Bachelor Thesis

Profile Caching for the Java Virtual Machine

Marcel Mohler
ETH Zurich

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 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 it 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 Hotspot **1**

1.1 Tiered Compilation . . . 1

1.2 Deoptimizations . . . 3

1.3 Compile Thresholds . . . 4

1.4 On-Stack Replacement . . . 4

2 Motivation **7**

2.1 Example 1 . . . 8

2.2 Example 2 . . . 9

2.3 Similar systems . . . 11

3 Implementation / Design **13**

3.1 Creating cached profiles . . . 13

3.2 Initializing cached profiles . . . 14

3.3 Using cached profiles . . . 15

3.4 Different usage modes for cached profiles . . . 16

3.4.1 Compile Thresholds lowered (mode 0) . . . 16

3.4.2 Unmodified Compile Thresholds (mode 1) . . . 17

3.4.3 Modified C1 stage (mode 2) . . . 17

3.5 Issues . . . 18

3.6 Debug output . . . 18

4 Performance **19**

4.1 Setup . . . 19

4.2 Startup performance . . . 19

4.3 Without Intrinsics . . . 21

4.4 Deoptimizations . . . 21

4.5 Effect on compile queue . . . 28

5 Possible Improvements **33**

6 Conclusion **35**

A Appendix **37**

A.1 Tiered Compilation Thresholds 38

A.2 SPECjvm Benchmark 38

A.3 Octane Benchmark 40

Bibliography **42**

1 Overview Hotspot

This chapter will provide the reader with an overview of the relevant parts of Java Hotspot. It explains the core concepts that are needed to understand the motivation and implementation of this thesis.

1.1 Tiered Compilation

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` [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.

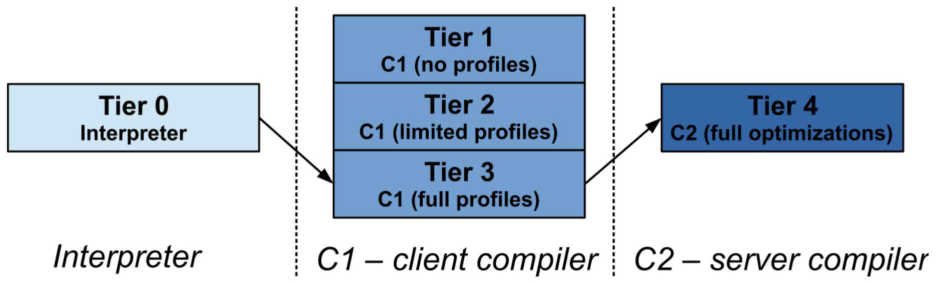


Figure 1.1: Overview of compilation tiers

All methods start being executed at Tier 0, which denotes the interpreter. The interpreter performs a pretty straight forward template-based replacement, meaning for each bytecode instruction it 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. Once one these counters exceed a predefined threshold the method is considered *hot* and calls back to the JVM 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. C1's goal is to provide a fast compilation with a

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

```
public static void m(int i) {
    if ( i == 0 ) { // very common branch (a)
        Math.sin(0);
    } else { // very uncommon branch (b)
        Math.sin(pi + i)
    }
}
// ----------------------------------------------------
// If the JVM realizes based on profiling information,
// that branch (a) is taken all the time:
// compiler could compile the method as follows:
// ----------------------------------------------------
public static void m(int i) {
    if ( i != 0 ) // very common branch (a)
        // UNCOMMON TRAP, call to JVM
    return 0; // result of sin(0)
}
```

low 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 are already 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.

