JOHANNES KEPLER
UNIVERSITY LINZ | JKU

Faculty of Engineering
and Natural Sciences

Generalized Trace Compilation for Java

DISSERTATION

submitted in partial fulfillment of the requirements
for the academic degree

Doktor der technischen Wissenschaften

in the Doctoral Program in Engineering Sciences

Submitted by

Dipl.-Ing. Dipl.-Ing. Christian Häubl

At the

Institute for System Software

Accepted on the recommendation of

o.Univ.-Prof. Dipl.-Ing. Dr. Dr.h.c. Hanspeter Mössenböck

Univ.-Prof. Dipl.-Ing. Dr. Michael Franz

Co-advisor

Dipl.-Ing. Dr. Christian Wimmer

Linz, January 2015

Abstract

Traditional method-based just-in-time (JIT) compilation translates methods to optimized machine code. Trace-based compilation only generates machine code for frequently executed paths, so-called traces, that may span multiple methods. This results in faster compilation, less generated machine code, and often better optimized machine code.

We present our implementation of trace recording and a trace-based JIT compiler in a modification of the Java HotSpot VM. Our tracing implementation records traces for Java applications during execution and supports traces that either start at loop headers or at method entries. After several times of trace recording, our trace-based JIT compiler merges the recorded traces into a structure suitable for compilation, a so-called trace graph. During compilation, traditional and trace-specific optimizations are applied and guarded with run-time checks if necessary. The generated machine code is then invoked by the interpreter or by other compiled traces. When a run-time guard fails or a method part must be executed that was not covered by traces and was therefore not compiled, execution falls back to the interpreter.

Our most profitable optimization is trace inlining, which is similar to method inlining in method-based JIT compilers. It widens the compilation scope and allows optimizing code parts of multiple methods as a whole, which increases the performance. The major advantage of trace inlining over method inlining is that the trace information is context-sensitive and therefore more accurate as it depends on the specific call site. This allows inlining different method parts for each specific call site which increases performance while reducing code size and compilation time.

In terms of peak performance, our trace-based JIT compiler outperforms the Java HotSpot client compiler by up to 59%. On some of the benchmarks, we also outperform the HotSpot server compiler in spite of the fact that our compiler performs significantly fewer optimizations and is therefore substantially faster.

Kurzfassung

Herkömmliche methodenbasierte just-in-time (JIT) Kompilierung übersetzt Methoden in optimierten Maschinencode. Pfadbasierte Kompilierung erzeugt nur Maschinencode für häufig ausgeführte Pfade, sogenannte Traces, die auch über mehrere Methoden hinweggehen können. Dies führt zu einer schnelleren Kompilierung sowie zu weniger generiertem und oft besser optimiertem Maschinencode.

Wir implementierten ein Verfahren, um Traces während der Ausführung von Java Programmen aufzuzeichnen, sowie einen pfadbasierten JIT Compiler, indem wir die Java HotSpot VM modifizierten. Traces können bei einem Schleifenkopf oder am Anfang einer Methode beginnen. Aufgezeichnete Traces werden an den JIT Compiler übergeben, welcher diese in einen Trace-Graphen zusammenführt. Beim Übersetzen des Trace-Graphen nach Maschinencode werden traditionelle, optimistische und pfadspezifische Optimierungen angewandt. Der erzeugte Maschinencode wird dann vom Interpreter oder von bereits kompiliertem Code aufgerufen. Wenn ein Methodenteil ausgeführt werden muss, der nicht durch Traces abgedeckt ist und somit nicht kompiliert wurde, dann wird der Maschinencode deoptimiert und die Ausführung im Interpreter fortgesetzt.

Unsere lohnendste Optimierung ist das Inlinen von Traces, welches ähnlich funktioniert, wie das Inlinen von Methoden in einem methodenbasierten JIT Compiler. Inlining erhöht die Optimierungsmöglichkeiten des Compilers und ermöglicht es, mehrere Methoden gemeinsam zu optimieren. Dies wiederum erhöht die Ausführungsgeschwindigkeit des generierten Maschinencodes. Der große Vorteil von Trace Inlining gegenüber dem Inlining von Methoden ist, dass die Trace-Information kontextabhängig ist. Wird an mehreren Programmstellen die gleiche Methode aufgerufen, so kann jeder Methodenaufruf gezielt durch die nötigen Teile der gerufenen Methode ersetzt werden. Dies führt zu besser optimiertem und weniger generiertem Maschinencode sowie zu einer schnelleren Kompilierung.

Programmen die mit unserem tracebasierten Compiler übersetzt wurden, werden um bis zu 59% schneller ausgeführt als bei einer Übersetzung mit dem HotSpot Client Compiler. Bei einigen Benchmarks übertrifft unser tracebasierter Compiler sogar den HotSpot Server Compiler, obwohl unser Compiler deutlich weniger Optimierungen durchführt.

Contents

Chapter 1

Introduction

Today, it is common to develop Java applications in an object-oriented way so that complex functionality is split up into a large number of classes and methods that are usually organized in class hierarchies. This abstraction simplifies development but may result in reduced performance because more indirection steps are necessary and virtual method calls must be executed more frequently. To increase the performance, the just-in-time (JIT) compiler must remove as much abstraction as possible when translating Java applications to machine code.

1.1 Java

Java [35] is an object-oriented programming language with a syntax similar to C++. However, complex and error prone C++ features such as multiple inheritance and pointer arithmetic were omitted. Another simplification is that Java uses a garbage collector (GC) for automatic memory management so that it is not necessary to deallocate memory explicitly. Java also comes with a large standard class library that provides implementations for commonly used data structures and operating system functionality. This greatly simplifies the life of developers and provides a good ecosystem for application development.

Figure 1.1 shows that unlike C++, Java is not compiled to machine code but instead to Java bytecode using an ahead-of-time (AOT) compiler such as *javac*. For every compiled Java class or interface, there is one Java class file that contains the bytecode and additional meta data such as information about methods, exception handlers, and constants. A Java application is then executed on top of a Java virtual machine (JVM) [54] that loads the application's class files and executes its bytecode. This extra layer of abstraction ensures that Java applications are platform independent and can execute on a large variety of systems such as mobile phones, desktop computers, and large servers.

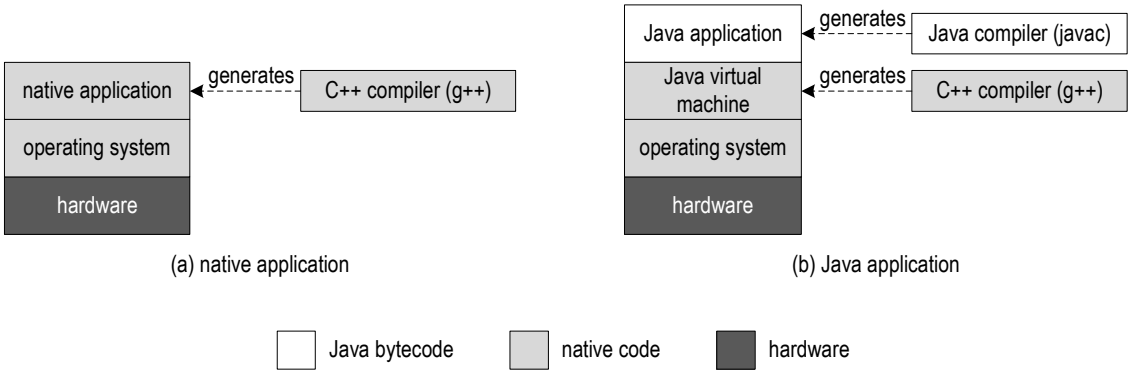


Figure 1.1: Layers involved when executing native and Java code

Figure 1.2 (a) shows a simple method that returns the sum of two integer values. When a Java AOT compiler, such as *javac*, compiles the source code to a class file, the bytecode in Figure 1.2 (b) is generated. The first byte of every JVM instruction is the opcode which defines the operation that should be executed. Currently, the JVM specification defines 202 of the 256 possible opcodes [54]. The abstract concept behind a JVM is a stack machine where the opcode defines how to manipulate the operand stack by pushing and popping values. Figure 1.2 (c) shows that `iadd` pops two integer values from the operand stack and pushes their sum back onto the operand stack.

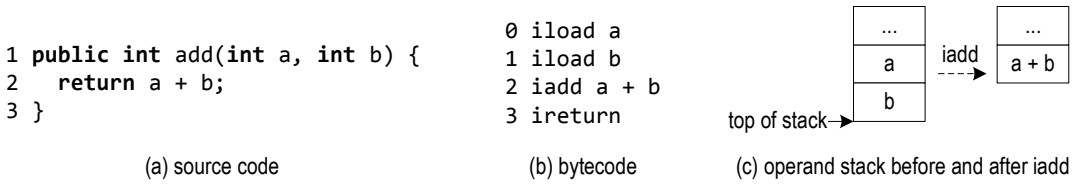


Figure 1.2: Java example

During execution, a JVM may collect information about the run-time behavior of the executed Java application. For example, it may detect frequently executed methods or may record how often certain method parts are executed. This information can then be used to trigger just-in-time (JIT) compilation for frequently executed code. The JIT compiler analyzes the executed code, applies optimizations, and generates platform-specific machine code that is then executed on the bare machine instead. During compilation, the JIT compiler can use information that was recorded about the application behavior to guide its optimizations. This may result in better optimized code that executes faster.

1.2 Motivation and Contributions

While traditional JIT compilation focuses on the compilation of whole methods, trace-based compilation compiles frequently executed paths, so-called traces [9]. A trace describes one specific path through a sequence of instructions and is not necessarily limited to the scope of a method. Compiling traces instead of methods can result in less generated machine code, faster compilation, and a higher performance.

Previously, trace compilation was successfully used for binary translation and dynamically typed languages such as JavaScript (see Chapter 9). *TraceMonkey* was the first JIT compiler for JavaScript. It also used trace compilation and started as a research project at the University of California, Irvine [17]. Later on it was integrated into Mozilla's *Firefox* browser. After Google's success with its method-based JavaScript JIT compiler *V8* [34], tracing compilers for JavaScript became unpopular and Mozilla decided to replace *TraceMonkey* with the method-based JIT compiler *IonMonkey* [56]. To the best of our knowledge, all of today's production quality JavaScript JIT compilers use a method-based approach. Trace compilation was also tried for statically typed languages, such as Java, but tracing compilers could not compete with production-quality method-based compilers in terms of peak performance.

We chose to implement a trace-based JIT compiler for Java by modifying the Java HotSpot VM which is one of today's most popular production quality JVMs. Unlike previous trace-based JIT compilers, we do not focus on just optimizing loop-intensive code, because most larger Java applications are not particularly loop intensive. Instead, we combine tracing ideas with ideas from method-based compilation so that the resulting trace-based JIT compiler achieves good speedups on large applications that are not loop intensive. This thesis contributes the following:

- We present a scope-based trace recording variant that supports all Java features such as exception handling, Java subroutines, reflection and invocation of native methods within traces.

- We describe the required changes for supporting trace recording and trace-based compilation in a production quality VM (the Java HotSpot VM) that originally uses a mature method-based JIT compiler.

- We establish loops as top-level compilation units to allow complete separation of compiled method traces and compiled loop traces. We discuss and address issues that arise from this separation and we compare our system to other trace-based and method-based JIT compilers. The resulting trace transitioning system is flexible enough to allow arbitrary transitions between interpreted and compiled code.

- We describe how to perform trace inlining and show its advantages when compared to method inlining. Furthermore, we evaluated multiple trace inlining heuristics implemented for our trace-based JIT compiler. We also evaluate which high-level compiler optimizations do benefit from trace inlining due to the widened compilation scope.
- We present a generalized exception handling approach so that exceptions can be thrown and caught efficiently within traces or across trace boundaries.
- Our extensive evaluation shows the impact of trace compilation on startup performance, peak performance, size of generated machine code, and compilation time for the benchmark suites SPECjbb2005 [63], SPECjvm2008 [64], and DaCapo 9.12 Bach [13]. All results are compared to the method-based Java HotSpot client and server compilers.
- We propose a runtime system where it is possible to derive accurate code coverage information from profiling data that was originally intended for guiding JIT compiler optimizations. This technique has very low impact on peak performance and is applicable to all modern VMs that have an aggressive JIT compiler and use instrumentation for recording profiling data. We compare our approach to other commonly used code coverage approaches and discuss its advantages and drawbacks. Furthermore, we evaluate our approach and compare it to a state-of-the-art code coverage tool for Java that uses bytecode instrumentation.

1.3 Structure of the Thesis

During the development of this thesis, several papers were published that document certain parts of the implementation [37–41]. Significant parts of these papers are integrated into various chapters of this thesis.

Chapter 2 gives an overview over the Java HotSpot VM and its components such as the interpreter and the JIT compilers.

Chapter 3 deals particularly with the Java HotSpot client compiler on which we based our trace-based JIT compiler. It covers the client compiler's intermediate representation and its most important optimizations as those are inherited by our trace-based JIT compiler.

Our novel scope-based trace recording approach is described in Chapter 4. The decision to record short traces and to use trace linking instead of recording long traces is crucial for our approach. It delays the inlining decision to the time of JIT compilation when more information is available than during trace recording. This simplifies inlining decisions and makes trace inlining our most profitable optimization.

Chapter 5 focuses on the changes that we had to apply to the method-based Java HotSpot client compiler while reshaping it into a trace-based JIT compiler. This chapter also mentions additional optimizations that our compiler performs to reach good peak performance.

Trace inlining, which is our key optimization, is discussed in Chapter 6. This chapter also compares trace inlining to method inlining and describes why trace inlining is more powerful and why it affects performance significantly.

Chapter 7 focuses on a method to derive code coverage information from recorded traces. Using the profiling data that is intended for the JIT compiler to derive exact code coverage information eliminates the need of instrumenting the executed application. This ensures that the impact on peak performance is minimal.

Our trace-based compiler is evaluated in Chapter 8. For the evaluation, we report numbers for multiple benchmark suites and compare our trace-based compiler to both the Java HotSpot client and server compilers. Especially larger object-oriented applications profit from our approach to trace-based compilation. Furthermore, we evaluate our novel technique for deriving exact code coverage from the recorded traces.

Work related to the topics in this thesis is presented in Chapter 9 and we describe similarities and differences between our and their work.

Chapter 10 concludes this thesis and discusses the outcomings as well as future work.

Chapter 2

The Java HotSpot Virtual Machine

The Java HotSpot VM is a production-quality JVM that is being developed by Oracle Corporation (formerly Sun Microsystems). It is available for multiple platforms such as Windows, Linux, and Mac OS, and supports several processor architectures such as x86 and SPARC.

Figure 2.1 shows the runtime system of the Java HotSpot VM. Execution of a Java application starts with class loading. The class loader loads the class files, parses and verifies them, and builds run-time data structures such as the constant pool or method objects. Then, the interpreter starts executing the bytecodes. Whenever the interpreter executes a method, it increments the method's invocation counter to detect so-called hotspots. When the invocation counter exceeds a certain threshold, the JIT compiler compiles the method to optimized machine code. Subsequent invocations of the method directly invoke this machine code instead of interpreting the bytecodes.

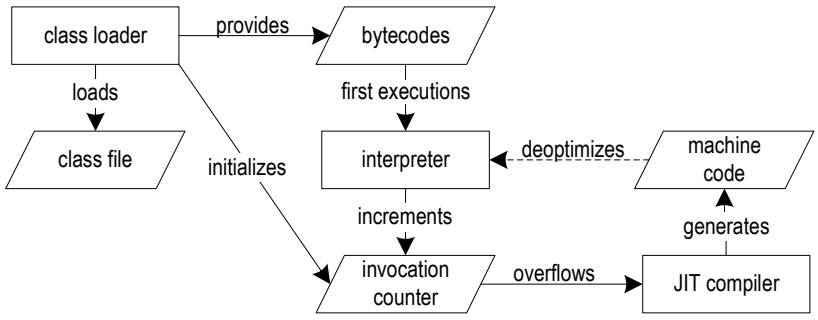


Figure 2.1: System overview

The Java HotSpot VM comes with two different JIT compilers: the *client compiler* and the *server compiler*. Usually, one of them is selected as the sole JIT compiler, however there is also a *tiered* mode where hot methods are first compiled using the client compiler and later on using the server compiler.

While the bytecode is executed in the interpreter, the VM may also record profiling data about the application's behavior. This profiling data is used to guide optimistic optimizations during JIT compilation. However, during later executions it may be necessary to invalidate such optimistic optimizations because the previously recorded profiling data no longer matches the application's behavior. In that case, the VM invalidates the machine code that contains this optimistic optimization and falls back (deoptimizes) to the interpreter.

2.1 Interpreter

For the initial executions, the Java HotSpot VM uses a highly efficient interpreter implementation [36]. This interpreter consists of hand-written assembler templates for every Java opcode. The machine code for this interpreter is generated during VM startup from the assembler templates. Infrequently executed and complex operations are implemented as calls to the C-based runtime of the interpreter.

The interpreter is instrumented to record profiling data for frequently executed methods. However, this instrumentation is only embedded in the generated interpreter machine code if the server compiler is used for JIT compilation because the client compiler does not use any profiling data. In the following, the most important instrumented instructions are listed:

- The interpreter records for *branches* and *switch instructions* how often the instruction jumped to which branch target. This information helps determining frequently executed code parts within methods so that those parts can be optimized more aggressively. Furthermore, the JIT compiler may exclude code parts from compilation that were not executed during profiling. This increases performance while decreasing the amount of generated machine code and the time required for JIT compilation.

- For *virtual* and *interface calls*, a maximum of two receiver types and their frequencies are recorded. If no more than two receiver types have been seen for a call, then it is often profitable to inline the invoked method(s) optimistically.

- For *array stores*, *typecasts* and *instanceof* instructions the seen types and their frequencies are recorded. This is profitable as the JIT compiler may use that information to optimistically simplify type checks.

Figure 2.2 shows that the profiling data for a method is stored in a contiguous block of memory where space is reserved for every instrumented instruction. During profiling, the interpreter holds a pointer into the block of profiling data for the currently executing method. This pointer is updated at every branching instruction so that it always points

to the profiling data for the next instrumented instruction. Recording the profiling data and keeping the pointer into the profiling data in sync with the execution decreases the performance of the interpreter and therefore impacts startup performance. However, after JIT compilation, this is outweighed by the increased peak performance because the JIT compiler can generate better optimized machine code.

Figure 2.2: Java HotSpot VM profiling data

2.2 Just-in-time Compilers

Oracle's Java HotSpot VM ships with two different JIT compilers that share most parts of the VM infrastructure. The *client compiler* is designed for best startup performance and implements only basic optimizations to achieve a decent peak performance [50]. It splits method compilation into three phases: generation of the high-level intermediate representation (HIR), generation of the low-level intermediate representation (LIR), and code generation. Compilation starts with generating the HIR that is in static single assignment (SSA) form [22] and represents the control flow graph. On this level, optimizations such as constant folding, null-check elimination, and method inlining are applied. The resulting optimized HIR is used to generate the LIR, which is close to machine code but still mostly platform independent. Virtual registers are used instead of physical machine registers for nearly all instructions. Then, linear scan register allocation [70] maps the virtual registers to physical ones. Finally, code generation translates every LIR instruction to platform-dependent machine code.

The *server compiler* is designed for long-running server applications and produces highly efficient code to reach best-possible peak performance [60]. Servers execute mostly long-running applications so that the longer compilation time is only a small overhead when considering the total execution time. In comparison to the client compiler, the server compiler performs many additional optimizations such as escape analysis, loop invariant code motion, and loop unrolling.

