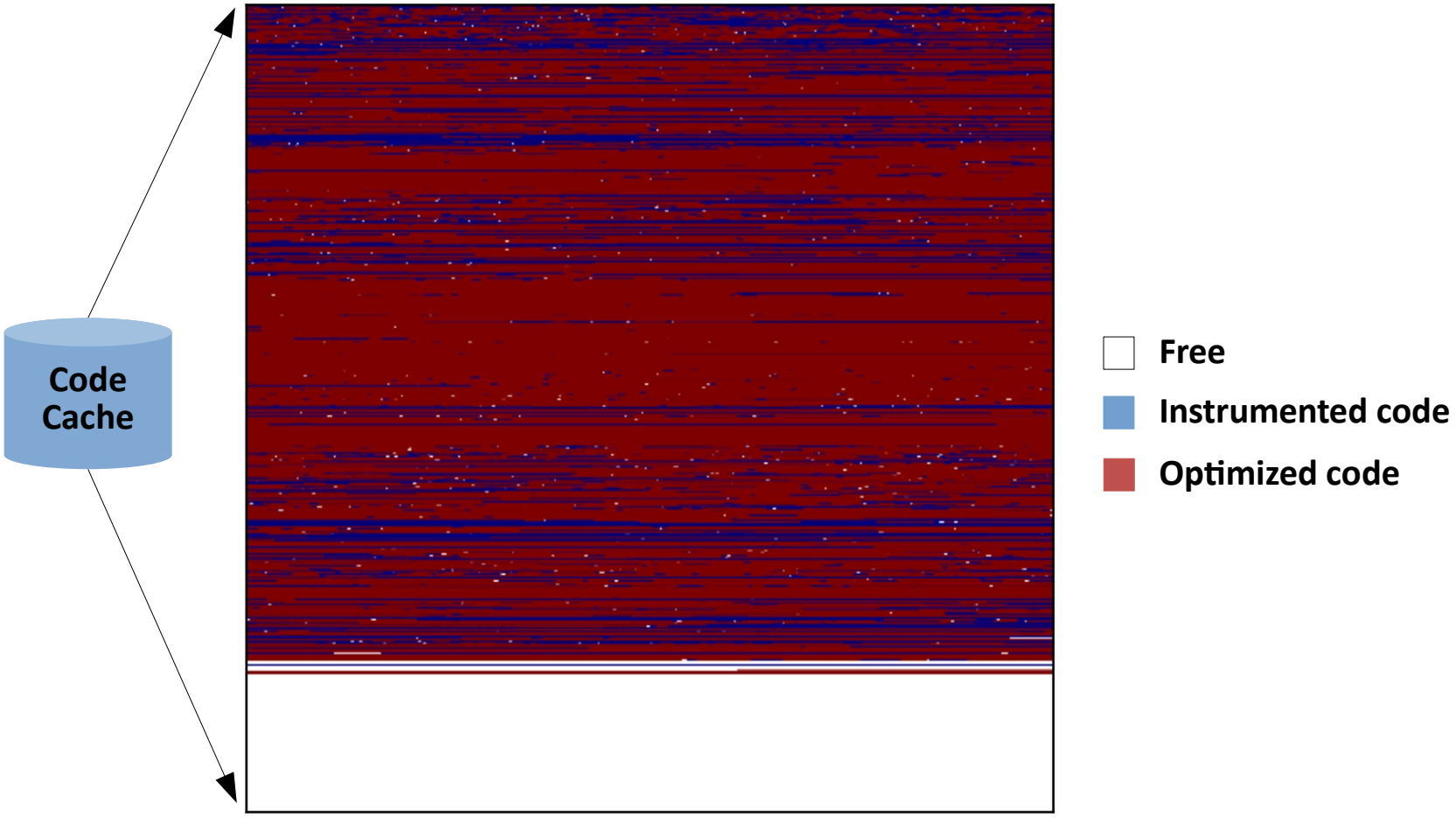
Efficient Code Cache Management for Dynamic Multi-Tiered Compilation Systems

Tobias Hartmann, ETH Zurich, Oracle Corp.
Albert Noll, Oracle Corporation
Thomas R. Gross, ETH Zurich

