Profile Caching for the Java Virtual Machine

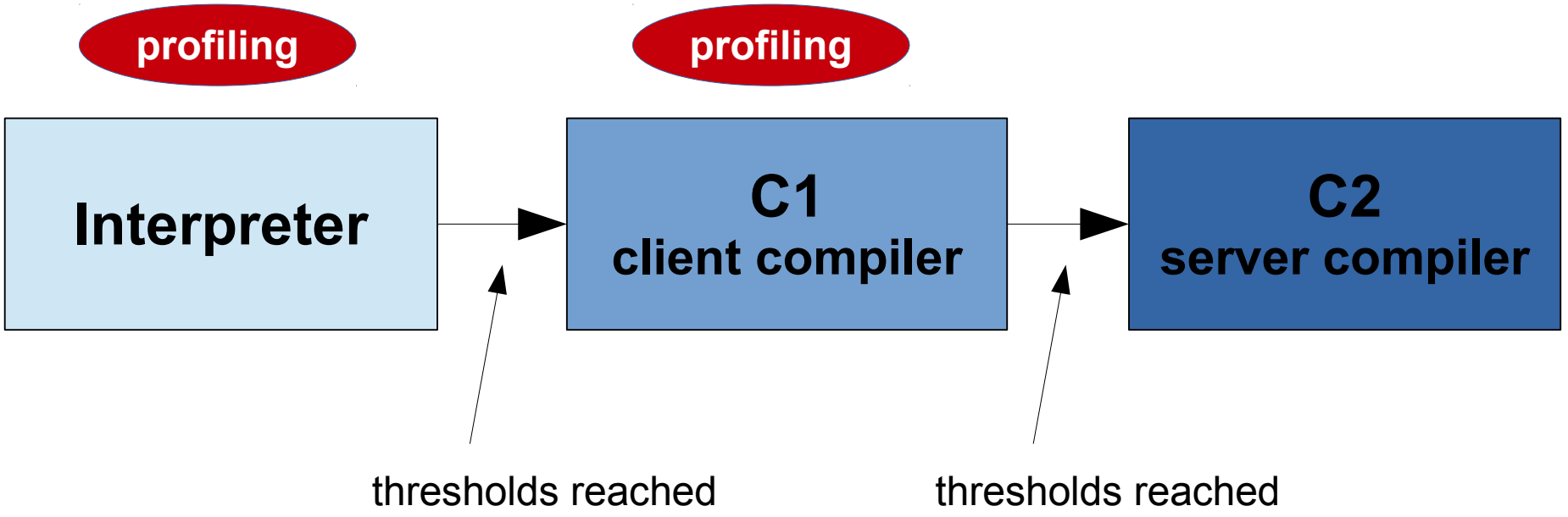
Marcel Mohler, ETH Zurich

Bachelor Thesis

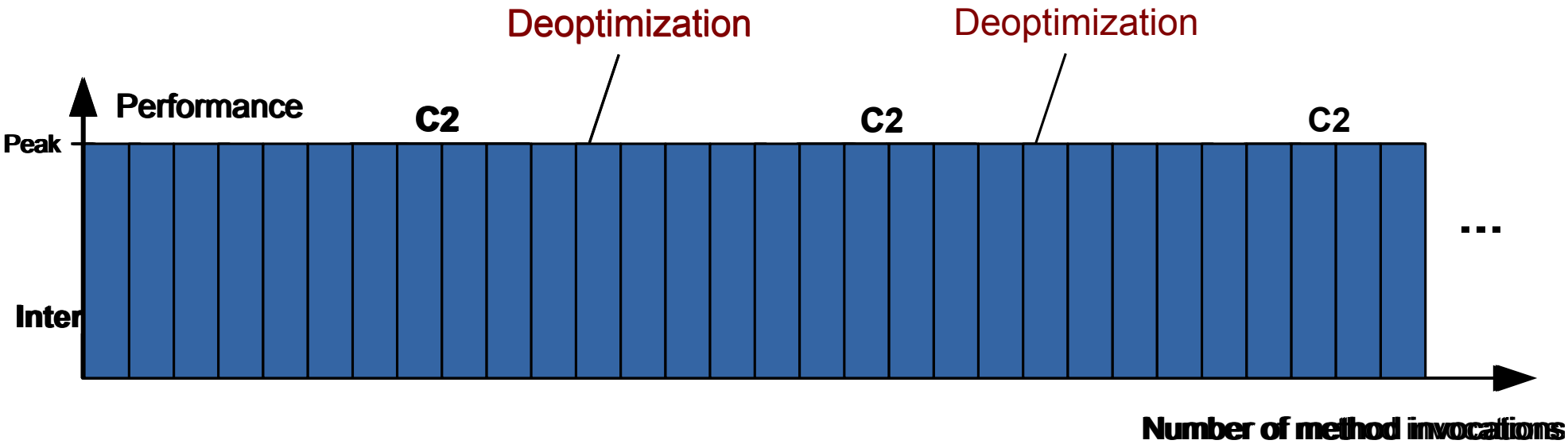
Supervisors: Zoltan Majo, Oracle
Tobias Hartmann, Oracle

Prof. Thomas Gross, Laboratory for Software Technology, ETH

Tiered Compilation in HotSpot JVM



Problem: performance fluctuations



Goal: Decrease performance fluctuations

Observation

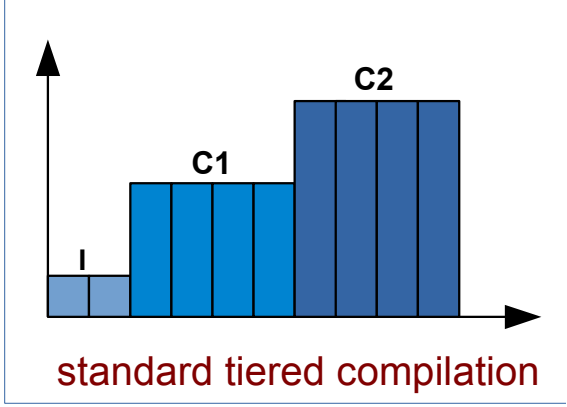
- Profiles need to be gathered each time the JVM starts
- Most frequently used methods often do not change

→ Idea: cache and reuse the profiles!