It is also possible to combine both JIT compilers in a so-called *tiered* mode where hot methods are initially compiled with the client compiler. The generated machine code is instrumented to record further profiling data but executes significantly faster than the interpreter. If a method is sufficiently hot, then it is recompiled later on using the server compiler. The server compiler guides its optimizations with the recorded profiling data to

reach best-possible peak performance. With tiered compilation it is therefore possible to achieve both good startup and good peak performance.

Both JIT compilers use optimistic optimizations to generate better optimized machine code. Calls to virtual or interface methods occur frequently in Java so that one of the most important optimistic optimizations is method inlining. The JIT compilers use class hierarchy analysis (CHA) [23] to check if a method is not overridden by any loaded subclass as the call can be inlined optimistically in this case. If a subclass is loaded later on that overrides an optimistically inlined method, the previously performed inlining must be voided. The Java HotSpot VM deoptimizes affected methods that are currently being executed, so that their execution continues in the interpreter [45].

2.3 Memory Management

Java uses a garbage collector (GC) for automatic memory management [48] to avoid the error-prone manual memory management. The Java HotSpot VM features several generational GCs with different strength and key characteristics [55]. Which GC performs best depends on factors such as the heap size and the number of available cores/processors.

- The *serial collector* is a stop-the-world GC (i.e., it stops all application threads during garbage collection) that uses only one thread for garbage collection. So, it is best suited for single-core machines or for applications with a small heap.

- The *parallel collector* is optimized for throughput and uses multiple threads for garbage collection but is otherwise still a stop-the-world GC. On machines with multiple hardware threads, garbage collections are significantly faster than with the serial collector. However, with large heaps the long pause-time for a full garbage collection can still be an issue. This especially applies to interactive applications that are expected to respond to user input.

- If a low maximum pause-time is important, it is possible to use the *concurrent mark-sweep collector (CMS)*. It performs many small garbage collections concurrently to the running application so that full garbage collections are avoided as much as possible. However, the additional concurrency is an overhead that results in a lower throughput performance.

- Another concurrent GC is the *garbage-first collector (G1)* [24]. It is best suited for large heaps and machines with multiple CPUs and aims for a low pause time similar to CMS. However, unlike CMS it does compact the heap so that it should replace CMS eventually.

2.4 Deoptimization

Deoptimization [45] is the process of falling back to the interpreter from compiled code. Figure 2.3 shows the machine stack before, during, and after a deoptimization. Initially, there is a compiled method on the stack as shown in Figure 2.3 (a). When deoptimization is requested for a thread, all values live in the current compiled method frame are rescued to the heap. Then, the compiled stack frame is popped, which results in the stack shown in Figure 2.3 (b). Depending on the inlining depth of the currently executed instruction, one or multiple interpreter frames are pushed on the stack. In the final step, those interpreter frames are filled with the values that were previously rescued to the heap, resulting in the stack shown in Figure 2.3 (c). After that, the execution continues with the latest values in the interpreter.

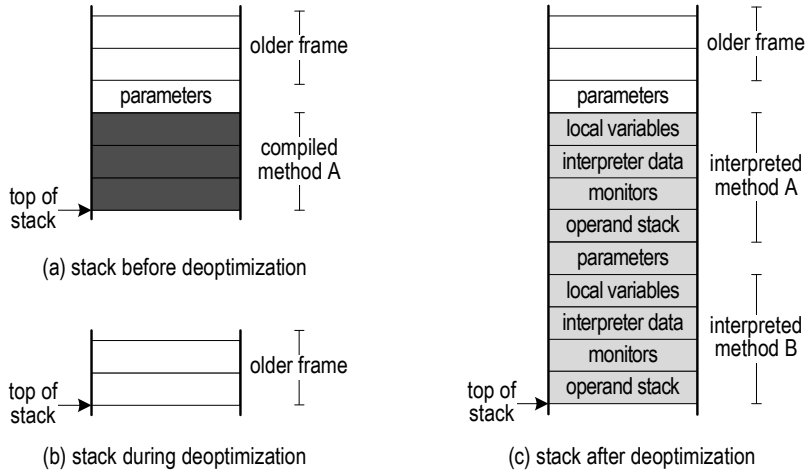


Figure 2.3: Deoptimization

For deoptimization, exact information is necessary that indicates which values the compiled code stores in registers and stack slots. This information is stored in a compact way in the debugging information that the JIT compiler generates while emitting machine code. Using this information, the necessary interpreter stack frames can be reconstructed and filled.

One important example where deoptimization might occur is when the JIT compiler uses profiling data to perform optimistic optimizations. Profiling data always has a certain degree of uncertainty as it is only recorded during a certain period of execution. Optimizations that are based on the profiling data may be invalidated because the application behavior may change over time so that the profiling data does no longer reflect the application behavior. In such a case, the compiled code is discarded and it deoptimizes to the interpreter. Then, the profiling data is updated and a new compilation is triggered after

spending some time in the interpreter. This compilation uses the updated profiling data so that the optimizations can be adjusted to the new application behavior.

Another optimistic optimization is inlining, although it does not necessarily rely on profiling data. The JIT compiler uses CHA to check if a certain method is not overridden by any subclass. If this is not the case, the method can be inlined optimistically. When the compiler does the inlining, it stores additional meta data about the assumption on the class hierarchy together with the generated machine code. If a class is loaded later on that violates this assumption by overriding the inlined method, the machine code is invalidated. Furthermore, the affected machine code deoptimizes to the interpreter if it is currently being executed.

Debugging also relies on deoptimization. If a breakpoint is placed in a method that was already compiled, it is necessary to invalidate the method's machine code and if the code is currently executing it deoptimizes to the interpreter. This ensures that the application runs with full speed when no break points are present but allows adding breakpoints at arbitrary source code locations during any point of execution.

Chapter 3

The Client Compiler

The Java HotSpot client compiler is designed for best startup performance and implements only basic optimizations to achieve a decent peak performance [50]. It splits method compilation into three phases: generation of the high-level intermediate representation (HIR), generation of the low-level intermediate representation (LIR), and code generation.

3.1 Front End

The client compiler's high-level intermediate representation (HIR) is in SSA form [22] and is structured as a control flow graph (CFG). When compiling a method, the compiler first iterates all control-flow-changing bytecodes and uses that information to build a control flow graph with the corresponding basic block structure. The blocks of the resulting control flow graph are then processed in reverse-post-dominator order, as shown in Figure 3.1. For every block, the JIT compiler then iterates over the corresponding bytecodes and generates HIR instructions.

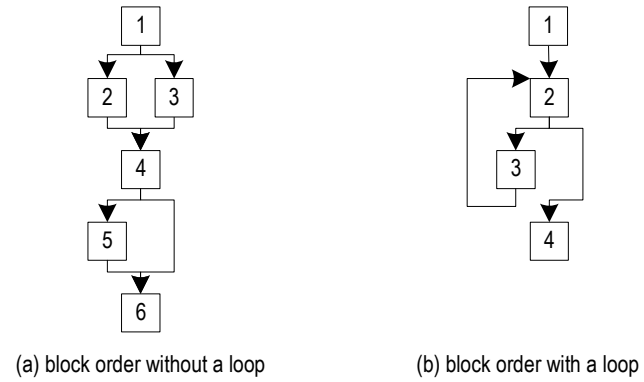


Figure 3.1: Reverse-post-dominator processing order

The reverse-post-dominator order ensures that every block can rely on the fact that all its predecessor blocks have already been processed. This greatly simplifies data-flow analysis and the generation of ϕ instructions when the control flow merges. The only exception are loop headers, for which ϕ instructions are generated eagerly for all variables that are modified in the loop. Unnecessary ϕ instructions are then eliminated later on in a separate step.

The Java HotSpot client compiler applies optimizations such as constant folding, load elimination, store elimination, and value numbering locally while parsing a block. This limits the scope of the optimization but also has the effect that little time is spent on JIT compilation as local optimizations can be applied efficiently. In a few cases, for example when inlining a small method, consecutive control flow graph blocks are merged, which increases the scope and therefore the effectiveness of local optimizations. After parsing all bytecodes, null-check elimination, conditional elimination, block merging, and global value numbering are applied globally. Furthermore, minor loop optimizations are applied, e.g., converting the conditional forwards jump of a loop to a backwards jump. After those optimizations, the control flow in the optimized HIR is frozen so that no further changes to the control flow are allowed on the lower level. This simplifies the back end.

Exception-throwing bytecodes do not end basic blocks as this would result in many short basic blocks. Instead, all exception handlers are recorded in an exception handler table. If an exception occurs at run time, the exception handler table is accessed and if there is a corresponding exception handler, execution continues there. Otherwise, the current frame is popped from the stack and the search for an exception handler continues in the parent frame.

3.2 Back End

The low-level intermediate representation (LIR) is generated from the optimized HIR by iterating it in reverse-post-dominator order and mapping each HIR instruction to a number of LIR instructions. It is still mainly platform independent as virtual registers are used instead of physical machine registers. For certain platform-dependent parts, such as calling conventions, it is possible to use fixed platform-dependent registers. Eventually, linear scan register allocation [70] maps the virtual registers to physical ones. Unlike the HIR, the LIR is not in SSA form so that ϕ instructions are resolved using move instructions. For that, the register allocator contains some optimizations that try to detect and avoid unnecessary moves.

Code generation iterates over all LIR instructions and translates every LIR instruction to platform-dependent instructions. On this level, optimizations such as template-based

instruction selection are applied to generate efficient machine code. In addition to machine code, debugging information is generated which is used for garbage collection and deoptimization.

3.3 Optimizations

The Java HotSpot client compiler mostly uses local optimizations. This ensures that little time is spent on JIT compilation as local optimizations can be applied efficiently. However, this also limits the scope of optimizations which has a negative impact on the quality of the generated machine code. To maximize the startup performance, no profiling data is gathered in the interpreter when the client compiler is selected as the JIT compiler. However, this also means that the client compiler does not guide its optimizations with profiling data.

The client compiler performs the following optimizations: method inlining, canonicalization (constant folding and elimination of type checks), load/store elimination, conditional expression elimination, common subexpression elimination, dead code elimination, value numbering (locally and globally), and null check elimination. Furthermore, it performs a limited form of array bounds check elimination. In a more recent early access version of OpenJDK 8, the array bounds check elimination became more powerful as the technique of Würthinger et al. [72] was integrated. However the work presented in this PhD thesis is based on the version with the limited array bounds check elimination.

The most profitable optimization of the Java HotSpot client compiler is method inlining which replaces calls with copies of the actually called code. This removes the overhead of the call, increases the compilation scope and therefore also increases the effectiveness of optimizations. Inlining heuristics can be categorized into static and dynamic approaches. While static inlining heuristics rely on static metrics such as the callee size, dynamic inlining heuristics use profiling data to decide if a call is worth inlining.

The Java HotSpot client compiler uses a simple static method inlining heuristic that compares the callee's number of bytecodes to a fixed limit that decreases with the inlining depth. This approach avoids a time-consuming analysis by assuming that the inlining of each call can be decided independently. Virtual methods and interface methods are only considered for inlining if CHA reveals that only one particular target method is possible in the specific context. For such inlinings, an assumption is stored together with the generated machine code. If a class is loaded later on that violates this assumption, the generated machine code is invalidated, see Chapter 2.4.

In contrast to that, the Java HotSpot server compiler uses profiling data to guide many of its local and global optimizations. Furthermore, it applies sophisticated loop optimizations

such as loop peeling, loop unrolling and loop-invariant code motion. Another feature of the server compiler that the client compiler lacks is escape analysis [21]. Escape analysis enables optimizations such as scalar replacement and lock elimination. Those optimizations and the graph coloring register allocator ensure that best-possible code is generated. However, this is also the reason why the server compiler requires roughly one order of a magnitude more time for JIT compilation than the client compiler.

Chapter 4

Trace Recording

Figure 4.1 (a) shows the control flow graphs of two methods, while Figure 4.1 (b) shows three traces that span both methods. All traces start at block 1, which is their *trace anchor*. Which basic blocks are selected as trace anchors depends on the trace recording implementation as there are multiple ways for detecting trace anchors [12, 27, 44].

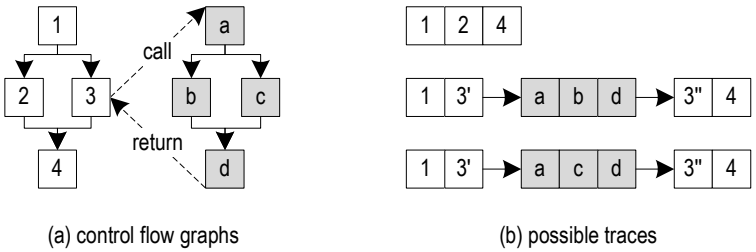


Figure 4.1: Possible traces through two methods

4.1 Overview

In a VM, traces can be recorded by instrumenting bytecode execution. Our implementation supports two different kinds of traces: *loop traces* anchored at loop headers, and *method traces* anchored at method entries. Figure 4.2 shows the runtime system of our trace-based VM. Execution of a Java application starts with the class loader that loads, parses, and verifies the class files. This results in run-time data structures such as the constant pool and method objects. Then, a static loop analysis step is performed on the loaded bytecodes to detect loop headers and to create tracing-specific data structures. This explicit loop detection step avoids the detection of false loops [42], which may occur in other approaches where every backwards branch target is assumed to be a potential loop header. The detected loop headers and all method entries are then used as potential trace anchors.

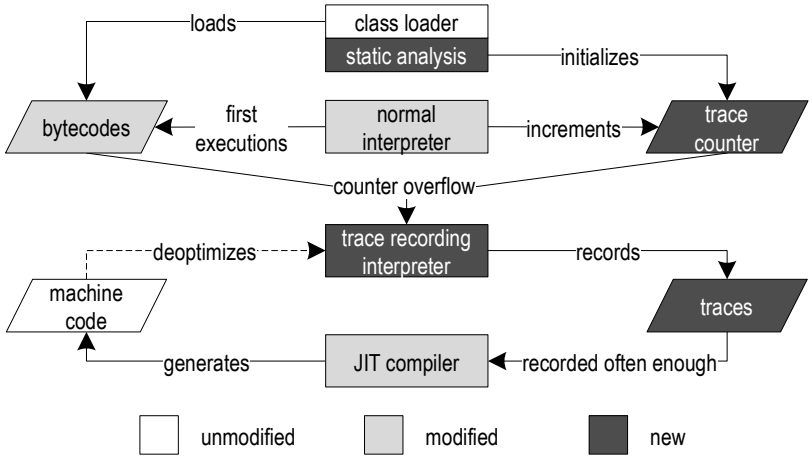


Figure 4.2: Trace-based Java HotSpot VM

The Java HotSpot VM template interpreter is a highly efficient interpreter implementation and consists of hand-written assembler templates (see Chapter 2.1). Our trace-based Java HotSpot VM uses two different versions of this template-based interpreter, for which we duplicated and instrumented the interpreter. This results in a normal and a trace recording interpreter. Execution starts in the *normal interpreter*, which counts the execution frequencies of trace anchors. When a trace anchor was executed frequently enough, it is marked as hot and execution switches to the *trace recording interpreter*. This interpreter is instrumented to record the executed path and the observed type information in a thread-local buffer. Traces that start at this hot anchor are then recorded for a certain number of times until they are considered hot and therefore compiled to optimized machine code. When execution reaches a trace anchor for which compiled machine code already exists, this machine code is invoked instead of interpreting the bytecodes. Switching between the normal and the trace recording interpreter avoids trace recording for rarely executed code parts and allows executing bytecodes at nearly the same speed as the interpreter of the unmodified VM.

4.2 Bytecode Preprocessing

Traditionally, trace recording treats every backwards branch target as a possible loop header [9]. However, this may lead to the detection of false loops, which can affect the performance negatively. To accurately detect loop headers, we do a dedicated bytecode preprocessing step after class loading. For every method, we temporarily build a control flow graph and run a loop detection algorithm on that graph. The bytecode preprocessing performs the following steps:

1. Loop headers are detected using the method's control flow graph and for every loop header, a data structure describing the loop is allocated.
2. Basic blocks are assigned to the loops in which they are contained.
3. The relationships between loops are computed so that inner loops are correctly nested in the corresponding outer loop.
4. Loop exits are computed.
5. Additional metrics are calculated. This includes a complexity metric that estimates the maximum number of different paths that can be taken through the loop.

The preprocessing step is performed only once per class and is cheap when compared to other tasks performed during class loading such as verification. For every processed method, this results in a set of loop-describing data structures. Every data structure contains the loop header's bytecode index (BCI), the BCIs of all loop exits and additional information such as the complexity metric or the number of inner loops. This information is stored in a compact way while all other data used during bytecode preprocessing is discarded.

To reduce the tracing overhead for the interpreter, we mark loop headers directly in the bytecodes. For that, we introduce a new VM internal bytecode `loop_header`. Introducing VM internal bytecodes for optimized operations is a common pattern and those bytecodes may use opcodes that are unused according to the JVM specification [54]. To preserve the original bytecode, we rescue it to the data structure that describes the loop before replacing the original bytecode with the bytecode `loop_header`.

From the interpreter's point of view, detecting loop headers is now trivial as every loop header is marked by the bytecode `loop_header`. Whenever the bytecode `loop_header` is executed, the interpreter first executes trace-anchor-specific tasks such as incrementing an execution counter. Then, the original bytecode is obtained from the corresponding loop's data structure and this bytecode is executed.

4.3 Normal Interpreter

The normal interpreter does not perform any trace recording and executes bytecodes at nearly the same speed as the interpreter of the unmodified VM. To avoid trace recording for rarely executed traces, each trace anchor has an execution counter that is incremented by the normal interpreter. When the execution counter overflows, the trace anchor is marked as hot and execution switches to the trace recording interpreter. During garbage collection, the trace anchor execution counter is cleared regularly to avoid rarely executed

parts being falsely identified as hot on long-running applications. If compiled machine code does already exist for the executed trace anchor, the interpreter invokes the machine code instead of interpreting the bytecodes.

4.4 Trace Recording Interpreter

Other trace recording implementations [9, 12, 14, 17, 30–32, 47] typically store information about encountered bytecodes in a single trace, even if this trace crosses method boundaries. Thus, inlining is done during trace recording and may result in large traces that cannot be split and must be compiled as a whole. Furthermore, other trace recording implementations often abort trace recording on special events such as when the executed application throws an exception.

To address these limitations, we use a *scope-based* trace recording approach where traces are restricted to span at most one method. When a method invocation is encountered during trace recording, a separate trace is started for the invoked method and this separate trace is linked to its caller trace. This restricts individual traces to span at most one method while the linking preserves the call structure and also makes the traces context-sensitive so that each call site knows exactly which traces were called there. The linking results in a data structure that is similar to a dynamic call graph and it allows delaying any inlining decisions to the time of JIT compilation when more information is available. Our trace recording approach supports all Java features such as exception handling, Java subroutines, reflection, and invocation of native methods. There are no situations in which we have to abort trace recording. For best-possible performance, all frequently executed trace recording operations, such as recording information for specific bytecodes, are directly implemented in the assembler templates of the trace recording interpreter. More complex operations, such as storing a recorded trace (see Figure 4.5) are implemented in the C-based runtime of the interpreter.

Our trace recording infrastructure supports efficient multi-threading so that every Java thread can switch between the normal and the trace recording interpreter independently. For trace recording, every thread holds a thread-local tracing stack, which is a virtual call stack of traces that is managed by the trace recording interpreter. Information about bytecodes that modify the control flow is stored in the topmost trace of the tracing stack and the tracing stack is modified if necessary (see below).

