4 Performance

This section evaluates the performance of the cached profile implementation using modern benchmark suites.

4.1 Setup

To provide reliable and comparable results all tests were done on a single node of the Data Center Observatory provided by ETH [3]. A node features 2 8-Core AMD Opteron 6212 CPUs running at 2600 MHz with 128 GB of DDR3 RAM. The node is running Fedora 19 and GCC 4.8.3. All JDK builds got created on the node itself.

To compare performance the following benchmarks were used:

1. **SPECjvm 2008:** A benchmark suite developed by Standard Performance Evaluation Corporation for measuring the performance of the Java Runtime Environment [12]. I use version 2008 and I run a subset of 17 out of a total of 21 benchmarks. 4 are omitted due to incompatibility with openJDK 1.9.0.
   Once finished, SPECjvm prints out the number of operations per minute. This is used to compare the performance and higher is better.

2. **Octane 2.0:** A benchmark developed by Google to measure the performance of JavaScript code found in large, real-world applications [4]. Octane runs on Nashorn, a JavaScript Engine on top of Hotspot. The version used is 2.0 and consists of 17 individual benchmarks of which 16 are used.
   Octane gives each benchmark a score reflecting the performance, the higher the score, the better the performance.

The benchmarking process was automated using a number of self-written python scripts. The graphs in this chapter always show the arithmetic mean of 50 runs and the error bars display the 95% confidence intervals.

4.2 Startup performance

The main goal of cached profiles is to improve the startup performance of the JVM. Having a rich profile from an earlier execution will allow the JIT compiler to use a highly optimized version right

from the beginning. I will start by looking at SPECjvm since it offers ways to focus on the warmup. An individual description of each benchmark being used can be found in Appendix A.2.

The longer a program is running the less impact a faster warmup has. Considering most benchmarks include a warmup phase which does not count towards the final score simply running the complete benchmark suite is not an option. Instead I limited SPECjvm to 1 single operation which, depending on the benchmark take around 10 to 20 seconds. Additionally, the JVM gets restarted between each single benchmark to prevent methods shared between benchmarks being compiled already.

I run each benchmark with all cached profiling features disabled. This run is called the *baseline* and displays the current openJDK 1.9.0 performance.

I then use a single benchmark run where I dump the profiles to disk. This run is not limited to a single operation and instead uses the default values of the benchmark. By default the benchmark is limited by time and runs for about 6 minutes. The idea is that these profiles include information that are usually not available during warmup and result in less deoptimizations and better code quality.

These profiles are then used in 3 individual runs using the introduced `-XX:CacheProfiles` flag. Each run is using one of the 3 different CacheProfilesModes. Figures 4.1 and 4.2 shows the number

