Bachelor Thesis

Profile Caching for the Java Virtual Machine

Marcel Mohler

Zoltán Majó
Tobias Hartmann
Oracle Cooperation

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

August 2015

Introduction

Virtual Machines like the Java Virtual Machine (JVM) are used as the execution environment of choice for many modern programming languages. The VMs interpret a suitable intermediate language (e.g., Java Byte Code for the JVM) and provide the runtime system for application programs and usually include a garbage collector, a thread scheduler, interfaces to the host operating system. As interpretation of intermediate code is time-consuming, VMs include usually a Just-in-Time (JIT) compiler that translates frequently-executed functions or methods to "native" code (e.g., x86 instructions).

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 and 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 VM (execution at Tier 0). The interpreter performs profiling, and if a method is determined to be "hot", 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 Tier 2. 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 "hot" methods does not change from one execution to another and there is no reason to gather again and again the profiling information.

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. This would allow the JIT compiler to use more sophisticated profiles early in program execution and avoid gathering the same profiling as well as prevent further compilations when more information about the method is available. While this will not influence the peak performance of the program, the hope is to decrease the time it's needed to achieve it. I present an implementation on top of the Java Hotspot Virtual Machine as well as profound performance analysis using state-of-the-art benchmarks.

Contents

1 Motivation

1.1 Tiered Compilation in Hotspot

As mentioned in the introduction, Programming Language Virtual Machines 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`. The Bytecode is meant to be interpreted by the virtual machine or further compiled into platform dependend machine code. Hotspot includes one interpreter and two different compilers with different profiling levels resulting in a total of 5 different tiers. Since in literature and the JVM source code use the *tiers* are also called *compilation levels* I use both as synonyms. The following Figure 1.1 gives a short overview as well as showing the standard transitions.

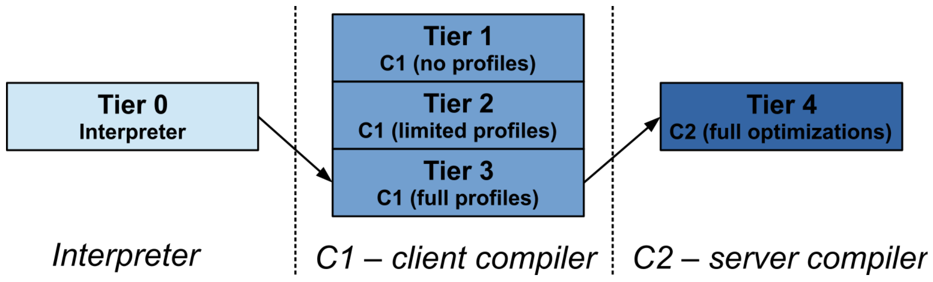


Figure 1.1: Overview of compilation tiers

All methods start being executed by Tier 0 also called the Interpreter. The interpreter is template-based, meaning for each bytecode instruction it emits a predefined assembly code snippet. During execution this code is also profiled. This means method execution counters, loop back-branches and additional statistics are counted. 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. Once one these counters exceed a predefined, constant threshold the method is considered *hot* which 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, and continues gathering full profiles. C1's goal is to provide a fast compilation with a low memory footprint. The client compiler performs simple optimizations such as constant folding, null check elimination and method inlining. More information about C1 can be found in [5] and [3]. The levels 1 and 2 include the same optimization but offer no or less profiling information and are used in special cases, for example if the compiler is already very busy or enough profiles are available.

Eventually, when further compile thresholds are exceeded, the JVM further compiles the method with C2, also known as *server* compiler. The server compiler makes use of the gathered profiles in Tier 0 and Tier 3 and produces highly optimized code. C2 includes far more 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. A more detailed look at the server compiler can be found in [4].

The naming scheme *client/server* comes from back in the days were tiered compilation was not available and one had to choose the 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 used.