4.4.1 Tracing Stack

Figure 4.3 shows a trace recording example that focuses on the tracing stack and trace linking. The tracing stack shown in Figure 4.3 (b) grows from right to left. (1) When

```
 1 public class Adder {
 2   private int value;
 3
 4   public void addData(int[] data) {
 5     int value = getValue();
 6     for (int i = 0; i < data.length; i++) {
 7       if (data[i] < 0) value += Math.abs(i);
 8       else value += i;
 9     }
10     setValue(value);
11   }
12
13   public int getValue() {
14     return this.value;
15   }
16
17   public void setValue(int value) {
18     this.value = value;
19   }
20 }
21
22 public static void main(String[] args) {
23   int[] data = new int[] { 0, 1, -1 };
24   Adder adder = new Adder();
25   adder.addData(data);
26   System.out.println(adder.getValue());
27 }
```

(a) source code

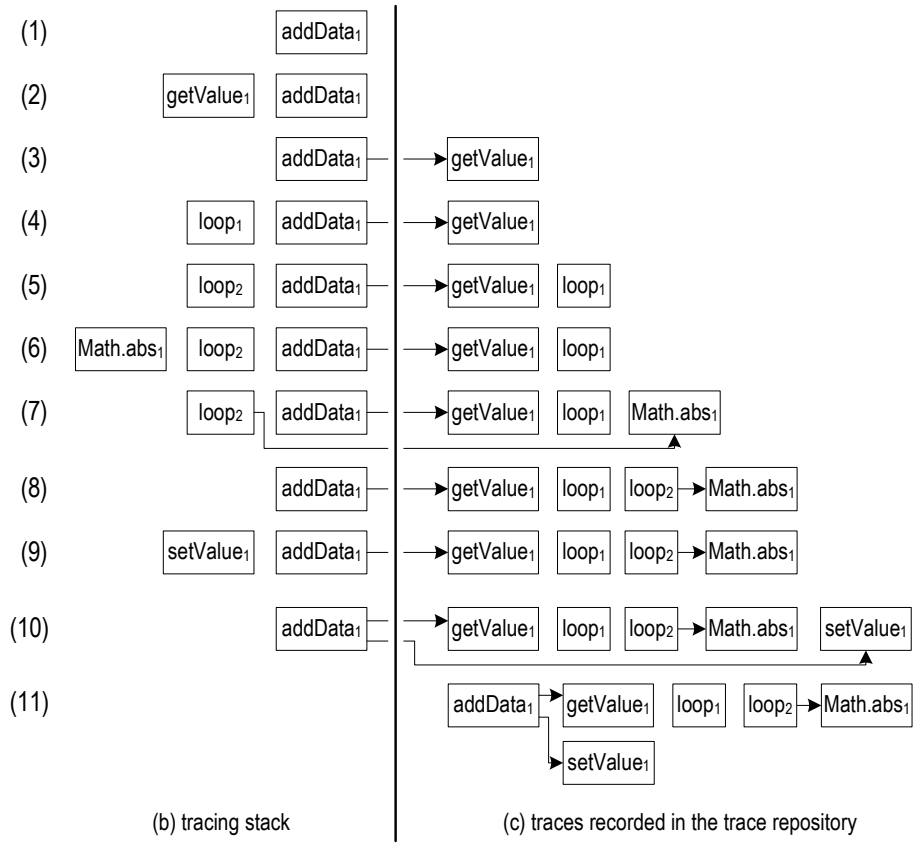


Figure 4.3: Tracing stack while trace recording

the trace anchor at the method entry of `addData()` is marked as hot, execution switches to the trace recording interpreter and a method trace is pushed on the tracing stack. The method is executed from the beginning up to the invocation of the virtual method `getValue()`. When doing the virtual call, the invocation and the receiver class are stored in the caller trace. (2) Upon entering the method `getValue()`, a new method trace is pushed on the tracing stack and trace recording continues there. (3) When `getValue()` returns, the corresponding trace is popped from the tracing stack and stored in the trace repository. Then, the traces are linked by storing a pointer to the trace of `getValue()` in the trace of `addData()`. Execution and trace recording continues for `addData()` and reaches the loop header. (4) For recording the loop, a new loop trace is pushed on the tracing stack. (5) After the first loop iteration, when execution is back at the loop header, the loop trace is popped from the tracing stack and stored. For the next loop iteration, a new loop trace is pushed on the tracing stack. The second loop iteration executes the same path as the first iteration, so the system recognizes that the same trace was already recorded and does not store it again but only increments the counter within the previously recorded trace. (6) The third loop iteration takes a different path so that the method `Math.abs()` is invoked for which a new method trace is pushed on the tracing stack. (7) When `Math.abs()` returns, the corresponding trace is stored and linked to its caller trace. (8) Then, execution reaches the loop header and the loop exits. So, the loop trace is popped from the tracing stack and stored. (9) After the loop, the virtual method `setValue()` is invoked. So, the invocation and the receiver class are stored in the caller trace, and a new method trace is pushed on the tracing stack upon entering `setValue()`. (10) When `setValue()` returns, the corresponding trace is popped from the tracing stack, stored, and linked to its caller trace. (11) Eventually, the method `addData()` returns so that also this trace is popped from the tracing stack and stored. After that, the tracing stack is empty and execution switches back to the normal interpreter.

In the example above, it was assumed that no traces had been compiled for the invoked methods and the loop. If traces for the method `getValue()` had already been compiled earlier, the invocation of `getValue()` would execute the compiled machine code instead of interpreting the method. So, the trace recording interpreter cannot push a new method trace on the tracing stack, nor can it record any control flow in the invoked method. In that case, our trace recording approach does not preserve exact control flow information over method boundaries. It would be possible not to invoke compiled code and instead force this code to be executed in the trace recording interpreter if a trace is currently being recorded. However, this would drastically reduce the startup performance because the application would be interpreted for a significantly longer time. Furthermore, as shown in the previous example, loop traces are never linked to their caller so that exact control flow information is also not preserved in that case. The same also applies to recursive method invocations, which we also do not link to their parent trace.

4.4.2 Recorded Trace Information

In the trace recording interpreter, certain bytecodes are instrumented so that information about them is recorded in the traces. All other bytecodes are processed in the same way as in the normal interpreter.

- For every branching bytecode, including jump subroutine (`jsr`) and return from subroutine (`ret`), we store the target BCI.
- For non-virtual method invocations, we store the BCI of the invocation, and a pointer to the callee. Upon entering the callee, a new method trace is pushed on the tracing stack and recording continues there. When the callee returns, we store the method trace and pop it from the tracing stack. Then, we link the traces by storing a pointer to the callee's trace in the caller's trace and trace recording continues for the caller.
- For the invocation of virtual methods and interface methods, we additionally store the class of the receiver before doing the same steps as for a non-virtual method invocation.
- If a loop header is encountered when a trace for that loop is not already on top of the tracing stack, we store the current BCI and a pointer to the data structure that describes the loop. Then, we push a new loop trace on the tracing stack and recording continues there. When a loop exit is taken, we store the loop trace, pop it from the tracing stack, and trace recording continues for the outer trace.
- If the current trace is a loop trace and execution looped back to the loop header, the current trace is stored and removed from the tracing stack. A new loop trace may be started for the next loop iteration if trace recording should be continued.
- For exceptions, we store the exception source BCI, the exception handler BCI, and a pointer to the class of the thrown exception. Because a thrown exception may exit the current method or loop, we pop traces from our tracing stack until the topmost trace is the one in which exception handling continues or until our tracing stack is empty. Then, we append information about the exception to all popped traces and store those traces.

Figure 4.4 shows a trace recording example that focuses on the information that is recorded for instrumented bytecodes. All bytecodes that are executed during the following example are in black type, while unexecuted bytecodes are in gray type.

When the trace anchor at the method entry of `accessArray()` is marked as hot, execution switches to the trace recording interpreter and a method trace is pushed on the tracing stack. The method is executed from the beginning and the first recorded instruction is

```
int accessArray(int val, int[] arr)
 0 iconst 0
 1 istore index
 2 goto 14
 5 iload index
 6 invokestatic computeIndex()
 9 iadd
10 istore index
11 iinc val by -1
14 loop_header (iload val)
15 ifge jump to 5
18 aload this
19 iload index
20 invokevirtual modifyIndex()
23 istore index
24 aload arr
25 iload index
26 iaload arr[index]
27 ireturn
28 astore exception
30 iconst -1
31 ireturn

int modifyIndex(int value)
 0 iload value
 1 iflt jump to 10
 4 iload value
 5 bipush 20
 7 if_icmple jump to 12
10 iconst 0
11 ireturn
12 iload value
13 ireturn
```

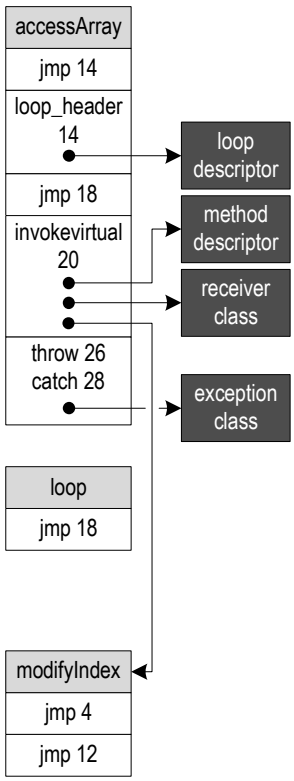


Figure 4.4: Recorded traces

the jump to BCI 14. There, a loop header is located, which is recorded together with a pointer to the corresponding loop descriptor. For recording the loop, a new loop trace is pushed on the tracing stack. Assuming that the loop is not executed but immediately left by falling through to BCI 18, only this BCI is recorded. Then, the loop trace is stored, popped from the tracing stack and we are back in the trace of `accessArray()`. There, another jump to BCI 18 is recorded to let the caller trace know that the loop exited to BCI 18. BCI 20 invokes the virtual method `modifyIndex()` for which the current BCI and pointers to the invoked method and the receiver class are stored. Upon method entry, a new method trace is pushed on the tracing stack. During execution of the method, two branching instructions are encountered where the first one falls through to BCI 4 and the second one jumps to BCI 12. When `modifyIndex()` returns, the trace is stored and popped from the tracing stack. Then, the traces are linked by storing a pointer to the trace of `modifyIndex()` in the trace of `accessArray()`. Execution continues for `accessArray()` until BCI 26, where the array access throws an exception. The exception is caught by the exception handler at BCI 28 and the method returns after that. So, the trace is stored and popped from the tracing stack. Because there are no more traces on the tracing stack, execution switches back to the normal interpreter.

Figure 4.5 shows the function that is used to store recorded traces. For most trace anchors, only a small number of traces is recorded. So, storing a new trace is required rarely while in most cases only the execution count of an already recorded trace is incremented. Therefore, we store the recorded traces in a data structure that avoids locks and atomic instructions when data is read. When it seems that a new trace was found, we lock our data structure for other writing threads and recheck under the lock if this trace is really new before adding it to the recorded traces. So, for the most frequent case, we can avoid synchronization and atomic machine instructions, which significantly increases the trace recording performance for multi-threaded applications. If a not yet seen trace was recorded, the thread-local trace is copied to global accessible memory and stored in the list of all traces for that trace anchor.

```
1 void storeTrace(TraceAnchor traceAnchor, Trace trace) {
2  Trace existingTrace = traceAnchor.find(trace);
3
4  if (existingTrace == null) {
5    traceAnchor.lock();
6    try {
7      // recheck under lock
8      existingTrace = traceAnchor.find(trace);
9      if (existingTrace == null) {
10       Trace globalTrace = copyToGlobalMemory(trace);
11       traceAnchor.store(globalTrace);
12     } else {
13       existingTrace.incrementCount();
14     }
15   } finally {
16     traceAnchor.unlock();
17   }
18 } else {
19   existingTrace.incrementCount();
20 }
21 }
```

Figure 4.5: Code for storing a recorded trace

To simplify the detection of already recorded traces, each trace is identified by a hash code. This hash code is computed in a way that two traces which took the same path but invoked different traces have different hash codes. This ensures that context-sensitive information is preserved over method boundaries.

4.4.3 Thresholds

How often trace recording is performed for a trace anchor before the recorded traces are compiled to machine code depends on various aspects. Our runtime system uses the following thresholds to trigger trace recording and trace compilation:

- The *method tracing threshold* determines how often a method trace anchor has to be executed before the execution switches to the trace recording interpreter so that traces starting from the method entry are recorded. To avoid recording infrequently executed method traces, we wait 3,000 executions before we consider a method trace anchor as hot and start trace recording.

- The *method traces compilation threshold* determines how often traces are recorded for a trace anchor at a method entry before those traces are merged and compiled to optimized machine code. To ensure that we capture the most relevant traces, we perform trace recording up to 1,000 times before compilation (depending on the complexity metric described below).

- The *loop tracing threshold* determines how often a loop header has to be executed before the execution switches to the trace recording interpreter so that traces starting from the loop header are recorded. Loop headers are executed significantly more frequently than methods so that we wait 25,000 executions before we start trace recording.

- The *loop traces compilation threshold* determines how often traces are recorded for a loop header before the recorded traces are compiled to optimized machine code. Here, we also ensure that we capture the most relevant traces before compilation so that we perform trace recording up to 1.000 times before compilation.

Therefore, loops are compiled before their surrounding method if the loop body is executed at least 6.5 times per method invocation. At the first glance, the used thresholds may seem high but they are already lower than the thresholds that are used in the method-based HotSpot server compiler which compiles methods after 10,000 executions. We chose the high tracing thresholds to avoid compiling infrequently executed method and loop traces to machine code. The fairly high compilation thresholds have a different reason. Our trace-based JIT compiler uses the recorded trace information for optimistic optimizations and we tried to minimize the probability that those optimistic optimizations have to be invalidated at a later point of execution because of a change in the application behavior.

To increase the startup performance and to reduce the amount of trace recording, we use the complexity metric that is computed during bytecode preprocessing so that simple methods and loops are less frequently recorded than complex ones. This heuristic increases the startup performance without a significant effect on the quality of the recorded traces. If the trace recording interpreter encounters a trace anchor for which compiled machine code already exists, it invokes this machine code instead of interpreting the bytecodes.

4.5 Partial Traces

The trace-based JIT compiler only compiles method parts that are covered by traces and it may use the recorded traces for optimistic optimizations that are guarded with run-time checks. If code must be executed that was not compiled or if a run-time check of an optimistic optimization fails, we trigger an extended version of the Java HotSpot deoptimization mechanism so that execution falls back to the trace recording interpreter. When the trace recording interpreter takes over, it records a partial trace that directly starts at the point of deoptimization instead of at the trace anchor but which is still associated to its enclosing trace anchor. In every other aspect, there is no difference between normal traces and partial traces and both use the same trace recording mechanism. To detect too frequent deoptimization of compiled code, a counter is incremented every time a deoptimization occurs. After reaching a threshold, the compiled machine code is invalidated and another compilation is triggered that uses the originally recorded traces and all partial traces. The additional information from the partial traces is used to increase method coverage or to disable specific optimistic optimizations which in return reduces the deoptimization frequency. Trace recompilation is required infrequently as shown in Chapter 8.

Chapter 5

Trace-based Compilation

Trace-based compilation only compiles frequently executed paths, so-called traces, instead of whole methods to optimized machine code [9]. This can increase the peak performance, while reducing the amount of generated machine code and the time required for JIT compilation.

Our trace-based JIT compiler is based on the Java HotSpot client compiler because it has a simple structure so that it can be modified easily. Although all techniques are general enough to be also applicable to the server compiler, the complex structure of the server compiler is less approachable for the changes that are required for trace-based compilation, especially in the context of a research project.

Compiling the traces of a trace anchor to optimized machine code involves multiple steps. First, our trace-based compiler builds its HIR by merging the recorded traces into a trace graph. On this level, we remove traces that start at the trace anchor but have not been executed frequently enough. Then, all client compiler optimizations and tracing-specific optimizations such as aggressive trace inlining are applied to the HIR. The optimized HIR is used to generate the LIR that is used for register allocation and code generation. Eventually, linear scan register allocation maps virtual registers to physical ones. The LIR is then translated to optimized machine code which is invoked by one of the interpreters or by other compiled traces.

5.1 Front End

Figure 5.1 (a) shows the control flow graph (CFG) of a method, which is typically used as the HIR in a method-based compiler. Assuming that the traces in Figure 5.1 (b) were recorded, the traditional high-level intermediate representation for a trace-based JIT compiler is the trace tree [31] shown in Figure 5.1 (c). A trace tree does not allow any inner merge nodes but instead duplicates the control flow. This simplifies certain

optimizations but may lead to exponential growth of compiled code because of the control flow duplication. Figure 5.1 (d) shows that our trace-based JIT compiler uses a hybrid between a traditional control flow graph and a trace tree, a so-called trace graph, so that both control flow duplication and inner merge nodes are allowed. A trace graph allows duplicating code in cases where it makes sense while avoiding code bloat that may occur for trace trees. Merging the traces is explicitly done by the JIT compiler and not during trace recording, because by the time of JIT compilation all relevant traces have been recorded and more information is available. Information that was obtained during trace recording, such as the receiver class of a virtual call, are attached to the nodes of the trace graph, so that the compiler can directly access that information.

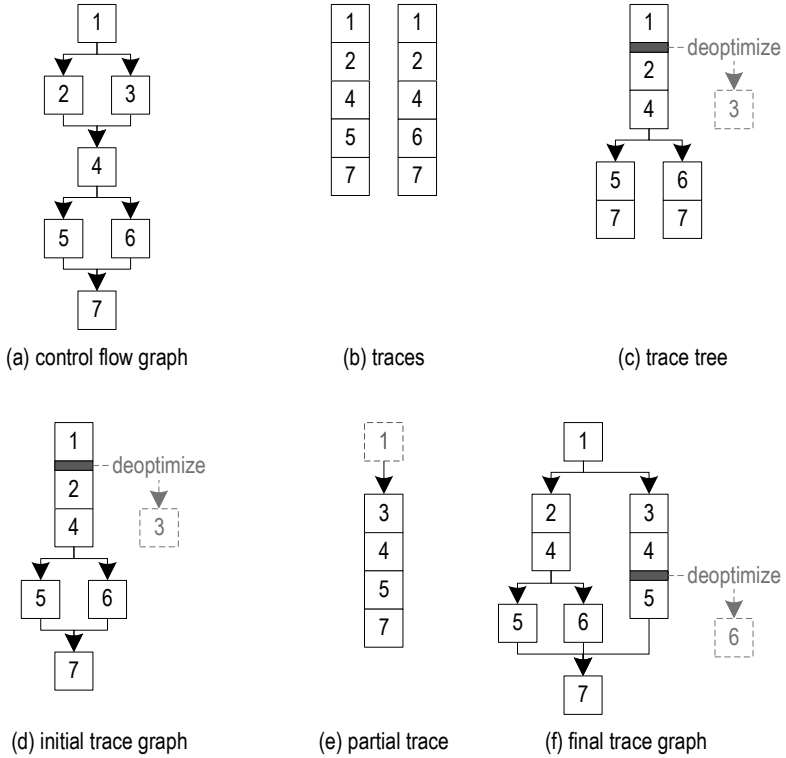


Figure 5.1: Different high-level intermediate representations and partial traces

On the HIR, we apply traditional optimizations such as constant folding, null-check elimination, and dead code elimination. Furthermore, we apply tracing-specific optimizations such as trace inlining, which is our most profitable optimization. Trace inlining is more powerful than method inlining because traces contain context-sensitive information. This information helps to avoid inlining of method parts that were executed frequently in total but are not required for the current caller (see Chapter 6). This is shown in Figure 5.2, where two different callers invoke the same method. Each caller just needs a certain part of the invoked method so that different traces are recorded for the two callers. When com-

piling the caller traces, trace inlining is used to inline only those traces that are needed by the specific caller. This is a significant advantage over the context-insensitive profiling data that is typically used by method-based compilers, as such compilers would inline all executed method parts into both callers.

