Master Thesis

Code Cache Optimizations for Dynamically Compiled Languages

Author(s): Hartmann, Tobias

Publication Date: 2014

Permanent Link: https://doi.org/10.3929/ethz-a-010098048

Rights / License: In Copyright - Non-Commercial Use Permitted
Master Thesis

Code Cache Optimizations for Dynamically Compiled Languages

Tobias Hartmann

Albert Noll
Supervisor

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

February 2014
Abstract

Past activities in optimizing the performance of the HotSpot™ Java Virtual Machine focused on the performance of the dynamic compilers and the supporting runtime. Since dynamically compiled code is stored in a code cache to avoid recompilations, the organization and maintenance of the code cache has a significant impact on the overall performance. The organization of the code cache became even more important with the introduction of tiered compilation in Java Platform, Standard Edition (Java SE) 7. By using two dynamic compilers with different characteristics, not only the amount, but also the number of different types of compiled code increased.

The current code cache is optimized to handle homogeneous code, i.e., only one type of compiled code. The code cache is organized as a single heap data structure on top of a contiguous chunk of memory. Therefore, profiled code which has a predefined limited lifetime is mixed with non-profiled code, which potentially remains in the code cache forever. This leads to different performance and design problems. For example, the method sweeper has to scan the entire code cache while sweeping, even if some entries are never flushed or contain non-method code.

This thesis addresses these issues at a lower layer by redesigning the structure of the code cache. The code cache is segmented into multiple code heaps, each of which contains compiled code of a particular type and therefore separates code with different properties.

The disadvantage of having a fixed size per code heap is then minimized by lazily creating and dynamically resizing these code heaps at runtime.

The main advantages of this design are (i) more efficient sweeping, (ii) improved code locality, (iii) the possibility for fine grained (per code heap) locking and (iv) improved management of heterogeneous code.

A detailed evaluation shows that this approach improves overall performance. The execution time is improved by up to 7% and the more efficient code cache sweeping reduces the time taken by the sweeper by up to 46%. This, together with a decreased fragmentation of the non-profiled code heap by around 98%, leads to a reduced instruction translation lookaside buffer (44%) and instruction cache (14%) miss rate.
Zusammenfassung

Vergangene Bemühungen die Leistung der HotSpot™ Java Virtual Machine zu optimieren, konzentrierten sich vorrangig auf die Leistung der dynamischen Compiler und der unterstützenden Laufzeitumgebung. Da dynamisch kompilierter Code aber, um Rekompilierungen zu vermeiden, in einem Code Cache gespeichert wird, hat die Organisation und Verwaltung dieses Codes einen signifikanten Einfluss auf die Gesamtleistung. Dieser Sachverhalt gewann noch an Bedeutung als mit Java SE 7 die Tiered Compilation eingeführt wurde. Durch gleichzeitige Verwendung zweier dynamischer Compiler, die den Code teilweise instrumentieren, erhöhte sich nicht nur die Menge kompilierten Codes, sondern auch die Anzahl der verschiedenen Codetypen.


Eine detaillierte Auswertung der Implementierungen zeigt, dass der Ansatz vielversprechend ist. Die Laufzeit ist um bis zu 7% verringert, wobei das effizientere Sweepen die Zeit im Sweeper um bis zu 46% reduziert. Dies, zusammen mit der verringerten Fragmentierung des Non-Profiled Code Heaps um ca. 98%, führt zu einer geringeren Instruction TLB (44%) und Instruction Cache (14%) Miss Rate.
Acknowledgments

First of all, I want to thank my supervisor Albert Noll for giving me the opportunity to take part in this challenging project. I would like to thank you for your guidance and help with just the right mix of support and personal responsibility.

This work was performed in cooperation with Oracle. Advice given by Vladimir Kozlov and Christian Thalinger has been a great help in improving the design and implementation. I would like to thank Azeem Jiva for making this possible.

Further, I would like to offer my special thanks to Patrick von Reth for his constructive feedback and Jens Schuessler for his proofreading and improvement of my orthography.

Last but not least, I want to thank Yasmin M¨ulhaupt for her feedback, encouragement and endless support while writing this thesis. Life is so much better with you.

Tobias Hartmann

February, 2014
Contents

Acknowledgments vii

1 Introduction 1
   1.1 Motivation .................................................. 1
   1.2 Structure of the thesis ..................................... 3
      1.2.1 Timeline .............................................. 3

2 Background information 7
   2.1 Dynamic compilation ..................................... 7
   2.2 Java Language ............................................ 8
   2.3 The HotSpot™ Java Virtual Machine .................. 9
      2.3.1 Overview ............................................ 10
      2.3.2 Interpreter ........................................ 12
      2.3.3 Dynamic compilation ............................... 12
      2.3.4 Tiered compilation ................................. 16
      2.3.5 Code cache ......................................... 17
      2.3.6 Code cache sweeper ................................. 18
      2.3.7 Serviceability Agent ............................. 20
   2.4 Nashorn JavaScript engine .............................. 20

3 Related work 23
   3.1 Maxine Virtual Machine ................................. 23
   3.2 Graal Compiler .......................................... 24
   3.3 Jikes Research Virtual Machine .................... 25
   3.4 Dalvik Virtual Machine ................................. 25
4 Design
  4.1 Segmented code cache ...................................................... 27
  4.2 Dynamic code heap sizes ................................................... 29

5 Implementation
  5.1 Segmented code cache ...................................................... 33
    5.1.1 Code cache .......................................................... 33
    5.1.2 Code cache sweeper .................................................. 37
    5.1.3 Serviceability Agent ................................................ 38
    5.1.4 Dynamic tracing framework DTrace ................................ 39
    5.1.5 Stack tracing tool Pstack .......................................... 40
    5.1.6 Adding new code heaps .............................................. 40
  5.2 Dynamic code heap sizes .................................................. 41
    5.2.1 Virtual space ....................................................... 41
    5.2.2 Code heap ............................................................ 44
    5.2.3 Code cache ........................................................... 46
    5.2.4 Lazy creation of method code heaps ................................ 47
    5.2.5 Serviceability Agent ................................................ 48
    5.2.6 Dynamic tracing framework DTrace ................................ 48
    5.2.7 Stack tracing tool Pstack .......................................... 48

6 Evaluation
  6.1 Experimental setup ...................................................... 49
  6.2 Default code heap sizes .................................................. 50
  6.3 Dynamic code heap sizes .................................................. 52
  6.4 Overall performance ..................................................... 53
  6.5 Sweep time ............................................................... 56
  6.6 Memory fragmentation .................................................... 58
  6.7 ITLB and cache behaviour ............................................... 61
  6.8 Hotness of methods ...................................................... 64
7 Conclusion

7.1 Future work ........................................... 67

A Appendix ................................................. 69

A.1 Additional graphs ...................................... 69
A.2 Benchmarks ............................................. 71
  A.2.1 Google Octane JavaScript benchmark ................ 71
  A.2.2 SPECjbb2005 ........................................ 71
  A.2.3 SPECjbb2013 ........................................ 71
A.3 New JVM options ....................................... 72
A.4 Used tools/frameworks ................................. 72
  A.4.1 Eclipse .............................................. 72
  A.4.2 Python .............................................. 72
  A.4.3 yEd Graph Editor ................................. 72

Bibliography ............................................. 72
1 Introduction

1.1 Motivation

The HotSpot\textsuperscript{TM} Java Virtual Machine (JVM) was subject to many changes during the last ten years of development. Past activities in optimizing the performance focused on the dynamic compilers and the supporting runtime. But since the dynamically compiled code is stored in a code cache to avoid frequent recompilations, the organization and maintenance of the code has a significant impact on the overall performance. For example, the bugs JDK-8027593\textsuperscript{1} and JDK-8020151\textsuperscript{2} report serious performance regressions due to the code cache taking the wrong actions.

The organization of the code cache became even more important with the introduction of tiered compilation in Java SE 7. Previously only the interpreter gathered profiling information before compiling a method with the server compiler [43], which then used the profiling information. Tiered compilation uses the client compiler to generate compiled versions of methods that collect profiling information. Methods compiled by the client compiler are potentially later compiled using the server compiler. This leads to faster startup and more precise profiling information. As a result, not only the amount, but also the number of different types of compiled code increased.

The current design of the code cache, however, is optimized to handle only one type of compiled code. The code cache is organized as a single heap data structure on top of a contiguous chunk of memory. To add new code to the code cache, the code cache allocates space independently of the type of code that needs to be stored. For example, profiled code which has a limited lifetime can be placed next to non-profiled code, which potentially remains in the code cache forever. Further, JVM internal structures, such as the interpreter or adapters that are used to jump from compiled to interpreted code, are mixed with compiled Java methods. Mixing different code types leads to different performance and design problems. For example, the method sweeper, which is responsible for removing methods from the code cache, must scan all code types while sweeping. This results in a serious overhead because some entries are never flushed or even contain non-method code.

The approach presented in this thesis addresses these issues at the structural level of the code cache. Instead of having a single code heap, the code cache is segmented into distinct code heaps, each of which contains compiled code of a particular type. Such a design enables to separate code with different properties.

\textsuperscript{1}The implementation of the method sweeper causes a performance regression for small code cache sizes.
\textsuperscript{2}Large performance regressions occur when the code cache fills up.
As described in the last paragraph, there are three different types of compiled code: (i) JVM-internal (non-method) code, (ii) profiled-code, and (iii) non-profiled code. This thesis evaluates an approach in which each of the before-mentioned code types is stored into an individual code heap. Available code heaps are:

- a **non-method code heap** containing non-method code, such as buffers and bytecode interpreter. This code type will stay in the code cache forever.

- A **profiled code heap** containing lightly optimized, profiled methods with a short lifetime and

- a **non-profiled code heap** containing fully optimized, non-profiled methods with a potentially long lifetime.

The main advantages of this design are:

- **Efficient sweeping:** It is possible to skip non-methods through specialized iterators, sweep methods with shorter lifetime more frequently and easily parallelize the sweeping process.

- **Improved code locality:** Code of the same type is likely to be accessed close in time. As a result the instruction cache and the instruction translation lookaside buffer (TLB) misses are reduced.

- **Fine grained locking:** It is possible to use one lock per code heap or code type respectively, instead of locking the entire code cache for each access. Fine grained locking enables both fast sweeping as well as parallel allocation in different code heaps.

- **Improved management of heterogeneous code:** Future versions of the HotSpot™ JVM may include GPU (Project Sumatra\(^3\)) or ahead-of-time (AOT)-compiled code that should be stored in a separate code heap. Furthermore, the segmented code cache helps to better control the memory footprint of the JVM by limiting the space that is reserved for individual code types.

On the other hand, fixing the size per code heap has disadvantages: The required size for non-method code, such as adapters, the bytecode interpreter or compiler buffers, does not only depend on client code, but also on the machine architecture and the JVM settings. Additionally, the size needed for profiled and non-profiled code, respectively, depends on the amount of profiling that is done which further depends on the application, runtime, and JVM settings. During the startup phase of an application, more space for the profiled code heap is needed. After profiling of hot code is completed, the hot methods are compiled by the C2 compiler and less space for the profiled methods is needed. It is therefore difficult to set default values for the corresponding code heap sizes.

---

\(^3\)Enables Java applications to take advantage of graphical processing units (GPUs). For detailed information see [http://openjdk.java.net/projects/sumatra/](http://openjdk.java.net/projects/sumatra/)
1. Introduction

To solve these issues the presented approach allows to dynamically resize the method code heaps according to runtime requirements. A long running application would, for example, first increase the size of the profiled method code heap because a lot of profiling is done at application startup. Later, when enough profiling information is gathered, the size of the non-profiled code heap is increased. Additionally, the method code heaps are created lazily at runtime, such that the size of the non-method code heap is fixed after JVM initialization.

The contributions of this thesis are (i) the design and implementation of a dynamically sized segmented code cache and (ii) a complete documentation and evaluation of the system regarding performance, runtime behaviour and memory consumption. Change (i) is delivered as two incremental patches to the code in the HotSpot™ JVM mercurial repository changeset 5765:9d39e8a8ff61 from 27 December 2013. The patches fix bug JDK-8015774: Add support for multiple code heaps [29] in the JDK Bug System.

1.2 Structure of the thesis

Chapter 2 provides technical background information needed for this thesis. After a short overview of dynamic compilation and the Java language implementation, this section presents the HotSpot™ Java Virtual Machine in detail. The description focuses on those modules affected by the changes introduced with this work.

Chapter 3 presents related work and compares the runtime systems to the solutions presented in this thesis.

Chapter 4 introduces the high-level design of the segmented code cache and the dynamic code heap sizes. Important design decisions, e.g., the code heap design and memory layout, are described and justified. This chapter builds the basis for the implementation described in Chapter 5.

Chapter 5 describes the implementation of the segmented code cache and the dynamic code heap sizes based on the design decisions presented in Chapter 4. The development steps are listed chronologically and illustrated with examples.

Chapter 6 provides a complete evaluation of the system using various benchmarks executed on different platforms. The segmented code heap and the dynamic code heap sizes version are compared to the baseline version.

Chapter 7 summarizes the results of this thesis and suggests future work.

1.2.1 Timeline

In this section the development steps are presented chronologically to give an overview of the changes to the baseline version, which where necessary to implement the system. The first part lists the changes for the segmented code cache and the second part lists the changes for the dynamic

4http://hg.openjdk.java.net/hsx/hotspot-comp/hotspot
resizing of code heaps. Only major progress is described, minor changes like bug fixing, refactoring, and adaptations according to reviews by Oracle are omitted.

**Segmented code cache**

- **Code heap management**: Implement and adapt the necessary functionality like macros, iterators and array structures to manage multiple code heaps in the code cache.
- **Code cache interface**: Changes to the code cache interface to support multiple code heaps. Changes to client modules, such as the compile broker.
- **Code blob types**: Integration of code blob types to assign compiled code blobs to the corresponding code heaps.
- **Code heap layout**: Definition of one code heap per code blob type (methods, buffers, adapters, runtime stubs and singletons). Code heaps are still defined statically using macros.
- **Code cache sweeper**: Changes to the code cache sweeper to support multiple code heaps. Only sweeping method code heaps.
- **Code heap**: Changes to implementation of code heaps to support explicit setting of a reserved code space. Now splitting one chunk of space into parts and distributing them among the code heaps to ensure a contiguous memory layout.
- **Final code heap layout**: Code heaps are now defined dynamically at runtime. Depending on the JVM configuration, such as tiered compilation and used compilers, a non-method code heap, a profiled method code heap and a non-profiled method code heap is used.
- **Memory service**: The memory service used for JVM monitoring and management support is modified to be able to register multiple code heaps.
- **JVM options**: New command line arguments for setting the code heap sizes.
- **Serviceability Agent**: The Java classes of the Serviceability Agent (see Section 2.3.7) are adapted to support multiple code heaps.
- **External tool support**: Changes to the DTrace (see Section 5.1.4) scripts and the Pstack (see Section 5.1.5) support libraries for Solaris and BSD to support multiple code heaps.
- **Optimizations and bug fixes**: Remove direct code heap references needed for the Serviceability Agent and tool support. Now directly referencing the code heap array and using specialized iterator classes to select individual code heaps.
- **Evaluation**: Detailed evaluation of the system with respect to performance and other runtime characteristics, using a set of Python scripts and multiple benchmarks suites.

**Dynamic code heap sizes**
1. Introduction

- **Virtual space**: Changes to the virtual space class to support growing downwards. Integration of functions to expand, shrink and grow into another virtual space.

- **Code heap**: Changes to internal implementation of code heaps to support virtual spaces that are growing down. Add functionality to shrink and grow into another, adjacent code heap.

- **Dynamic code heap sizes**: Implement dynamic resizing of profiled and non-profiled method code heaps if allocation in the code cache fails. Add corresponding JVM option.

- **Serviceability Agent**: Adaptation of the Java classes of the Serviceability Agent to support code heaps that grow down.

- **External tool support**: Fix the DTrace scripts and Pstack support libraries for Solaris and BSD.

- **Optimizations and bug fixes**: Add additional asserts and comments to make the code more readable. Disabling dynamic resizing if tiered compilation is disabled.

- **Lazy method code heaps**: The method code heaps are lazily created at runtime when the first method is allocated in the code cache. This avoids setting the size of the non-method code heap statically.

- **Evaluation**: Detailed evaluation of the system compared to the segmented code heap version and the baseline version.
2 Background information

This chapter provides background information needed to understand this thesis. After a short introduction to dynamic compilation, the Java language implementation is described. Language features like virtual calls and garbage collection are explained and compared to other languages. Section 2.3 describes the HotSpot™ Java Virtual Machine (JVM), that implements the Java Virtual Machine Specification [27]. The focus is laid on modules affected by the changes introduced with this work. Finally, Section 2.4 presents the Nashorn JavaScript engine, a pure Java runtime for JavaScript running on top of the HotSpot™ JVM.