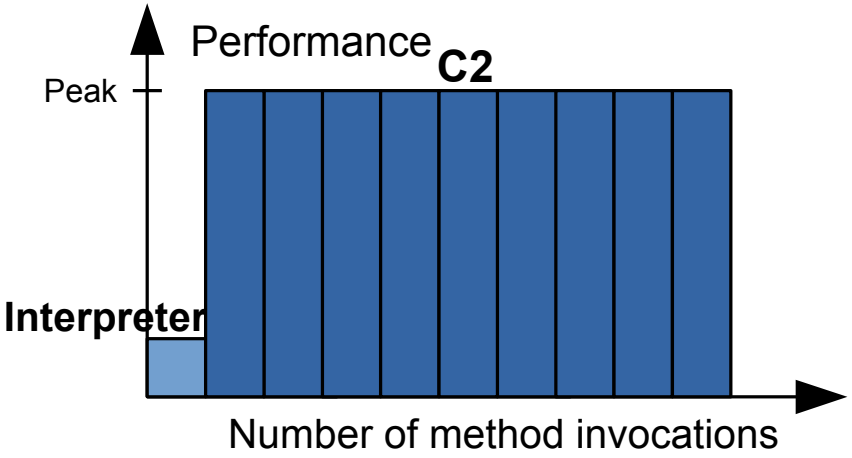
Outline

1. Design
2. Implementation
3. Performance evaluation
4. Conclusion

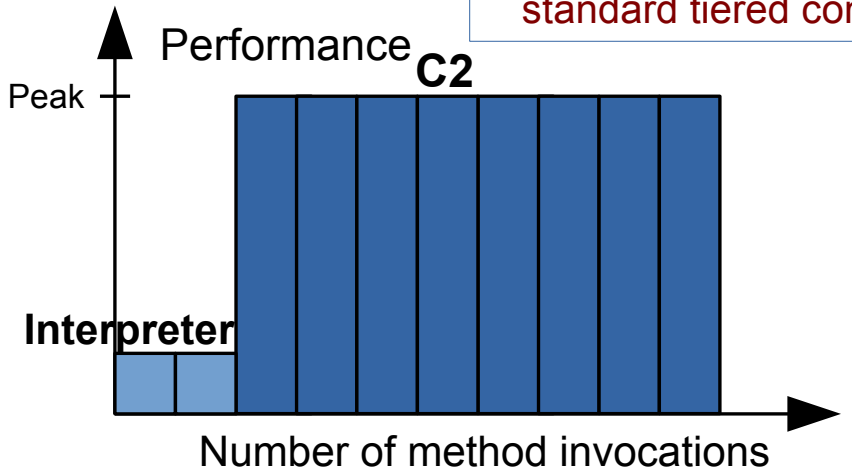
Design: 3 modes



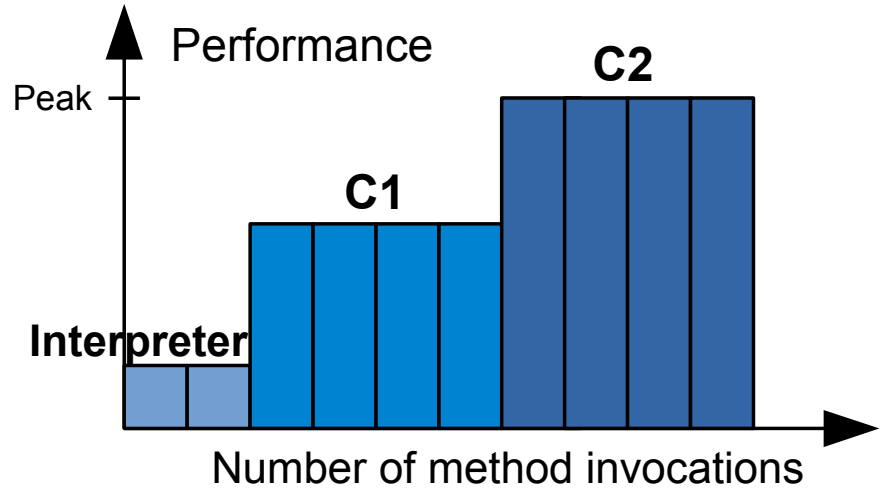
standard tiered compilation



Mode 0

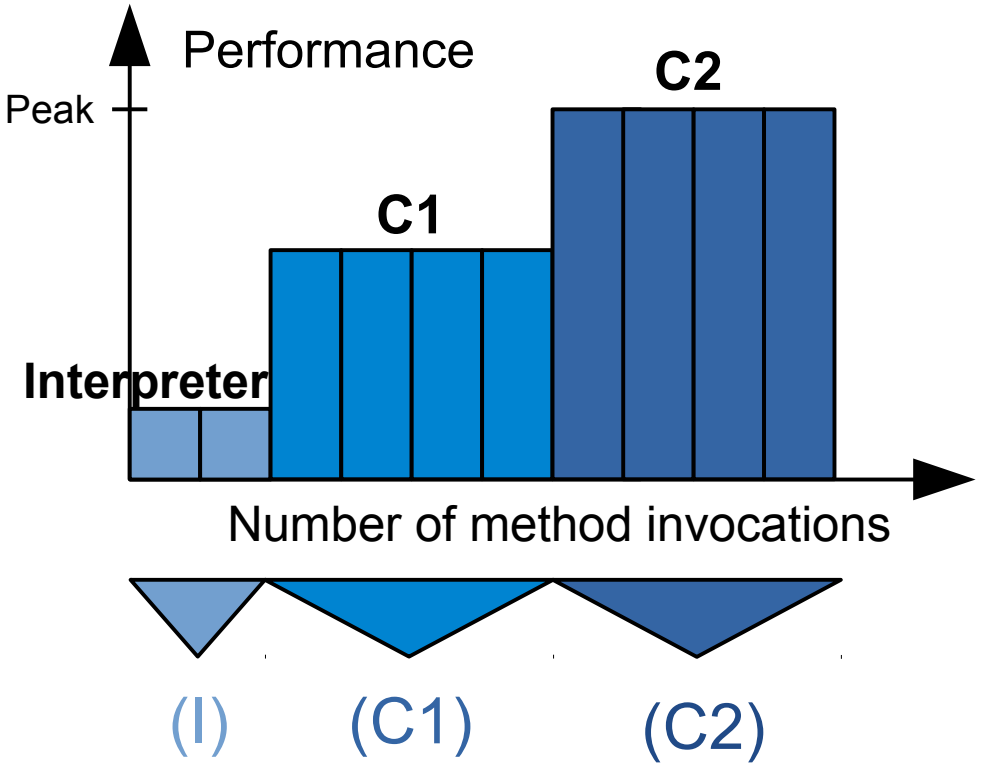


Mode 1



Mode 2

Design: Mode 2



(I) unmodified

(C1) remove profiling code

→ ~30% speedup

(C2) use cached profile

→ better code quality

Implementation

- Based on existing compilation replay functionality
- Configurable by JVM flags
- Select all or an arbitrary set of methods

Evaluation

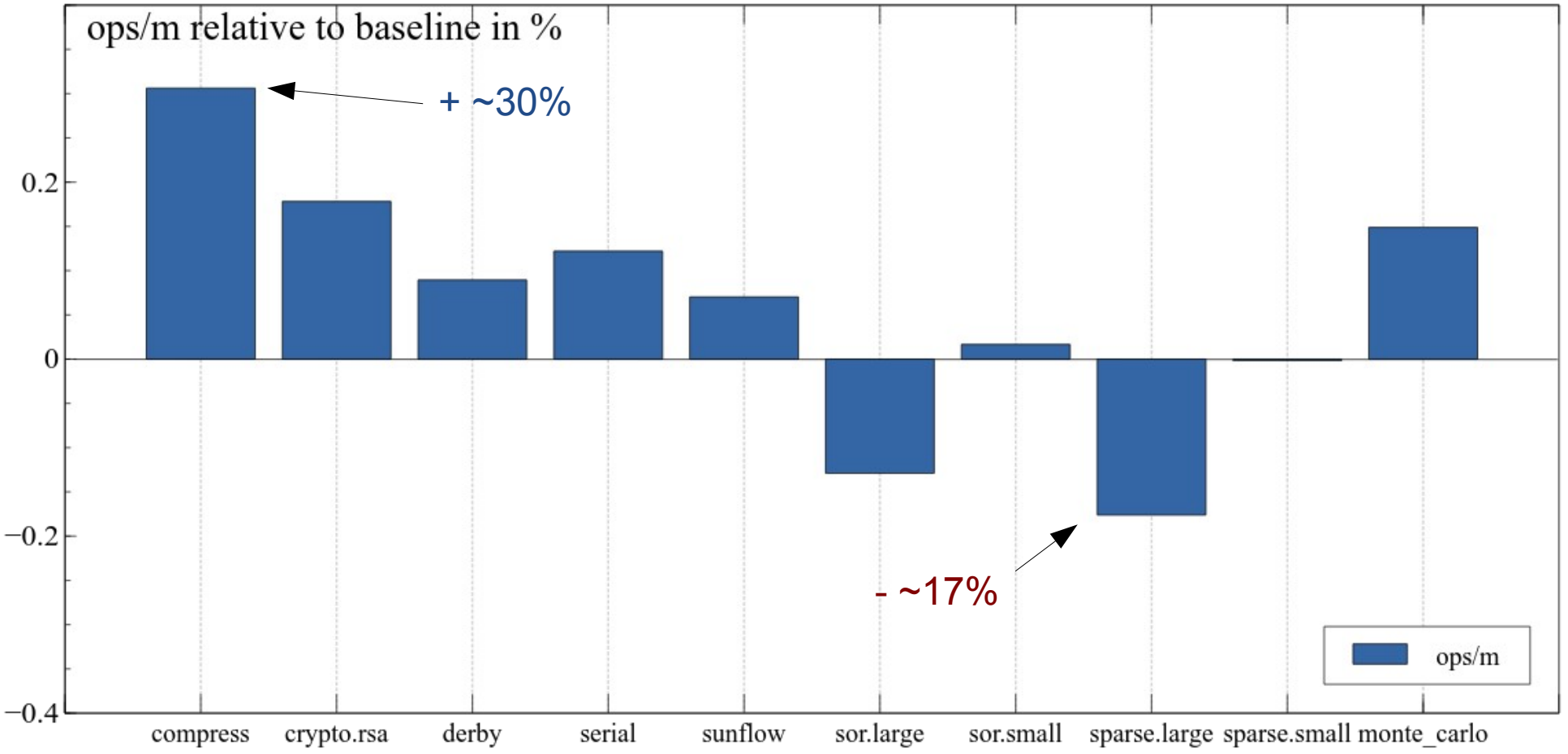
- ETH Data Center Observatory
- Focus on warmup
- 2 benchmark suites
  - SPECjvm 2008

    17/21 individual benchmarks used
  - Google Octane (using Nashorn)