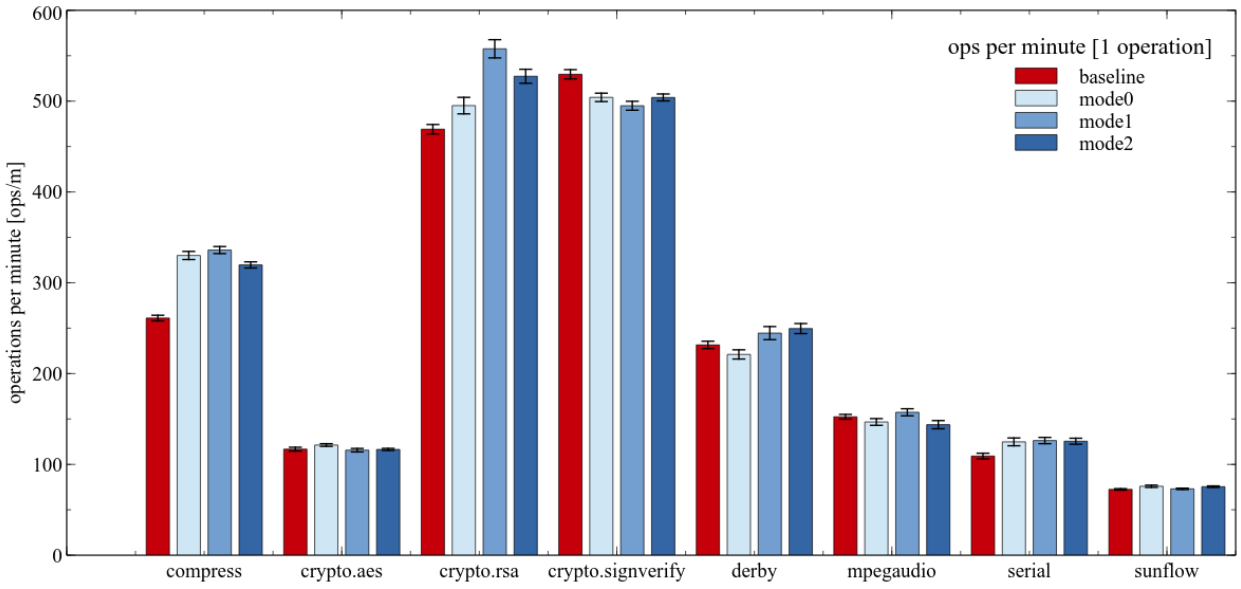

Figure 4.1: SPECjvm benchmarks on all different modes

of operations per minute, measured for each benchmark individually. The

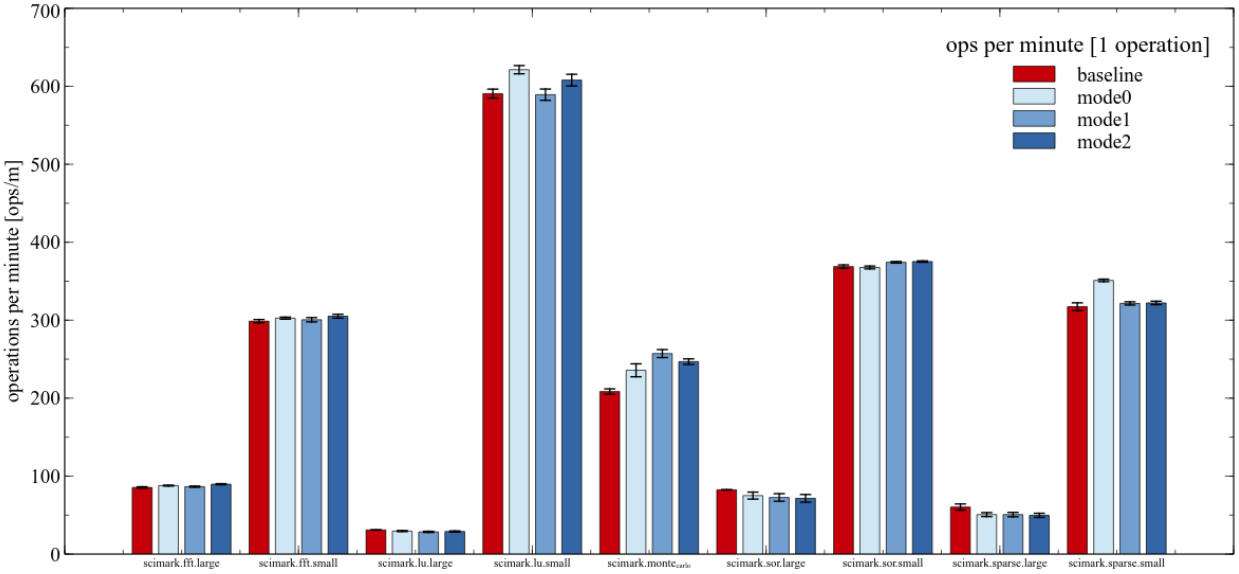

Figure 4.2: SPECjvm scimark benchmarks on all different modes

4.3 Without Intrinsics

4.4 Deoptimizations

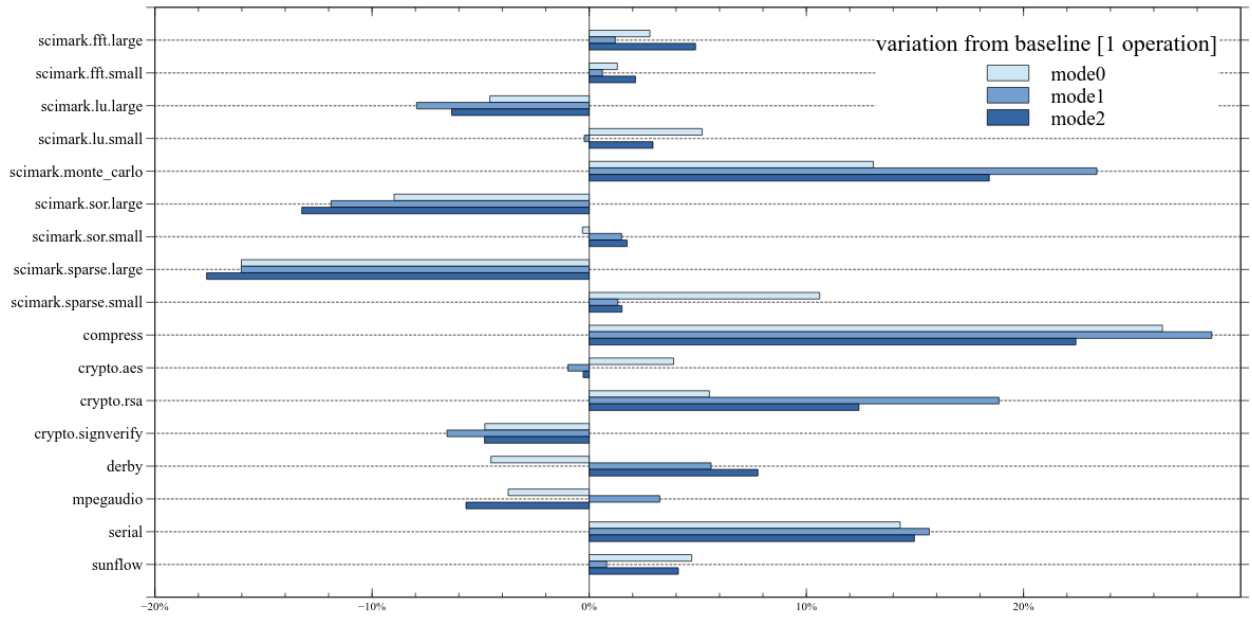

Figure 4.3: Relative performance from baseline for all SPECjvm benchmarks

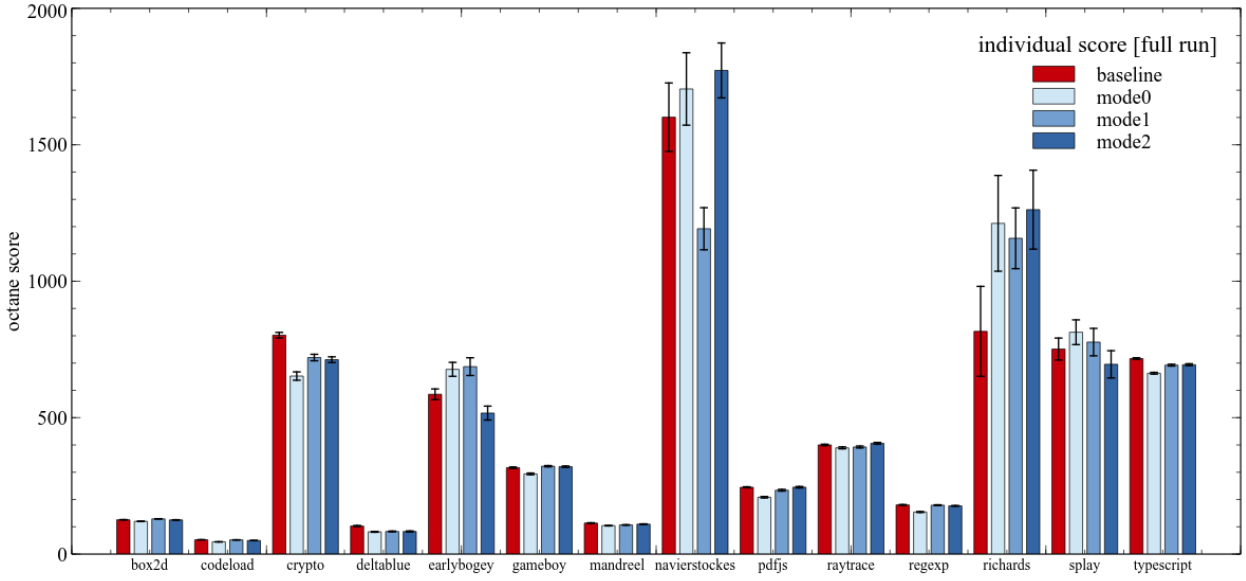

Figure 4.4: Octane benchmarks on all different modes

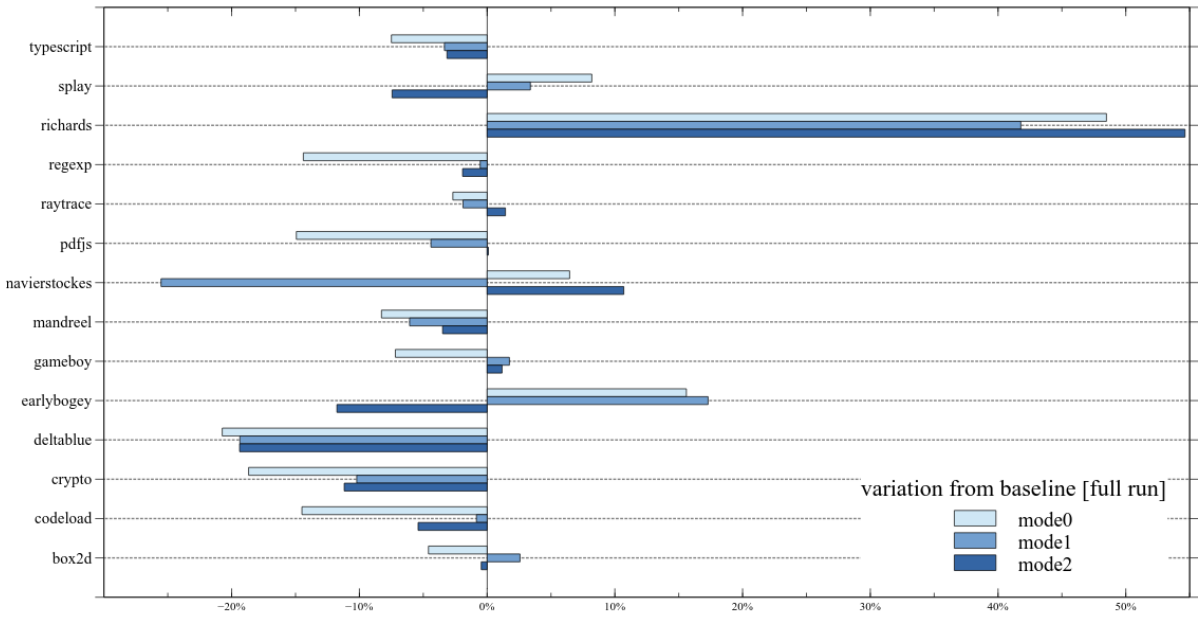

Figure 4.5: Relative performance from baseline for all Octane benchmarks

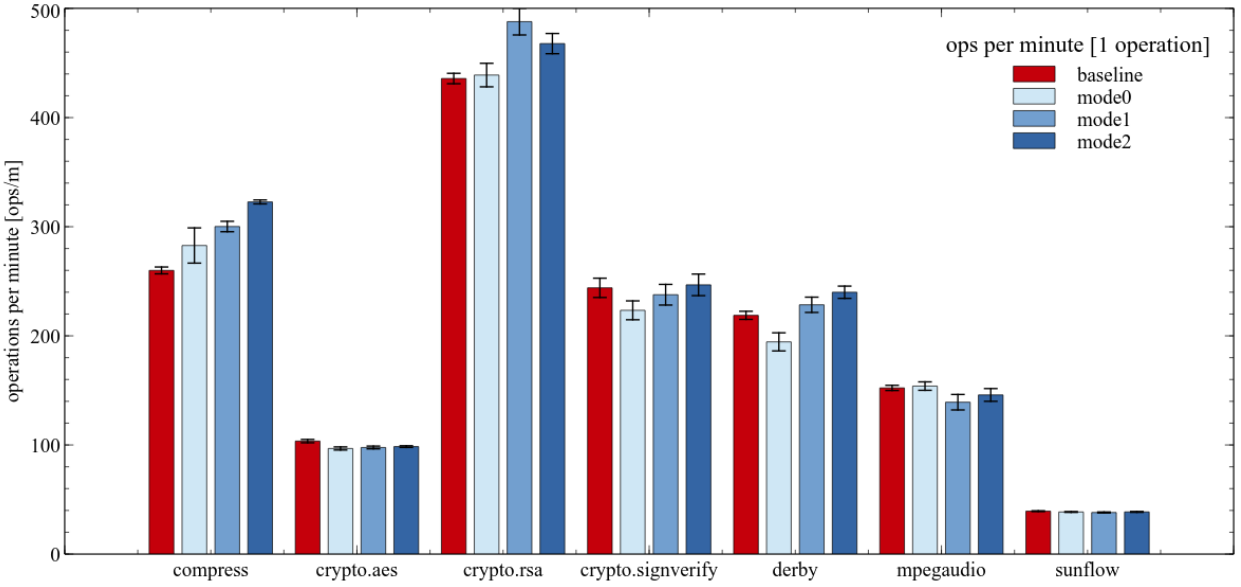

Figure 4.6: SPECjvm benchmarks on all different modes without intrinsified methods