2.1 Dynamic compilation

To explain the term dynamic compilation, the counterpart static compilation is presented first. Static or ahead-of-time compilation describes the process of creating a native executable from source code. Most work is therefore done before execution. Complex and time consuming analyses can be performed because they do not affect the runtime performance. The information is then used to highly optimize the code. Further, static compilation introduces no additional startup time and memory consumption because compilation is already done. Static compilers mostly use static information, for example, type information, but may also rely on profiling information that was gathered during previous runs of a compiled version of the program.

Statically compiled executables are often required to run on different platforms. It is therefore impossible to use platform specific features like vectorization instructions (e.g., SSE instructions) because at compilation time it is not known if the target platform supports these features. Further, static compilers must ensure that all assumptions always hold at runtime. If an highly optimistic assumption is violated at runtime, the program may crash or deliver an invalid result. Some static compilers are, in general, conservative and effective optimizations are not done. For example, inlining of virtual functions is only reasonable if the static compiler can prove that the target class of the call does not change at runtime, which is hard or even impossible to determine statically [4].

Dynamic or just-in-time (JIT) compilation tries to solve some of these issues by performing compilation at runtime while the program is being executed. For example, the Java runtime environment first compiles the Java source code to an intermediate, platform independent representation (Java bytecode), which is then delivered to the target platform (Section 2.2 and Section 2.3.3). The JVM executes Java bytecode by interpreting the bytecode or compiling Java bytecode to machine code.
on the fly. This allows to only partly compile the code by specifically selecting methods that are heavily used and highly optimize them. Dynamic compilers can use program information collected at runtime to specialize code. They may instrument the code to gather detailed runtime profiling information, for example, the number of invocations, runtime types and branches taken. This information can then be used to generate highly optimized machine code.

In contrast to static compilers, dynamic compilers can also perform highly optimistic optimizations and simply recompile if the assumptions on which these optimizations are based do not longer hold (called deoptimization, see Section 2.3.3). Examples of optimistic optimizations are global optimizations like inlining of library functions or removing unnecessary synchronization. Further, it is possible to fine tune the code for a particular platform by, for example, using vectorization operations specific to the available CPU.

The main disadvantage of dynamic compilers, however, is the startup delay that is caused by loading and initializing the supporting runtime and the initial compilation of code. Since compiling all methods at runtime causes an unacceptable delay for many applications, the JVM typically starts with interpreting bytecode. Only hot code is compiled to machine code, which provides high performance. In general, dynamic compilers must find a trade-off between compilation time and quality of the generated code. Additionally, an application executed by a dynamic compiler has a higher memory consumption (footprint) than a statically compiled program because the dynamic compilers and the supporting runtime consume memory at runtime.

More information about the implementation of the dynamic compilers in the HotSpot™ Java Virtual Machine can be found in Section 2.3.3.

2.2 Java Language

Java is a general-purpose, object-oriented and platform independent language originally developed by Sun Microsystems (now Oracle). As of early 2014, Java is the second most popular programming language (according to [1] and [2]) with more than 9 million developers [40] only outreached by the low-level language C.

Java was initially designed for interactive television, but became popular by allowing to execute untrusted applications (so called Java applets) from the world wide web inside a sandbox enforcing network- and file-access restrictions. Today, Java is mostly used for business applications because of the variety of frameworks available and the platform independence that simplifies and reduces the costs of software development.

Java is designed to be easy to learn by using a simple object model and a concise C++ - like syntax. It is object-oriented, supports modular and reusable code, and is extensible by allowing classes to be explicitly loaded at runtime or whenever needed. Java supports various security features that are part of the design of the language and the runtime, to allow untrusted code to be executed in a sandbox. Because Java is statically typed, the compiler catches many errors already at compile time where other languages would only fail at runtime.
2. Background information

A Java application is compiled to a `.class` file, which is the binary format for the Java Virtual Machine (JVM). The JVM interprets or dynamically compiles the `.class` file to native code (see Section 2.3). Java is platform independent, both at the source and the binary level, allowing the same program to be executed on different systems. For example, personal computers and mobile devices. The slogan "Write once, run anywhere" (WORA)\(^1\) emphasizes this platform-independence.

The performance of programs written in Java greatly exceeds the performance of purely interpreted languages like Python\(^2\), but can be slower than programs written in C or C++, which are compiled to native machine code. This performance difference might not only come from dynamic compilation. Other Java language features such as array bound checks have a performance impact as well. However, modern JVMs, e.g., Oracles HotSpot\(^\text{TM}\) Java Virtual Machine (see Section 2.3), use sophisticated dynamic compilers that reach a performance comparable to C/C++ applications by making use of optimizations that are only possible at runtime. For example, in contrast to C++ where virtual calls are expensive at runtime and are therefore avoided whenever possible, Java embraces virtual methods. The JVM therefore has to implement them to be fast, using techniques like inlining.

Despite that, Java applications can use functions or libraries written in other languages, such as C or Assembly, through the Java Native Interface. Hence, computationally intensive parts can be hand written and optimised in a platform-specific language.

All these features are described in the Java Language Specification [13] and are implemented by the Java Platform, Standard Edition (Java SE). Because Java has no formal standardization, for example by the International Organization for Standardization (ISO), the Oracle implementation is the de facto standard. It consists of two different distributions: (i) The Java Runtime Environment (JRE) containing the supporting runtime needed to execute Java programs and (ii) the Java Development Kit (JDK) containing development tools like the Java-to-bytecode compiler.

The main part of the Java SE is the platform specific JVM that executes the platform independent bytecode and implements all features defined by the Java Language Specification. Because the specification is abstract, different implementations of the JVM are possible.

2.3 The HotSpot\(^\text{TM}\) Java Virtual Machine

This section presents the design and implementation of the HotSpot\(^\text{TM}\) Java Virtual Machine (JVM), the reference implementation of the Java Virtual Machine Specification [27] developed by Oracle Corporation.

The state of the art described here serves as the baseline version for this thesis and the design and implementation presented in Chapters 4 and 5, respectively. More information about the HotSpot\(^\text{TM}\) Java Virtual Machine is available in [41].

\(^1\)Slogan used by Sun Microsystems to emphasize the cross-platform benefits of the Java programming language.
\(^2\)A widely used high-level programming language that focuses on readability and fast development.
2.3.1 Overview

The HotSpot™ Java Virtual Machine is the code execution component of the Java platform and the core component of the Java Runtime Environment. The HotSpot™ JVM is responsible for executing Java applications and available on many platforms and operating systems. Supported platforms include Sun Solaris, Microsoft Windows and Linux on IA-32 and IA-64 Intel architectures, Sparc, PPC and ARM.

The HotSpot™ JVM is mainly written in C++ and the source code contains approximately 250,000 lines of code [35]. Figure 2.1 shows the HotSpot™ JVM in the context of the main components of the Java Runtime Environment.

Java source code in the form of .java files is processed by the static Java compiler javac and compiled to platform-independent Java bytecode. In this process the semantics of the Java language is mapped to bytecode instructions that are stored in a standardized form in .class files. Because each bytecode occupies one byte, there exist 256 possible bytecodes. Currently, the Java bytecode instruction set uses only 205 bytecodes, the remaining bytecodes are used internally by the JVM or, for example, by debuggers to set breakpoints.

![Figure 2.1: Overview of the components of the Java Runtime Environment.](image)

Bytecode always resides within a method and a method is always contained in a class. Because a class file contains the bytecode of exactly one class, multiple class files can be packaged into a Java Archive by the Java Archiver jar. The HotSpot™ JVM then loads the class or jar files, verifies the bytecode correctness and executes the bytecode by interpreting or using a dynamic compiler (see the following sections for details).

Because the JVM executes bytecode and is therefore independent of the source programming language, it is possible to run other languages on top of the JVM by implementing a compiler that compiles the source language to Java bytecode. Originally, the JVM instruction set was statically typed, making it hard to run dynamic languages. The Da Vinci Machine Project by Oracle aims to support dynamic languages by adding a new invokedynamic instruction to allow method invocation with dynamic type checking (see "the Da Vinci Machine Project - a multi-language renaissance for the Java Virtual Machine architecture" [34]). For example, jython, a Python programming
language interpreter, generates Java bytecode from Python code (see Figure 2.1) that is executed by the JVM.

Figure 2.2 shows the internal architecture of the HotSpot™ JVM. Important subsystems, such as garbage collection and the heap- and stack management are omitted for simplicity.

![HotSpot™ Java Virtual Machine architecture](image)

Figure 2.2: Overview of the HotSpot™ Java Virtual Machine architecture.

The class loader is responsible for dynamically loading a class when it is first referenced at runtime. The bytecode verifier performs checks to ensure that the bytecode is safe and does not corrupt the JVM. For example, it checks that a branch instruction always targets an instruction in the same method.

The JVM can then execute the program by interpreting or dynamically compiling the bytecode to machine code, using one of two available dynamic compilers (see Section 2.3.2 and Section 2.3.3, respectively). To decide if a method is compiled, the interpreter gathers profiling information at runtime. Methods that are used extensively, so called hot spots, are scheduled for compilation whereas cold methods are interpreted (this is where the name HotSpot™ comes from). Compiling only hot methods pays off because often 90% of the execution time of a computer program is spent executing 10% of the code\(^3\) and therefore the JVM can highly optimize these 10% of the code.

Finally, the security manager checks the behaviour of the application against a security policy and stops the program in case a prohibited instruction is executed. Typically, web applets are executed with a strict security policy to ensure safe execution of untrusted code. The JVM serves as an

\[^3\]Also known as the 90/10 law, an application of the Pareto principle to software engineering.
additional layer between the operating system and the underlying hardware.

The HotSpot™ JVM provides several command line options and environment variables to control the runtime behaviour and enable or disable functionality. For example, the size of the code cache (see Section 2.3.5) can be set by passing `-XX:ReservedCodeCacheSize=` at JVM startup. The work presented in this thesis adds additional command line options introduced in Chapter 5 and described in Section A.3 of the Appendix. A full list of options that are available for the baseline version can be found at [32].

The following sections describe components of the JVM that are important for the work presented in this thesis.

### 2.3.2 Interpreter

The bytecode interpreter of the HotSpot™ JVM is a template-based, artificial stack machine. The interpreter iterates over the bytecodes executing a fixed assembly code snippet for each bytecode.

The interpreter is generated at startup by the JVM by using a template table and stored in the code cache (see Section 2.3.5). This table contains a template, i.e., a description of each bytecode and the corresponding assembly code. The assembly code snippets are compiled to machine code while the interpreter is loaded and then executed when the corresponding bytecode is encountered. For complex operations that are hard to implement in assembly, like a constant pool lookup, the interpreter calls into the JVM runtime.

This approach is faster than a classic switch-statement (see Section "Interpreter" in [42]) and enables sophisticated adaptations of the interpreter to specific processor architectures, speeding up the interpretation. For example, the same application can be executed on an old processor architecture as well as making use of features of the newest processor generation.

The interpreter performs basic profiling. Method entries and back-branches in loops are counted and dynamic compilation is triggered on counter overflow. This ensures that "hot" methods are identified and compiled to machine code. Such a two-tiered execution model is characterized by a fast startup and reaches peak performance when hot methods are compiled to optimized machine code.

Additional data, such as branch and type profile information, may be gathered to support optimizations in the dynamic compilers. Further, the warm-up period caused by the initial interpretation allows the dynamic compilers to make better optimization decisions. After initial class loading, dynamic compilers can base their optimizations on a more complete class hierarchy.

### 2.3.3 Dynamic compilation

This section presents the dynamic compilation system of the HotSpot™ JVM. Dynamic compilation, in general, is introduced in Section 2.1.

The constant pool is a set of constants used by a type. For example, string and integer constants.
In the HotSpot™ JVM dynamic compilation enables fast application execution as well as efficient runtime profiling. Figure 2.3 shows the state transitions of code in the JVM. First, a method \( m() \) is interpreted. If \( m() \) is hot, the JVM decides to dynamically compile it. Because profiling in the interpreter is slow and only limited information is available, in a first step the dynamic compiler may add profiling code to the compiled version to gather detailed information while the code is executed (see Section 2.3.4). If the profile information later suggests that the method is still executed a lot, it is re-compiled without profiling.

The profile information gathered in the first step enables optimistic optimizations by making assumptions that may be violated at runtime. For example, information about common call targets can be used for optimistic inlining\(^5\). Optimistic inlining results in a substantial performance gain for Java applications, since virtual calls are used frequently.

Code generated by the dynamic compilers is stored in a code cache (see Section 2.3.5) to avoid recompilation of already compiled methods. If compiled code calls a not-yet compiled method, control is transferred back to the interpreter.

If an optimistic assumption is violated at runtime, the corresponding optimization becomes invalid and the compiled code is invalidated. If the code is currently being executed, the JVM has to stop execution, undo the compiler optimizations and transfer control back to the interpreter. This procedure is called deoptimization. In general, deoptimization is hard because the JVM needs to reconstruct the interpreter state from already compiled code. Hence, the compiled code contains metadata for deoptimization, garbage collection and exception handling. Detection and deletion of such invalidated code is done by the code cache sweeper (see Section 2.3.6).

If a method contains a long running loop, it may never exit but still get hot. To identify such methods and make sure that they are eventually compiled, back branches are counted. If a threshold is reached, the method is replaced with a compiled version while running. To achieve this, the stack

---

\(^5\)If the JVM notices that the target of a virtual call is always the same at runtime, the JVM may optimistically inline it. The inlining is based on the assumption that the call target will indeed always be the same and may become invalid if the assumption is violated at runtime (e.g., if a new class is loaded).
frame of the compiled method is initialized to the interpreted stack frame and control is transferred to the compiled version at the next back branch. This process is called on-stack replacement (OSR).

The HotSpot™ JVM has two dynamic compilers. The client compiler, also called C1, generates lightly optimized code and has a small memory footprint. C1 is fast and therefore good for interactive programs with graphical user interfaces. The server compiler, also called C2, is a highly optimizing compiler. C2 generates code of higher quality than C1 at the cost of a longer compilation time. C2 is good for long running server applications where startup and reaction time are less important.

Traditionally, only one dynamic compiler was used and the user had to choose between a server or a client JVM. In version 7 of the JDK Tiered Compilation was introduced, supporting multiple levels of compilation where both compilers are used (see Section 2.3.4). The client and the server compiler are described in more detail in the following sections. Additional information can be found in [20].

2.3.3.1 Client compiler

The client compiler (C1) is a dynamic compiler designed for a low startup time and a small memory footprint. Because of its high compilation time, C1 is used for interactive applications. C1 implements simple optimizations what leads to a lower peak performance of the generated code, compared to the server compiler.

![Compilation phases of the client compiler](image)

Figure 2.4: Compilation phases of the client compiler. The platform-independent phases are colored blue, whereas the platform-dependent parts are brown.

The compilation of a method is divided into three phases (see Figure 2.4). First, the platform-independent front-end analyses the bytecode by abstract interpretation and builds a high-level intermediate representation (HIR). The HIR consists of a control flow graph and is used to perform several high-level optimizations, for example, constant folding and null check elimination. The optimizations aim at improving local code quality. Only few global optimizations are performed. C1 uses method inlining as well, but not as aggressive as it is done by the server compiler (see Section 2.3.3.2).

In the second phase, a low-level intermediate representation (LIR) is generated from the HIR by the platform-specific back end. The LIR is similar to machine code, but partly platform-independent. Some low-level peephole optimizations, such as combining of operations, are performed and virtual registers are assigned to physical registers. Finally, in phase three, machine code is generated by
traversing the LIR and emitting instructions.

The client compiler supports adding of profiling instructions to the LIR. This profiling code is then executed at runtime and gathers profiling information about methods, similar to the profiling performed in the interpreter. This feature is used with tiered compilation (see Section 2.3.4) to get a more accurate runtime profile for the server compiler.

More detailed information about the client compiler can be found in [20] and [45].

### 2.3.3.2 Server compiler

The server compiler (C2) is a dynamic compiler designed to generate highly optimized code that reaches peak performance. C2 is slower than the client compiler and therefore more suitable for long running applications, where the compile time is paid back by faster execution and a slower startup is negligible.

The phases of the server compiler include parsing, aggressive optimization, instruction selection, global code motion, register allocation, peephole optimization and code generation. An intermediate representation (IR) based on the static single assignment form (SSA) is used for all phases. Register allocation is implemented with a graph coloring algorithm [43] that is slower than the linear scan algorithm of the client compiler, but produces better results especially for large register sets found in many modern processors.

The server compiler tries to optimize aggressively by making use of detailed profiling information gathered by the interpreted or profiled code. This includes class hierarchy aware inlining, including optimistic inlining of virtual calls that may trigger deoptimization in case the assumptions do not hold (see Section 2.3.3). Other optimizations are global code motion, loop unrolling, common subexpression elimination and global value numbering. Also Java specific optimizations are performed, for example, elimination of null- and range-checks [31].

