4 Performance

This section evaluates the performance of the cached profile implementation using two modern benchmark suites. The goal is to provide indicators of the performance influence and analyze where these performance differences come from.

4.1 Setup

To provide reliable and comparable results, all benchmarks are executed on single nodes of the Data Center Observatory provided by ETH [3]. A node features two 8-Core AMD Opteron 6212 CPUs clocked at 2600 MHz with 128 GB of DDR3 RAM and a solid state disk for storage. The nodes are running Fedora 19 with Linux kernel 3.14.27 and GCC 4.8.3. All JDK builds are compiled on the nodes itself. The benchmarks suites are each executed on a dedicated node to prevent any influence of inter node performance differences.

The following benchmark suites are used:

1. **SPECjvm 2008:** Developed by Standard Performance Evaluation Corporation, SPECjvm aims to measure the performance of the Java Runtime Environment [14]. We use version 2008 and we run a subset of 17 out of a total of 21 benchmarks. 4 are omitted due to incompatibility with openJDK 1.9.0.
   SPECjvm reports the number of operations per minute (ops/m). This is used to compare the performance and more ops/m equals better performance.

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 of Hotspot. We use version 2.0, which consists of 17 individual benchmarks, of which 16 are compatible with openJDK 1.9.0.
   Octane gives each benchmark execution a score reflecting the performance. the higher the score, the better the performance.

A more detailed description of the benchmark suites used in this performance evaluation can be found in Appendix A.4.

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

4.2 Benchmark performance

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

The different CacheProfilesModes were implemented to be able to compare the performance influence of design decisions. We expect the modes to produce different results and the following list suggests reasons for these performance differences:

- If the cached profiles fit nicely to the current method execution, compiling these methods earlier than in the baseline will save executions at lower tiers and decrease the time needed for warmup. Benchmarks with these properties should achieve a performance improvement and favor `Mode 0`.
- In case many methods are compiled early this could overload the compile queue and delay compilation of these methods. Also, if the cached profile does not fit the current method execution (the limit of 10 deoptimization was reached) the JVM will use freshly generated profiles. These profiles could be sparse when using lowered compilation thresholds. In these scenarios we expect `Mode 1` to outperfom `Mode 0`.
- `Mode 2` keeps the steps of the original tiered compilation and is considered the most conservative mode. `Mode 2` also does not modify the compilation thresholds and therefore puts the same load on the compile queue than the baseline version. When using this mode, we can exclude the effect of earlier compilation and the performance only originates from the faster C1 phase and the better code quality in C2.

4.2.1 SPECjvm warmup performance

The longer a program is running, the less impact a faster warmup has. Instead of using SPECjvm's default values, we limited SPECjvm to 1 single operation, which depending on the benchmark, takes around 6 to 40 seconds. Additionally, the JVM gets restarted between each single benchmark, to prevent methods shared between benchmarks being compiled already.

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

We then use a single benchmark run, where we configure the JVM to dump the profiles to disk. This run is not limited to a single operation but instead uses the default values of the benchmark. By default the benchmark is limited by time and runs for about 6 minutes, including 2 minutes warmup, which do not count towards the output. The reason is, that we want to speed up the warmup. but still keep the same performance for the rest of the benchmark execution. Dumped profile from a single operation might contain less matured information, which are fine if the benchmark is short running, but could negatively affect long term performance.

These profiles are then used in 3 individual runs by specifying the `-XX:CacheProfiles` flag. Each run is using one of the 3 different CacheProfilesModes. This process is repeated 50 times, while the profiles are only recreated after 10 iterations each. This is used to check whether different profile creation runs have any impact on the following performance measurements. However, the evaluation shows, that this is not the case.

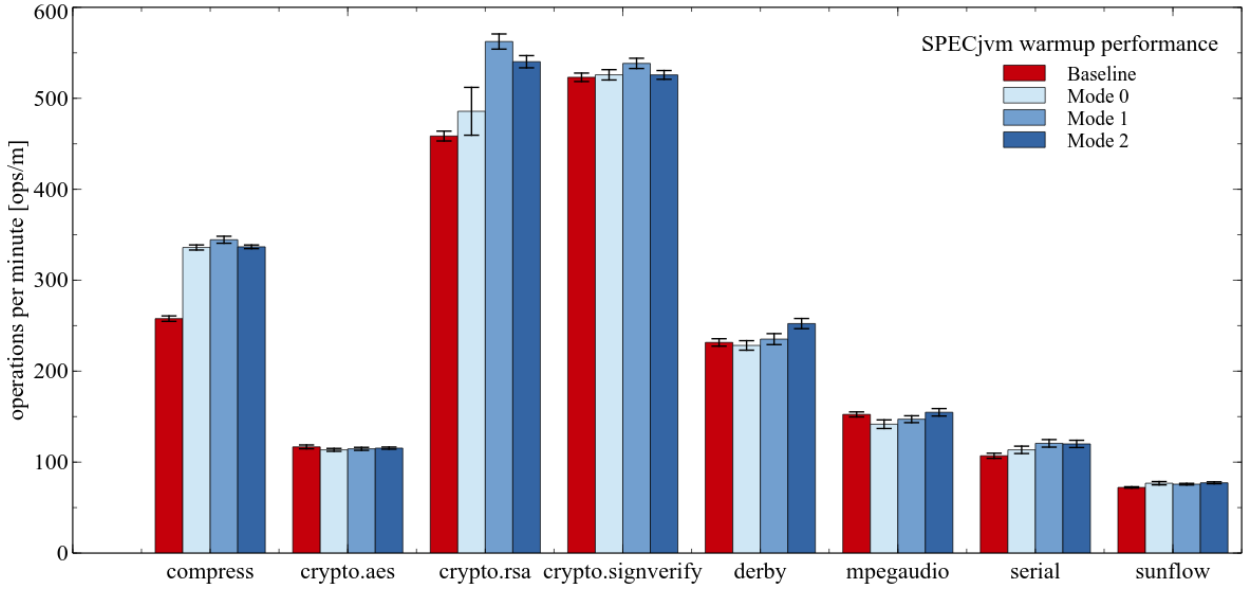

Figure 4.1: SPECjvm benchmarks on all different modes

Figures 4.1 and 4.2 shows the number of operations per minute, measured for each benchmark individually. Note, that the operations per minute is not to be confused with the previously mentioned limit of *1 operation* of the benchmark itself. Figure 4.3 summarizes the results by showing the relative performance compared to the baseline.

The individual benchmarks show different effects on performance. We see a performance increase up to around 34% in the compress benchmark (Mode 1) and a performance decrease of down to 20% in scimark.sparse.large (Mode 0).

Interestingly, the performance differences between the modes are not the same when comparing the individual benchmarks. For example, in crypto.rsa `Mode 0` clearly performs worst but in scimark.sparse.small it performs best. JVM performance is known to be very hard to predict and it seems not to be different when cached profiles are used. On average the performance of the benchmark warmup is improved by 2.64%, 3.37%, and 2.67% for Mode 0, Mode 1 and Mode 2.

Between the three different modes there is no clear *winner*. Each mode wins and loses in certain benchmarks against the others in terms of performance. However, in 12 out of 17 benchmarks

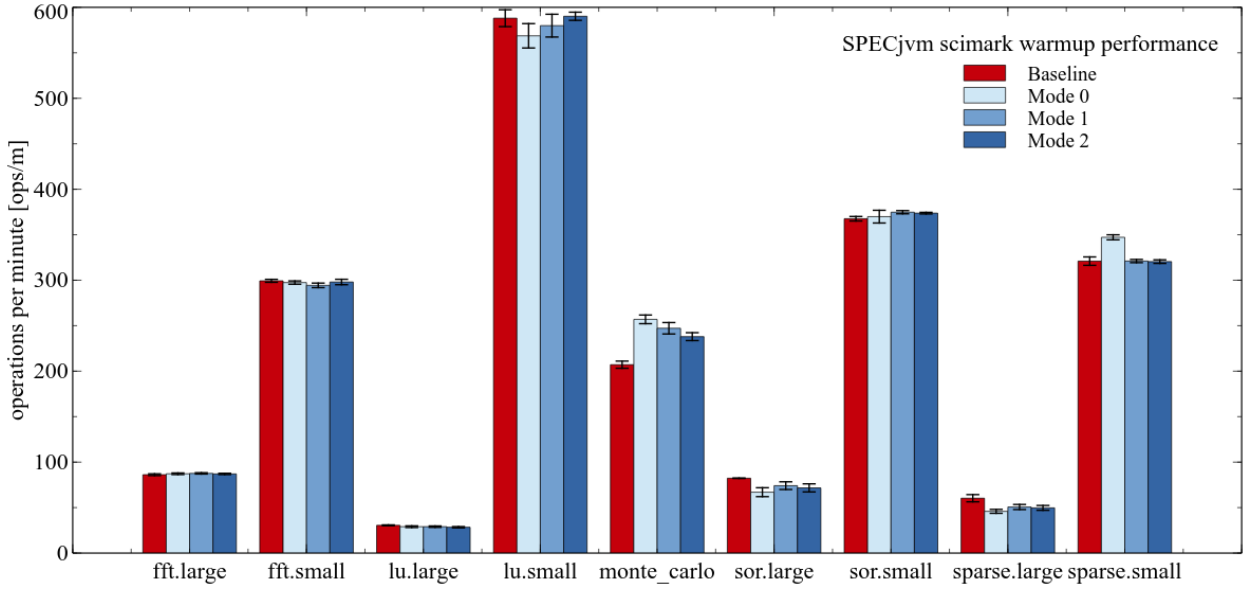

Figure 4.2: SPECjvm scimark benchmarks on all different modes

at least one of the CacheProfileModes improves performance.

As stated before, we expect the influence of cached profiles to be low, when running specJVM for the standard duration. Figure 4.4 shows the relative performance for all SPECjvm benchmarks running the default duration of 6 minutes. We see that the influence is not significant. More importantly, using the cached profiles does not decrease the performance of the long running benchmarks.

We will take a more detailed look at single benchmarks later in this chapter.

4.2.2 Octane performance

Since the individual Octane benchmarks are rather short (most of them run for between 4 and 30 seconds) and there is no simple way to run a fixed number of iterations, we run the Octane benchmarks completely. We still split up the execution in the individual benchmarks. The rest of the setup is identical to SPECjvm in Section 4.2.1.

The absolute results are shown in Figure 4.5 and a relative comparison with the baseline in Figure 4.6. Compared to SPECjvm the Octane performance is more scattered. The Richards benchmark increases by around 50% in Mode 0 while NavierStokes decreases by around 25% in Mode 1. In most benchmarks (9 out of 14) Mode 0 performs worst. `Mode 1` and `Mode 2` generally perform a little bit better, but in total only 6 out of 14 benchmarks result in a performance improvement in at least one mode. On average the performance differences over all individual benchmarks compared to the baseline are -5.94%, 0.58%, and 0.11% for Mode 0, Mode 1 and Mode 2.

