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

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Laboratory for Software Technology

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 levels. The following Figure 1.1 gives a short overview as well as showing the standard transitions.

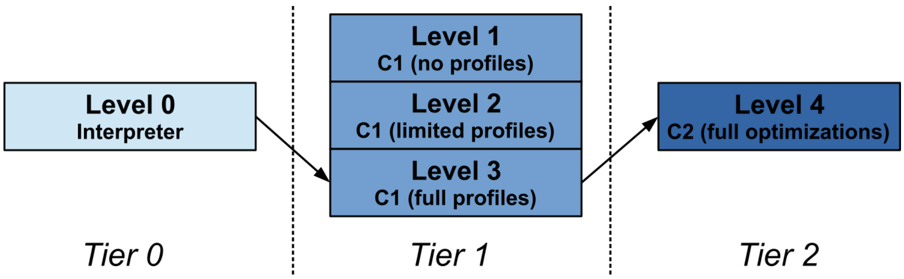

Figure 1.1: Overview over compilation tiers

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 1). This means 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 [4] and [2]. 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 already 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 1 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 [3].

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 so less deoptimizations (see 1.3 occur.

1.2 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 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.

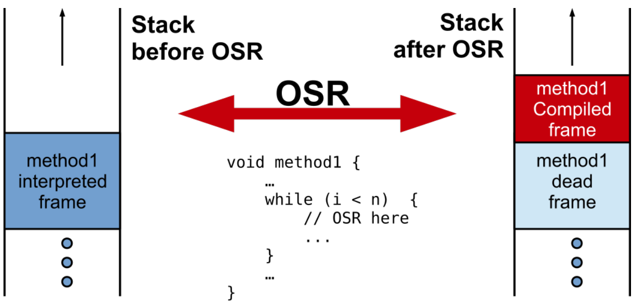

Figure 1.2: Overview over compilation tiers

1.3 Deoptimizations

Ideally we compile a method with as much profiling information as possible. 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 eventually the method might get recompiled with the newly gained information.

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 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 further.

1.5 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).

Listing 1.1: Example usage of the listing package

```
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;
   }
}
```

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.

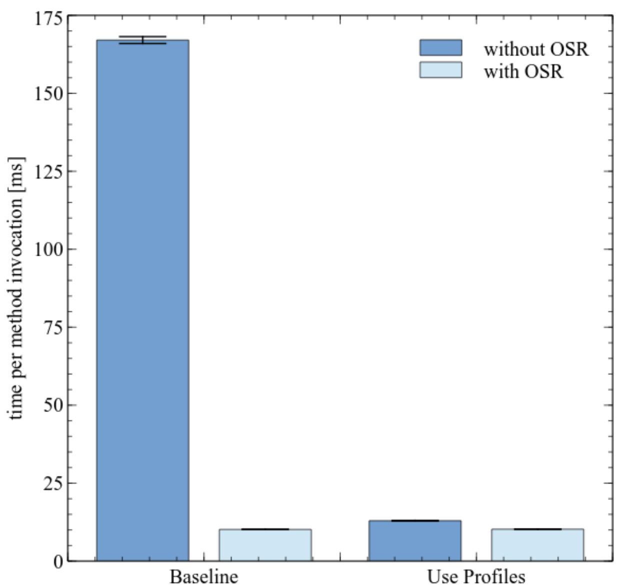

Figure 1.3: NoCompile.method1 - per method invocation time comparison

Enabling OSR again and the difference between with and without cached profiles vanishes. This

Listing 1.2: Example usage of the listing package

```
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 < 10000000l; k++) {
            if (count < 10000000l) {
                count = count + 1;
            } else if (count < 30000000l) {
                count = count + 2;
                .
                .
                .
            } else if (count < 50500000000l) {
               count = count + 100;
            }
            count = count + 50000;
        }
        return count;
   }
}
```

happens because Hotspot quickly realizes the hotness of the method and compiles it during the first invocation already.