Optimistic inlining is one of the most effective compiler optimization. It not only expands the scope of the compiler, and therefore enhances the efficiency of other optimizations, but also improves the performance of virtual calls because some of the virtual calls can be converted to static calls. In contrast to other programming languages, such as C++, Java embraces the use of virtual calls. It is therefore important that virtual calls are optimized to be fast. A Class Hierarchy Analysis (CHA) [7] determines if virtual calls can be converted to a static call and therefore be inlined. If a new class is loaded at runtime, the CHA is adjusted and a deoptimization is performed if the call is no longer static. In this case an inline cache (IC) [18] is used to speedup the virtual call. If the call target is not cached in the inline cache, a (slow) runtime lookup by the JVM is needed.

More detailed information about the server compiler can be found in [43].

---

6A property of the intermediate representation that allows each variable to be assigned only once. It simplifies and improves many compiler optimizations. [11]
2.3.4 Tiered compilation

Although there exists two dynamic compilers, originally only one of them was used at runtime. With Version 7 of the JDK, Tiered Compilation was introduced to get the best of both compilers. The JVM uses the interpreter and C1 compiler for fast startup and profiling, and the C2 compiler to eventually reach peak performance with highly optimized code.

Tiered compilation executes code at different "tiers", in the following called execution levels. In addition to the interpreter gathering profiling information, the client compiler is used to generate compiled versions of methods that collect profiling information. The main advantage is a faster execution during the profiling phase, since the C1 compiled code is considerably faster than interpreted code. Further, the compiled versions provide more accurate profile data that can be used by the server compiler to re-compile the code with sophisticated optimizations. A consequence of tiered compilation is that more code is generated than in a non-tiered setting. More code requires more space in the code cache.

Figure 2.5 lists the execution levels. At level 0 only the interpreter is used and no code is compiled by the dynamic compilers. Level 1 uses the C1 compiler with full optimization. Level 2 and level 3 use the C1 compiler as well, but different amounts of profiling code is added, leading to a decreased performance. In general, level 2 is faster than level 3 by about 30%. Finally, level 4 uses the C2 compiler. The C2 compiler does not instrument compiled code.

Different transitions between the execution levels are possible. A policy decides the next level for each method depending on runtime information, for example, the number of compilation tasks that are currently waiting for processing by the C1 and C2 compilers, profiling information and thresholds. The dotted arrows in Figure 2.5 show the most common transition: Execution starts with the interpreter, then the policy decides to compile the method at level 3. After profiling is completed the transition is made to level 4 where the method is fully optimized and remains until it is removed from the code cache (see Section 2.3.6).

In general, tiered compilation performs better than a single dynamic compiler. The startup time may be even faster than with the client compiler because C2 compiled code may be already available.
during the initialization phase. Because profiling is a lot faster with C1 compiled code, a longer profiling phase is possible resulting in a more detailed runtime profile. This potentially leads to better optimizations by the server compiler and a better peak performance compared to a non-tiered compilation.

### 2.3.5 Code cache

The compiled code is stored in a code cache for later reuse. The code cache also contains JVM internal structures that are generated at startup, such as the bytecode interpreter (see Section 2.3.2) or support code for transitions between the Java application and the JVM.

![Simplified architecture and interface of the code cache and related components as implemented by the corresponding C++ classes.](image)

The code cache is a core component of the JVM, being directly referenced from more than 150 places in the source code and indirectly through the Serviceability Agent (see Section 2.3.7) and supporting tools. Figure 2.6 shows the simplified architecture and interface of the code cache and related components.

The main part of the code cache is the **CodeHeap**, a heap-based data structure that provides functionality for allocating and managing blocks of memory. Internally the code heap uses memory segments of fixed size that are linked together. The code heap maintains a free list to reuse deallocated blocks and implements basic operations, like iteration and searching. The underlying memory is managed by a **VirtualSpace** that allows committing of a reserved address range in smaller chunks. Committing is the process of backing up previously reserved memory with physical memory. This reserved memory is represented by a **ReservedSpace** that basically contains the size and alignment constraints and abstracts the allocation of memory by the operating system.

To summarize, the code cache consists of a heap data structure on top of a contiguous chunk of memory that is reserved on JVM startup and then committed in small chunks when needed.

The code heap contains all different types of code. Different code types are abstracted by a
CodeBlob. This includes, for example, methods compiled with different optimization levels, so called nmethods and non-methods like buffers and adapters. Buffers are used by the compilers to temporarily store the generated code, whereas adapters are code snippets that transfer control from compiled code to interpreted code or vice versa.

The interface of the code cache abstracts the underlying implementation by providing high-level functions to allocate, deallocate and iterate over CodeBlobs and gather statistics, such as the capacity or free space. The code cache is referenced by many components, for instance, the sweeper (see Section 2.3.6) or the compile broker, which manages the compilation of methods.

Because the size of the code cache is fixed and cannot change at runtime, the code cache can get full and there is not enough memory left to allocate space for new methods. If this is the case, compilation is disabled and the code cache sweeper starts removing methods. If enough memory is released, compilation is re-enabled.

Detailed information about the implementation of the code cache and its components is presented in Chapter 5.

2.3.6 Code cache sweeper

The code cache sweeper cleans the code cache by removing methods. Compiled methods can become invalid for various reasons, for example, if an optimization is not longer valid due to class loading. Figure 2.7 shows the different states of a method in the code cache.

![Figure 2.7: State transitions of methods after compilation.](image_url)

After compilation the method is alive, meaning that the code is valid and can be executed. A method can then be made not entrant. A not entrant method cannot be called. This transition is necessary if the code is not longer needed or invalid and can be initiated by the following components:

- **the sweeper**: if the code cache is full (described below),

- **deoptimization**: if the optimized code becomes invalid because an assumption does not longer hold (see 2.3.3),

- **dependency invalidation**: dependencies are runtime assertions that may trigger deoptimization if violated due to dynamic linking or class evolution,

- **tiered compilation**: if the code is replaced by a different version (see 2.3.4).
If a method is made non entrant, it cannot be executed, but may still be active on the stack. Hence, the method cannot immediately be removed. To account for this, the code cache sweeper removes methods in several steps.

First, the code cache sweeper performs stack scanning and marks all methods that are active on the stack. If a non entrant method is not marked, and therefore is not active on the stack, it is converted to a zombie state. It is ”half dead” in the sense that it is not longer executed, but may still be referenced by inline caches (ICs).

In the second step, all inline caches that refer to zombie or not entrant methods are flushed and zombie methods change their state to marked for reclamation. If a zombie method that is marked for reclamation is encountered during the next sweeper iteration, it can simply be removed from the code cache because no inline caches refer to it.

Stack scanning happens at so called safepoints. At a safepoint the JVM stops all execution and Java threads cannot modify the Java heap or stack. Safepoints are also needed for garbage collection, deoptimization and various debugging operations. They are implemented by a global polling page that is frequently accessed by all threads and readable during normal operation, but made unreadable when a safepoint is requested. More information can be found in [15].

The code cache sweeper starts sweeping if at least one of the following conditions is met:

1. The code cache is getting full.
2. There is sufficient state change in the code cache since the last sweep. Currently, this is the case if more than 1% of the bytes in the code cache changed.
3. Enough time has passed since the last sweep. This time threshold depends on the size of the code cache and is only checked if methods are compiled. The smaller the code cache, the more often it is swept.

As mentioned above, apart from deoptimization, dependency invalidation and tiered compilation, the sweeper can decide to make a method not entrant. Making methods not entrant is especially important if the code cache is getting full and the sweeper must remove the least used methods to gain enough space to re-enable compilation. This process is performed during sweeping and based on the hotness of a method.

The hotness measures the utilization of a method. Initially, the hotness is set to a high value that is decremented every time the method is encountered by the sweeper and reset if a method is found on the stack of a Java thread. A hot method is likely to be frequently encountered on the stack and therefore maintains a high hotness value. On the other hand, a cold method will not be found on the stack and will therefore tend towards a low hotness value. Methods with a low hotness value are first considered for removal by the sweeper.
2.3.7 Serviceability Agent

In general, it is hard to debug native code running in the JVM or even debugging the JVM itself. For example, debuggers for the C++ programming language require debug information and are not aware of the internal memory structures used by the JVM. Java debuggers, on the other hand, are only able to analyse Java programs but not the underlying JVM.

The Serviceability Agent (SA) is a collection of Java APIs designed for internal usage by the JVM developers to analyse crashed JVMs. The SA is based on low level debugging primitives and works by reading the process memory of a JVM. The SA analyses the JVM internal data structures, for example, stack frames, garbage collection statistics and code cache entries.

![Diagram of Serviceability Agent](image)

Figure 2.8: The Serviceability Agent running on top of a JVM and attaching to a target JVM through an external tool.

The Serviceability Agent is almost completely written in Java and thus allows cross OS and cross CPU debugging. The Java classes are basically re-implementations of the HotSpot™ C++ classes and access the in-memory structures created by the target JVM. Hence, the SA does not rely on support code running inside the JVM but uses a native tool, such as `ptrace`, to read the process memory or directly parses core files. Figure 2.8 shows the overall architecture.

Detailed information about the Serviceability Agent can be found in [47] and [44]. Sections 5.1.3 and 5.2.5 describe the changes to the SA that where necessary to support the segmented code cache and the dynamic code heap sizes.

2.4 Nashorn JavaScript engine

Nashorn is a lightweight runtime for the scripting language JavaScript (JS). Nashorn is purely written in Java and runs on top of the HotSpot™ Java Virtual Machine. It allows Java developers to embed JavaScript in their Java applications, as well as execute independent JS applications using a command-line tool. Nashorn is developed by Oracle and released with Java 8. It is fully compatible with the ECMAScript [9], the standardized scripting language implemented by JavaScript.

Compared to Rhino, an open source JS engine developed by the Mozilla Foundation, the current version of Nashorn is around 2 to 10 times faster [25]. Nashorn replaces Rhino with Java 8.
The engine is based on the Da Vinci Machine [34], a project that aims to add support for dynamic languages to the JVM by providing additional features and bytecode instructions. Although JavaScript is originally interpreted, Nashorn does not implement an own interpreter. It directly generates bytecode that is executed by the JVM.

Nashorn makes heavy use of MethodHandles and the \texttt{invokedynamic} bytecode introduced by the Da Vinci Machine project and described in the Java Community Process (JSR) 292 [33]. This is necessary because in contrast to Java, which is statically typed, JavaScript is a dynamically typed language where the actual type of an object can only be determined at runtime. The \texttt{invokedynamic} bytecode can call methods with such a looser specification by allowing to customize the linkage between a call site and the method. Such a dynamic call site is linked just before its first execution, using MethodHandles to specify the actual behaviour. Detailed information and examples can be found at [39].

Since Nashorn runs on top of the JVM it can provide additional functionality for interoperating with the Java platform. For example, it is possible to create and manipulate Java objects from JavaScript or extend Java classes. Because it is shipped with the Java Runtime Environment, which is privileged code, it is designed for secure execution. Nashorn does not implement a web browser API like the HTML5 canvas or audio support.

Because Nashorn generates a lot of bytecode, it is mainly used for testing and benchmarking the implementation presented in this thesis (see Section 5 and 6).

Additional information about Nashorn can be found in [50] and [26].
3 Related work

This chapter presents related work that solves similar issues in a different context and compares the approaches to the solutions presented in this thesis.

3.1 Maxine Virtual Machine

The Maxine Virtual Machine (MVM) [24] is a Java Virtual Machine completely written in Java and developed by Oracle Labs. The MVM exploits advanced Java language features, like type safety, garbage collection, annotations and reflection and features a modular architecture. The code base is fully compatible with modern Java IDEs and the standard JDK.

The Maxine source code is translated to Java bytecode by the standard javac compiler. Instead of directly using a JVM to execute this bytecode, a so called boot image is generated. It contains a near-executable memory image of the running MVM consisting of a heap populated with class data and objects, and a code cache populated with MVM code compiled by Maxines optimizing compiler C1X. The boot image generator is a Java application using large parts of the Maxine code and is executed by an existing JVM, for instance, the HotSpot™ JVM. A small C program then maps the boot image into memory and calls the MVM entry point.

In contrast to the HotSpot™ JVM, the Maxine VM does not interpret bytecode, but always compiles method bytecodes to machine code on first method invocation. It uses the lightly optimizing, template-based baseline compiler T1X. Frequently executed (hot) methods are then compiled by the highly optimizing C1X compiler. Both compilers are written in Java with a small amount of assembly code and compiled at MVM build time. The C1X compiler is a Java-port of the C1 compiler of the HotSpot™ JVM.

Because Maxine does not use an interpreter, but dynamically compiles all bytecodes encountered, large amounts of machine code have to be stored. The code cache consists of three regions. The boot code region is unmanaged, i.e., not affected by garbage collection, and contains all machine code needed for MVM startup. The run-time region is unmanaged as well and stores the code generated by the optimizing compiler. The managed baseline region contains code generated by the T1X compiler.

Similar to the HotSpot™ JVM, baseline code compiled with the T1X compiler can become unused or obsolete if the corresponding method is recompiled by the optimizing C1X compiler. Code eviction
removes this code if allocation of new space fails, by using a semi-space garbage collection scheme. With this scheme, memory is split into a from-space and a to-space. All newly compiled code is allocated into the to-space. If it is full, garbage collection is triggered and the to-space becomes the from-space. All reachable code is then copied to the to-space and the code remaining in the from-space is removed. Newly compiled code is then again stored in the to-space.

The advantage of this code cache design is the fact that garbage collection implicitly compacts the to-space memory. In contrast to the code cache in the HotSpot™ JVM, no free list is necessary as new space is allocated incrementally at the top of the to-space (bump-pointer allocation). This design avoids free list management overhead, such as updating and merging of free blocks. Additionally, the fragmentation is essentially zero as the compiled code is not interspersed with free blocks.

The boot code region is similar to the non-method code heap introduced in Chapter 4, with the exception that it only contains ahead-of-time compiled MVM code. The baseline and the run-time regions are similar to the profiled and the non-profiled code heaps, but are not dynamically resized. The run-time region is unmanaged and therefore optimized code stays in the code cache forever, even if it is not longer used.

More information, including performance evaluations, can be found in [52] and [14].

### 3.2 Graal Compiler

The Graal compiler is an enhanced version of the C1X compiler from the Maxine code base. It is completely written in Java and uses a new high level intermediate representation [8] to improve the code quality. Graal tries to improve the performance of programs executed in a JVM to match server compiler or even native performance. By using Java language features, the compiler is highly extensible and implements sophisticated optimizations.

The Graal VM project aims to integrate the Graal compiler back into the HotSpot™ JVM to improve its extensibility and performance. The source code of the JVM is adapted to support the Graal compiler. Graal uses the existing JVM infrastructure, such as the garbage collector and object locking and is invoked by the CompileBroker similar to the C1 and C2 compilers. Graal does not implement an own code cache, but uses the code cache of the HotSpot™ JVM to store generated machine code.

To ease the implementation of programming languages on top of the Graal VM, a Java framework called Truffle [53] is provided that relies on abstract syntax tree (AST) interpretation. Graal performs a partial evaluation of the AST and generates optimized machine code. Because Graal and Truffle Java code is not part of the executed application, it should be treated specially. For example, the code that is part of the Graal compiler is not swept and stays in the code cache forever. It may be beneficial to account for this by creating an own code heap for this code in future versions of the HotSpot™ JVM (see also Section 7.1).
3. Related work

3.3 Jikes Research Virtual Machine

The Jikes Research Virtual Machine (JRVM) [21] is an open source Java virtual machine that is completely implemented in the Java programming language. Similar to the Maxine VM (see Section 3.1), Jikes is meta-circular, i.e., it relies on an external JVM for boot image generation and uses a small C program to load the image and boot up Jikes.

Jikes does not interpret bytecode but provides three dynamic compilers. A lightly optimizing baseline compiler is responsible for initially compiling bytecode, and is also included in the boot image. Jikes directly translates bytecodes to machine code, reducing the initial overhead of dynamic compilation. A JNI compiler processes native methods according to the Java Native Interface (JNI) and generates adapters to transition from Java code to native code and vice versa. The optimizing compiler uses an intermediate representation and performs sophisticated optimizations available in three levels. It is slower than the baseline compiler but generates code with high performance.

The adaptive optimization system [22] decides which compiler optimization level is used to compile a method. The compiled code is stored in a Java object, which basically is an array of machine instructions with additional metadata. A compiled method can become obsolete if it is no longer valid (e.g., due to classloading) or replaced by a new version. The stack scanning phase of the garbage collector sets obsolete methods to dead if they are not longer used. They are then removed by the next garbage collection cycle.