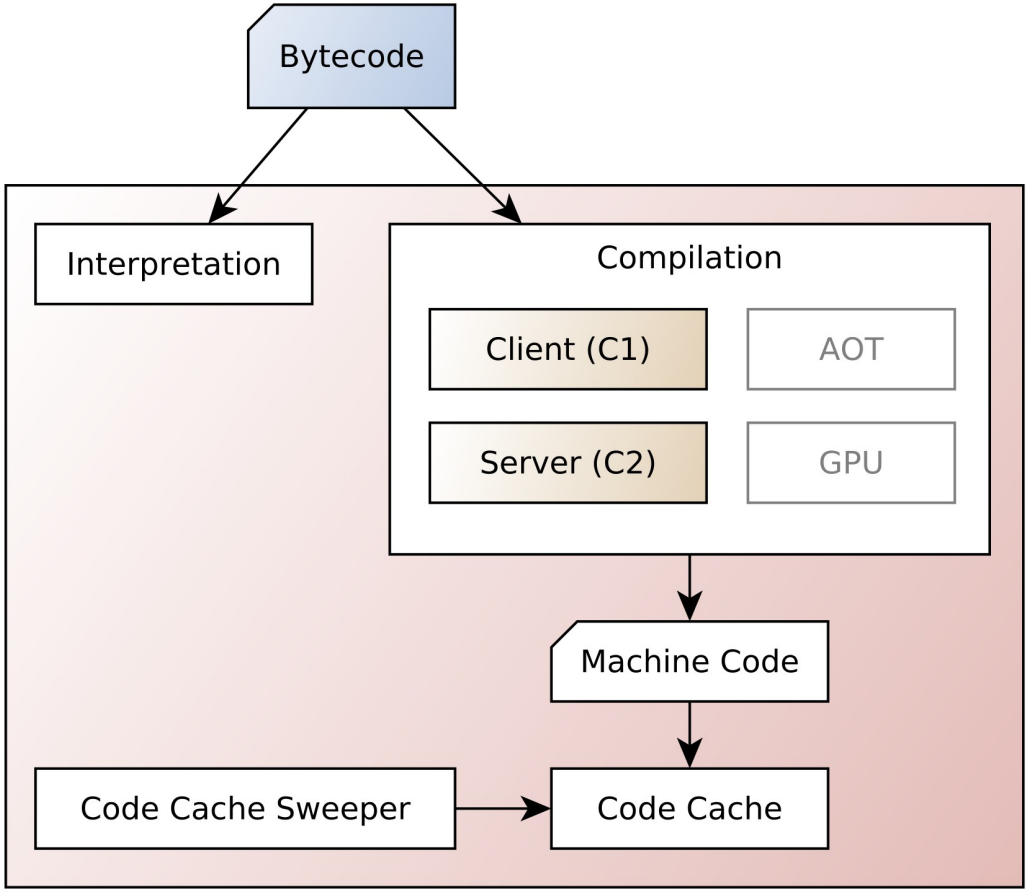
Introduction



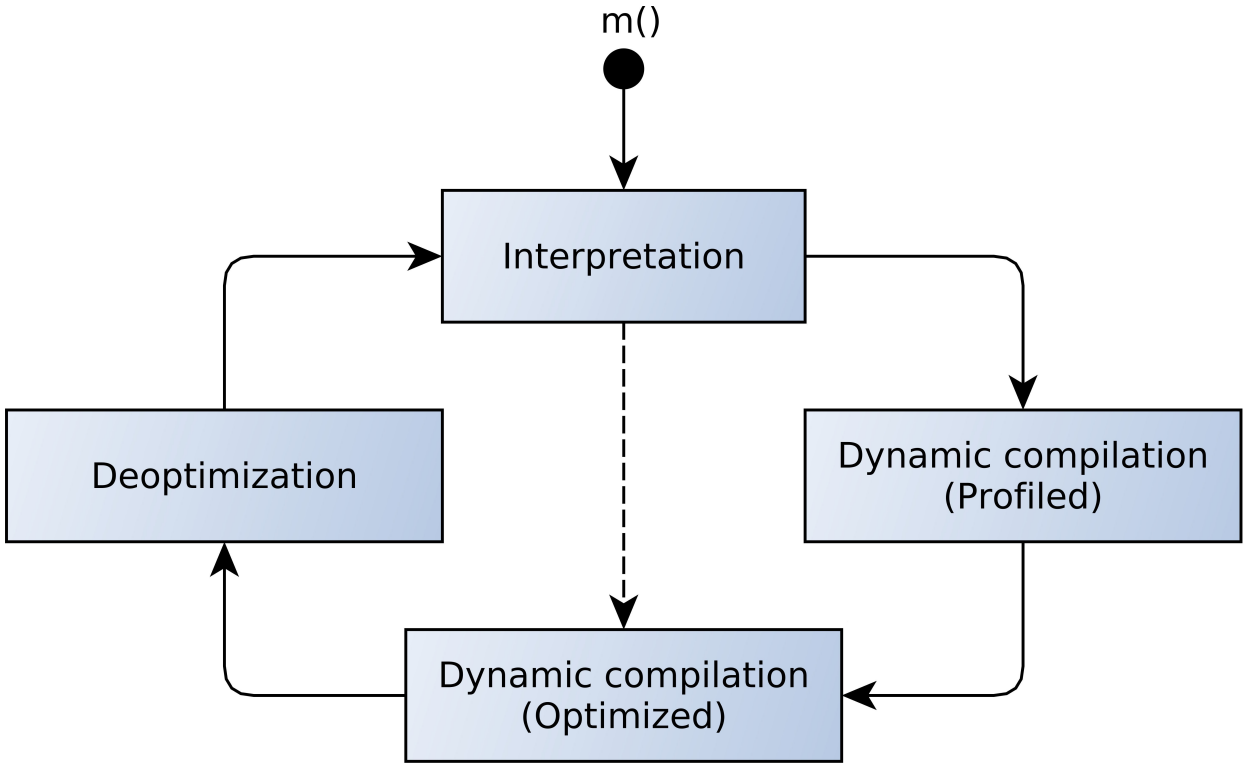
Outline

- Hotspot™ JVM
- Design
- Implementation
- Evaluation
- Conclusion

Hotspot™ JVM

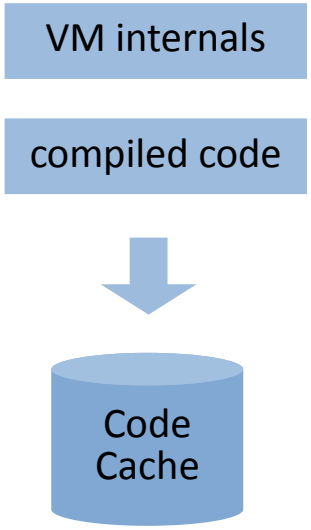

Bytecode
Interpretation
Compilation
Client (C1)
AOT
Server (C2)
GPU
Machine Code
Code Cache Sweeper
Code Cache

Dynamic compilation in the JVM

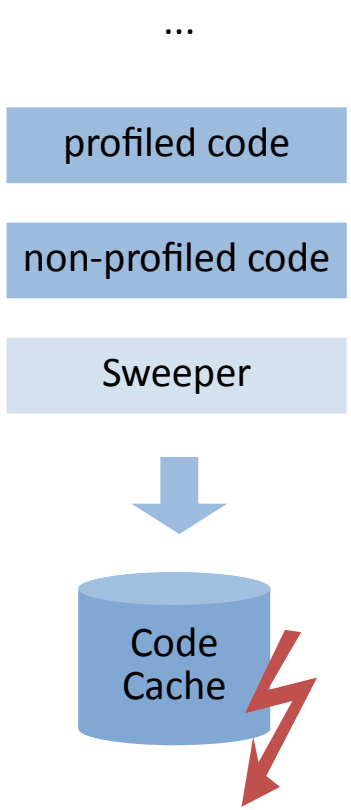


History

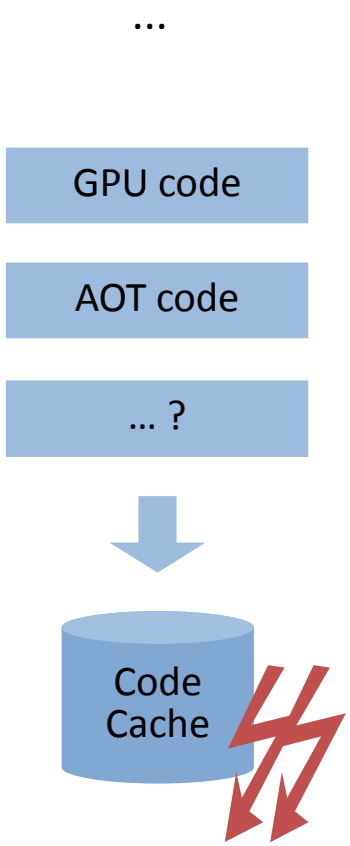
- **JDK 6**



- **JDK 7 / 8**

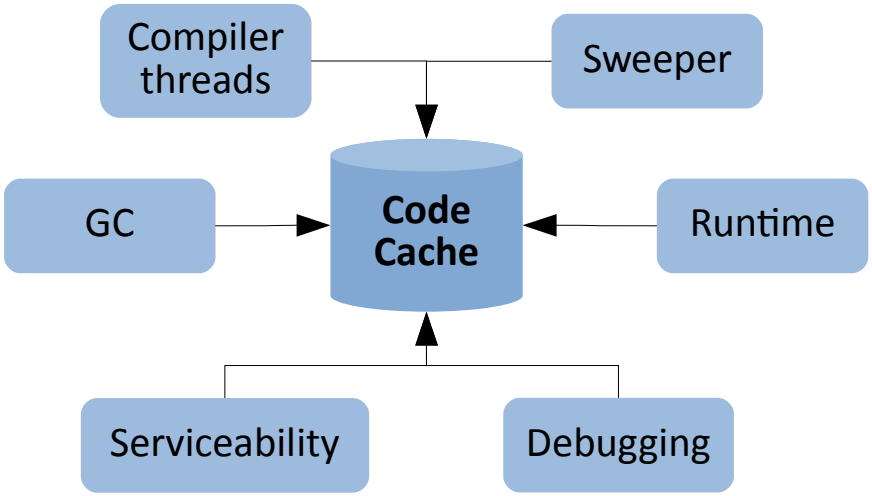


- **JDK 9 / Future**



Code cache

- **Central component**



- **Continuous chunk of memory**
  - **Fixed size**
  - Bump pointer allocation with free list

Challenges

- **With tiered compilation amount of code increased by 2-4 X**
- **All code in one cache**
  - **Different types** and characteristics
  - Access to specific code: **full iteration**

- **Code cache fragmentation**

Challenges

- **With tiered compilation amount of code increased by 2-4 X**
- **All code in one cache**
  - **Different types** and characteristics
  - Access to specific code: **full iteration**