The levels 1 and 2 include the same optimization 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. Anyway, since the profiles generated by C1 are further used in C2, Hotspot is usually interested in creating full profiles and therefor use 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 *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 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 one 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 means that there is zero wait time until the method is executed since one does not need to wait until a compilation is finished. Also, consider that there are always parts of the code that get executes only once, where the compilation overhead would exceed the performance gain. 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. 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 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 (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, 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 expensive. 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:

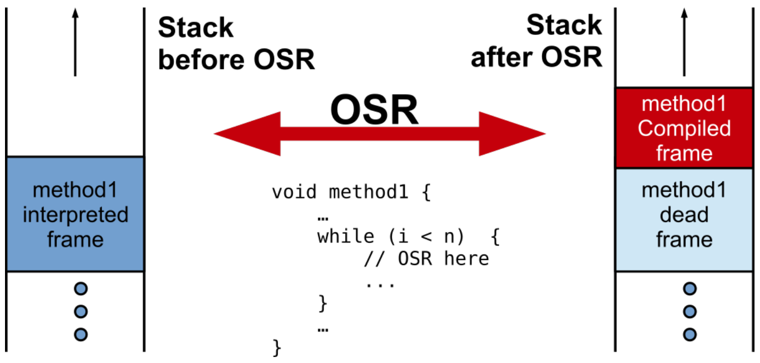
$$b > TierXBackEdgeThreshold * s$$

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 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.



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

Figure 1.2: Graphical schema of OSR

2 Motivation

I continue with presenting two very simple example methods that illustrate the motivation and benefit from using cached profiles. This should provide the reader with an understanding how and why cached profiles can be beneficial for the performance of a Java Virtual Machine. I will omit any implementation details on purpose 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:** Having information about the program flow already, the compiler can avoid gathering profiles and compile methods earlier and directly at higher compilation levels.

2. **Less Deoptimizations:** Since cache profiles get 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 less deoptimizations occur.

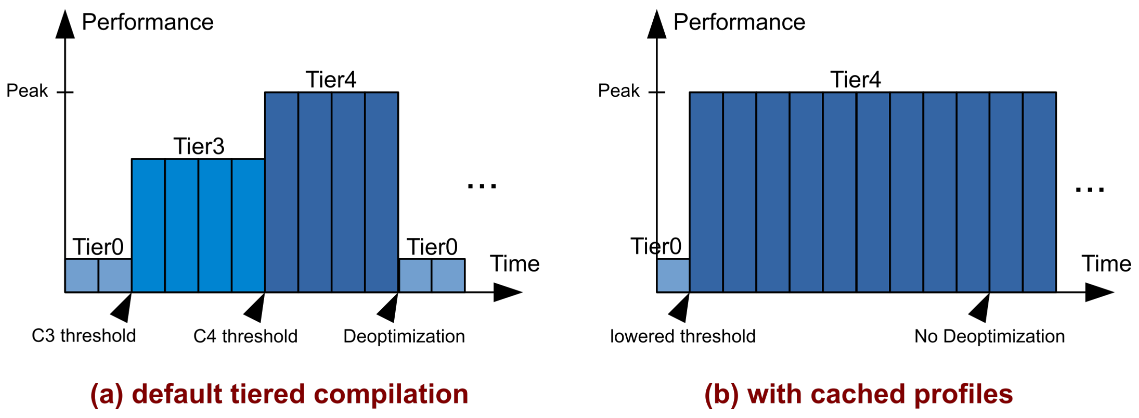


Figure 2.1: schematic visualization of cached profile benefit

Figure 2.1 gives a schematic visualization of the expected effect on performance of a single method when using cached profiles compared to the current state without such a system and standard tiered compilation. The blue bars roughly represent method invocations and higher bars equal 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.