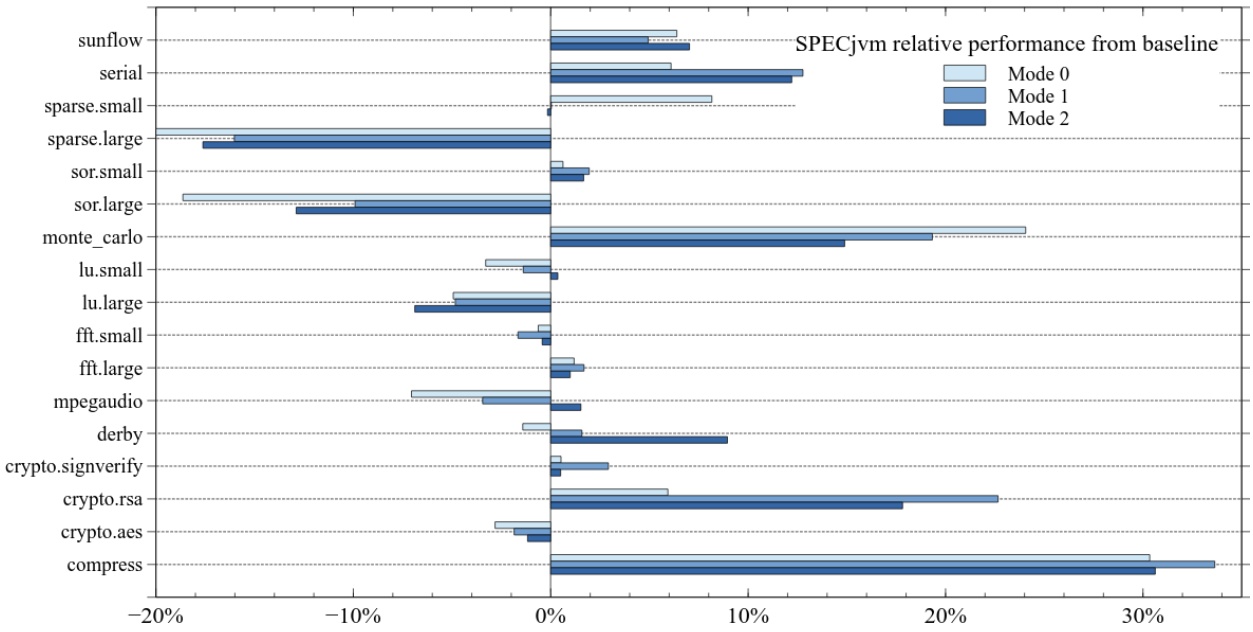

Figure 4.3: Relative performance from baseline for all SPECjvm benchmarks

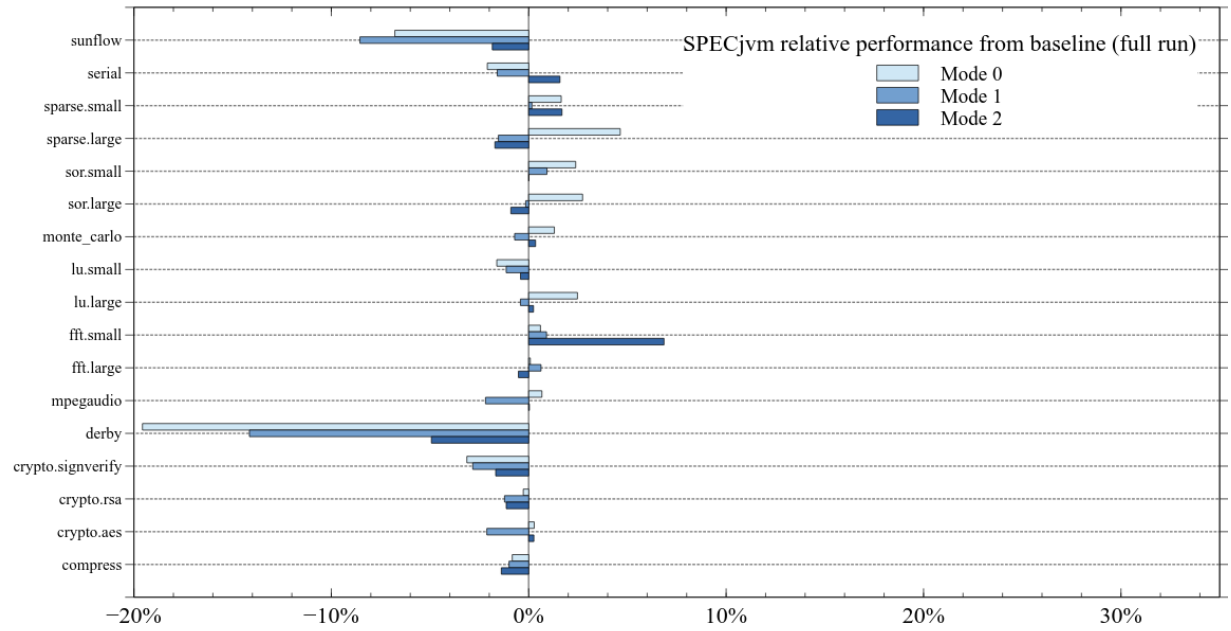

Figure 4.4: Relative performance from baseline for all SPECjvm benchmarks using a full run

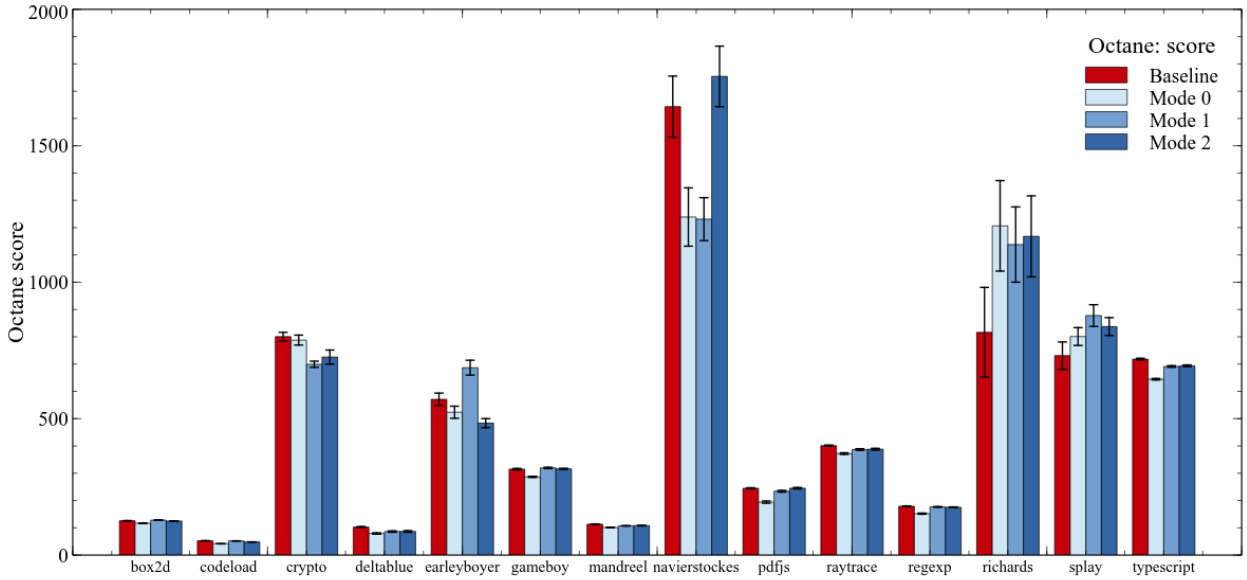

Figure 4.5: Octane benchmarks on all different modes

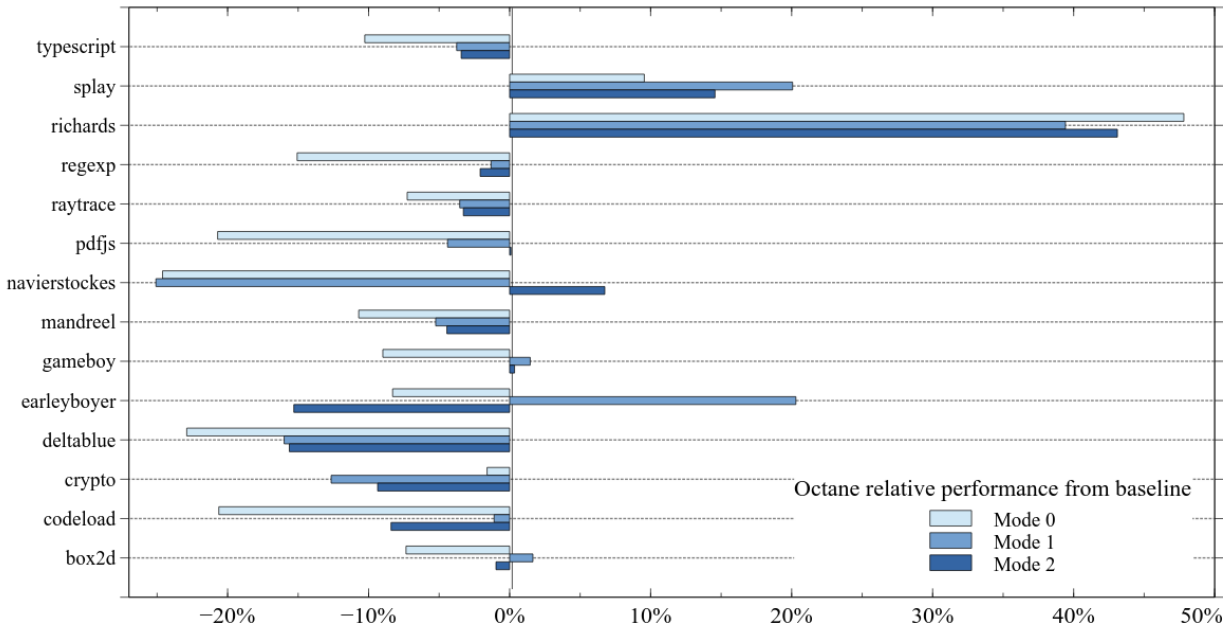

Figure 4.6: Relative performance from baseline for all Octane benchmarks

4.3 Deoptimizations

We aim to lower the time needed for warmup, by compiling methods earlier and at lower tiers. We also expect to decrease the number of deoptimizations by having more complete profiles available earlier, which ideally results in better compiled code quality. To measure the total amount of deoptimizations we added a new compiler flag `-XX:+PrintDeoptimizationCount`. The number of deoptimizations of the SPECjvm benchmarks are shown in Figure 4.7 and Figure 4.8. The Octane numbers are drawn in Figure 4.10. Again, we also included graphs that show the number of deoptimizations relative to the baseline runs in Figure 4.9 and Figure 4.11.

The measurements show, that when using `Mode 1` or `Mode 2`, we are able to reduce the deoptimizations significantly in all benchmarks except one (GameBoy). In `Mode 0`, there is a clear difference between SPECjvm and Octane. While in SPECjvm. the number of deoptimizations is similar to the other modes, in Octane, `Mode 0` increases the number by 30%. `Mode 0` also has the largest performance regression in Octane, the high amount of deoptimizations could be a sign for this result.

And while a low deoptimization number is a good indication of the increased code quality for methods being compiled with cached profiles, we could not find a direct correlation between number of deoptimizations and the performance results.

One possible reason is, that the amount of deoptimizations does not necessarily describe the performance impact. Especially, when considering multi-threaded systems, there can be a huge number of deoptimizations in performance uncritical threads that are avoided by using cached profiles and therefore heavily reduce the total counter. But if there is one very important method in a performance critical thread, which has executions that are not reflected in the cached profiles, this method could trigger only very few deoptimizations but still influence performance significantly.