- **Code cache fragmentation**
- **Solution: Segmented Code Cache**

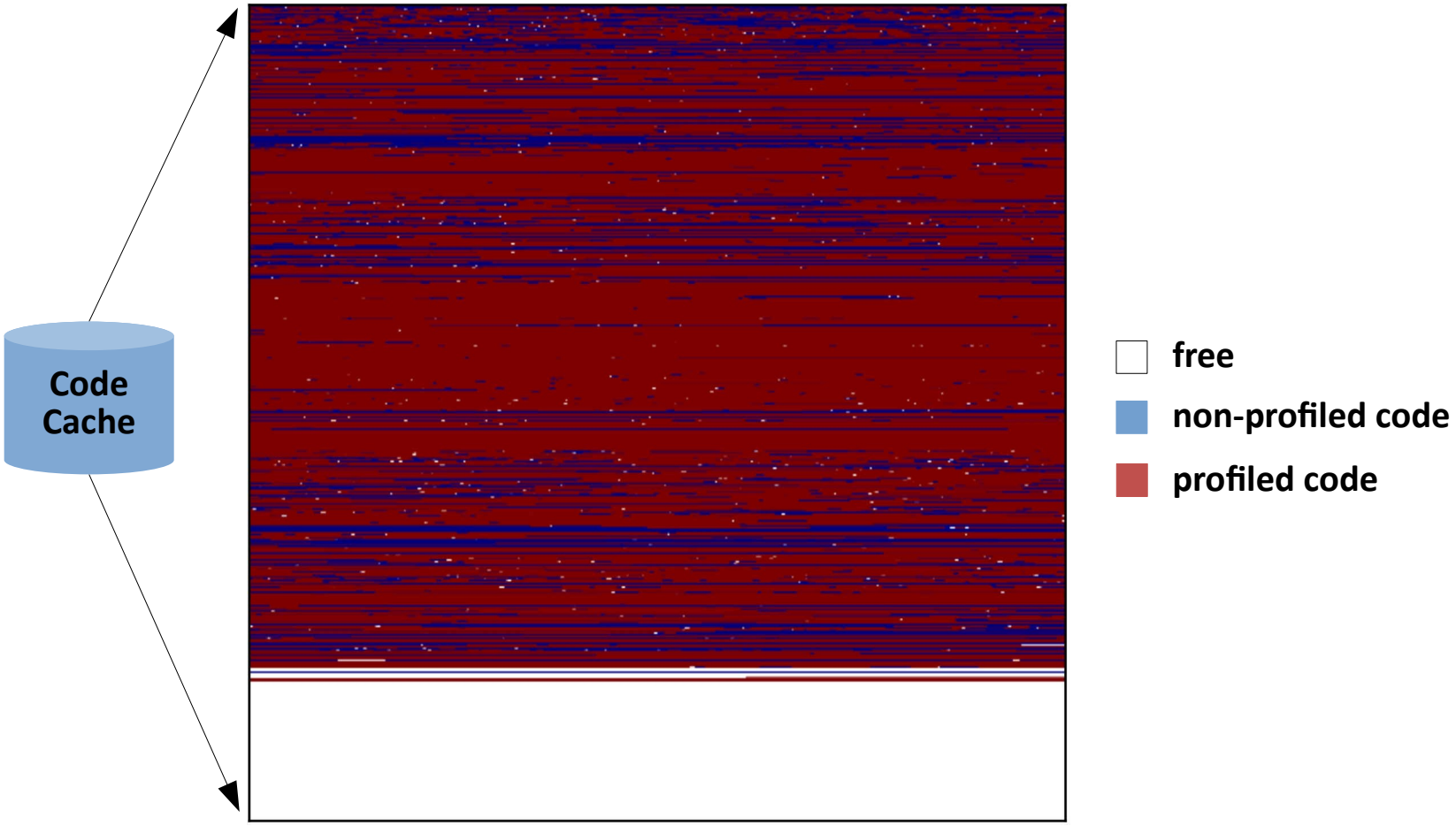
Properties of compiled code

- **Lifetime**
- **Size**
- **Cost of generation**

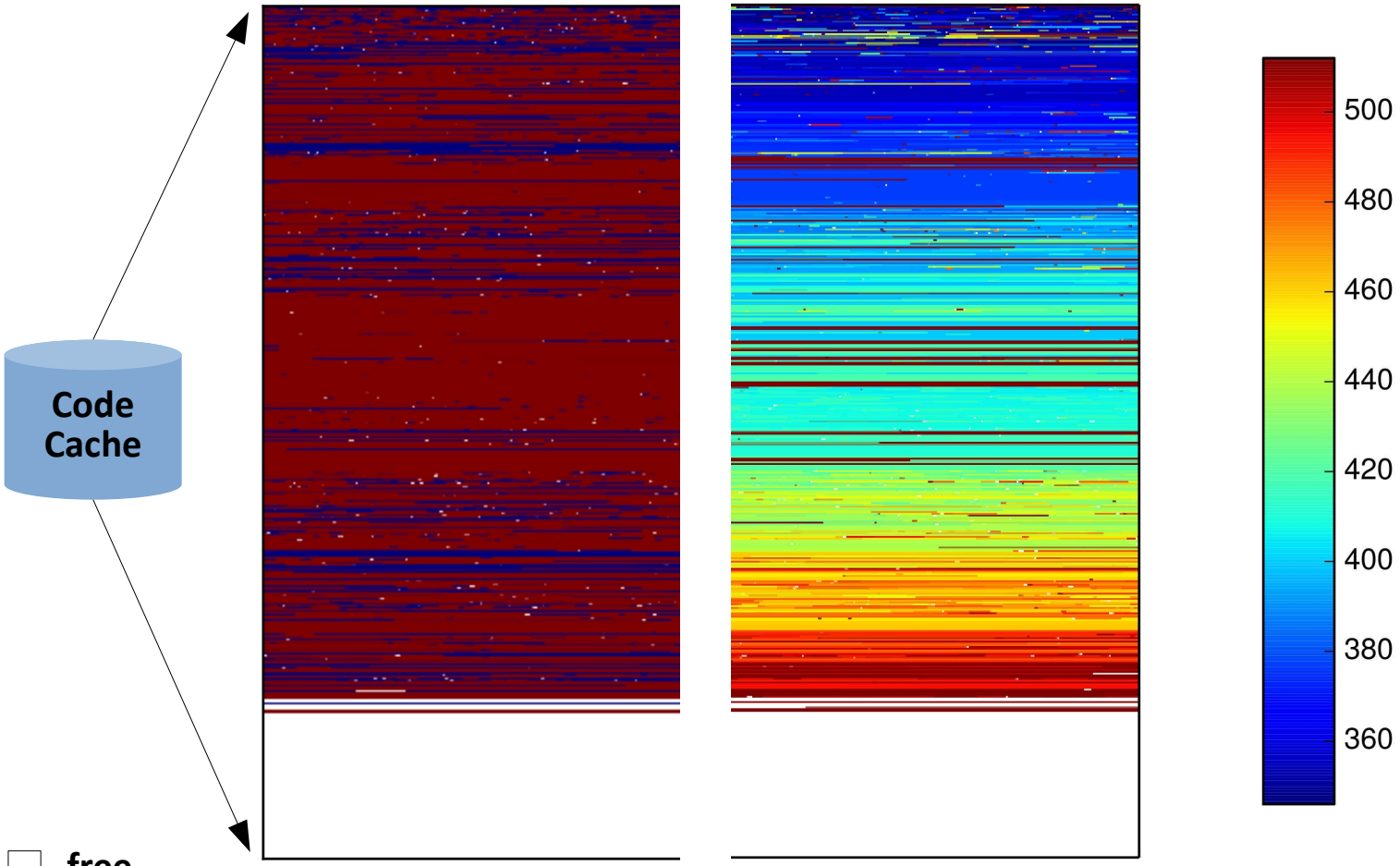
Types of compiled code

- **Non-method code**
- **Profiled method code**
  - Instrumented (C1)
  - **Limited lifetime**
- **Non-profiled method code**
  - Highly optimized (C2)
  - **Long lifetime**