Tiered compilation was introduced to improve start-up performance of the JVM. Starting with the interpreter means that there is zero wait time until the method is executed since one does not need to wait until a compilation is finished. C1 allows the JVM to have more optimized of the code available early which then can be used to create a richer profile to be used when compiling with C2. 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. This is further described in the Section 1.2 *Deoptimizations*.

1.2 Deoptimizations

Ideally we compile a method 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. 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, very 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 costly. The previous interpreter state has to be restored and the method will be executed using the slow interpreter. Eventually the method might get recompiled with the newly gained information.

1.3 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 3 and 3 to 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.

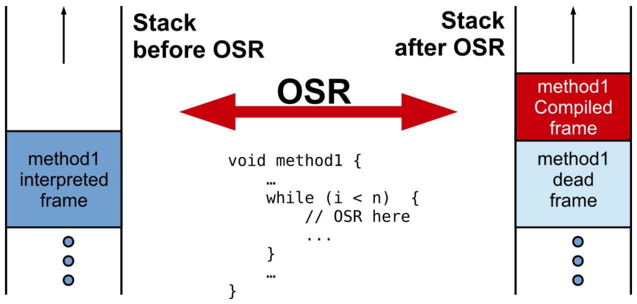
On Stack Replacement uses a simpler predicate:

$$b > TierXBackEdgeThreshold * s$$

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

1.4 On Stack Replacement

Since the JVM does not only count method invocations but also loop back branches (see also Section 1.3) it can happen that a method gets 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 stack (see Figure 1.2) and is called *on stack replacement*. An example would be a simple method consisting of a very long running loop.



```
void method1 {
    …
    while (i < n)  {
        // OSR here
        ...
    }
    …
}
```

Figure 1.2: Graphical schema of OSR

Listing 1.1: Simple method that does not get compiled

```java
class NoCompile {
    double result = 0.0;
    for(int c = 0; c < 100; c++) {
      result = method1(result);
    }
    public static double method1(double count) {
        for(int k = 0; k < 10000000; k++) {
            count = count + 50000;
        }
        return count;
    }
}
```

1.5 Simple Examples

I continue with presenting two very simple examples that illustrate the usage and benefit from using cached profiles. To start I consider a standard Java Hotspot execution with OnStackReplacement disabled. I'm using my implementation described in Chapter 2 in CachedProfileMode 0 (see 2.4.1). The measurements are done on a Dual-Core machine with 8GB of RAM. To measure the method invocation time I use hprof and the average of 10 runs. The error bars show the 95% confidence interval.

1.5.1 Example 1

Example one is a simple class that invokes a method one hundred times. The method itself consists of a long running loop. The source code is shown in Listing 1.1. Since OSR is disabled and a compilation to level 3 is triggered after 200 invocations this method never leaves the interpreter. I call this run the *Baseline*. To show the influence of cached profiles I 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 I use these profiles and achieve significantly better performance as you can see in Figure 1.3. This increase comes mainly from the fact that having a cached profile available allows the JVM to compile methods earlier since there is no need to gather the profiling information.
Enabling OSR again and the difference between with and without cached profiles vanishes. This happens because Hotspot quickly realizes the hotness of the method and compiles it during the first invocation already. Since the method is simple the OSR compiled version already includes enough profiling information and does not trigger any deoptimizations. Therefore there is also no performance difference when using the cached profile.

1.5.2 Example 2

However, OSR is one of the main features of Hotspot to improve the JIT performance and disabling that does not give us any practical results. Since we want an example which demonstrates the influence of cached profiles, I came up with the example sketched in Listing 1.2 which is slightly

