Bachelor Thesis

Profile Caching for the Java Virtual Machine

Marcel Mohler
ETH Zurich

Zoltán Majó
Tobias Hartmann
Responsible engineers

Prof. Thomas R. Gross
Laboratory for Software Technology
ETH Zurich

August 2015

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

LST
Laboratory for Software Technology

Introduction

Virtual machines (VMs) like the Java Virtual Machine (JVM) are used as the execution environment of choice for many modern programming languages. VMs interpret a suitable intermediate language (e.g., Java Byte Code for the JVM) and provide the runtime system for application programs. VMs usually include a garbage collector, a thread scheduler, and interfaces to the host operating system. As interpretation of intermediate code is time-consuming, VMs usually include a *Just-in-time* (JIT) compiler that translates frequently-executed functions or methods to *native* machine code.

The JIT compiler executes in parallel to a program's interpretation by the VM and, as a result, compilation speed is a critical issue in the design of a JIT compiler. Unfortunately, it is difficult to design a compiler such that the compiler produces good or excellent code while limiting the resource demands of this compiler. The compiler requires storage, CPU cycles and even on a multi-core processor, compilation may slow down the execution of the application program.

Consequently, most VMs adopt a multi-tier compilation system. At program startup, all methods are interpreted by the virtual machine (execution at Tier 0). The interpreter gathers execution statistics called *profiles* and if a method is determined to be executed frequently, this method is then compiled by the Tier 1 compiler. Methods compiled to Tier 1 are then profiled further and based on these profiling information, some methods are eventually compiled at higher tiers. One of the drawbacks of this setup is that for all programs, all methods start in Tier 0, with interpretation and profiling by the VM. However, for many programs the set of the most used methods does not change from one execution to another and there is no reason to gather profiling information again.

The main idea of this thesis is to cache these profiles from a prior execution to be used in further runs of the same program. Having these *cached profiles* available avoids the JIT compiler to gather the same profiling information again. As well as allow the compiler to use more sophisticated profiles early in program execution and prevent recompilations when more information about the method is available. While this in general should not significantly influence the peak performance of the program, the hope is to decrease the time the JVM needs to achieve it, the so called *warmup*.

This thesis proposes a design and an implementation of a profile caching feature for *HotSpot*, an open source Java virtual machine maintained and distributed by Oracle Corporation as well as a profound performance analysis using state-of-the-art benchmarks.

Contents

1 Overview of HotSpot

This chapter will provide the reader with an overview of the relevant parts of Java HotSpot. The chapter explains the core concepts that are needed to understand the motivation of this thesis and the implementation of the system described in this thesis.

1.1 Tiered compilation

As mentioned in the introduction, virtual machines (VMs) like Java HotSpot feature a multi-tier system when compiling methods during execution. Java VM's typically use Java Bytecode as input, a platform independent intermediate code generated by a Java Compiler like `javac` [9]. The bytecode is meant to be interpreted by the virtual machine or further compiled into platform dependent machine code (e.g., x86 instructions). HotSpot includes one interpreter and two different just-in-time compilers with different profiling levels resulting in a total of 5 different *compilation tiers*. Since in literature and the JVM source code use the *tiers* are also called *compilation levels* they will be used synonymously.

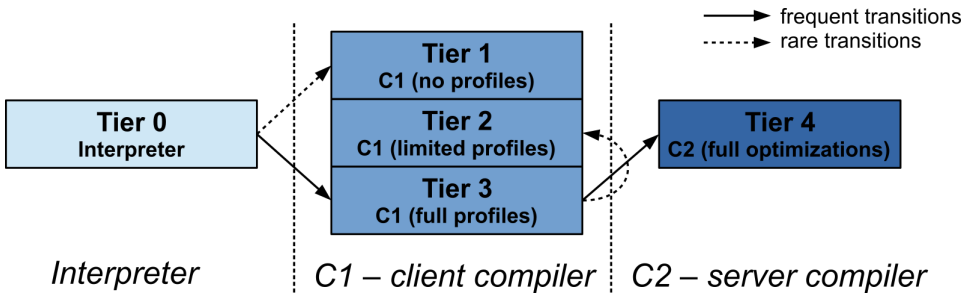

Figure 1.1: Overview of compilation tiers

All methods start being executed at Tier 0, which denotes the interpreter. The interpreter performs a template-based replacement, that is, for each bytecode instruction the interpreter emits a predefined assembly code snippet. During execution, the assembly code is also profiled. The snippets also contain structures to gather method information like execution counters or loop back-branches. a counter exceeds a predefined threshold, the method is considered *hot* and a call back to the JVM is initiated that usually results in a compilation at a higher tier.

The standard behavior of HotSpot is to proceed with Level 3 (Tier 3). The method gets compiled with C1, also referred to as *client* compiler. C1's goal is to provide a fast compilation with a low

Listing 1.1: Example that show potential compilation based on profiling information

```
1  public static void m(int i) {
2      if ( i == 0 ) { // very common branch (a)
3          Math.sin(0);
4      } else { // very uncommon branch (b)
5          Math.sin(pi + i)
6      }
7  }
8  // ------------------------------------------------
9  // If the JVM realizes based on profiling information,
10 // that branch (a) is taken all the time:
11 // compiler could compile the method as follows:
12 // ------------------------------------------------
13 public static void m(int i) {
14     if ( i != 0 ) // very common branch (a)
15         // UNCOMMON TRAP, call to JVM
16     return 0; // result of sin(0)
17 }
```

memory footprint. The client compiler performs simple optimizations such as constant folding, null check elimination, and method inlining based on the information gathered during interpretation. Most of the classes and methods have already been used in the interpreter and allow C1 to inline them to avoid costly invocations. More importantly, information about the program flow and state are gathered. These information contain for example which branches get taken or the final types of dynamically typed objects. For example, if certain branches were not taken during execution further compilations might abstain from compiling these branches and replace them with static code to provide a faster method execution time (see the example in Listing 1.1). The uncommon branch will include an *uncommon trap* which notify the JVM that an assumption does not hold anymore. This then leads to so called *deoptimizations* which are further explained in the separate Section 1.2.

Level 1 and Level 2 include the same optimizations but offer no or less profiling information and are used in special cases. Code compiled at these levels is significantly faster than Level 3 because it needs to execute none or little instructions creating and managing the profiles. Since the profiles generated by C1 are further used in C2, HotSpot is usually interested in creating full profiles and therefore uses Level 3. There are, however, rare instances where a compilation of Level 1 or Level 2 is triggered. For example, if enough profiles are available and a method can not be compiled by a higher tier, HotSpot might recompile the method with Tier 2 to get faster code until the higher tier compiler is available again. A compiler can become unavailable if its compilation queue exceeds a certain threshold.

More information about C1 can be found in [11] and [6].

Eventually, when further compile thresholds are exceeded, the JVM further compiles the method with C2, also known as the *server* compiler. The server compiler uses the profiles gathered in Tier 0 and Tier 3 and produces highly optimized code. C2 includes far more and more complex optimizations like loop unrolling, common subexpression elimination and elimination of range and

null checks. It performs optimistic method inlining, for example by converting some virtual calls to static calls. It relies heavily on the profiling information and richer profiles allow the compiler to use more and better optimizations. While the code quality of C2 is a lot better than C1 this comes at the cost of compile time. Since a C2 compilation includes A more detailed look at the server compiler can be found in [10]. Figure 1.1 gives a short overview as well as showing the standard transition.

The naming scheme *client/server* is due to historical reasons when tiered compilation was not available and users had to choose the JIT compiler via a HotSpot command line flag. The *client* compiler was meant to be used for interactive client programs with graphical user interfaces where response time is more important than peak performance. For long running server applications, the highly optimized but slower *server* compiler was the choice suggested.

Tiered compilation was introduced to improve start-up performance of the JVM. Starting with the interpreter results in instantaneous execution (i.e. a method is executed right away as there is no delay caused by the method's compilation). Also, there are always methods that are executed infrequently. In these the compilation overhead can exceed the performance gain that results from having a compiled version of the method. C1 allows the JVM to have optimized code available early on. That code can be used to create a richer profile which are then be used by the C2 compiler later on. Ideally this profile already contains most of the program flow and the assumptions made by C2 hold. If that is not the case the JVM might need to go back, gather more profiles and compile the method again. In this case, being able to do quick compilations with C1 decreases the amount of C2 recompilations which are even more costly.

1.2 Deoptimizations

Ideally a method is compiled with as much profiling information as possible. For example, since the profiling information are usually gathered in Levels 0 and 3, it can happen that a method compiled by C2 wants to execute a branch it never used before (again see Figure 1.1). In this case the information about this branch are not available in the profile and therefore have not been compiled into the C2-compiled code. This is done to allow further even more optimistic optimization and to keep the compiled code smaller. So instead, the compiler places an uncommon trap at unused branches or unloaded classes which will get triggered in case they actually get used at a later time in execution.

The JVM then stops execution of that method and returns the control back to the interpreter. This process is called *deoptimization* and considered very expensive. The previous interpreter state has to be restored and the method will be executed using the slow interpreter. Eventually the method might be recompiled with the newly gained information.

1.3 On-Stack Replacement

Since the JVM does not only count method invocations but also loop back branches (see also Section 1.4) it can happen that a method is compiled while it is still running and the compiled method is ready before the method has finished. Instead of waiting for the next method invocation, HotSpot can replace the method directly on the program stack. The JVM sets up a new stack frame for the compiled method which replaces the interpreters stack frame and execution will continue using the native method.

This process is called *on-stack replacement* and usually shortened to OSR. The Figure 1.2 presented in a talk by T. Rodriguez and K. Russel [11] gives a graphical representation. The benefits of OSR will become more obvious when looking at the first example in Chapter 2.

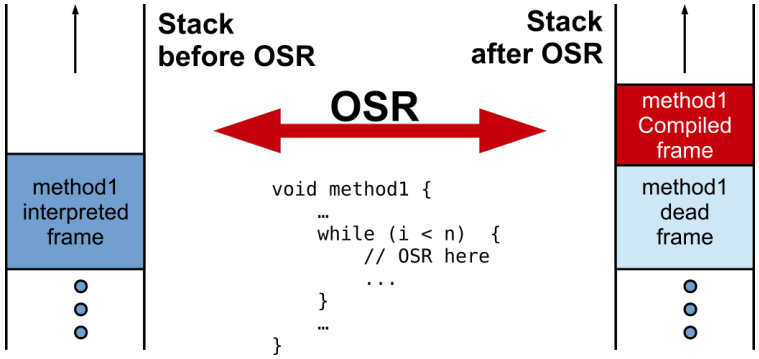

Figure 1.2: Graphical schema of OSR

1.4 Compile thresholds

The transitions between the compilation levels (see Fig. 1.1) are chosen based on predefined constants called *compile thresholds*. When running an instance of the JVM one can specify them manually or use the ones provided. A list of thresholds and their default values relevant to this thesis are given in Appendix A.1. The standard transitions from Level 0 to Level 3 and Level 3 to Level 4 happen when the following predicate returns true:

$$\begin{aligned} & i > TierXInvocationThreshold * s \\ || \; & (i > TierXMinInvocationThreshold * s \;\&\&\; i + b > TierXCompileThreshold * s) \end{aligned}$$

where X is the next compile level (3 or 4), i the number of method invocations, b the number of backedges and s a scaling coefficient (default = 1). The thresholds are relative and individual for interpreter and compiler.

On-stack replacement uses a simpler predicate:

$$b > TierXBackEdgeThreshold * s$$

Please note that there are further conditions influencing the compilation like the load on the compiler which will not be discussed.

2 Motivation

We continue with presenting two simple example methods that illustrate the motivation for using cached profiles. The examples should provide the reader an indication of how and why cached profiles can be beneficial for the performance of a Java Virtual Machine. We will omit any implementation details as they will be discussed in Chapter 3 in detail.

Ideally, being able to reuse the profiles from previous runs should result in two main advantages:

1. **Lower start-up time of the JVM:** From having information about program execution, the compiler can avoid gathering profiles and compile methods earlier and directly at higher compilation levels.

2. **Fewer Deoptimizations:** Since cache profiles are dumped at the end of a compilation, when using these profiles the compiler can already include all optimizations for all different method executions. The compiled code includes less uncommon traps and therefore fewer deoptimizations occur.

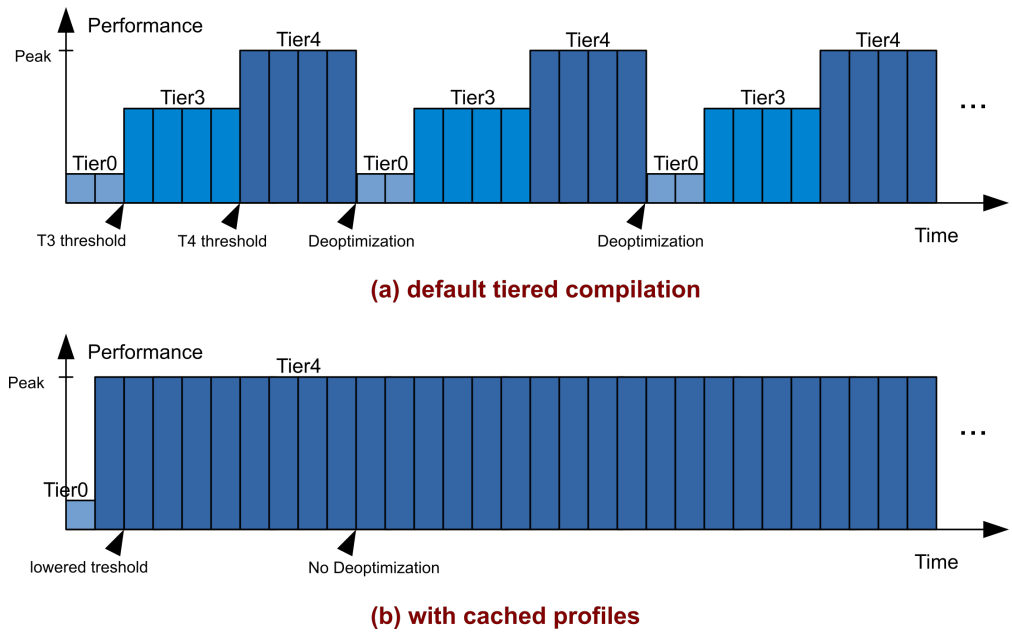

Figure 2.1: Schematic visualization of cached profile benefit

Figure 2.1 gives a schematic visualization of the expected effect on the performance of a single method when using cached profiles compared to the current state without such a system and standard tiered compilation. Each blue bar corresponds to an invocation of the method. Higher bars mean higher compilation levels and therefore higher performance. The x-axis represents time since the start of the JVM. The figure shows the ideal case and abstracts away many details and other possible cases. However, it provides a good visualization for the examples provided in this chapter. A more detailed performance analysis, also considering possible performance regressions is done in Chapter 4.