Code cache fragmentation

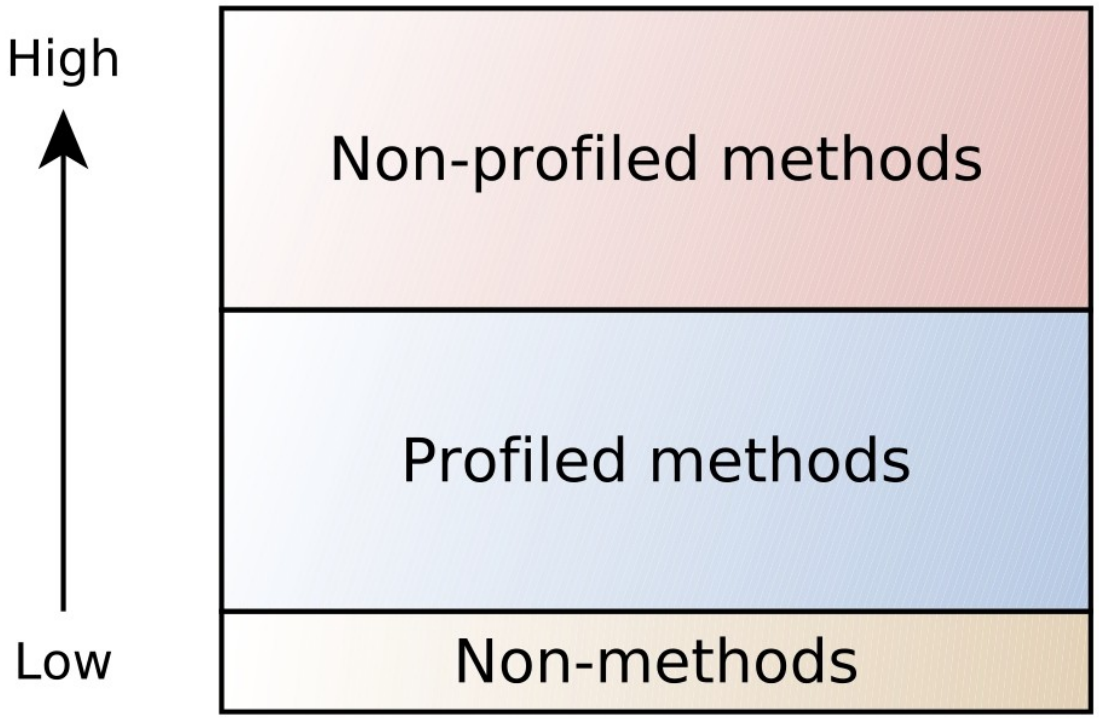


Hotness of code



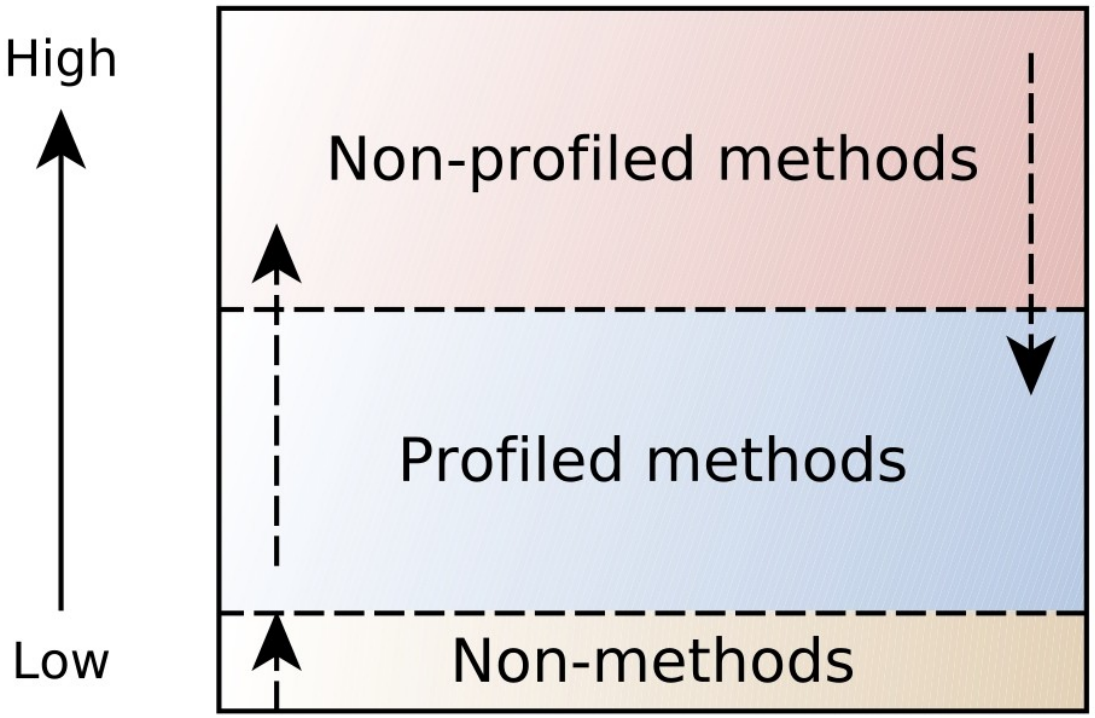
Segmented code cache

- **Dividing code cache into distinct segments**



Dynamic resizing

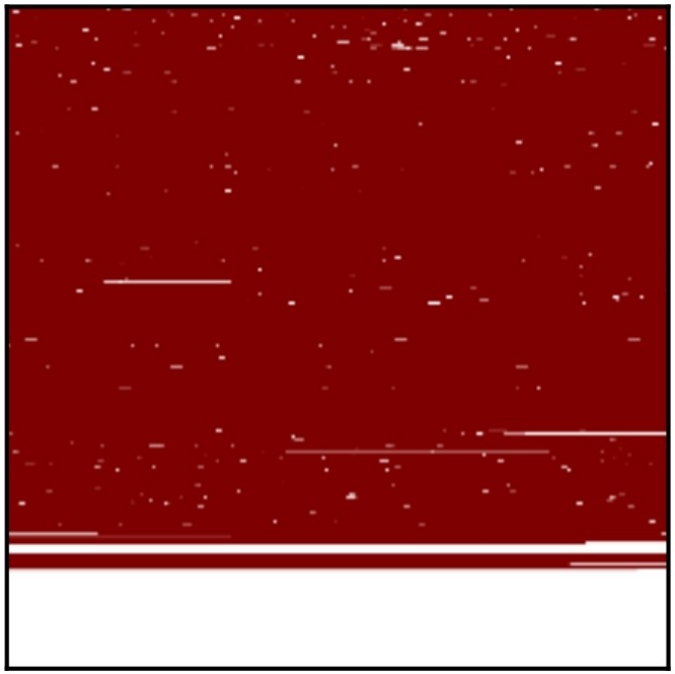
- **Allowing the segments to resize**

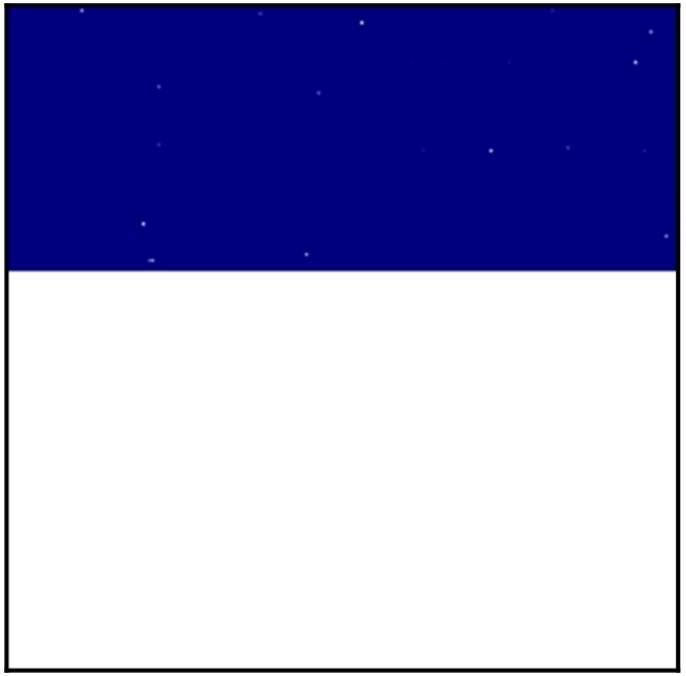


Implementation

- **Two prototype implementations**
  - Fully functional
  - With and without **resizing**
- **Corner cases**
  - Small code cache sizes
  - Different compiler configurations
  - Code cache sweeper
- **Several optimizations possible**