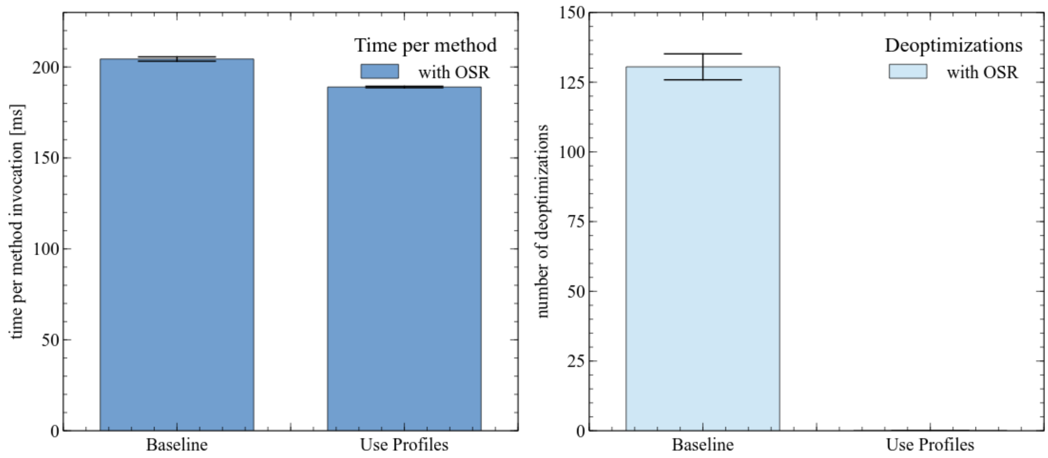

Figure 1.4: ManyDeopt.method1 - per method invocation time and deoptimization count comparison

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 the method gets compiled. The first option can be specified as part of the `-XX:CompileCommand` or `-XX:CompileCommandFile` flag and allows one to force single methods to dump their profile. The second commands dumps all profiles of all compiled methods into a single file called *cached_profiles.dat*.

As soon as a method gets compiled all information about the methods used in the compiled method as well as their profiling information get converted to a string and written on disk. Since method often get compiled multiple times 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 data written to the file last. Since profiling information only grow, the compilation that happened last contains the richest profile and is considered the best to avoid 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. 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.

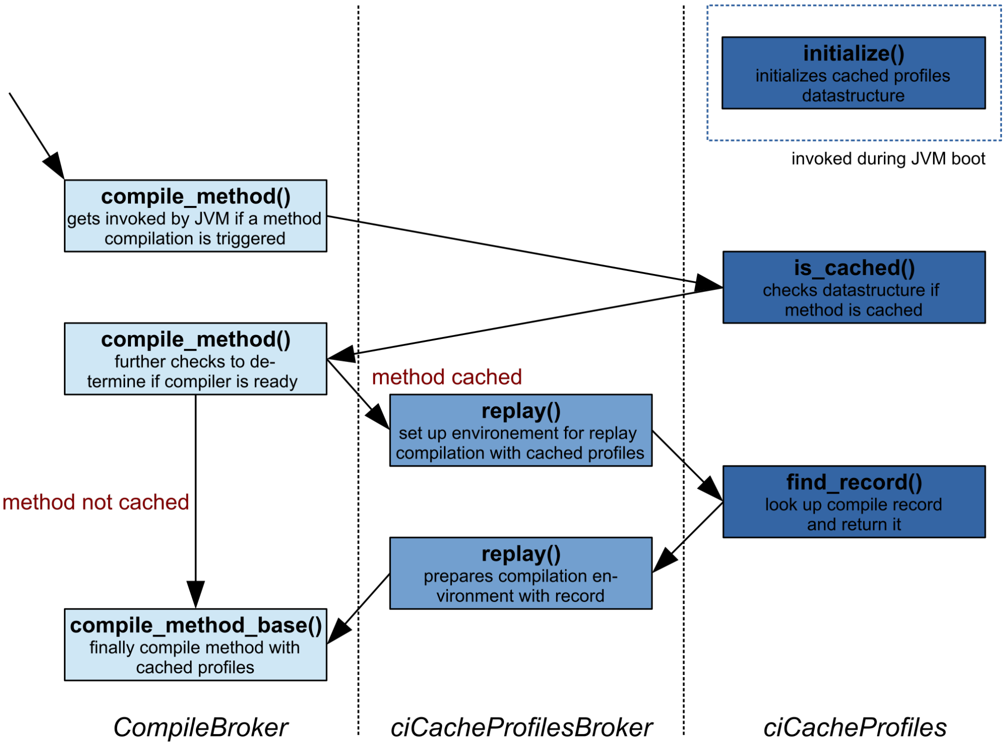

Figure 2.1: program flow for compiling a method

2.4 Different usage modes for cached profiles

2.4.1 Compile Thresholds lowered (mode 0)

2.4.2 Unmodified Compile Thresholds (mode 1)

2.4.3 Modified C1 stage (mode 2)

2.5 Debug outout

For debugging and benchmarking purposes I implemented four debug flags that can be used along with `-XX:+CacheProfiles`. -XX:+PrintCacheProfiles -XX:+PrintDeoptimizationCount -XX:+PrintDeoptimizationCountVerbose -XX:+PrintCompileQueue

3 Performance

3.1 Examples

3.2 SPECjvm 2008

3.3 Nashorn / Octane

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] 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.

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

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

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