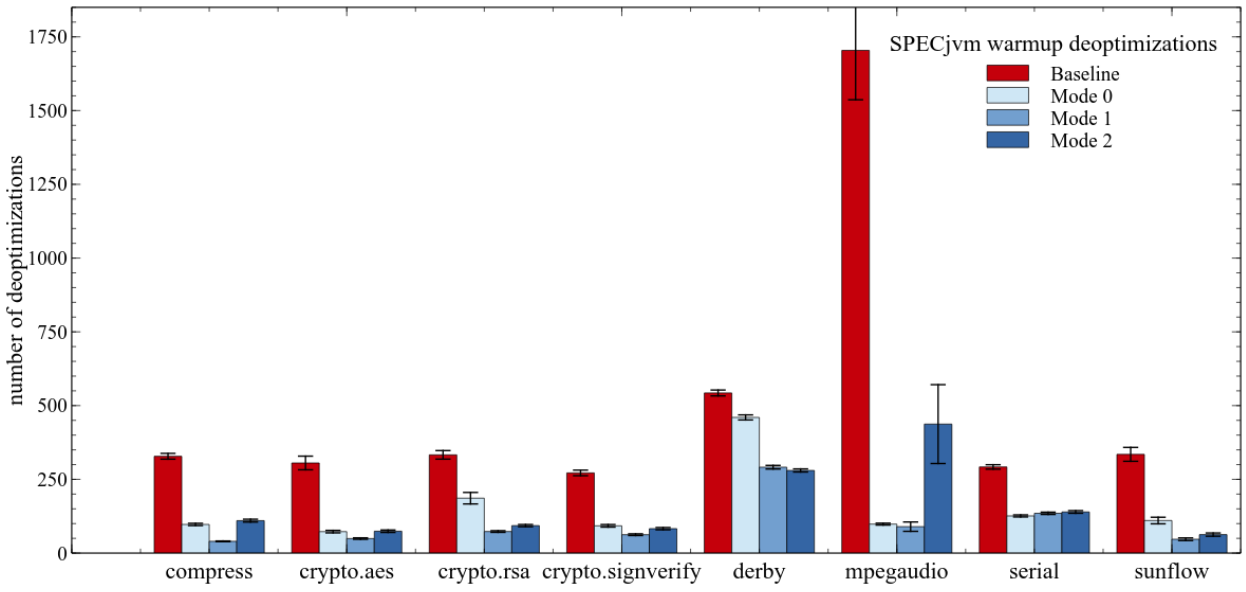

Figure 4.7: SPECjvm deoptimizations of all modes

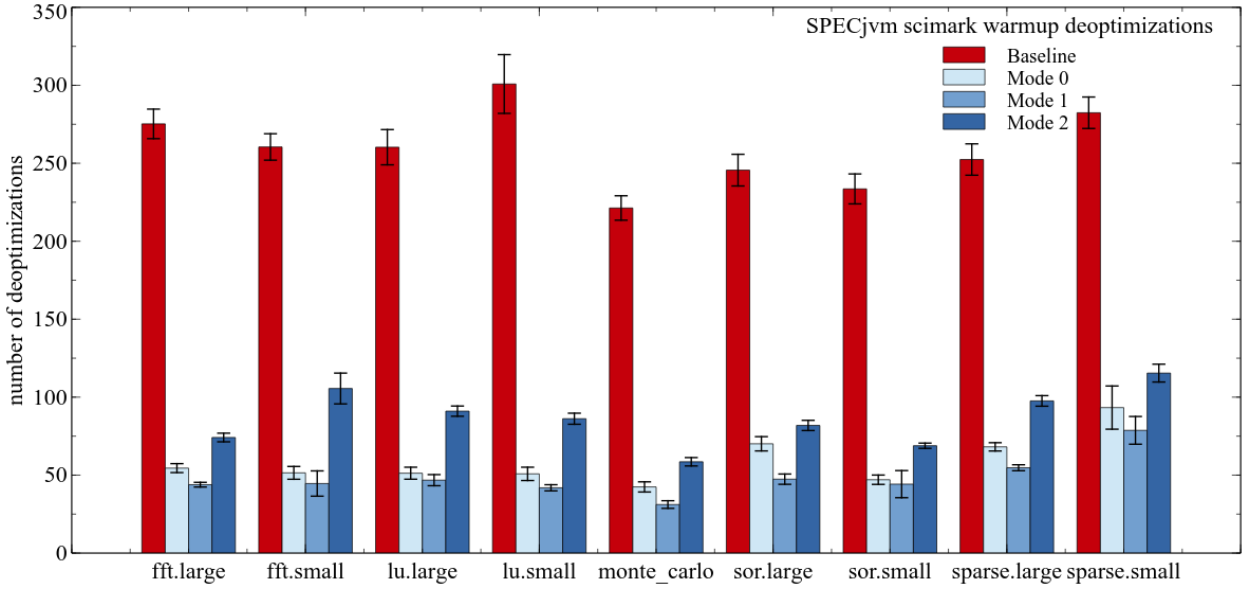

Figure 4.8: SPECjvm scimark deoptimizations of all modes

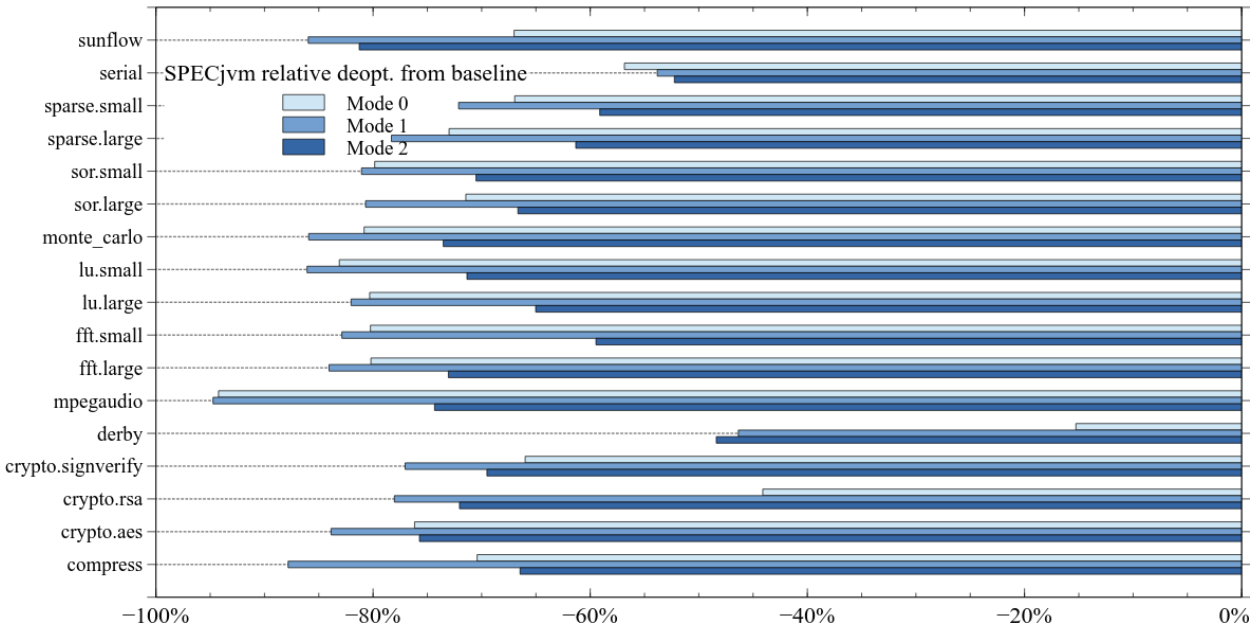

Figure 4.9: Relative deoptimizations from baseline for all SPECjvm benchmarks

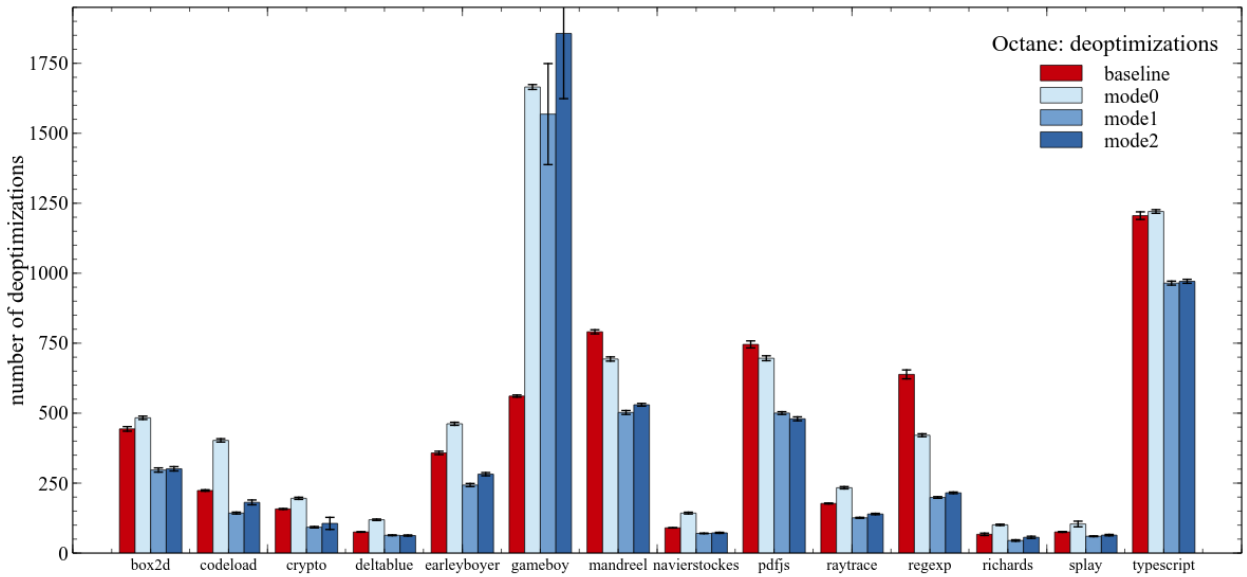

Figure 4.10: Octane deoptimizations of all modes

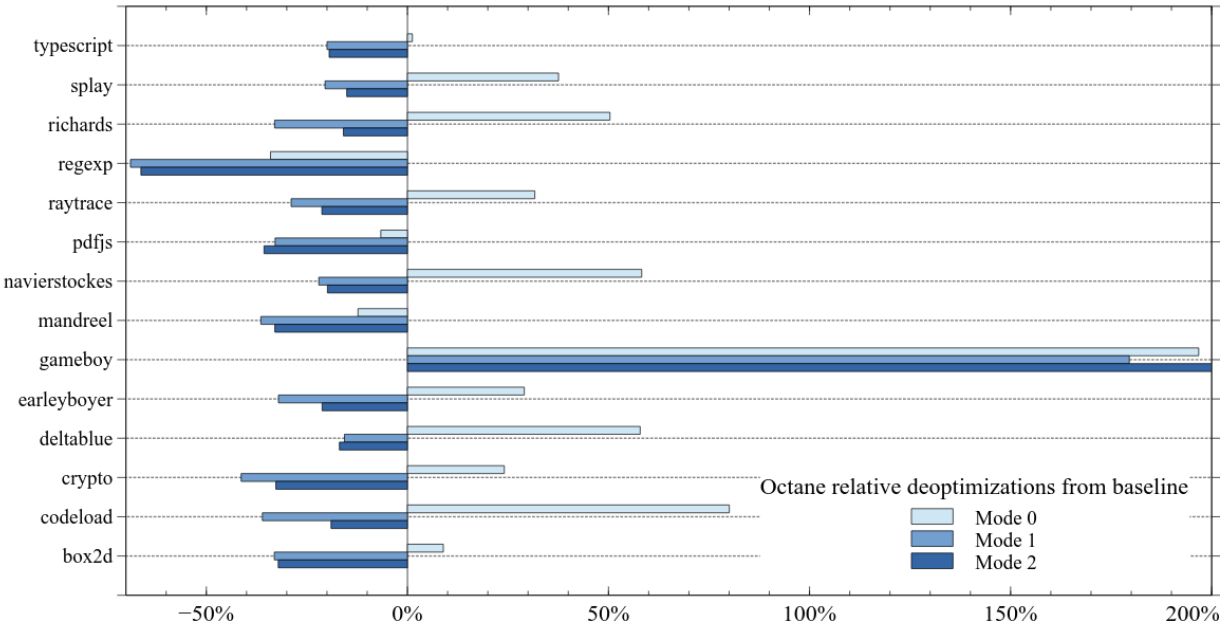

Figure 4.11: Relative deoptimizations from baseline for all Octane benchmarks

4.4 Effect on compile queue

When designing the different CacheProfilesModes we thought, that lowering the compilation thresholds will increase the load on the compiler, especially early in program execution. This results in the compiler not being able to handle all requests immediately and the compile queue fills up. Execution of compiled methods can be delayed and therefore performance degradation occur.

We added a new HotSpot flag `-XX:+PrintCompileQueueSize`, which allows us to trace the current number of methods that are scheduled for compilation. We selected 6 individual benchmarks and printed their C1 and C2 compile queue for all three CacheProfileModes in Figure 4.12 to Figure 4.19. The selected benchmarks and the reason why an individual analysis was performed are listed below:

- **Octane Richards:** This benchmarks achieved the highest performance benefit from using cached profiles in all three modes. We are interested to see if the compile queue load differs from worse performing benchmarks.
- **Octane EarleyBoyer:** This is a benchmark, were `Mode 1` performs significantly better than the other two modes. We chose this to clarify, whether `Mode 1` shows a different compile queue behavior.
- **Octane NavierStokes:** Navierstokes has a 8% performance increase in `Mode 2` but a 25% performance decrease in `Mode 0` and `Mode 1`. Motivation is the same as EarleyBoyer, except we focus on `Mode 2`.
- **Octane Deltablue:** This benchmark achieves the highest performance loss from using cached profiles in all three modes. Together with the best performing benchmark (Richards) we as ourselves, if the load on the compile queue indicates any performances differences.
- **SPECjvm compress:** Compress is the best performing SPECjvm benchmark when using cached profiles.
- **SPECjvm scimark.sparse.large:** This is the worst performing SPECjvm benchmark when using cached profiles.

Keep in mind that the reason for the runs using cached profiles starting their main amount of compilations delayed, is because the JVM need to parse the cached profile file first.

We realize that analyzing the compile queue does not really help us understanding the performance variations when using cached profiles. The graphs' courses, showing the C1 compile queue size over time, do not significantly differ from the baseline, nor is there a difference between the individual modes.

The C2 compile queue graphs show different courses but we again can not connect this with the benchmark performance.

Figure 4.13 shows the C2 compile queue of the Octane Richards benchmark. As expected, due to removing steps from the tiered compilation, we increased the load on C2 in `Mode 0` and `Mode 1` with compile queue peaks at around 20 scheduled compilations. Nevertheless, these modes have a performance increase of close to 50% better than the baseline. `Mode 2`, which was designed to keep the original tiered compilation steps unmodified, does not have similar peaks but nevertheless achieves similar performance.

EarleyBoyer's compile queue is displayed in Figure 4.13. `Mode 1` performs better than the other two modes and compared to `Mode 0` puts even more pressure on the compile queue. It is interesting, that in this particular benchmark even the baseline version puts a lot of pressure on the compile queue early on.

In Figure 4.17 we see NavierStokes' compile queue. `Mode 2` performs best but we can not derive any indications why this is the case from looking at the queue size.

The Deltablue benchmark shown in Figure 4.19 has the worst performance when using cached profiles but the compile queue size looks very similar to the one of the Richards benchmark, where performance is significantly better.

We will omit looking at the SPECjvm benchmarks since they do not offer any new insights. The graphs can be found in the Appendix A.6.

The detailed analysis of the compile queue shows that our thoughts about the effect on the compile queue were not unfounded for most of the selected benchmarks. However, we were not able to relate these influences to actual performance effects. Especially, overloading the compile queue does not necessarily affect performance negatively.