We are using my implementation described in Chapter 3 in CachedProfileMode 0 (see 3.4.1) built into openJDK 1.9.0. All measurements in this chapter are done on a Dual-Core machine running at 2 GHz with 8GB of RAM. To measure the method invocation time we use hprof [8] and the average of 10 runs. The evaluation process has been automated using a couple of python scripts. The error bars show the 95% confidence interval.

2.1 Example 1: Benefit of early compilation

For Example 1, on-stack replacement (OSR) has been disabled to keep the system simple and easy to understand.
Example 1 is a simple class that invokes a method one hundred times. The method consists of a long running loop. The source code is shown in Listing 2.1. Since OSR is disabled and a compilation to level 3 is triggered after 200 invocations this method never leaves the interpreter. We call this run the *baseline*. To show the influence of cached profiles we use a compiler flag to lower the compile threshold explicitly and, using the functionality written for this thesis, tell HotSpot to cache the profile. In a next execution we use these profiles and achieve a significantly lower time spend executing the cached method as one can see in Figure 2.2. This increase comes mainly from the fact that having a cached profile available allows the JVM to compile highly optimized code for hot methods earlier (at a lower threshold) since there is no need to gather the profiling information first.

Since the example is rather simple neither the baseline nor the profile usage run trigger any deoptimizations. This makes sense because after the first invocation, all the code paths of the method have been taken already and are therefore known to the interpreter and saved in the profile.

When OSR is enabled, the performance difference between using cached profiles and not using them vanishes. That happens because HotSpot realizes the hotness of the method and the simplicity of the method allows the JIT compiler to produce optimized code already. The interpreted version is replaced on the stack by the compiled version during he first method invocation. This example, although rather artificial, shows the influence of early compilation with OSR disabled.

Listing 2.1: Simple method that does not get compiled

```
1  class NoCompile {
2      public static void main() {
3        double result = 0.0;
4        for(int c = 0; c < 100; c++) {
5          result = method1(result);
6        }
7      }
8      public static double method1(double count) {
9          for(int k = 0; k < 10000000; k++) {
10             count = count + 50000;
11         }
12         return count;
13     }
14 }
```

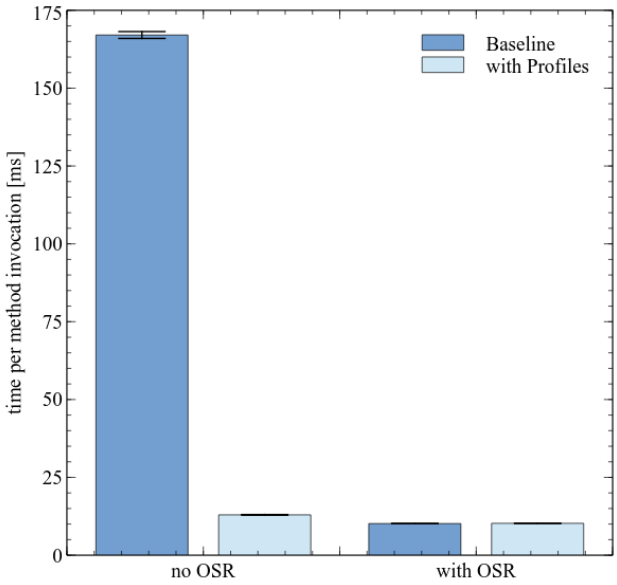

Figure 2.2: NoCompile.method1 - per method invocation time

2.2 Example 2: Benefit of fewer deoptimizations

OSR is one of the core features of HotSpot to improve startup performance of a JVM and disabling that does not give us any practical results. We came up with a second more complex example sketched in Listing 2.2, that demonstrates the influence of cached profiles without disabling any HotSpot functionalities.

The idea is to create a method that takes a different, long running branch on each of its method invocations. Each branch has been constructed in a way that it will trigger an OSR compilation. When compiling this method during its first iteration only the first branch will be included in the compiled code. The same will happen for each of the 100 method invocations. As one can see in Figure 2.3 the baseline indeed averages at around 134 deoptimizations and a time per method

Listing 2.2: Simple method that causes many deoptimizations

```
class ManyDeopts {
    public static void main() {
      double result = 0.0;
      for(int c = 0; c < 100; c++) {
        result = method1(result);
      }
    }
    public static long method1(long count) {
        for(int k = 0l; k < 100000001; k++) {
            if (count < 100000001) {
                count = count + 1;
            } else if (count < 300000001) {
                count = count + 2;
                .
                .
                .
            } else if (count < 505000000001) {
               count = count + 100;
            }
            count = count + 50000;
        }
        return count;
    }
}
```

invocation of 186 ms.

Now we use a regular execution to dump the profiles and then use these profiles. So theoretically the profiles dumped after a full execution should include knowledge of all branches and therefore the compiled method using these profiles should not run into any deoptimizations. As one can see in Figure 2.3 this is indeed the case. When using the cached profiles no more deoptimizations occur and because less time is spent profiling and compiling the methods the per method execution time is even significantly faster with averaging at 169 ms now.

2.3 Similar systems

In commercially available JVMs the idea of caching profiles is not new. The JVM developed and sold by Azul Systems® called Zing® [2] already offers a similar functionality. Zing® includes a feature set they call ReadyNow!™ [1] which aims to increase startup performance of Java applications. Their system has been designed with financial markets in mind and to overcome the issue of slow performance in the beginning and performance drops during execution.

Azul Systems clients reported that their production code usually experiences a significant performance decrease as soon as the market goes live and the clients start trading. The reasons are deoptimizations, that occur for example due to uncommon branch paths being taken or yet unused methods are invoked. In the past Azul Systems' clients used techniques to warm up the JVM,

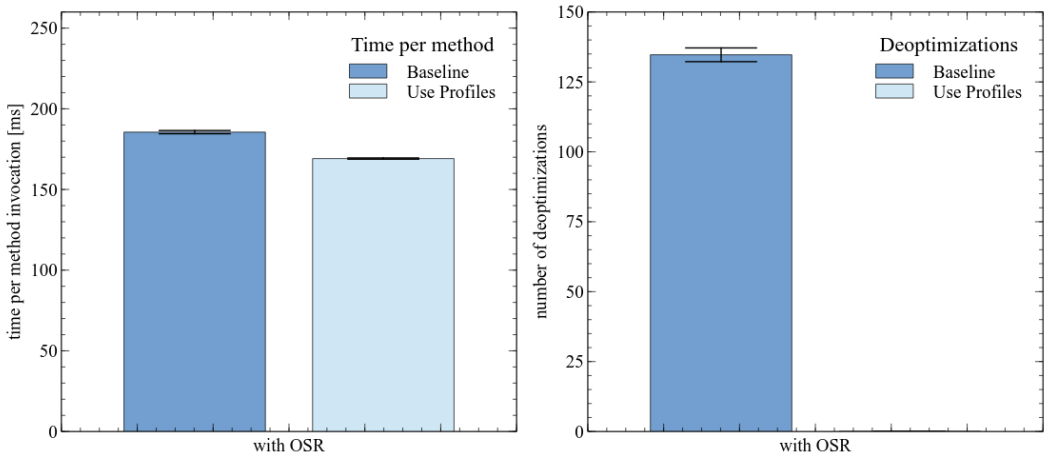

Figure 2.3: ManyDeopts.method1 - per method invocation time and deoptimization count

for example doing fake trades prior to market opening. However this does not solve the problem sufficiently well, since the JVM optimizes for these fake trades and still runs into deoptimizations once actual trades happen, because the code includes methods or specific code snippets that differentiate between the fake and the real trades.

ReadyNow!™ is a rich set of improvements how a JVM can overcome this issues. It includes attempts to reduce the number of deoptimizations in general and other not further specified optimizations. As one of the core features Azul Systems® implemented the ability to log optimization statistics and decisions and reuse this logs in future runs. This is similar to the approach presented in this thesis. However they do not record the actual optimization but the learning and the reasons why certain optimizations happen. This gives them the ability to give feedback to the user of the JVM whether or not certain optimizations have been applied. They also provide APIs for developers to interact with the system and allow further fine-grained custom-designed optimizations.

Unfortunately, Azul Systems does not provide any numbers about how their JVM actually improves performance when executing a software application or any analysis where the speedup originates from in detail.

3 Implementation / Design

This chapter describes the implementation of the cached profiles functionality for HotSpot, written as part of this thesis. HotSpot is a vital part of the open source Java Platform implementation, `OpenJDK`, and the source code is available at `http://openjdk.java.net/`.

Most of the code additions are included in two new classes `/share/vm/ci/ciCacheProfiles.cpp` and `/share/vm/ci/ciCacheProfilesBroker.cpp` as well as significant changes to `/share/vm/ci/ciEnv.cpp` and `/share/vm/compiler/compileBroker.cpp`.
The core functionality is located in `/share/vm/ci/ciCacheProfiles.cpp`, a class that takes care of setting up the cached profile data structure as well as providing public methods to check if a method is cached or not. The class `/share/vm/ci/ciCacheProfilesBroker.cpp` is used before a cached method is compiled. It is responsible for setting up the compilation environment, so the JIT compiler can use the cached profiles.

A full list of modified files and the changes can be seen in the webrev at `http://mohlerm.ch/b/webrev.01/` or Appendix A.3. The changes are provided in form of a patch for HotSpot version 1aef080fd28d. In the following, the original version is referred to as *baseline*.

We will describe and explain the functionality and the implementation design decision in the following sections, ordered by their execution order.

3.1 Creating cached profiles

The baseline version of HotSpot already offers a functionality to replay a compilation based on previously saved profiling information. This is mainly used in case the JVM crashes during a JIT compilation to replay the compilation process and allow the JVM developer to further investigate the cause of this incident. Apart from this automatic process, there exists the possibility to invoke the profile saving manually by specifying the `DumpReplay` compile command option per method.

We introduce a new method option called `DumpProfile` as well as a new compiler flag `-XX:+DumpProfiles` that appends profiling information to a file as soon as a method gets compiled. The first option can be specified as part of the `-XX:CompileCommand` or `-XX:CompileCommandFile` flag and allows the user to select single methods to dump their profile. The second command dumps profiles of all compiled methods. The profile are converted to a string and saved in a simple text

file called *cached_profiles.dat*.

The system will only consider compilations of Level 3 or Level 4. Level 1 and Level 2 are rarely used in practice and do only include none or little profiling information. The user can also restrict the profiles to Level 4 ones by using the compiler flag: `-XX:DumpProfilesMinTier=4`.

The dumped profiling information consists of multiple `ciMethod` entries, `ciMethodData` entries, and one `compile` entry. They are separated by line breaks and keywords to make sure the data can be parsed easily. A shortened example of a cached profile can be found in Appendix A.2. The `ciMethod` entries contain information about the methods used in the compilation and Table 3.1 describes it in more detail. The `ciMethodData` (see Table 3.2) includes all profiling data about the methods itself to be able to redo the compilation. The compile entry saves the bytecode index in case of OSR, the level of the compilation and lists all inlining decisions (Table 3.3).

A method can be compiled multiple times and at different tiers, thus results compilation information for the same method can be dumped multiple times. This is intentional and is taken care of when loading the profiles (see Section 3.2).

Table 3.1: content of ciMethod entry in cached profile

name	**description**
class_name, method_name, signature	used to identify the method
invocation_counter	number of invotations
backedge_counter	numbe of counted backedges
interpreter_invocation_count	number of invocations during interpreter phase
interpreter_throwout_count	how many times method was exited via exception while interpreting
instructions_size_name	rough size of method before inlining

Table 3.2: content of ciMethodData entry in cached profile

name	**description**
class_name, method_name, signature	used to identify the method
state	if data is attached and matured
current_mileage	maturity of the oop when snapshot is taken
orig	snapshot of the original header
data	the actual profiling data
oops	ordinary object pointers, JVM managed pointers to object

Table 3.3: content of compile entry in cached profile

name	**description**
class_name, method_name, signature	used to identify the method
entry_bci	byte code index of method
comp_level	compilation level of record
inline	array of inlining information

3.2 Initializing cached profiles

The information dumped in step 3.1 can now be used in a next run of that particular program. To specify that profiles are available, we introduce a new compiler flag `-XX:+CacheProfiles` that enables the use of previously generated profiles. By default, it reads from a file called *cached_profiles.dat* but a different file can be specified using `-XX:CacheProfilesFile=other_file.dat`.