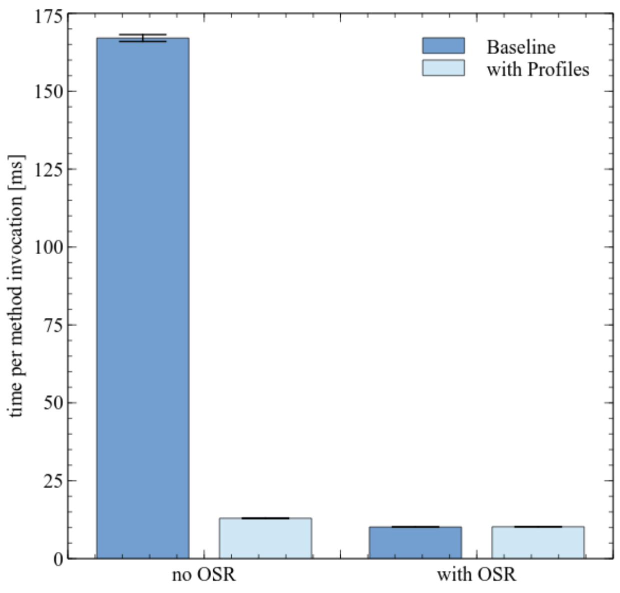


Figure 1.3: NoCompile.method1 - per method invocation time

more complex but still easy to understand.

The idea is to create a method that takes a different, long running branch on each of it's 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 1.4 the baseline indeed averages at around 130 deoptimizations and a time per method invocation of 200 ms.

Now I 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 1.4 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 190ms now.

Listing 1.2: Simple method that causes many deoptimizations

```java
class ManyDeopts {
    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;
    }
}
```

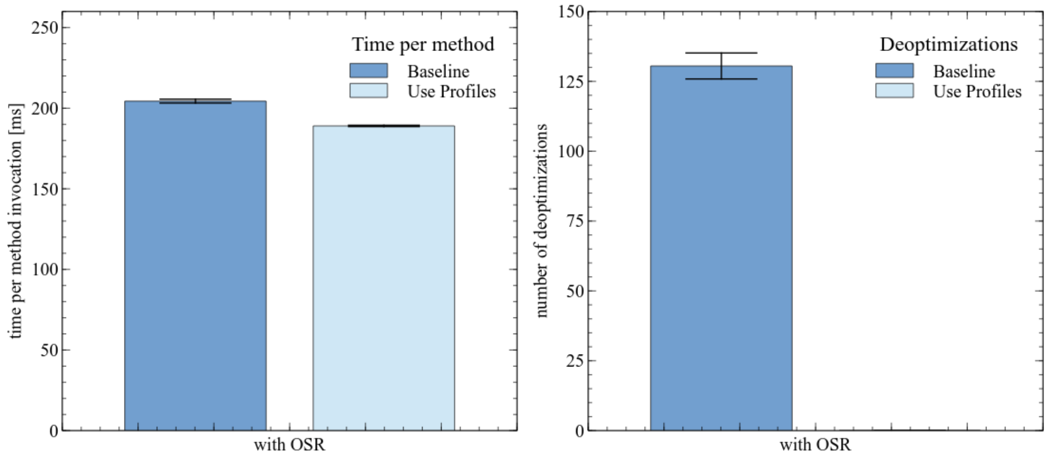


Figure 1.4: ManyDeopts.method1 - per method invocation time and deoptimization count

2 Implementation / Design

This chapter describes the implementation of the cached profiles implementation for Hotspot, written as part of this thesis.

Hotspot is an Java virtual machine implementation maintained by Oracle Cooperation. It is part of the open source project `OpenJDK` and the source code is available at `http://openjdk.java.net/`.

Most of the work is included in two new classes `/share/vm/ci/ciCacheProfiles.cpp` and `/share/vm/ci/ciCacheProfilesBroker.cpp` as well as modifications to `/share/vm/ci/ciEnv.cpp` and `/share/vm/compiler/compileBroker.cpp`.
Most of the code is located in `/share/vm/ci/ciCacheProfiles.cpp`, a class that takes care of setting up a datastructure for the cached profiles as well as providing public methods to check if a method is cached or not. The class `/share/vm/ci/ciCacheProfilesBroker.cpp` gets called before a method that has a profile available gets 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 or appendix TODO.