    16/17 individual benchmarks used

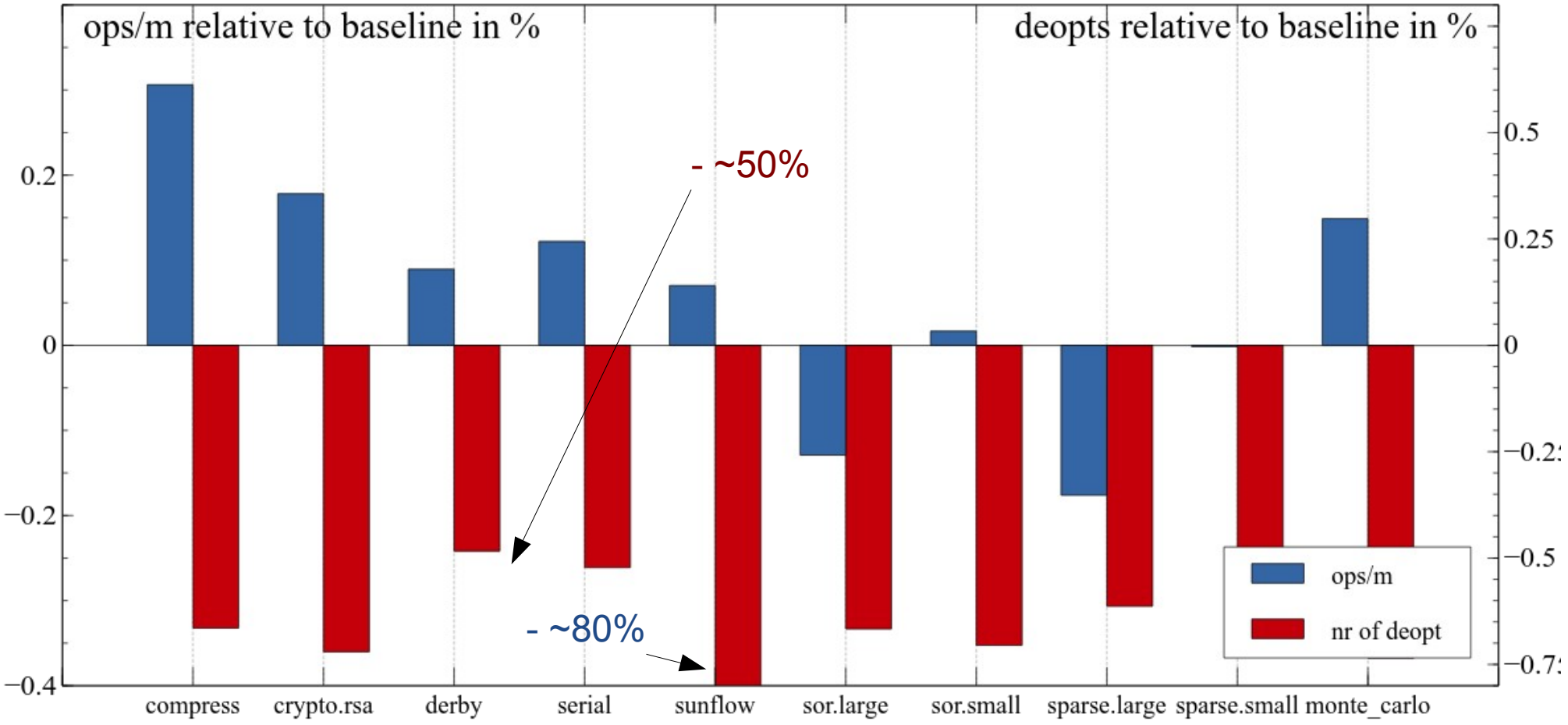
Performance evaluation

- Performance (higher is better)



Performance evaluation

- Deoptimizations (lower is better)

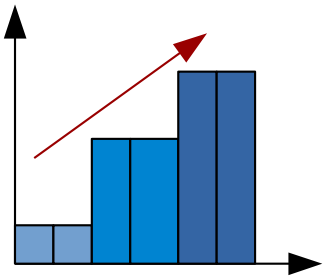


Other benchmark results

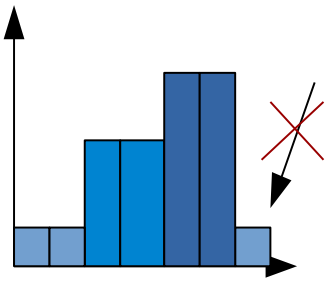
- Benefit mainly from cached C2 compilations. Disabling interpreter profiles rarely affects performance
- Around 70% of the compilations use cached profiles
- No association between performance and load on compile queue

Conclusion

- Cached profiles can improve warmup performance
- System allows fine tuning
- Main benefits:



faster warmup



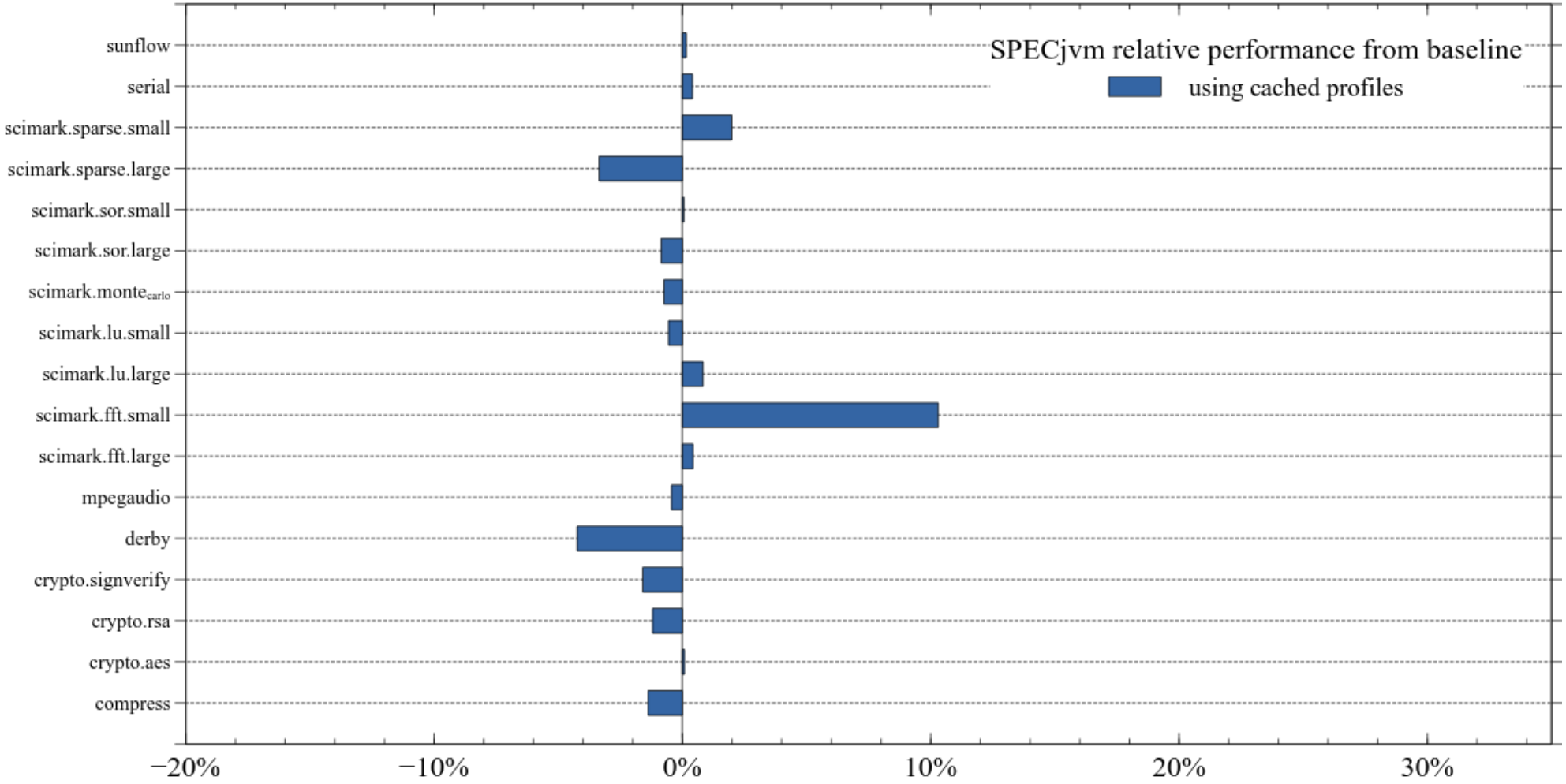
fewer deoptimizations

- Room for future work

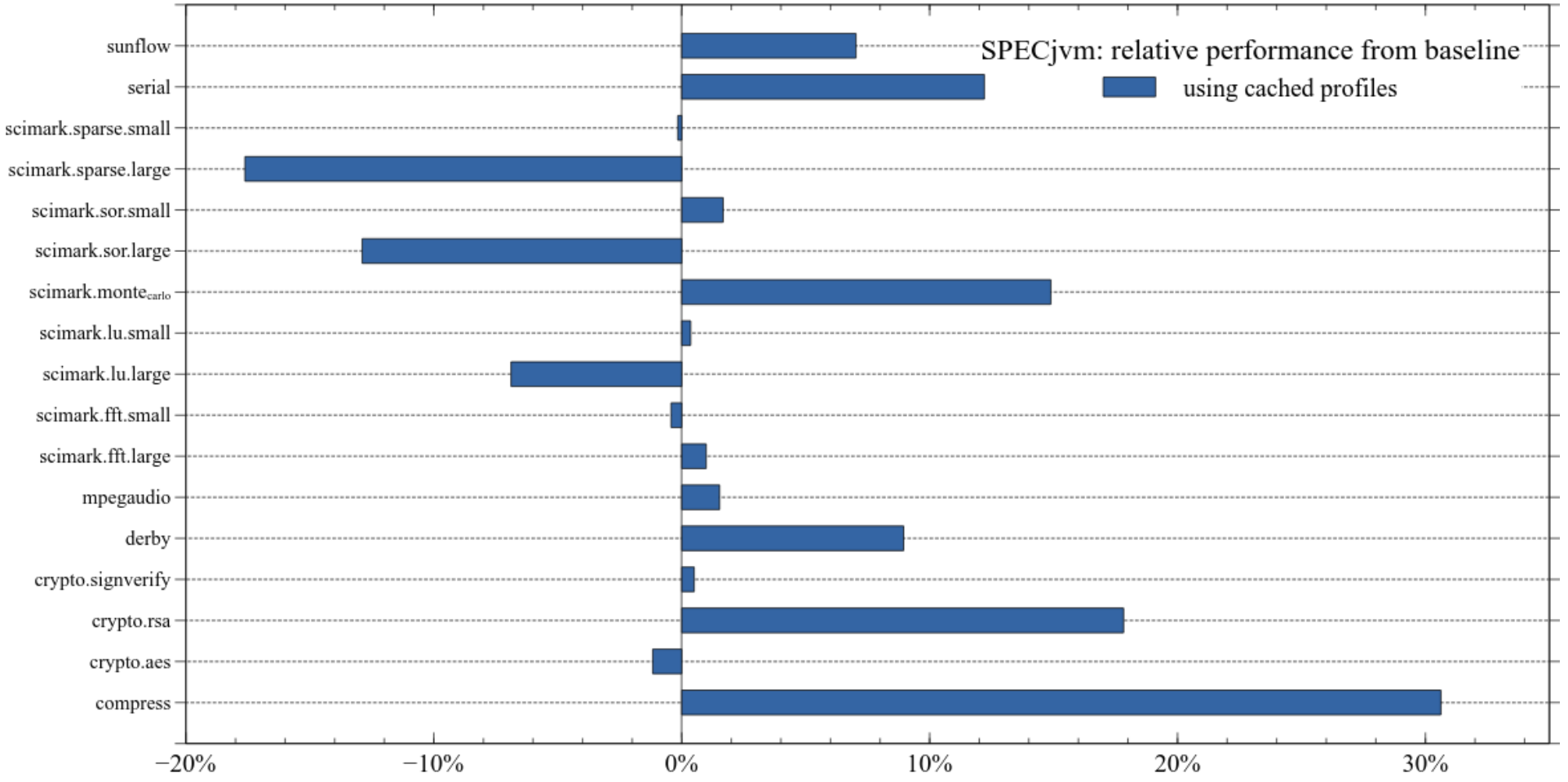
End

Additional graphes follow

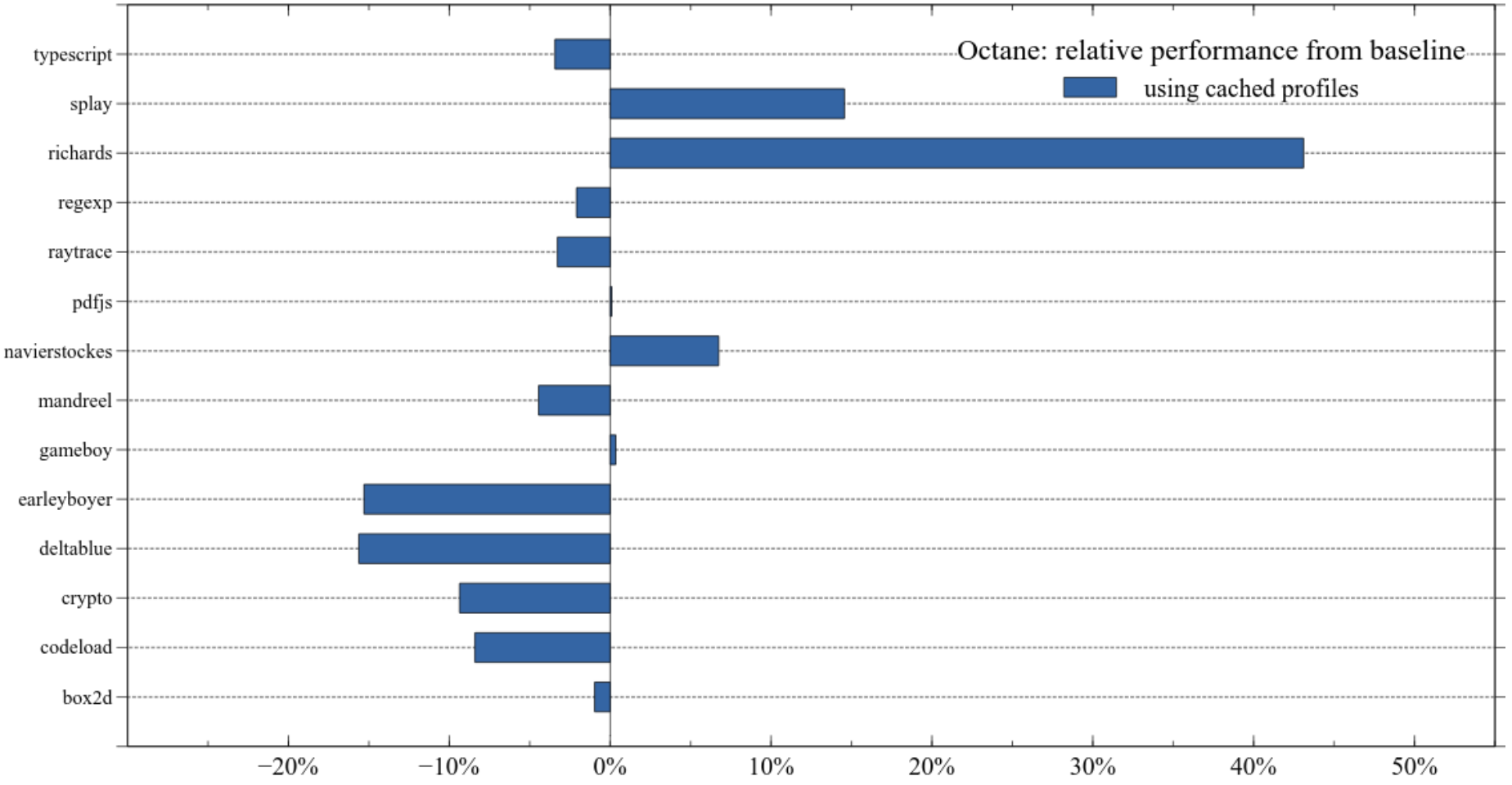
SPECjvm: relative performance full runs (2+4 minutes)



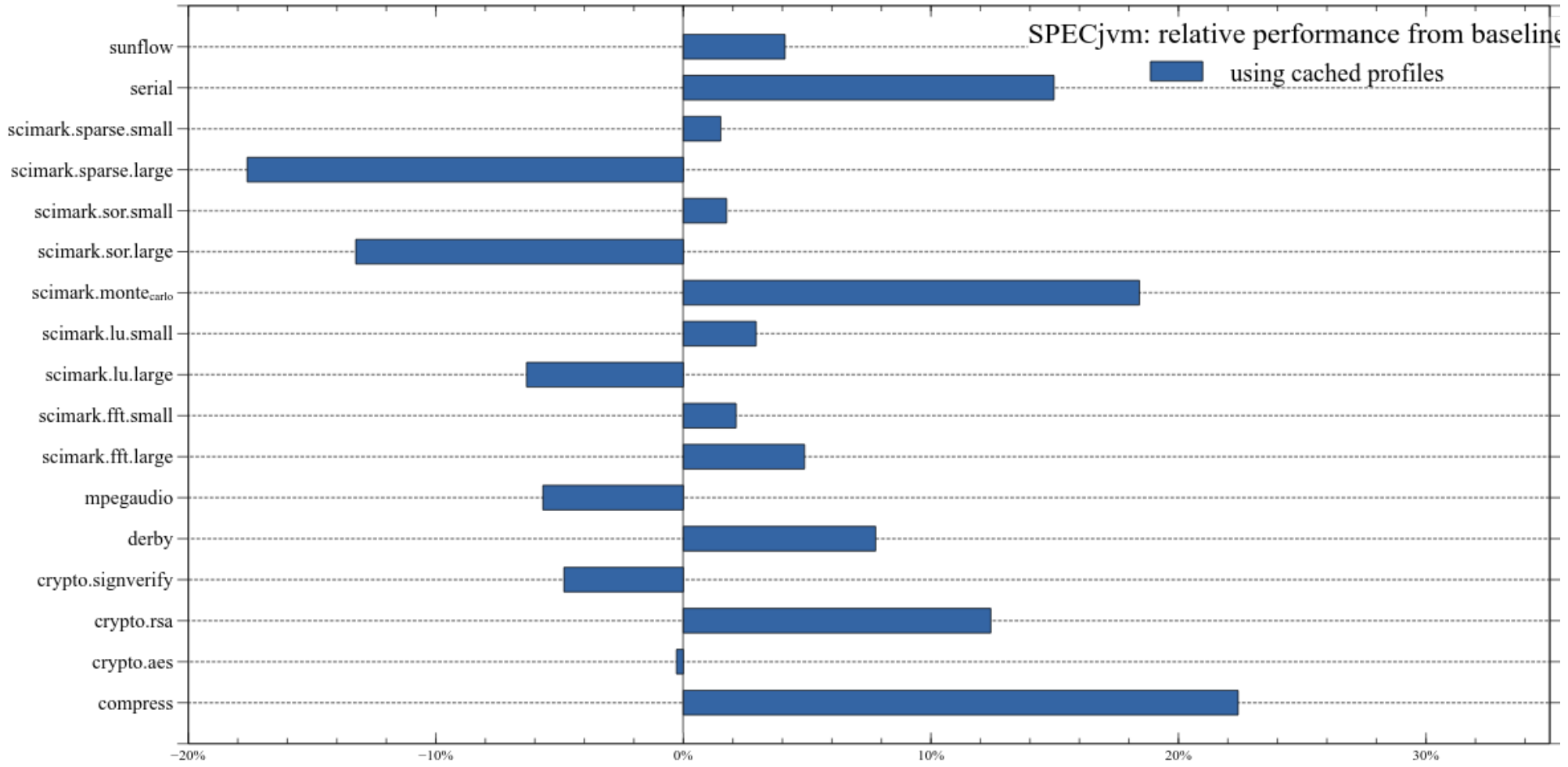
SPECjvm: relative performance warmup: all benchmarks