Before any cached profiles can be used the virtual machine has to parse that file and organize the profiles and compile information in a data structure. This data structure is completely kept in memory during the whole execution of the JVM to avoid multiple disk accesses. The parsing process is invoked during boot up of the JVM, directly after the compileBroker gets initialized. This happens before any methods get executed and blocks the main thread of the JVM until finished.

As mentioned in Section 3.1, the file consists of method information, method profiles, and additional compile information. The parser scans the file once and creates a so called `CompileRecord` for each of the methods that include compilation information in the file. This compile record also includes the list of method information (`ciMethod`) and their profiling information (`ciMethodData`). As mentioned previously, a method's compile information could have been dumped multiple times, which results in multiple `CompileRecords` for the same method. In this case, HotSpot will only keep the `CompileRecords` based on the latest data written to the file but never overwrite an existing higher level profile. Because a profile dumped by the C1 compiler can not be used by the C2 compiler and the other way around, the level of the profile matters as it influences the compile level transitions described in Section 3.4. And since profiling information only grow, the compilation that happened last contains the richest profile and is considered the best. This is based on the fact that the richer the profile, the more information about the method execution is known and influences the compiled version of that method. For example, a profile for a method might include data for all its branches and can therefore help avoid running into uncommon traps and trigger deoptimizations.

The `CompileRecord` as well as the lists of methods information and profiles are implemented as an array located in HotSpot's heap space. They get initialized with a length of 8 and grow when needed. This choice has been done for simplicity and leaves room for further improvements.

3.3 Using cached profiles

The implementation offers three different modes `Mode 0`, `Mode 1`, and `Mode 2`, that differ in the way they use the cached profiles. The following paragraph applies to all three modes and I will discuss the differences of the modes in detail in Section 3.4.

The idea is to modify the compiler to use cached profiles if available and continue as usual otherwise. A simplified graphical overview of the program flow for compiling a method with the changes introduced in this thesis can be found in Figure 3.1.

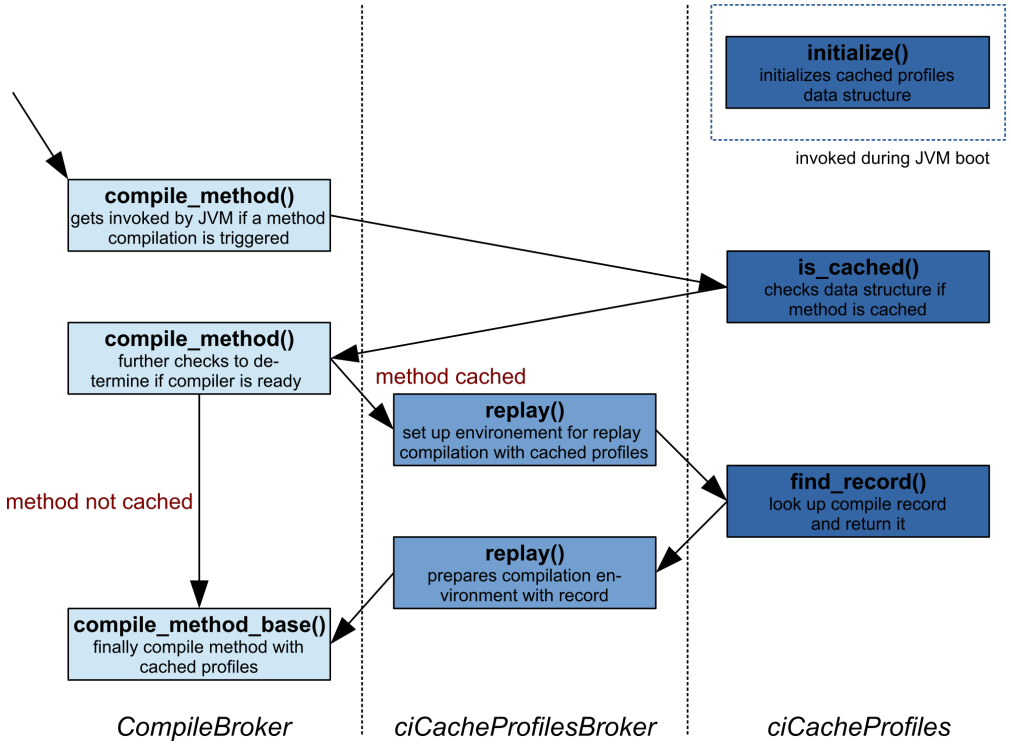

Figure 3.1: program flow for compiling a method

As mentioned before, once compilation thresholds are exceeded a method is scheduled for compilation. This means that the JVM will invoke a method called `compile_method()`, located in the `compileBroker` class. This method tests if certain conditions hold, for example, it checks if the compile queue is not full or if there is already another compilation of that particular method running. We extended this method with a call to `ciCacheProfiles::is_cached(Method* method)` which does a linear scan through the `CompileRecord` array data structure. The method returns either 0 if the method is not cached or returns an integer value, reflecting the compile level, in case a cached profile of this method is available. Because only methods compiled with level 3 or 4 are cached, this call to `is_cached()` only gets executed if the compilation request is also of level 3 or higher.

Depending on the compilation level of the profile, the level of the requested compilation, and the

`CacheProfileMode`, the `compileBroker` then schedules either a compilation using freshly gathered profiles or calls into `ciCacheProfilesBroker` to replay the compilation, based on a cached profile. In contrast to cached cached profiles, fresh profiles describe profiles gathered during the current run of the JVM. Since these decisions are different in each mode, I describe them in detail in the next section. In case the method is not cached, the execution continues like in the baseline version. Otherwise, the `ciCacheProfilesBroker` class then initializes the replay environment and retrieves the compile record from `ciCacheProfiles`. Subsequently, the needed cached profiles get loaded to make sure they are used by the following compilation. `ciCacheProfilesBroker` then returns the execution to the `compileBroker`, which continues with the steps needed to compile the method. Again some constraints are checked (e.g. if there is another compilation of the same method finished in the meantime) and a new compile job is added to the compile queue. Eventually the the method is going to be compiled using the cached profiles.

Since the implementation is only invoked by the static class `compileBroker`, `ciCacheProfiles` and `ciCacheProfilesBroker` are static classes as well. The `compileBroker` is solely called by the JVM main thread, therefore there is no need to make the compileRecord data structure or any of the new implementations thread safe.

3.4 Different usage modes for cached profiles

The implementation of cached profiles offers 3 different modes, which distinguish from each other in the transitions between the compilation tiers. The motivation as well as the advantages and disadvantages of the modes are described in the following three subsections. While `Mode 0` and `Mode 1` are similar except for the compile thresholds, `mode2` differs significantly. Figure 3.2 provides a graphical overview of the differences in the compilation tier transitions of the modes.

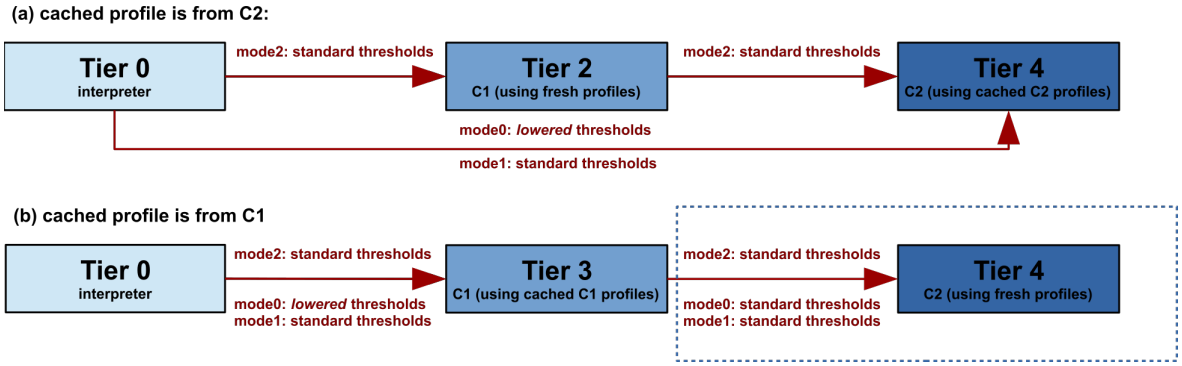

Figure 3.2: Tier transitions of different modes

All three modes have advantages and disadvantages and their performance is evaluated in Chapter 4. Since depending on the methods, different modes might perform best, it is up to the JVM user to decide which mode he or she wants to use. Because `Mode 2` is the most conservative one, which does not modify thresholds it is considered the default mode and used if not further specified.

3.4.1 Compile Thresholds lowered (Mode 0)

The first mode is based on the consideration that a method that has a profile available does not require profiling anymore. Therefore, the compile thresholds (see Section 1.4) of these methods are lowered. If the cached method has a C2 profile, all thresholds are lowered, in case of a C1 profile, only the thresholds that affect the tiers below and equal to Tier 3. This differentiation is done to prevent an increase in early C2 compilations using fresh profiles. Because a lower Tier 4 threshold would mean, that a method which only got compiled at Tier 3 when creation the profile, might now trigger a Tier 4 compilation very early. At this point, the fresh profiles generated by the Tier 3 compiled version are still immature. Using them to compile with Tier 4 results in code that includes only limited profiling information, which causes costly deoptimizations when used.

By default, the thresholds are lowered to 1% of their original values but the threshold scaling can be modified with the JVM parameter: `-XX:CacheProfilesMode0ThresholdScaling=x.xx`. 1% results in the level 3 invocation counter being reduced from 200 to 2. This means that the method will be interpreted once but then directly trigger a compilation on the next invocation.

Since the interpreter also handles class loading, this decision has been made to avoid the need of loading classes in C1 or C2 which was considered out of the scope for this thesis. HotSpot expects classes of standard libraries to be loaded in a very specific order. Moving class loading to C1 or C2 would mess with that order and can therefore not be done without huge changes to the JVM.

In `Mode 0`, the JIT compiler will always use a cached profile for compilations of Level 3 or Level 4 in case there is a cached profile which has been generated by a compiler of the same level. However, if a method to be compiled on Level 3 has a cached profile available for Level 4, the compiler will skip the C1 compilation and immediately compile with C2. In this case, HotSpot directly uses the highly optimized version generated by C2 and in an ideal case the method will need less time to reach peak performance.

However, since the thresholds of all methods with cached profiles get lowered and some of the C1 compilations are promoted to C2 compilations, the C2 compiler is put under heavy load. Especially during startup of a program, where many compilations happen naturally, C2 might not be able to handle all these requests at the same time and the compile queue fills up. This might negatively affect performance and is analyzed in Section 4.4.

3.4.2 Unmodified Compile Thresholds (Mode 1)

`Mode 1` is doing exactly the same as `Mode 0` but does not lower the compilation thresholds of methods with cached profiles. This is done to decrease the load increase on C2 as mentioned in Subsection 3.4.1. Apart from this change `Mode 1` has the same behaviour as `Mode 0`.

3.4.3 Modified C1 stage (Mode 2)

Both modes mentioned before use cached profiles as soon as a compilation of level 3 and 4 are triggered. Since the thresholds for level 3 are smaller than the level 4 thresholds (see Appendix A.1) a method reaching a level 3 threshold could actually trigger a level 4 compilation, if the cached profile is one of level 4. So even if `Mode 1` is used and the thresholds are untouched, C2 might get overloaded since compilations occur earlier.

`Mode 2` has been designed to make as little changes as possible to the tiered compilation and prevent C2 being more used than usual. It does so by keeping the original tiered compilation steps and compilation thresholds and compiles methods with C1 prior to C2. But since there are already profiles available, there is no need to run at Tier 3 to generate full profiles but instead it uses Tier 2. Tier 2 does the same optimizations but offers only limited profiles like method invocation and backbranch counters. They are needed to know when to trigger the C2 compilation and therefore Tier 1 can not be used. By avoiding Tier 3 and using Tier 2 instead methods spend less time in code gathering profiling information and therefore method execution is considered about 30% faster [7]. Eventually, if the Tier 4 thresholds are reached, the method is compiled using C2 and the cached profiles. This still maintains the benefit from having more complete profiles available early but avoids modifying thresholds which could result in a very different load to the compiler.

The above only makes sense if the cached profile is a C2 profile. If only a C1 profile is available, C1 should gather fresh, full profiles since they might be needed in C2 later. HotSpot will then only use the cached profile during the C1 compilation and then use the generated profiles for possible C2 compilations. In theory this transition is considered rare, because if a method has not been compiled with C2 when creating the profile it is unlikely to get compiled with C2 in the future.

To summarize, we expect a performance benefit C1 compiled code by using Level 2 instead of Level 3 and a benefit for C2 compiled code by using more comprehensive profiles.

3.5 Problems

If the profiles generated by multiple runs of the program deviate sharply it is likely that a cached profile does not fit to the current execution. In this case the compiled version would still trigger many deoptimizations and the method could end up having even worse performance since it's going to use the profile over and over again. For each method, the JVM maintains a deoptimization counter. Cache profiles are used if the counter is below a certain limit. If they are above that limit a standard compilation using freshly gathered profiles will be used instead. The limit is 10 to allow a small number of recompilations. This could for example be useful when the method is deoptimized due to classes not being loaded. The value of 10 seems reasonable for all executed measurements.