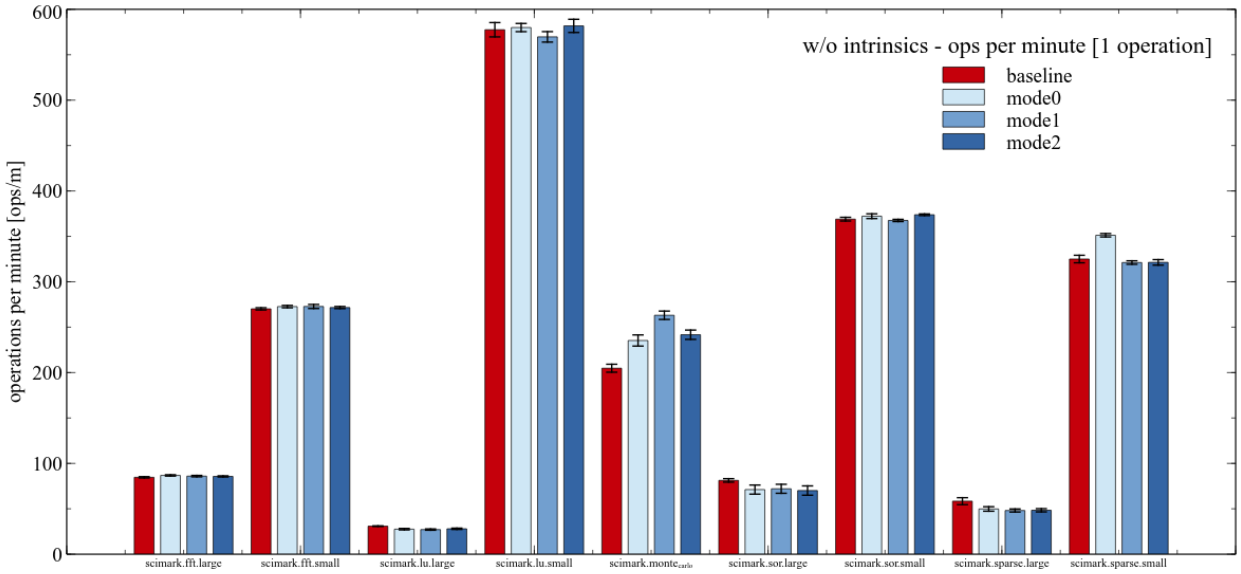

Figure 4.7: SPECjvm scimark benchmarks on all different modes without intrinsified methods

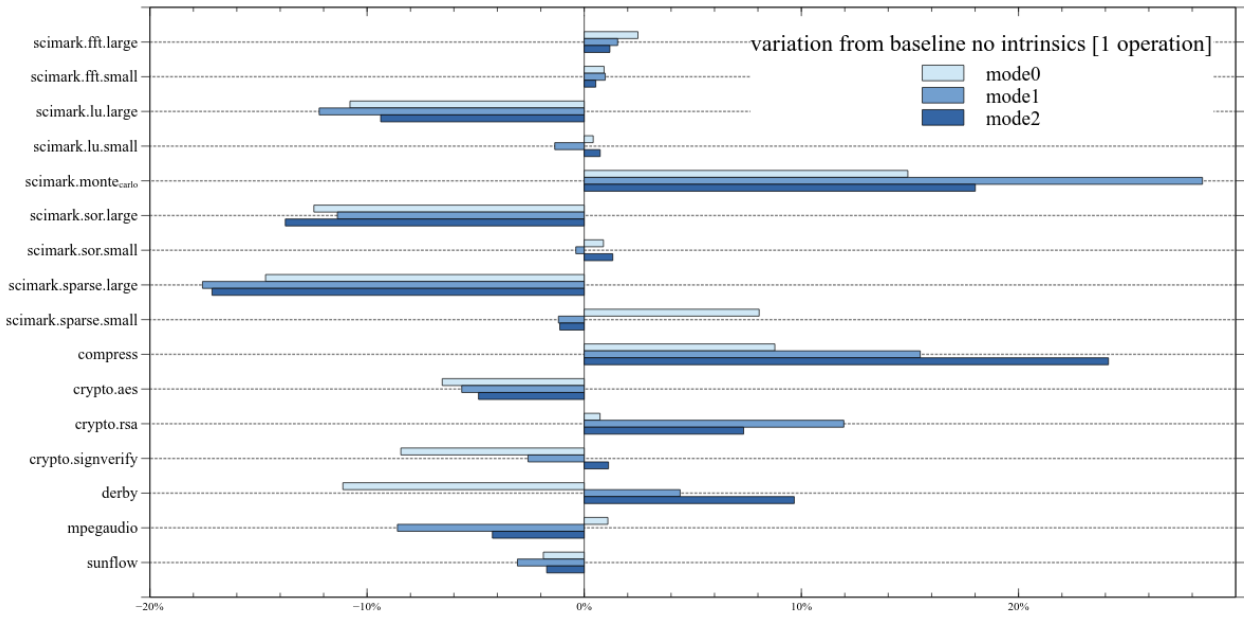

Figure 4.8: Relative performance from baseline for all SPECjvm benchmarks without intrinsified methods