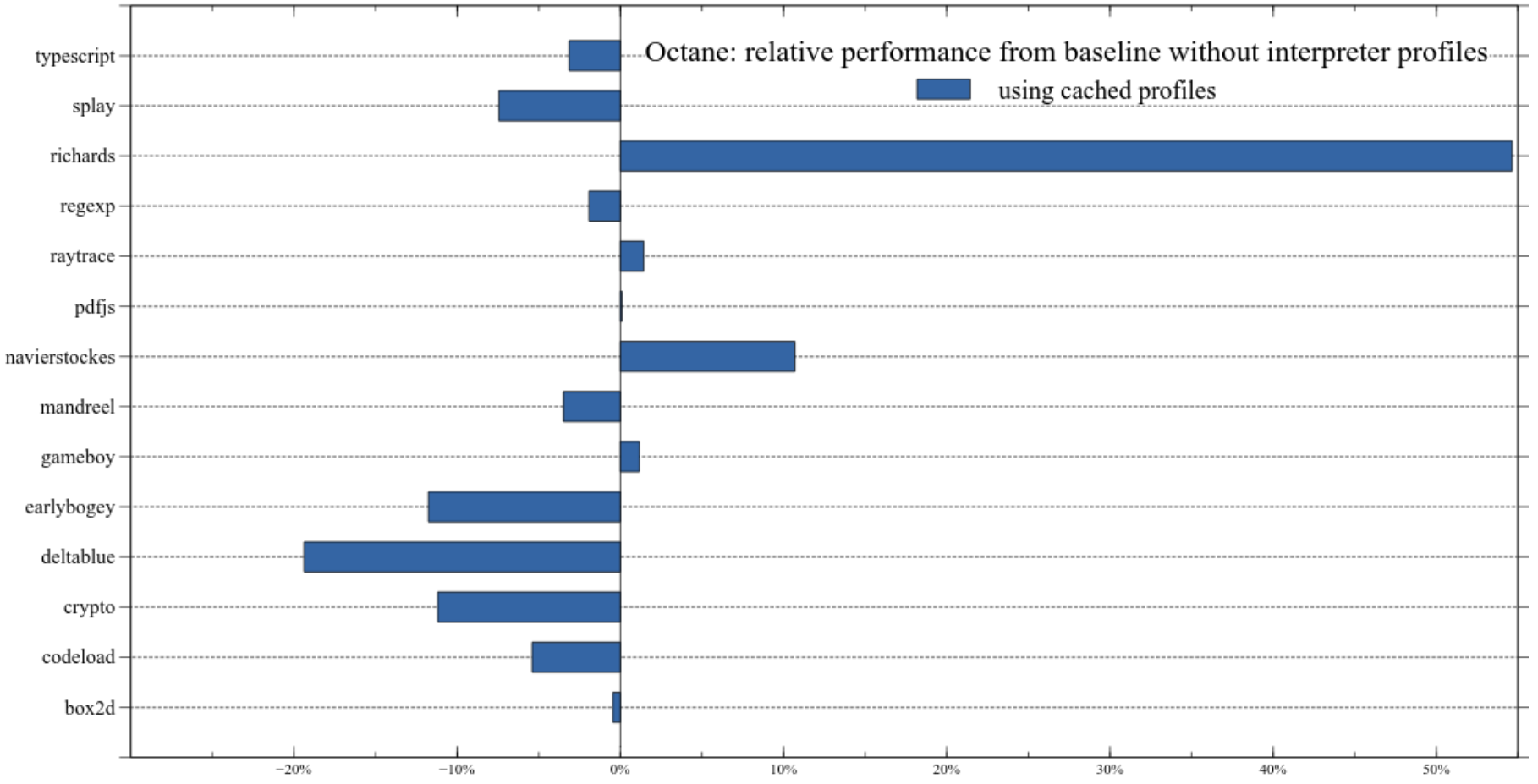
Octane: relative performance all benchmarks



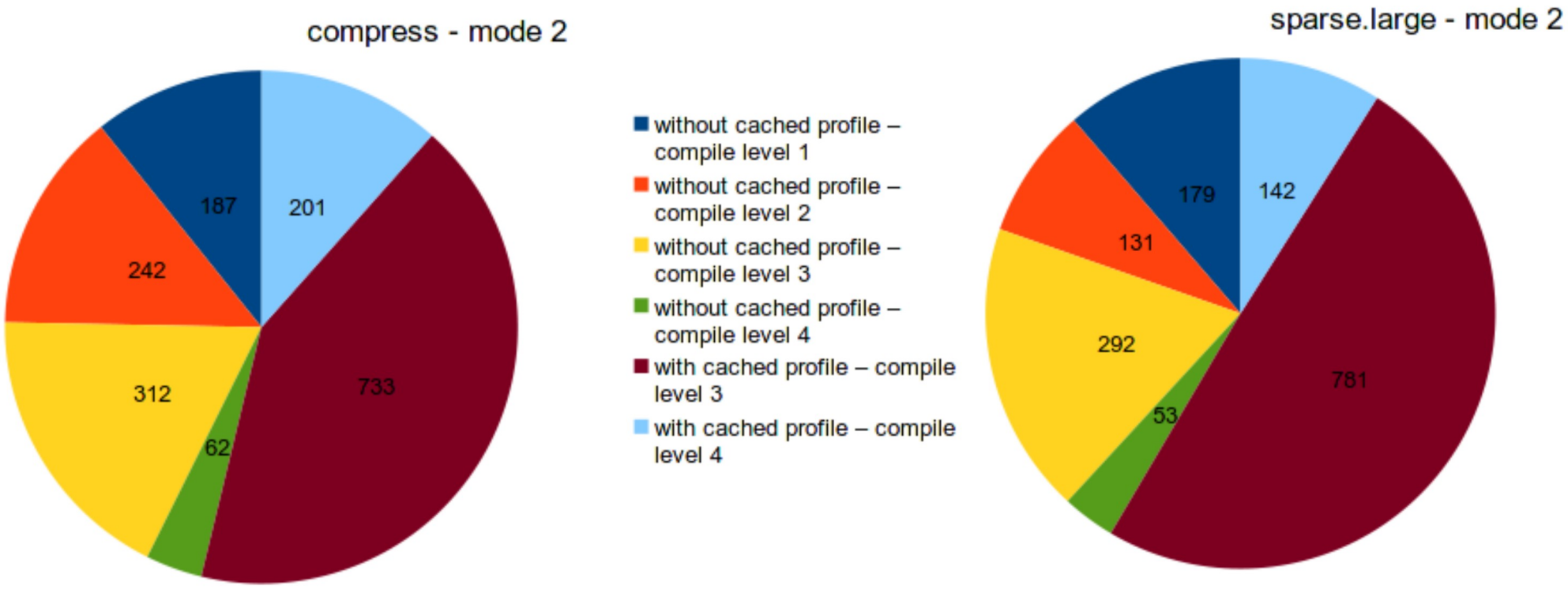
SPECjvm: relative performance without interpreter profiles