3.6 Debug output

For debugging and benchmarking purposes four debug flags are implemented, that can be used along with `-XX:+CacheProfiles`.

flag	**description**
-XX:+PrintCacheProfiles	enable command line debug output for cached profiles
-XX:+PrintDeoptimizationCount	prints amount of deoptimizations when the JVM gets shut down
-XX:+PrintDeoptimizationCountVerbose	prints total the amount of deoptimizations on each deoptimization
-XX:+PrintCompileQueueSize	prints the total amount of methods in the compile queue each time a method gets added

4 Performance

This section evaluates the performance of the cached profile implementation using modern benchmark suites. The goal is to provide indicators on the performance influence and try to analyze where this performance influence comes from. Since we try to decrease warmup time and decrease the number of deoptimizations

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 Benchmark 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. We expect the modes to produce different results. The following list suggests reasons for these performance differences:

- Some benchmarks might profit from compiling methods very early and therefore favor Mode 0.
- That could however result in many early compilations that overload the compilation queue resulting in worse performance. In this case Mode 1 will perform better than Mode 0.
- In case a cached profile can not be used (i.e. the limit of 10 deoptimizations was reached) the JVM needs to use freshly generated profiles. In case the thresholds were lowered (Mode 0) this compilation might have happened very early and only very incomplete profiles were available. This effect is less a problem when Mode 1 is used.
- Mode 2 keeps the steps of the original tiered compilation and is considered the most conservative mode. It puts the same load on the compile queue than the baseline version.

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.4.

4.2.1 SPECjvm warmup performance

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 6 to 40 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.

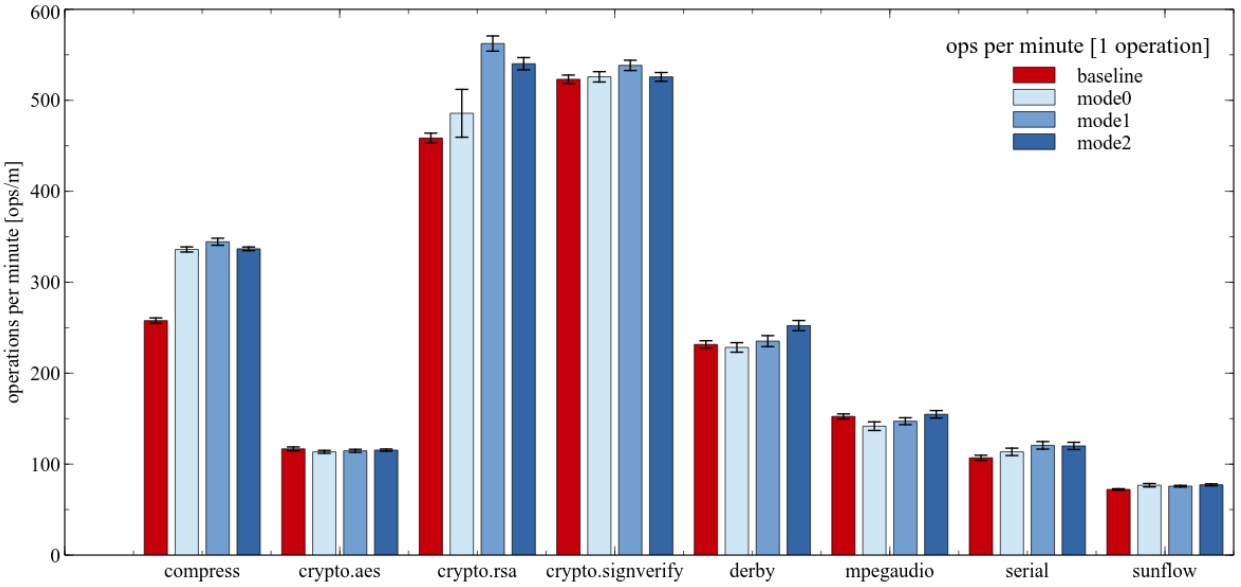

Figure 4.1: SPECjvm benchmarks on all different modes

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 *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. Taking the average of all modes, 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 is 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 looses in certain benchmarks against the others in terms of performance. However, in 12 out of 17 benchmarks at least one of the CacheProfileModes improves performance.

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

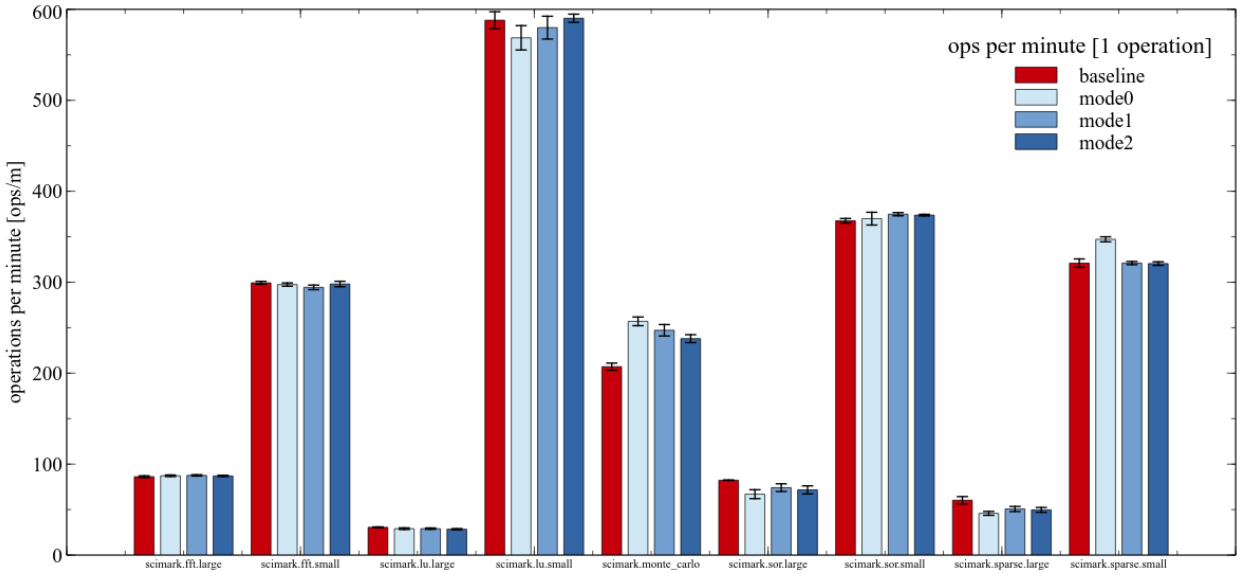

Figure 4.2: SPECjvm scimark benchmarks on all different modes

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 way to run a fixed number of iterations (without modifying the Octane source) we run the Octane benchmarks completely. We still split up the execution in the individual benchmarks to achieve many JVM restarts. The rest of the setup is identical to the SPECjvm run in Section 4.2.1.

The absolute results are shown in Figure 4.4 and a relative comparison with the baseline in Figure 4.5. Compared to SPECjvm the Octane performance is more scattered. The richards benchmark increases by around 50% in Mode 0 while navierstockes decreases by around 25% in Mode 1. In most benchmarks (9 out of 14) Mode 0 performs worst. We assume this is related to the increased load of the compile queue and will therefore take a more detailed look at this in Section 4.4. The performance of the two other modes is better in most benchmarks, but in total only 6 out of 14 benchmarks result in a performance improvement in at least one mode.

4.3 Deoptimizations

We are still eager to figure out where the performance increase and decrease come from. We aim to lower the time needed for warmup by compiling methods earlier and or at lower tiers but also expect to decrease the number of deoptimizations by having more complete profiles early, which ideally results in better compiled code quality. The total number of deoptimizations of the SPECjvm benchmarks is shown in Figure **??** and Figure 4.7. The Octane numbers are drawn in Figure **??**. Again, we also included graphs that show the number of deoptimizations relative to the baseline

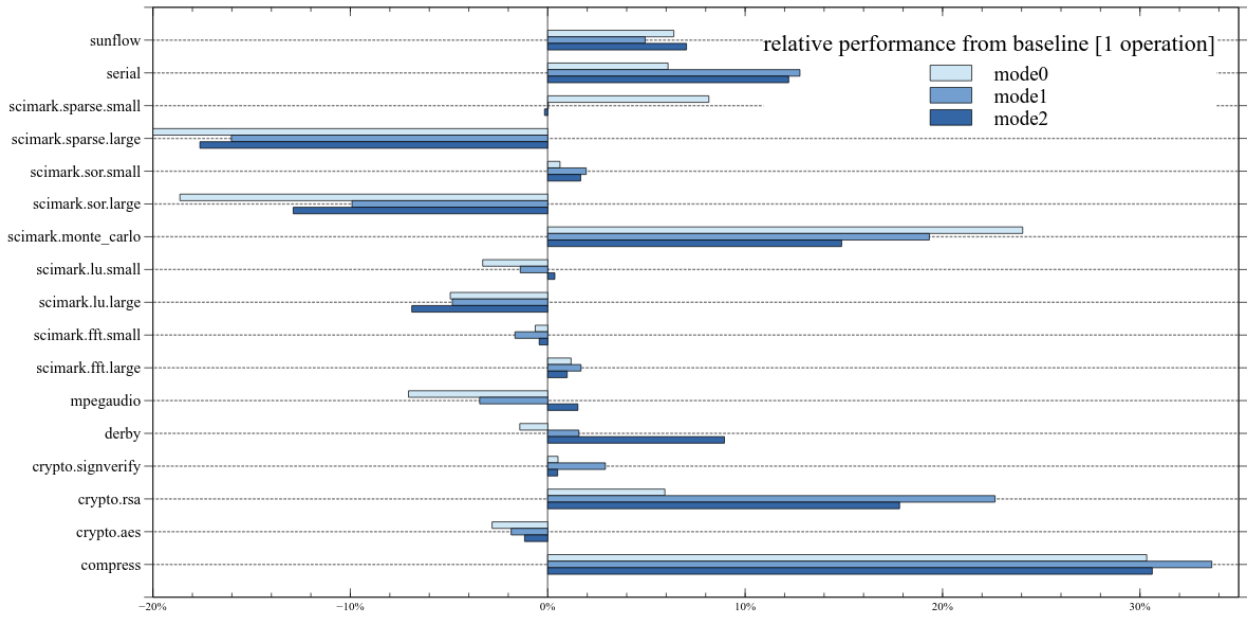

Figure 4.3: Relative performance from baseline for all SPECjvm benchmarks

runs in Figure 4.8 and Figure 4.10.

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 on average increases the number by 30%. Mode 0 also had the worst performance for Octane and we assume the amount of deoptimizations to be one of the reasons for that regression.

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 result on the performance.