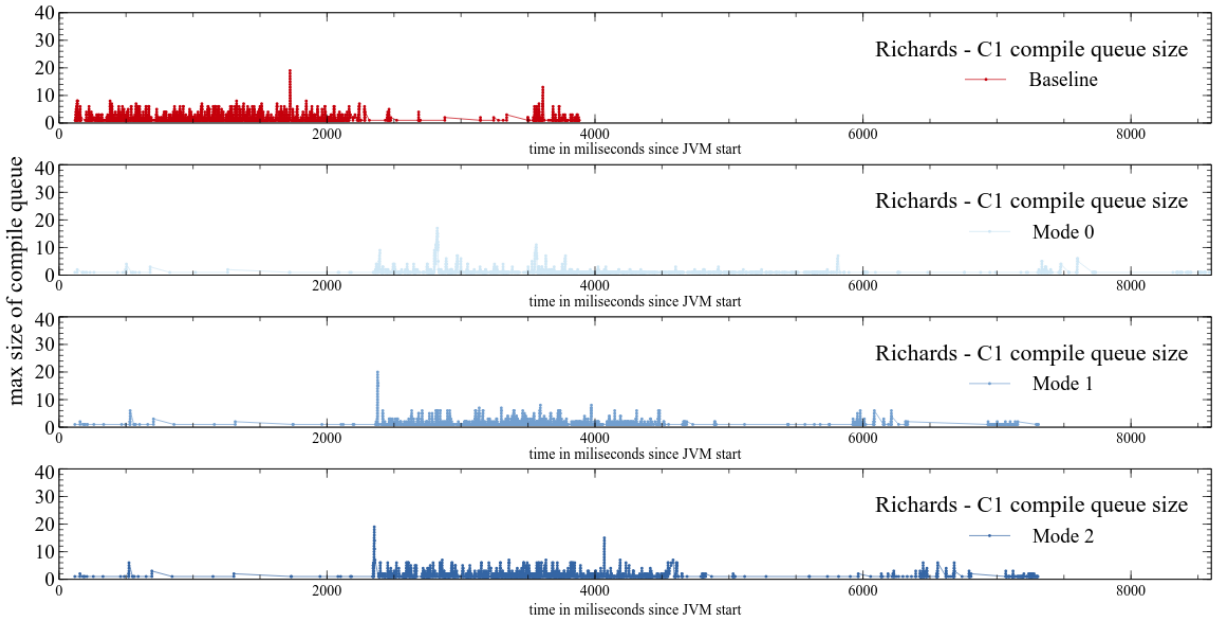

Figure 4.12: C1 Compile queue size over time Octane Richards benchmark

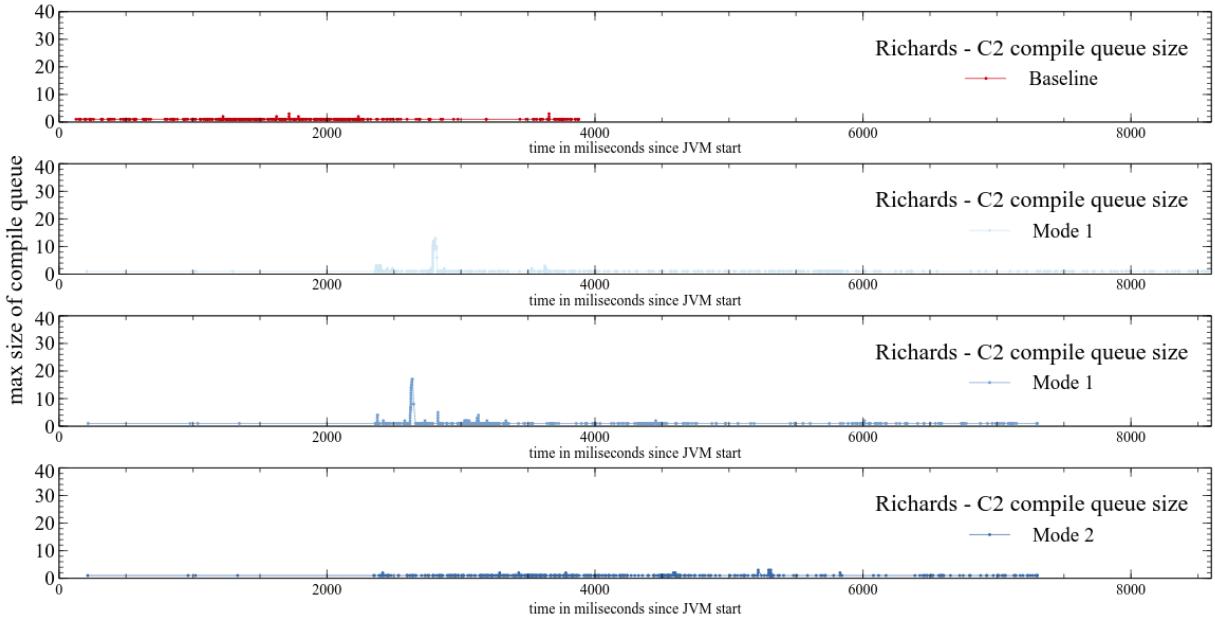

Figure 4.13: C2 Compile queue size over time Octane Richards benchmark

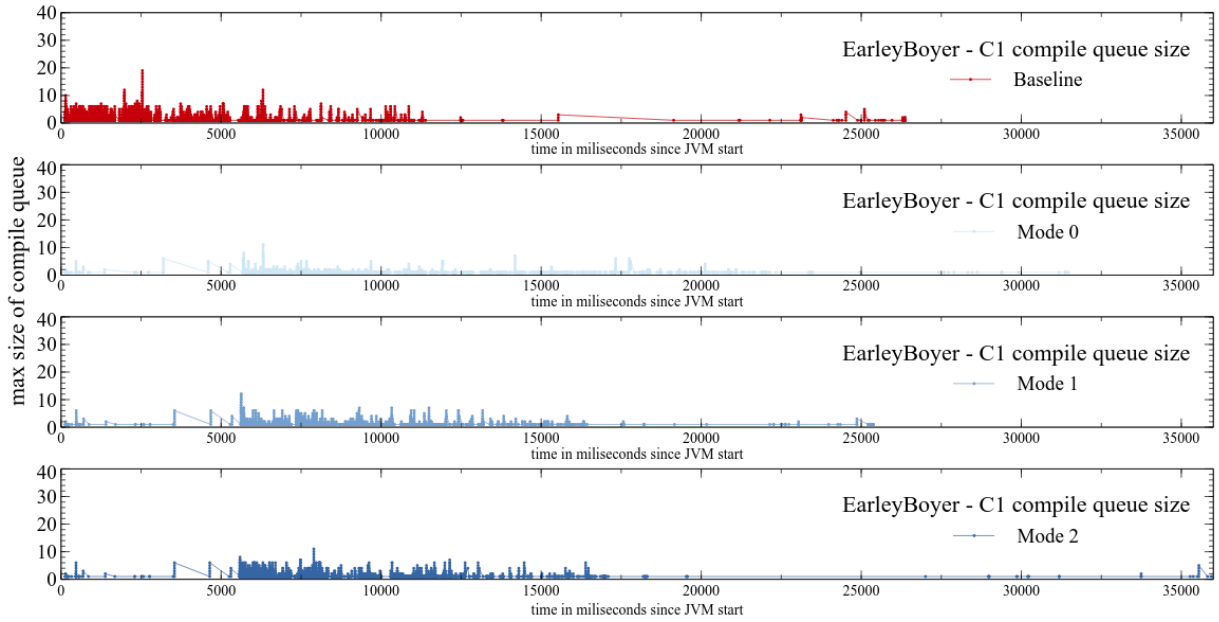

Figure 4.14: C1 Compile queue size over time Octane EarleyBoyer benchmark

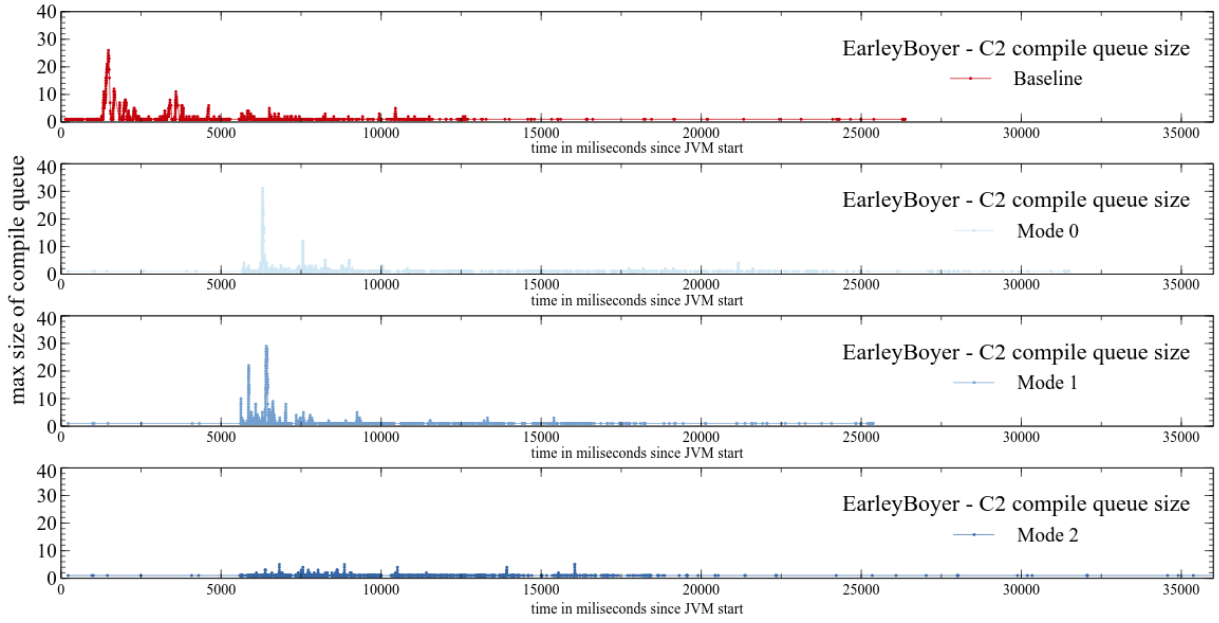

Figure 4.15: C2 Compile queue size over time Octane EarleyBoyer benchmark

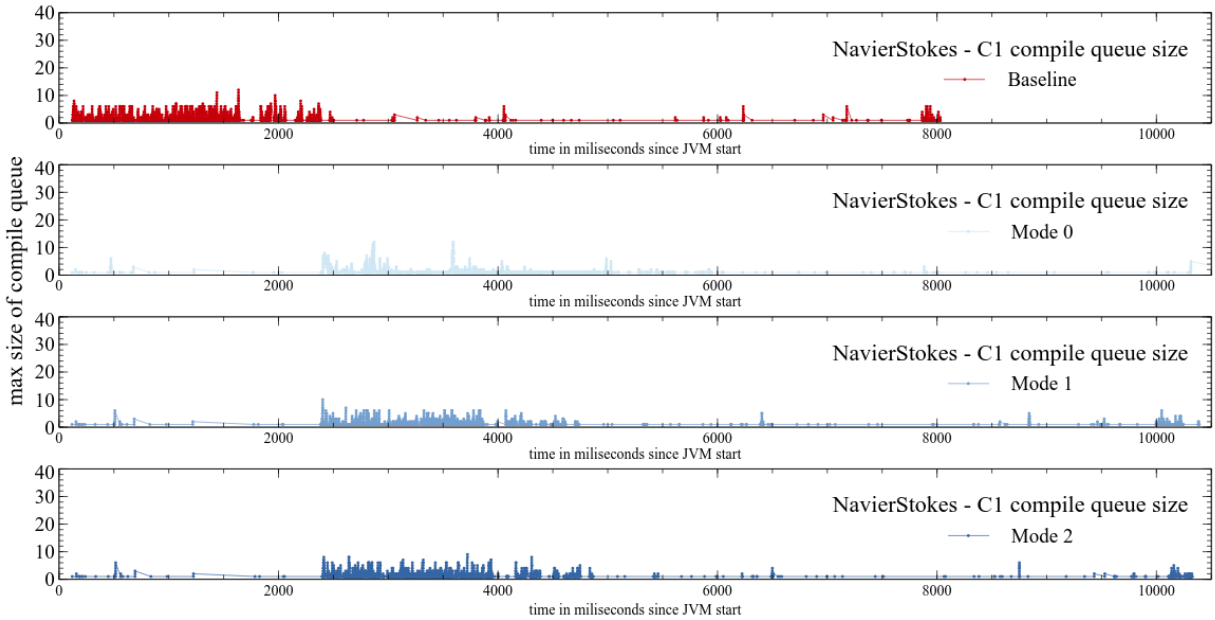

Figure 4.16: C1 Compile queue size over time Octane NavierStokes benchmark

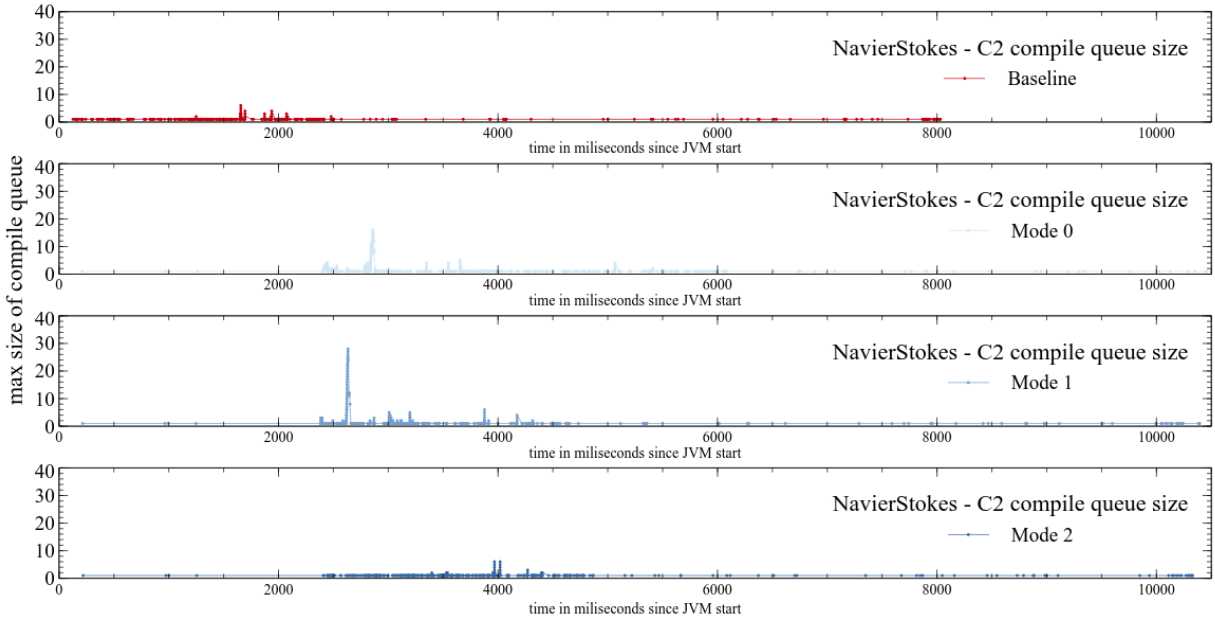

Figure 4.17: C2 Compile queue size over time Octane NavierStokes benchmark

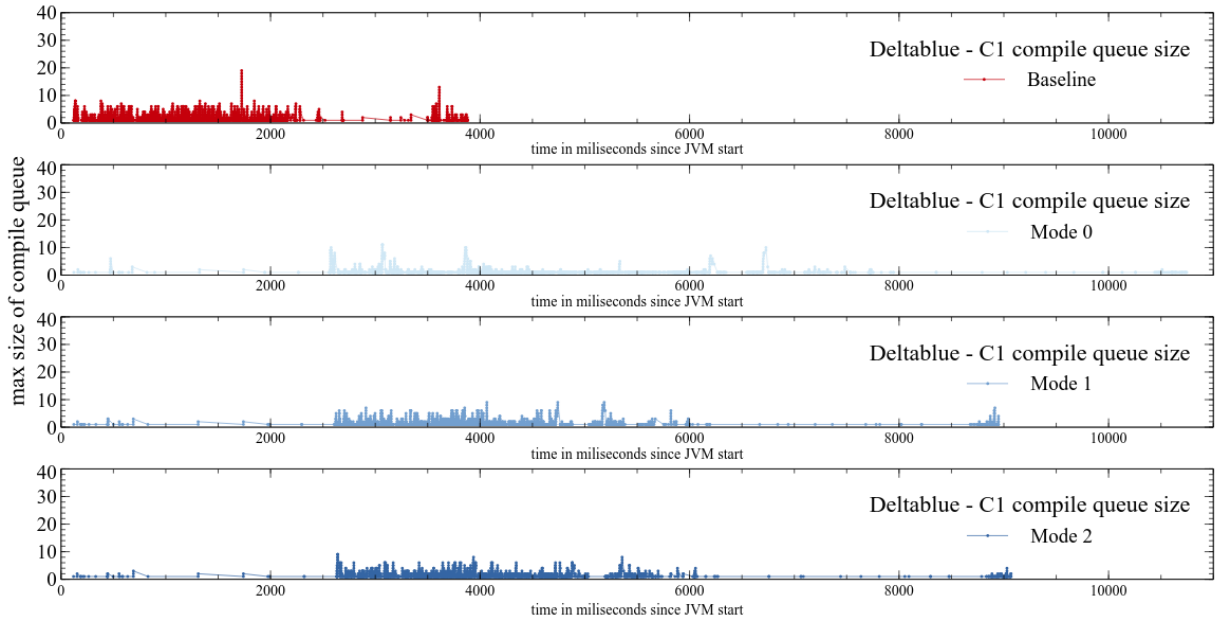

Figure 4.18: C1 Compile queue size over time Octane Deltablue benchmark

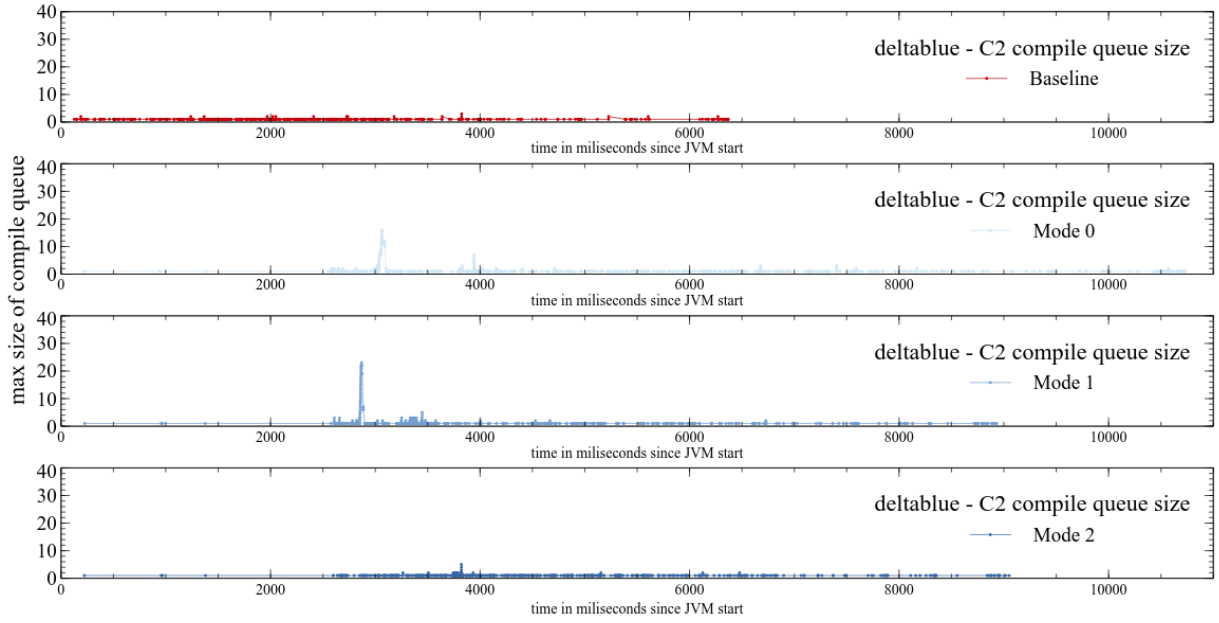

Figure 4.19: C2 Compile queue size over time Octane Deltablue benchmark

4.5 Number and type of compilations

In this section, we take a look on how cached profile modify the ratio of C1 and C2 compilations and if there is a correlation between percentage of methods using cached profiles and the resulting performance.

We continue the focus on the 6 individual benchmarks, selected in Section 4.4. We use the new HotSpot flag `-XX:+PrintCacheProfiles`, that prints out the level of each compilation and whether or not it uses cached profiles.

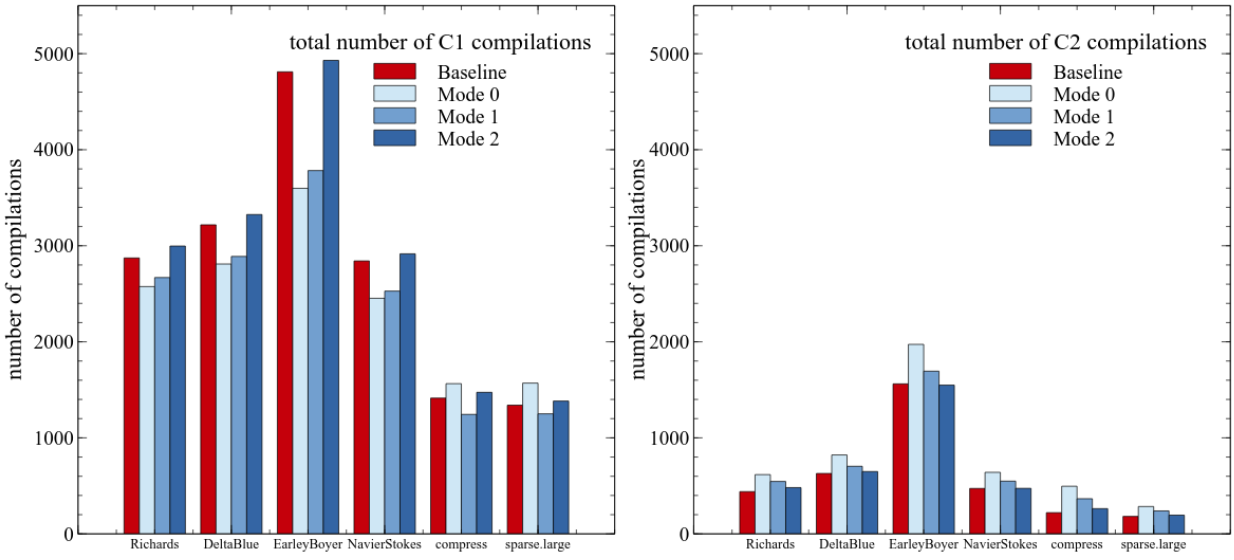

Figure 4.20: Number of compilations for some specJVM and octane benchmarks

Figure 4.20 shows the total amount of compilations, split in C1 and C2. We see, that the Octane benchmarks and the SPECjvm benchmarks behave differently. While the 4 Octane ones achieve a lower amount of C1 compilations in `Mode 0` and `Mode 1`, `Mode 2` is similar to the baseline. The two SPECjvm benchmarks have more C1 compilations in `Mode 0`, less in `Mode 1` and the same amount in `Mode 2` compared to the baseline.

The changes of the amount of C2 compilations are very similar in all benchmarks. Using `Mode 0` and `Mode 1` results in more C2 compilations than the baseline and `Mode 2` achieves around the same amount as the baseline.

