy Old and update refs concurrently?

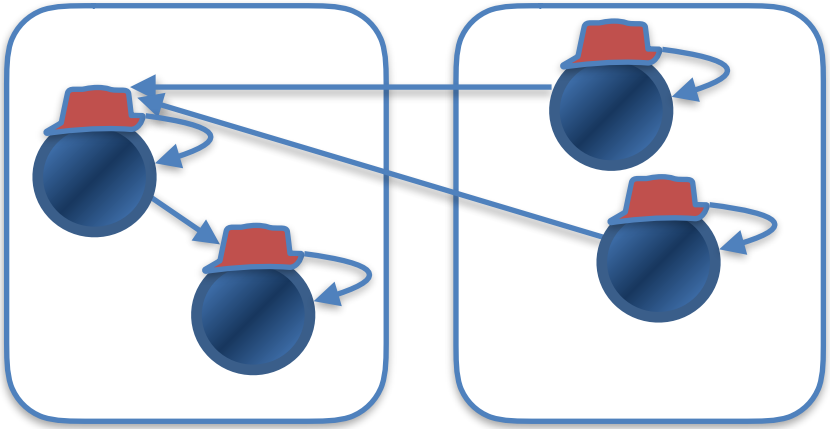


All problems in computer science can be solved by…?

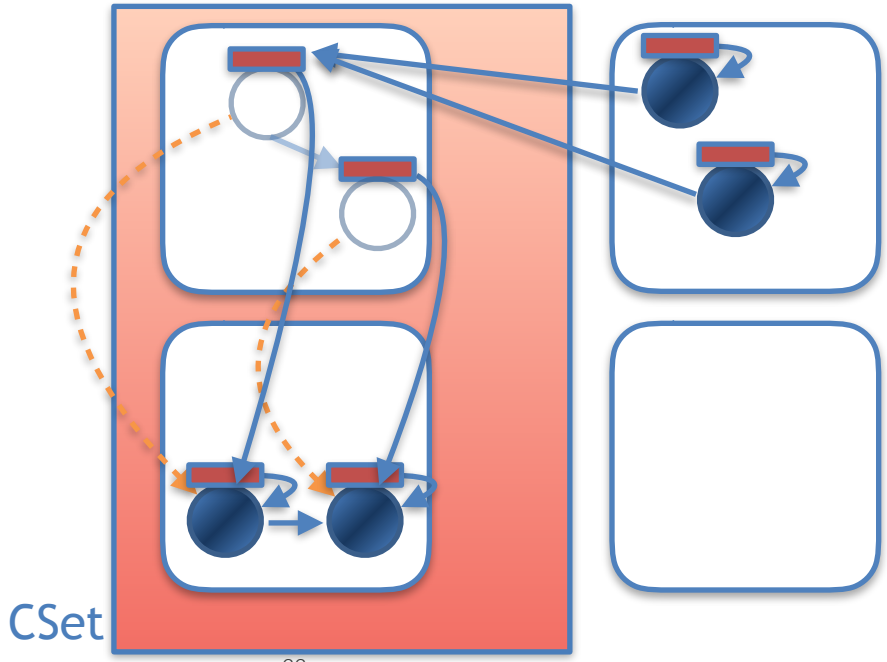
Jim Coplien - David Wheeler