Code cache fragmentation



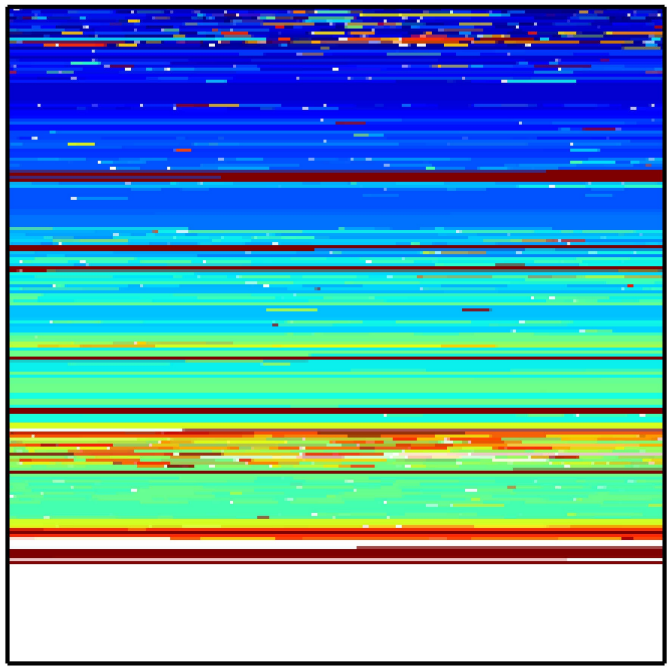


free

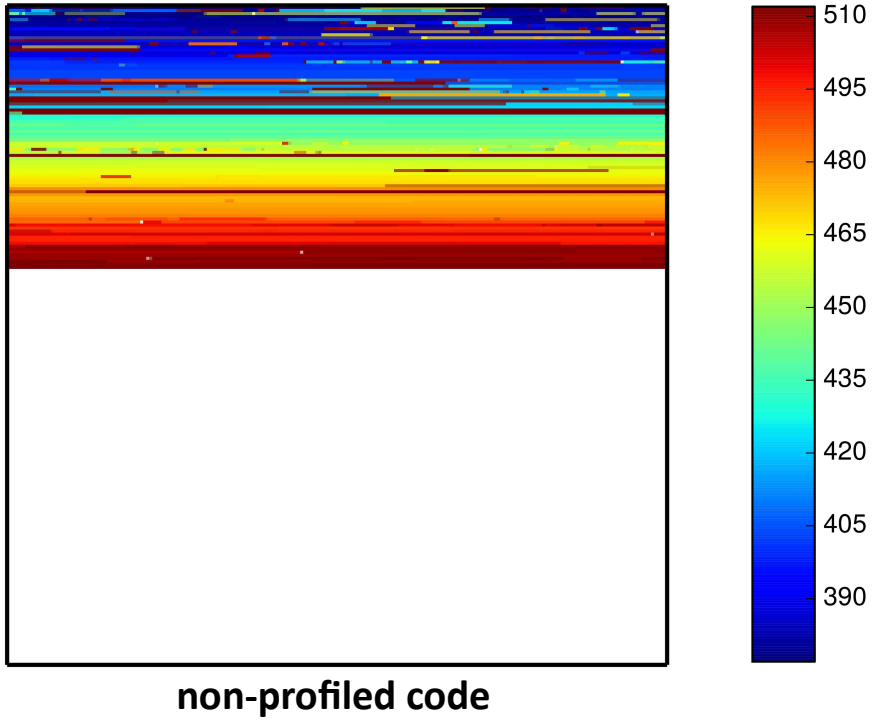
non-profiled code

profiled code

Hotness of code



profiled code



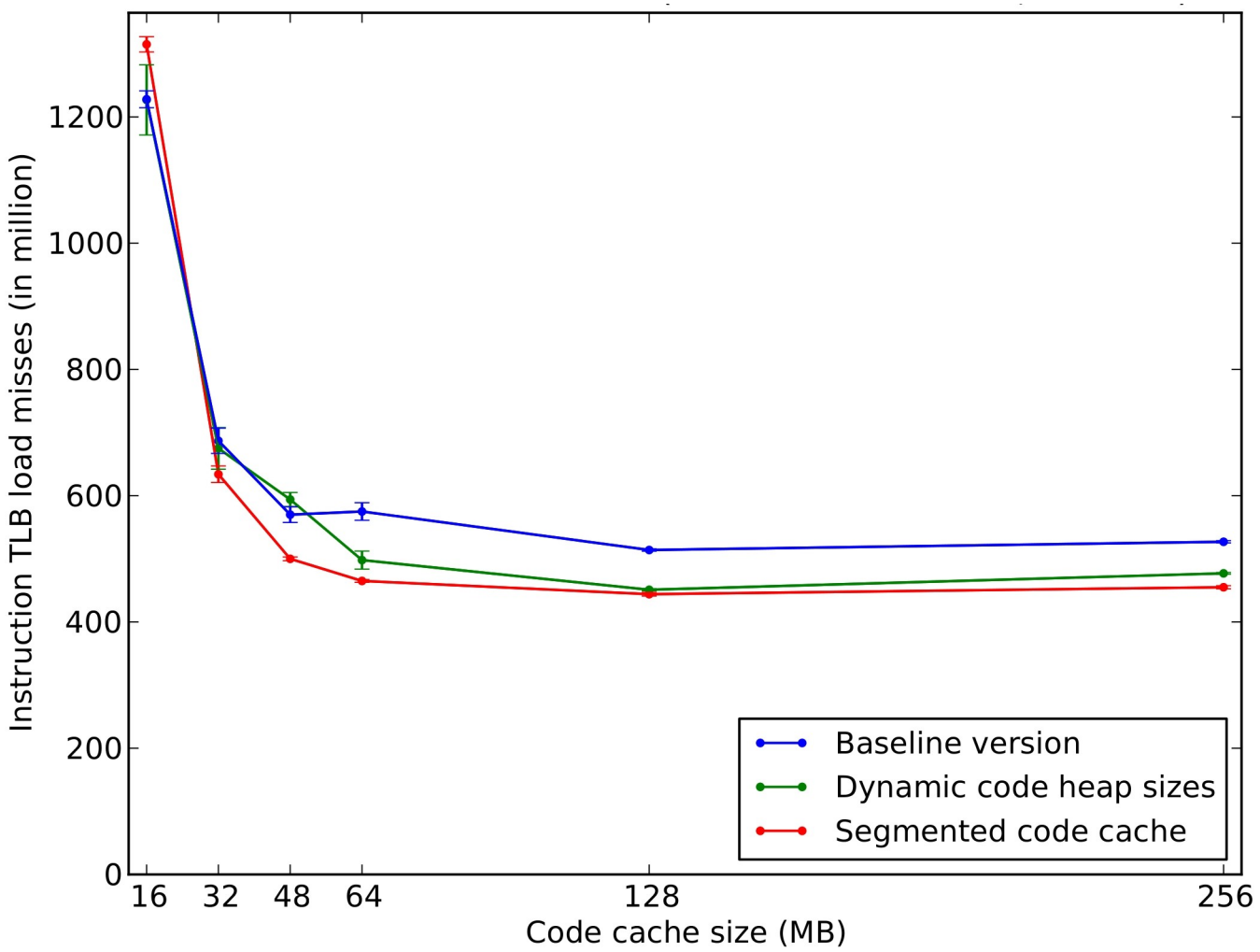
non-profiled code

Performance evaluation

- **Segmented code cache**
- **Segmented code cache with dynamic resizing**

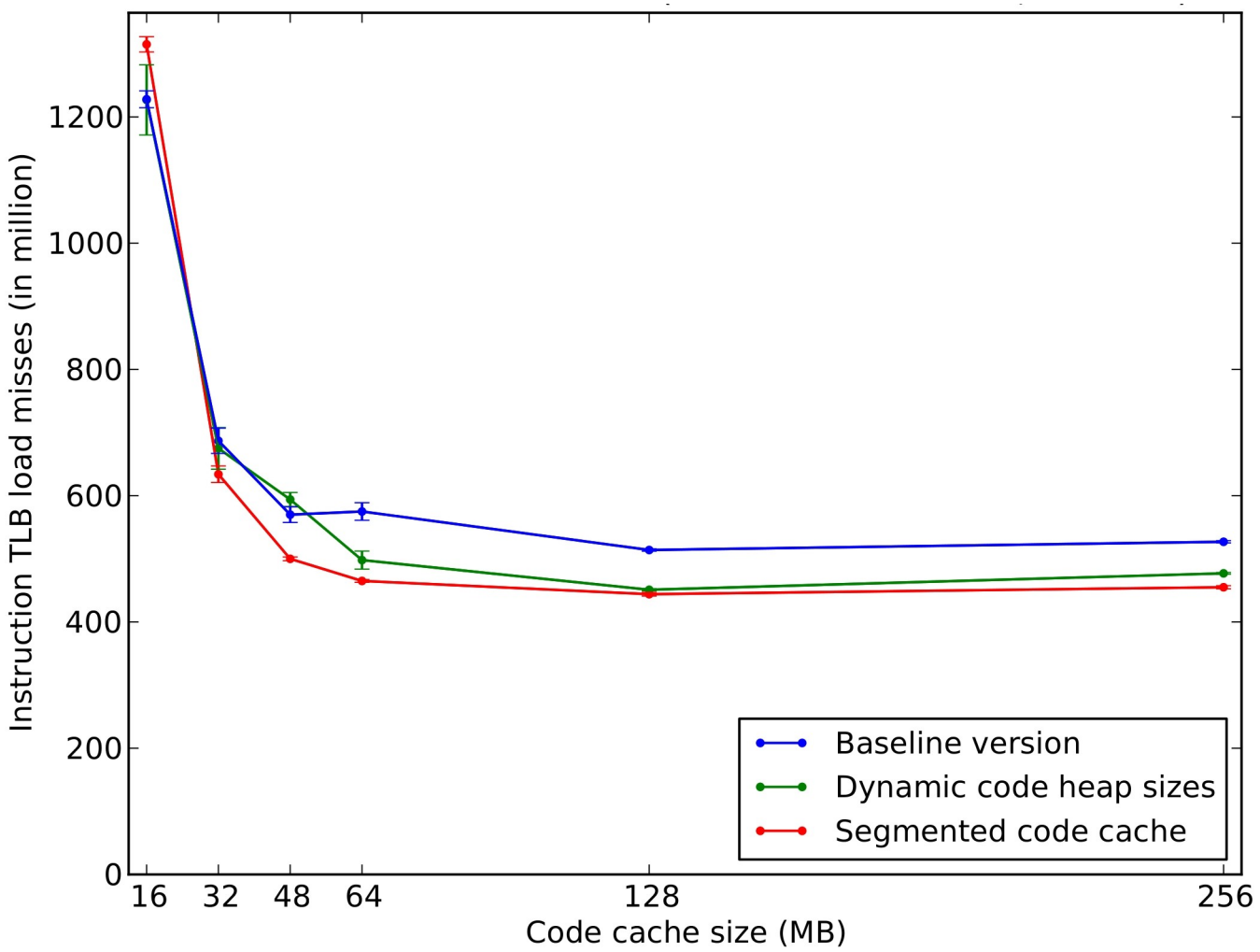
- **Hardware setup**
  - 4 Intel Xeon E7-4830 CPUs at 2.13 GHz with 24 MB cache
  - 64 GB main memory