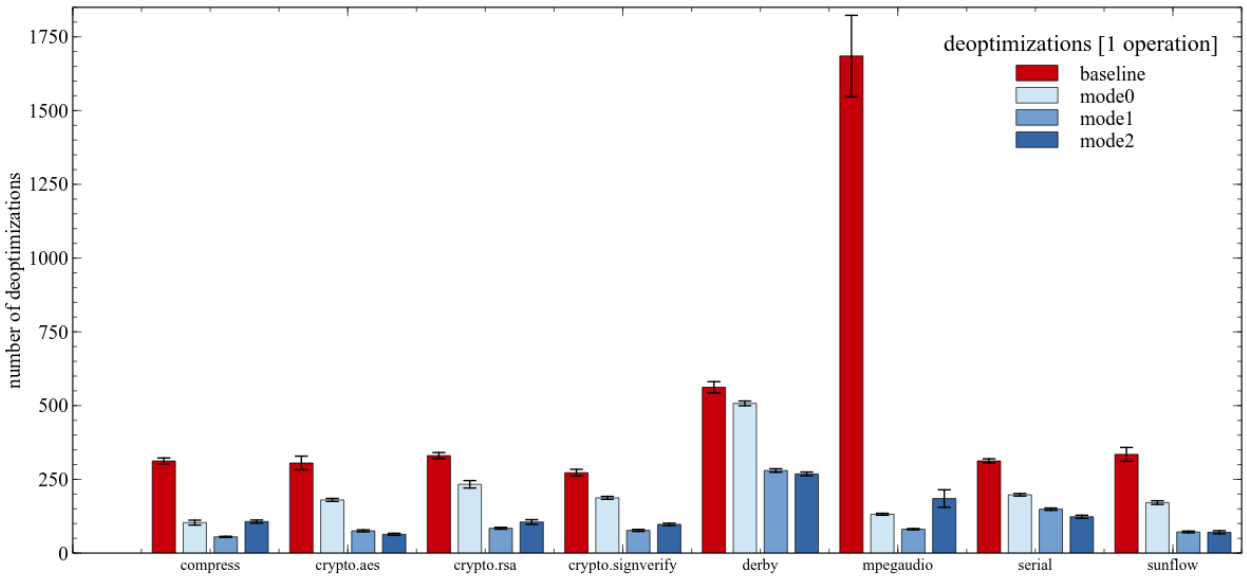

Figure 4.9: SPECjvm deoptimizations of all modes

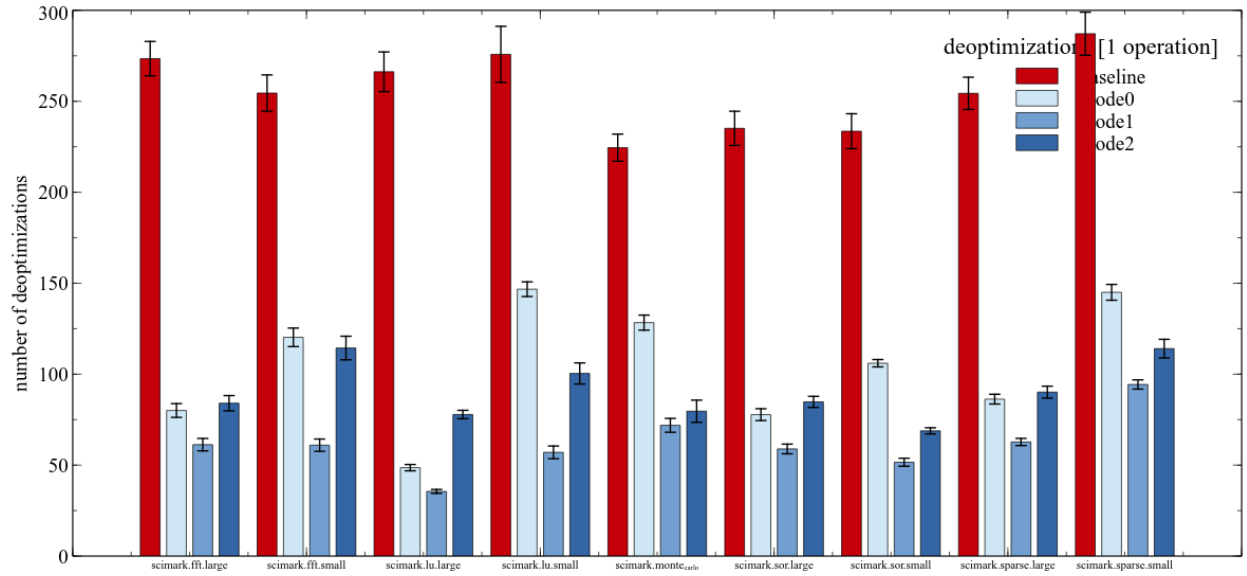

Figure 4.10: SPECjvm scimark deoptimizations of all modes

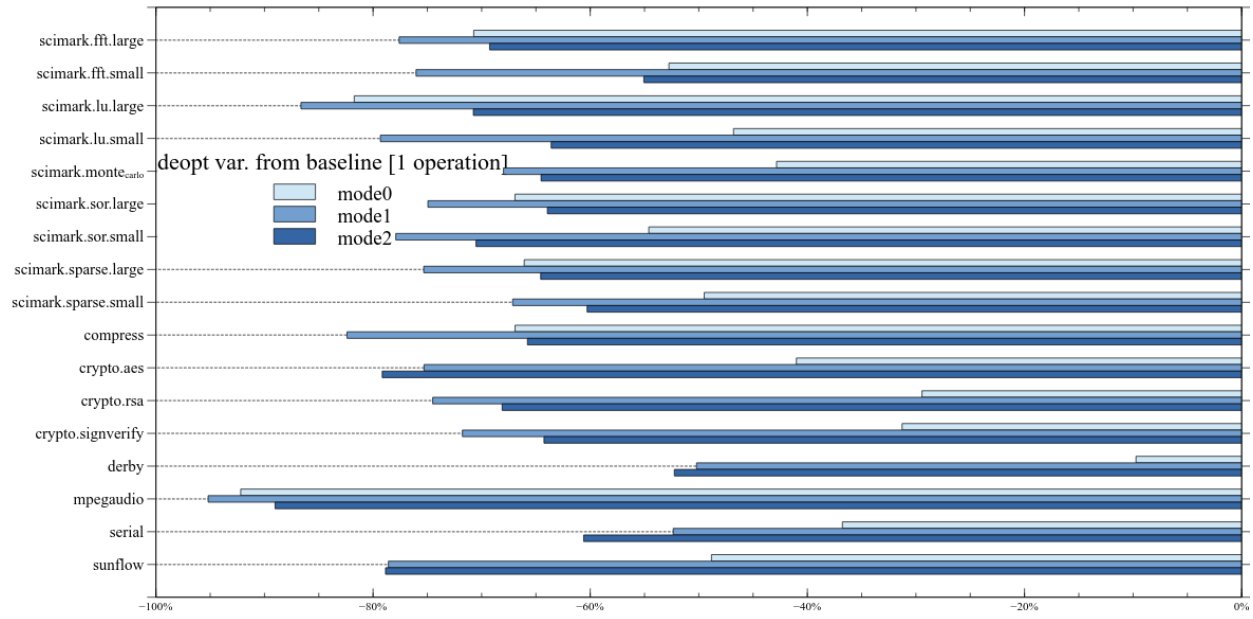

Figure 4.11: Relative deoptimizations from baseline for all SPECjvm benchmarks

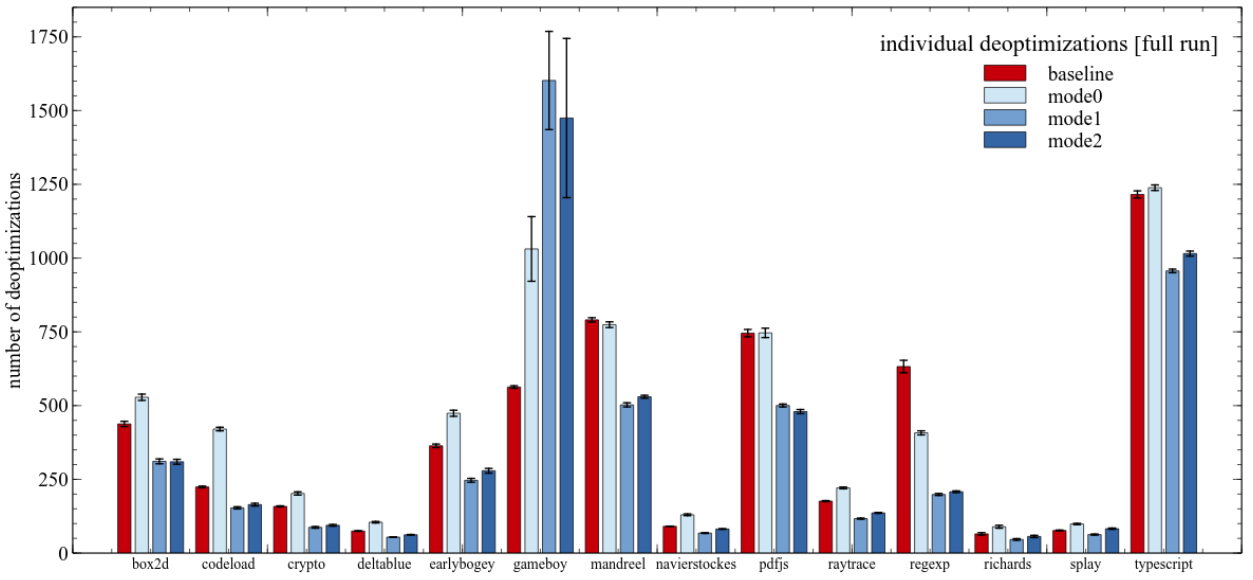

Figure 4.12: Octane deoptimizations of all modes

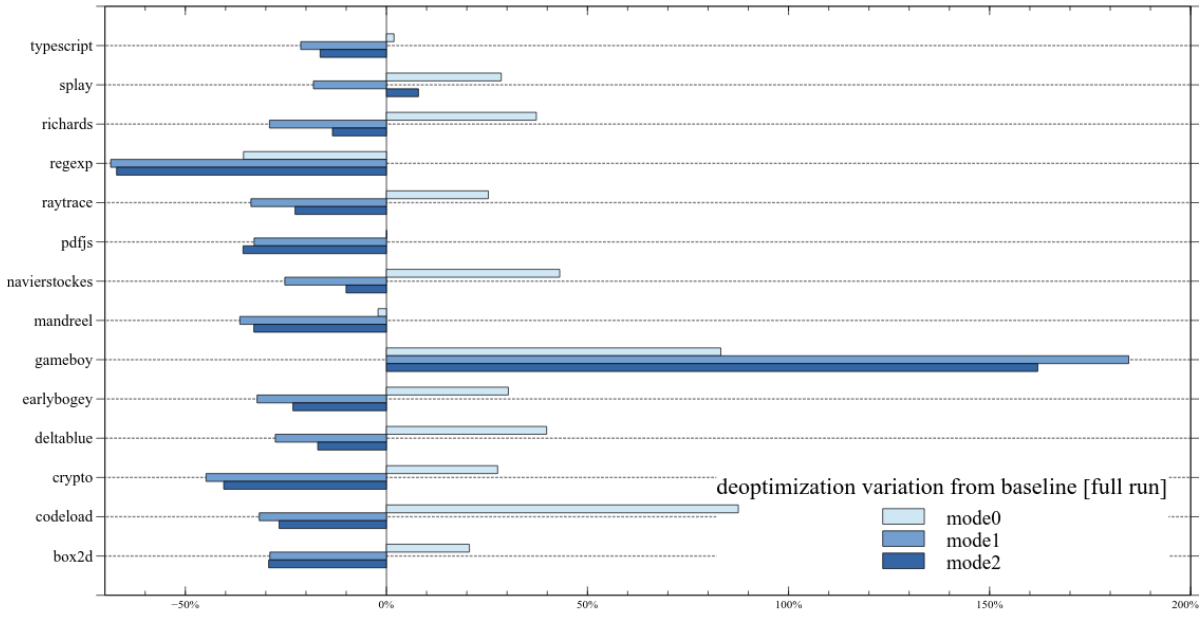

Figure 4.13: Relative deoptimizations from baseline for all Octane benchmarks

4.5 Effect on compile queue

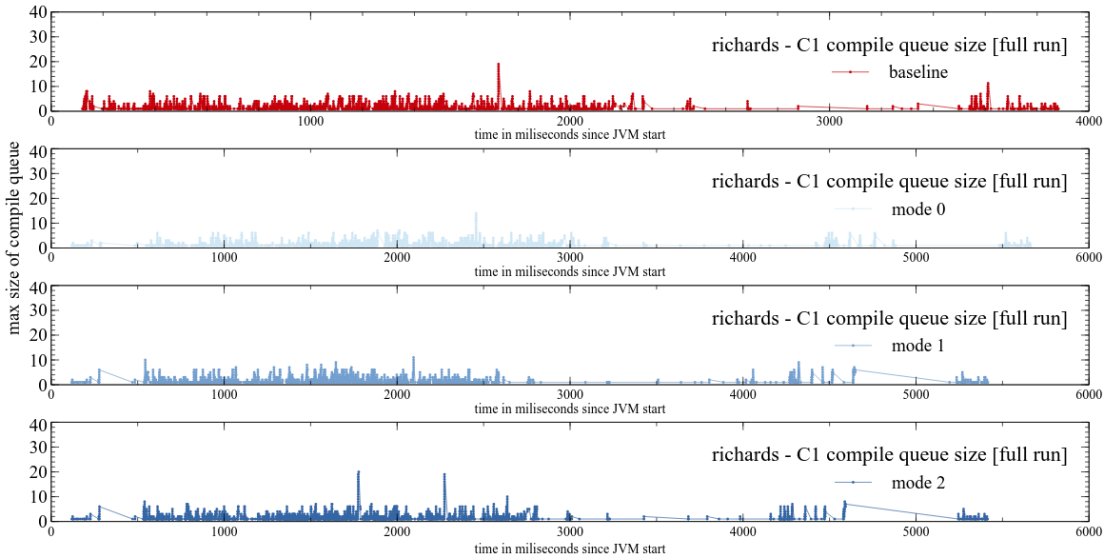

Figure 4.14: C1 Compile queue size over time Octane Richards benchmark

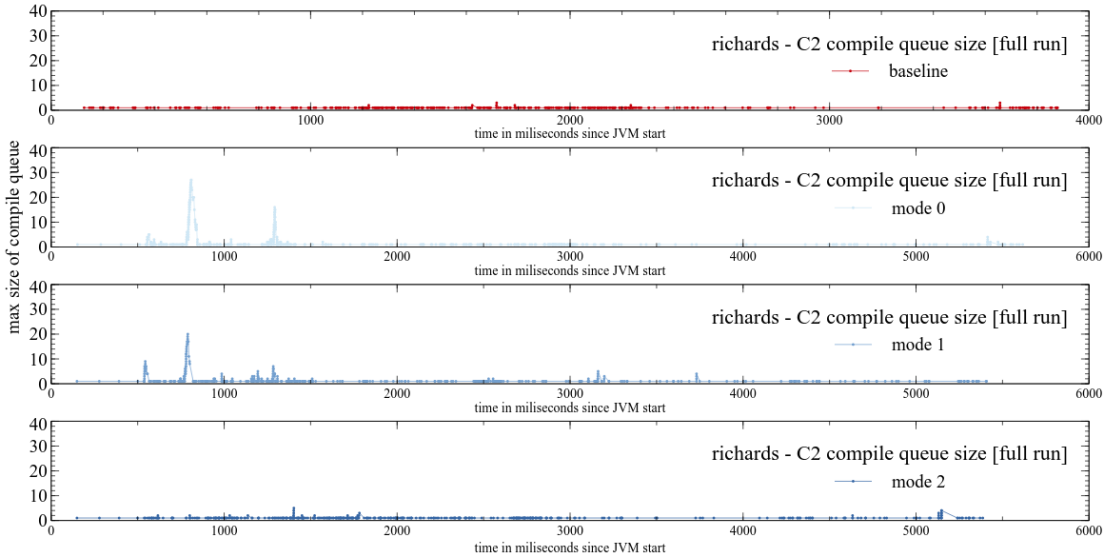

Figure 4.15: C2 Compile queue size over time Octane Richards benchmark

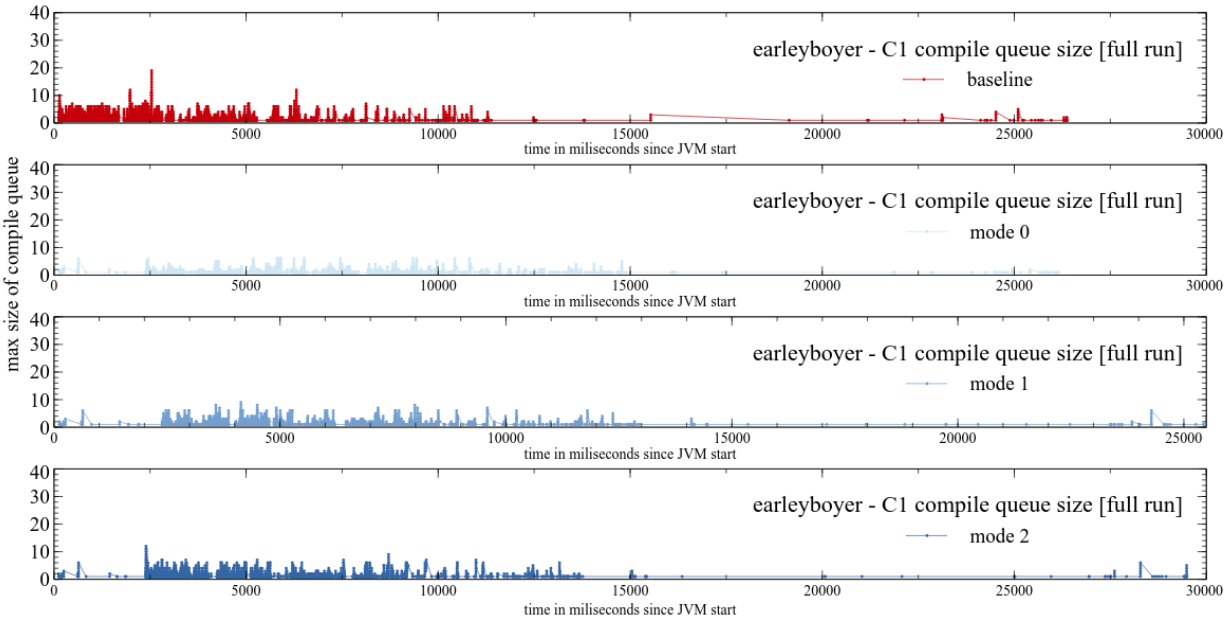

Figure 4.16: C1 Compile queue size over time Octane EarleyBoyer benchmark

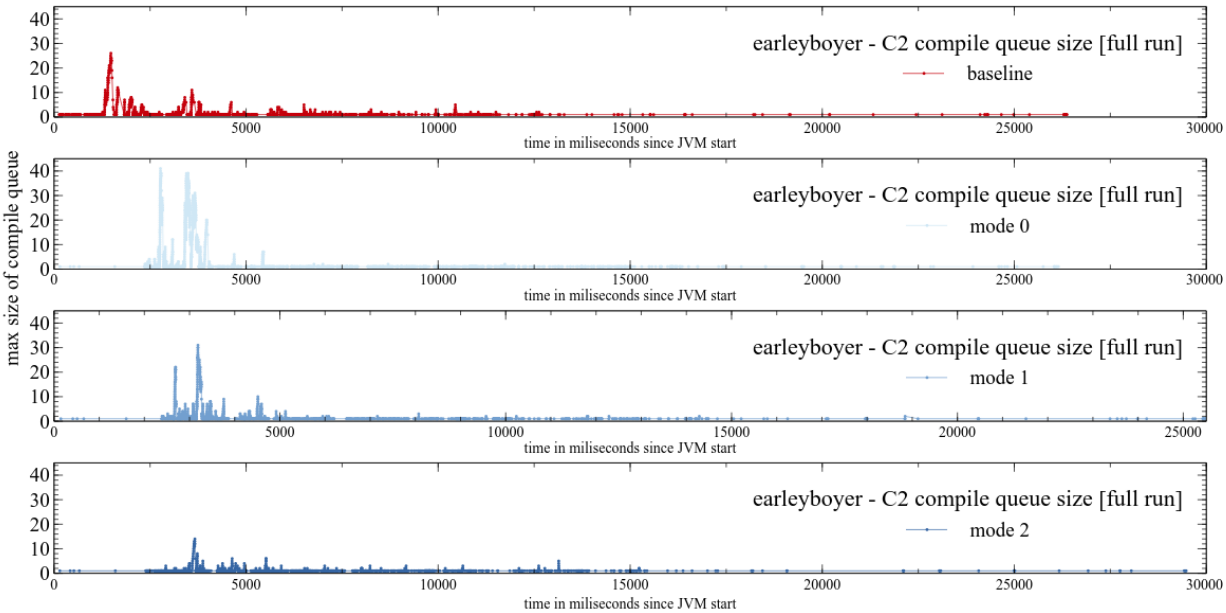

Figure 4.17: C2 Compile queue size over time Octane EarleyBoyer benchmark

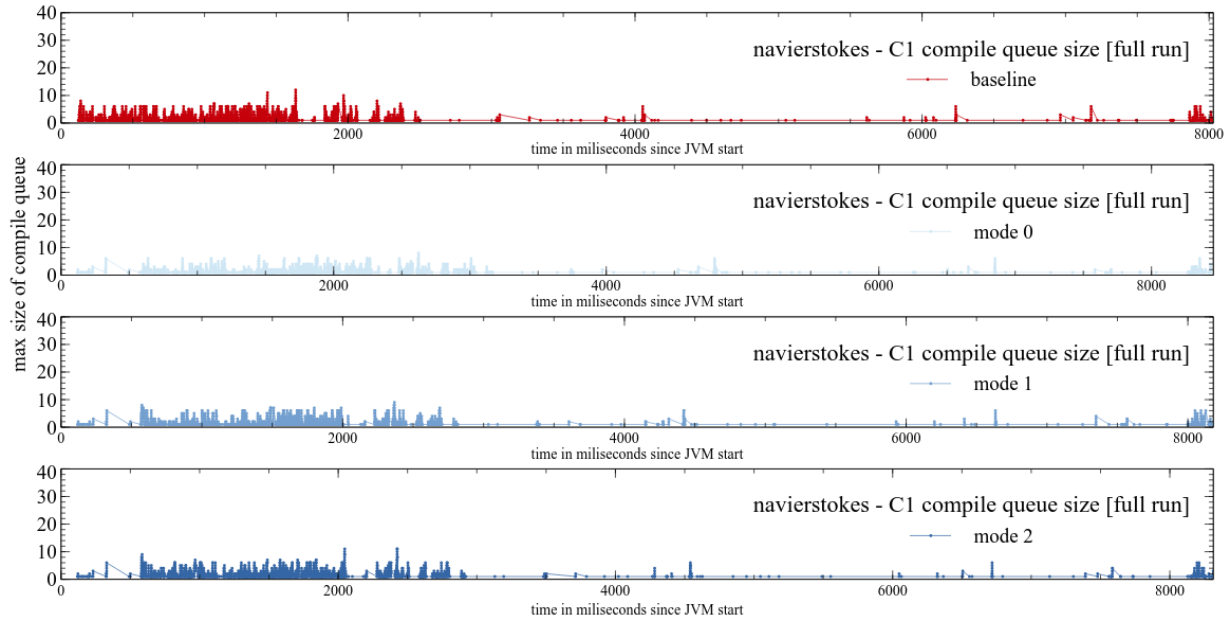