Another level of indirection

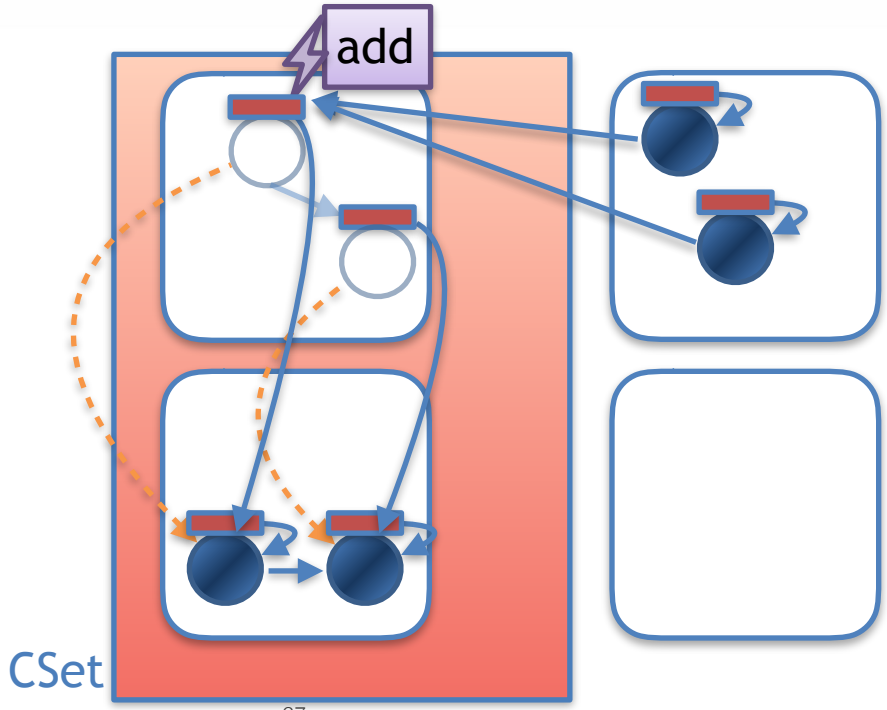
- forwarding pointer



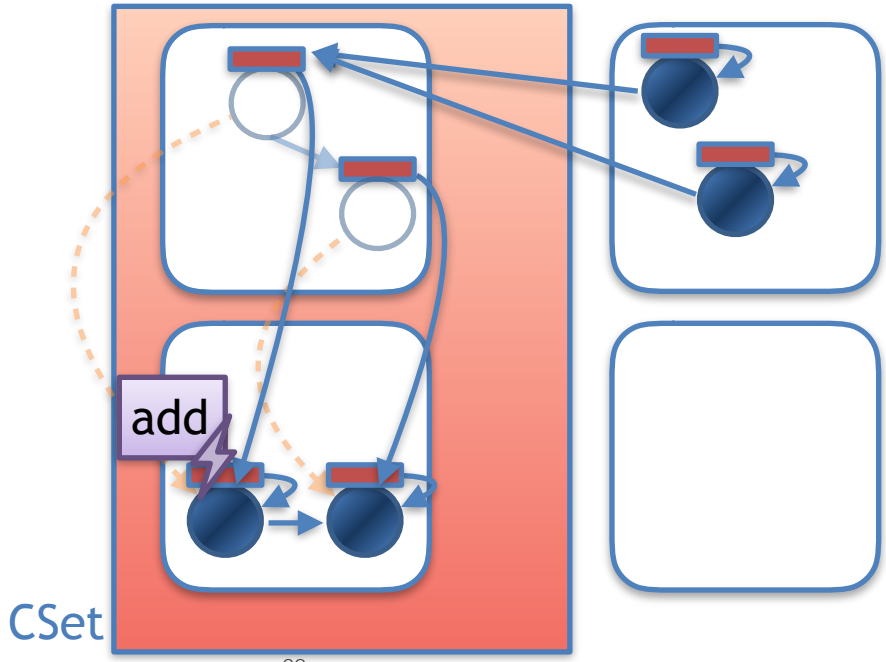
Copy Old and update refs concurrently by fwd pointer