If we recall the differences between the modes, these results make sense. `Mode 0` lowers the thresholds of C1 compilations in case the method has a cached profile and compiles with C2 instead. This reduces the number of C1 compilations in favor of more C2 compilations. `Mode 1` does not lower the thresholds but it still promotes some C1 compilations to C2 compilations due to the fact that

C1 requests of methods with a cached profile get compiled with C2 immediately. `Mode 2` leaves the tiered compilation completely untouched and therefore has very similar compilation numbers to the baseline.

Furthermore, we are interested how many of the compilations make use of cached profiles. We want to be as close to 100% as possible, but the experiments show that around 65%-70% of the compilations use cached profiles. The compilation replay functionality does not support certain methods, e.g. lambda expressions. Since the profile caching implementation is based on compilation replay, it will also not compile these methods using cached profiles. Additionally, we do not use any profiles for compilation Level 1 and Level 2 as described in Section 3.3.

In Figure 4.21 to Figure 4.26 we show pie charts that visualize the portion of specific compilation types. When comparing different benchmarks of the same CacheProfilesMode, we realize, that the share of each compile type is constant. The pie charts of `Mode 0` and `Mode 1` differ only slightly. In all benchmarks, using `Mode 1` invokes less compilations using cached profiles than `Mode 1`. In `Mode 2`, we see Level 2 compilations appearing due to the changed tiered compilation transitions. The number of Level 3 compilations is almost unchanged compared to `Mode 1` because these are compilations of methods where no profiles from C2 compilations are available. The Level 2 compilations only happen if a C2 profile is cached and usually result in an additional Level 4 compilation.