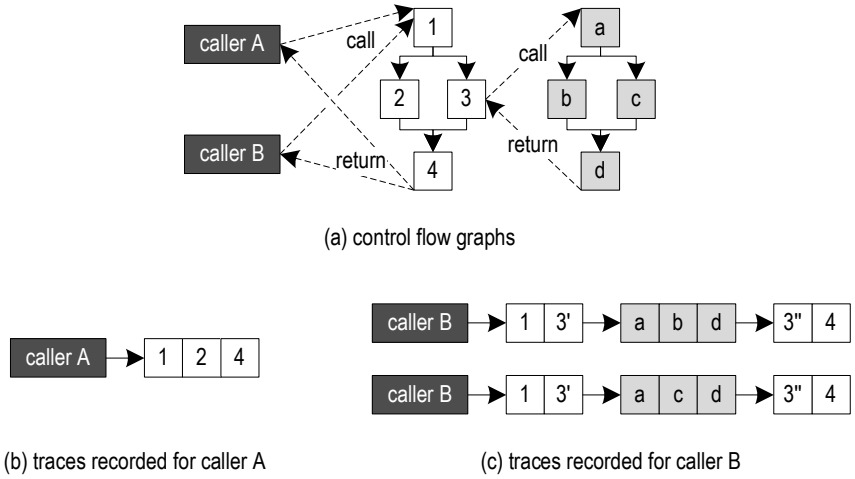


Figure 5.2: Context-sensitive trace information

During trace recording, we record type information for the receiver objects of virtual calls. This information is also context-sensitive so that our trace-based compiler can inline virtual calls more aggressively without causing code bloat. Other optimizations such as optimistic type-check elimination also profit from the context-sensitive type information.

Our trace-based compiler only compiles method parts that are actually covered by traces. Untraced code parts are replaced by *deoptimize* [45] instructions that fall back to the trace recording interpreter if executed. By omitting untraced code parts, we decrease the number of merge points in the compiled code so that our trace-based compiler can optimize more aggressively. This results in better peak performance. Furthermore, it reduces the size of the generated machine code and decreases the amount of time spent on JIT compilation. The recorded traces may also keep track of observed types or values that can then be used for aggressive compiler optimizations. If aggressive optimizations are performed based on the recorded trace information, the compiler emits run-time checks to guard those assumptions. When a run-time check fails, execution deoptimizes to the trace recording interpreter.

When compiled code deoptimizes, the trace recording interpreter records a partial trace that directly starts at the point of deoptimization instead of the trace anchor, see Figure 5.1 (e). The Java HotSpot VM deoptimization mechanism is rather slow but it avoids generating compensation code for infrequently executed cases. If deoptimization is required too frequently for a certain piece of machine code, we invalidate the machine code

and trigger a recompilation that uses the originally recorded traces and all partial traces. The additional information from the partial traces helps to disable optimistic optimizations selectively or it increases the method coverage, as shown in Figure 5.1 (f). This reduces the deoptimization frequency.

5.2 Back End

While we applied massive changes to the front end, only few changes were necessary in the back end. The largest changes in the back end were for the transitioning from compiled traces to other interpreted or compiled code (see Chapter 5.3). For this, we had to introduce new LIR instructions and map them to machine code. Other back end parts such as the register allocator or the code generation only required minor changes because we ensure that every compiled trace has its own stack frame.

Both, the Java HotSpot client and server compilers use intrinsics to optimize certain native JDK methods. Inlining the intrinsic avoids the native call and allows optimizing the code before, during, and after the call as a whole. The server compiler maps significantly more native methods to compiler intrinsincs than the client compiler so that fewer native calls remain in the compiled code. Our trace-based compiler inherits the intrinsics used by the client compiler but we also ported a small subset of simple intrinsics from the server compiler to our trace-based compiler. This allows a better comparability of our results to the Java HotSpot server compiler as fewer native calls remain that hinder optimizations.

5.3 Trace Transitioning

Unlike other trace-based compilers [9, 12, 14, 17, 30–32, 47], we do trace inlining during JIT compilation instead of during trace recording. This allows using sophisticated inlining heuristics to avoid unnecessary code bloat as more information is available during JIT compilation. However, it requires a good trace transitioning system as traces may invoke other interpreted or compiled code. For our trace-based compiler, invoking traces is as important as invoking methods is for a method-based compiler.

Invoking traces that start at method entries is fairly simple as we can reuse the method invocation mechanisms of the Java HotSpot VM. In terms of invocation, there is no difference between methods and method traces. However, for loop traces, a flexible trace transitioning system is necessary that allows arbitrary transitions between interpreted and compiled code.

5.3.1 Separating Loops from Methods

Method-based compilers use *on-stack-replacement* (OSR) [46] for long-running loops to switch from interpreted to compiled code in the middle of a method. Which parts of a method are compiled for on-stack-replacement is up to the compiler. If the whole method is compiled, the profiling data for the method parts before and after the loop are inaccurate as those parts were executed infrequently. This may result in insufficiently optimized machine code. If only the loop is compiled, the optimized code falls back to the interpreter when the loop exits.

When switching from interpreted code to compiled code within a loop, the interpreted stack frame and its values are converted into a compiled stack frame. This conversion can be expensive, but is done only once so that execution can continue in the compiled machine code. If the loop's parent method is compiled at a later point of execution, this compilation includes the long-running loop so that the OSR machine code will not be executed anymore.

Our trace-based approach is a generalization of OSR. It establishes loops as top-level compilation units to allow complete separation of compiled method traces and compiled loop traces. A compiled loop trace has its own stack frame and its own block of machine code that does not necessarily become obsolete when traces of the parent method are compiled. This adds a significant amount of flexibility to our compiler as we can choose to extract loops into their own compilation units when advantageous. However, it still allows reusing large parts of the method-based compiler infrastructure such as the linear scan register allocator or the code generator. Separating loops from methods helps us to reduce the size of each compilation unit and affects the compilation time positively as the compilation time often increases more than linearly with the size of the compilation unit.

Invoking loop traces is difficult since many values may flow into loops. Loops can also modify the values that flow into them so that multiple values may have to be returned to the caller when the loop exits. Otherwise, the caller would continue working with outdated values.

To simplify the invocation of loop traces, we enforce the following restrictions: we do not compile loop traces separately if they use monitors or operand stack values that were defined outside the loop. So, it is never required to pass monitors or operand stack values to loop traces. Furthermore, we do not compile loop traces separately that exit while there are still locked monitors or remaining values on the operand stack. This simplifies returning from loop traces as it is never required to return monitors or operand stack values, while not restricting parameters and local variables that flow into loops.

Both restrictions are small and affect less than 1% of all loops we have seen so far. To enforce those restrictions, we use abstract bytecode interpretation to simulate how the loop interacts with the operand stack and with monitors. Figure 5.3 shows that our simulation starts at the loop header with an empty operand stack and without any monitors. During the simulation, we check whether the loop tries to use a non-existing operand stack value or a non-existing monitor. If this is the case, the loop traces cannot be compiled separately. Furthermore, we also verify for every loop exit that the operand stack is empty (only exception handlers may have one stack value, which must be the exception object) and that there are no remaining locked monitors. When the simulation ends, the loop is marked as compilable or not compilable according to the simulation result. Loops that are not separately compilable have all their traces inlined into their parent trace when the parent trace is compiled.

Figure 5.4 (a) shows the control flow graph of a method that contains a loop, which has the blocks 4 and 7 as its loop exits. A method-based compiler would compile the method as a whole, while our trace-based compiler can compile the loop independently from the remaining method. Figure 5.4 (b) shows a possible trace graph that is created when the recorded method traces are compiled by our trace-based compiler. The loop is invoked by the trace in block 1 and all required values are passed to the loop upon invocation. When the loop exits, it returns all modified values as well as the bytecode index (BCI) of the taken loop exit. Execution returns to the artificial block d that uses the returned BCI to dispatch to the correct successor. In the given example, we assume that only block 7 was covered by traces so that execution deoptimizes to the interpreter if the loop exits to block 4.

Figure 5.4 (c) shows the loop trace graph that is built when the recorded loop traces are compiled. This compilation is independent from the compilation of the method traces so that it can occur earlier or later. During its execution, the loop uses the values that were passed upon loop invocation. When a loop exit is encountered, all modified values as well as the BCI of the taken loop exit are returned to the caller.

5.3.2 Loop Calling Conventions

Restricting the way how loop traces can use monitors and the operand stack greatly simplifies the invocation of loop traces, as it is only necessary to transfer parameters and local variables. Figure 5.5 shows the calling conventions we use for loop traces. When a loop is invoked, we pass an *invocation pointer* to an *invocation area* containing the parameters and the local variables of the caller so that the loop can access those parameters and local variables that it actually requires. Figure 5.5 (a) shows that this is simple when the loop is invoked by interpreted code because the parameters and local variables are

```
 1 boolean canBeCompiled(Loop loop) {
 2   Queue worklist = new Queue();
 3   worklist.push(loop.header());
 4
 5   while (!worklist.is_empty()) {
 6     Block block = worklist.pop();
 7     if (loop.isLoopExit(block)) {
 8       State state = block.state();
 9       if (state.hasMonitors() || state.stackValues() > 1 ||
10           state.stackValues() > 0 &&
11           !block.isExceptionHandler()) {
12         return false;
13       }
14     } else {
15       boolean success = simulateBytecodes(block);
16       if (success) {
17         for (Block sux in block.successors()) {
18           sux.mergeState(block.endState());
19           if (!sux.alreadyProcessed()) worklist.push(sux);
20         }
21       } else {
22         return false;
23       }
24     }
25   }
26   return true;
27 }
28
29 boolean simulateBytecodes(Block block) {
30   State state = block.state();
31   boolean result = true;
32   for (Bytecode c in block.bytecodes()) {
33     result = result && state.popStack(c.requiredStackValues());
34     state.pushStack(c.createdStackValues());
35     result = result && state.popMonitors(c.requiredMonitors());
36     state.pushMonitors(c.createdMonitors());
37   }
38   block.setEndState(state);
39   return result;
40 }
```

Figure 5.3: Bytecode simulation to determine if a loop can be compiled separately

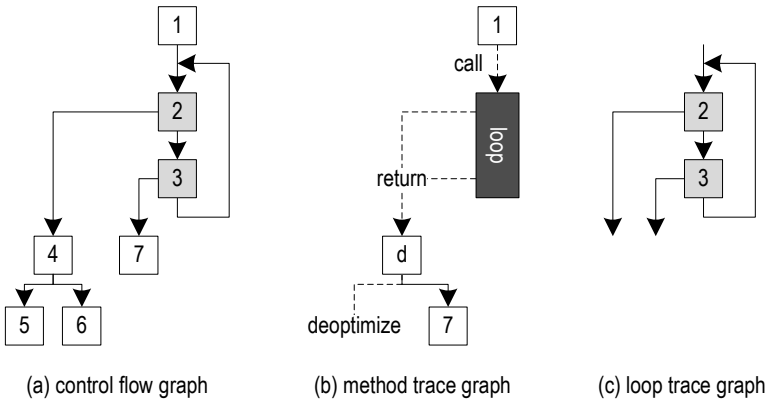


Figure 5.4: Separating loops from methods

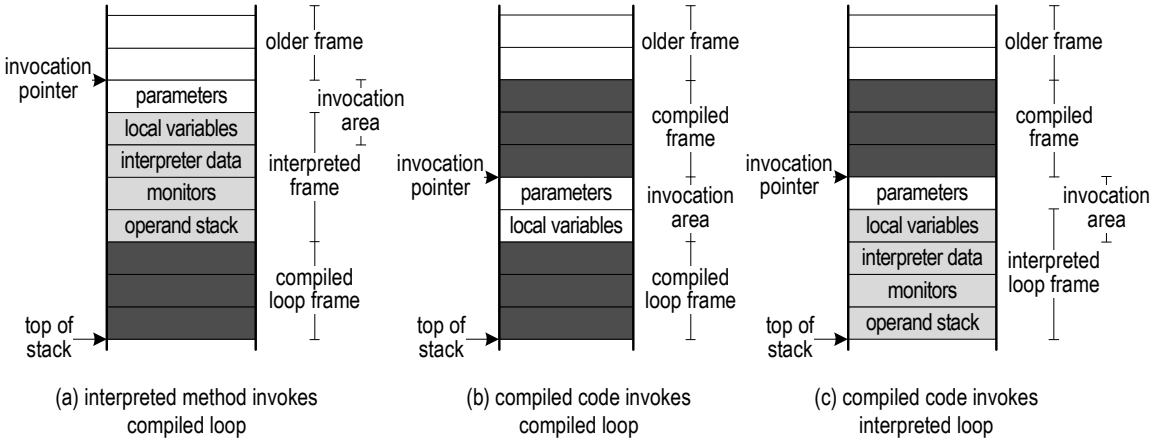


Figure 5.5: Calling conventions for loop traces

explicitly contained in the interpreter frame. When the callee loop starts execution, it reads the required values by addressing them relative to the invocation pointer. During execution, it holds them in registers or spills them to the loop's stack frame. Eventually, execution reaches a loop exit where it writes back all modified values that the caller might access, again using the invocation pointer as the base address. Finally, the loop returns the BCI of the taken loop exit in a specific register. So, the interpreter can continue execution at the returned BCI since the latest values of the parameters and the local variables were written back to the interpreter frame.

If the loop is invoked from compiled code, such as in Figure 5.5 (b), the compiled caller holds its values in registers or spill slots. So, the caller allocates an invocation area at the top of its compiled frame, where it spills its live parameters and local variables before invoking the loop. This brings the parameters and local variables into a defined order and allows the caller to pass an invocation pointer to the callee. The callee loop reads the required values upon loop entry and writes back all modified values when the loop exits. The BCI of the taken loop exit is then returned in a specific register so that the caller can determine where to continue execution. If the invoked loop has only one loop exit, the caller can continue directly without checking the returned BCI. If the invoked loop has more than one exit, the compiled caller uses the returned BCI to dispatch to the correct location in the compiled code. Before execution actually continues, it first reads the latest values from the invocation area.

When a loop is contained in an infrequently executed method part, it is often not useful to compile the infrequently executed loop together with its hot caller trace. Therefore, our calling conventions also support calling an interpreted loop from compiled code, as shown in Figure 5.5 (c). At loop invocation, the compiled caller spills the live parameters and local variables to the invocation area at the top of its frame. Because the parameters and local variables are in the same order as in an interpreter frame, the interpreter just needs to

allocate a partial stack frame containing space for the interpreter-specific data, monitors, and the operand stack. So, the invocation area becomes a part of the interpreter frame. During loop execution, the interpreter can modify the parameters and local variables in place. When the loop exits, the interpreter-specific part of the stack frame is discarded and the loop exit BCI is returned in a specific register. Because the parameters and local variables were modified in place, the invocation area already contains the latest values so that no data has to be written back explicitly. The compiled caller then continues execution as if a compiled loop was invoked.

The case shown in Figure 5.5 (c) also occurs after deoptimization of a compiled loop trace that was invoked by compiled code. After deoptimization, the stack looks exactly as if the compiled code invoked an interpreted loop. If this calling convention variant was not supported, the compiled caller would have to be deoptimized as well, which could result in cascading deoptimizations. By giving the deoptimized loop a separate interpreter frame, we can isolate the effect of deoptimization so that it affects just a single stack frame.

Figure 5.6 shows that our calling conventions enable arbitrary transitions between interpreted and compiled traces. This adds a great amount of flexibility to our system. The transitions can be grouped into the following 4 categories:

1. The first category covers transitions from interpreted to interpreted code. Those are mainly relevant during startup when the application is executed in the interpreter and hardly any code has been compiled yet. In later execution phases, those transitions are still required because the compiled code may deoptimize to the interpreter.

2. The second category covers transitions from compiled code to interpreted code. Those transitions occur, for example when frequently executed traces are compiled and less hot traces are inlined during JIT compilation. The inlined less hot traces may invoke other code that was not jitted yet.

3. The third category covers transitions from interpreted code to compiled code, which occurs during startup when not yet compiled code invokes code that was already jitted. This happens because more frequently executed code, e.g., code with multiple callers, is hotter and therefore compiled earlier. After startup, those transitions still occur when the caller was deoptimized and is therefore interpreted.

4. The fourth category covers transitions from compiled to compiled code. In non-trivial applications, those transitions occur most frequently because all relevant code parts are jitted eventually. So, to achieve a good peak performance most trace transitions must fall into this category.

from frame \ to frame	**interpreted method traces** infrequently executed code or after deoptimization	**interpreted loop traces** infrequently executed code	**compiled method traces** frequently executed code	**compiled loop traces** frequently executed code
interpreted method traces infrequently executed code or after deoptimization	yes	same frame (no explicit transition)	yes	yes
interpreted loop traces after deoptimizing a compiled loop or invoking an infrequently executed loop from compiled code	yes	same frame (no explicit transition)	yes	yes
compiled method traces frequently executed code	yes	yes	yes	yes
compiled loop traces frequently executed code	yes	yes	yes	yes

1 2 3 4

Figure 5.6: Transitions between interpreted and compiled traces

from frame \ to frame	**interpreted method** infrequently executed code or after deoptimization	**interpreted loop** infrequently executed code	**compiled method** frequently executed code	**compiled loop** frequently executed code
interpreted method infrequently executed code or after deoptimization	yes	same frame (no explicit transition)	yes	OSR converts the frame
interpreted loop	interpreted loops never have a separate stack frame			
compiled method frequently executed code	yes	impossible	yes	same frame (no explicit transition)
compiled loop	compiled loops never have a separate stack frame			

Figure 5.7: Transitions between interpreted and compiled methods

In comparison to that, Figure 5.7 shows the transitions possible in a traditional method-based system. There, loops are always compiled with their enclosing method and do not have separate stack frames.

We also considered a concept for register-based loop calling conventions. Such calling conventions could be more efficient and could decrease the loop invocation overhead but would significantly complicate the transition between interpreted and compiled code. Because the loop body is usually executed multiple times, execution tends to stay within a called loop for a longer time than within a called method. Furthermore, for best peak performance, it is more profitable to inline the loop, as mentioned in Chapter 6.3. So, it is unlikely that peak performance would profit significantly from register-based calling conventions for loop traces.

5.4 Exception Handling

Although exceptions should only be thrown in rare cases, some Java applications heavily rely on exception handling. A few applications even use exception handling for control flow decisions. Therefore, also a trace-based JIT compiler must support the most frequent exception handling cases in an efficient way.

Exception handling in method-based compilers always either continues in the exception handler of the executing method or it unwinds the call stack and handles the exception in one of the callers. When a method frame is unwound, all values that were used within the method are discarded as they are no longer needed. For a trace-based compiler, unwinding the stack is more difficult. If an exception is thrown in a loop and caught in the loop's parent, any values that were modified in the loop have to be returned to the parent when the loop frame is discarded. This is necessary so that execution can continue in the exception handler using the latest values.