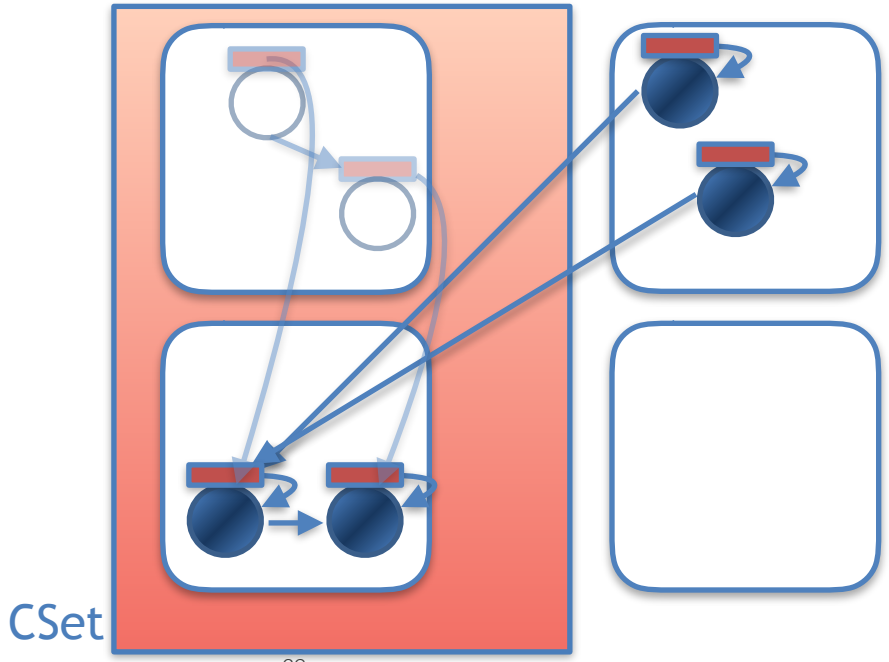
Application thread can write objects being evacuated



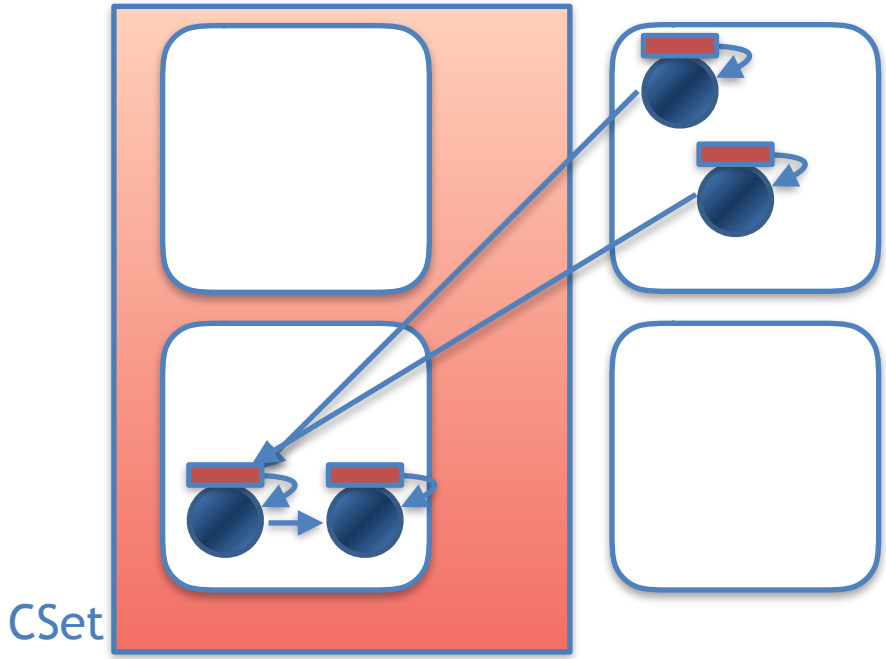
Application thread writes to new copy of object