Instruction TLB

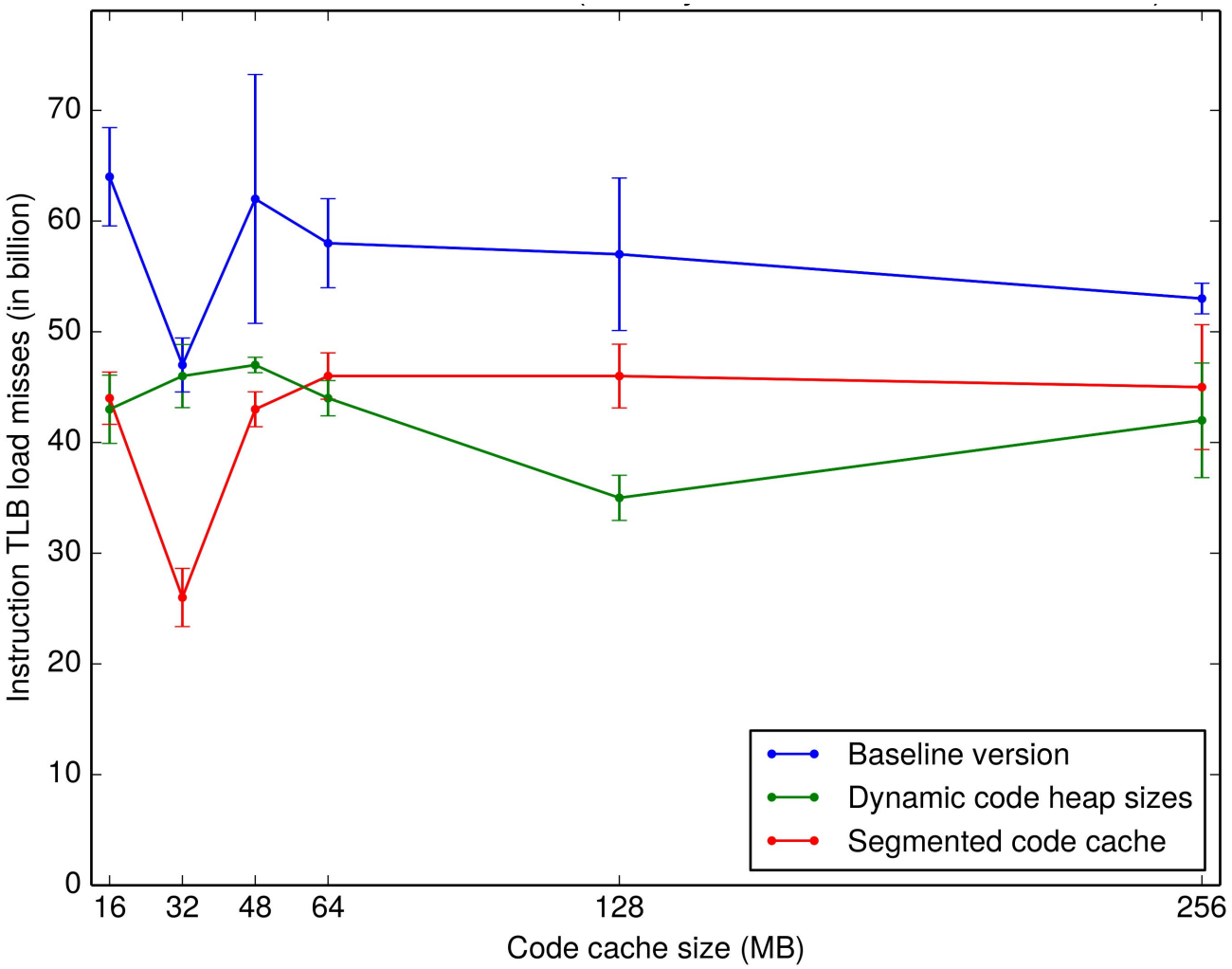


Instruction TLB

- 19 %

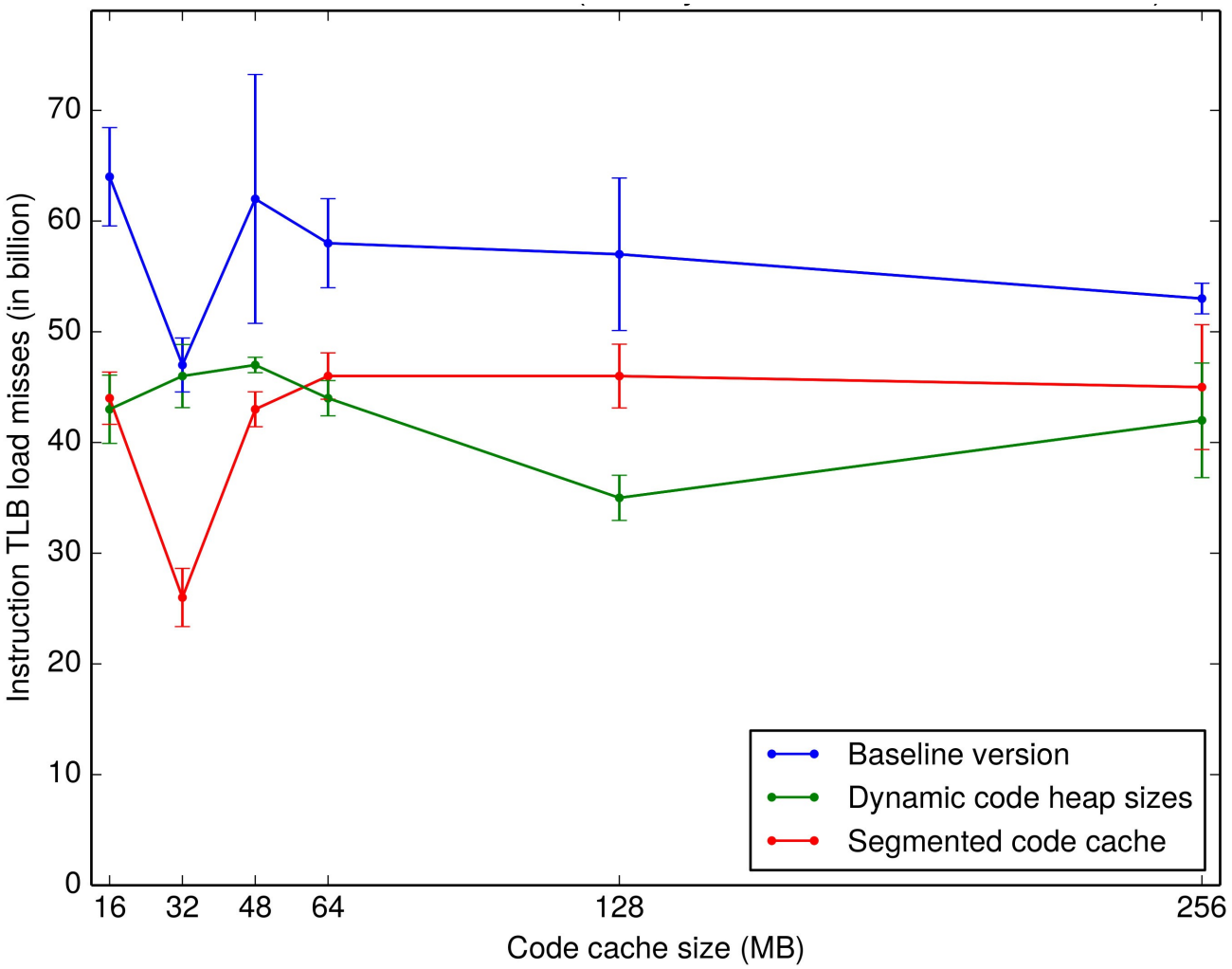


Instruction TLB (long running)

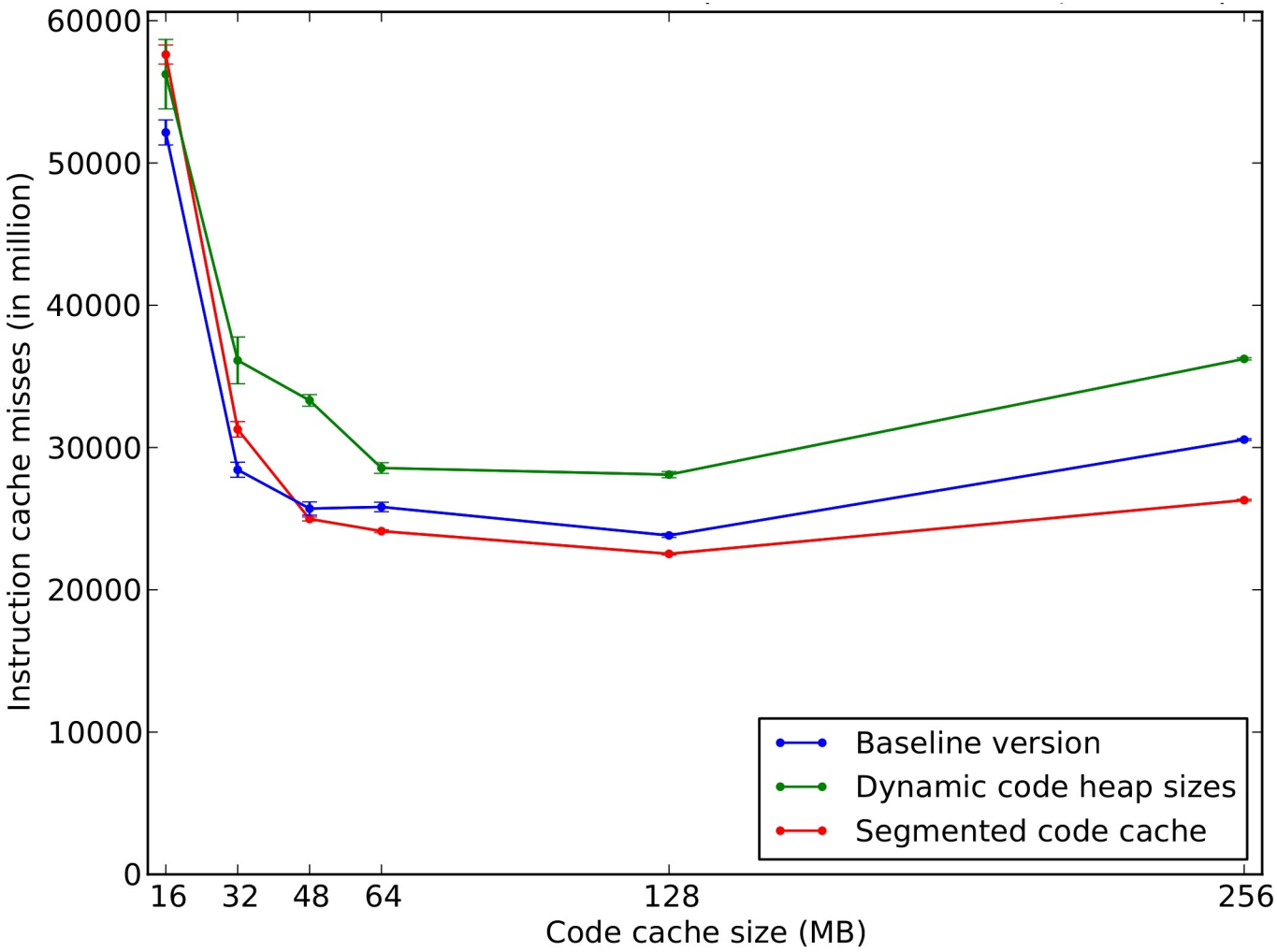


Instruction TLB (long running)