Previous trace-based compilers for Java [12, 31, 32, 47] either stopped or aborted trace recording when an exception was thrown. Aborting trace recording has the effect that exception throwing traces are never compiled to machine code. Stopping trace recording also results in inefficient exception handling as it precludes that exception sources and exception handlers are compiled together in the same compilation unit. Our trace recording approach supports exceptional control flow so that exception sources and exception handlers frequently are within the same compilation unit, as shown in Figure 5.8 (a). If the exception handler was covered by recorded traces and was therefore compiled to machine code, exception handling is as efficient as possible because the compiled code can dispatch directly to the exception handler. If the exception handler was not compiled, execution deoptimizes to the interpreter. Because a trace-based compiler tends to have larger compilation units than a method-based compiler (traces are smaller than methods and therefore have a higher potential to be inlined), exceptions stay within the same compilation unit most of the time.

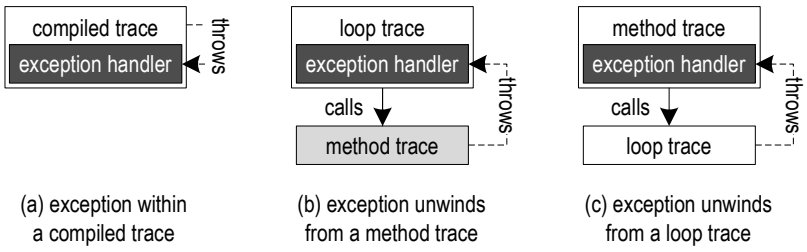


Figure 5.8: Exception handling

Figure 5.8 (b) shows the case where an exception unwinds from a method trace and execution continues within a loop trace. When unwinding the method trace, all live values can be discarded as they are not needed by the caller. In that case, our exception handling is similar to exception handling in a method-based compiler. The same approach is also taken when a loop trace throws an exception that is neither caught within the loop trace nor within its enclosing method.

Figure 5.8 (c) shows the case where an exception unwinds from a loop trace and execution continues in the trace of its enclosing method. The loop trace might have modified some parameters or local variables, so execution must use the latest values when it continues in the exception handler. This case is complicated by the fact that the involved frames can be either interpreted or compiled. When a compiled loop trace throws an exception that is caught in its enclosing method that was compiled separately, we deoptimize the loop trace to the interpreter. This creates the case shown in Figure 5.5 (c) and ensures that the loop's latest values are written back to the invocation area of the caller. Then, the frame of the loop trace is unwound and exception handling continues in the caller. The caller loads the latest values and rethrows the exception using the BCI of the instruction where the exception initially occurred in the loop trace. Eventually, execution continues in the correct exception handler with the latest values.

If this case occurs for an interpreted loop that was invoked by compiled code, we immediately unwind the interpreter-specific part of the loop frame because the interpreter modified the invocation area in place. So, the invocation area already contains the latest values and exception handling can directly continue in the caller.

If an exception unwinds from a loop trace and the exception is caught in the enclosing method, our approach has some overhead as it relies on deoptimization. This overhead is only relevant for frequently thrown exceptions and can be addressed easily using loop inlining (see Chapter 6.3) so that exceptions are thrown within the same compilation unit, which is the most efficient way to handle exceptions.

5.5 Type-specific Optimizations

In addition to our calling conventions and the better exception handling, we added further optimizations to our trace-based compiler. We added a type system that can store different kinds of type information for every SSA value because the original type system of the Java HotSpot client compiler was limited. Our new type system is especially useful for inlining of polymorphic calls, where the receiver is represented by a single SSA value but its type may vary for every inlined method. Furthermore, it is also possible to keep track of the

```
 1: CharSequence concat(CharSequence a, CharSequence b) {
 2:   int totalLength = a.length() + b.length();
 3:   AbstractStringBuilder result;
 4:   if (a instanceof StringBuffer || b instanceof StringBuffer) {
 5:     result = new StringBuffer(totalLength);
 6:   } else {
 7:     result = new StringBuilder(totalLength);
 8:   }
 9:   return result.append(a).append(b);
10: }
```

(a) concatenation source code

```
A concat(string, string);
B concat(stringOrStringBuilder, string);
```

(b) concatenation callers

```
1: CharSequence concat(CharSequence a, CharSequence b) {
2:   guard(a.getClass() == String.class);
3:   int totalLength = inline(a.length());      // String.length()
4:   guard(b.getClass() == String.class);
5:   totalLength += inline(b.length());         // String.length()
6:   StringBuilder result = new StringBuilder(totalLength);
7:   return inline(result.append(a).append(b)); // StringBuilder.append()
8: }
```

(c) optimized concatenation for caller A

```
 1: CharSequence concat(CharSequence a, CharSequence b) {
 2:   int totalLength;
 3:   if (a.getClass() == String.class) {
 4:     totalLength = inline(a.length());        // String.length()
 5:   } else if (a.getClass() == StringBuilder.class) {
 6:     totalLength = inline(a.length());        // StringBuilder.length()
 7:   } else {
 8:     deoptimizeToInterpreter();
 9:   }
10:   guard(b.getClass() == String.class);
11:   totalLength += inline(b.length());         // String.length()
12:
13:   StringBuilder result = new StringBuilder(totalLength);
14:   return inline(result.append(a).append(b)); // StringBuilder.append()
15: }
```

(d) optimized concatenation for caller B

Figure 5.9: Type-specific optimizations

fact that the type of a value is a subtype of a specific class or that a value has multiple possible types.

Figure 5.9 (a) shows the Java source code of the method `concat()` which concatenates two character sequences. Two call sites that invoke `concat()` with different parameters are shown in Figure 5.9 (b). When the callers are getting compiled, each caller inlines those `concat()` traces that were recorded for this caller. This context-sensitivity allows

specializing the inlined code for the specific caller and reduces the number of types observed for each value so that aggressive type-specific optimizations can be used more frequently.

The first caller only passes `String` objects to `concat()`, which is also reflected by the type information recorded in the traces. When inlining those traces, the JIT compiler generates the optimized pseudo-code shown in Figure 5.9 (c) and uses run-time checks to guard the assumption that only `String` objects are passed as *a* and *b*. When such a run-time check fails, the execution deoptimizes to the trace recording interpreter.

For the remaining compilation unit, the compiler knows that *a* and *b* are always `String` objects so that it is possible to change the `CharSequence.length()` interface calls to `String.length()` calls. Unlike the interface calls, those calls can be inlined easily. Furthermore, it is now also possible to eliminate the *instanceof* checks as neither *a* nor *b* are `StringBuffer` objects. So, it is known that *result* is a `StringBuilder` object so that the calls to `append()` can also be inlined easily.

Figure 5.9 (d) shows the optimized pseudo-code for the second caller that either passed a `String` or a `StringBuilder` object as the first parameter and always used a `String` object as the second parameter. The first call to `CharSequence.length()` is polymorphic as the type information in the traces indicates that *a* is either a `String` or a `StringBuilder`. So, two type checks are added that dispatch to the inlined versions of `String.length()` and `StringBuilder.length()`. If *a* is neither a `String` nor a `StringBuilder`, the execution deoptimizes to the trace recording interpreter. For the invocation of `CharSequence.length()` on *b*, a guard is added to ensure that *b* is a `String` object. In the next step, it is possible to eliminate the instanceof checks as neither *a* nor *b* is a `StringBuffer`. So, the *result* is a `StringBuilder` object and the calls to `append()` can be inlined.

By using the type information for optimizations such as removal of type checks and elimination of redundant guards, we increase the performance and help the compiler to reduce the bookkeeping information that is required for deoptimization.

5.6 Tail Duplication

One optimization typical for trace-based compilers is tail duplication which duplicates control flow along frequently executed paths [9]. If performed excessively, this results in a trace tree [32] without any inner merge nodes and with exponential growth of compiled code because of the control flow duplication. While this increases the size of the generated machine code, it avoids ϕ instructions that are otherwise introduced due to control flow merges. Having fewer ϕ instructions increases the optimization opportunities and therefore the performance.

Our implementation avoids excessive tail duplication which is otherwise common to trace compilation (see Chapter 9) by merging the control flow of multiple traces into a trace graph. This hybrid between a traditional control flow graph and a trace tree supports both control flow duplication and inner merge nodes. Merging the traces is explicitly done by the JIT compiler, because by the time of JIT compilation all relevant traces have been recorded.

Combining tail duplication and aggressive trace inlining can easily result in code bloat because tail duplication may duplicate calls that are then inlined multiple times. Tail duplication can also prevent inlining if the probabilities of the recorded traces are too equally distributed. Due to the duplication, the trace graph may contain multiple rarely executed calls instead of one call that was executed frequently. This is difficult for inlining heuristics that use the execution probability to determine if a call is worth inlining.

When building the trace graph from the recorded traces, we control the amount of used tail duplication with a tail duplication heuristic. We evaluated multiple simple heuristics and observed that tail duplication hardly increases the average peak performance of Java applications. However, it significantly increases the amount of generated machine code and the time required for JIT compilation. When the same time and code size budget is used to apply trace inlining instead of tail duplication, the increase in peak performance is significant. For Java, tail duplication therefore mainly seems interesting when no further peak performance increase can be achieved using trace inlining.

5.7 Runtime Changes

We modified the original deoptimization mechanism to support deoptimization of compiled traces instead of compiled methods. Figure 5.10 (a) shows the initial stack where a compiled method trace invoked a compiled loop trace. First, we rescue all values that are live in the current compiled frame to the heap.

Then, we remove the compiled loop frame from the stack as shown in Figure 5.10 (b). In the next step, the appropriate number of interpreter stack frames is pushed. To fill the interpreter frames with the rescued values, we access the debugging information that the trace-based JIT compiler generated for the deoptimizing instruction. This information tells us which rescued values should be placed at which locations in the interpreter frames. Due to trace inlining, one compiled trace may correspond to multiple interpreted method and loop traces. In this example, one method trace was inlined into the loop trace so that the resulting stack looks as shown in Figure 5.10 (c). Furthermore, one partial trace is pushed on the tracing stack for every inlined loop or method trace. The resulting tracing stack is shown in Figure 5.10 (c), assuming that the tracing stack was empty before

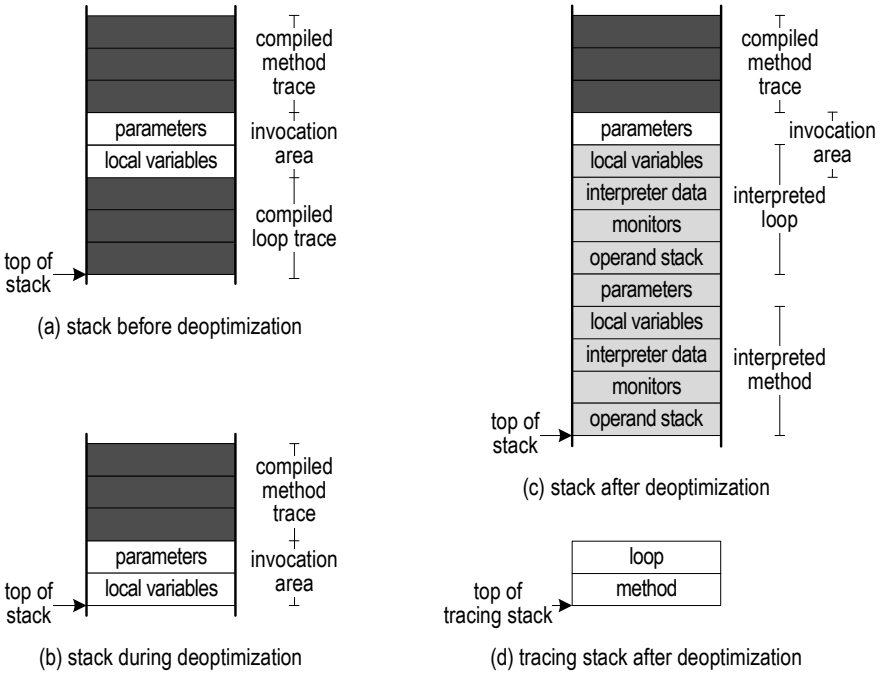


Figure 5.10: Deoptimizing a loop trace

deoptimization. When execution resumes in the trace recording interpreter, it continues trace recording using the partial traces from the tracing stack.

Chapter 6

Trace Inlining

Similar to method inlining, trace inlining replaces calls with copies of the actually called code. In terms of performance, this is one of the most profitable optimizations. It removes the call overhead and increases the compilation scope so that more code can be optimized as a whole. This increases the effectiveness of many optimizations and may simplify both the caller and the inlined callee. Our trace-based JIT compiler supports both static and dynamic inlining heuristics by making use of the recorded trace information.

We start trace inlining by computing the maximum trace size that is acceptable to be inlined at the current call site. This mainly depends on the call site's relevance (see Chapter 6.4) for program execution. Then, we use a heuristic to decide if it is worth to inline the invoked traces at the current call site. To a large degree, this depends on the size of the traces because inlining large traces causes code bloat. When doing trace inlining, we distinguish between the inlining of method traces and loop traces.

6.1 Advantages Over Method Inlining

Trace inlining has several advantages over method inlining:

- The recorded traces contain context-sensitive information (see Chapter 6.5) about which method parts are used by which caller. This information is preserved over method boundaries and can be used to avoid inlining of method parts that were executed frequently in total but are not required for the current caller.
- Trace inlining does only inline frequently executed traces instead of whole methods. Method-based compilers try to use profiling data to avoid compilation of infrequently executed method parts [28, 66, 68]. This is less effective than trace inlining because the method-based profiling data typically is not context-sensitive.

- Traces also store information about the receivers of virtual calls and due to our trace linking, this information is also context-sensitive. So, it might turn out that a certain call site invokes only methods of a specific receiver type. This information can be used for aggressive inlining of virtual methods. Method-based compilers also use profiling data for aggressive inlining of virtual calls, but in most compilers this information is not context-sensitive so that the profiling data is possibly polluted.

6.2 Method Traces

Inlining method traces is similar to method inlining except that the traces usually do not cover all bytecodes of the callee. So, we build a trace graph from the traces that should be inlined and replace the method invocation with the contents of that trace graph. Then, `return` instructions that are located within the inlined bytecodes are replaced with direct jumps to the instruction after the call and exception-throwing instructions are wired to exception handlers located in the caller's trace graph.

Figure 6.1 (a) shows the control flow graphs of two methods. Two traces through those methods are shown in Figure 6.1 (b). After performing trace recording frequently enough, the recorded traces are getting compiled. The resulting trace graph after trace inlining (but without explicit control flow duplication) is shown in Figure 6.1 (c). This trace graph is then compiled to optimized machine code. If one of the removed blocks must be executed later on, the compiled code deoptimizes to the interpreter.

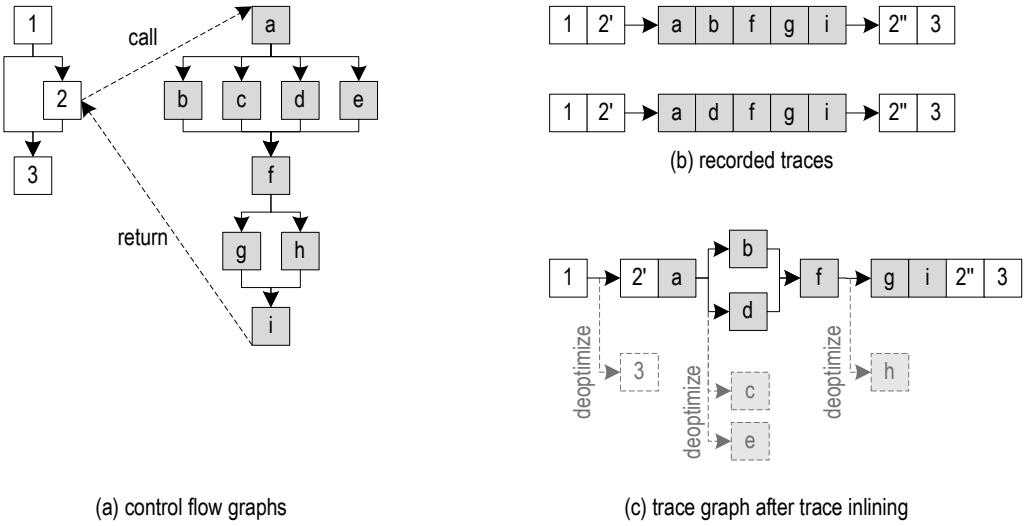


Figure 6.1: Inlining method traces

Another interesting aspect in this example is that we also remove the edge from block 1 to block 3 although the trace graph does contain block 3. This is advantageous because it

avoids control flow merges, which otherwise could constrain compiler optimizations. So, removing edges that are not executed results in better optimized machine code.

In most cases, we inline only those traces that were executed upon invocation by the current caller. However, if the callee traces were compiled before trace recording was started for the caller, the caller does not know which of the compiled traces it needs. In those cases, we conservatively consider all callee traces as inlining candidates, except those for which we can proof that they cannot be invoked by the current caller because of the specific parameters that the caller passes to the callee. The used technique behind that is similar to dead code elimination in a method-based compiler but allows eliminating whole traces instead of basic blocks. To further reduce the number of inlined traces, we also filter out infrequently executed traces (see Chapter 6.7).

For virtual method invocations, we combine the recorded trace information with the Java HotSpot client compiler's CHA to determine the possible receiver types for the current call site. If the CHA identifies a single possible target method, the invoked method traces are inlined in a similar way as the Java HotSpot client compiler inlines methods. If the CHA finds multiple possible target methods, we try to use the recorded receiver classes for inlining the method traces aggressively. For this, we add a run-time check that compares the actual receiver type with the expected type and deoptimizes to the interpreter if the types do not match. By combining CHA and context-sensitive trace information, we can inline virtual calls with less code bloat than most method-based compilers while emitting run-time checks only where necessary.

For inlining virtual calls and interface calls optimistically, based on the recorded type information, we use the following run-time checks:

- If the recorded traces indicate that the invoked method has always belonged to the same type of receiver, we do the inlining and guard it with a *type guard* that compares the actual receiver type to the expected one and deoptimizes to the interpreter if the types do not match.

- If a call site always invokes the same method but does it on different receiver types, we enable this kind of inlining by guarding it with a so called *method guard* that accesses the virtual method table of the actual receiver and compares the invoked method with the expected method. If the methods do not match, we deoptimize to the interpreter. This kind of inlining is especially advantageous if an abstract base class implements a method that is not overridden by subclasses. However, it only works for virtual calls and not for interface calls because interface calls do not have an index into the virtual method table.

- If a call site always invokes the same method but does it via a receiver of an interface type, we extend type guards to a switch-like structure so that they can check for multiple receiver types. This is cheaper than the interface lookup and allows us to inline invocations of interface methods in many cases. If the actual receiver type does not match any of the expected types we deoptimize to the interpreter.

- Another optimization is the inlining of polymorphic calls. Figure 6.2 (a) shows a method, where a virtual call might invoke two different methods. Because these scenario methods are small, it pays off to inline them both. This results in the control flow shown in Figure 6.2 (b) where block 2' dispatches to one of the inlined methods depending on the type of the actual receiver. Here, we also use switch-like semantics so that several types can dispatch to the same inlined method. If the actual receiver type does not match any of the expected types, we deoptimize to the interpreter.

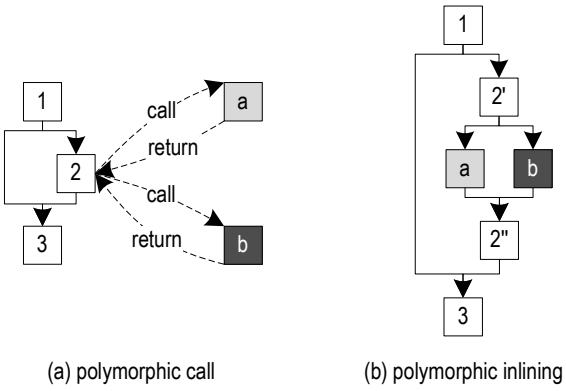


Figure 6.2: Polymorphic inlining

The Java HotSpot server compiler also inlines polymorphic calls but limits the number of inlined methods to at most two, as a higher number could easily result in code bloat. Our trace-based compiler inlines method parts more selectively due to the context-sensitivity of the recorded traces. So, we can avoid inlining of method parts that were executed frequently in total, but are not required for the current caller. Furthermore, the recorded type information is also context-sensitive, which reduces the number of inlining candidates. So, our trace-based compiler does not have to limit the number of inlined methods but instead only limits the total size of all inlined methods depending on the execution frequency of the specific call site. For applications with a high number of polymorphic calls, this results in significantly better inlining and therefore a higher performance, while avoiding issues with code bloat.