Concurrent update refs



Concurrent cleanup



Shenandoah pause times

- 4 short stw pauses
  - init mark, final mark, init UR, final UR

```
GC(3) Pause Init Mark 0.771ms
GC(3) Concurrent marking 76480M->77212M(102400M) 633.213ms
GC(3) Pause Final Mark 1.821ms
GC(3) Concurrent cleanup 77224M->66592M(102400M) 3.112ms
GC(3) Concurrent evacuation 66592M->75640M(102400M) 405.312ms
GC(3) Pause Init Update Refs 0.084ms
GC(3) Concurrent update references  75700M->76424M(102400M) 354.341ms
GC(3) Pause Final Update Refs 0.409ms
GC(3) Concurrent cleanup 76244M->56620M(102400M) 12.242ms
```

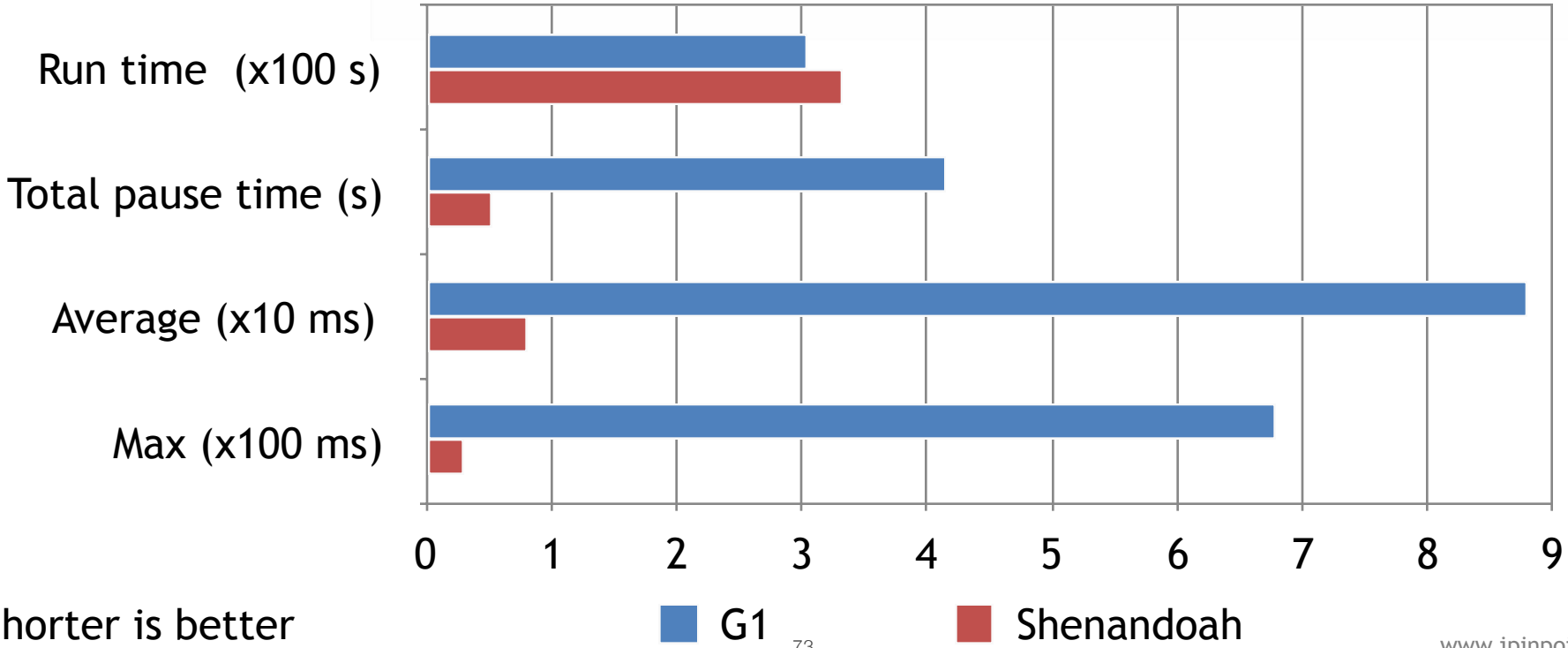
Shenandoah pause times

- 4 short stw pauses
  - init mark, final mark, init UR, final UR

```
GC(3) Pause Init Mark 0.771ms
GC(3) Concurrent marking 76480M->77212M(102400M) 633.213ms
GC(3) Pause Final Mark 1.821ms
GC(3) Concurrent cleanup 77224M->66592M(102400M) 3.112ms
GC(3) Concurrent evacuation 66592M->75640M(102400M) 405.312ms
GC(3) Pause Init Update Refs 0.084ms
GC(3) Concurrent update references  75700M->76424M(102400M) 354.341ms
GC(3) Pause Final Update Refs 0.409ms
GC(3) Concurrent cleanup 76244M->56620M(102400M) 12.242ms
```

Elastic search benchmark