In contrast to the HotSpot™ JVM, the JRVM does not store the generated code in a contiguous memory region, but allocates Java objects to store the code. A specific feature of the JRVM is that all calls are performed indirect through the Jikes RVM Table of Contents (JTOC) for static methods or the Type Information Block (TIB) for virtual methods (see [23]). Hence, the memory space the objects are allocated into can be a moving space and the garbage collector defines the code management policies. The garbage collector ensures that compiled code that is not longer needed is removed and the free space can be reused. This design limits the amount of code that can be generated to the available amount of memory.

3.4 Dalvik Virtual Machine

The Dalvik Virtual Machine (DVM) is a Java virtual machine for the Android\textsuperscript{1} operating system. It uses its own bytecode format \textit{dex-code}, which is translated from the Java bytecode. Because it is used on mobile devices, the design emphasizes compactness, memory usage and security. Execution is done by a highly optimized and fast interpreter. Important OS and library code is statically compiled. With Android 2.2, a dynamic compiler was introduced that focuses on minimal memory usage and fast compilation.

Many dynamic compilers used in JVMs operate at method-level granularity, i.e., always compile entire methods. The disadvantages are long compilation delays and high memory usage. Further,
cold parts within the hot methods may be compiled as well. The HotSpot™ JVM places *uncommon traps* on cold paths that transfer control to the interpreter if executed. The Dalvik compiler uses trace granularity by identifying hot execution parts, compiling the corresponding blocks and storing them into a translation cache. In this way, only the hot parts of a method are compiled, resulting in lower memory usage, fast compilation and low reaction time. However, it requires more state synchronization with the interpreter because transfers between the compiled code and the interpreter occur frequently.

An own translation cache is used for each JVM (there are approaches to share the cache [16]). Similar to the run-time region of the Maxine VM (see Section 3.1), the translation cache is unmanaged. A contiguous memory space is reserved at DVM startup and bump-pointer allocation is used to allocate space for compiled code. If the code cache is full, it is flushed and populated again. For detailed information see the Dalvik source code\(^2\), especially the files `/vm/compiler/Compiler.cpp` and `/vm/compiler/codegen/x86/CodegenInterface.cpp`.

\(^2\)The Dalvik source code is available at https://android.googlesource.com/platform/dalvik/+/HEAD/
4 Design

This chapter explains the high-level design of the segmented code cache and the dynamic code heap sizes. Chapter 5 presents the detailed implementation based on these decisions.

The current design of the code cache is optimized to handle only one type of compiled code, although there are multiple types, created and accessed by different components of the JVM. The segmented code cache divides the code cache into distinct segments, each of which contains compiled code of a particular type with specific properties.

The dynamic code heaps allow the code cache segments to dynamically adapt their sizes to the runtime needs, reducing the limitations of having segments of fixed size. Further, it is now possible to lazily create the segments when they are first used, and set the size according to the runtime needs.

4.1 Segmented code cache

As described above, the code cache contains code with different characteristics. An intuitive design suggests to separate code blobs by putting each code type into a distinct code cache segment. There are multiple disadvantages with this approach. First, a default size for each segment has to be defined, which is hard because it is usually not clear how much memory each segment needs at runtime. Second, this approach increases memory fragmentation because there will be free space between the segments and there are types of code of which only a few actual instances exist at runtime (e.g., the deoptimization blob\(^1\)). Further, the code locality may decrease because of the highly separated code segments. The decreased code locality, together with the increased memory fragmentation, leads to a higher instruction TLB miss rate which additionally affects performance.

Taking these disadvantages into account, a more coarse grained segmentation of the code cache seems appropriate. The main distinguishing feature of a code segment is the separation of compiled methods and non-method code, such as runtime stubs and JVM internals. Method code is highly dynamic, has different compilation levels and lifetimes and makes up most of the code cache. Non-method code is more static, persistent and limited in size, occupying only around 2% of total size of the code cache.

\(^1\)An entry in the code cache written in assembly and used for deoptimization. On deoptimization, the return address of the corresponding compiled method is patched to redirect execution to the deoptimization blob.
For methods, one can further distinguish between profiled and non-profiled methods. Profiled methods are lightly optimized and have a limited lifetime, whereas non-profiled methods are highly optimized and possibly remain in the code cache forever. Therefore, the code cache is segmented into three parts, corresponding to the following three types of code:

- **non-method code**: non-method code, such as buffers and the bytecode interpreter,
- **profiled code**: lightly optimized, profiled methods and
- **non-profiled code**: fully optimized, non-profiled methods.

To define the memory needed for the code of each type at runtime, experiments with different benchmarks are performed (see Section 6.2). The code cache memory consumption highly depends on the architecture, JVM settings and application. For example, if tiered compilation is disabled, no profiled code is generated and the profiled code heap is not created. Additional JVM options are introduced to enable the user to explicitly set the reserved memory for each code type.

Per default, 5 MB are used for the non-methods and the remaining code cache size is distributed equally among the profiled and the non-profiled methods. Figure 4.1 shows the simplified memory layout of the segmentation from low to high addresses. Currently, the JVM does only support code cache sizes smaller than 2 GB\(^2\). To ensure that the maximum distance of two segments in the code cache does not exceed 2 GB, the segments are placed adjacent to each other in memory. The boundaries between the segments are fixed because currently the top level code heap data structures do not support resizing of their address spaces. Detailed information about how the memory layout is implemented can be found in Section 5.1.1.2.

The interface of the code cache is adapted to provide access to the code of a specific type. For example, instead of iterating over the entire code cache, it is now necessary to specify the type of code to iterate over.

As already stated in Section 2.3.5, the code cache is directly referenced from more than 150 places in the source code and indirectly through the Serviceability Agent. The components accessing the

\(^2\)The code cache size is limited to 2 GB because currently the assembler used by the JVM generates 32-bit immediates for control flow transfers, such as jumps and calls, and therefore only supports an address space of 2 GB.
code cache were not designed to support different types of code or a segmented code cache. The most important components that must be adapted are the `AdvancedThresholdPolicy` and the code cache sweeper.

The `AdvancedThresholdPolicy` class implements the tiered compilation policy and manages the transitions of a compiled method between the execution levels (see Section 2.3.4). It uses information about free space in the code cache to change the compile thresholds accordingly and to prevent the code cache from filling up too fast. The policy is modified to use the type of the method to determine the corresponding segment of the code cache and uses the free space in this segment instead of the overall free space in the code cache.

The code cache sweeper is adapted to skip non-method code by only processing the method code segments. This reduces the time that is needed for sweeping. Further, profiled methods, that have a shorter lifetime, can now easily be swept more frequently than non-profiled methods.

Future versions of the HotSpot™ JVM may include GPU or ahead-of-time (AOT) compiled code, making the code stored in the code cache even more heterogeneous. This is taken into account by an extensible design that easily allows to add new code types (see Section 5.1.6). Additional code can then be stored in a separate code cache segment.

Section 5.1 describes the implementation of the segmented code cache. The segments are implemented as multiple code heaps, heap-based data structures that provide functionality for allocating and managing blocks of memory.

### 4.2 Dynamic code heap sizes

One disadvantage of a segmented code cache is that in the original design, the size of the segments is fixed. The JVM is highly dynamic and static default values for the sizes are not always applicable. For example, a long running application will generate mostly profiled code in the beginning and use this profile information to generate highly optimized and non-profiled code that potentially stays in the code cache forever. This means that at the beginning there is a majority of profiled methods and later more space for non-profiled methods is needed. The code cache fills up even faster with small code cache sizes.

The same problem applies to non-method code. On the one hand, the memory space needed depends on JVM settings, such as if tiered compilation is used. The memory space depends on the architecture as well, such as the number of cores (more cores induce more compiler threads). On the other hand, it also depends on the application. Again, a static default value for the non-method code cache segment is hard to define.

One solution to this problem would be to dynamically allocate more code cache segments and deallocate segments that are not longer needed. Despite the fact that this would increase memory fragmentation, it is not always possible because the maximum distance between the segments must not exceed 2 GB (see Section 4.1).
Another solution uses an already existing code cache segment if one segment is full. For instance, if the segment for profiled methods is full, the non-profiled segment is used to store the additional profiled methods. This solution solves the problems, but extinguishes the advantages of the segmented code cache. Code of different types would be mixed again. It is also hard to implement because optimizations dependent on a segmented code cache would have to be reverted at runtime if code of different types is not longer separated.

The approach taken in this thesis is based on the idea to dynamically move the boundary between segments to expand one and shrink the other. The segments for non-profiled and profiled methods are adjacent to each other in memory. To be able to move the boundary between the segments, both segments need to fill up towards the boundary. This means that the non-profiled segment has to grow downwards and the profiled segment has to grow upwards. Figure 4.2 illustrates the memory layout of the code cache segments. The dotted arrows show the direction of growth.

![Figure 4.2: Memory layout of the code cache segments with dynamic code heap sizes. The dotted arrows show the direction of growth. The size of the non-method segment is lazily set at runtime.](image)

Initially, all segments are created with a fixed size as described in Section 4.1. If, for example, the profiled segment gets full and allocation fails, the non-profiled segment shrinks by moving the boundary towards higher addresses until there is enough space in the profiled segment.

The design is extensible and allows to change the layout of the code cache segments, for example, if new segments are introduced or the order of the existing ones needs to be changed.

The code cache segment for non-methods is a special case. It is not possible to move the boundary into the profiled segment at runtime because it grows towards the higher addresses. But because most non-method code, such as the interpreter and runtime stubs, is generated at JVM startup and compilation of methods starts afterwards, the profiled and non-profiled method segments are created lazily.

This means in detail that the non-method segment first occupies the entire memory space reserved for the code cache. When the first method is allocated, i.e., JVM startup is completed, the size of the non-method segment is fixed and the method segments are created using the remaining space. Because after the startup a small amount of non-method code is created as well, the non-method segment is fixed to its current size plus a buffer space to account for this. The size of the additional
buffer is equal to the JVM option `CodeCacheMinimumFreeSpace` (500 kB per default).

If the C2 compiler is enabled, additional space for the scratch buffers\(^3\) allocated at runtime is needed. The following formula is used to compute the space by using 1% of the memory reserved for the code cache (but at least 500 kB) and additional 128 kB for each compiler thread:

\[
\text{max}(500 \text{ kB}, \text{ReservedCodeCacheSize} \cdot 0.001) + (\text{CICompilerCount} \cdot 128 \text{ kB})
\]

Evaluation shows that the additional space is sufficient for non-method code generated after JVM startup. Hence, the JVM option to control the size of the non-method segment is needless and removed. A JVM option is introduced to enable or disable the dynamic resizing of code segments because there are scenarios where it is not needed. (for example, without tiered compilation).

Because the segments are now resized, the definition of free space that is available in a segment is refined. The possibilities for the profiled segment are illustrated in Figure 4.3. The `AdvancedThresholdPolicy` is changed to consider the space that is free in the entire code cache, instead of only one segment (Figure 4.3 (c)). The code cache sweeper is adapted to consider only the space in the current segment (Figure 4.3 (a)) because resizing is not always possible. For example, if the space at the boundary of the adjacent segment is occupied, resizing is not possible, even if there is a lot of free space available. Hence, the sweeper has to start sweeping. Figure 4.3 (b) shows a setting where only a part of the free space is taken into account. Evaluation shows that version (b) and (c) result in insufficient sweeping.

Figure 4.3: Possibilities to measure the available space in the profiled method segment. The memory space marked in white is free, the space marked in green is considered to be available for the profiled method segment.

\(^3\)A scratch buffer is a temporary buffer blob created by the C2 compiler to emit code into.
5 Implementation

This chapter presents the implementation of the two major contributions of this thesis in detail: the segmented code cache and the dynamic code heap sizes. The code version described here is based on the design decisions introduced in Chapter 4 and thoroughly evaluated in Chapter 6.

The changes are provided as two patches to the code in the HotSpot™ JVM mercurial repository\(^1\) changeset 5765:9d39e8a8ff61 from 27 December 2013, which is called baseline version in the following. The patch for the segmented code cache fixes bug JDK-8015774: *Add support for multiple code heaps* [29] in the JDK Bug System. The patch for the dynamic code heap sizes builds on these changes. An overview of the changes in chronological order is provided in Section 1.2.1.

The code of the HotSpot™ JVM is stored in the `src` folder. Paths to files listed in the following sections always start in this folder.

5.1 Segmented code cache

This section describes the implementation of the segmented code cache, including changes to other components that are necessary to support multiple code cache segments. The implementation of the code cache can be found in the file `/share/vm/code/codeCache.cpp`.

Section 4.1 presents the types of code and the three segments the code cache is divided into. The following sections describe the management and layout of these code segments, now called *code heaps* in detail. The adaptations include changes to other components, for example, the code cache sweeper and the Serviceability Agent. Further, the sections describe the changes to support-code for external tools that access the code cache.

As already described previously, extensibility is of utter importance since there will be new code types in the future. Section 5.1.6 therefore describes the integration of new code types and the corresponding code heaps.

5.1.1 Code cache

As described in Section 2.3.5 the code cache of the baseline version contains a single code heap for storing all code. There is no functionality to distinguish between code blobs of different types. For

\(^1\)http://hg.openjdk.java.net/hsx/hotspot-comp/hotspot
example, the function `CodeCache::allocate` takes the size as an argument and returns a pointer to the newly allocated space. The code cache does not know the code type that will be stored.

### 5.1.1.1 Code blob types

The first step is to keep track of the code types when new code is compiled and stored in the code cache. A struct `CodeBlobType`\(^2\) defines the types `MethodNonProfiled`, `MethodProfiled` and `NonMethod` for non-profiled methods, profiled methods and non-method code. The struct is then used throughout the code cache interface to select the appropriate code type and serves as an abstraction of the code heaps. For example, the iterator functions `CodeCache::first_blob` and `CodeCache::next_blob` now take a `CodeBlobType` as an argument and iterate over the corresponding code blobs or code heap, respectively. The same applies to other functions, such as allocation and deallocation of code blobs, where the destination code heap has to be specified.

To set the code type right from the beginning, the implementation of `CodeBlob`\(^2\) and `nmethod`\(^3\) is adapted to propagate the code type through the new operator to `CodeCache::allocate`. For non-method code, such as runtime stubs, the code type is simply equal to `NonMethod`. For methods the code type is determined by the compilation level that corresponds to the execution level described in Section 2.3.4. The function `CodeCache::get_code_blob_type` implements the translation between the compilation level and the code type by taking a compilation level and returning the corresponding code type.

### 5.1.1.2 Code heaps

As described in Section 2.3.5, the `CodeHeap`\(^4\) is a heap-based data structure that provides functionality for allocating and managing memory blocks. The baseline version of the code cache contains only one code heap for storing and retrieving all code.

To add support for multiple code segments, multiple code heaps must be created. A one-to-one relationship between the code types and the code heaps is established because code of a specific type should be stored in its own code heap. To support this relationship, fields for storing the name and code type are added to the implementation of the code heap. The function `CodeHeap::accepts` checks if a code heap stores code of the given type.

The code heaps are created in `CodeCache::initialize_heaps` during initialization of the code cache. To make sure that the maximum distance in memory between the code heaps does not exceed 2 GB (see Section 4.1), the underlying `ReservedSpace`\(^5\) is first created and then split into three parts using existing functionality. Each part is used to initialize the `VirtualSpace`\(^5\) of a code heap. Because in the baseline version each code heap creates its own `ReservedSpace`, the

\(^2\)Defined in `/share/vm/code/codeBlob.hpp`  
\(^3\)Defined in `/share/vm/code/nmethod.hpp`  
\(^4\)Defined in `/share/vm/memory/heap.hpp`  
\(^5\)Defined in `/share/vm/runtime/virtualspace.hpp`
implementation of the function `CodeHeap::reserve` is changed to take a `ReservedSpace` that is previously created. Figure 5.1 shows the overall picture.

![Layered structure of the segmented code cache](image)

**Figure 5.1:** Layered structure of the segmented code cache. `ReservedSpace` is created during startup, split up and used to initialize the `VirtualSpaces` of the code heaps. The `CodeBlobTypes` are used to access the code of a specific type residing in a specific code heap.

As described in Section 4.1 the following code heaps are created by default:

- a **non-method code heap** containing non-method code, such as buffers and runtime stubs,
- a **profiled code heap** containing lightly optimized, profiled methods, and
- a **non-profiled code heap** containing fully optimized, non-profiled methods.

If tiered compilation is disabled, the profiled code heap is not created because profiling is only done in the interpreter. The non-profiled code heap is expanded accordingly.

The JVM options\(^6\) `NonMethodCodeHeapSize`, `ProfiledCodeHeapSize` and `NonProfiledCodeHeapSize` are introduced to control the size of each code heap. Checks\(^7\) for consistency of code heap and code cache sizes are added to validate user provided values.