- 44 %

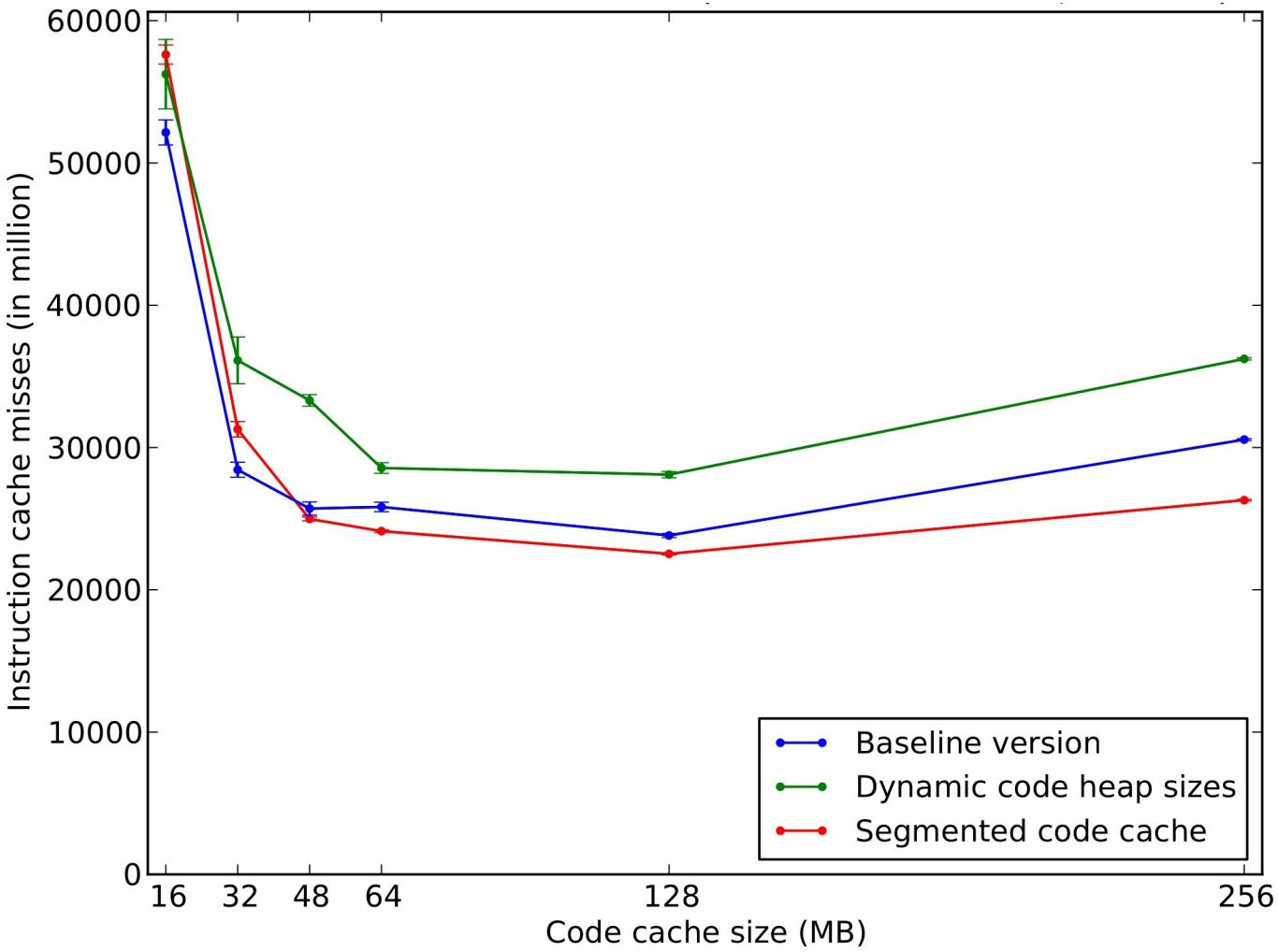


Instruction cache

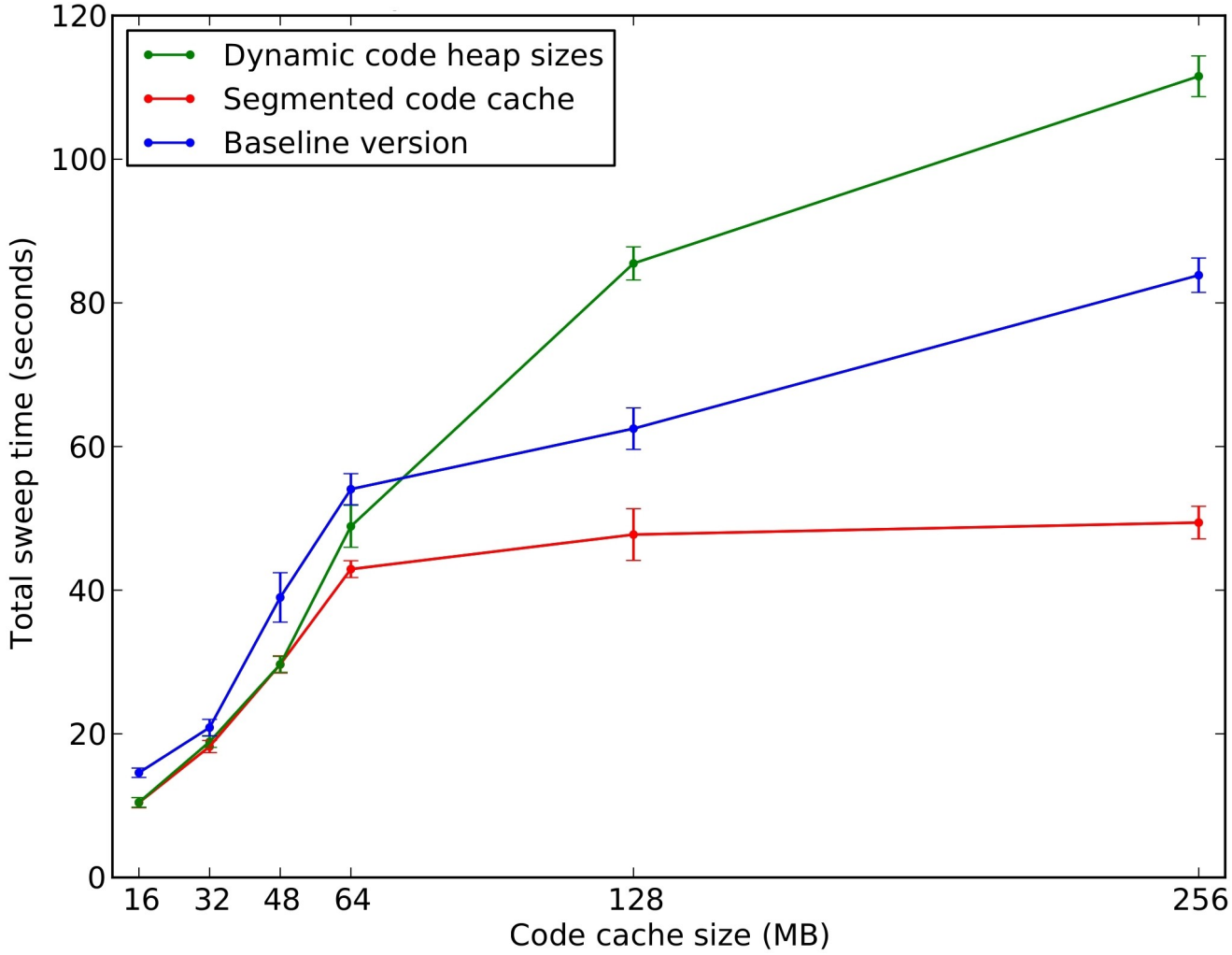


Instruction cache

- 14 %

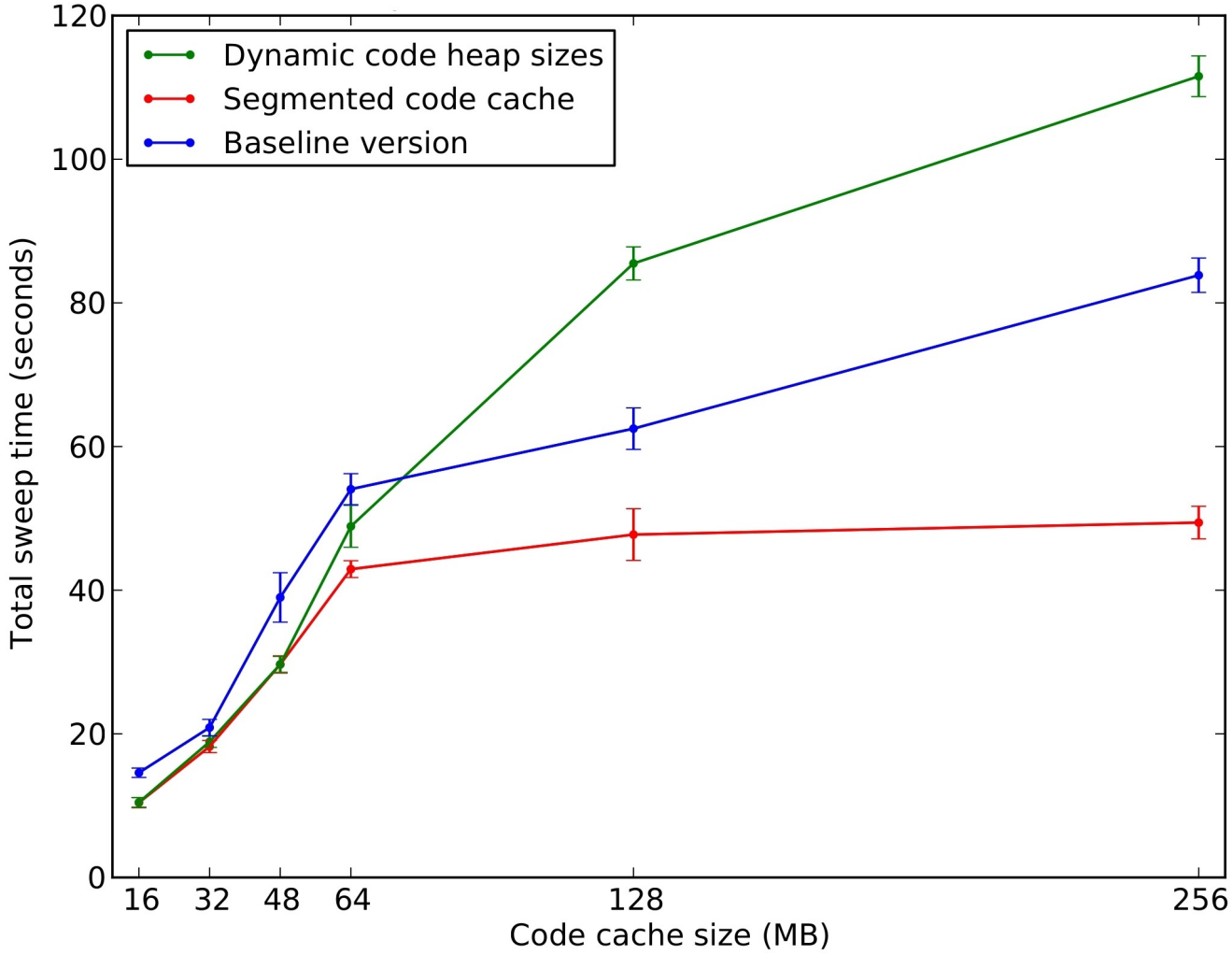


Sweep time

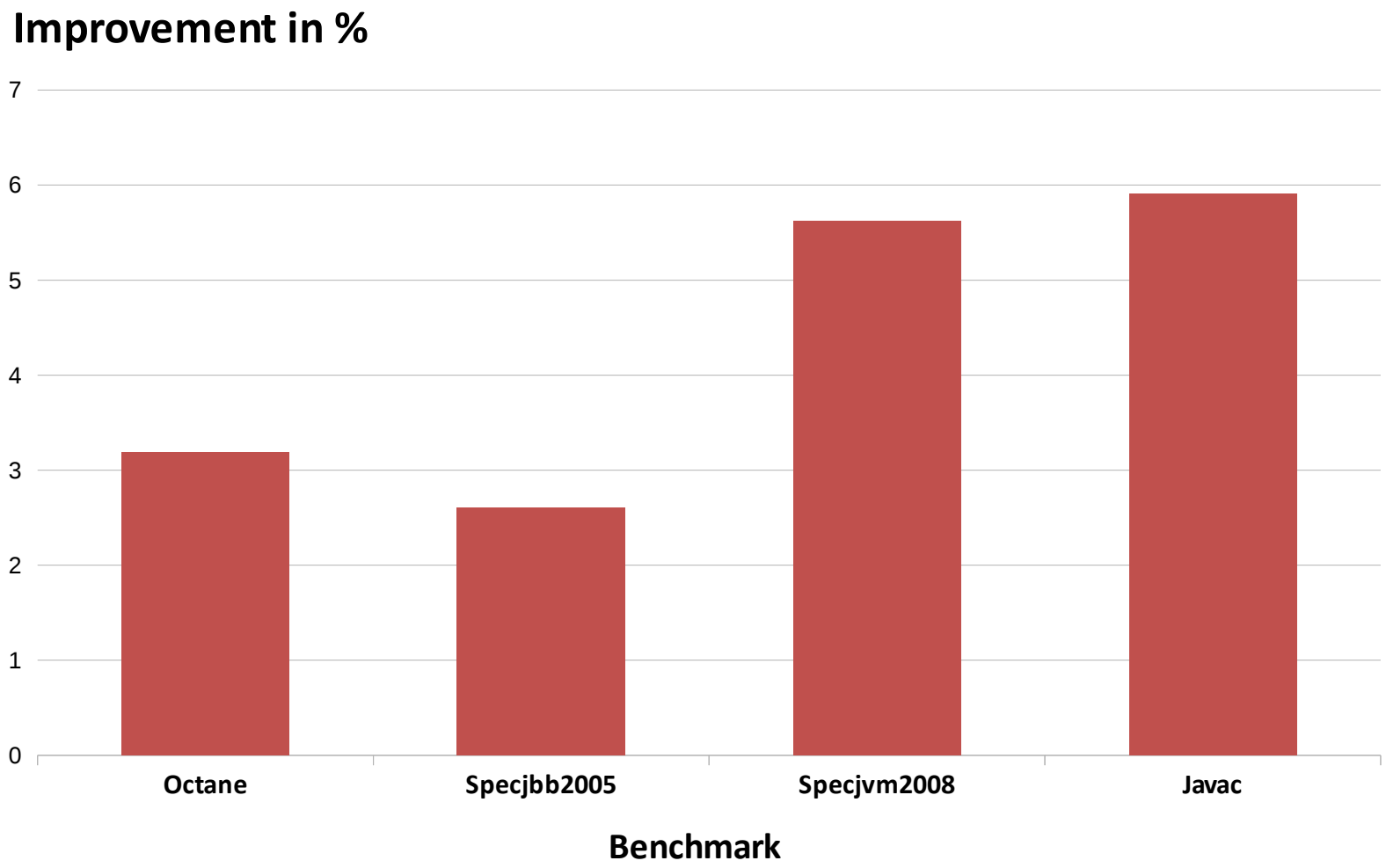

Dynamic code heap sizes
Segmented code cache
Baseline version
Total sweep time (seconds)
0
20
40
60
80
100
120
16
32
48
64
128
256
Code cache size (MB)

Sweep time

- 46 %



Execution time



Evaluation summary

- **Performance improvement for regular sizes**
  - Execution time: up to 6%
  - Sweep time: up to 46%
  - Fragmentation: up to 98%
  - iTLB and iCache miss rates: up to 44%, 14%
- **Resizing does not pay off**
- **Only enable segmentation with**
  - Tiered compilation
  - Large code cache (> 240 MB)

Conclusion