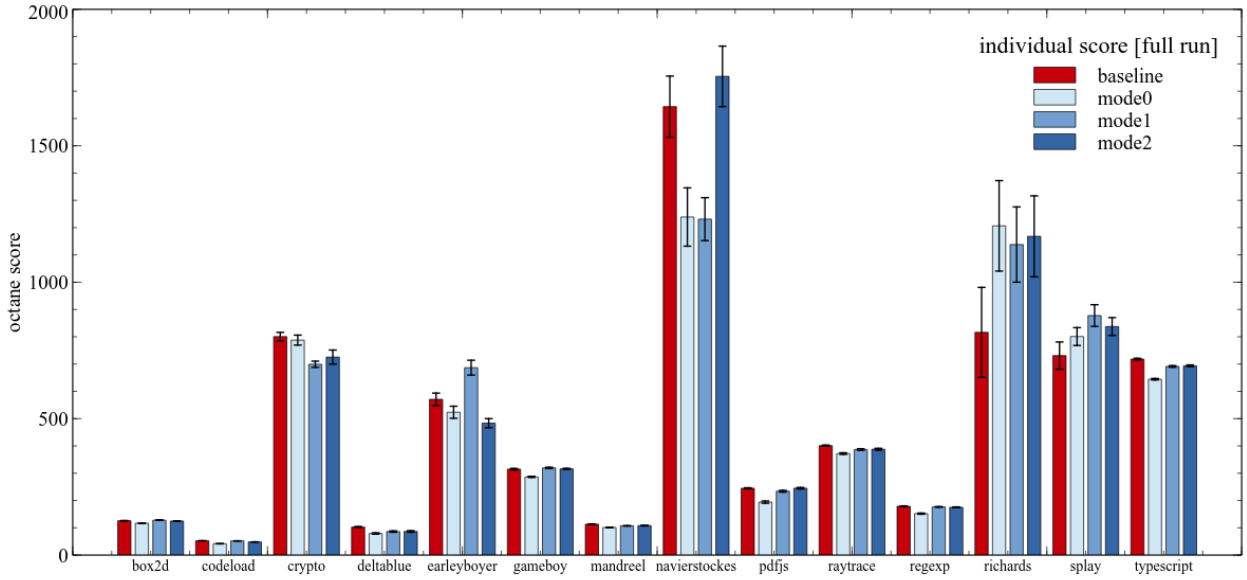

Figure 4.4: Octane benchmarks on all different modes

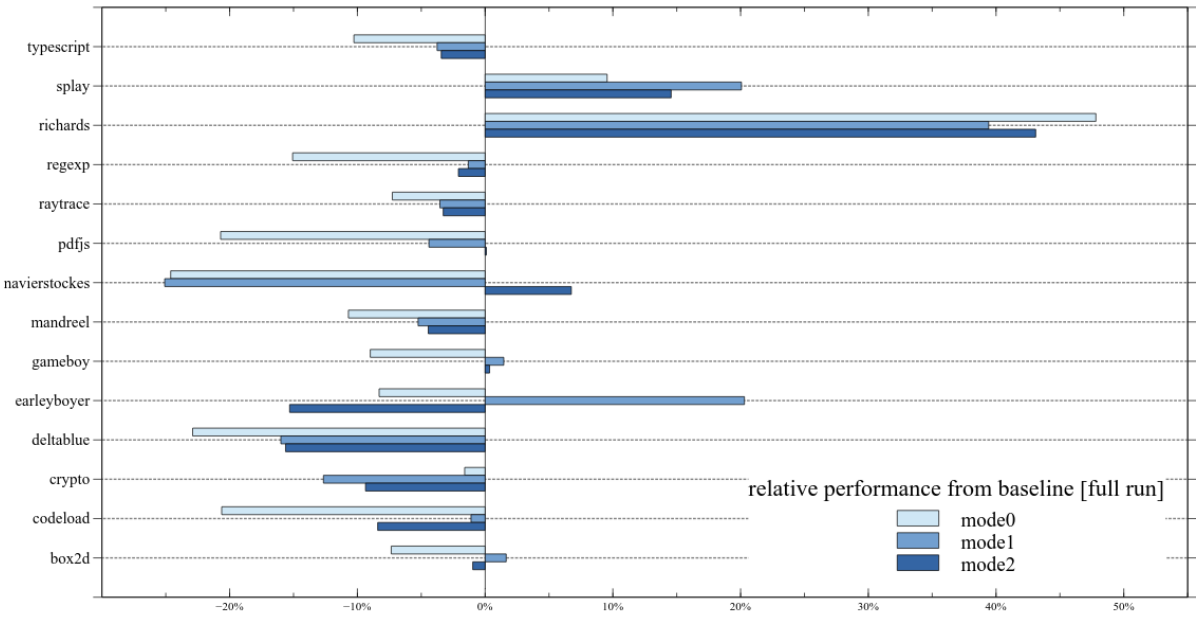

Figure 4.5: Relative performance from baseline for all Octane benchmarks

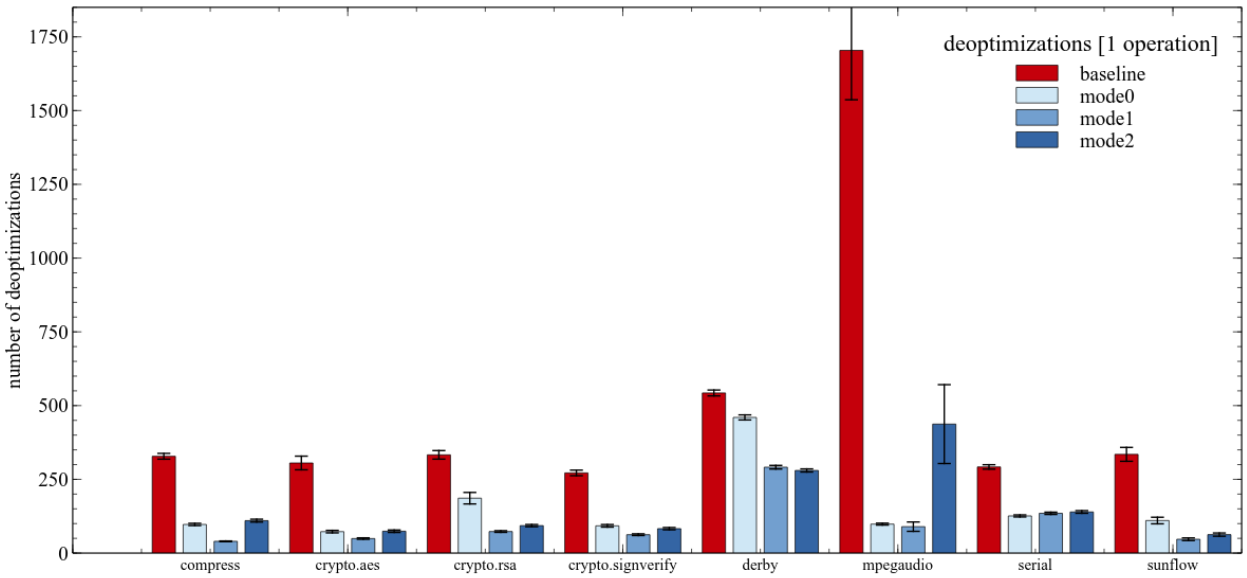

Figure 4.6: SPECjvm deoptimizations of all modes

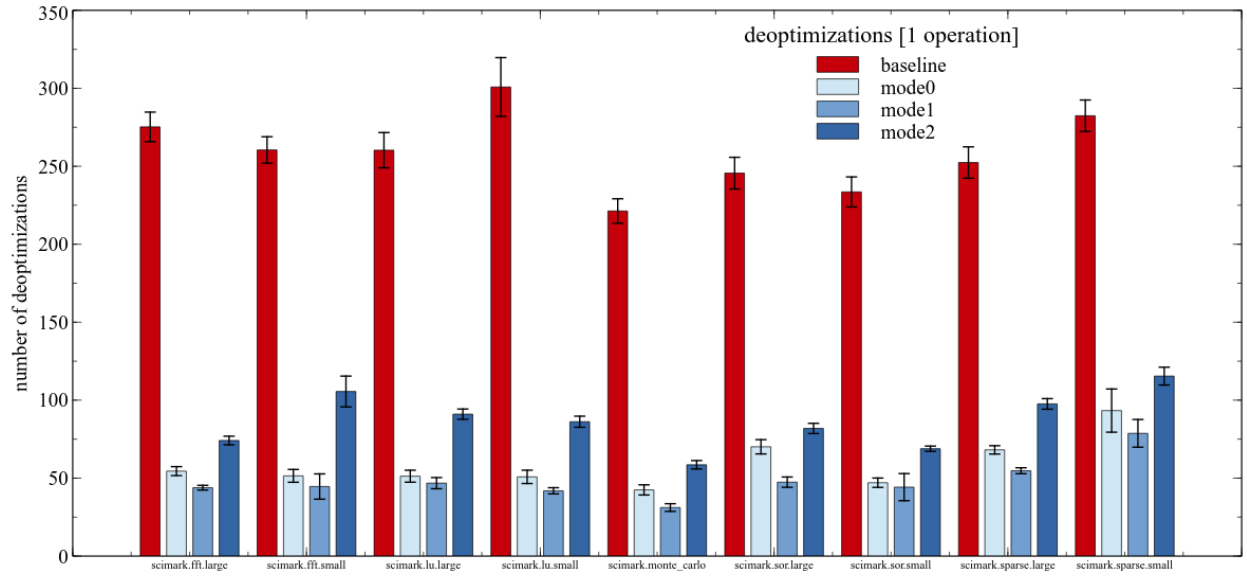

Figure 4.7: SPECjvm scimark deoptimizations of all modes

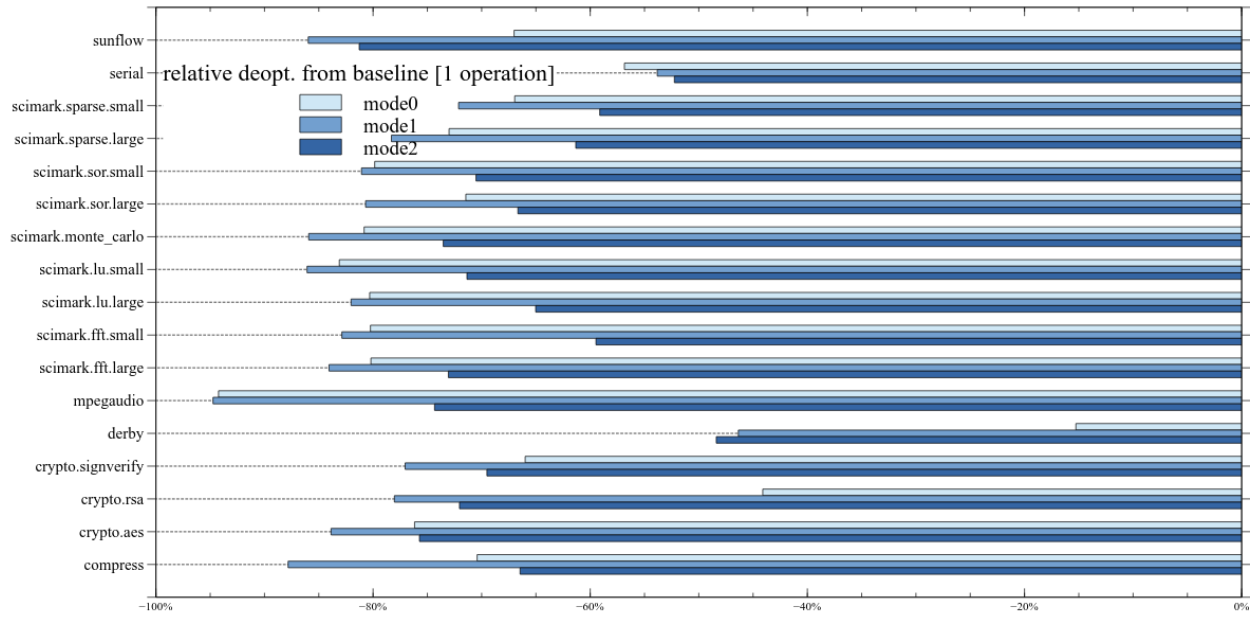

Figure 4.8: Relative deoptimizations from baseline for all SPECjvm benchmarks

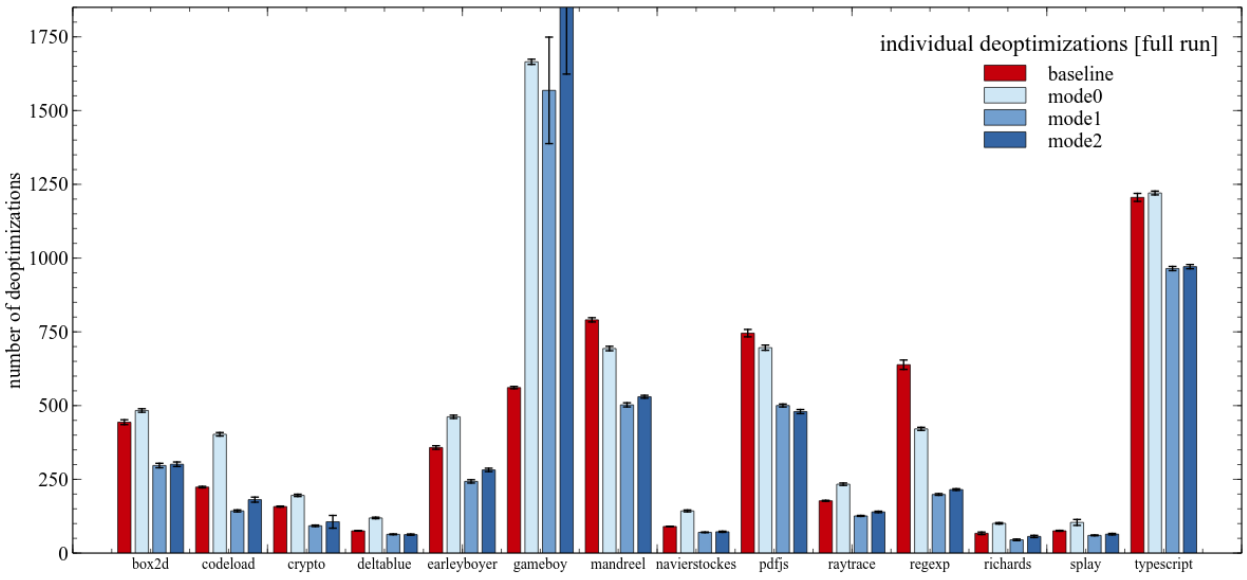

Figure 4.9: Octane deoptimizations of all modes

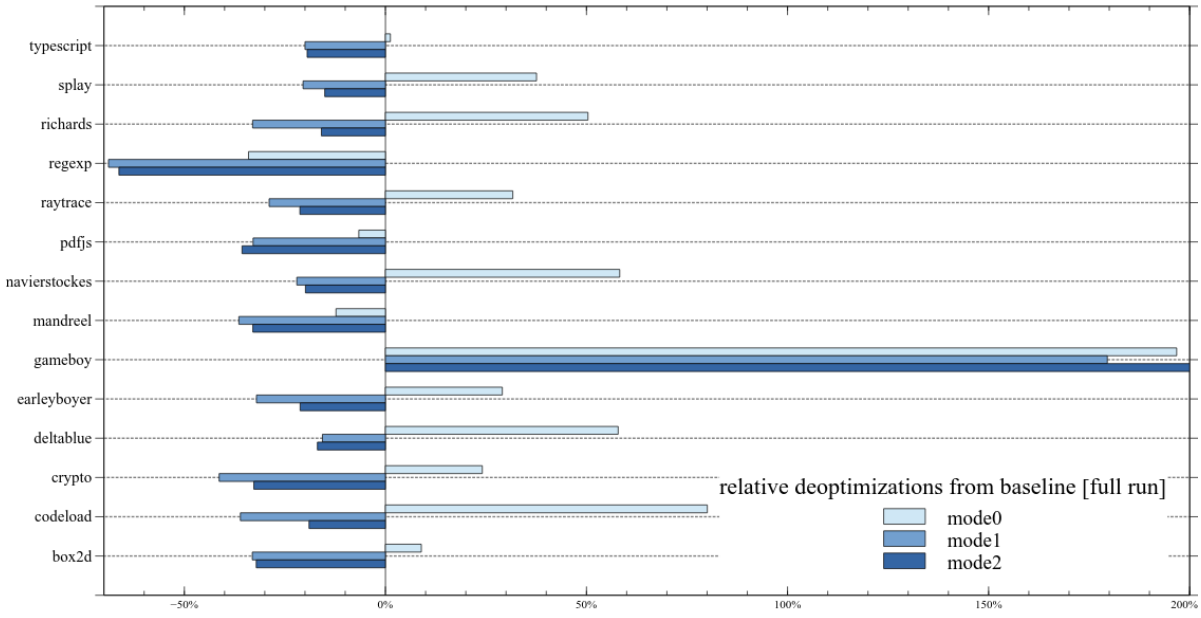