The code cache keeps track of the code heaps by storing them in a `GrowableArray`\(^8\), a dynamic extensible array data structure, very similar to a `vector` or `list` from the C++ Standard Template Library (STL). The code heaps are only used by the code cache and not directly accessible from outside. The internal function `CodeCache::get_code_heap` performs the mapping between a `CodeBlobType`, used in the interface, and the corresponding code heap, by iterating over the array and returning the code heap that accepts the code type.

All functions of the code cache are adapted to access multiple code heaps. For example, the `CodeCache::allocate` function now takes a code blob type as argument to allocate space for this type of code in the code cache. The function then invokes `CodeCache::get_code_heap` to get the corresponding code heap, allocates memory in this code heap and returns a pointer to the code heap.

---

\(^6\)Defined in `/share/vm/runtime/globals.hpp`  
\(^7\)Implemented in `/share/vm/runtime/arguments.cpp`  
\(^8\)Defined in `/share/vm/utilities/growableArray.hpp`
Custom iterators for the `GrowableArray` provide simple access to multiple code heaps. A `GrowableArrayIterator` is used to iterate over all entries of a `GrowableArray`, whereas a `GrowableArrayFilterIterator` iterates over elements that satisfy a given predicate. For example, a predicate may specify that the code heaps have to accept a given set of code blob types. The custom iterators implement the STL iterator interface\(^9\).

Currently, the `GrowableArrayFilterIterator` is used to iterate over all code heaps containing methods (i.e., the profiled and the non-profiled code heaps). To select the code heaps, a predicate called `IsMethodPredicate` is used that returns true for all code heaps accepting the method `CodeBlobTypes`. Additional predicates can easily be defined (see Section 5.1.6).

If one of the code heaps is full, memory allocation fails. This is noticed at several locations in the code and reported to the `CompileBroker`\(^10\), an intermediate component handling compilation requests. The `CompileBroker` disables the dynamic compilers until the code cache sweeper has freed enough space and prints a warning message. To provide detailed information about a full code heap, the JVM reports the code blob type for which the allocation failed. The `CompileBroker` forwards this information to the `report_codemem_full` function of the code cache, which prints the warning message containing the code heap that is full.

The code cache also provides functions to obtain statistical information about the capacity, in particular the maximum, unallocated and current capacity. In a segmented code cache, the statistical information can be computed for one code heap or for the entire code cache. Hence, existing functions are kept and additional functions are added to compute the values for one code heap by providing the corresponding code type.

The `AdvancedThresholdPolicy` class uses information about the free space in the code cache to change compile thresholds for tiered compilation. Adaptive thresholds prevent the code cache from filling up too fast. The policy uses the function `CodeCache::reverse_free_ratio` which returns the reverse of the free ratio of the code cache. E.g., if 25% (1/4) of the code cache is full, the function returns 4.

In a segmented code cache there can be space available in one code heap, even if all other code heaps are full. The compile thresholds must be set according to the free space that is available for the destination code type. If, for instance, a method moves from execution level 3 (C1, full profiling) to execution level 4 (C2), the thresholds should be set according to the space that is available in the non-profiled code heap. Consequently, `reverse_free_ratio` is modified to take the code type provided by the `AdvancedThresholdPolicy` and computes the reverse free ratio of the corresponding code heap. The function is also used by the code cache sweeper (see Section 5.1.2).

Finally, debugging functions like printing of status information (`CodeCache::print_internals` and `CodeCache::print_summary`) and verify functions (`CodeCache::verify`, self-verification functions only executed in debug builds) are adapted to work with a segmented code cache.

---

\(^9\)See http://www.cplusplus.com/reference/iterator/

\(^10\)Defined in /share/vm/compiler/compileBroker.hpp
5. Implementation

5.1.1.3 Optimizations

Some of the functions provided by the code cache only address method code. For example, first_nmethod, find_nmethod or nmethods_do iterate over compiled methods and ignore non-method code. Because the baseline version stores all code in one code heap, the code cache must iterate over all entries and skip non-method code.

Although non-method code makes up only around 2% of the code cache, skipping non-methods pollutes the source code with runtime checks and decreases the performance of method operations on the code cache. The code cache sweeper is mostly affected because it periodically scans all methods but must skip non-method code (see Section 5.1.2).

With the custom iterators the functions are changed to only iterate over code heaps that contain compiled methods by using the GrowableArrayFilterIterator in combination with the IsMethodPredicate. All runtime is_nmethod()-checks are removed from the code cache functions. To summarize, the sweeper now iterates over less code cache entries and performs no runtime checks. Section 6.5 evaluates the performance gain of these optimizations.

5.1.2 Code cache sweeper

As described in Section 2.3.6, the code cache sweeper is responsible for cleaning up the code cache. The sweeper scans the code cache, updates the states and hotness values of methods and removes methods not longer needed, invalid or if the code cache is full. Sweeping is done by compiler threads. To reduce the time spent sweeping one full traversal of the code cache is split up into several smaller iterations that eventually cover all methods. The number of invocations is controlled by the JVM option NmethodSweepFraction.

Because in the baseline version the sweeper sweeps a single code heap, the implementation is changed to support sweeping multiple code heaps. The sweeper scans all method code heaps, starting with the non-profiled code heap, and skips the non-method code heap (non-method code is not swept). A field is added to keep track of the current code type, i.e., the current code heap, and continue with this code heap in the next iteration.

Since it is guaranteed that only methods are encountered while scanning the code heaps, all is_nmethod-checks are removed. The fact that significantly less code cache entries are processes reduces the time spent sweeping the code cache (see evaluation in Section 6.5).

The function NMethodSweeper::possibly_sweep invokes the sweeper (i) if the code cache is getting full, (ii) if there is sufficient state change in the code cache compared to the last sweep or (iii) if enough time has passed since the last sweep. A formula is used that invokes the sweeper more often for small code cache sizes or if the code cache is getting full by making use of the CodeCache::reverse_free_ratio function. Because there are multiple code heaps, the maximum reverse free ratio of all code heaps is used. That means that the sweeper is invoked if one of the code heaps reaches its maximum capacity.
It may be more efficient to only sweep the fullest code heap or in general sweep the profiled code heap more often because profiled methods have a shorter lifetime. Such a selective sweeping mechanism is described in Section 7.1 and may be implemented in future versions.

When processing a method its hotness value is compared to a threshold value. If the hotness value is below the threshold, the method is set to not entrant and removed during the next iterations. The threshold value is computed using the reverse free ratio of the code heap the current method is stored in. The fuller the code heap, the bigger the threshold gets and the more methods are removed from this code heap.

5.1.3 Serviceability Agent

As described in Section 2.3.7, the Serviceability Agent (SA) is a collection of Java APIs and tools to debug the HotSpot™ JVM. The SA is based on low level debugging primitives and works by directly reading the process memory and analysing data structures, such as stack frames, garbage collection statistics and code cache entries.

The Serviceability Agent is executed in its own JVM and does not execute code in the target JVM. The SA relies on the VMStructs class\textsuperscript{11} that contains a table with descriptions of the classes and corresponding fields of the HotSpot™ JVM source code. The SA also contains processor and platform dependent declarations, for example, CPU registers and the memory size of different types. Most classes are declared with VMStructs as a friend, so that even private fields can be accessed. For example, there are entries that describe the fields of the mmethod class, allowing the SA to gather information about compiled methods.

The SA is almost completely written in Java and basically re-implements the C++ classes of the HotSpot™ JVM (the Java code can be found in /agent/src/share/classes/sun/jvm/hotspot/). These Java classes access fields declared in the VMStructs class at runtime. They are referenced by their name and read out of the memory space of the target JVM. For example, the main functionality of the code cache, such as searching for methods and iterating over the code, can be found in the Java class sun.jvm.hotspot.code.CodeCache. It builds upon the Java implementation of a code heap (sun.jvm.hotspot.memory.CodeHeap).

The SA is adapted to support a segmented code cache with multiple code heaps. First, the GrowableArray field of the C++ implementation of the code cache is added to the VMStructs class, to be able to access the array from Java code. During initialization, the SA reads the field and instantiates a local copy of this GrowableArray implemented in Java\textsuperscript{12}. The Java functions contains, findBlob and iterate are adapted to access the code heaps in the GrowableArray. Additional helper functions are added to extract the necessary information.

To test the adapted functionality, the HotSpot™ Debugger (HSDB) [3] is used. The debugger allows to list the compiled methods in the code cache and only succeeds if the Java classes can

\textsuperscript{11}Defined in /share/vm/runtime/vmStructs.hpp
\textsuperscript{12}Declared in sun.jvm.hotspot.utilities.GrowableArray
access the code heaps.

More information about the implementation of the SA can be found in [36] and [47].

5.1.4 Dynamic tracing framework DTrace

DTrace is a dynamic tracing framework developed to debug the operating system kernel and applications. DTrace is available for several operating systems, for example, Solaris, Mac OS X and FreeBSD. Tracing scripts written in the D programming language\(^{13}\) contain so called probes and associated actions that are executed if the conditions of a probe are met. For example, a probe can fire if certain functions are executed or a process is started in the profiled application. The probe may then examine the call stack or supplied arguments and print information.

The HotSpot\(^{TM}\) JVM provides probes that can be used in a D script to monitor the state of the JVM or running Java applications (a list of probes can be found in [38]). This includes probes for garbage collection, method compilation and class loading. Currently, only the Solaris and BSD operating systems are supported. The DTrace support code for these platforms can be found in \(/src/os/solaris/dtrace/\) and \(/src/os/BSD/dtrace/\), respectively. A description of the detailed implementation can be found in [36].

To enable DTrace to also show Java frames in the stack traces and resolve the name of the corresponding Java functions, a helper script \(jhelper.d\)^{14} is provided that implements the lookup of function addresses. Because these addresses point to compiled methods in the code cache, the helper script has to be updated to support a segmented code cache.

The helper script accesses the code cache in memory by referring to the corresponding symbol defined in the shared library. The offsets that are necessary to compute the addresses of the fields are generated by the file \(generateJvmOffsets.cpp\). The helper script of the baseline version uses these offsets to directly access the entries of the code heap and then resolve the function address.

Because the segmented code cache has multiple code heaps, the script is changed to first access the \(GrowableArray\) which stores pointers to all code heaps. The \(generateJvmOffsets.cpp\) file is adapted to additionally generate the offsets of the \_len\) and \_data\ fields used by the \(GrowableArray\) to store the number of elements and the actual data in array form. New probes are added to first obtain the destination code heap and read its configuration, such as the address of the segment table, and then continue by resolving the function address in this code heap.

One limiting feature of the D language is that there is no support for loop statements. It is therefore impossible to iterate over the \(GrowableArray\) to search for the destination code heap. At the moment the helper script supports up to five code heaps in the code cache, by specifying a probe for each. If more code heaps are added, the probes have to be extended.

More information can be found in the DTrace User Guide [37].

\(^{13}\)A language inspired by C consisting of so called probes with conditions and actions that are executed if the corresponding probe fires, i.e., the condition is met.

\(^{14}\)Located in \(/os/solaris/dtrace/\)
5. Implementation

5.1.5 Stack tracing tool Pstack

Pstack is an utility that prints the stack trace of all threads currently executed by a process. It is used for debugging purposes, for example, to figure out where a process is stuck. The HotSpot™ JVM includes Pstack support to not only show the stack frames of JVM internal threads, but also find the names of Java methods on the stack of those threads currently executing Java code. The support code can be found in `/os/solaris/dtrace/` and `/os/bsd/dtrace/`, respectively.

Pstack performs the name lookup by calling into the shared library `libjvm_db.so`15 shipped with the HotSpot™ JVM. This support library gathers the necessary information by directly reading the memory space of the JVM, using the same technique as the DTrace script (see Section 5.1.4). Similar to DTrace, changes to the code are necessary to support a segmented code cache with multiple code heaps. Instead of referencing the symbol in the shared library, the corresponding entry in the `VMStructs` class (see Section 5.1.3) is used to access the `GrowableArray` of code heaps. Compared to DTrace, the advantage is that the Pstack support library is written in C and therefore loops can be used to iterate over the code heap array. On initialization, a local array is created to store the code heap configurations. The implementation of the `contains` function, to check if the code cache contains a method, and `find_start`, to find the start segment of a method in a code heap, is changed to support multiple code heaps.

More information about Pstack can be found in [36].

5.1.6 Adding new code heaps

As described in Section 4.1, future versions of the HotSpot™ JVM are likely to include GPU and/or ahead-of-time (AOT) compiled code that should be stored in a separate code heap. The implementation of the segmented code cache is extensible, allowing to define a new code type and the corresponding code heap in just a few steps. If, for example, a new code heap for GPU code needs to be added to the code cache, the following steps are necessary:

- Definition of a new code type: Creation of a new `CodeBlobType` for GPU code. This type is used to access the GPU code in the code cache, including allocation and deallocation of memory. If the GPU code should be treated similar to method code, for example, by the sweeper, the `IsMethodPredicate` must be adapted.

- Creation of the code heap: `CodeCache::initialize_heaps` creates and initializes the new code heap with a part of the memory space reserved for the code cache. A new JVM option can specify the size of the new code heap.

- Define code heap availability: If the code is not always available or used, the availability criteria can be defined in `CodeCache::heap_available` so that the code heap is only created if necessary.

15The implementation can be found in `/os/solaris/dtrace/libjvm_db.c`
GPU code created with the new `CodeBlobType` is then stored in a separate code heap.

## 5.2 Dynamic code heap sizes

This section describes the implementation of the dynamic resizing of code heaps, including changes to other components that are necessary to support dynamic code heap sizes. As described in Section 4.2, the design is based on the idea to dynamically move the boundary between the code heaps to expand one code heap at the cost of the other code heap. The implementation builds upon the changes introduced by the segmented code cache (see Section 5.1).

To be able to move the boundary between two adjacent code heaps, both code heaps need to fill up towards the boundary, i.e., one code heap has to grow upwards and the other code heap has to grow downwards. As shown in Figure 5.1, the memory used by a code heap is managed by a `VirtualSpace` on top of a `ReservedSpace`.

The baseline version does only support upwards growing code heaps. Therefore, the implementation of all related components has to be adapted to support downwards growing and moving of boundaries. In the following sections the changes are described from bottom-up. First, Section 5.2.1 presents the implementation of the `VirtualSpace` class. Section 5.2.2 then describes the changes to the code heap to support a downward growing `VirtualSpace` and expansion into an adjacent code heap. Finally, Section 5.2.3 changes the implementation of the code cache such that the code heaps are dynamically resized if one code heap is full.

### 5.2.1 Virtual space

The `VirtualSpace` class allows committing of a `ReservedSpace` in smaller chunks by providing functions to expand and shrink the initially committed memory. The `ReservedSpace` contains the size and alignment constraints and abstracts the allocation of memory by the operating system. The reserved space is therefore independent of the direction of growth. The virtual space, however, has to be adapted to support growing top-down.

To add the missing functionality to the virtual space implementation, either (i) a subclass is added that grows downwards, (ii) a class is added that reimplements the virtual space but grows downwards, or (iii) an additional parameter is introduced which controls the direction of growth. The first solution drops out because there is no subclass relation between a virtual space that is growing downwards and one that is growing upwards. A new superclass would have to be added that combines the common functionality. The second solution drops out because there would be a lot of code duplication. The third solution is implemented and described below.

To support large pages\(^{16}\) the virtual space is split into three parts, called lower, middle and upper region. The lower and the upper regions are always pages size aligned and used to correctly align

---

\(^{16}\)Normally the kernel provides memory that is allocated in chunks of 4K bytes, named pages. However, most CPU architectures and operating systems support bigger pages that are allocated in one block and not swapped to disk.
the low and high addresses of the middle region. They are usually very small or may not exist at all if the low and high addresses of the virtual space are already aligned. The middle region is large page size aligned if the platform supports large pages.

Figure 5.2 (a) shows the regions and corresponding pointers of an upwards growing virtual space. High memory addresses are on top, low addresses on the bottom. The lower_high_boundary, middle_high_boundary and upper_high_boundary pointers define the three regions by marking their high boundaries and are set on initialization of the virtual space. The white part represents unused and the blue part represents already committed memory. The lower_level, middle_level and upper_level pointers mark the current usage of each region. They are aligned according to the regions alignment and updated if the committed space is expanded or shrunked. For example, the lower region is full, therefore the lower level is equal to the lower high boundary. The level pointer marks the usage level of the entire virtual space without region alignment. This means that it is equal to the requested space and may be lower than the actual committed space due to alignment constraints. It corresponds to the high or low water mark (depending on the direction of growth) for the last allocated byte. In Figure 5.2 (a), for example, the level pointer is smaller than the middle level because the middle level is large page aligned.

Figure 5.2: Regions and pointers of the VirtualSpace to support large pages. The white parts represent unused and the blue parts represent committed memory.

Figure 5.2 (b) shows a virtual space growing downwards. The boundary pointers do not need to change, since the regions do not change. The level pointers are now initialized to the high boundaries of the corresponding region and move from high to low addresses.