(Oct 2, 2017)



Parallel GC: max = 161 ms

```
jeroen@jeroen-VirtualBox:~/Proj/HelloJUG/out/HelloJUG$ time /usr/local/jdk-9-sh/bin/java -cp . -XX:+UseParallelOldGC -Xlo
g:gc -Xmx384m com.jpinpoint.jfall.CreateCalendars
[0.009s][info][gc] Using Parallel
Starting
[0,523s][info][gc] GC(0) Pause Young (Allocation Failure) 24M->4M(90M) 13,693ms
[0,599s][info][gc] GC(1) Pause Young (Allocation Failure) 28M->9M(114M) 22,478ms
[0,732s][info][gc] GC(2) Pause Young (Allocation Failure) 57M->17M(114M) 17,825ms
Iteration 1 took ms: 594
[0,980s][info][gc] GC(3) Pause Young (Allocation Failure) 65M->25M(162M) 13,308ms
Iteration 2 took ms: 226
[1,249s][info][gc] GC(4) Pause Young (Allocation Failure) 121M->42M(162M) 22,769ms
Iteration 3 took ms: 255
[1,511s][info][gc] GC(5) Pause Young (Allocation Failure) 138M->57M(164M) 42,638ms
Iteration 4 took ms: 272
[1,718s][info][gc] GC(6) Pause Young (Allocation Failure) 137M->71M(167M) 21,359ms
[1,879s][info][gc] GC(7) Pause Full (Ergonomics) 71M->67M(197M) 161,373ms
[2,039s][info][gc] GC(8) Pause Young (Allocation Failure) 147M->80M(209M) 29,507ms
Iteration 5 took ms: 486
Number of string numbers: 100000

real    0m2,167s
user    0m1,832s
sys     0m0,152s
```