6.3 Loop Traces

Method-based JIT compilers compile whole methods (including loops) to optimized machine code. During compilation, the compiler collects information about the values that are used within the method and the loops. This information may consist of types, constant values, or the observation that an object does not escape the method. Common optimizations such as inlining of polymorphic calls or constant folding make use of this information. If a loop and its caller are compiled together this information can be used for optimizations in the loop. However, if the loop is compiled separately before its caller, the following problems occur:

- The *invoked loop* does not have much information about the incoming values. As a large number of values may flow into a loop, this lack of information can preclude optimizations in the invoked loop.
- The *loop caller* has the problem that a loop invocation kills all information about the values that could be modified by the loop. This may preclude optimizations in the caller and is a significant problem for nested loops, where the inner and the outer loops operate on the same values.

The severity of those issues can be reduced by recording profiling data about the passed values. To mitigate all disadvantages, trace-based compilation must perform loop inlining. Figure 6.3 (a) shows a trace graph that was built for method traces that invoked loop traces. The loop traces were not inlined yet, so the loop is represented as a black box that is still unknown to the compiler. In the next step, a separate trace graph is built from the loop traces as shown in Figure 6.3 (b). The actual inlining then replaces the black box in the caller trace graph with the loop trace graph and links all loop exits to their correct successor blocks using jump instructions. In this example, block b is linked to block e and block c is linked to block d, resulting in the trace graph shown in Figure 6.3 (c). This allows the compiler to optimize the loop and its caller as a whole, which results in a similar effect as method inlining. We use the recorded probability of entering the loop to decide whether we should compile the loop separately or if it should be inlined.

When inlining loop traces, we consider all traces that were recorded for a specific loop as inlining candidates. This is necessary because loop traces are never explicitly linked to their caller trace, so no context-specific call information is available. However, we use the information about the parameters and locals that flow into the loop to eliminate those traces for which we can proof that they cannot be invoked by the current caller. Furthermore, we also eliminate traces that were not executed frequently enough.

A more difficult case is that the inlined loop can have a loop exit for which no successor exists in the caller's trace graph. For example, in Figure 6.3 (a), block d could be missing

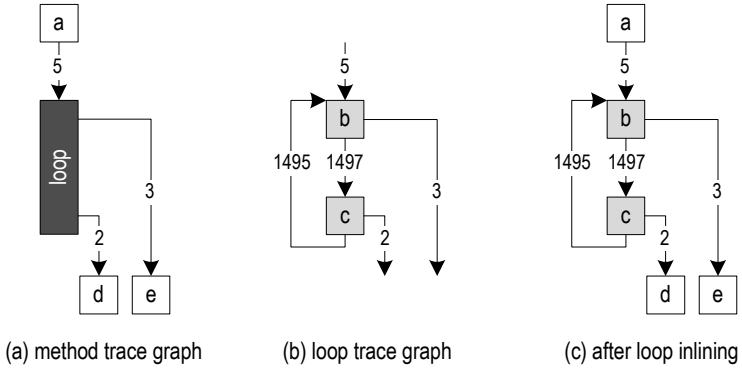


Figure 6.3: Inlining loop traces

because it was never recorded. However, both loop exits could still be present in the recorded loop traces as shown in Figure 6.3 (b). One way how this can happen is when the loop traces are compiled before trace recording is started for the method trace. In a first approach, we addressed this issue by explicitly adding deoptimization points for all loop exits that could not be linked to a successor, so that execution deoptimized to the interpreter when such a loop exit was taken. However, it turned out to be better if we simply eliminate loop traces that end in a loop exit that was not recorded for the current caller. This reduces the number of inlining candidates, results in less generated machine code, and also deoptimizes in case that the specific path has to be executed.

6.4 Relevance

The relevance of a call site is determined by the relevance of the trace graph block in which the call site is located. We evaluated three different algorithms for computing the relevance and illustrate their behavior on the two trace graph examples A and B shown in Figure 6.4. Example A was built from four different traces that hardly share any blocks. Example B also shows a trace graph built from four traces, but every block is shared with at least one other trace. For computing the relevance of the trace graph blocks, we first determine how often each block was executed by recorded traces. Figure 6.4 (a) shows the trace graphs where every block is annotated with its execution frequency. Then, we compute the relevance of each block by dividing its execution frequency with a reference value. Depending on the reference value, the relevance is scaled differently. So, we use one of the following algorithms to choose that reference value:

- *Simple:* The simplest way of computing the relevance of a trace graph block is to divide its execution frequency by the total execution frequency of all traces merged into the trace graph. The resulting value is in the range]0, 1] and assigns a high

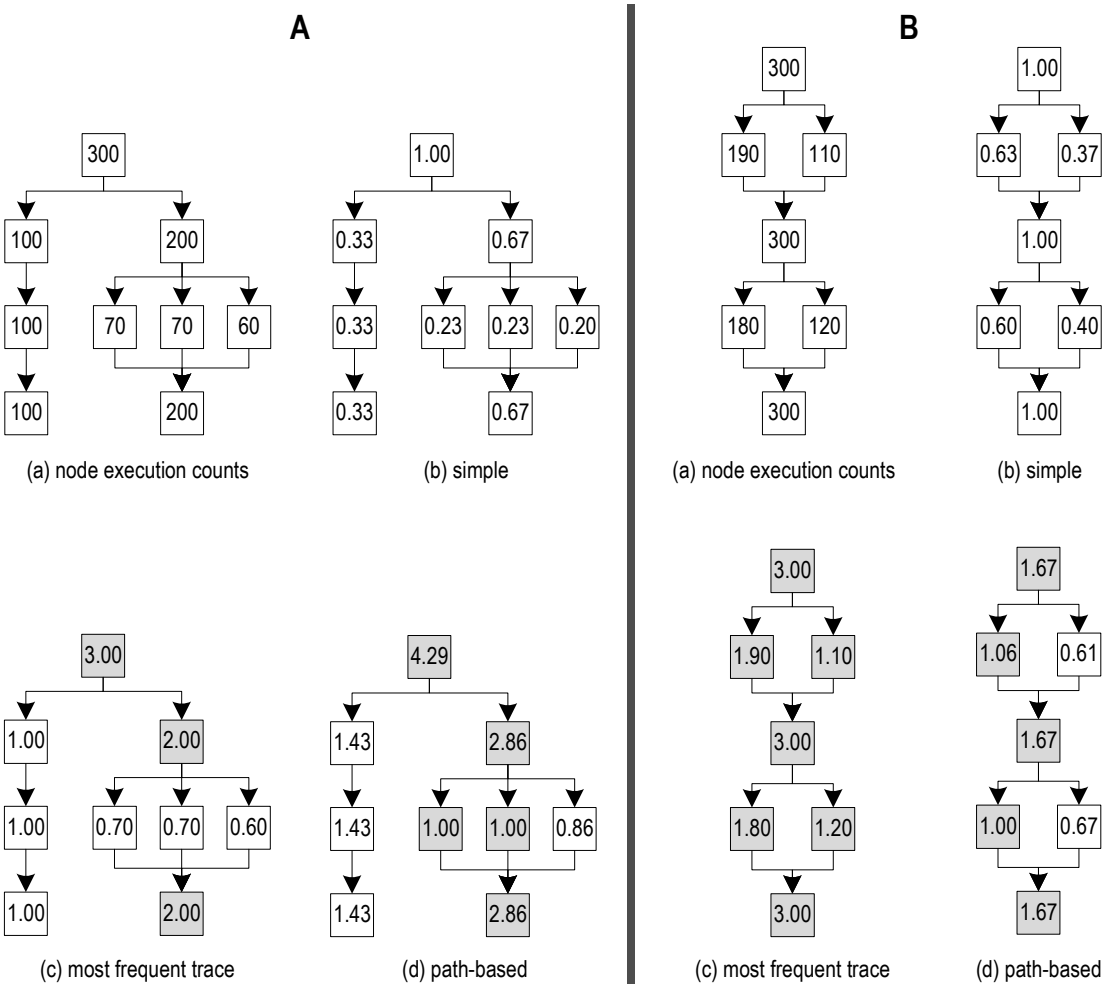


Figure 6.4: Different relevance computation algorithms

relevance to those blocks in which inlining has a positive effect during most executions, as shown in Figure 6.4 (b). This algorithm is well suited for minimizing the size of the generated machine code, while obtaining a decent peak performance.

- *Most frequent trace:* Another way is to divide the block execution frequency by the execution frequency of the most frequently executed trace ever merged into the trace graph. Because traces are merged, trace graph blocks that are shared between multiple traces have a higher execution count than they would have without merging. So, this metric returns a high relevance for call sites that are within such shared blocks, while returning a value in the range]0, 1] for call sites that are only contained in individual traces. In Figure 6.4 (c), the colored blocks are shared and therefore get a higher relevance. If many different traces were recorded and many blocks are shared in the trace graph, then it can happen that every block in the trace graph has a relevance greater than 1, as shown in example B of Figure 6.4 (c).

This algorithm can be used for aggressive trace inlining that results in best-possible peak performance while generating large amounts of generated machine code.

- *Path-based:* Our third approach computes a variant of the most frequently executed path through the trace graph. We start at the root block of the trace graph, mark it as visited and determine the most frequently executed successor block. Then, we mark this successor block as visited and continue with it recursively until we either reach a block without successors or until we are back at the loop header. All blocks that are visited during this algorithm are colored in Figure 6.4 (d). Then, we use the lowest execution frequency of all visited blocks to compute the relevance of all other blocks in the trace graph. This has the advantage that important call sites, i.e., those on this path and on frequently executed split/merge points, have a value in the range $[1, \infty[$, while less important calls have a value in the range $]0, 1[$. This algorithm also results in excellent peak performance but it often results in less generated machine code than *most frequent trace*.

In detail, we evaluated those relevance computation algorithms with multiple trace inlining heuristics in [38] and [39].

6.5 Context Sensitivity

Our trace recording infrastructure restricts traces to span at most one method so that the trace-based compiler has to rely on aggressive trace inlining [38]. The trace recording mechanism preserves context-sensitive information over method boundaries so that each caller knows which parts of the callee it should inline. This helps the compiler to avoid inlining of method parts that were executed frequently in total, but are irrelevant for the current caller. It reduces the generated amount of machine code, and decreases the number of merge points, which increases peak performance.

Also method-based compilers use profiling data to remove never executed code. However, their profiling data typically lacks the context-sensitivity so that they cannot decide which method parts are required for each specific caller. Context-sensitive profiling data could in principle also be recorded for a method-based compiler but trace recording and trace-based compilation clearly simplifies it.

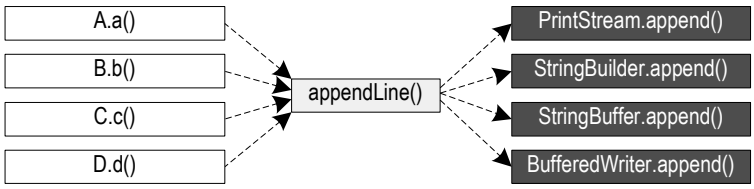
Figure 6.5 shows the method `indexOf()` of the JDK class `ArrayList`. The first part of the method handles the rare case of searching null, while the second part searches the list for non-null objects. Most callers will only require the second part of the method. However, if there is at least one caller in the application that executes the first part of the method, the profiling data in a method-based compiler would indicate that the first part has been executed. So, whenever the method-based compiler inlines the method

```
 1 public int indexOf(Object o) {
 2   if (o == null) {
 3     for (int i = 0; i < size; i++) {
 4       if (elementData[i] == null) {
 5         return i;
 6       }
 7     }
 8   } else {
 9     for (int i = 0; i < size; i++) {
10       if (o.equals(elementData[i])) {
11         return i;
12       }
13     }
14   }
15   return -1;
16 }
```

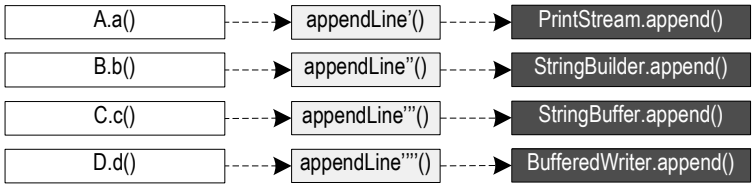
Figure 6.5: Method `ArrayList.indexOf()`

```
 1 public class LineBuilder {
 2   private final Appendable buffer;
 3
 4   public LineBuilder(Appendable buffer) {
 5     this.buffer = buffer;
 6   }
 7
 8   public void appendLine(CharSequence sequence) {
 9     buffer.append(sequence);
10     buffer.append("\n");
11   }
12 }
```

(a) code pattern



(b) possible method invocations



(c) preferred inlining

Figure 6.6: Context-sensitive type information

`indexOf()`, it does also inline this rarely executed method part. Due to our context-sensitive trace information, our trace-based compiler can avoid inlining method parts that were never executed for the current caller.

Because trace inlining is more selective in what it does inline, our trace-based compiler can use a more aggressive inlining policy, i.e., it can inline individual traces through a method where the whole method would be too large to be inlined. This allows more and deeper inlining for a given maximum inlining size and thus increases the compilation scope significantly. Especially, for complex applications, this results in better optimized machine code and has a significant positive effect on peak performance.

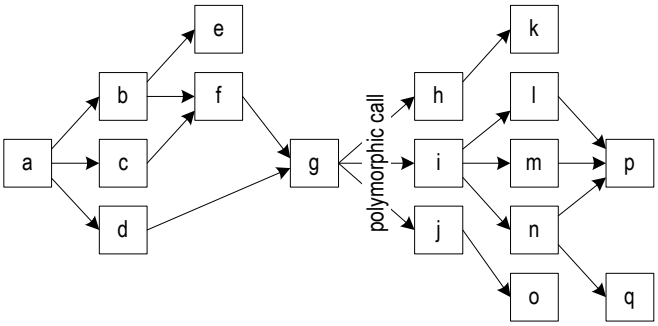
Figure 6.6 (a) shows the class `LineBuilder` that wraps an `Appendable` object and provides the method `appendLine()`. If multiple `LineBuilder` objects are used to wrap instances of different classes, such as `PrintStream`, `StringBuilder`, `StringBuffer`, and `BufferedWriter`, then the invocations of `append()` on lines 9 and 10 will be polymorphic calls that cannot be inlined easily, as shown in Figure 6.6 (b).

However, if the dispatch in `appendLine()` depends on its call site, e.g., because different `LineBuilder` objects are used at different call sites, the inlining in Figure 6.6 (c) would be preferable. Our context-sensitive trace information also stores the receiver types of virtual calls. So, our trace-based compiler can do the preferable inlining indicated in Figure 6.6 (c) by using this context-sensitive information for aggressive inlining of virtual calls. If a compiler does not record the profiling data in a context-sensitive way, but just accumulates all encountered types (i.e., `PrintStream`, `StringBuilder`, `StringBuffer`, and `BufferedWriter` at `buffer.append()`) it will not have enough information to inline such virtual calls.

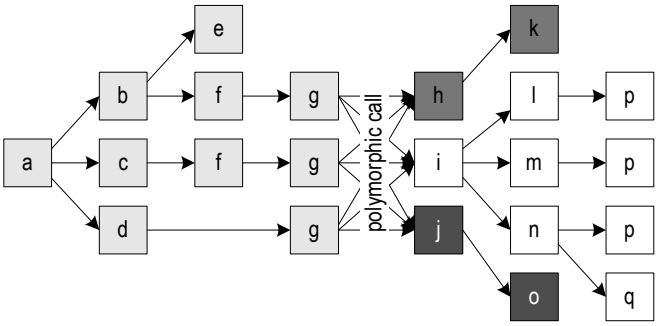
6.6 Compilation Units

Figure 6.7 (a) shows the call graph for a certain set of methods. Every box represents a method, and arrows are method calls. Within the call graph, method g contains a polymorphic call where three target methods are possible.

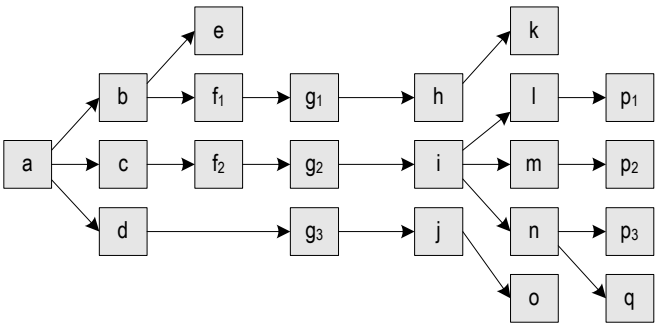
Figure 6.7 (b) shows the compilation units for a method-based compiler that uses aggressive method inlining. When method a is getting compiled, the methods b, c, and d are inlined, leading to new calls that are inlined again. When the inlining reaches the polymorphic call the inlining heuristic can either choose to inline none, some, or all target methods. Inlining all target methods can cause significant code bloat so that, for example, the Java HotSpot server compiler only inlines polymorphic calls with a maximum of two target methods. So, the inlining stops at the polymorphic call and each target method is



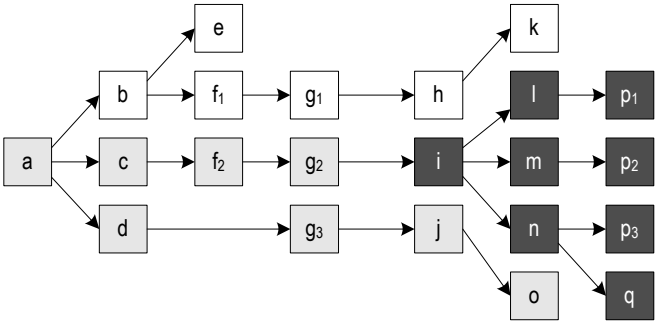
(a) call graph



(b) compilation units in a method-based compiler



(c) compilation units in a trace-based compiler with aggressive inlining



(d) compilation units in a trace-based compiler with less aggressive inlining

Figure 6.7: Compilation units

compiled as a separate compilation unit. This example results in 4 compilation units that have *a*, *h*, *i*, and *j* as their root methods.

Figure 6.7 (c) shows the compilation units of our trace-based compiler that uses aggressive trace inlining. When the method traces of *a* are getting compiled, the context-sensitive trace information is used for inlining so that when the traces of *f* are getting inlined into *b* and *c*, each caller only inlines those traces that it actually requires. The context-sensitive traces and the type information for the receiver objects of virtual calls help the compiler to avoid generating polymorphic calls. Assume that in this example, the polymorphic call depends on the caller so that only one target method has been observed for every variant of *g*. So, with a sufficiently aggressive inlining policy, the JIT compiler can inline all relevant code parts into one compilation unit.

Figure 6.7 (d) shows possible compilation units for a trace-based compiler that uses less aggressive trace inlining. Here, the inlining stops earlier which results in smaller compilation units so that the method entries of *a*, *b*, and *i* become the roots of the resulting 3 compilation units. However, the context-sensitive trace information still helps the compiler to avoid the polymorphic call in this example.