To control the direction of growth, the additional variable grows_up is introduced and added to the initialization functions of the virtual space. If set to true, the virtual space grows towards the higher memory addresses.

In the baseline version the level pointers are called ... high (for instance, upper_high) because
they always determine the highest committed address in the corresponding region. They are renamed to \ldots \_level, as shown in Figure 5.2 because they may now point to the lowest address if the virtual space grows downwards. The level pointers are initialized to the corresponding region boundary to represent an empty region. For example, middle\_level is initialized to lower\_high\_boundary for an upwards growing and set to middle\_high\_boundary for a downwards growing virtual space.

The main functionality of the virtual space, namely expanding and shrinking the committed memory space is implemented by the functions expand\_by and shrink\_by. Expanding works by first calculating the unaligned new level, based on the number of bytes that are needed. Then, the unaligned new levels for each region are determined by simply comparing the overall level to the boundaries of each region. After aligning the region levels based on the corresponding region alignment, their initial values are compared to the new values to determine the regions that are affected by the growth. Memory in those regions is committed in pages of the corresponding size and the level pointers are adapted.

The implementation of expand\_by is changed to support a downwards growing virtual space if grows\_up is set to false. This affects the calculation of the new levels (levels are decreased), alignment (rounding down instead of up), comparison of level pointers and boundaries (the level pointer is compared to the upper boundary of the region) and committing of memory.

To uncommit previously committed memory, the function shrink\_by is used. Shrinking works similar to expanding by first calculating the new levels, determining which regions are affected and finally uncommitting the memory. The function is adapted to support downwards growing.

As already described above, the actual committed size of the virtual space may be greater than the requested size because of the alignment constraints. Hence, the implementation of the virtual space provides not only the function committed\_size that works by simple computing the difference of the level pointer and the boundary, but also a function actual\_committed\_size that sums up the committed memory of all regions. The function is adapted to support a downwards growing virtual space. A function actual\_uncommitted\_space, to calculate the actual uncommitted space, is added and used for the dynamic resizing of code heaps (see Section 5.2.2). Further, the runtime assertions\footnote{A predicate leading to an error and stopping the program if violated at runtime. It may, for example, check if the level addresses are always valid, i.e., point to an address inside the virtual space boundaries.} and verification functions\footnote{Verification functions are executed periodically in debug builds to verify consistency of JVM internal structures. For example, VirtualSpace::check\_for\_contiguity verifies the correctness of the level and boundary pointers of a virtual space.} for debug builds are adapted to work with a downwards growing virtual space.

Having support for upwards and downwards growing virtual spaces, it is now possible to implement expanding into another virtual space that is adjacent and has the inverse direction of growth. The following new functions are added:

- \textbf{set\_high\_boundary}: Sets the high boundary of the virtual space and adapts the boundaries of the middle and upper region accordingly. The middle and upper region level may be affected
as well, but the overall level is not changed.

- **set_low_boundary**: Sets the low boundary of the virtual space and adapts the boundaries of the lower and middle region accordingly. The lower and middle region level may be affected as well, but the overall level is not changed.

- **grow_into**: Moves the boundary between the virtual space and an adjacent virtual space such that the size of this space is increased by a given number of bytes and the size of the adjacent virtual space is decreased accordingly. The already committed area remains unchanged for both virtual spaces.

The **grow_into** function is used by the code heap to implement the dynamic resizing (see Section 5.2.2). The function first checks if there is enough uncommitted memory available in the other virtual space and then moves the boundary into the lower part if growing upwards, or into the upper part of the other virtual space if growing downwards. Because of the complexity of the code, numerous assertions are added to check that the spaces are adjacent, the directions of growth are correct and the resulting virtual spaces are valid.

### 5.2.2 Code heap

The implementation of the **CodeHeap** is based on an underlying **VirtualSpace**, called **memory** in the following. The memory is split into so called **segments** of fixed size (defined by **CodeCacheSegmentSize**, 64 bytes by default), numbered in ascending order. If a block of space in the code heap is allocated, multiple segments are reserved and linked together, starting with a special header segment. The header segment contains information about the length (number of segments) of the block and if it is used. If a block is deallocated, it is added to a free list (a linked list of free blocks) and is later reused. Already committed but not previously used memory is marked as unused and used if no suitable block is found in the free list. If allocation fails, additional space in the virtual space is committed and initialized as unused.

Figure 5.3 shows a simplified example memory layout of the virtual space of an upwards growing code heap. The segments 0 and 1 belong to a block of length one that is in the free list. Segments 2 to 5 are part of a block of size three and currently used. Segments 6 and 7 are committed but not yet used.

A second virtual space, the so called **segment map**, is used to efficiently find the header segment of a block, given a segment inside this block. This is needed by different components of the JVM, e.g., inline caches. For each segment in the memory virtual space, there is an entry in the segment map. This entry contains the number of the segment in the corresponding block. The arrows in Figure 5.3 show an example lookup. To find the header segment of the block the segment number 5 (1.) belongs to, the corresponding entry in the segment map is consulted (2.). The entry states that segment 5 is the third segment in the block. Therefore, the header is located three segments lower. A second lookup in the segment map (3.) verifies the location, the distance to the header is now
zero, and the header segment is read from memory (4.). The second lookup is needed because the entries in the segment map are limited to one byte and therefore multiple lookups are necessary if the value of the block size does not fit into one byte (see `CodeHeap::find_start` for details about the implementation). Unused segments are marked with the special value 0xFF in the segment map.

To implement dynamic resizing, the code heap has to support a downwards growing virtual space. A parameter `grows_up` is added to the constructor to control the direction of growth. If the code heap grows downwards, the segments of the virtual space must be numbered in ascending order from top to bottom because expansion takes place towards the lower addresses. The same applies to the segment map.

Allocation of new blocks is adapted such that the header segment still resides at the lowest address of the block (now the segment with the highest number) and the segment map is initialized accordingly. The implementation of the iterator functions `first_block` and `next_block` is modified to iterate from top to bottom if the code heap is growing down. The helper functions `segment_for`, to get the segment number for an address, and `block_at` to get the segment for a segment number, and debug functions are adapted accordingly.

To enable dynamic resizing of the code heaps, by expanding one code heap into another code heap, the function `grow_into` is added. The function tries to move the boundary of the code heap into the space of another adjacent code heap such that its virtual space is increased while the space of the other code heap is decreased. Thereby the code heap is not expanded, i.e., the committed memory stays the same. If there is not enough uncommitted space available in the other code heap, the function tries to shrink the code heap by uncommitting already committed but unused memory. An increased number of segments needs a larger segment map. To be able to increase the size of the segment maps accordingly, not only the memory virtual spaces, but also the segment maps need to be adjacent to each other.
The previously unimplemented function \texttt{shrink\_by} is now implemented to support shrinking of code heaps. The function shrinks the committed memory by uncommitting already committed space and free blocks from the code heap boundary.

Figure 5.4 shows the dynamic resizing of the method code heaps. The virtual spaces of the profiled and the non-profiled code heap are adjacent in memory. The profiled code heap grows upwards, whereas the non-profiled code heap grows downwards. In Figure 5.4 (a) the non-profiled code heap is almost full, only some small free blocks are available on the free list. In contrast, the profiled code heap has unused and even not yet committed space at its upper boundary, marked in white and blue. The grey part is memory that is already committed by the virtual space due to alignment constraints, but not yet initialized by the code heap.

![Figure 5.4: Dynamic resizing of method code heaps. The non-profiled code heap is full and grows into the profiled code heap.](image)

The code heaps are resized to increase the size of the non-profiled code heap. First, the \texttt{shrink\_by} method shrinks the profiled code heap by uncommitting the alignment space and the unused space and removing some of the free blocks from the free list. Now there is enough uncommitted space to lower the boundary between the code heaps. The non-profiled code heap can be expanded by committing the additional space. Figure 5.4 (b) shows the result.

### 5.2.3 Code cache

To be able to dynamically resize the profiled and non-profiled code heap, their virtual spaces for memory and segment map have to be placed adjacent to each other in memory. The function
5. Implementation

CodeHeap::reserve is adapted to take the ReservedSpace objects for memory and segment map as an argument instead of allocating them internally at a random position in memory. The code cache creates the reserved spaces adjacent in memory and passes them to the code heaps. They are then used to initialize the virtual spaces. The function get_segmap_size is added to compute the size of the segment map according to the size of the code heap.

The resizing of the profiled and non-profiled code heap is performed as described in Section 5.2.2. It is necessary if allocate fails at runtime due to a lack of space in one of the method code heaps. Allocate is adapted to use a new function expand_heap, that tries to resize the code heaps if one code heap is full, instead of only trying to reserve more memory in the current code heap.

The function expand_heap first checks if the given code heap can be expanded by allocating more memory in its own virtual space. If this is not possible and the code heap is a method code heap, the functions makes use of the helper function get_adjacent_heap to find the adjacent code heap. If there is enough free space in this code heap, the boundary between the code heaps is moved accordingly. The newly gained space is committed and therefore available for allocations. A new JVM option MoveCodeHeapBoundaries is introduced that controls the behaviour of expand_heap. If it is set to false, no resizing is performed. For example, if tiered compilation is disabled, the profiled code heap does not exists and dynamic resizing is disabled.

With the segmented code cache, the AdvancedThresholdPolicy uses the reverse free ratio of the destination code heap to set the compile thresholds for tiered compilation (see Section 5.1.1.2). It is adapted to use the reverse free ratio of the entire code cache if dynamic resizing of code heaps is enabled. This is justified by the fact that free space in other code heaps can be used by moving the boundary and therefore the compile thresholds should only increase if all method code heaps are full.

5.2.4 Lazy creation of method code heaps

As described in Section 4.2, the method code heaps are created lazily after JVM startup when the first method allocation takes place. The function create_heaps initially assigns all space reserved for the code cache to the non-method code heap. Particularly, the function initializes the virtual space of the non-method code heap with the entire ReservedSpace. Only after the first method allocation is requested by allocate, the function initialize_method_heaps is executed. It fixes the size of the non-method code heap and initializes the method code heaps.

Because after JVM startup there still occur new allocations in the non-method code heap, for example, for adapters, the code heap is fixed to its current size plus the CodeCacheMinimumFreeSpace. If tiered compilation is enabled, additional space for the C2 scratch buffers is needed. To account for this additional space, 1% of the memory reserved for the code cache (but at least 500 kB) and additional 128 kB for each compiler thread are allocated. The non-method code heap is expanded or shrinked accordingly and the upper boundary of the virtual space is set (see Figure 4.2).

The underlying reserved space is split according to the non-method code heap size and the remaining
space is used for the method code heaps. The non-method code heap size is thereby subtracted from the non-profiled code heap size. The JVM option `NonMethodCodeHeapSize`, to set the size of the non-method code heap, is removed because its size can now be implicitly controlled by either setting the overall code cache size or the `CodeCacheMinimumFreeSpace`.

If the existing JVM option `PrintCodeCacheExtension` is enabled, detailed debug output about the dynamic resizing of code heaps and especially about the lazy creation of method code heaps is printed.

5.2.5 Serviceability Agent

To be able to determine the direction of growth of a code heap, the `_grows_up` field is added to the `VMStructs` class. The `..._high` fields of the `VirtualSpace` are renamed to `..._level` as described in Section 5.2.1 and all references are adapted.

The Java implementation of the code heap is changed to support downwards growing virtual spaces. A field `growsUp` is added, determining the direction of growth, and initialized by reading the corresponding `_grows_up` field of the code heap as defined in the `VMStructs` table. The function `begin` is adapted to return the lowest used address instead of the lower boundary and the helper functions `segmentFor` and `blockAt` are changed to count the segments from top down if the code heap is growing downwards. The function `blockBase`, returning the header segment of a code block, is updated to start at the high address and use a negative offset in the segment map if the code heap is growing downwards.

5.2.6 Dynamic tracing framework DTrace

The `generateJvmOffsets.cpp` file is adapted to generate the offset of the `_grows_up` field in the `CodeHeap` class. This offset is then used by the DTrace helper script to read the actual value and determine the direction of growth of a code heap. Depending on this direction the probes for finding the header segment of a block and finding the block for a segment number either count the segments top down or bottom up.

5.2.7 Stack tracing tool Pstack

The Pstack support library `libjvm_db.c` is adapted to additionally save the direction of growth for each code heap in the local array `heap_grows_up`. The functions `segment_for` and `block_at` then use this information and either start at the high addresses while indexing the code heap if it grows down, or start at the low addresses if it grows up. The same applies to the `find_start` method that is responsible for finding the header segment of a code block.
6 Evaluation

This chapter evaluates the performance of the implementation of the segmented code cache and the dynamic code heap sizes and compares the results to the baseline version.

Section 6.1 presents the experimental setup with details about the machines and benchmarks used. Section 6.2 determines the best default values for the code heap sizes and Section 6.3 assesses the dynamic adaption of the code heaps. Section 6.4 evaluates the overall performance and the fraction needed by the code cache sweeper to remove unused methods.

Section 6.5 measures the time taken by the code cache sweeper. Section 6.6 determines the code cache memory fragmentation of the baseline version and compares the fragmentation to the segmented code cache. Because the Instruction Translation Lookaside Buffer (ITLB) miss rate is likely to be affected as well, Section 6.7 measures it together with the instruction cache miss rate using hardware performance counters. Section 6.8 illustrates the hotness of methods in the different code heaps and compares the hotness to the alignment in the baseline version.

6.1 Experimental setup

To account for different usage scenarios of the JVM, testing and evaluation is performed on two different machines:

- **4-core system:** Desktop computer with an Intel Core i7-3820 CPU at 3.60 GHz (4 physical and 8 virtual cores, 10 MB cache) and 8 GB main memory running Ubuntu 12.04.3 (precise). GCC version 4.6.3 is used to build the JVM.

- **32-core system:** Server with four Intel Xeon E7-4830 CPUs at 2.13 GHz (8 physical and 16 virtual cores each, 24 MB cache) and 64 GB main memory running Ubuntu 11.10 (oneiric). GCC version 4.6.1 is used to build the JVM.

The implementation is also tested under Solaris and Windows to account for platform specific properties, for example, large page support or external tools (see Section 5.1.4 and 5.1.5). Additionally, Oracles internal regression test facility JDK Putback Reliability Testing (JPRT), that tests the implementation on different platforms, is used to verify correctness.

To get detailed performance measurements under real-world conditions, the following benchmark suites are used:
- **Octane**: A JavaScript benchmark developed by Google to measure the performance of real-world JavaScript applications [12]. The benchmark has a runtime of about 7 minutes. Its version 2.0 is executed using the JDK 8 JavaScript engine Nashorn (see Section 2.4) version 1.8.0 build b121.

- **SPECjbb2005**: A Java benchmark developed by the Standard Performance Evaluation Corporation (SPEC) to evaluate the performance of server side Java applications [48]. It emulates a three-tier client/server system and has a runtime of about 2 hours and 20 minutes. The latest version 1.07 is used.

- **SPECjbb2013**: A Java benchmark developed by SPEC to measure the performance of the latest Java 7 application features [49]. It models a world-wide supermarket IT infrastructure and has a runtime of about 2 hours and 20 minutes. The latest version 1.0 is used.

The Octane benchmarks are used primarily because in combination with Nashorn the dynamic compilers generate a lot of code, which is well suited to stress-test the code cache. The SPECjbb benchmarks have a longer runtime but generate less code. More detailed information about the benchmarks can be found in Section A.2 of the Appendix.

The execution and evaluation of the benchmarks is automated using a collection of Python scripts to be able to reproduce the results later. The graphs presented in the following sections always show the arithmetic mean of multiple runs together with the 95% confidence interval displayed as errorbar.

The segmented code cache and dynamic code heap sizes are provided as two patches to the code in the HotSpot™ JVM mercurial repository¹ changeset 5765:9d39e8a8ff61 from 27 December 2013, which is called *baseline version* in the following.

### 6.2 Default code heap sizes

The implementation of the segmented code cache provides JVM options to set the sizes of the non-method, the profiled and the non-profiled code heap. All values are dependent on the memory reserved for the code cache, as this space is shared between the code heaps. To determine how the code cache memory should be distributed, reasonable default values are determined by measuring performance while executing benchmarks with different code heap sizes. Although it is possible to define platform dependent default values, currently the same conservative values are used for all platforms.

To determine the default size for the non-method code heap, all benchmarks (see Section 6.1) are executed and the required space for non-method code is measured. On the 4-core system a code heap size of 4 MB is sufficient, whereas the 32-core system needs around half a megabyte more space. This is because more compiler threads are executed and therefore more C2 code buffers

---

¹http://hg.openjdk.java.net/hsx/hotspot-comp/hotspot
are created. The default non-method code heap size is set to 5 MB to make sure the JVM runs efficiently on all platforms.