- **Organization of code cache important**
  - Code locality
  - Fragmentation
- **Impact on overall performance**
- **Fully integrated into latest version**
  - Including tool support
  - Integration into JDK 9 in process

Future work

- **Separation of code and metadata**
- **Fine grained sweeping**
  - Sweep profiled code heap more often
- **Code heap partitioning**
- **Heterogeneous code**
  - More code heaps

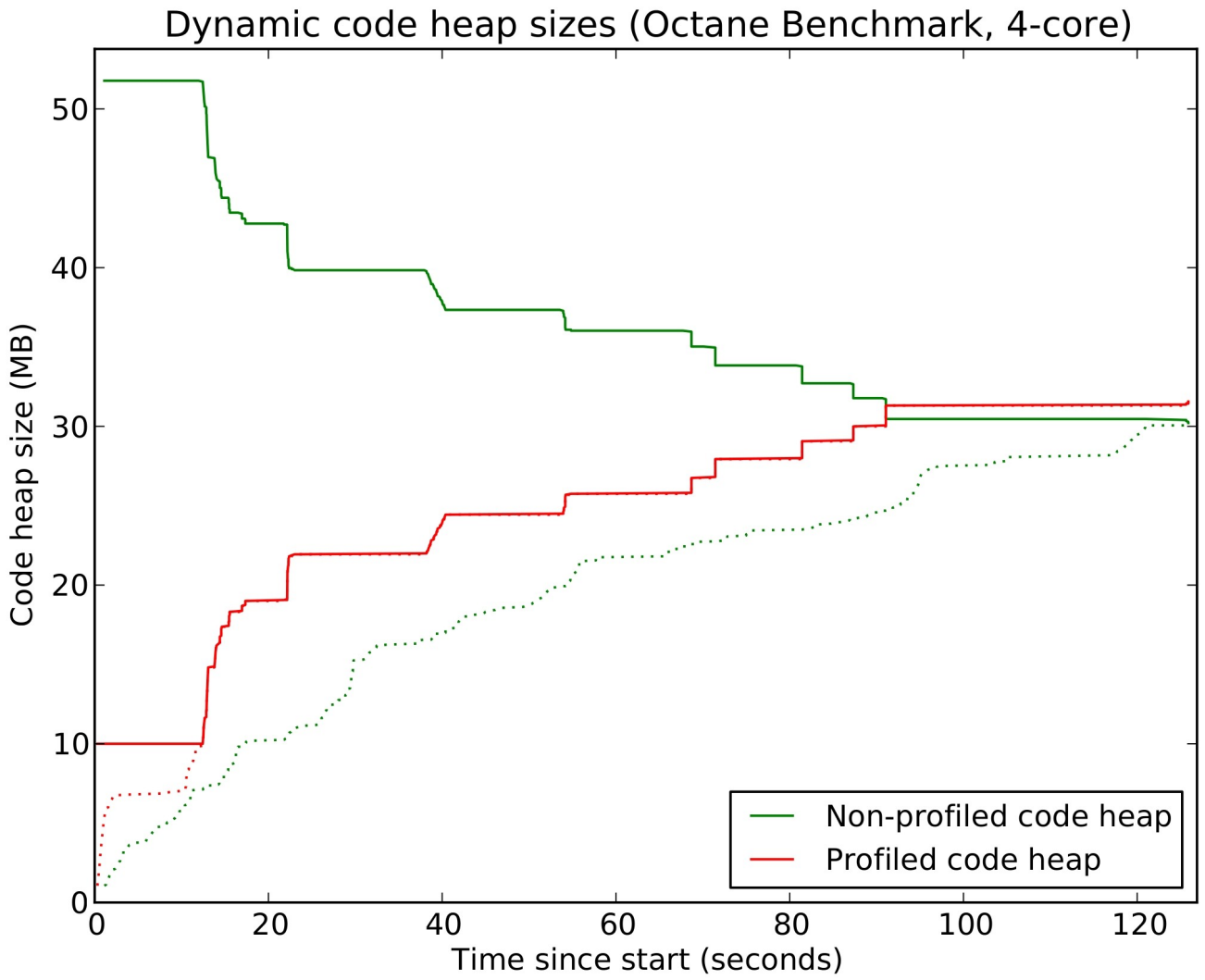
Thank you for your attention!

http://openjdk.java.net/jeps/197

Related work

- **Java Virtual Machines**
  - Jikes RVM
  - Maxine JVM
  - Dalvik JVM
- **Dynamic Binary Translators**
  - **Generational** code cache [Hazelwood and Smith]
- **Garbage collectors**

Resizing of code heaps



Resizing of code heaps