Figure 4.18: C1 Compile queue size over time Octane NavierStokes benchmark

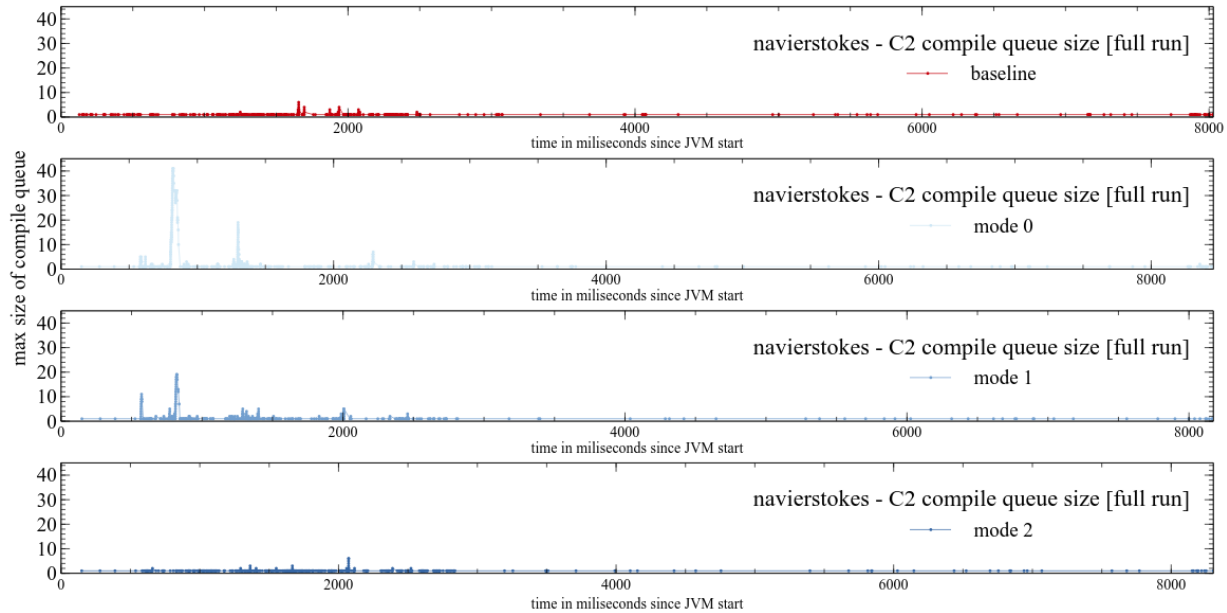

Figure 4.19: C2 Compile queue size over time Octane NavierStokes benchmark

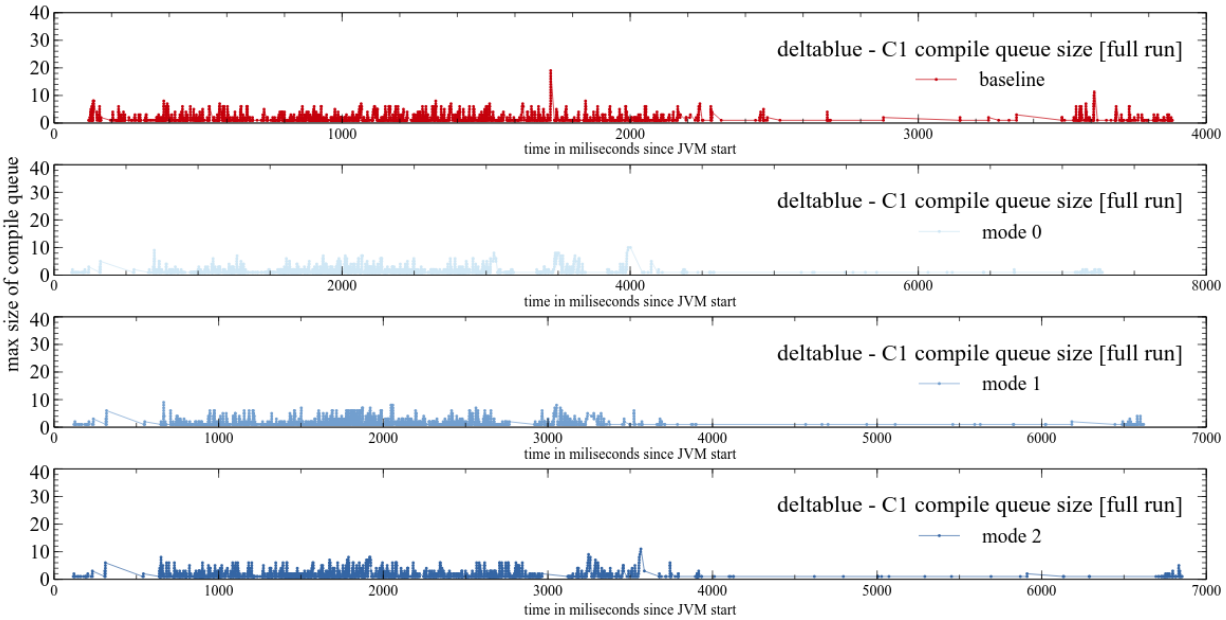

Figure 4.20: C1 Compile queue size over time Octane DeltaBlue benchmark

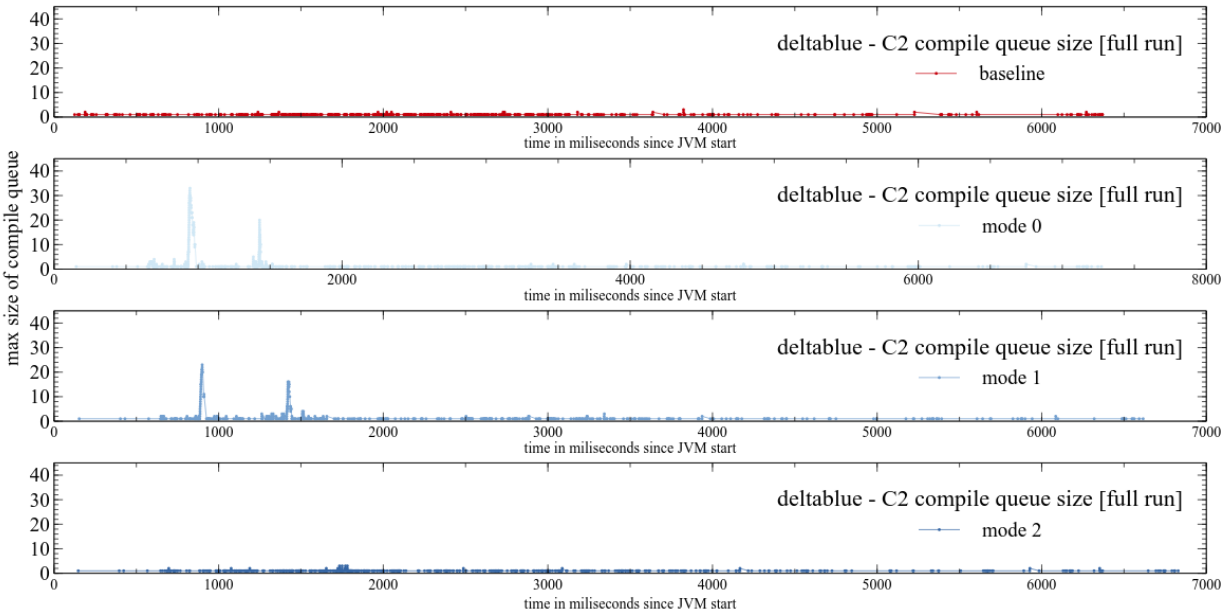

Figure 4.21: C2 Compile queue size over time Octane DeltaBlue benchmark

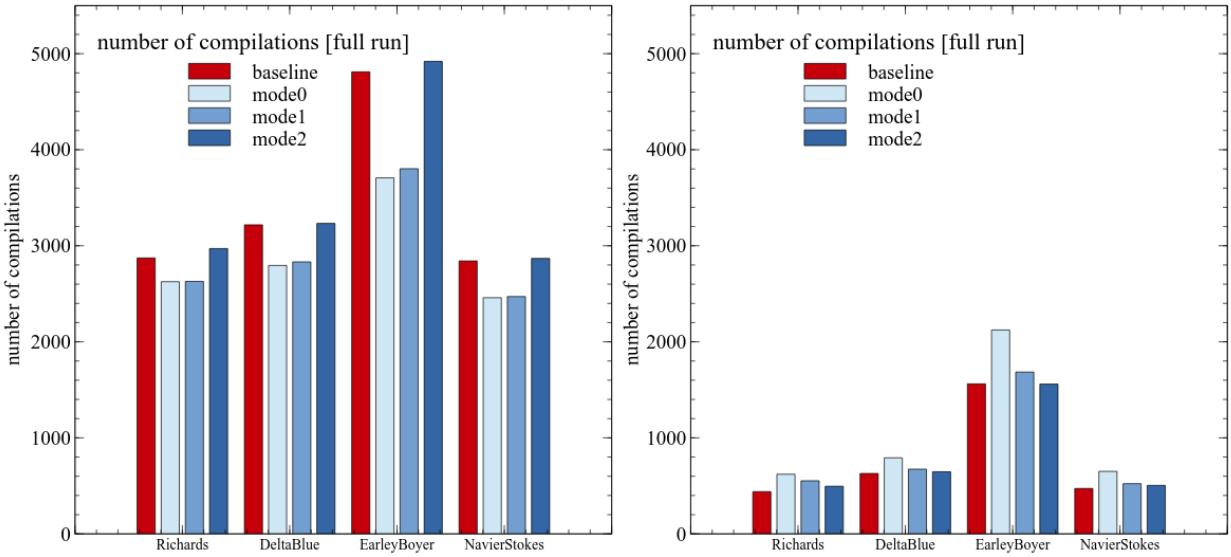

Figure 4.22: Number of compilations

5 Possible Improvements

- Better datasctructure - Merging multiple profiles cleverly - editing profiles - improve read of profile file

6 Conclusion

A Appendix

A.1 Tiered Compilation Thresholds

flag	description	default
CompileThresholdScaling	number of interpreted method invocations before (re-)compiling	1.0
Tier0InvokeNotifyFreqLog	Interpreter (tier 0) invocation notification frequency	7
Tier2InvokeNotifyFreqLog	C1 without MDO (tier 2) invocation notification frequency	11
Tier3InvokeNotifyFreqLog	C1 with MDO profiling (tier 3) invocation notification frequency	10