Figure 4.10: Relative deoptimizations from baseline for all Octane benchmarks

4.4 Effect on compile queue

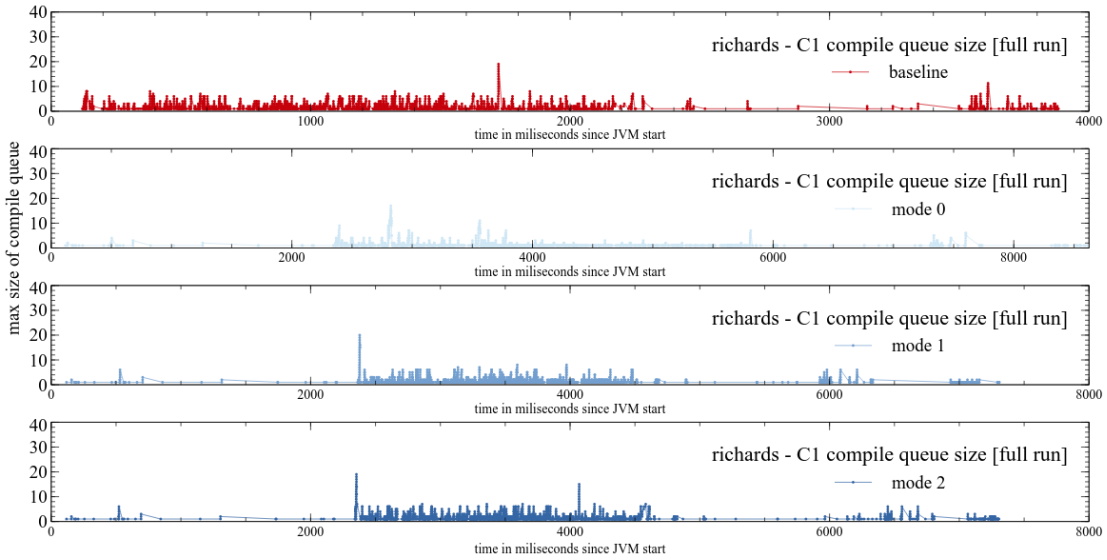

Figure 4.11: C1 Compile queue size over time Octane Richards benchmark

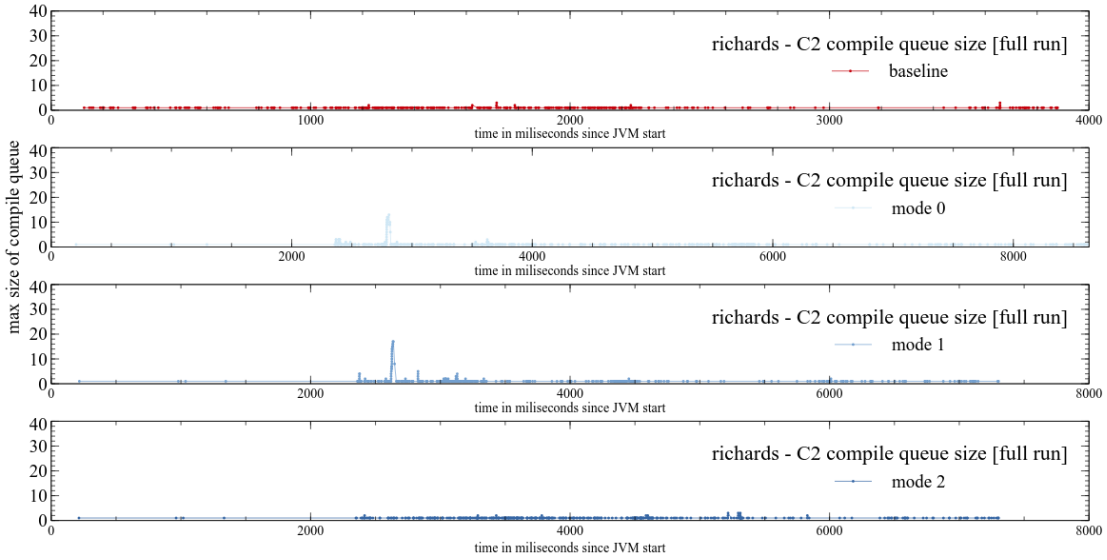

Figure 4.12: C2 Compile queue size over time Octane Richards benchmark

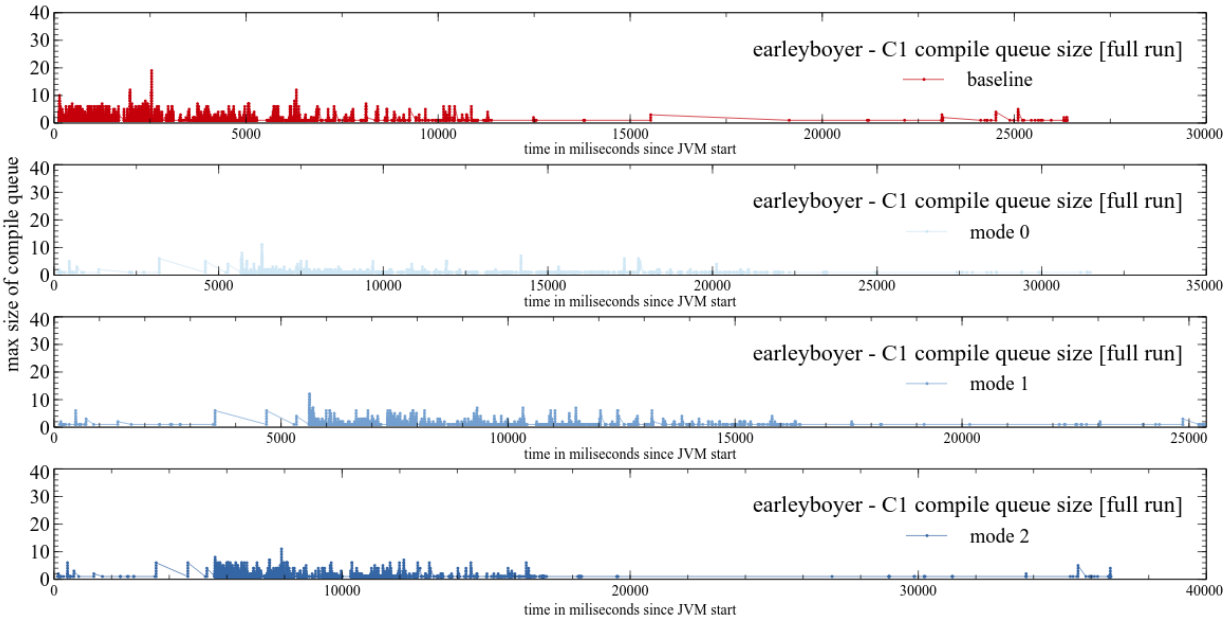

Figure 4.13: C1 Compile queue size over time Octane EarleyBoyer benchmark

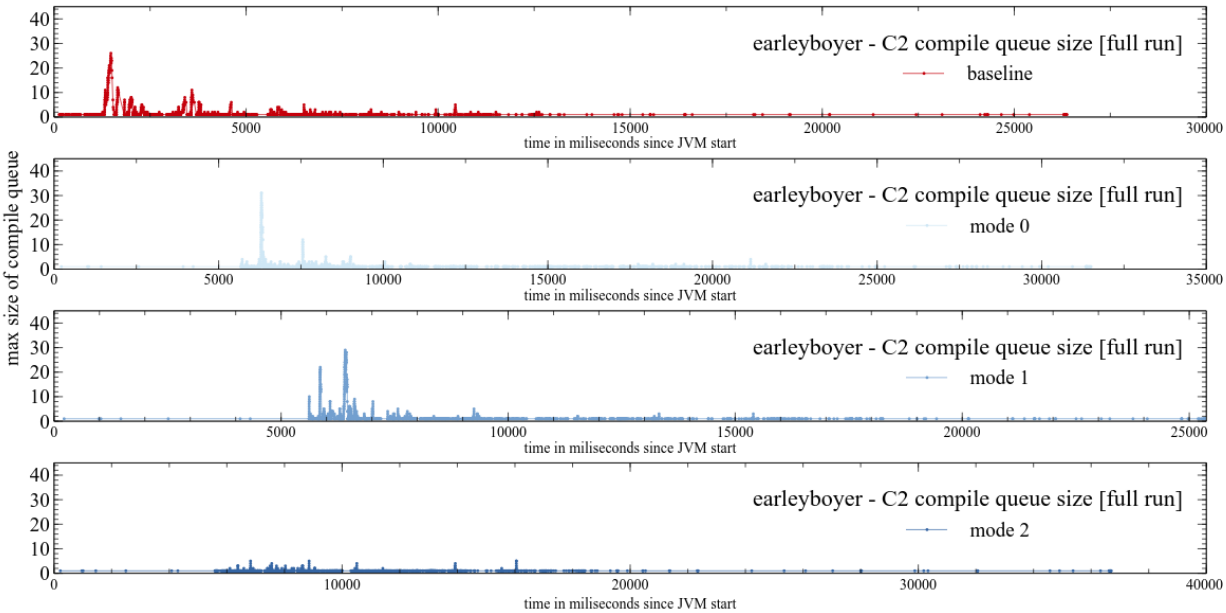

Figure 4.14: C2 Compile queue size over time Octane EarleyBoyer benchmark

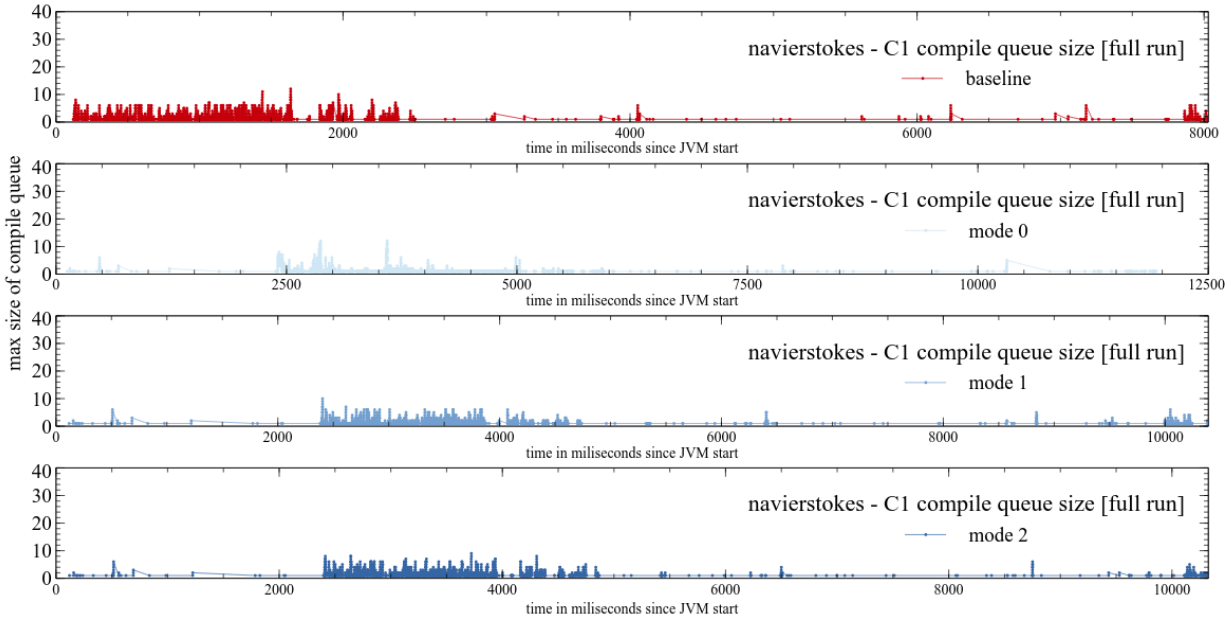

Figure 4.15: C1 Compile queue size over time Octane NavierStokes benchmark

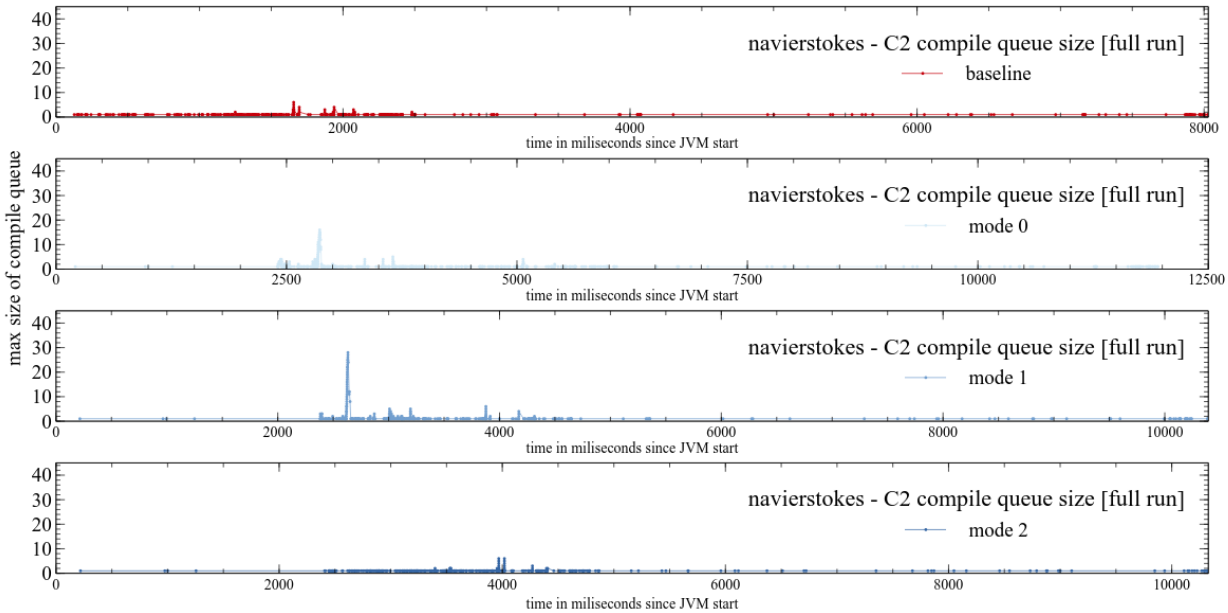

