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

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

LST
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. 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.

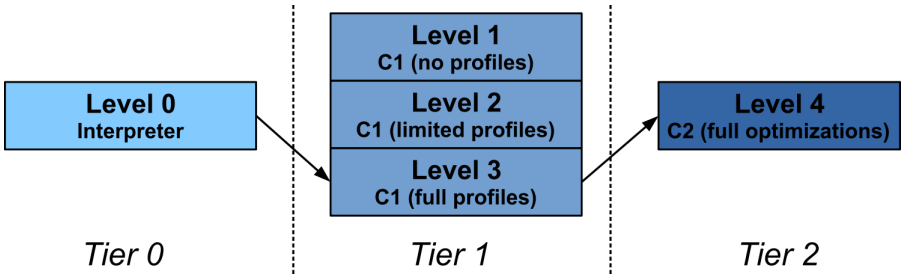

Figure 1.1: Overview over compilation tiers

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 and loop back-branches and additional statistics are counted. Once one these counters exceed a predefined, constant threshold the method is further progressed, for example compiled with 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.

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

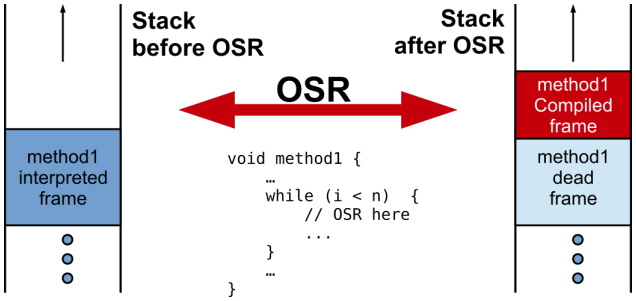

Figure 1.2: Overview over compilation tiers

1.3 Deoptimizations

1.4 Compile Thresholds

1.5 Examples

2 Implementation / Design

3 Performance

3.1 Examples

3.2 SPECjvm 2008

3.3 Nashorn / Octane

4 Conclusion

A Extra Stuff

Additional material such as long mathematical derivations.

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.