The changes are provided in form of a patch for Hotspot version 8182 TODO. This original version is referred to as *Baseline*.

I will describe and explain the functionality and the implementation design decision in the following sections, ordered by the appearance in execution.

2.1 Creating cached profiles

The baseline version of Hotspot already offered a functionality to replay a compilation based on dumped profiling information. This is mainly used in case the JVM crashes during JIT compilation to replay the compilation again and help finding the cause of this crash. Dumping the data needed for the replay is either be done automatically in case of a crash or can be invoked manually by specifying the `DumpReplay` compile command option per method. I introduce method option called `DumpProfile` as well as a 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 one to select single methods to dump their profile. The second commands dumps profiles of all compiled methods into a single file. The file can be opened with any text editor and is called *cached_profiles.dat*.

As soon as a method gets compiled on level 3 or level 4 all information about the methods used in the compiled method as well as their profiling information get converted to a string and written to disk. Methods compiled with level 1 and 2 will not be considered. Both are rarely used in practice and do only include none respectively little profiling information.

Since method often get compiled multiple times and at different tier, this can result in dumping compilation information about the same method multiple times. How this will be taken care of is described in Section 2.2. Together with some additional information about the compilation itself, for example the bytecode index of the compiled method in case of OSR, the compiler will be able to redo the same compilation on a future run of the java virtual machine.

2.2 Initializing cached profiles

I introduce a new compiler flag `-XX:+CacheProfiles` that enables the use of profiles that have been written to disk in a previous run of the Java Virtual Machine. Per 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 simple datastructure. This datastructure is kept in memory during the whole execution of the JVM to avoid multiple scans of the file. The parsing process gets invoked during boot up of the JVM, directly after the compileBroker gets initialized. This happens before any methods get executed and blocks the JVM until finished. As mentioned in Section 2.1 the file consists of method informations, 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 and their profiling information. A method's compile information could have been dumped multiple times, so it can happen that there are multiple CompileRecords for the same method. In this case, Hotspot will only keep the CompileRecords that are created based on the last data written to the file. 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 therefore no branches with uncommon traps which result in costly 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. The choice has been done for simplicity and leaves up room for further optimizations.

2.3 Using cached profiles

The idea is to use cached profiles whenever possible and if none are available continue as usual.

The thesis offers three different modes `mode0`, `mode1` and `mode2` on how the profiles are used. The following paragraph describes the behaviour of mode0 and mode1 and I will discuss the differences in detail in Section 2.4, especially `mode2` in Section 2.4.3.

A graphical, simplified overview of the program flow for compiling a method with the changes introduced in this thesis can be found in Figure 2.1. As mentioned before once certain thresholds