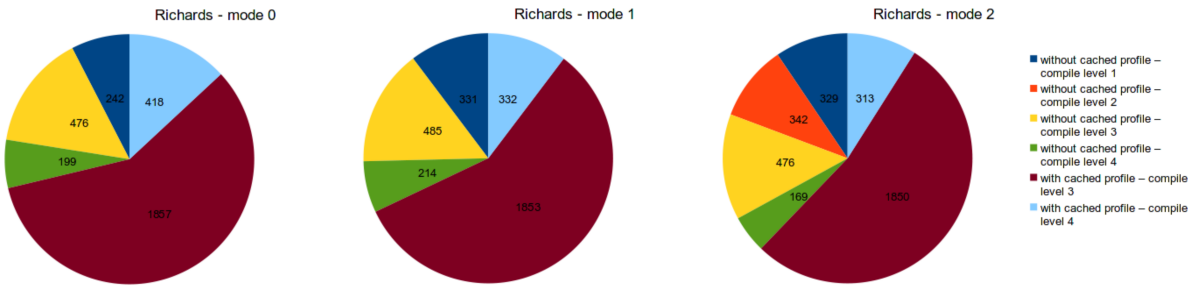

Figure 4.21: Ratio of compilations Octane Richards benchmark

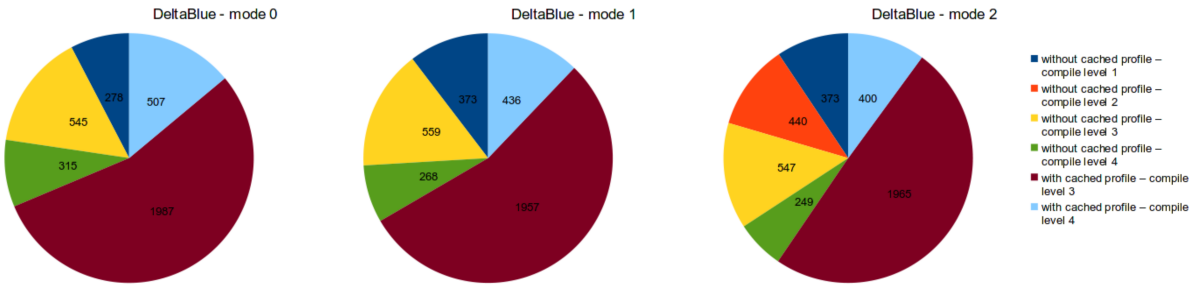

Figure 4.22: Ratio of compilations Octane Deltablue benchmark

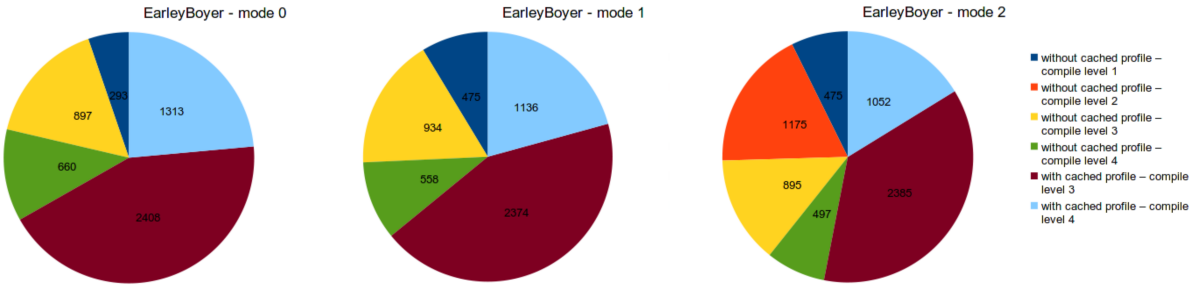

Figure 4.23: Ratio of compilations Octane EarleyBoyer benchmark

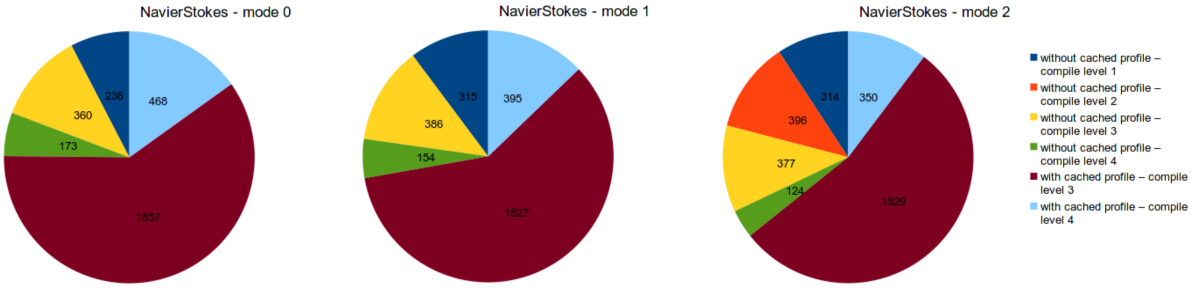

Figure 4.24: Ratio of compilations Octane NavierStokes benchmark

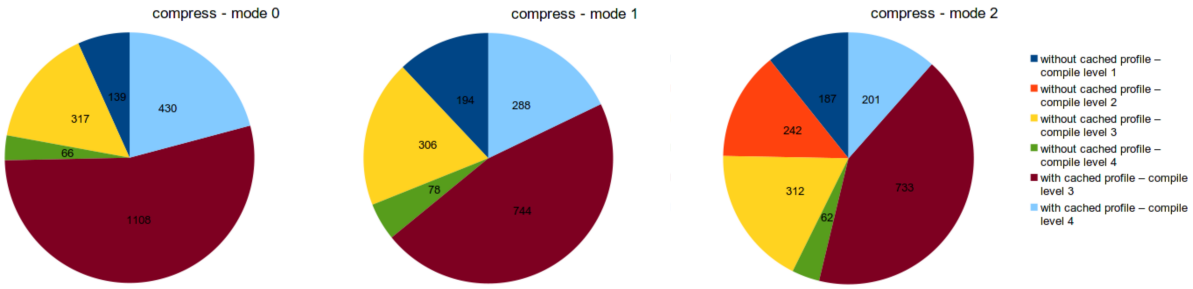

Figure 4.25: Ratio of compilations SPECjvm compress benchmark

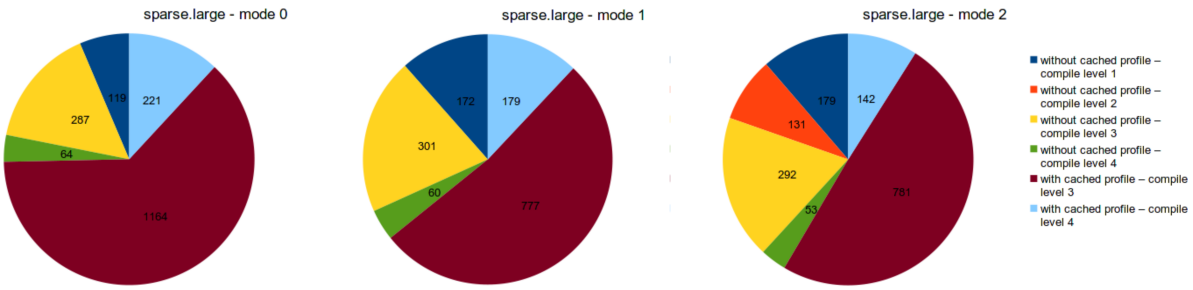

Figure 4.26: Ratio of compilations SPECjvm sparse.large benchmark

4.6 Effect of interpreter profiles

Our system makes use of two types of cached profiles. Profiles, that are gathered by the interpreter and used by the C1 compiler and profiles, that are gathered by a C1 compiled method and used when compiling with C2.

We added a HotSpot flag that allows us to specify the minimum level of a compilation that dumps profiles (`-XX:DumpProfilesMinTier=`level). Previous measurements were done using Level 3 which dumps profiles during C1 and C2 compilations.

However, we are also interested how the system performance changes when only C2 compiler profiles are used. The system will then only use cached profiles where a C2 compilation took place in the previous profile generation run. We use the same setup as before and run the individual SPECjvm (see Figure 4.27 and Octane (see Figure 4.28 benchmarks.

Most of the benchmarks do not show significantly different results compared to Section 4.2, where both types of cached profiles are used. There are a few benchmarks, where individual modes now improve the performance, while having a performance drop when both, C1 and C2 profiles, are used (e.g. NavierStokes `Mode 0`). But we also experience the other way around, for example in benchmark Splay `Mode 2`. In these individual cases, we believe that for example a benchmarks C1 compilation does not profit from having cached profiles and therefore using them will even decrease performance (also see Section 3.2). This happens, because methods are first compiled using cached profiles and after ten deoptimizations freshly generated profiles are used instead.

The results let us conclude, that the performance differences to the baseline are mostly due to the code quality of C2 compilations. Even though the number of C1 compilations is usually a lot higher than the number of C2 compilations, C2 compilations seem more crucial to the methods performance. Since C2 is the maximum compilation level, a program might spend most of it's time in C2 compiled code.

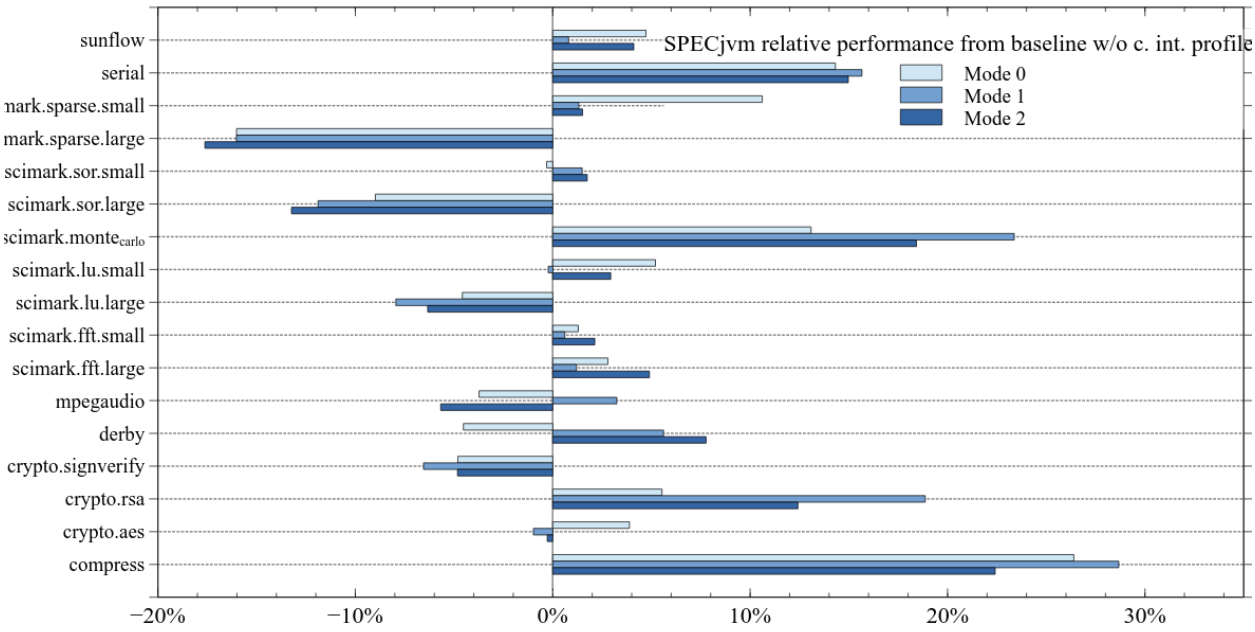

Figure 4.27: Relative performance from baseline for all SPECjvm benchmarks without using cached interpreter profiles

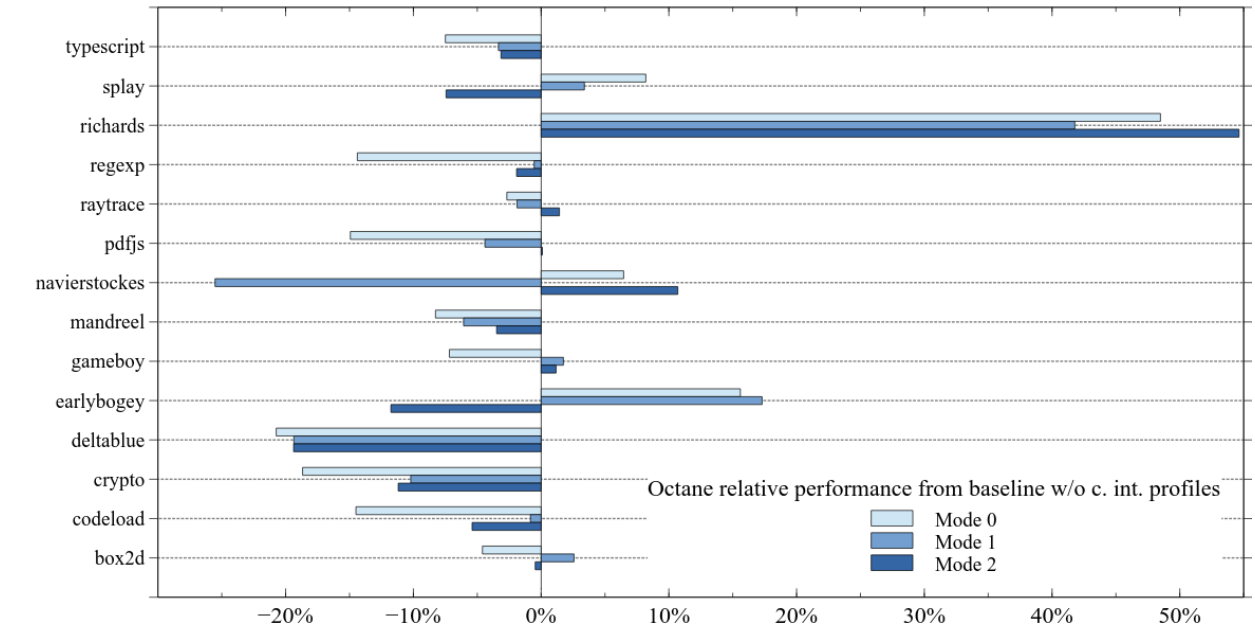

Figure 4.28: Relative performance from baseline for all Octane benchmarks without using cached interpreter profiles

4.7 Effect of intrinsified methods

Most modern JVMs implement use *method intrinsics* to further optimize commonly used Java core library methods [7]. This means, that the JIT compiler will not compile the method based on the Java bytecode, but instead replace it with a predefined, and manually optimized assembly code snippet. The current list of methods where intrinsics are available can be found in the code reference [11].

Intrinsics are mostly used in C1 and C2 compilations and the emitted code is independent of the current available profiling information. This means if many methods of a benchmark are intrinsified, the influence of profiles, and therefore cached profiles as well, decreases. We want to know, whether this has a large influence on the performance of cached profiles. An compilation of an intrinsified method has no advantage of having a rich profiling information but will still be influenced by modified compilation thresholds. E.g. lowering the threshold will intrinsify methods earlier and therefore speed up execution.

The results of both benchmark suites with disabled method instrinsics can be found in Figures 4.29 and Figure 4.30. For SPECjvm, We see that there are small performance differences in individual benchmarks but we can not conclude a major influence to the behavior of cached profiles and their influence on performance. Note, that the serial benchmark does not work with disabled intrinsics.

Most of the Octane benchmarks do not work when intrinsics are disabled and the ones that work run a lot slower. We think in these benchmarks other unconsidered side effects occur and an analysis regarding cached profiles would not be accurate.