Next, the default sizes for the method code heaps are determined. Octane is used as a short running and SPECjbb2005 is used as a long running benchmark. Both are executed with a small code cache size of 64 MB, to make sure that the code cache is getting full, and different non-profiled code heap sizes. Octane is executed 20 times (7 minutes each) and SPECjbb2005 is executed 5 times (2 hours and 20 minutes each) for each configuration on the 32-core system. Figure 6.1 shows the benchmark results.

Figure 6.1: Performance evaluation with different non-profiled code heap sizes on the 32-core system. The code cache size is 64 MB, where 5 MB are used for the non-method code heap and the rest is distributed equally between the profiled and the non-profiled code heap.

Figure 6.1(a) shows that there is a performance degradation for small and large non-profiled code heap sizes. The system performs best with a non-profiled code heap size of 20 MB, i.e., with a profiled code heap size of around 60 MB (64 MB minus the space needed for the non-method code heap). This is because Octane consists of multiple short running benchmarks and therefore the JVM profiles a lot of code.

The SPECjbb2005 benchmark, presented in Figure 6.1(b), shows no performance change with different non-profiled code heap sizes. This is because there is not enough code generated to fill up the code heaps and it therefore makes no difference if the code cache is segmented or not.

To account both for short running and long running applications, the available memory space is equally distributed among the non-profiled and the profiled code heaps. For example, with a code cache size of 65 MB, 5 MB are used for the non-method code heap and 30 MB are used for each method code heap.
6.3 Dynamic code heap sizes

To monitor the dynamic resizing of the code heaps, the JVM option `PrintCodeCacheExtension` is used. The option prints information about a code heap whenever it is expanded or resized. The Octane benchmarks are executed on the 4-core machine with a code cache size of 64 MB. To force the JVM to resize the code heaps, the profiled code heap is set to only 10 MB and the non-profiled code heap is set to 54 MB.

The output shows that the internal JVM startup is completed around 0.3 seconds after the start and the non-method code heap is fixed to 2.2 MB, which is sufficient for the 4-core system. The method code heaps are created and the JVM allocates 10 MB to the profiled code heap and 51.8 MB (54 MB minus 2.2 MB for the non-method code heap) to the non-profiled code heap.

![Dynamic code heap sizes (Octane Benchmark, 4-core)](image)

**Figure 6.2:** Dynamic resizing of code heaps with the Google Octane benchmark. The code cache size is 64 MB, the profiled code heap is initially set to 10 MB and the non-profiled code heap to 54 MB. The solid lines show the reserved size whereas the dotted lines represent the used part.

Figure 6.2 shows the variation of the code heap sizes in time after the start. The solid lines show the reserved size whereas the dotted lines represent the memory that is used by the code heap. The used size corresponds to the committed size described in Section 5.2.2 and is illustrated by the red, green and blue parts in Figure 5.4.

At the beginning, the profiled code heap grows fast because a lot of code is profiled. The used part is always equal to the reserved space that is continuously increased by growing into the non-profiled code heap. The non-profiled code heap shrinks due to the memory consumption of the profiled code heap. However, the usage of the non-profiled code heap increases, albeit more slowly than for the profiled code heap. After 91 seconds the sizes of the code heaps have stabilized, but the usage of the non-profiled code heap still grows. This is because at this stage the profiled methods are now replaced by non-profiled and highly optimized versions. The total runtime of the benchmark
6. Evaluation

is 223 seconds, but after 125 seconds both code heaps use all their reserved memory. Compilation is not disabled because the method sweeper starts removing methods and space is added to the free lists of the code heaps (counted as "used" here). The final sizes are 32.6 MB for the profiled and 20.2 MB for the non-profiled code heap.

6.4 Overall performance

To measure the overall performance of the segmented code cache and the dynamic code heap sizes, the benchmarks described in Section 6.1 are executed with different code cache sizes. Small sizes, between 16 MB and 64 MB, are used to make sure that the code cache fills up and method sweeping as well as dynamic resizing takes place. Large code cache sizes, from 128 MB to 256 MB, evaluate the implementation without code cache contention. The short running Octane benchmark is repeated 20 times and the long running SPECjbb benchmarks are repeated three times for each configuration.

Figure 6.3: Octane benchmark with different code cache sizes on the 4-core machine.

Figure 6.3 shows the results of executing the Octane benchmark with each implementation on the 4-core machine with different code cache sizes. For very small sizes, the baseline version clearly performs best. This is because on average the fragmentation of a segmented code cache with multiple code heaps is higher than the fragmentation of a single code heap (see Section 6.6). Therefore, the code heaps fill up faster and compilation is disabled until the method sweeper freed enough memory, leading to a performance regression. The dynamic code heap sizes perform slightly better (around 17%) than the segmented code cache because the code heaps can dynamically adapt to the runtime needs and therefore fill up more slowly.

With code cache sizes greater 32 MB, the segmented code cache performs on average 1% to 4%
better than the baseline version, but with a high variation it is hard to confirm this. The performance gain is probably due to a lower sweep time (see Section 6.5) and a better ITLB behaviour (see Section 6.7). For 16 MB and 48 MB the dynamic code heap sizes perform worse than the baseline version and partly even worse than the segmented code cache, although the code heaps are able to adapt to the runtime requirements. This performance degradation is due to the code cache sweeper. The sweeper removes too many methods because the code heaps seem full prior to resizing (see Section 6.5). With a larger code cache the dynamic code heap sizes perform better, with a performance gain of around 4%.

![Figure 6.4: Octane benchmark with different code cache sizes on the 32-core machine.](image)

Figure 6.4 shows the same configuration executed on the 32-core system. On average, the implementations perform worse than on the 4-core system. The lower performance is due to the processor being slower (2.13 GHz vs. 3.60 GHz) and a limited parallelizability / scalability of the Octane benchmark. Comparing the performance for different code cache sizes, the implementations perform comparable to the execution on the 4-core system. The segmented code cache performs up to 7% better than the baseline version, except for a very small code cache size of 16 MB. The dynamic code heap sizes perform worse for small code cache sizes and similar to the segmented code cache for larger code cache sizes.

Figure 6.5 displays the performance of the SPECjbb2005 benchmark executed on the 4-core machine. On average the difference in performance of the segmented code cache and the dynamic code heap sizes compared to the baseline version is below 0.5%. Additionally, the 95% confidence intervals show that there is no measurable difference between the implementations. To verify this result, the same configuration is executed on the 32-core system. The results are shown by Figure A.1 in Section A.1 of the Appendix. The average performance is significantly better than on the 4-core system because the SPECjbb2005 benchmark scales better than the Octane benchmark. The average performance gain compared to the baseline version is below 1%. 
6. Evaluation

Figure 6.5: SPECjbb2005 benchmark with different code cache sizes on the 4-core machine.

Figure 6.6: SPECjbb2013 benchmark with different code cache sizes on the 32-core machine.

The SPECjbb2013 benchmark is executed on the 32-core as well. Figure 6.6 shows that the confidence intervals are high and the implementations perform equally well. The first version of the SPECjbb2013 benchmark that is used here still seems to have a very high variance and even more than three runs do not decrease the confidence intervals. Anyway, it is noticeable that the performance of the segmented code cache and the dynamic code heap sizes is around 68% worse than the baseline version for a very small code cache size of 16 MB. As stated earlier, this is due to the increased fragmentation resulting in the individual code heaps filling up more rapidly. The same configuration is executed on the 4-core system (see Figure A.2 in the Appendix).
6.5 Sweep time

The code cache sweeper performs stack scanning at safepoints to update the hotness values of methods and determines methods not longer needed. The sweeper removes methods in multiple steps (see Section 2.3.6). To measure the time taken by the sweeper, a patch to the HotSpot™ JVM is implemented that adds the JVM option `PrintMethodFlushingStatistics`. If enabled, additional information about the code cache sweeper, for example, the total time taken, is printed (see Section A.3). The patch also fixes bug JDK-8025277: Add -XX: flag to print code cache sweeper statistics [30] in the JDK Bug System.

The Octane benchmarks are executed on the 4-core machine with different code cache sizes and the time taken by the sweeper is measured. Figure 6.7 shows the results for 20 iterations per code cache size.

![Sweep time (Octane Benchmark, 4-core)](image)

Figure 6.7: Time taken by the method sweeper on the 4-core machine.

The sweeper of the segmented code cache performs better than the baseline version by around 19% to 46%. The dynamic code heap sizes perform between 12% and 21% better for small code cache sizes up to 64 MB, but up to 54% worse for larger code cache sizes. More sophisticated evaluations show that this is because the code cache sweeper is invoked too often and sweeps to many methods.

In more detail, the `NMethodSweeper::possibly_sweep` method uses the maximum reverse free ratio of all method code heaps (`CodeCache::reverse_free_ratio`) to decide if the sweeper should be invoked. The function `NMethodSweeper::process_nmethod` then uses the reverse free ratio of the corresponding code heap to compute the hotness threshold that decides if a method should be removed. The problem is that with the dynamic code heap sizes the reverse free ratio of a code heap may be large even if there is still enough space in the adjacent code heap allowing the code heap to grow. Additionally, there is more code generated than with the segmented code cache.
version because the AdvancedThresholdPolicy is adapted to use the reverse free ratio of the entire code cache (around 37% more methods are compiled on the 4-core system). Especially the profiled code heap quickly gets full and is then expanded dynamically by growing into the non-profiled code heap. Because the dynamic resizing is done in small steps, the sweeper assumes that the profiled code heap is almost always full and sweeps as often as possible (see also Section 6.6). For small code cache sizes up to 64 MB, the behaviour of the sweeper is appropriate because the code heaps are indeed full and sweeping is necessary. The sweep time is almost identical to the value of the segmented code cache.

Simply changing the implementation of the sweeper to use the reverse free ratio of the entire code cache does not improve, but greatly degrade performance. This is because the sweeper then sweeps too little, resulting in the code cache getting full and compilation being disabled. Also adapting the AdvancedThresholdPolicy does not solve the problem because then not enough code is generated to make use of the dynamic resizing of the code heaps. Multiple solution approaches are described in Section 7.1.

Simply changing the implementation of the sweeper to use the reverse free ratio of the entire code cache does not improve, but greatly degrade performance. This is because the sweeper then sweeps too little, resulting in the code cache getting full and compilation being disabled. Also adapting the AdvancedThresholdPolicy does not solve the problem because then not enough code is generated to make use of the dynamic resizing of the code heaps. Multiple solution approaches are described in Section 7.1.

Figure 6.8 shows the results of the same benchmark on the 32-core machine. On average, the sweep time is larger than on the 4-core machine. This is partly because the runtime is higher (426 seconds instead of 261 seconds) and more compiler threads (18 instead of 4 compiler threads) are used, leading to an increased amount of code. The trend of the sweep time is similar, the segmented code cache performs up to 41% better and the dynamic code heap sizes perform worse than the baseline version for larger code cache sizes.

To measure the sweep time while executing a long running program, the SPECjbb2005 benchmark is used. Figure 6.9 shows the results of executing the benchmark on the 32-core machine with 3 repetitions for each code cache size. On average the sweep time is extremely low (0.1 to 1.6 seconds)
Figure 6.9: Time taken by the method sweeper to remove methods on the 32-core machine.

compared to the Octane benchmark. This is because the SPECjbb2005 benchmark generates
less code, so that even with small code cache sizes almost no sweeper activity is necessary. Due
to the high variance it is not possible to make a statement about the differences between the
implementations.

Although the sweep time is greatly improved for the segmented code cache, the overall performance
is affected only little (see Section 6.4). This can be explained by the fact that the code cache sweeper
is executed by only one (compiler-) thread in parallel to normal execution and therefore affects
performance to a lesser extent. Stack scanning must be executed during a safepoint. However, the
performance gain may be improved by using separate locks for each code heap (see Section 7.1),
instead of one lock for the entire code cache that has to be acquired each time the sweeper processes
a method.

6.6 Memory fragmentation

To evaluate the fragmentation of the code heaps and compare the fragmentation to the fragmenta-
tion of the single code heap of the baseline version, an additional patch extends the code cache
by a function `print_usage_graph` that prints the length and usage information of each code heap
block. The function is executed once at the end of the execution, analysed by a Python script and
visualized by a graph. The graph always grows down and is independent of the direction of growth
of the corresponding code heap.

To be able to easily compare the fragmentation of different versions, the external fragmentation\(^2\)
is computed by using the following formula described in [51]:

\(^2\)External fragmentation is the fragmentation that occurs if the allocated memory is interspersed by free segments.
External Memory Fragmentation = \(1 - \frac{\text{Largest Block Of Free Memory}}{\text{Total Free Memory}}\)

![Code cache fragmentation (Octane Benchmark, 4-core)](image)

Figure 6.10: Fragmentation of the code cache of the baseline version. Profiled methods are marked in red, non-profiled methods and non-method code is marked in blue.

Figure 6.10 shows the fragmentation of the code cache of the baseline version after a single run of the Octane benchmark on the 4-core machine. The code cache size is set to 256 MB because for small code cache sizes the fragmentation varies a lot due to the frequent sweeping. Segments containing profiled methods are marked in red, non-profiled methods and non-method code is marked in blue. The vast majority of the code cache is occupied by profiled methods that are mixed with non-profiled methods and non-method code. Because only a few blocks between the allocated segments are free and most of the free space is present in one block at the top of the code heap, the external fragmentation for this execution is 7.02%.

Figure 6.11 shows the fragmentation of the code heaps for the segmented code cache version after executing the same benchmark configuration. The sizes of the graphs are fixed and not related to the size of the corresponding code heap. The graph titled with ”All code heaps” shows the overall memory layout of the code cache containing the three (adjacent) code heaps. The numbers in parentheses specify the external fragmentation of each code heap for this run. As expected, non-method, profiled and non-profiled code is perfectly separated. The fragmentation of the non-method code heap is similar to the fragmentation of the baseline version, whereas the fragmentation of the profiled code heap is worse and the fragmentation of the non-profiled code heap improved.

Figure 6.12 shows the same information for the dynamic code heap sizes version. Because the non-method code heap size is fixed lazily, the non-method code heap is smaller than the non-method code heap of the segmented code cache version. Although the amount of non-method code is the same for both versions, the non-method code heap appears to be fuller here. As already stated in Section 6.5 more profiled code is generated and the profiled code heap is therefore subject to frequent resizing and sweeping, resulting in high fragmentation. The non-profiled code heap has a very low external fragmentation. Table 6.1 lists the average external fragmentation values for the code heaps of all three implementations while running 20 repetitions of the Octane benchmark with a code cache size of 256 MB on the 4-core machine. The values after the ± sign correspond to the 95%
Figure 6.11: Fragmentation of the code heaps of the segmented code cache version. Profiled methods are marked in red, non-profiled methods and non-method code is marked in blue.

Figure 6.12: Fragmentation of the code heaps of the dynamic code heap sizes version. Profiled methods are marked in red, non-profiled methods and non-method code is marked in blue.
6. Evaluation

The fragmentation values for the non-method and the non-profiled code heap confidence interval. The fragmentation values for the non-method and the non-profiled code heap

<table>
<thead>
<tr>
<th>Version</th>
<th>Non-method</th>
<th>Profiled</th>
<th>Non-Profiled</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline version</td>
<td>4.9% ± 0.73</td>
<td>4.9% ± 0.73</td>
<td>4.9% ± 0.73</td>
<td>4.9% ± 0.73</td>
</tr>
<tr>
<td>Segmented code cache</td>
<td>5.16% ± 0.96</td>
<td>19.46% ± 1.42</td>
<td>0.09% ± 0.07</td>
<td>24.94% ± 0.37</td>
</tr>
<tr>
<td>Dynamic code heap sizes</td>
<td>23.52% ± 2.56</td>
<td>55.5% ± 12.4</td>
<td>0.15% ± 0.04</td>
<td>15.89% ± 7.98</td>
</tr>
</tbody>
</table>

Table 6.1: Average external fragmentation.

are most important because the code stored there has the longest lifetime and is hot, i.e., is used permanently (see also Section 6.8). In contrast, the profiled code is only stored temporarily and will be replaced by an optimized, non-profiled version. Table 6.1 shows that the fragmentation of the non-method code heap with the segmented code cache is equal to the fragmentation of the baseline version code cache. For the dynamic code heap sizes, the fragmentation is higher. This is probably due to the lazy fixing of the non-method code heap size and needs further investigation. The fragmentation of the non-profiled code heaps is greatly improved. With the segmented code cache it is 98% better and with the dynamic code heap sizes around 97% compared to the fragmentation of the baseline version code heap.

In theory, the segmentation of the code cache should improve the instruction TLB and instruction cache hit rate because code of the same type, that is likely to be accessed close in time, is now located at the same place. Additionally, a lower fragmentation leads to less unused blocks that may pollute the instruction cache. Section 6.7 evaluates the instruction TLB and instruction cache hit rate in detail.