Octane: relative performance without interpreter profiles

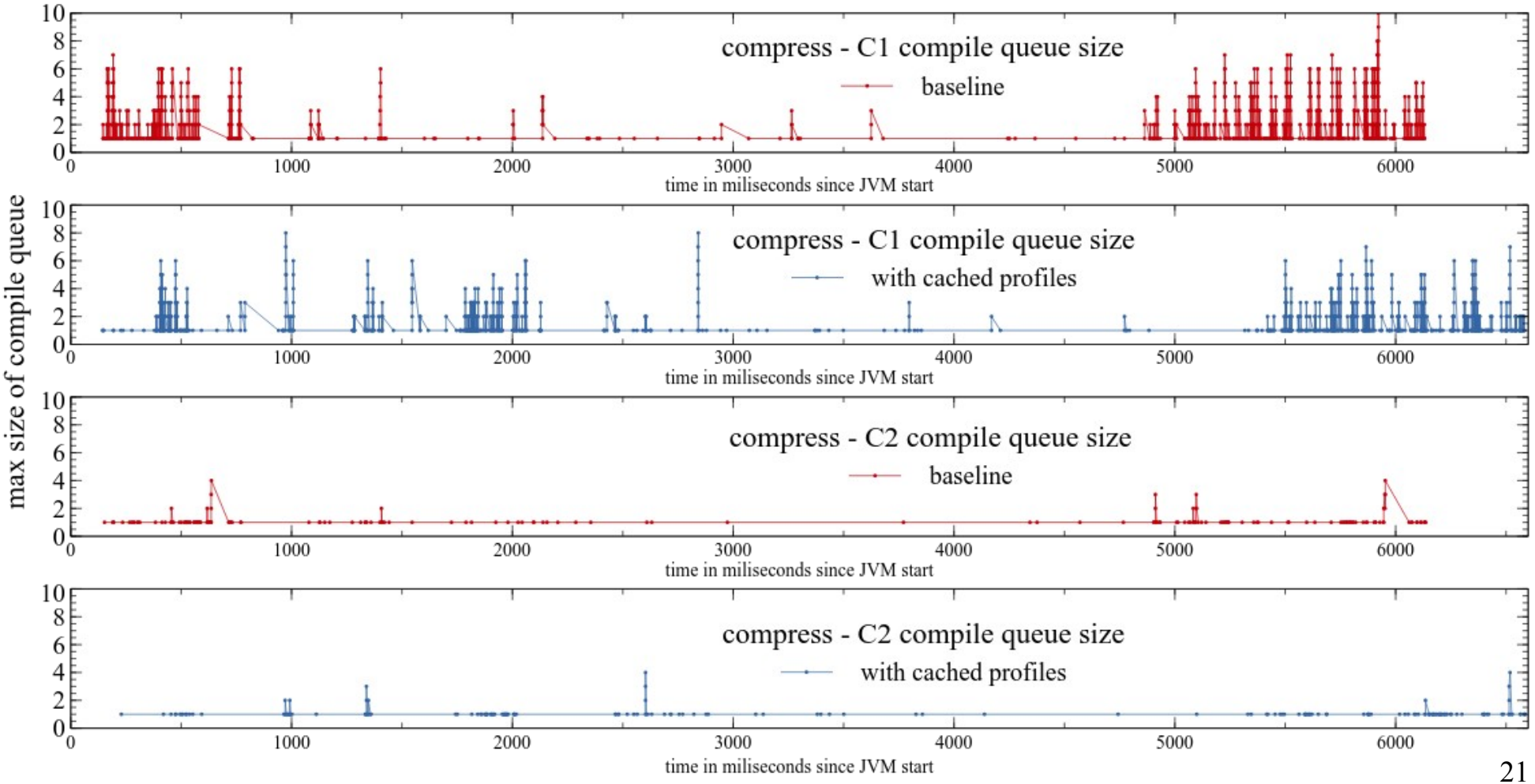


Type of compilations



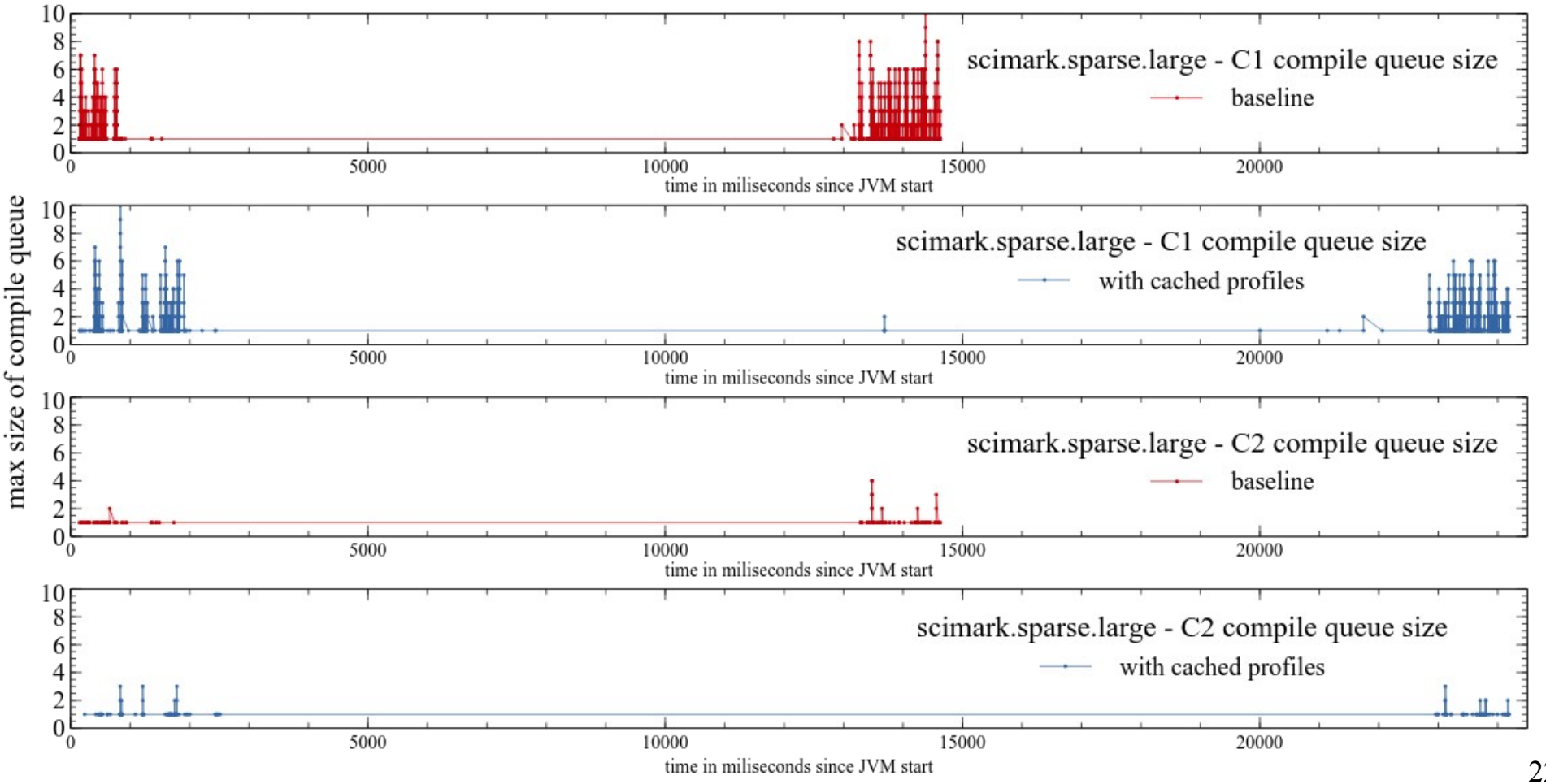
Compile queue

- compress



Compile queue

- scimark.sparse.large



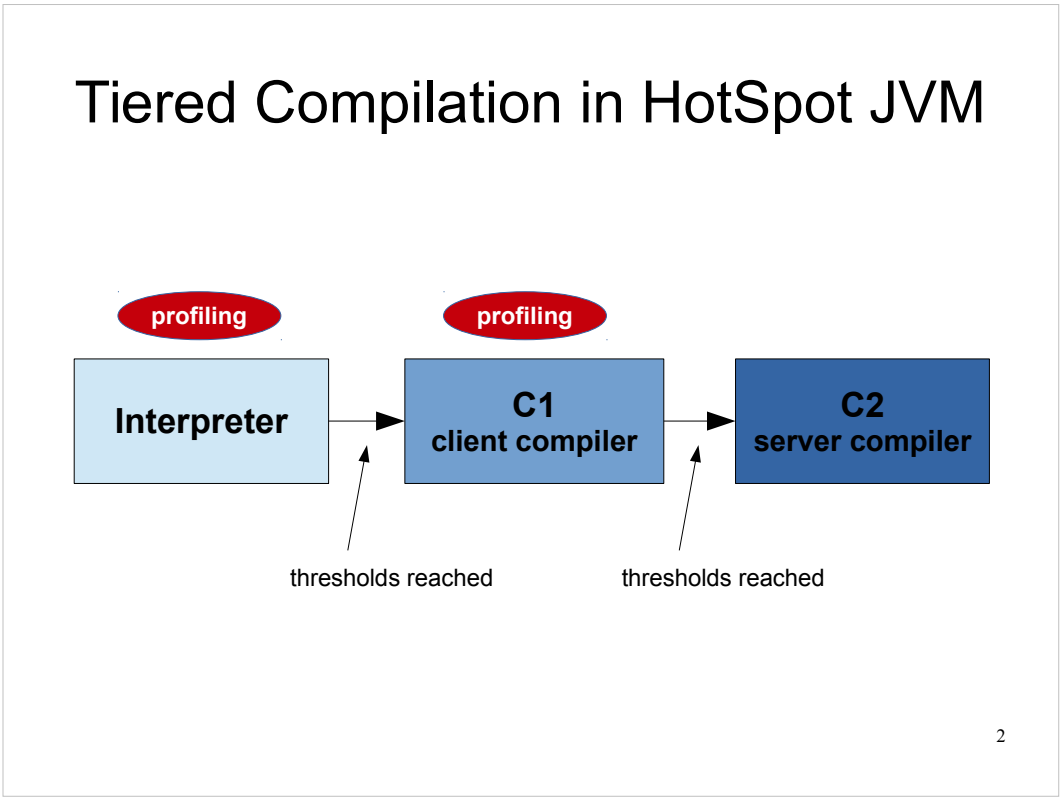
Profile Caching for the Java Virtual Machine

Marcel Mohler, ETH Zurich
Bachelor Thesis

Supervisors: Zoltan Majo, Oracle
Tobias Hartmann, Oracle

Prof. Thomas Gross, Laboratory for Software Technology, ETH

Welcome

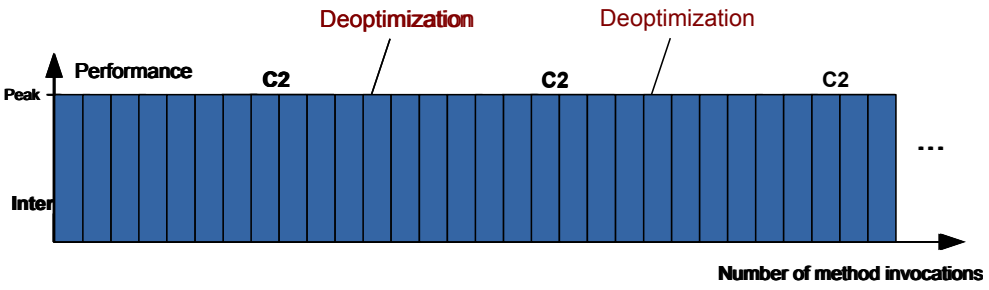


First off, a few things about the Just-in-time compiler of Hotspot

Hotspot is a JVM maintained by Oracle and the most used JVM world wide
Very complex system, 250 thousand LoC

Profiles crucial to the performance of the code

Problem: performance fluctuations



Goal: Decrease performance fluctuations

- What is the current problem?
- Methods start compiling at Interpreter and have to go through multiple tiers until reaching peak performance
- In case of deoptimizations we start from the beginning again
- Ideally we want this