Depending on the size of the resulting compilation units, the JIT compiler can perform different optimizations. Large compilation units offer more opportunities for optimizations but they also require more time for compilation and more machine code is generated because inlining duplicates code parts. The time required for JIT compilation usually increases more than linear with the size of the compilation unit as some optimizations have a non-linear run time. For startup performance, small compilation units are usually better so that the JIT compilation quickly results in machine code that can be executed instead of interpreting the bytecodes. However, for best peak performance, the compilation units should be sufficiently large to allow good optimizations.

6.7 Trace Filtering

When a trace is recorded, chances are good that the trace is important and will be executed frequently. However, sometimes recorded traces turn out to be rarely executed. By eliminating such traces, we can ensure that only important paths are compiled. Figure 6.8 (a) shows the trace graph after merging all recorded traces. The graph edges are annotated with the execution frequencies.

For every block, we determine the most frequently executed outgoing edge and compare its frequency to those of all other outgoing edges of the same block. Then, we remove all edges with a 100x lower execution frequency. After processing all blocks, we remove no

longer reachable blocks from the trace graph. Figure 6.8 (b) shows the resulting graph after filtering.

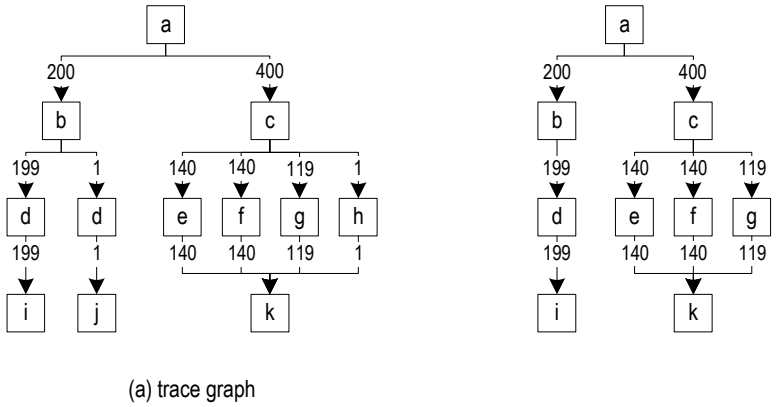


Figure 6.8: Filtering out infrequently executed traces

The recorded trace information conserves the program behavior that was observed during a specific time frame. At a later point of execution, infrequently executed (and therefore eliminated) paths might become important as the program behavior may change over time. This results in frequent deoptimization because not compiled paths get to be executed. If too frequent deoptimization is detected, the compiled machine code is invalidated and another compilation is triggered. This compilation avoids trace filtering for those cases that resulted in frequent deoptimization.

Trace filtering has the following corner cases, where extra care must be taken:

- For most loops, the loop body is executed significantly more frequently than the loop exits, see Figure 6.3 (c). So, the execution frequencies of the loop exits have to be compared to the frequency of the loop entry instead of to the frequency of the backwards branch. Otherwise, the loop exit edges would be filtered out, so that deoptimization to the interpreter is required after executing a loop. This would increase the deoptimization frequency and it would limit the possible compilation scope when doing loop inlining.

- Aggressive trace inlining may also inline infrequently executed traces. Those inlined traces may not necessarily reflect the typical execution behavior yet so that trace filtering might eliminate important traces. This would result in frequent deoptimization so that trace filtering should be avoided for insufficiently trace-recorded methods and loops.

6.8 Effect on Compiler Intrinsics

Java code can call native methods using the Java Native Interface (JNI). This mechanism is mainly used to implement platform-specific features that could not be expressed in Java otherwise. Some methods of the Java standard library, e.g., `System.arraycopy()`, are implemented in a platform-specific way directly in the JVM. As no Java code is executed for such methods, trace recording is not possible for those methods.

The Java HotSpot VM uses compiler intrinsics for the most important platform-specific methods so that the JIT compiler can inline such methods. Those intrinsics are modeled as HIR and LIR instructions so that the compiler can apply optimizations. If our trace-based JIT compiler compiles a trace graph that contains the invocation of a native method that is implemented as a compiler intrinsic, we do exactly the same inlining as the method-based compiler.

Still, our trace-based compiler has one advantage: traces are smaller than methods so that our trace-based compiler can inline Java traces more aggressively than a method-

```
public static void primitiveArraycopy(Object src, int srcPos, Object dest,
           int destPos, int length) {
  if (src == null || dest == null) {
    throw new NullPointerException();
  }
  if (srcPos < 0 || destPos < 0 || length < 0 ||
      srcPos + length > src.length || destPos + length > dest.length) {
    throw new IndexOutOfBoundsException();
  }
  if (!src.isArray() || !dest.isArray() || src.getClass() != dest.getClass()) {
    throw new ArrayStoreException();
  }

  if (src == dest && isOverlapping(srcPos, destPos, length)) {
    copyOverlapping(src, srcPos, dest, destPos, length);
  } else {
    copyNonOverlapping(src, srcPos, dest, destPos, length);
  }
}
```

(a) unoptimized arraycopy for primitive type arrays

```
public static void primitiveArraycopy(Object src, int srcPos, Object dest,
           int destPos, int length) {
  // src != null && dest != null && src.isArray() && dest.isArray() &&
  // length >= 0 && src != dest && src.getClass() == dest.getClass()
  // srcPos >= 0 && srcPos + length <= src.length &&
  // destPos >= 0 && destPos + length <= dest.length
  copyNonOverlapping(src, srcPos, dest, destPos, length);
}
```

(b) optimized arraycopy for primitive type arrays

Figure 6.9: Pseudo-code for `System.arraycopy()` when copying primitive type arrays

based compiler could inline Java methods. This results in a larger compilation scope so that it is more likely that the caller of a native method has specific knowledge about the parameters that are passed to the native method. The JIT compiler can use this information to optimize inlined compiler intrinsics more aggressively.

Figure 6.9 (a) shows pseudo-code for the implementation of `System.arraycopy()`, which is used to copy primitive type arrays. Depending on the compiler's information about the parameters that are passed to `System.arraycopy()`, it can optimize the intrinsic. Figure 6.9 (b) shows an optimized version of the method where the compiler could optimally exploit the parameter values. The necessary parameter information is for example available when the source and the destination arrays are allocated in the same compilation scope in which `System.arraycopy()` is inlined. So, increasing the compilation scope can help the compiler to increase the performance of inlined compiler intrinsics.

Chapter 7

Deriving Code Coverage Information from Recorded Traces

Trace recording and the recorded trace information is what fuels a trace-based JIT compiler so that it can perform optimistic optimizations that increase performance. However, the recorded traces can also be used for other purposes outside the JIT compiler. We propose an efficient technique for deriving exact code coverage information from the recorded traces. Code coverage metrics are used to determine the statements, branches or paths that were covered during the execution of a program. These metrics are useful for finding out whether the program was tested extensively enough and are usually obtained by instrumenting the application.

Figure 7.1 (a) shows the steps that are typically involved to record code coverage information for a Java application. The first step is to instrument the application either in an offline preprocessing step or online while the application is executed. The code coverage tool inserts *probes* at all relevant positions of the control flow. When a probe is executed, it marks a certain part of the application as covered. When the application exits, the coverage data is persisted, either by storing it in a file or transmitting it to another machine. The actual computation of the covered code parts is then done offline by analyzing the recorded coverage data and combining this information with the application's source code. The problem with this approach is that instrumentation degrades the performance of the executed program and is therefore typically only enabled during testing and switched off in daily operation.

For Java, several tools exist that can be used for obtaining exact code coverage. To the best of our knowledge, all of them add instrumentation code to the application or to the VM. However, instrumenting an application has some disadvantages such as slowing down the execution or even introducing bugs that cannot be reproduced without instrumentation. Approaches to reduce the overhead of the code coverage probes can be divided into two

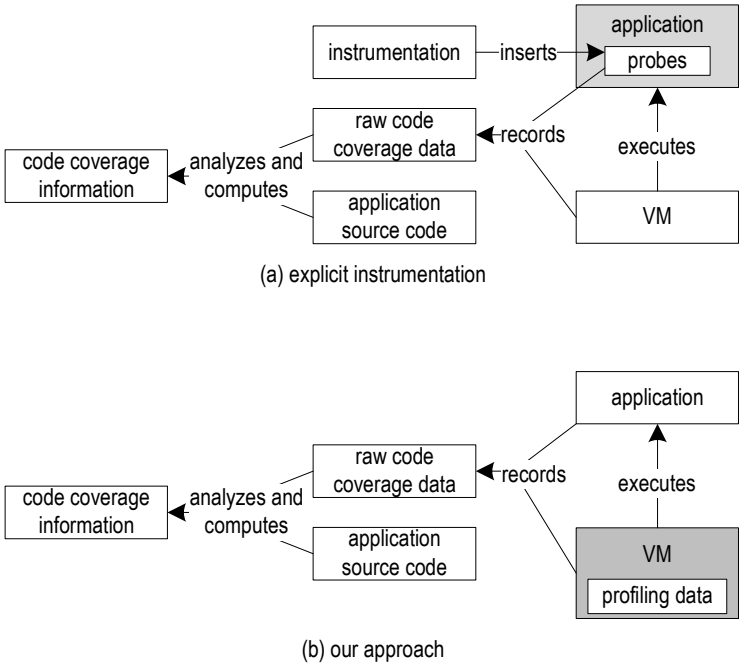


Figure 7.1: Ways to obtain code coverage information

categories (see Chapter 9.3). The first category of optimizations tries to avoid redundant probes so that fewer probes have to be inserted and less instrumentation code has to be executed at run time. The second category removes code coverage probes when they are no longer needed. Removing no longer required probes is efficient but increases the complexity of the code coverage tools.

We propose a system where no instrumentation code has to be added explicitly to record code coverage information. Instead, the code coverage information is derived from the profiling data that is recorded by modern high-performance VMs for JIT compilation, as shown in Figure 7.1 (b). We exploit the fact that modern high-performance VMs already have an instrumented interpreter or baseline compiler that records profiling data. By guaranteeing certain system properties, it is possible to derive code coverage information from the profiling data that is recorded for JIT compilation. Normally, this profiling data would only be used to guide aggressive and optimistic optimizations such as type specialization and removal of never executed code during JIT compilation. Exploiting this profiling data allows minimizing the implementation effort to obtain code coverage information, while ensuring that the impact on peak performance is minimal because it is unnecessary to instrument the executed application explicitly.

7.1 Runtime System and Requirements

Figure 7.2 shows our modified runtime system that we use to derive accurate code coverage information from the traces recorded for JIT compilation. Unlike our original trace-based runtime system, the threshold for trace recording is set to zero so that the trace recording interpreter is used from the beginning. While this has a negative impact on startup performance, it ensures that every executed code is traced. Unexecuted code is not traced and therefore not compiled. In other words, we can be sure that all compiled code has been executed and traced before. When compiled code branches off to uncompiled code, execution falls back to the trace recording interpreter that starts recording a new partial trace. Therefore, the recorded traces always comprise those parts of the code that were executed. So, code coverage information can be collected during trace recording and the compiled code does not have to be instrumented. This avoids the negative impact on the peak performance.

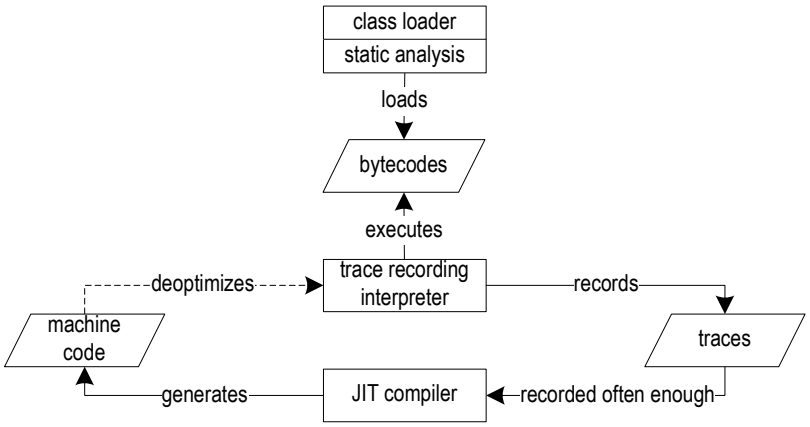


Figure 7.2: Runtime system for recording coverage information

Our approach is not only applicable to runtime systems with trace-based JIT compilers, but can be generalized to any JIT compiler that uses profiling data to guide its optimizations. It is only necessary to fulfill the following requirements and guarantees:

- The VM must use an interpreter or a baseline compiler that is instrumented for recording profiling data. In particular, it is necessary that the recorded profiling data indicates which code parts have been executed. To obtain edge coverage information, the recorded profiling data must also store which control flow edges have been executed.

- Profiling data must be recorded whenever uncompiled code is executed. Furthermore, the JIT compiler must only compile code that has been executed before. For edge coverage, it is also required that the JIT compiler only compiles control flow

edges that have been executed. So, even if a block is contained in the compiled code, all unexecuted edges to that block must fall back to the interpreter so that the profiling data is updated. Our trace-based JIT compiler does that as described in Chapter 5.

- If code needs to be executed that has not been executed before, the system must fall back to the instrumented interpreter or to the baseline compiled code so that the corresponding profiling data is updated.

Our requirements for determining code coverage are few so that it should be easy to fulfill them in most modern VMs. Thus, our approach could, for example, also be used for Oracle's method-based HotSpot server compiler as proposed in Chapter 10.1.2.

7.2 Computing Code Coverage

There are many different code coverage metrics such as method coverage, basic block coverage, instruction coverage, edge coverage, or path coverage [33]. Our work concentrates on deriving instruction and edge coverage from the recorded traces. In most cases, we could also derive path coverage information from the recorded trace information. The only problem with path coverage is that if partial traces were recorded, the path before the partial traces might be indeterminable because partial traces do not start at a trace anchor. If multiple paths from the enclosing trace anchor to the start of the partial trace are possible, we cannot decide which path should be considered as the predecessor of the partial trace.

Similar to the profiling data for a method-based JIT compiler, the recorded traces are stored in main memory while the VM is running. When the VM exits (or upon user request) we compute the code coverage information and write it to a file. Figure 7.3 shows how we compute the code coverage information from the recorded traces. At first, we do some pre-processing where we analyze the method bytecodes to compute all control flow edges. Then, we query all trace anchors for the current method and iterate over all their traces. The recorded traces only contain the raw information about the recorded control flow decisions so that we explicitly simulate the control flow decisions of each trace to determine which bytecodes it covers. Every bytecode that was covered by a trace is then marked as covered. Furthermore, we check for every covered bytecode if it is a branching instruction and if so, we mark the control flow edge that remains inside the trace as covered.

After processing all traces, the covered bytecodes and control flow edges are mapped to the source code using the line number table within the class file. However, the line number tables are optional and are only placed in the class files when the Java source to class file

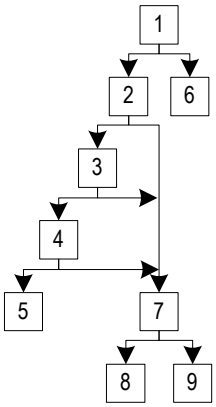
```
1 Coverage coverage(Method m, LineNumberTable lineTable) {
2   List<Bytecode> coveredBytecodes = new List<>();
3   Edges edges = computeControlFlowEdges(m.bytecodes());
4
5   for (TraceAnchor traceAnchor: m.traceAnchors()) {
6     for (Trace trace: traceAnchor.traces()) {
7       List<Bytecode> tracedBytecodes =
8           computeTracedBytecodes(trace, m.bytecodes());
9
10      for (int i = 0; i < tracedBytecodes.size(); i++) {
11        Bytecode b = tracedBytecodes.at(i);
12        coveredBytecodes.addIfMissing(b);
13        if (b.isBranch()) {
14          Bytecode target = tracedBytecodes.at(i + 1);
15          edges.markCovered(b, target);
16        }
17      }
18    }
19  }
20
21  return mapToSourceCode(lineTable, coveredBytecodes,
22      edges);
23 }
```

Figure 7.3: Computing code coverage from recorded traces

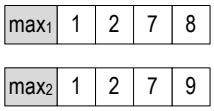
compiler is explicitly instructed to do so. If no line number tables are available, or for a more in-depth look at the recorded code coverage, the coverage information can still be visualized on the bytecode-level. The resulting coverage data is stored to a file and the actual visualization of the covered code parts is performed offline.

The recorded traces also contain further data that could be visualized, such as execution counts, type information, and information about the call targets of virtual calls. However, unlike the code coverage information, this data is not accurate and only represents the data observed during the executions in the trace recording interpreter.

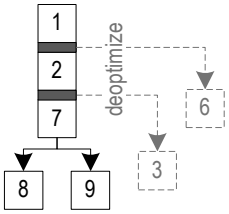
Figure 7.4 shows line and edge coverage examples for the method `Math.max()`. Figure 7.4 (a) shows the CFG of the method `Math.max()`, while Figure 7.4 (b) shows two traces for this method. Merging the traces results in the trace graph shown in Figure 7.4 (c). Figure 7.4 (d) visualizes the covered lines and control flow edges for this trace graph on the source code level. Some of the source code lines are only partially covered as not all bytecodes/edges have been executed. The presence of partial traces, such as in Figure 7.4 (e), would change the trace graph to the one shown in Figure 7.4 (f). This would also increase code coverage as indicated by Figure 7.4 (g).



(a) control flow graph



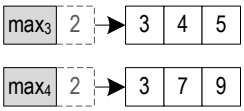
(b) traces



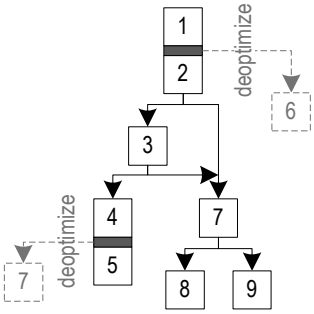
(c) initial trace graph

byte-codes	edges	
67%	1/2	`1: public static double max(double a, double b) {`
50%	1/4	`2:   if (a != a) return a; // a is NaN`
0%	0/2	`3:   if (a == 0.0d && b == 0.0d`
0%	0/0	`4:       && hasNegativeZeroDoubleBits(a)) {`
		`5:     return b;`
		`6:   }`
100%	2/2	`7:   return (a >= b) ? a : b;`
		`8: }`

(d) coverage for the initial trace graph



(e) partial traces



(f) final trace graph

byte-codes	edges	
67%	1/2	`1: public static double max(double a, double b) {`
100%	4/4	`2:   if (a != a) return a; // a is NaN`
100%	1/2	`3:   if (a == 0.0d && b == 0.0d`
100%	0/0	`4:       && hasNegativeZeroDoubleBits(a)) {`
		`5:     return b;`
		`6:   }`
100%	2/2	`7:   return (a >= b) ? a : b;`
		`8: }`

(g) coverage for the final trace graph

Figure 7.4: Coverage for the method `Math.max()`

7.3 Comparison to Other Code Coverage Techniques

Figure 7.5 compares different instrumentation techniques that can be used to record code coverage information for Java. When source code is instrumented, probes are inserted at the source code level and the modified source code is compiled to Java bytecode. Instrumenting on this level greatly simplifies the mapping of the recorded code coverage information to the source code so that this approach can even show within lines which source code parts have been executed. All other techniques must use the line number table within the Java class files to map the executed bytecodes to source code lines. The biggest disadvantage of source code instrumentation is the language dependency, especially as there are several languages that can be compiled to Java bytecode. So, a separate implementation is necessary for every language.

	source code instrumentation	offline bytecode instrumentation	online bytecode instrumentation	our approach
works without access to source code	no	yes	yes	yes
works with classes created at run-time	no	no	yes	yes
does not affect reflection	depends on the implementation	depends on the implementation	depends on the implementation	yes
negative impact on peak performance	high	high	high	low
works for very large methods	no	no	no	yes
possible to instrument all JDK classes	no	no	no	yes
can instrument container files (.jar, .war)	only if instrumented before packaging	yes	yes	yes
can instrument signed container files	only if instrumented before packaging	only if instrumented before packaging	yes	yes
supports explicitly thrown exceptions	yes	yes	yes	yes
supports implicitly thrown exceptions	only if probes are inserted after instructions that may throw	only if probes are inserted after instructions that may throw	yes	yes
works without additional instrumentation	no	no	no	yes
independent of source language	no	yes	yes	yes
difficulty to map coverage to source code	low	high	high	high
VM independent	yes	yes	yes	no

Figure 7.5: Comparison of code coverage instrumentation techniques

Bytecode instrumentation inserts probes into the bytecodes after the source code has been compiled to Java bytecode. This is either done in an offline preprocessing step or during execution of the Java program. Offline bytecode instrumentation can directly modify the class files, which reduces the run-time overhead. However, it causes problems for signed class file containers as the signature no longer matches the modified class files. Online bytecode instrumentation attaches an agent to the JVM Tool Interface (JVMTI) [58] and

modifies the bytecodes after they have been loaded by the VM class loader. This makes code coverage tools convenient to use as it allows transparent instrumentation of class file containers such as .jar or .war files. Furthermore, the attached agent can be notified when the application throws an implicit exception. This simplifies recording exact code coverage information in case of implicit exceptions. Bytecode instrumentation must use the line number table within the Java class files to map the executed bytecodes to source code lines. However, several bytecodes may map to the same line so that it can be hard to determine which parts of a line have been executed. So, it is common to report a coverage percentage for those lines where not all parts have been executed [1, 2, 6].