Figure 4.16: C2 Compile queue size over time Octane NavierStokes benchmark

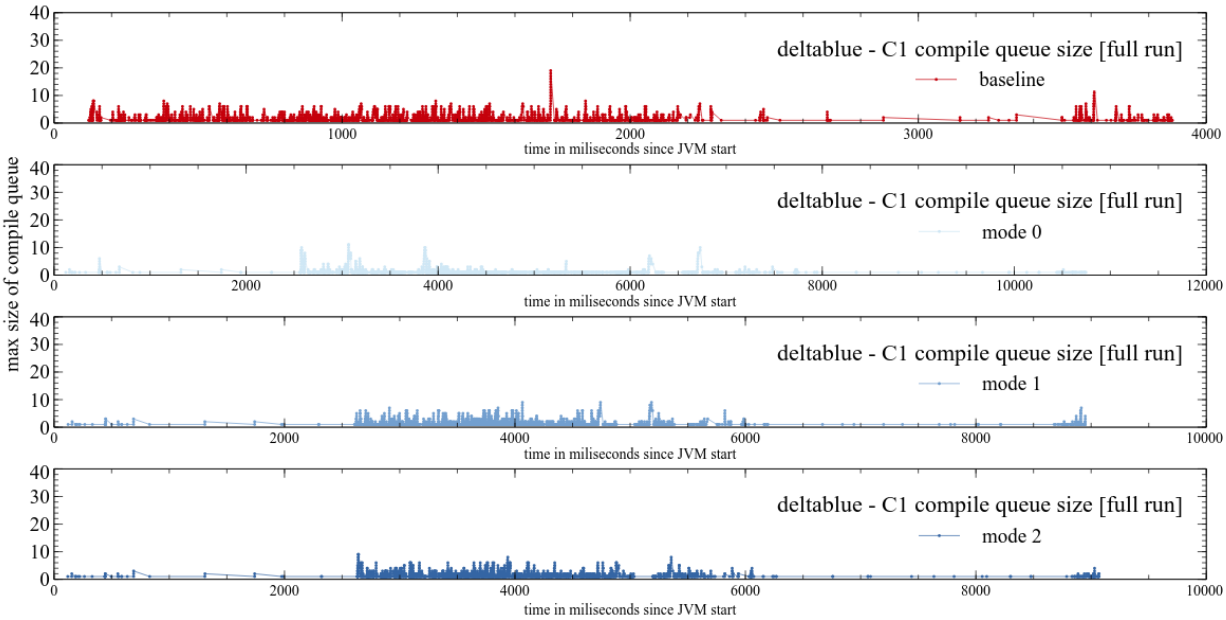

Figure 4.17: C1 Compile queue size over time Octane DeltaBlue benchmark

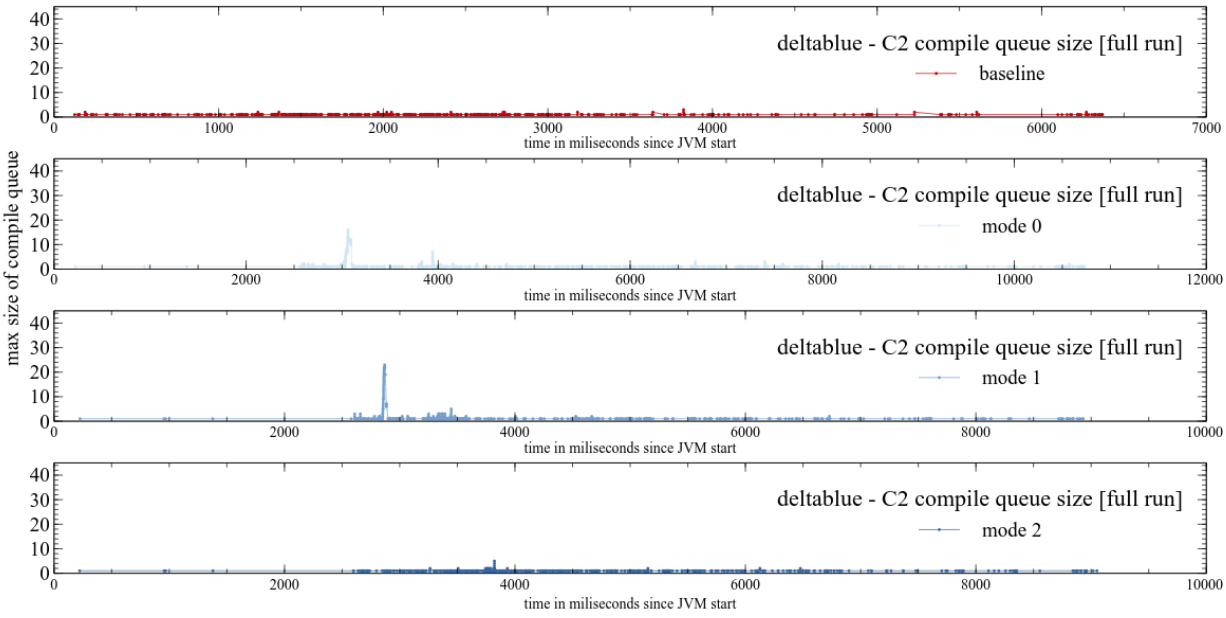

Figure 4.18: C2 Compile queue size over time Octane DeltaBlue benchmark

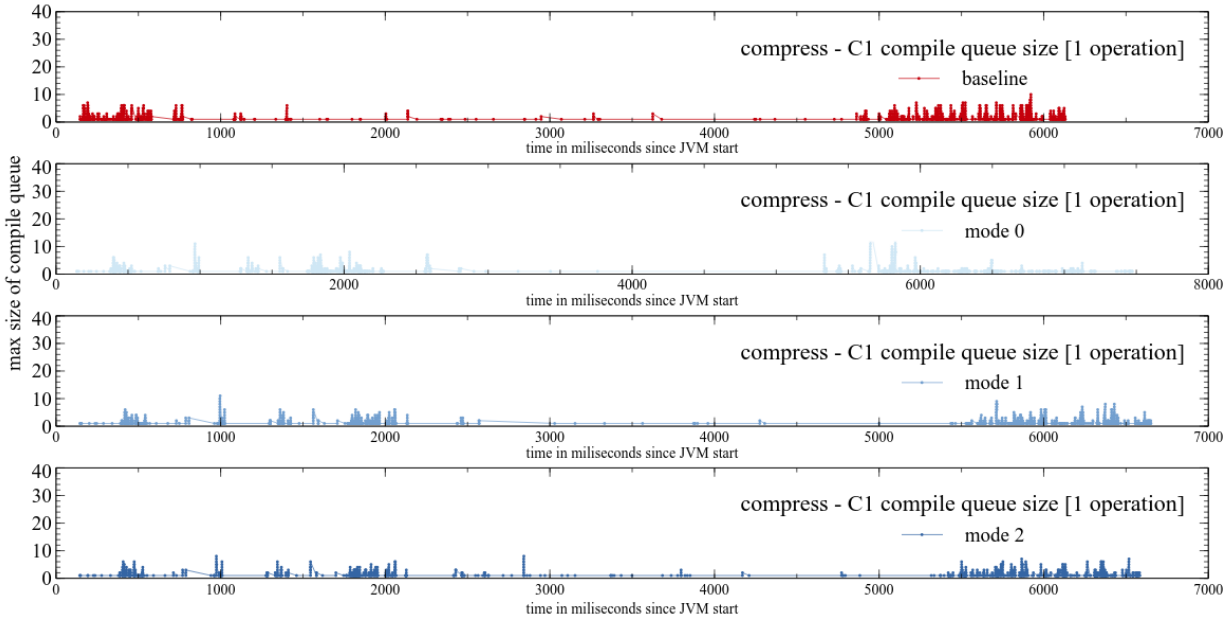

Figure 4.19: C1 Compile queue size over time SPECjvm compress benchmark

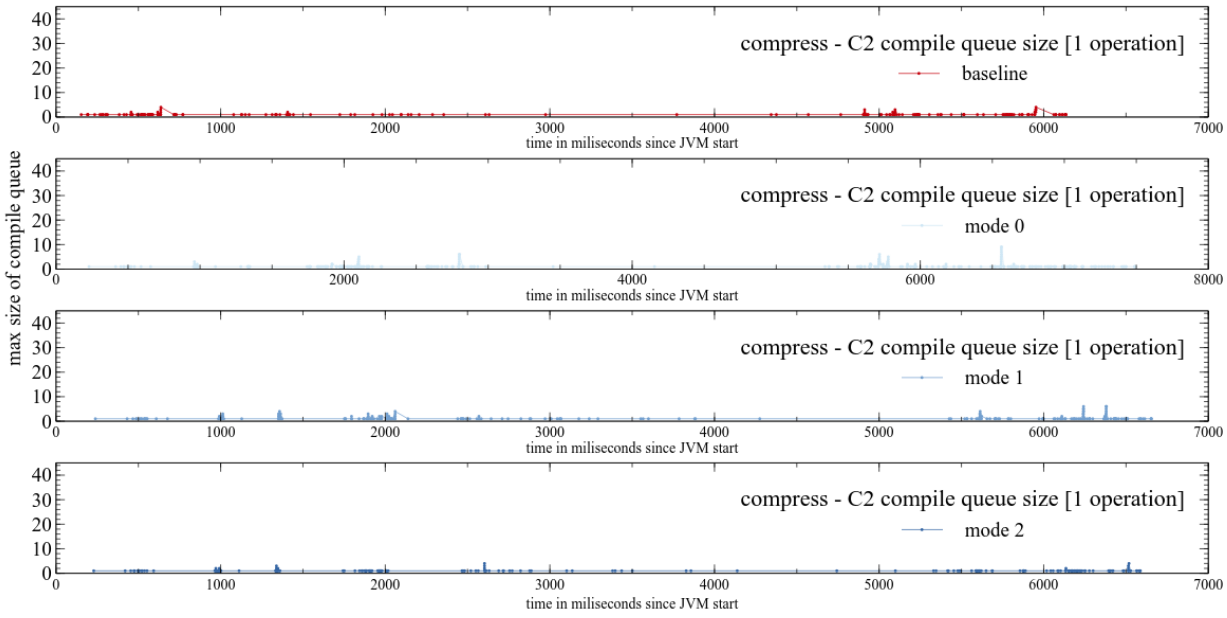

Figure 4.20: C2 Compile queue size over time SPECjvm compress benchmark

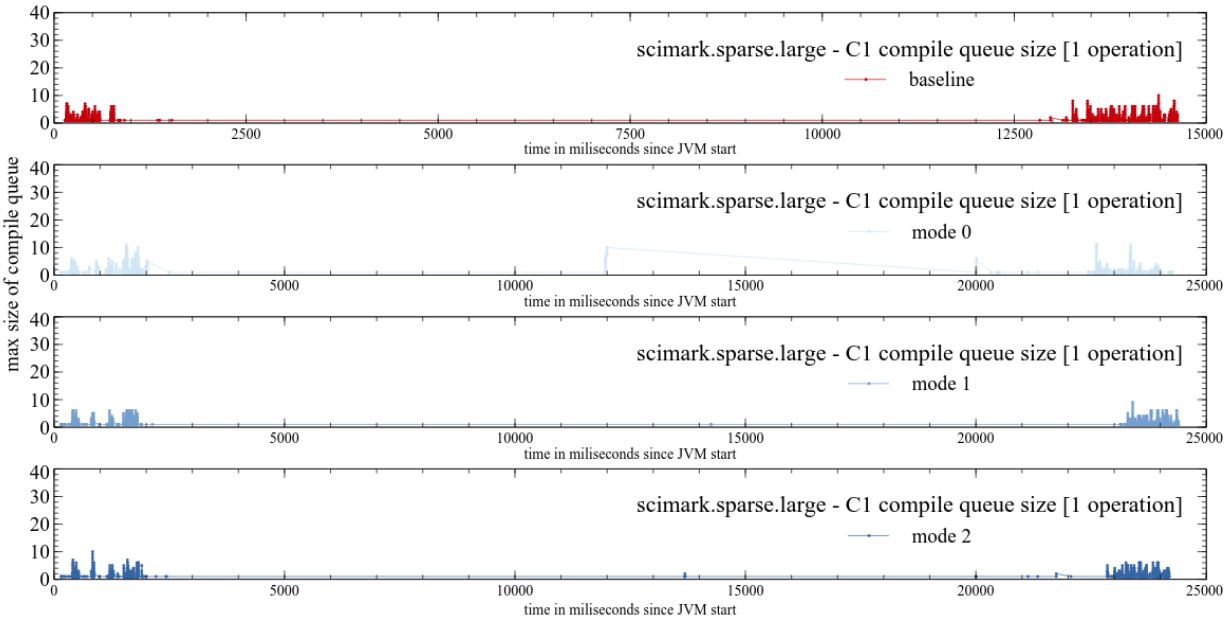

Figure 4.21: C1 Compile queue size over time SPECjvm scimark.sparse.large benchmark