6.7 ITLB and cache behaviour

The instructions stored in the code cache are executed by the processor at runtime. To speed up the fetching of executable instructions from memory, the processor uses an instruction cache that contains frequently used memory pages. To speed up the virtual-to-physical address translation because the user processes accesses virtual memory addresses, an instruction translation lookaside buffer (ITLB) is used. The ITLB caches the operating systems page table that contains the corresponding physical address for each virtual address.

In general, an instruction cache read miss is most expensive because the thread has to stop execution until the instruction is fetched from memory. This fetching from memory may cause an ITLB miss which is costly as well. It is therefore important to optimize the code cache with respect to ITLB and instruction cache behaviour. The segmented code cache should improve because code locality is increased and fragmentation reduced (at least for the non-profiled code, see Section 6.6).

To measure the ITLB and instruction cache behaviour of the implementations, hardware performance counters are used. The hardware performance counters are enabled and accessed using

---

3Special registers built into modern CPUs to measure activities. For example, the number of cache misses or the number of instructions that are executed is measured. The registers are limited and have a low overhead. An overview for the Intel architecture can be found in [19].
the `perf` tool, available in the Linux kernel. The events measured by the hardware performance counters are CPU specific and can be found in the Intel Software Developer’s Manual [19]. The following events are used to measure instruction cache and ITLB misses:

- **ITLB_MISSES.MISS_CAUSES.A_WALK** (Event 85H, Umask 01H): "Misses in ITLB that causes a page walk of any page.” ([19], page 19-5)

- **ICACHE.MISSES** (Event 80H, Umask 02H): "Number of Instruction Cache, Streaming Buffer and Victim Cache Misses. Includes UC accesses.” ([19], page 19-5)

Figure 6.13 shows the instruction TLB miss rate while executing the Octane benchmark on the 32-core system. Different code cache sizes are used and each configuration is executed 20 times (142 minutes altogether).

![Instruction TLB behaviour (Octane Benchmark, 32-core)](image)

As expected, the segmented code cache performs better than the baseline version by reducing the number of ITLB misses by up to 19%, except for very small code cache sizes below 32 MB. With the dynamic code heap sizes, the miss rate improves by up to 13% and is similar to the baseline version for small code cache sizes. This is due to the resizing of the method code heaps and the increased amount of sweeping activity, polluting the instruction cache (see below). This results in an increased amount of fetching from main memory, corrupting the ITLB. Figure A.3 in the Appendix shows the same configuration executed on the 4-core machine, leading to similar results.

Figure 6.14 shows the instruction cache misses on the 32-core system. With the segmented code cache the miss rate is up to 14% lower for larger code cache sizes and higher for small code cache

---

[See https://perf.wiki.kernel.org](https://perf.wiki.kernel.org)
6. Evaluation

Figure 6.14: Instruction cache misses while running the Octane benchmark on the 32-core machine.

sizes, compared to the baseline version. With the dynamic code heap sizes the instruction cache miss rate is up to 30% higher than with the baseline version. As already described above and in Section 6.6, the higher instruction cache miss rate is due to the resizing of code heaps and the increased sweeping activity. Executing the same configuration on the 4-core system provides similar results (see Figure A.4 in the Appendix).

Figure 6.15: ITLB misses while running the SPECjbb2005 benchmark on the 32-core machine.

The improvement can also partly be explained by the fact that each compiler thread is either assigned to the C1 or the C2 compiler and therefore either accesses profiled or non-profiled code. This means that with the segmented code cache a compiler thread only accesses one code heap,
improving code locality. Since threads are likely to be executed on the same CPU, that has its
own instruction cache and ITLB, the cache misses are reduced. This may also explain the slightly
higher miss rates on the 4-core machine (see Figure A.4 in the Appendix) because more compiler
threads are executed on the same CPU, mutually trashing the caches.

To evaluate the behaviour of a long running program, the SPECjbb2005 benchmark is executed
three times (6.60 hours) on the 32-core system. Figure 6.15 shows the instruction TLB miss rate.
The miss rate with the segmented code cache is improved by up to 44% compared to the baseline
version. This is because the benchmark is long running and therefore a lot of methods are highly
optimized and stored in the non-profiled code heap. This increases code locality and therefore
lowers the ITLB miss rate. In contrast to the Octane benchmark, also the dynamic code heap sizes
perform better than the baseline version. As shown in Figure 6.9 of Section 6.5, the sweep time
for the SPECjbb2005 benchmark is similar to the baseline version. Therefore, the code locality is
not degraded and the instruction cache miss rate is comparable to the segmented code cache (see
Figure A.5 in the Appendix). The dynamic code heap sizes version performs up to 38% better than
the baseline version.

It is also noticeable that the ITLB and instruction cache miss rate is not increasing with smaller
code caches sizes. This is because the SPECjbb2005 benchmark generates only a small amount of
code and is therefore only slightly dependent on the code cache size.

6.8 Hotness of methods

As described in Section 2.3.6, the sweeper decides to remove a method from the code cache based
on its hotness. The hotness value measures the utilization and is initially set to a high value that
is decremented every time the method is encountered by the sweeper in the code cache and reset
by stack scanning. Hot methods are scheduled for profiling and eventually optimized and stored in
the non-profiled code heap.

To be able to measure the hotness, the patch used in Section 6.6 is extended to also print the
hotness value for each code heap block. A Python script analyses the log file and visualizes the
hotness distribution in the code cache by using different colors for each value.

Figure 6.16 shows the hotness distribution in the code cache after running the Octane benchmark
with the baseline version on the 4-core system. Because the code cache size is set to 265 MB, the
maximum hotness is per definition $265 \text{ MB} \times \frac{2}{265 \text{ MB}} = 512$. The average hotness is 427.

The hottest methods accumulate at the bottom, i.e., the top of the code cache because those are
the methods allocated last. In general, one notices a top down trend from colder to hotter methods.
There are some hot methods in between that are either due to freed segments that were reused
by recently compiled methods or correspond to methods that are encountered on the stack by the
code cache sweeper.

Figure 6.17 shows the hotness distribution of the code heaps after running the same configuration
Figure 6.16: Hotness distribution in the code cache after running the Octane benchmark with the baseline version and a code cache size of 256 MB on the 4-core machine.

with the segmented code cache version. The size of the graph is not related to the size of the corresponding code heap. The code in the non-method code heap is not swept and therefore always hot. The profiled code heap contains mostly colder methods, but some hot code that was recently scheduled for profiling. The non-profiled code heap contains a large block of hot code, corresponding to the hot methods that were recently optimized.

Figure 6.18 shows the same measurements for the dynamic code heap sizes version. The hotness distribution in the non-profiled code heap is similar to the segmented code cache, but the profiled code heap contains a much greater percentage of hot methods. This can be explained by the increased amount of sweeping that is caused by the resizing of the profiled code heap. The profiled code heap is always filled and therefore the sweeper removes methods that are then recompiled and hot.

Table 6.2 lists the average hotness values after 20 runs of the Octane benchmark with a code cache size of 256 MB. The values after the ± sign correspond to the 95% confidence interval. With the segmented code cache, the average hotness in the profiled code heap is lower than in the non-profiled code heap because hot methods are eventually optimized and stored in the non-profiled code heap. This is not the case with the dynamic code heap sizes due to excessive sweeping of the profiled code heap.

<table>
<thead>
<tr>
<th>Version</th>
<th>Profiled Mean ± Standard Deviation</th>
<th>Non-Profiled Mean ± Standard Deviation</th>
<th>All Mean ± Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline version</td>
<td>430.68 ± 3.51</td>
<td>448.8 ± 3.0</td>
<td>427.18 ± 2.88</td>
</tr>
<tr>
<td>Segmented code cache</td>
<td>469.71 ± 10.06</td>
<td>410.63 ± 3.41</td>
<td>437.01 ± 3.08</td>
</tr>
<tr>
<td>Dynamic code heap sizes</td>
<td></td>
<td></td>
<td>452.69 ± 2.48</td>
</tr>
</tbody>
</table>

Table 6.2: Average hotness after 20 runs of the Octane benchmark.
Figure 6.17: Hotness distribution in the code heaps after running the Octane benchmark with the segmented code cache version and a code cache size of 256 MB on the 4-core machine.

Figure 6.18: Hotness distribution in the code heaps after running the Octane benchmark with the dynamic code heap sizes version and a code cache size of 256 MB on the 4-core machine.
7 Conclusion

Past activities in optimizing the performance of the HotSpot\textsuperscript{TM} Java Virtual Machine focused on the performance of the dynamic compilers and the supporting runtime. This thesis presents an approach that optimizes the JVM at a low layer by redesigning the structure of the code cache. The changes form the basis for further optimizations and make the JVM more extensible and adaptive.

Because the code cache is a core component of the JVM, being directly referenced from more than 150 locations in the source code, the implementation presented in this thesis is fairly complex. The two patches consist of around 3600 lines of code and affect 44 files of the baseline version.

A detailed evaluation shows that the approach is promising and that the organization of the code cache has a significant impact on the overall performance of the JVM. The execution time is improved by up to 7\% and the more efficient code cache sweeping reduces the time taken by the sweeper by up to 46\%. This, together with a decreased fragmentation of the non-profiled code heap by around 98\%, leads to a reduced instruction TLB (44\%) and instruction cache (14\%) miss rate.

It therefore seems worthwhile to include the changes into the product version of the HotSpot\textsuperscript{TM} Java Virtual Machine. As of February 2014, the segmented code cache patch is reviewed by Oracle as a fix for the bug JDK-8015774: Add support for multiple code heaps [29] in the JDK Bug System. It will most probably be included into one of the future releases.

7.1 Future work

As described in Section 6.5, there is too much sweeping activity with the dynamic code heap sizes. Although the method code heaps are resized if full, the sweeper already starts removing more methods if a code heap is getting full.

One solution is to resize the code heaps earlier. Instead of resizing a code heap if allocation fails and there is no more uncommitted space available, the code heap is already resized in advance when it starts getting full. This ensures that a code heap only fills up if there is not enough space available in the adjacent code heap. One problem with this solution is the tendency of oscillations of the boundary if both code heaps start to resize alternately while getting full.

Another approach is to adapt the threshold computation in the sweeper, such that free space in the adjacent code heap is taken into account as well, when deciding if the sweeper should be invoked.
The challenge of this solution is to find a compromise between sweeping too often and sweeping not often enough.

In general, it is also possible to generate less compiled code by adapting the tiered compilation threshold policies. By now, the free space in adjacent code heaps is considered as available for both code heaps. This is not always the case because blocks of free space that are not located at the boundary cannot be made available by resizing. Hence, it may be good to adapt the policies to only partly consider this space as available.

Some of these solutions may be combined. A detailed evaluation is necessary to assess their performance. In the following, further optimizations of the code cache are proposed that greatly differ in their complexity and size.

- **Concurrent sweeping:** By now, only one compiler thread is used for sweeping. With the segmented code cache, the sweeping process can easily be parallelized by assigning one sweeper thread to each code heap. This potentially improves the sweep time and allows for further optimizations. For example, code with a limited lifetime, such as profiled code, can be swept more often.

- **Selectively disabling compilation:** Currently, compilation is fully disabled if one of the code heaps is full and resizing fails. But most of the time there is still space available in the other code heaps, for example, on the free list. Hence, it may pay off to continue compiling code of the corresponding code types and only selectively turn off compilation for those code heaps that are full.

- **Fine grained locking:** Instead of using a single lock for the entire code cache, it is possible to use a lock per code heap, enabling multiple threads to access the code cache in parallel and possibly improving performance of the dynamic compilers and the code cache sweeper.

- **Separation of code and metadata:** Currently, the compiled method code stored in the code cache does not only contain executable code, but also metadata, for example, the header or relocation and debugging information. By separating this metadata from the actual code and storing it somewhere else, for example, in a separate code heap, the instruction cache and ITLB miss rates may further be reduced.

- **Code heap partitioning:** By now, the code heaps are partitioned into segments of a fixed size \(\text{CodeCacheSegmentSize}\). To decrease fragmentation the code heap could be split into regions with different segment sizes to account for methods of different sizes.

- **Heterogeneous code:** Future versions of the HotSpot\textsuperscript{TM} JVM may have to manage additional types of code. For example, the project Sumatra adds support for GPU code that is handed to the GPU drivers and then converted to machine code that is executable on the GPU (OpenCL code). Additional code heaps may be created to store code of such new types (see Section 5.1.6).
Appendix

A.1 Additional graphs

This section contains supplementary graphs referenced from the evaluation in Section 6.

Figure A.1: SPECjbb2005 benchmark with different code cache sizes on the 32-core machine.

Figure A.2: SPECjbb2013 benchmark with different code cache sizes on the 4-core machine.
Figure A.3: Instruction TLB misses while running the Octane benchmark on the 4-core machine.

Figure A.4: Instruction cache misses while running the Octane benchmark on the 4-core machine.

Figure A.5: ICache misses while running the SPECjbb2005 benchmark on the 32-core machine.
A. Appendix

A.2 Benchmarks

This section provides additional information about the benchmarks used to evaluate the implementations.

A.2.1 Google Octane JavaScript benchmark

The Google Octane benchmark is a JavaScript benchmark to measure the performance of large, real-world JavaScript applications. Its version 2.0 was released on the 6th of November 2013 and consists of 17 individual tests. These tests include a variety of sophisticated applications, for example, an OS kernel simulation, equation and constraint solvers and physical simulations, focusing on different aspects, such as code optimization, garbage collection or floating point operations. The time it takes to complete the individual tests is measured and a score is computed that is inversely proportional to this runtime. The higher the score, the better the performance.

Octane is executed in the JVM by using the Nashorn framework and generates a lot of code. It is therefore good to stress the code cache and evaluate different optimizations of the code cache structure. More detailed information about the Octane benchmark and the individual tests can be found in [12].

A.2.2 SPECjbb2005

The SPECjbb2005 benchmark was developed by the Standard Performance Evaluation Corporation (SPEC) to evaluate the performance of server side Java applications. Its current version 1.07 emulates a client/server system consisting of three tiers, including business logic and object manipulation to simulate real-world applications. To measure scalability the workload is slowly increased and a detailed report is created, rating performance in business operations per seconds (bops).

SPECjbb2005 was replaced by SPECjbb2013 on October 1, 2013 but is still used for evaluation purposes in this work because of its low variance. More information can be found in [48].

A.2.3 SPECjbb2013

The SPECjbb2013 benchmark was developed by the Standard Performance Evaluation Corporation (SPEC) to measure the performance of the latest Java 7 application features and replaces the SPECjbb2005 benchmark. It simulates a world-wide supermarket IT infrastructure including point-of-sale requests, online purchases and data-mining operations. It iteratively increases the workload to account for server systems with many CPUs and measures performance by using two metrics. A pure throughput metric in the form of maximum Java operations per seconds (jOPS) and a critical throughput metric under service level agreements constraining the response times. More information can be found in [49].
A.3 New JVM options

This section lists and describes the additional JVM command line options that are introduced by the segmented code cache and the dynamic code heap sizes. A complete list of already existing JVM options can be found in [32]. The JVM options can be set by specifying \(-XX: [name]= [value]\) on the command line. For example, \(-XX: \text{ReservedCodeCacheSize}=512M\) sets the \text{ReservedCodeCacheSize} option to 512 MB.

The following options are added:

- **NonProfiledCodeHeapSize**: Sets the size in bytes of the code heap containing non-profiled methods. Per default it is set to 50\% of the \text{ReservedCodeCacheSize}.

- **ProfiledCodeHeapSize**: Sets the size in bytes of the code heap containing profiled methods. It is only applicable if tiered compilation is enabled. Per default it is set to 50\% of the \text{ReservedCodeCacheSize}.

- **MoveCodeHeapBoundaries**: Dynamically resizes the method code heaps by adjusting the boundaries between them.

- **PrintMethodFlushingStatistics**: A diagnostic JVM option that first has to be unlocked by specifying \(-XX: +UnlockDiagnosticVMOptions\). It prints statistics about the sweeper and fixes bug JDK-8025277: Add \(-XX: \text{flag} \) to print code cache sweeper statistics [30] in the JDK Bug System.

A.4 Used tools/frameworks

A.4.1 Eclipse

Because the HotSpot\textsuperscript{TM} JVM is mostly written in C++, the Eclipse IDE for C/C++ Developers [10] is used for development.

A.4.2 Python

The plots presented in Section 6 are automatically generated by using a set of simple Python scripts to automate execution and analysis. For graph generation the Python 2D plotting library Matplotlib [17] is used.

A.4.3 yEd Graph Editor

To create the high-quality diagrams used in this thesis, the yEd graph editor is used. It is a free tool available from [54] and supports importing of own data, different types of diagrams (for example, UML, Flowchart and Entity Relationship diagrams) and a variety of output formats.
Bibliography