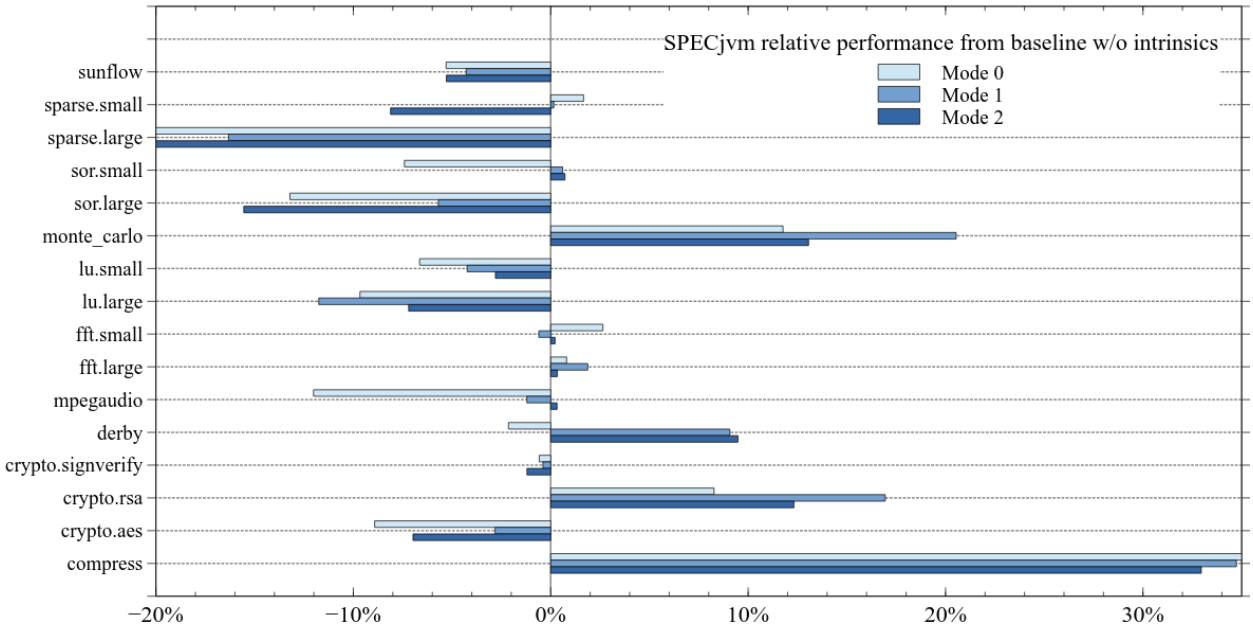

Figure 4.29: Relative performance from baseline for all SPECjvm benchmarks without intrinsified methods

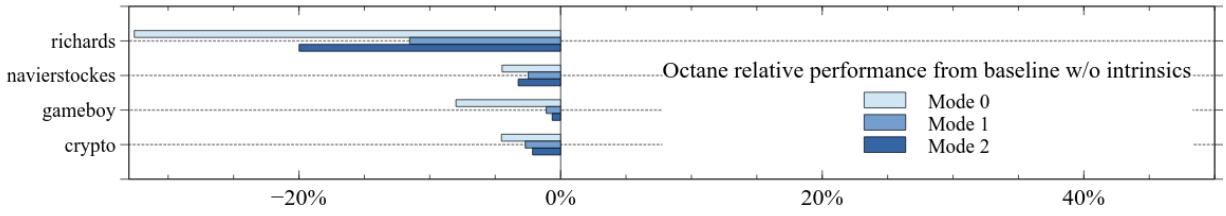

Figure 4.30: Relative performance from baseline for all working Octane benchmarks without intrinsified methods

5 Possible improvements

While this thesis provides a first look on implementing a system to reuse profiles from previous JVM runs, it leaves room for several improvements.

- There is a time overhead for parsing the cached profiles file when the JVM boots up. This could be lowered by finding more compact ways of storing profiles to disk or storing the cached profiles in main memory.
- The data structure of the cached profiles is a simple C heap array. The worst case lookup is $O(n)$ with n being the number of cached method compilations. By using a more advanced tree-like data structure we could improve this to $O(log(n))$.
- Additionally, all method compilations get dumped and stored. The system could be extended in a way, that only the last compilation record of a method is kept in the file. On cost of additional memory overhead, this can decrease the size of the cached profile file and result in a lower parsing time.
- Currently, only the profiles from a single run are used. A possible improvement is to use multiple executions for gathering the cached profiles and investigate in ways to merge the profiling information. More complete profiles can be achieved which could further reduce the number of deoptimizations.
- In addition to merging multiple profiles we also though about the possibility to modify the cached profile. That would allow the JVM user to manually improve profiling information by using his knowledge of the method execution which might not be available to the compiler.

6 Conclusion

Current Java Virtual Machines (JVM) like HotSpot gather profiling information about executed methods to improve the quality of the compiled code. This thesis presents a way to cache these profiling information to disk, so they can be reused in future runs of the JVM.

The expected advantage is a faster warmup of the JVM because the JVM does not need to spend time profiling the code and can use cached profiles directly. Furthermore, since the cached profiles originate from previous compilations where extensive profiling already happened, compilations using these profiles produce more optimized code which decreases the amount of deoptimizations.

We show, using two benchmark suites, that cached profiles can indeed improve warmup performance and significantly lower the amount of deoptimizations. In addition, we tested individual benchmarks for the impact of cached profiles on the load of the compile queue and the amount and type of compilations. Results show, that both do not give clear indications on the performance.

The ideas and functionality is implemented in the HotSpot JVM (openJDK9). It provides the user of the JVM several choices to use the system and allows fine-grained selection of the cached methods. We believe, that cached profiles are a valuable asset in scenarios where a fast JVM warmup is needed and we have shown that caching profiles can significantly decrease performance fluctuations of JIT compiled code.

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 Cached profile example

Listing A.1: Example of cached profiling information

```
1 # 189 ciObject found
2 ciMethod java/lang/Double toString (D)Ljava/lang/String; 25 1 7 0 0
3 ciMethod java/lang/Long toString (J)Ljava/lang/String; 9 1 3 0 0
4 ciMethod java/lang/Long valueOf (J)Ljava/lang/Long; 25 1 7 0 0
5 ciMethod java/lang/Long longValue ()J 9 1 3 0 -1
6 ciMethodData java/lang/Double toString (D)Ljava/lang/String; 1 7 orig 280 152 101 83 95 3 ↩
      127 0 0 88 140 138 72 3 127 0 0 144 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ↩
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 77 68 79 ↩
      32 101 120 116 114 97 32 100 97 116 97 32 108 111 99 107 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0↩
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0↩
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0↩
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 33 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 248 ↩
      3 0 0 248 31 0 0 2 0 0 0 0 0 0 0 0 0 0 0 48 0 0 0 254 255 255 255 0 0 0 0 2 0 4 0 0 0 ↩
      0 0 data 16 0x40002 0x4 0x80002 0x4 0xf0002 0x4 0x0 0x0 0x0 0x0 0x0 0x0 0x9 0x2 0x0 0↩
      x0 oops 0 methods 0
7 ciMethodData java/lang/Long toString (J)Ljava/lang/String; 1 3 orig 280 152 101 83 95 3 ↩
      127 0 0 248 47 139 72 3 127 0 0 40 2 0 0 136 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0↩
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 77 68 79 ↩
      32 101 120 116 114 97 32 100 97 116 97 32 108 111 99 107 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0↩
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0↩
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0↩
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 17 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 248 ↩
      3 0 0 248 31 0 0 2 0 0 0 0 0 0 0 0 0 0 0 200 0 0 0 254 255 255 255 0 0 0 0 2 0 5 0 0 0↩
       0 0 data 35 0x50002 0x2 0xd0007 0x2 0x30 0x0 0x170002 0x0 0x1e0007 0x2 0x48 0x0 0↩
      x230002 0x0 0x280003 0x0 0x28 0x2c0002 0x2 0x370002 0x2 0x400002 0x2 0x480002 0x2 0x0 ↩
      0x0 0x0 0x0 0x0 0x0 0x9 0x2 0x0 0x0 oops 0 methods 0
8 ciMethodData java/lang/Long valueOf (J)Ljava/lang/Long; 1 7 orig 280 152 101 83 95 3 127 0↩
       0 232 61 139 72 3 127 0 0 224 1 0 0 80 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ↩
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 77 68 79 32 ↩
      101 120 116 114 97 32 100 97 116 97 32 108 111 99 107 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ↩
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ↩
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ↩
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 33 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 248 3 0↩
       0 248 31 0 0 2 0 0 0 0 0 0 0 0 0 0 0 128 0 0 0 254 255 255 255 0 0 0 0 2 0 5 0 0 0 0 ↩
      0 data 26 0x50002 0x4 0xd0007 0x0 0x50 0x4 0x150007 0x4 0x30 0x0 0x270002 0x0 0x300002↩
       0x4 0x380002 0x4 0x0 0x0 0x0 0x0 0x0 0x0 0x9 0x2 0x0 0x0 oops 0 methods 0
9 ciMethodData java/lang/Long longValue ()J 1 3 orig 280 152 101 83 95 3 127 0 0 176 67 139 ↩
      72 3 127 0 0 120 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0↩
       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 77 68 79 32 101 120 116 114 ↩
      97 32 100 97 116 97 32 108 111 99 107 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ↩
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ↩
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ↩
      0 0 0 0 0 0 0 1 0 0 0 17 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 248 3 0 0 248 31 0 0 2 ↩
      0 0 0 0 0 0 0 0 0 0 0 32 0 0 0 254 255 255 255 0 0 0 0 2 0 5 0 0 0 0 0 data 13 0x50002↩
       0x2 0x110002 0x2 0x0 0x0 0x0 0x0 0x0 0x0 0x9 0x1 0x0 oops 0 methods 0
10 ciMethod NoCompile method1 (D)D 17 80000001 4 0 -1
11 ciMethodData NoCompile method1 (D)D 2 30002544 orig 280 152 101 83 95 3 127 0 0 200 8 192 ↩
      72 3 127 0 0 40 5 0 0 112 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ↩
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 77 68 79 32 101 120 116 114 ↩
      97 32 100 97 116 97 32 108 111 99 107 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ↩
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ↩
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ↩
```

```
    0 0 0 0 0 0 0 128 150 152 0 17 0 0 0 193 170 137 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 56 ↩
    0 0 0 2 0 0 0 1 0 12 0 2 0 0 0 200 3 0 0 254 255 255 255 0 0 0 0 2 0 4 0 0 0 0 0 data ↩
    131 0x40002 0x2 0xa0007 0x0 0x130 0x2 0x140002 0x2 0x190005 0x0 0x7f03580bbc90 0x2 0x0↩
     0x0 0x1d0002 0x2 0x200005 0x0 0x7f03580bbc90 0x2 0x0 0x0 0x250005 0x0 0x7f03580bbc90 ↩
    0x2 0x0 0x0 0x280005 0x0 0x7f03580bbc90 0x2 0x0 0x0 0x2b0005 0x0 0x7f031402b870 0x2 0↩
    x0 0x0 0x2e0002 0x2 0x390007 0x1 0x38 0x989edd 0x450003 0x989edc 0xfffffffffffffe0 0↩
    x480002 0x1 0x500007 0x0 0x1a0 0x1 0x5a0002 0x1 0x5f0005 0x1 0x0 0x0 0x0 0x0 0x6a0002 ↩
    0x1 0x6d0005 0x1 0x0 0x0 0x0 0x0 0x720005 0x1 0x0 0x0 0x0 0x0 0x760002 0x1 0x790005 0↩
    x1 0x0 0x0 0x0 0x0 0x7e0005 0x1 0x0 0x0 0x0 0x0 0x810005 0x1 0x0 0x0 0x0 0x0 0x840005 ↩
    0x0 0x7f031402b870 0x1 0x0 0x0 0x8c0002 0x1 0x8f0005 0x1 0x0 0x0 0x0 0x0 0x940002 0x1 ↩
    0x970005 0x0 0x7f031402b920 0x1 0x0 0x0 0xa20002 0x1 0x0 0x0 0x0 0x0 0x0 0x0 0x9 0x2 0↩
    x0 0x0 oops 7 10 java/lang/StringBuilder 18 java/lang/StringBuilder 24 java/lang/↩
    StringBuilder 30 java/lang/StringBuilder 36 java/io/PrintStream 99 java/io/PrintStream↩
     115 java/util/ArrayList methods 0
12 compile NoCompile method1 (D)D -1 4 inline 24 0 -1 NoCompile method1 (D)D 1 20 java/lang/↩
    StringBuilder <init> ()V 2 10 java/lang/AbstractStringBuilder <init> (I)V 3 8 java/↩
    lang/Object <init> ()V 1 25 java/lang/StringBuilder append (Ljava/lang/String;)Ljava/↩
    lang/StringBuilder; 1 32 java/lang/StringBuilder append (Ljava/lang/String;)Ljava/lang↩
    /StringBuilder; 1 37 java/lang/StringBuilder append (Ljava/lang/String;)Ljava/lang/↩
    StringBuilder; 1 90 java/lang/StringBuilder <init> ()V 2 10 java/lang/↩
    AbstractStringBuilder <init> (I)V 3 8 java/lang/Object <init> ()V 1 95 java/lang/↩
    StringBuilder append (Ljava/lang/String;)Ljava/lang/StringBuilder; 1 109 java/lang/↩
    StringBuilder append (Ljava/lang/String;)Ljava/lang/StringBuilder; 1 114 java/lang/↩
    StringBuilder append (Ljava/lang/String;)Ljava/lang/StringBuilder; 1 121 java/lang/↩
    StringBuilder append (Ljava/lang/String;)Ljava/lang/StringBuilder; 1 126 java/lang/↩
    StringBuilder append (Ljava/lang/String;)Ljava/lang/StringBuilder; 1 140 java/lang/↩
    Long valueOf (J)Ljava/lang/Long; 2 48 java/lang/Long <init> (J)V 3 9 java/lang/Number ↩
    <init> ()V 4 7 java/lang/Object <init> ()V 1 143 java/lang/Long longValue ()J 1 148 ↩
    java/lang/Long valueOf (J)Ljava/lang/Long; 2 48 java/lang/Long <init> (J)V 3 9 java/↩
    lang/Number <init> ()V 4 7 java/lang/Object <init> ()V
```