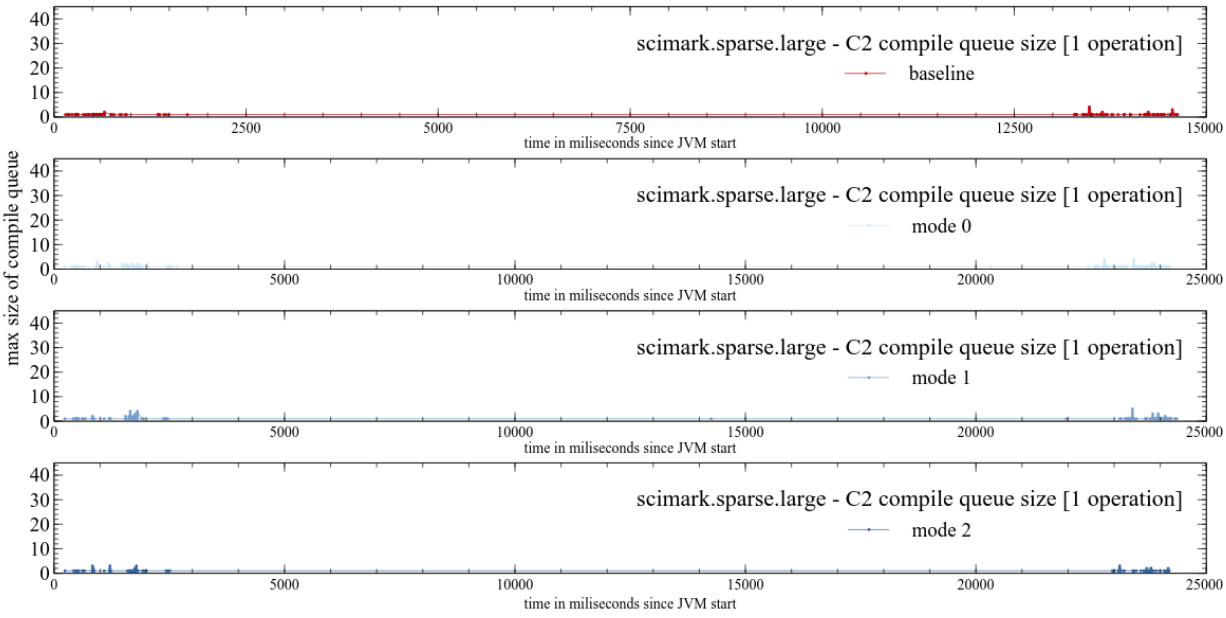

Figure 4.22: C2 Compile queue size over time SPECjvm scimark.sparse.large benchmark

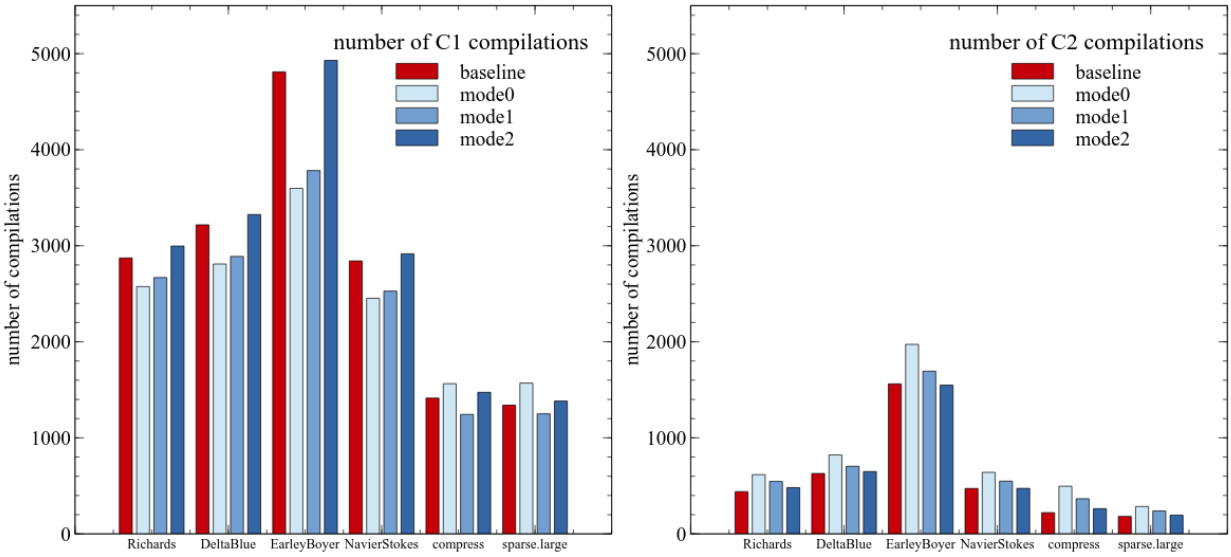

Figure 4.23: Number of compilations for some specJVM and octane benchmarks

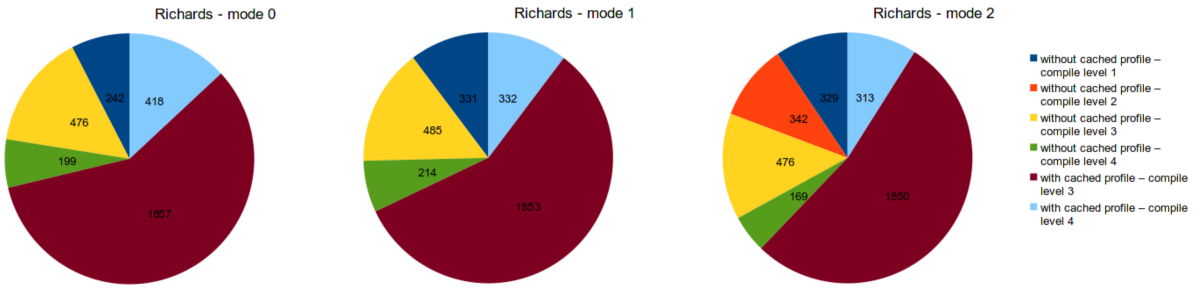

Figure 4.24: Ratio of compilations Octane Richards benchmark

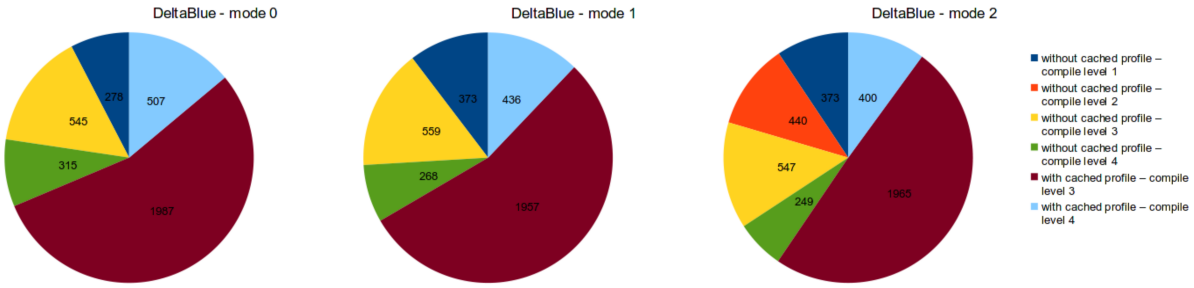

Figure 4.25: Ratio of compilations Octane DeltaBlue benchmark

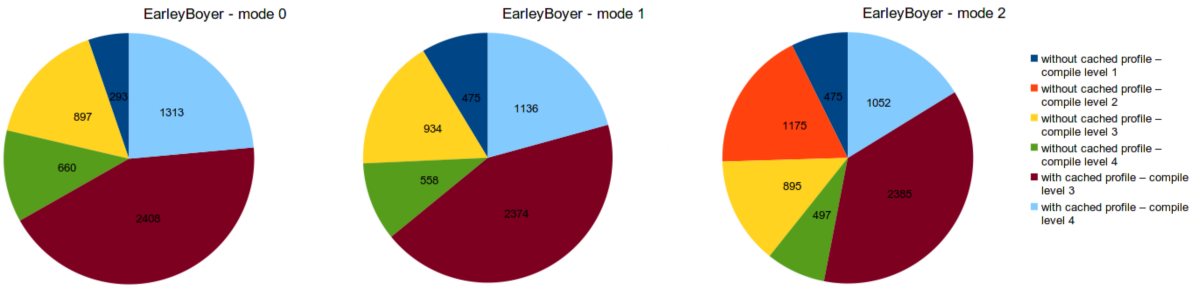

Figure 4.26: Ratio of compilations Octane EarleyBoyer benchmark

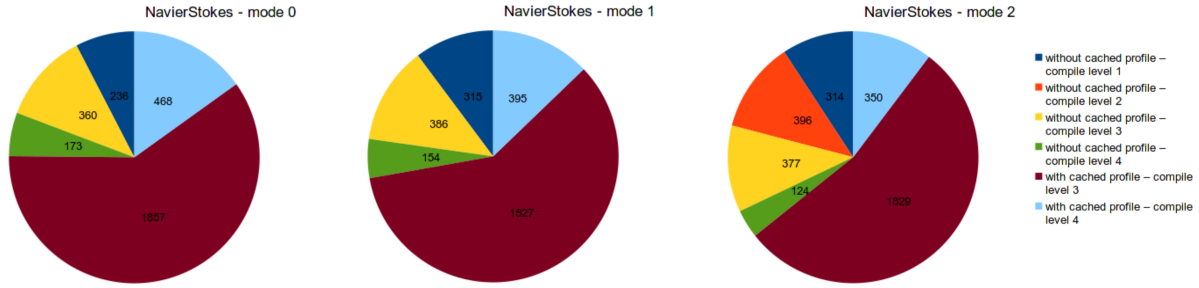

Figure 4.27: Ratio of compilations Octane NavierStokes benchmark

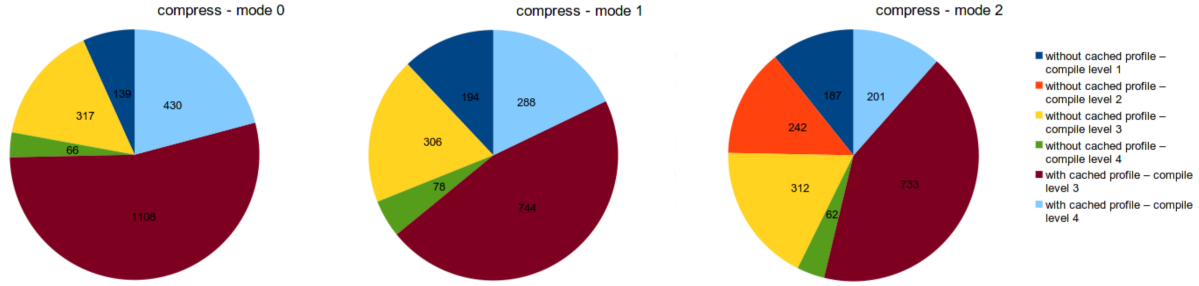

Figure 4.28: Ratio of compilations SPECjvm compress benchmark

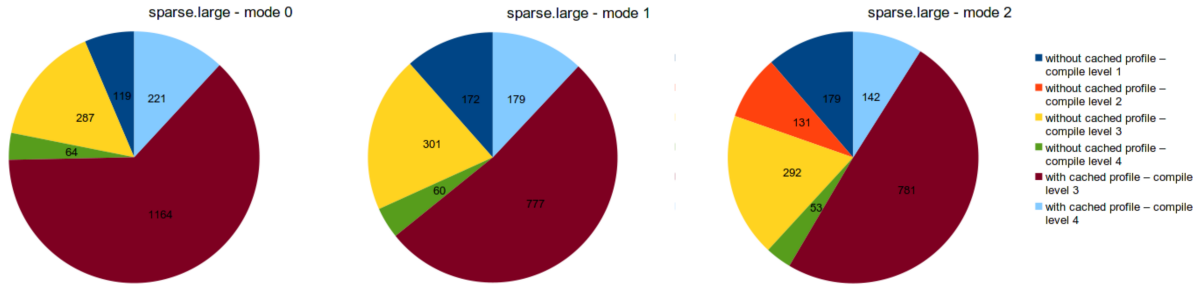

Figure 4.29: Ratio of compilations SPECjvm sparse.large benchmark

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.

- The datastructure of the cached profiles is a simple C heap array. The lookup is $O(n)$ with n being the number of cached method compilations. This could be improved to $O(nlog(n))$ by using a more complex tree-like data structure.
- Additionally, all method compilations get dumped and stored. The system could be improved in a way, so only the last compilation record of a method is kept in the file. This would decrease the size of the cached profile file and decrease the 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 merge the profiling information to achieve even more complete profiles.
- We think it could also be beneficial to be able 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, so they can be reused in future runs of the JVM.

The expected benefits are 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.

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 this can be a valuable asset in scenarios where a fast JVM warmup is appreciated and greatly help to 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 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 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 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 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 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 0 128 150 152 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
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

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

class	changed	inserted	deleted	modified	unchanged
src/share/vm/ci/ciClassList.hpp	2	2	0	0	121
src/share/vm/ci/ciEnv.cpp	58	56	0	2	1283
src/share/vm/ci/ciEnv.hpp	5	5	0	0	469

TODO

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 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*

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.