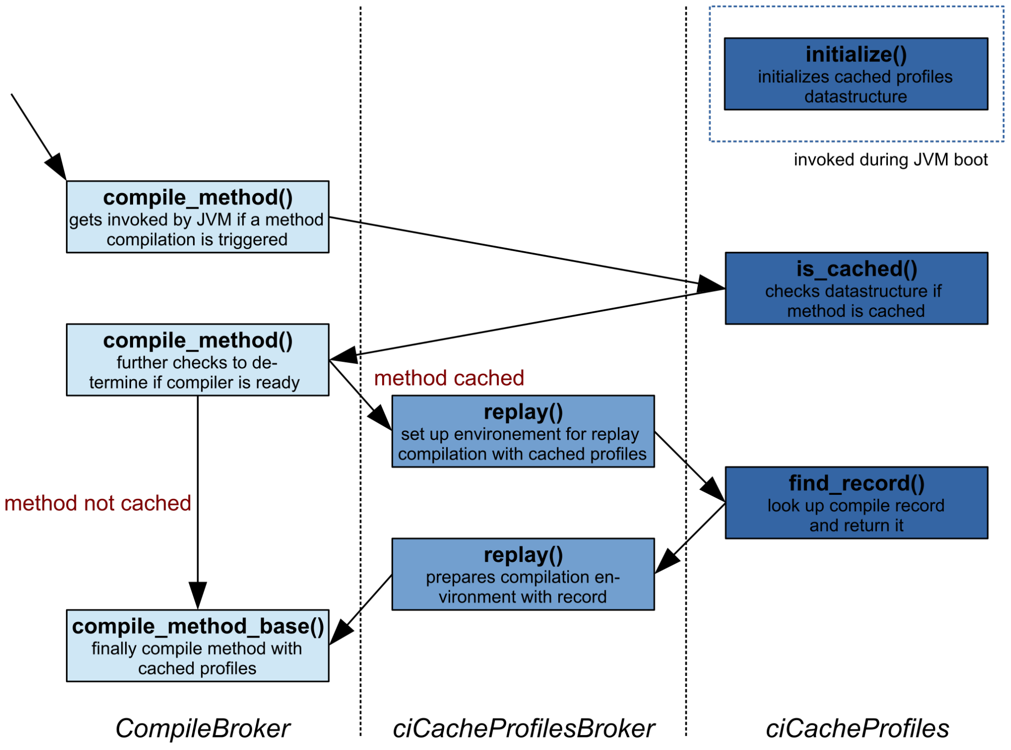

initialize()
initializes cached profiles datastructure
invoked during JVM boot
compile_method()
gets invoked by JVM if a method compilation is triggered
is_cached()
checks datastructure if method is cached
compile_method()
further checks to determine if compiler is ready
method cached
replay()
set up environement for replay compilation with cached profiles
find_record()
look up compile record and return it
method not cached
replay()
prepares compilation environment with record
compile_method_base()
finally compile method with cached profiles
CompileBroker
ciCacheProfilesBroker
ciCacheProfiles

Figure 2.1: program flow for compiling a method

are exceeded a method gets scheduled for compilation. This means that the JVM will invoke a method called `compile_method()` located in the `compileBroker` class. This method for example checks if the compile queue isn't full or if there is already another compilation of that particular method running. I extended this method with a call to `ciCacheProfiles::is_cached(Method* method)` which either returns 0 if the method is not cached or returns an integer value, reflecting the compile level, in case that method has a cached profile available. Because only methods compiled with level 3 or 4 get cached, this call only gets executed if the compilation request is also of level 3 or higher. Note that this also means that if a method compilation of level 3 is initiated and a CompileRecord of level 4 is available that the highest level profile will be used. Therefore the method gets immediately compiled with C2 instead of C1. In case the method is not cached the execution continues like in the baseline version. Otherwise, the `compileBroker` calls into `ciCacheProfilesBroker` to replay the compilation, based on the saved profile. 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 get used by a 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 basically an extension of the static class `compileBroker`, `ciCacheProfiles` and `ciCacheProfilesBroker` are static classes as well. The `compileBroker` gets invoked by the single JVM main thread and is not multi threaded, therefore there is no need to make the compilRecord datastructure or any of the new implementations thread safe.

2.4 Different usage modes for cached profiles

The implementation of cached profiles offers 3 different modes which differ in the transitions between the compilation tiers. The motivation as well as the advantages and disadvantages of each mode are described in the following three subsections. While `mode0` and `mode1` are similar except for the compile thresholds, `mode2` differs significantly.

2.4.1 Compile Thresholds lowered (mode 0)

The first mode is based on the consideration that a method that has a profile available does not require extensive profiling anymore. Therefore the compile thresholds (see Section 1.3) of these methods are lowered automatically. By default, they 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 doing class loading in C1 or C2 which was considered out of the scope for this thesis. As mentioned before the triggered compilation will use the latest available compile record. Eventually, most hot methods get compiled by C2 and therefore the used compiled record is usually a C2 one. In this case the JVM will jump directly from compile level 0 to compile level 4 and avoid a costly C1 compilation as well as gathering profiling information during level 0 and level 3. It will directly use the highly optimized version generated by C2 and ideally result in a lower time to reach peak performance. On the downside this increases the load on the C2 compiler and fill the compile queue more quickly. Mode 0 is the default mode and used if not further specified.