A.3 Code changes

A.4 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

Table A.1: Code lines changed, inserted, deleted, modified and unchanged

class (in src/share/vm/)	changed	inserted	deleted	modified	unchg.
ci/c1_Compilation.cpp	6	6	0	0	708
ci/ciClassList.hpp	2	2	0	0	121
ci/ciEnv.cpp	58	56	0	2	1283
ci/ciEnv.hpp	5	5	0	0	469
ci/ciMethod.cpp	4	4	0	0	1480
ci/ciMethod.hpp	1	1	0	0	350
ci/ciMethodData.cpp	4	4	0	0	797
ci/ciMethodData.hpp	1	1	0	0	595
compiler/compileBroker.cpp	78	77	1	0	2401
compiler/compileBroker.hpp	2	2	0	0	478
interpreter/invocationCounter.hpp	2	2	0	0	156
oops/instanceKlass.hpp	2	2	0	0	1383
oops/methodCounters.cpp	14	13	1	0	74
oops/methodCounters.hpp	4	4	0	0	208
oops/methodData.cpp	9	9	0	0	1686
opto/compile.cpp	9	9	0	0	4418
prims/jvmtiExport.hpp	2	2	0	0	542
runtime/advancedThresholdPolicy.cpp	5	5	0	0	537
runtime/deoptimization.cpp	15	15	0	0	2043
runtime/deoptimization.hpp	7	7	0	0	407
runtime/globals.hpp	41	38	0	3	3967
runtime/java.cpp	8	8	0	0	744
runtime/simpleThresholdPolicy.inline.hpp	51	49	0	2	80
/runtime/thread.cpp	7	7	0	0	4585
/ci/ciCacheProfiles.cpp	801	801	0	0	0
/ci/ciCacheProfiles.hpp	337	337	0	0	0
/ci/ciCacheProfilesBroker.cpp	292	292	0	0	0
/ci/ciCacheProfilesBroker.hpp	79	79	0	0	0

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.5 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*

A.6 Additional graphs

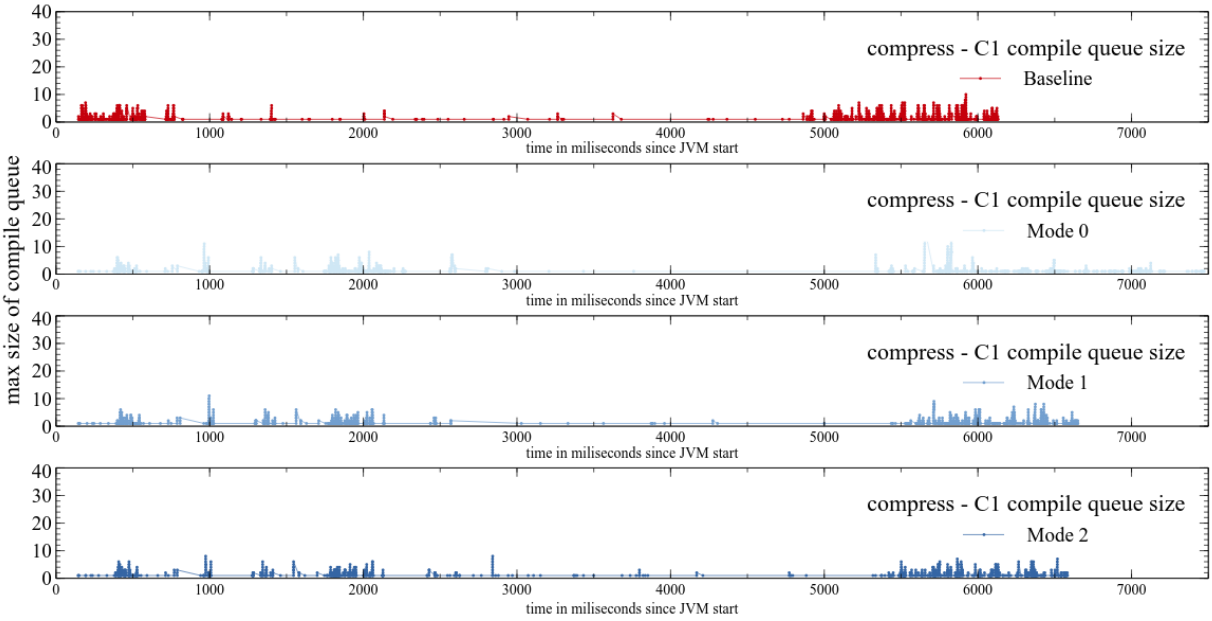

Figure A.1: C1 Compile queue size over time SPECjvm compress benchmark

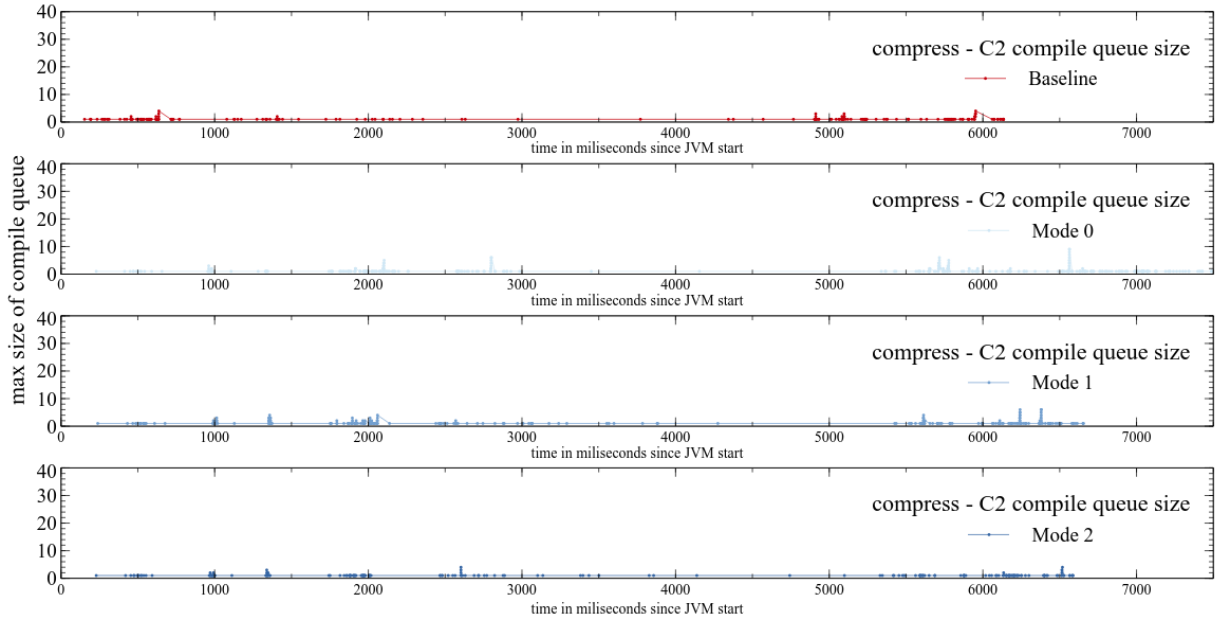

Figure A.2: C2 Compile queue size over time SPECjvm compress benchmark

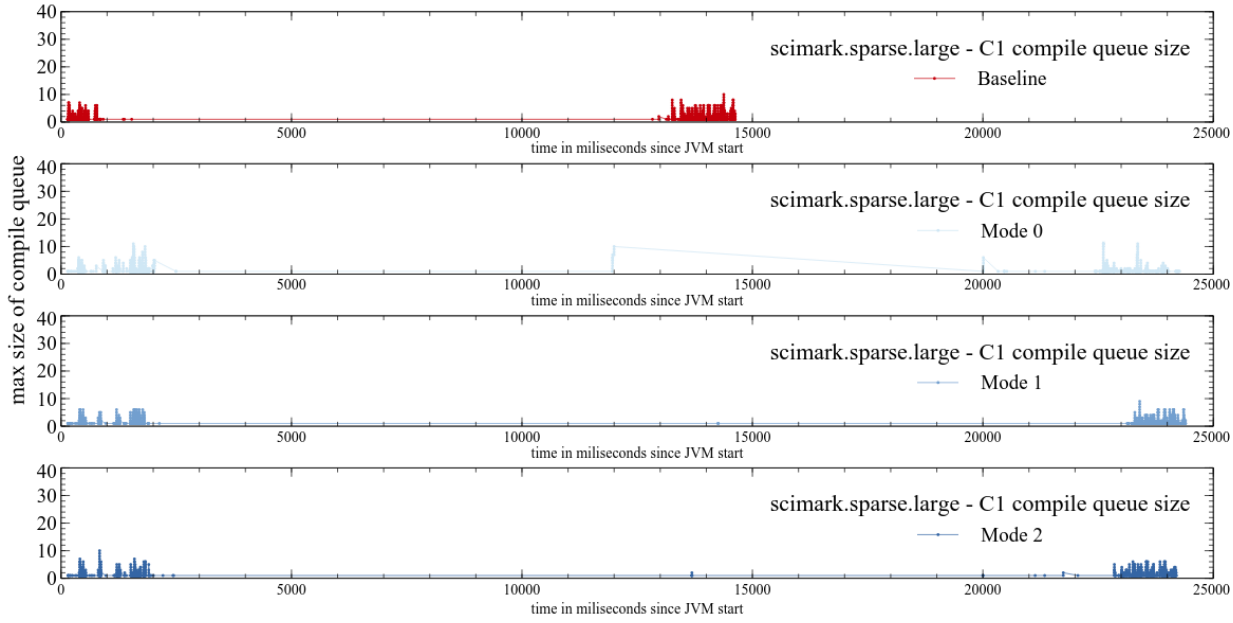

Figure A.3: C1 Compile queue size over time SPECjvm scimark.sparse.large benchmark

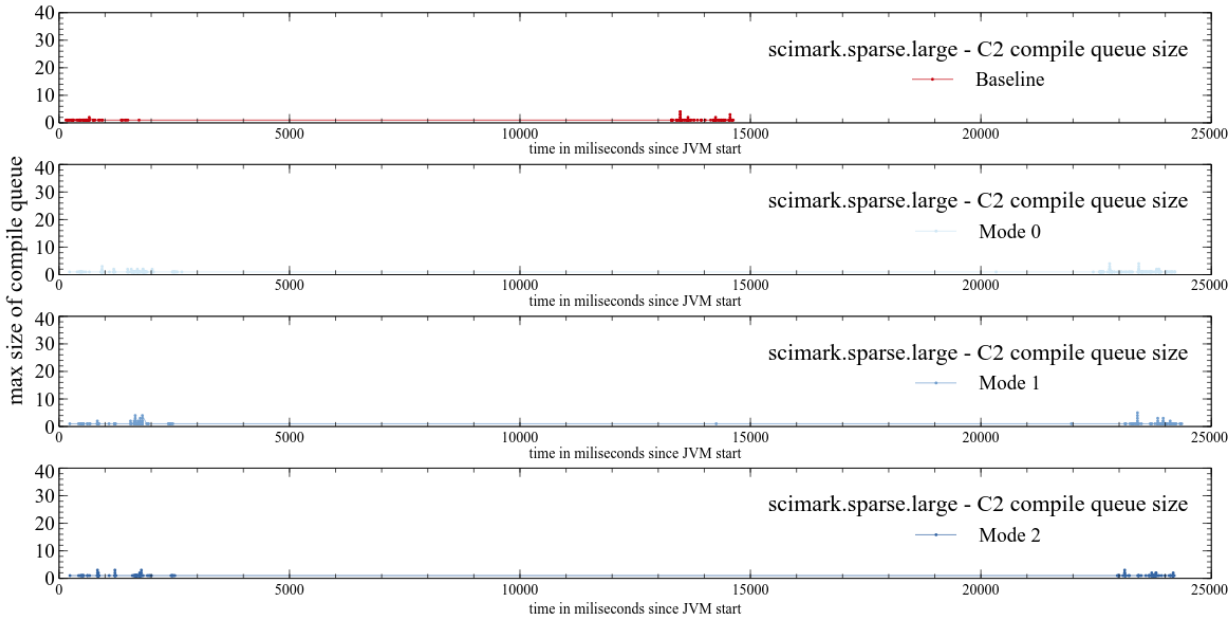

Figure A.4: C2 Compile queue size over time SPECjvm scimark.sparse.large benchmark