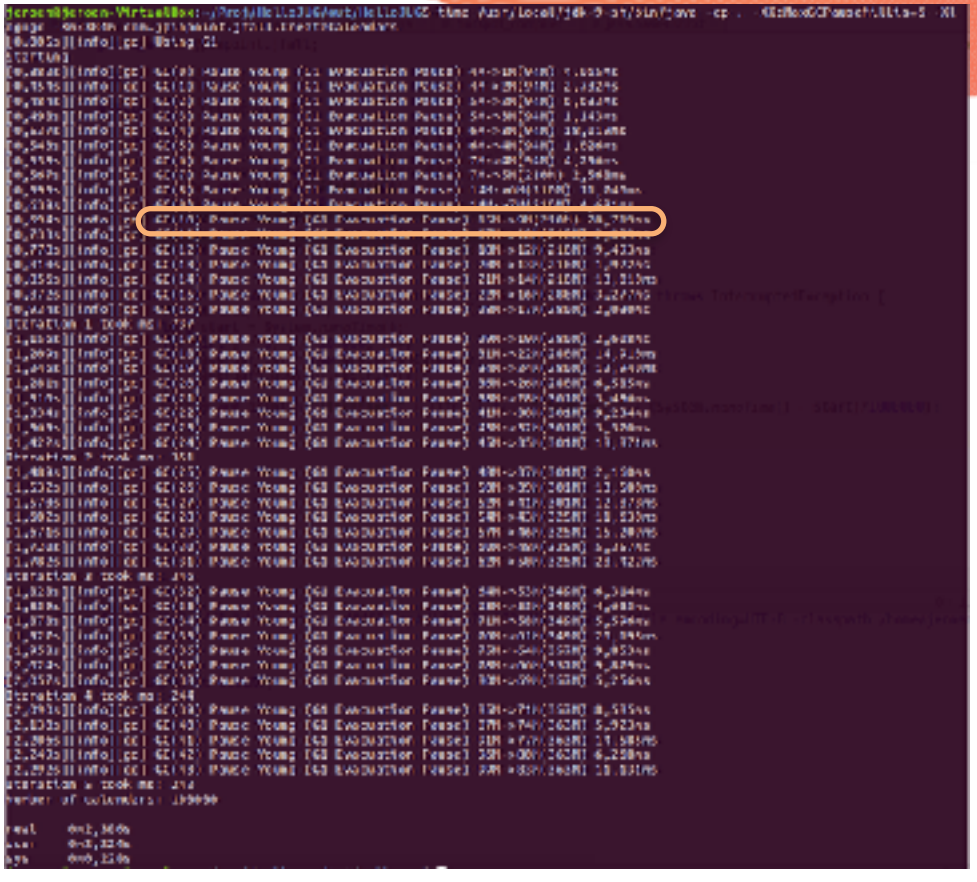
G1 GC: max = 38 ms

```
jeroen@jeroen-VirtualBox:~/Proj/HelloJUG/out/HelloJUG$ time /usr/local/jdk-9-sh/bin/java -cp . -Xlog:gc -Xmx384m com.jpin
point.jfall.CreateCalendars
[0.005s][info][gc] Using G1
Starting
[0,393s][info][gc] GC(0) Pause Young (G1 Evacuation Pause) 7M->2M(94M) 4,187ms
[0,588s][info][gc] GC(1) Pause Young (G1 Evacuation Pause) 57M->10M(94M) 7,150ms
[0,622s][info][gc] GC(2) Pause Young (G1 Evacuation Pause) 19M->13M(94M) 11,767ms
[0,670s][info][gc] GC(3) Pause Young (G1 Evacuation Pause) 29M->15M(94M) 2,678ms
Iteration 1 took ms: 537
[0,866s][info][gc] GC(4) Pause Young (G1 Evacuation Pause) 38M->19M(94M) 19,026ms
[0,969s][info][gc] GC(5) Pause Young (G1 Evacuation Pause) 49M->24M(94M) 24,498ms
[1,111s][info][gc] GC(6) Pause Young (G1 Evacuation Pause) 54M->30M(210M) 37,804ms
Iteration 2 took ms: 408
[1,409s][info][gc] GC(7) Pause Young (G1 Evacuation Pause) 73M->37M(210M) 29,955ms
[1,601s][info][gc] GC(8) Pause Young (G1 Evacuation Pause) 89M->46M(210M) 11,172ms
Iteration 3 took ms: 390
[1,817s][info][gc] GC(9) Pause Young (G1 Evacuation Pause) 110M->56M(210M) 17,451ms
[1,940s][info][gc] GC(10) Pause Young (G1 Evacuation Pause) 123M->68M(210M) 24,992ms
Iteration 4 took ms: 192
[2,062s][info][gc] GC(11) Pause Young (G1 Evacuation Pause) 134M->78M(245M) 26,447ms
Iteration 5 took ms: 169
Number of string numbers: 100000

real    0m2,187s
user    0m2,212s
sys     0m0,212s
```