2.4.2 Unmodified Compile Thresholds (mode 1)

Mode 1 is doing exactly the same as mode 1 but does not scale the compilation thresholds automatically. This is done to decrease the load increase on C2 as mentioned in Subsection 2.4.1. Apart from this change mode 1 has the same behaviour as mode 0.

2.4.3 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 we can not use Tier 1. Tier 2 generally is considered about 30% faster than Tier 3 [1].

If then the Tier4 thresholds are reached the method is compiled using C2 and the cached profiles.

The above only makes sense if there is a C2 profile available. If only a C1 profile is available, mode 2 will only use the profile during the C1 compilation and then use the new profiles.

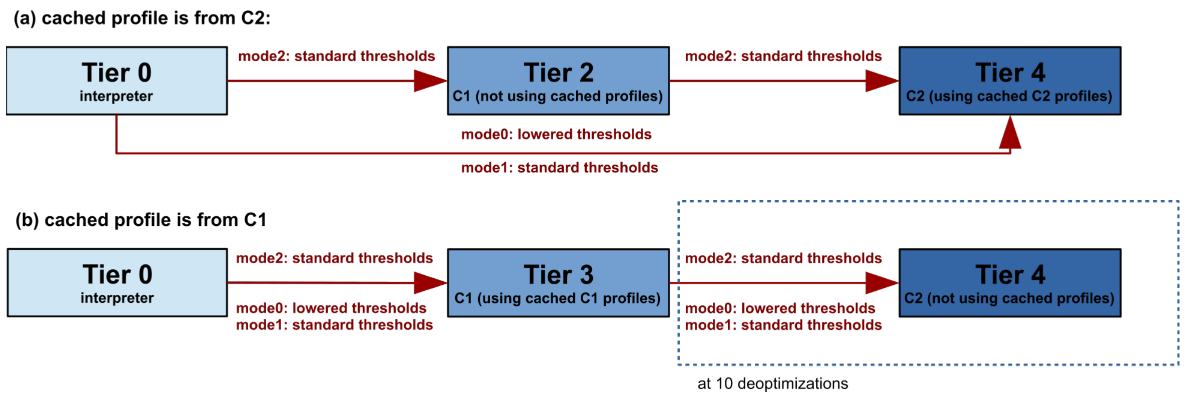

(a) cached profile is from C2:
Tier 0
interpreter
mode2: standard thresholds
Tier 2
C1 (not using cached profiles)
mode2: standard thresholds
Tier 4
C2 (using cached C2 profiles)
mode0: lowered thresholds
mode1: standard thresholds
(b) cached profile is from C1
Tier 0
interpreter
mode2: standard thresholds
mode0: lowered thresholds
mode1: standard thresholds
Tier 3
C1 (using cached C1 profiles)
mode2: standard thresholds
mode0: lowered thresholds
mode1: standard thresholds
Tier 4
C2 (not using cached profiles)
at 10 deoptimizations

Figure 2.2: Tier transitions of different modes

2.5 Issues

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. To circumvent that behavior I modified the code that only methods which have been deoptimized less then 10 times already will get compiled using cached profiles. If they are above that limit a standard compilation 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.

2.6 Debug output

For debugging and benchmarking purposes I implemented four debug flags 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:+PrintCompileQueue	prints the total amount of methods in the compile queue each time a method gets added

3 Performance

3.1 General peformance

3.2 Deoptimizations

3.3 Effect on compile queue

4 Possible Improvements

- Better datasctructure - Merging multiple profiles cleverly

5 Conclusion

A Appendix

A.1 Tiered Compilation Thresholds

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

Bibliography

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

[2] 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.

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

[4] 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.

[5] 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.