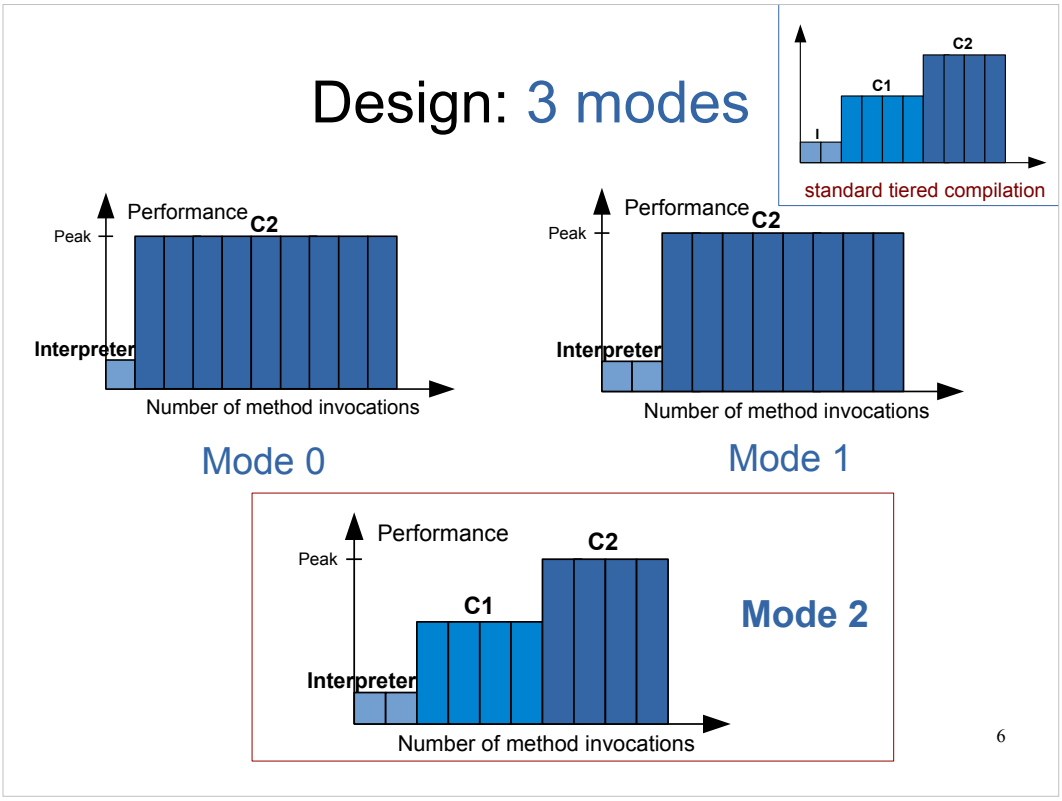
Observation

- Profiles need to be gathered each time the JVM starts
- Most frequently used methods often do not change

→ Idea: cache and reuse the profiles!

Outline

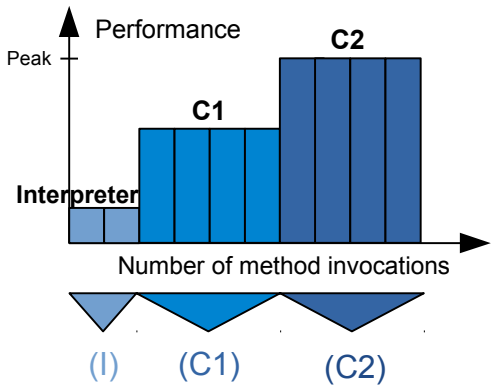
1. Design
2. Implementation
3. Performance evaluation
4. Conclusion



- Thesis includes 3 modes

- First one immediately uses cached profiles in C2 with lowered thresholds
- Interpreter step needed because of class loading
- We realized this could result in an increased load on the compile queue
- Mode 1: similar to 0 but does not lower thresholds
-
- Finally a Mode 2 which puts as much load on the compile queue as baseline

Design: Mode 2



(I) unmodified

(C1) remove profiling code
→ ~30% speedup

(C2) use cached profile
→ better code quality

Implementation

- Based on existing compilation replay functionality
- Configurable by JVM flags
- Select all or an arbitrary set of methods

Evaluation

- ETH Data Center Observatory
- Focus on warmup
- 2 benchmark suites
  - SPECjvm 2008

    17/21 individual benchmarks used
  - Google Octane (using Nashorn)

    16/17 individual benchmarks used

DCO provided by ETH

Some benchmarks skipped because of incompatibility with JDK9

Had to focus on warmup since it doesn't really affect performance for a long running benchmark

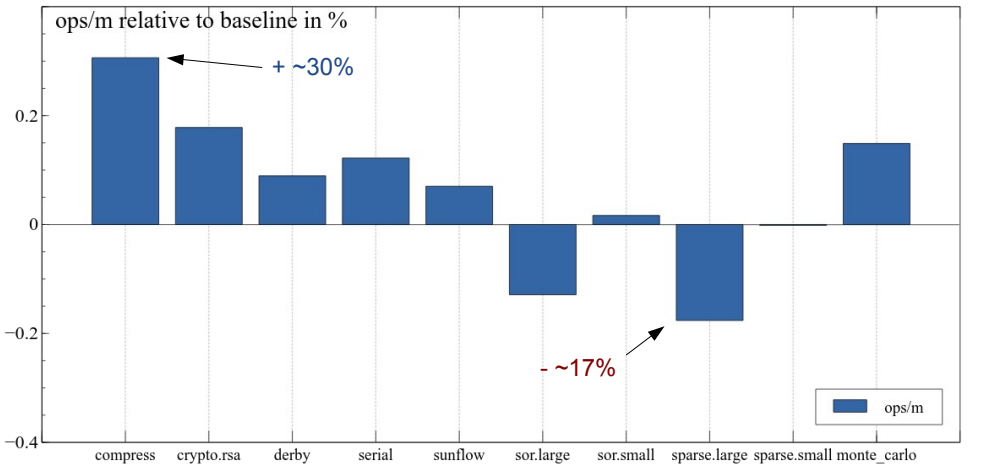
Divided benchmarks up in their parts and restarted JVM in between benchmarks

Also several microbenchmarks which are presented in the thesis

Going to present numbers from Mode2, there is a lot more in the thesis

Performance evaluation

- Performance (higher is better)



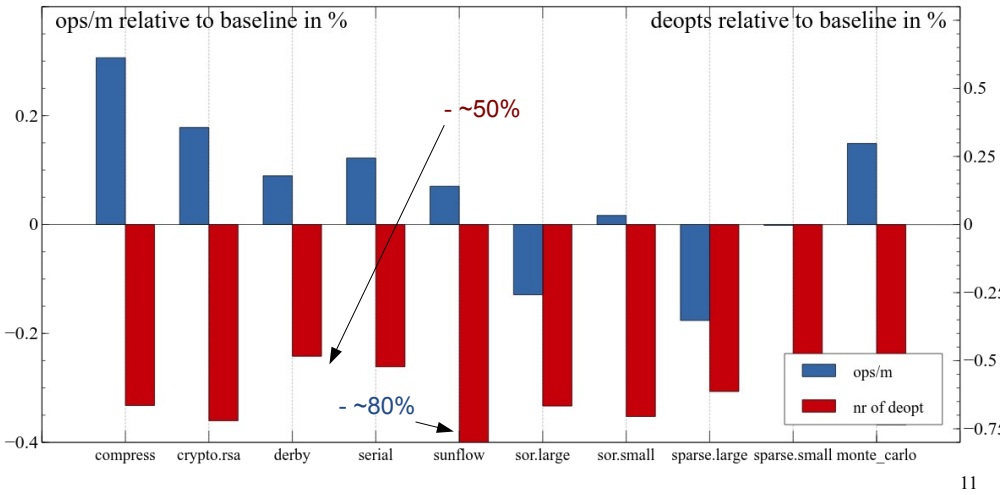
10

Performance example of SpecJVM benchmark

1 run, around 5-30 seconds

Performance evaluation

- Deoptimizations (lower is better)



Deoptimizations lowered significantly, good indicator of improved code quality

Still difficult to measure how much influence a deopt has

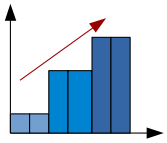
Other benchmark results

- Benefit mainly from cached C2 compilations. Disabling interpreter profiles rarely affects performance
- Around 70% of the compilations use cached profiles
- No association between performance and load on compile queue

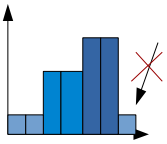
Rest 30% are mainly L1/L2 compilations and a few skipped ones (lambda expressions etc)

Conclusion

- Cached profiles can improve warmup performance
- System allows fine tuning
- Main benefits:



faster warmup



fewer deoptimizations

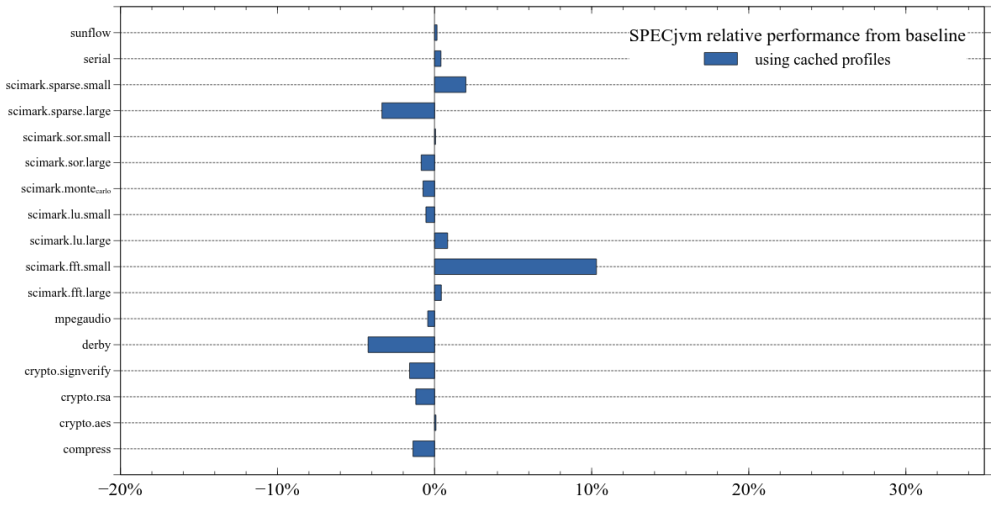
- Room for future work

Future work: modify / merging profiles
Improve data structure used

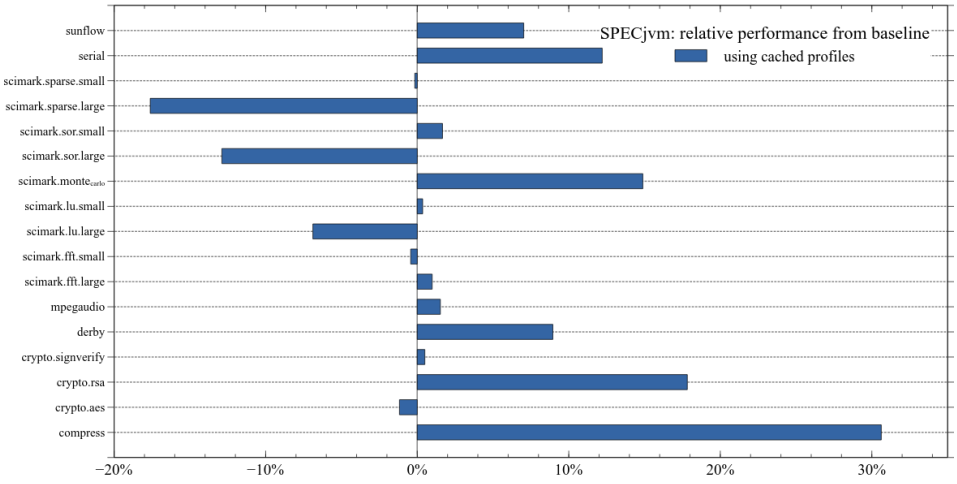
End

Additional graphes follow

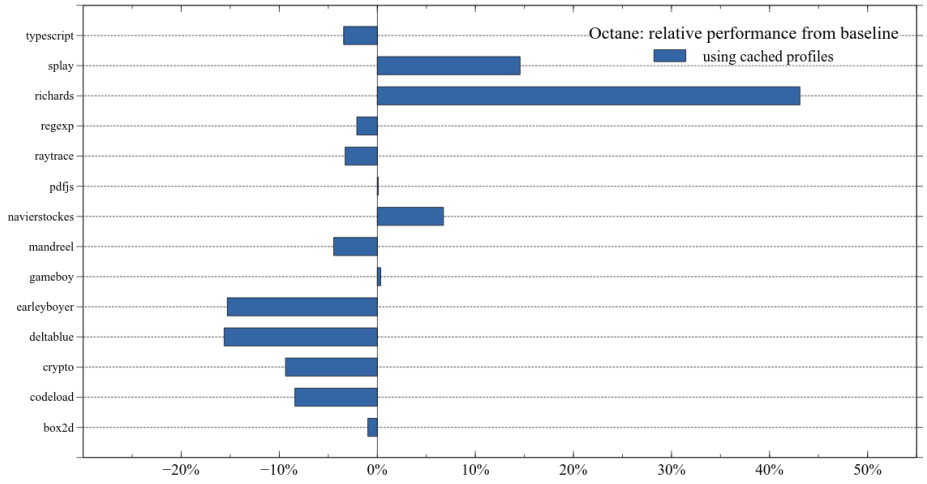
SPECjvm: relative performance full runs (2+4 minutes)



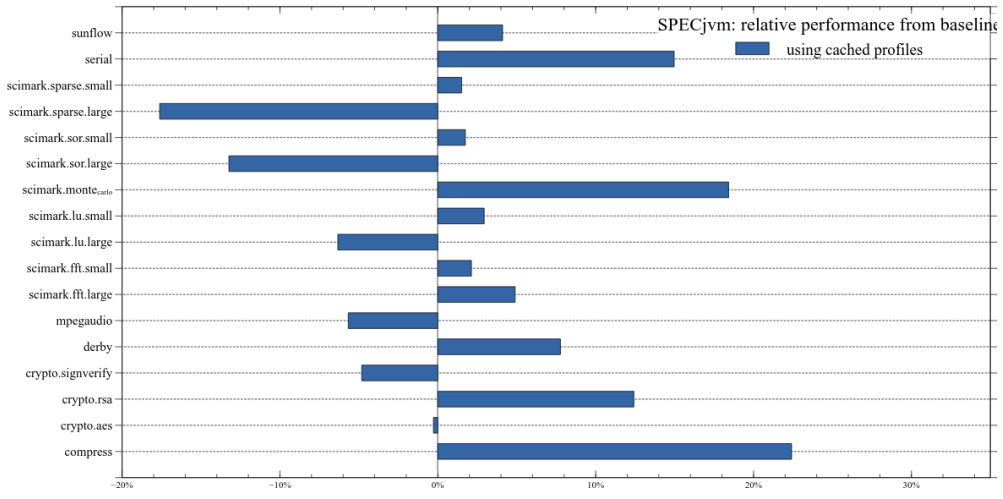
SPECjvm: relative performance warmup: all benchmarks



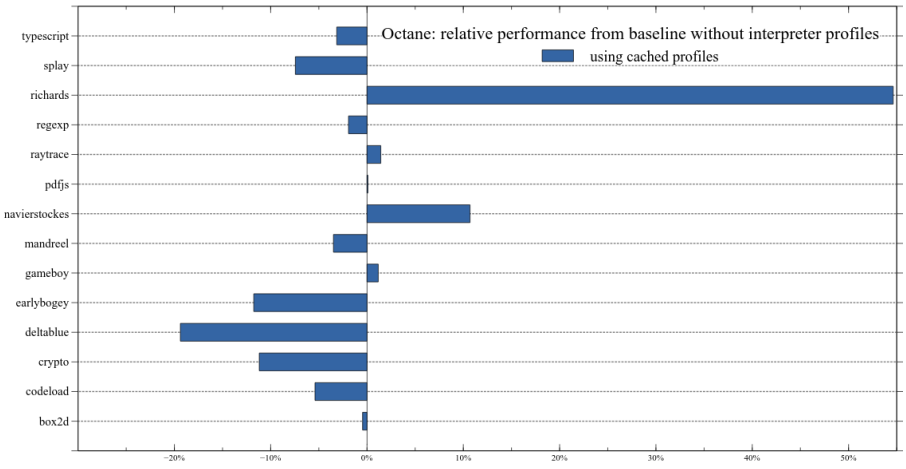
Octane: relative performance all benchmarks



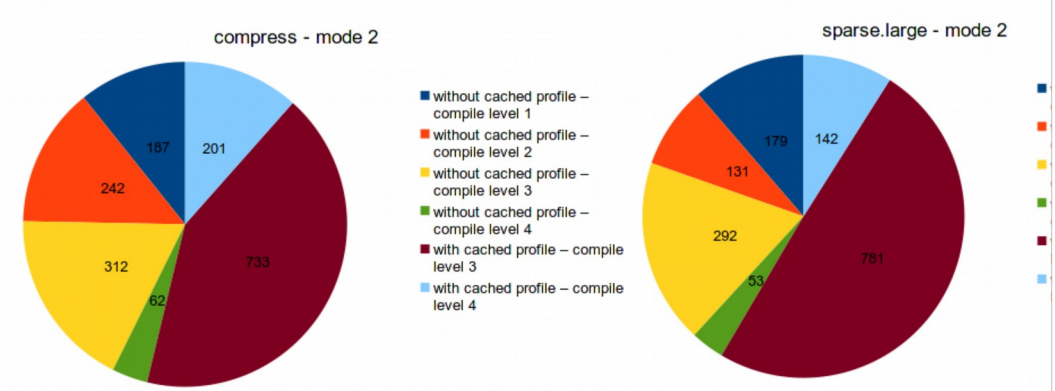
SPECjvm: relative performance without interpreter profiles



Octane: relative performance without interpreter profiles

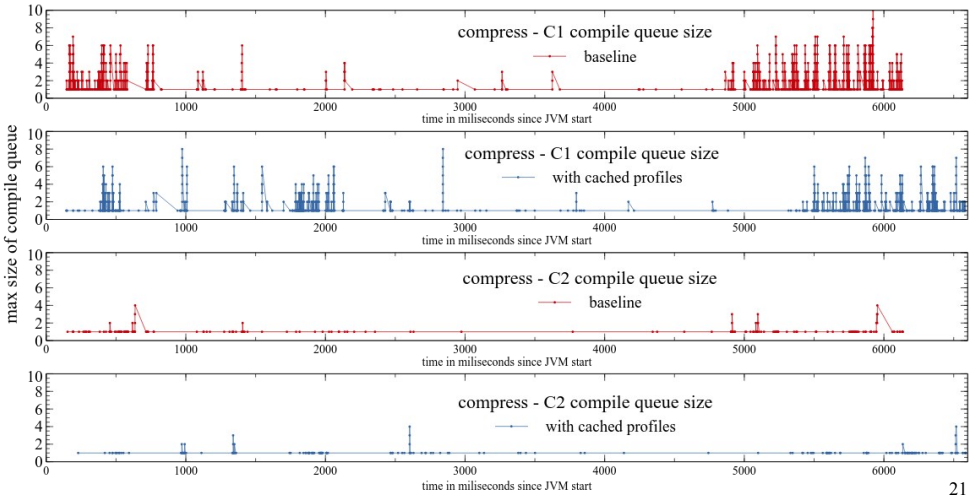


Type of compilations



Compile queue

- compress



Compile queue

- scimark.sparse.large