Listing 2.1: 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;
   }
}
```

A more detailed performance analysis, also considering possible performance regressions is done in Chapter 4.

I'm 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 I 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

For this very first example, on-stack replacement has been disabled to keep the system simple and easy to understand.
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 2.1. 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 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.

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 the JIT compiler produces perfectly optimized code during the first method invocation already. Even the OSR compiled code

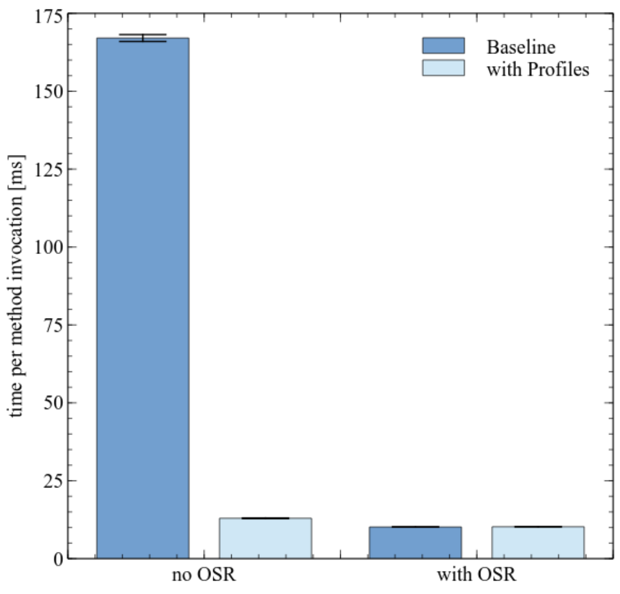


Figure 2.2: NoCompile.method1 - per method invocation time

never triggers any deoptimization due to the simplicity of the loop. So this example appears rather artificial since the same performance can be achieved with OSR already but nevertheless shows the influence of early compilation.

2.2 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 2.2 which is slightly 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 2.3 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 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

Listing 2.2: Simple method that causes many deoptimizations

```
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 < 100000000l; k++) {
            if (count < 100000000l) {
                count = count + 1;
            } else if (count < 300000000l) {
                count = count + 2;
                .
                .
                .
            } else if (count < 505000000000l) {
                count = count + 100;
            }
            count = count + 50000;
        }
        return count;
    }
}
```

is even significantly faster with averaging at 190ms now.

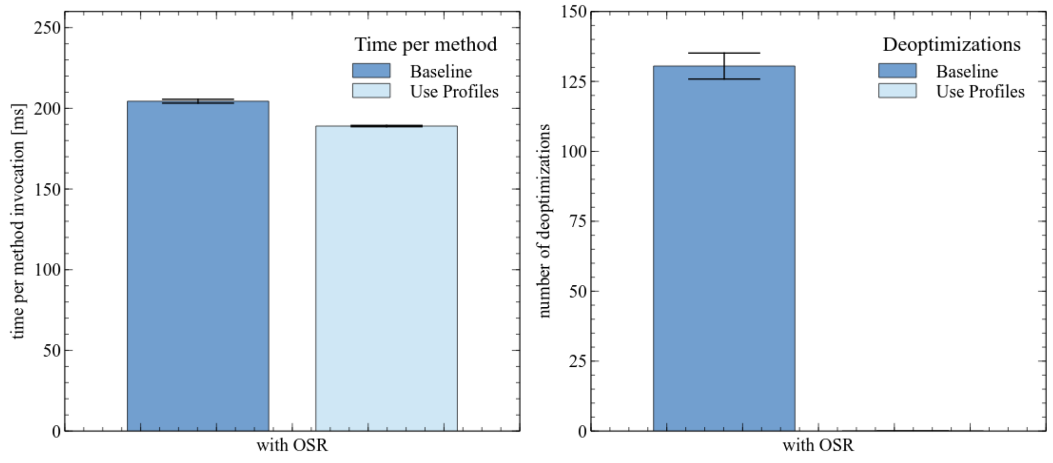


Figure 2.3: ManyDeopts.method1 - per method invocation time and deoptimization count

2.3 Similar systems

In commercially available JVMs the idea of caching profiles is not new. In fact 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] that aims to increase warmup 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 drop as soon as the market goes live and the clients start trading. The reasons are deoptimizations that occur because due to changed situations uncommon branch paths are taken or yet unused methods are invoked. In the past their clients used techniques to warm up the JVM, for example doing fake trades prior to market opening. However this does not solve the problem since the JVM optimizes for these fake trades and still runs into deoptimizations once actual trades are meant to happen.

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, they do not provide any numbers how their system actually improves performance applied to a real system..