Tier23InlineeNotifyFreqLog	Inlinee invocation (tiers 2 and 3) notification frequency	20
Tier0BackedgeNotifyFreqLog	Interpreter (tier 0) invocation notification frequency	10
Tier2BackedgeNotifyFreqLog	C1 without MDO (tier 2) invocation notification frequency	14
Tier3BackedgeNotifyFreqLog	C1 with MDO profiling (tier 3) invocation notification frequency	13
Tier2CompileThreshold	threshold at which tier 2 compilation is invoked	0
Tier2BackEdgeThreshold	Back edge threshold at which tier 2 compilation is invoked	0
Tier3InvocationThreshold	Compile if number of method invocations crosses this threshold	200
Tier3MinInvocationThreshold	Minimum invocation to compile at tier 3	100
Tier3CompileThreshold	Threshold at which tier 3 compilation is invoked (invocation minimum must be satisfied)	2000
Tier3BackEdgeThreshold	Back edge threshold at which tier 3 OSR compilation is invoked	60000
Tier4InvocationThreshold	Compile if number of method invocations crosses this threshold	5000
Tier4MinInvocationThreshold	Minimum invocation to compile at tier 4	600
Tier4CompileThreshold	Threshold at which tier 4 compilation is invoked (invocation minimum must be satisfied)	15000
Tier4BackEdgeThreshold	Back edge threshold at which tier 4 OSR compilation is invoked	40000

A.2 SPECjvm Benchmark

This list gives a short description of the benchmarks that are part of the SPECjvm 2008 Benchmark Suite. The list is directly taken from `https://www.spec.org/jvm2008/docs/benchmarks/index.html` and put in as a reference.

- `Compress:` This benchmark compresses data, using a modified Lempel-Ziv method (LZW). Basically finds common substrings and replaces them with a variable size code. This is deterministic, and can be done on the fly. Thus, the decompression procedure needs no input table, but tracks the way the table was built. Algorithm from "A Technique for High

Performance Data Compression", Terry A. Welch, IEEE Computer Vol 17, No 6 (June 1984), pp 8-19.

This is a Java port of the 129.compress benchmark from CPU95, but improves upon that benchmark in that it compresses real data from files instead of synthetically generated data as in 129.compress.