Shenandoah GC: max = 5,9 ms

```
jeroen@jeroen-VirtualBox:~/Proj/HelloJUG/out/HelloJUG$ time /usr/local/jdk-9-sh/bin/java -cp . -XX:+UseShenandoahGC -Xlog
:gc -Xmx384m com.jpinpoint.jfall.CreateCalendars
[0.014s][info][gc] Using Shenandoah
Starting
Iteration 1 took ms: 356
Iteration 2 took ms: 317
[1,027s][info][gc] GC(0) Pause Init Mark 1,459ms
[1,087s][info][gc] GC(0) Concurrent marking 271M->284M(384M) 59,876ms
[1,096s][info][gc] GC(0) Pause Final Mark 284M->284M(384M) 5,927ms
[1,096s][info][gc] GC(0) Concurrent cleanup 284M->284M(384M) 0,051ms
[1,160s][info][gc] GC(0) Concurrent evacuation  284M->341M(384M) 64,232ms
[1,187s][info][gc] GC(0) Pause Init Update Refs 0,040ms
[1,236s][info][gc] GC(0) Concurrent update references  341M->348M(384M) 49,862ms
[1,237s][info][gc] GC(0) Pause Final Update Refs 348M->348M(384M) 0,351ms
[1,237s][info][gc] GC(0) Concurrent cleanup 348M->79M(384M) 0,205ms
Iteration 3 took ms: 364
Iteration 4 took ms: 183
[1,664s][info][gc] GC(1) Pause Init Mark 2,443ms
Iteration 5 took ms: 318
[1,783s][info][gc] GC(1) Concurrent marking 277M->315M(384M) 118,404ms
[1,785s][info][gc] GC(1) Pause Final Mark 315M->315M(384M) 1,927ms
[1,789s][info][gc] GC(1) Concurrent cleanup 315M->315M(384M) 0,082ms
Number of string numbers: 100000
[1,905s][info][gc] Cancelling concurrent GC: Stopping VM
[1,907s][info][gc] GC(1) Concurrent evacuation  315M->360M(384M) 117,930ms

real    0m2,018s
user    0m2,112s
sys     0m0,292s
```

G1 with -XX:MaxGCPauseTimeMillis=6: 29 ms



Epsilon GC

- JEP Draft (by Aleksey Shipilev)
- stw pauzes = 0.0 ms!
- How?

Epsilon GC

- Garbage non-collector
- JVM shutdown when heap exhausted
- java -XX:+UnlockExperimentalVMOptions XX:+UseEpsilonGC
- Binary builds of patched JDK10 available

No-op GC use cases

- Performance testing
  - compare with real garbage collectors
- Minimize overhead
  - garbage free apps
  - short-lived apps
  - active failover before heap exhausted

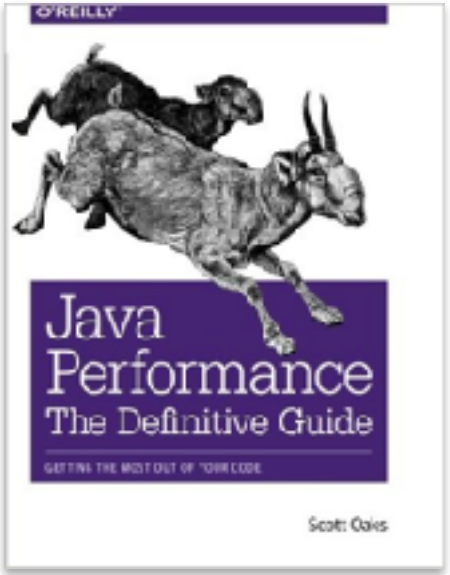
Conclusions new GC's

- Shenandoah GC clearly beats G1 for short pauses
- Likely to replace G1 as default after JDK9
  - Can try it out now on JDK8+!
- Epsilon GC eliminates all GC-overhead (and comfort)
  - if you can avoid GC
  - last-drop performance improvement

J-FALL
#jfall17

Questions?

- want to learn more?
  - resources: www.jpinpoint.com
  - accelerating Java applications training
    - covers Java 8 & 9
    - 12-14 March 2018
- thanks for the attention!



“Please rate my talk in the official J-Fall app”

#jfall17