- `Crypto:` This benchmark focuses on different areas of crypto and are split in three different sub-benchmarks. The different benchmarks use the implementation inside the product and will therefore focus on both the vendor implementation of the protocol as well as how it is executed.
  - `aes` encrypt and decrypt using the AES and DES protocols, using CBC/PKCS5Padding and CBC/NoPadding. Input data size is 100 bytes and 713 kB.
  - `rsa` encrypt and decrypt using the RSA protocol, using input data of size 100 bytes and 16 kB.
  - `signverify` sign and verify using MD5withRSA, SHA1withRSA, SHA1withDSA and SHA256withRSA protocols. Input data size of 1 kB, 65 kB and 1 MB.
- `Derby:` This benchmark uses an open-source database written in pure Java. It is synthesized with business logic to stress the BigDecimal library. It is a direct replacement to the SPECjvm98 db benchmark but is more capable and represents as close to a "real" application. The focus of this benchmark is on BigDecimal computations (based on telco benchmark) and database logic, especially, on locks behavior. BigDecimal computations are trying to be outside 64-bit to examine not only 'simple' BigDecimal, where 'long' is used often for internal representation.
- `MPEGaudio:` This benchmark is very similar to the SPECjvm98 mpegaudio. The mp3 library has been replaced with JLayer, an LGPL mp3 library. Its floating-point heavy and a good test of mp3 decoding. Input data were taken from SPECjvm98.
- `Scimark:` This benchmark was developed by NIST and is widely used by the industry as a floating point benchmark. Each of the subtests (`fft, lu, monte_carlo, sor, sparse`) were incorporated into SPECjvm2008. There are two versions of this test, one with a l̈arge d̈ataset (32Mbytes) which stresses the memory subsystem and a s̈mall d̈ataset which stresses the JVMs (512Kbytes).
- `Serial:` This benchmark serializes and deserializes primitives and objects, using data from the JBoss benchmark. The benchmark has a producer-consumer scenario where serialized objects are sent via sockets and deserialized by a consumer on the same system. The benchmark heavily stress the Object.equals() test.
- `Sunflow:` This benchmark tests graphics visualization using an open source, internally multi-threaded global illumination rendering system. The sunflow library is threaded internally, i.e. it's possible to run several bundles of dependent threads to render an image. The

number of internal sunflow threads is required to be 4 for a compliant run. It is however possible to configure in property specjvm.benchmark.sunflow.threads.per.instance, but no more than 16, per sunflow design. Per default, the benchmark harness will use half the number of benchmark threads, i.e. will run as many sunflow benchmark instances in parallel as half the number of hardware threads. This can be configured in specjvm.benchmark.threads.sunflow.

- `XML:` This benchmark has two sub-benchmarks: XML.transform and XML.validation. XML.transform exercises the JRE's implementation of javax.xml.transform (and associated APIs) by applying style sheets (.xsl files) to XML documents. The style sheets and XML documents are several real life examples that vary in size (3KB to 156KB) and in the style sheet features that are used most heavily. One "operation" of XML.transform consists of processing each style sheet / document pair, accessing the XML document as a DOM source, a SAX source, and a Stream source. In order that each style sheet / document pair contribute about equally to the time taken for a single operation, some of the input pairs are processed multiple times during one operation.

  Result verification for XML.transform is somewhat more complex than for other of the benchmarks because different XML style sheet processors can produce results that are slightly different from each other, but all still correct. In brief, the process used is this. First, before the measurement interval begins the workload is run once and the output is collected, canonicalized (per the specification of canonical XML form) and compared with the expected canonicalized output. Output from transforms that produce HTML is converted to XML before canonicalization. Also, a checksum is generated from this output. Inside the measurement interval the output from each operation is only checked using the checksum.

  XML.validation exercises the JRE's implementation of javax.xml.validation (and associated APIs) by validating XML instance documents against XML schemata (.xsd files). The schemata and XML documents are several real life examples that vary in size (1KB to 607KB) and in the XML schema features that are used most heavily. One "operation" of XML.validation consists of processing each style sheet / document pair, accessing the XML document as a DOM source and a SAX source. As in XML.transform, some of the input pairs are processed multiple times during one operation so that each input pair contributes about equally to the time taken for a single operation.

A.3 Octane Benchmark

What follows is an overview of the benchmarks Octane consists of. The original list can be found on `https://developers.google.com/octane/benchmark`.

- `Richards:` OS kernel simulation benchmark, originally written in BCPL by Martin Richards (539 lines).
  - Main focus: *property load/store, function/method calls*
  - Secondary focus: *code optimization, elimination of redundant code*

- `Deltablue:` One-way constraint solver, originally written in Smalltalk by John Maloney and Mario Wolczko (880 lines).
  - Main focus: *polymorphism*
  - Secondary focus: *OO-style programming*
- `Raytrace:` Ray tracer benchmark based on code by Adam Burmister (904 lines).
  - Main focus: *argument object, apply*
  - Secondary focus: *prototype library object, creation pattern*
- `Regexp:` Regular expression benchmark generated by extracting regular expression operations from 50 of the most popular web pages (1761 lines).
  - Main focus: *Regular expressions*
- `NavierStokes:` 2D NavierStokes equations solver, heavily manipulates double precision arrays. Based on Oliver Hunt's code (387 lines).
  - Main focus: *reading and writing numeric arrays.*
  - Secondary focus: *floating point math.*
- `Crypto:` Encryption and decryption benchmark based on code by Tom Wu (1698 lines).
  - Main focus: *bit operations*
- `Splay:` Data manipulation benchmark that deals with splay trees and exercises the automatic memory management subsystem (394 lines).
  - Main focus: *Fast object creation, destruction*
- `SplayLatency:` The Splay test stresses the Garbage Collection subsystem of a VM. SplayLatency instruments the existing Splay code with frequent measurement checkpoints. A long pause between checkpoints is an indication of high latency in the GC. This test measures the frequency of latency pauses, classifies them into buckets and penalizes frequent long pauses with a low score.
  - Main focus: *Garbage Collection latency*
- `EarleyBoyer:` Classic Scheme benchmarks, translated to JavaScript by Florian Loitsch's Scheme2Js compiler (4684 lines).
  - Main focus: *Fast object creation, destruction*
  - Secondary focus: *closures, arguments object*
- `pdf.js:` Mozilla's PDF Reader implemented in JavaScript. It measures decoding and interpretation time (33,056 lines).

  - Main focus: *array and typed arrays manipulations.*
  - Secondary focus: *math and bit operations, support for future language features (e.g. promises)*

- `Mandreel:` Runs the 3D Bullet Physics Engine ported from C++ to JavaScript via Mandreel (277,377 lines).
  - Main focus: *emulation*

- `MandreelLatency:` Similar to the SplayLatency test, this test instruments the Mandreel benchmark with frequent time measurement checkpoints. Since Mandreel stresses the VM's compiler, this test provides an indication of the latency introduced by the compiler. Long pauses between measurement checkpoints lower the final score.
  - Main focus: *Compiler latency*

- `GB Emulator:` Emulates the portable console's architecture and runs a demanding 3D simulation, all in JavaScript (11,097 lines).
  - Main focus: *emulation*

- `Code loading:` Measures how quickly a JavaScript engine can start executing code after loading a large JavaScript program, social widget being a common example. The source for this test is derived from open source libraries (Closure, jQuery) (1,530 lines).
  - Main focus: *JavaScript parsing and compilation*

- `Box2DWeb:` Based on Box2DWeb, the popular 2D physics engine originally written by Erin Catto, ported to JavaScript. (560 lines, 9000+ de-minified)
  - Main focus: *floating point math.*
  - Secondary focus: *properties containing doubles, accessor properties.*

- `Typescript:` Microsoft's TypeScript compiler is a complex application. This test measures the time TypeScript takes to compile itself and is a proxy of how well a VM handles complex and sizable Javascript applications (25,918 lines).
  - Main focus: *run complex, heavy applications*

Bibliography

[1] Azul Systems. ReadyNow! Technology for Zing. http://www.azulsystems.com/solutions/zing/readynow, 2015.

[2] Azul Systems. Zing: Java for the Real Time Business. http://www.azulsystems.com/products/zing/whatisit, 2015.

[3] ETH. ETH Data Center Observatory. https://wiki.dco.ethz.ch/bin/viewauth/Main/Cluster, 2015.

[4] Google. Octane 2.0 Benchmark. https://developers.google.com/octane/, 2015.

[5] T. Hartmann, A. Noll, and T. R. Gross. Efficient code management for dynamic multi-tiered compilation systems. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14, Cracow, Poland, September 23-26, 2014*, pages 51–62, 2014.

[6] V. Ivanov. JIT-compiler in JVM seen by a Java developer. http://www.stanford.edu/class/cs343/resources/java-hotspot.pdf, 2013.

[7] Oracle Corporation. Code for advancedThresholdPolicy.hpp. http://hg.openjdk.java.net/jdk9/hs-comp/hotspot/file/63337cc98898/src/share/vm/runtime/advancedThresholdPolicy.hpp, 2013.

[8] Oracle Corporation. hprof - A heap/CPU profiling tool. https://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html, 2014.

[9] Oracle Corporation. javac - Java programming language compiler. https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html, 2014.

[10] M. Paleczny, C. Vick, and C. Click. The Java HotSpot™Server Compiler. https://www.usenix.org/legacy/events/jvm01/full_papers/paleczny/paleczny.pdf, 2001. Paper from JVM '01.

[11] T. Rodriguez and K. Russell. Client Compiler for the Java HotSpot™Virtual Machine: Technology and Application. http://www.oracle.com/technetwork/java/javase/tech/3198-d1-150056.pdf, 2002. Talk from JavaOne 2002.

[12] Standard Performance Evaluation Corporation. SPECjvm2008 Benchmark. https://www.spec.org/jvm2008/, 2008.