All approaches that explicitly add instrumentation to the code suffer from the problem that the JIT compiler generates machine code for the inserted probes. So, the probes are also executed when the code is already compiled, which decreases the peak performance. Compiling the probes also increases both the JIT compilation time and the size of the generated machine code. Our approach avoids this problem as it does not need any additional instrumentation. Instead, we reuse the profiling data that is recorded for JIT compilation by already existing instrumentation. This allows retaining almost full peak performance so that code coverage can even be recorded when the application is already deployed on a production system.

Another advantage of our approach is that we also record exact code coverage in case of implicit exceptions. For performance reasons, other approaches often only insert probes at the end of every basic block so that the whole basic block is marked as executed when this probe is executed. When an implicit exception happens before the probe, then the whole block is not marked as covered although some instructions were executed.

Some implementations of source code or bytecode instrumentation add methods and fields to the instrumented classes so that the bytecodes can update the coverage data more efficiently. This reduces the instrumentation overhead but can have undesired effects as those fields are visible using reflection. Modifying top-level classes such as `java.lang.Object` may also cause serious problems because the VM often relies on a specific object layout for those classes. So, instrumenting an application and all its libraries in such a way is not always possible. Our approach is transparent and does not modify the executed application in any way.

The only drawback of using profiling data to compute code coverage information is that the technique is VM-dependent as every VM records profiling data in a different way.

7.4 Code Coverage Tools

For Java, there are many open source and commercial code coverage tools of which we list a few popular ones.

Source code instrumentation is, for example, used by *CodeCover* [3] (open source) and *Clover* [4] (commercial). Because of instrumenting the source code these tools are language dependent but have the advantage that they can map the executed probes to the source code more easily.

EMMA [1] and *Cobertura* [2] are both open source tools that use bytecode instrumentation and both have been popular for several years. However, EMMA and Cobertura are no longer under active development and therefore they do not support the new language features that were introduced with Java 7. *JaCoCo* [6] is a still supported open source tool that does both online and offline bytecode instrumentation and supports the latest Java features. The online bytecode instrumentation greatly simplifies the use of this tool as it is sufficient to add just one command line flag to instrument a Java application transparently. *EclEmma* [5] is an open source Eclipse plugin that can directly visualize the coverage information in Eclipse. Previously, EclEmma used EMMA to record the code coverage information but then switched to JaCoCo because EMMA was no longer under active development and did not support the latest Java features.

Many code coverage tools do not support implicitly thrown exceptions so that the coverage information is inaccurate when such an exception is thrown. For example, EMMA and JaCoCo assume that all instructions before a probe are executed as one atomic unit so that those instructions are only marked as covered after the successive probe has been executed. This reduces the instrumentation overhead and increases the performance but has the drawback that the code coverage information is inaccurate in the case of implicitly thrown exceptions.

Figure 7.6 (a) shows the source code and Figure 7.6 (b) the bytecode of a simple method that does some synchronization and returns the absolute value of a static field. When the code is instrumented with JaCoCo, the bytecode in black type (0 to 41) shown in Figure 7.6 (c) is generated. The bytecodes 0 to 3 are used to get an array of boolean values that contains the recorded coverage data for the current method. This array is then stored in a local variable that is accessed by the probes that update the code coverage information (see bytecodes 7 to 10, 19 to 22, 29 to 32, and 37 to 40).

In this specific example, the bytecode instrumentation causes an unexpected performance bug. Figure 7.6 (b) shows that no instruction in the synchronized block could throw an exception (bytecodes 3 to 14). JaCoCo inserts three probes in the synchronized block and

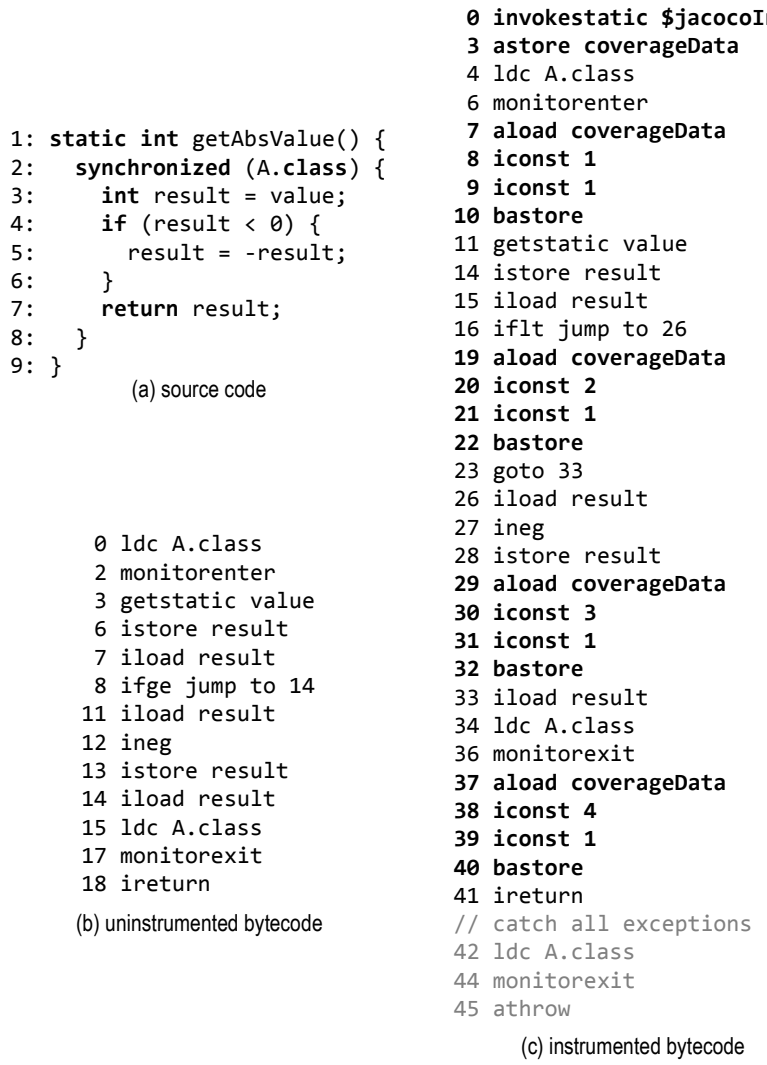


Figure 7.6: Code instrumented with JaCoCo

all execute an array store to update the coverage array. However, the array store instruction `bastore` may throw an exception. Although those bytecodes are valid according to the JVM specification [54], this results in a construct that a Java source to class file compiler such as *javac* would not generate. Those compilers always add a catch-all exception handler that unlocks the monitor in case of an exception and rethrows the exception, see bytecodes 42 to 45 in gray type in Figure 7.6 (c).

A Java JIT compiler can always decide to avoid compiling certain bytecode constructs for performance or implementation reasons. So, before compiling a method or method parts, the HotSpot compilers validate that the bytecode fulfills certain invariants and assumptions, especially when monitors are used. In this example, it turns out that the bytecodes 10, 22, and 32 may throw an `ArrayIndexOutOfBoundsException` that would unwind the method although there are still locked monitors remaining. HotSpot does not

allow this behavior for compiled code so that the bytecodes are not compiled by the JIT compiler and are executed in the interpreter instead. The interpreter is prepared for such bytecodes and in case of an exception it unlocks all remaining monitors before unwinding the stack frame.

The trace-based compiler that is described in this thesis also checks the same invariants and assumptions as the method-based HotSpot compilers. Thus, none of the HotSpot compilers would generate machine code for this instrumented method so that this method is only executed in the interpreter. Similar cases happened for a few methods in the evaluated benchmark suites (see Chapter 8.3) so that this affected the performance of some benchmarks when JaCoCo was used to determine code coverage. We also checked this for the code coverage frameworks EMMA and Cobertura, which also use bytecode instrumentation, and they suffer from the same issue.

7.5 Startup Performance

The unmodified trace-based runtime system uses two different interpreters: a normal and a trace recording interpreter. The normal interpreter does not do any tracing and is used during startup for executing the bytecodes. To obtain exact code coverage information, our modified runtime system must only use the trace recording interpreter as shown in Figure 7.2. This decreases the startup performance because tracing happens more frequently than before.

However, the recorded trace information is more accurate because more traces are being recorded before compiling them to machine code. The trace-based JIT compiler uses the recorded traces for optimistic optimizations so that more accurate information results in fewer invalidated optimizations. So, deoptimization is required less frequently which has a positive impact on the startup performance. Therefore, the startup performance depends on how often traces are recorded for a trace anchor before a JIT compilation is triggered as the positive effect of the more accurate trace information competes against the negative effect of the higher trace recording overhead.

Figure 7.7 shows the used thresholds for three different configurations. The first configuration is the unmodified trace-based system that uses both the normal and the trace recording interpreter.

For determining code coverage, the tracing thresholds for method entries and loop headers were reduced to zero, so that only the trace recording interpreter is used. This results in the configuration *trace-based coverage untuned* that mimics the compilation behavior of the unmodified trace-based compiler by increasing the compilation thresholds accordingly.

	original trace-based system	trace-based coverage untuned	trace-based coverage tuned
method tracing threshold	3000	0	0
method traces compilation threshold	1000	4000	3000
loop tracing threshold	25000	0	0
loop traces compilation threshold	1000	26000	4000

Figure 7.7: Thresholds used for trace recording and JIT compilation

Depending on the executed application, this can have a significant impact on startup performance because especially loops are trace recorded many times. In our configuration *trace-based coverage tuned*, we significantly decreased the compilation threshold for loops and we slightly decreased the compilation threshold for method traces. The evaluation in Chapter 8.3 on page 86 shows that this improves the startup performance to the level of the unmodified trace-based runtime system.

Chapter 8

Evaluation

We evaluate several variants of our trace-based compiler and our code coverage technique on the benchmark suites SPECjvm2008 [64], SPECjbb2005 [63], and DaCapo 9.12 Bach [13].

8.1 Methodology

Our trace-based JIT compiler was implemented for the IA-32 architecture of Oracle's Java HotSpot VM using the early access version b12 of the upcoming JDK 8 [59]. For benchmarks we use an Intel Core-i5 processor with 4 cores running at 2.66 GHz, 4 * 256 kb L2 cache, 8 MB shared L3 cache, 8 GB main memory, and with Windows 8 Professional as the operating system. All presented configurations use the parallel stop-the-world garbage collector. Each benchmark suite was executed 15 times and we report the average of the individual benchmark results along with the 95% confidence interval. Furthermore, we show for each benchmark suite, the geometric mean of all its benchmarks, which is also the official result for the benchmark suite.

In the following, we frequently use the term *amount of generated machine code*. For all configurations, this includes both the executable machine but also the debugging information necessary for deoptimization. This is necessary to ensure a fair comparison with our trace-based compiler as it generates less executable machine code but more debugging information because of its optimistic optimizations.

8.1.1 SPECjvm2008

The SPECjvm2008 benchmark contains 38 benchmarks that are grouped into 11 benchmark categories of which 10 measure peak performance. The 11^{th} benchmark category is the *startup* category, in which the startup performance is determined by starting a new

VM for each sub-benchmark and measuring the time for performing one benchmark operation. The left half of Figure 8.1 shows those categories and roughly characterizes their workload. Furthermore, the figure also shows the average amount of machine code that is generated when each category is executed with the Java HotSpot server compiler. This indicates that the SPECjvm2008 benchmark suite contains several benchmarks, such as *mpegaudio* and *scimark*, which are loop-intensive and very small in terms of hot code size. Those benchmarks only consist of a few loops where all hot code is located and which are executed over and over again.

The figures on the following pages only show 9 peak performance categories because we combine *scimark.small* and *scimark.large* to reduce the verbosity. For computing the average results, we internally treat them as two categories to ensure comparability with other SPECjvm2008 peak performance results. Furthermore, we omit the startup category as we did a better and more detailed startup performance analysis (see Chapter 8.2.4).

We use a heap size of 1024 MB and each SPECjvm2008 benchmark executes a 2 minutes warmup phase before measuring the peak performance for 4 minutes. Before starting the warmup, every benchmark determines the available number of hardware threads and spawns that number of worker threads. We executed the benchmarks on a machine with 4 cores so that 4 worker threads are spawned. The benchmark result is then the average number of benchmark operations per minute executed during the 4 minutes of peak performance measurement.

SPECjvm2008	description	size[1]
startup	startup performance	0.3
compiler	javac compilation	8.8
compress	compression algorithm	0.5
crypto	cryptography (AES, RSA,...)	0.9
derby	database	3.0
mpegaudio	MP3 audio decoding	0.8
scimark.small	FFT, LU factorization,... (small dataset)	0.5
scimark.large	FFT, LU factorization,... (big dataset)	0.4
serial	(de)serialization	1.1
sunflow	raytracing	1.1
xml	xml processing	4.7

DaCapo 9.12 Bach	description	size[1]
avrora	microcontroller simulation	0.9
batik	SVG processing	3.2
eclipse	Eclipse IDE performance tests	22.6
fop	PDF generation	3.4
h2	database	4.1
jython	Python implementation	9.1
luindex	builds a search index	1.4
lusearch	keyword search in a dataset	1.6
pmd	analysis of Java classes	6.7
sunflow	raytracing	0.6
tomcat	queries a local webserver	7.8
tradebeans	daytrading via JavaBeans	4.5
tradesoap	daytrading via SOAP	10.2
xalan	exports XML to HTML	3.8

[1] megabytes of generated machine code and debugging information when executed with the Java HotSpot server compiler

Figure 8.1: SPECjvm2008 and DaCapo 9.12 Bach benchmarks

8.1.2 SPECjbb2005

The SPECjbb2005 benchmark simulates a client/server business application where all operations are performed on an in-memory database that is partitioned into so-called warehouses where each warehouse is processed by one thread. How often the benchmark is executed and which runs are considered as either warmup or peak performance depends on the number of hardware threads. We used a system with 4 cores and a heap size of 1200 MB for benchmarking so that the official SPECjbb2005 throughput in business operations per second (bops) is defined as the geometric mean of the performance for the warehouses 4 to 8 which were each executed for 4 minutes. The warehouses 1 to 3 were considered the warmup phase and each warehouse was only executed for 30 seconds.

When executed with the Java HotSpot server compiler 1.3 MB of machine code and debugging information are generated. So, in this aspect the benchmark has a comparable size to SPECjvm2008 *serial* or DaCapo 9.12 Bach *luindex*.

8.1.3 DaCapo 9.12 Bach

The DaCapo 9.12 Bach benchmark suite consists of 14 object-oriented applications, as shown in the right half of Figure 8.1. We executed each benchmark with a heap size of 1024 MB for 20 times with the default data size so that the execution time converges. The first run shows the startup performance of the VM, while the fastest run shows the peak performance. Many benchmarks of the DaCapo 9.12 Bach benchmark suite are considerably larger than the SPECjbb2005 benchmark or the SPECjvm2008 benchmarks. Some examples for those large benchmarks are *eclipse*, *jython*, and *tradesoap*.

8.2 Trace-based Compilation

In terms of peak performance, we evaluated the following configurations:

- Our baseline is the unmodified method-based Java HotSpot client compiler from the early access version b12 of the upcoming JDK 8 because our trace-based JIT compiler is based on the client compiler. All results are normalized to the results of this configuration which is only shown implicitly in the figures as the 100% mark.
- The configuration *tracing peak performance* represents our trace-based JIT compiler and uses aggressive trace inlining and all exception handling mechanisms described in this thesis. It is optimized for peak performance while generating still reasonable amounts of machine code. For nested loops we ensure that inner loops are at least inlined into the outermost loop of the same method. This increases the performance

when the inner and the outer loops operate on the same values, because the loops are optimized as one compilation unit.

- The configuration *tracing minimum code* also represents our trace-based JIT compiler and uses all exception handling mechanisms described in this thesis. However, less aggressive trace inlining is used so that loop traces are less frequently inlined and more often compiled separately. This decreases the amount of generated machine code while not sacrificing too much peak performance.

- The configuration *HotSpot server* represents the unmodified, method-based Java HotSpot server compiler from the early access version b12 of the upcoming JDK 8. This JIT compiler is explicitly designed for best peak-performance and performs significantly more optimizations than our trace-based compiler, e.g., escape analysis and sophisticated loop optimizations.

8.2.1 SPECjvm2008

Figure 8.2 shows that all our trace-based configurations outperform the method-based HotSpot client compiler. Our tracing configurations show the highest speedups on the benchmarks *derby* and *serial*. On the benchmarks *compress* and *sunflow*, our configuration *tracing peak performance* even outperforms the Java HotSpot server compiler. This is mainly due to the aggressive trace inlining but the type system with the context-sensitive information also has some impact. So, this indicates that a significantly simpler structured trace-based compiler can outperform one of today's best method-based JIT compilers. In general, our tracing configurations achieve a peak performance that is somewhere between the Java HotSpot client and the HotSpot server compiler. On average, the speedup to the client compiler is 17% with aggressive inlining and 12% with moderate inlining.

For rather small and loop-intensive benchmarks such as *crypto*, *mpegaudio*, and *scimark* we achieve only a small speedup compared to the client compiler. This is the case because our trace-based compiler does not perform any sophisticated loop optimizations yet. Furthermore, trace inlining does not have a significant advantage over method inlining on small and loop-intensive benchmarks because a method-based compiler can also inline all relevant code parts [38].

The dark bars in Figure 8.3 show the total amount of generated machine code, while the light bars indicate the amount of machine code that was invalidated because optimistic optimizations deoptimized too frequently. The figure shows that our tracing configurations generate less machine code than the client compiler on average. Especially our configuration *tracing minimum code* is efficient in terms of generated machine code. For the benchmarks *compiler* and *xml*, some of our tracing configurations use too aggressive

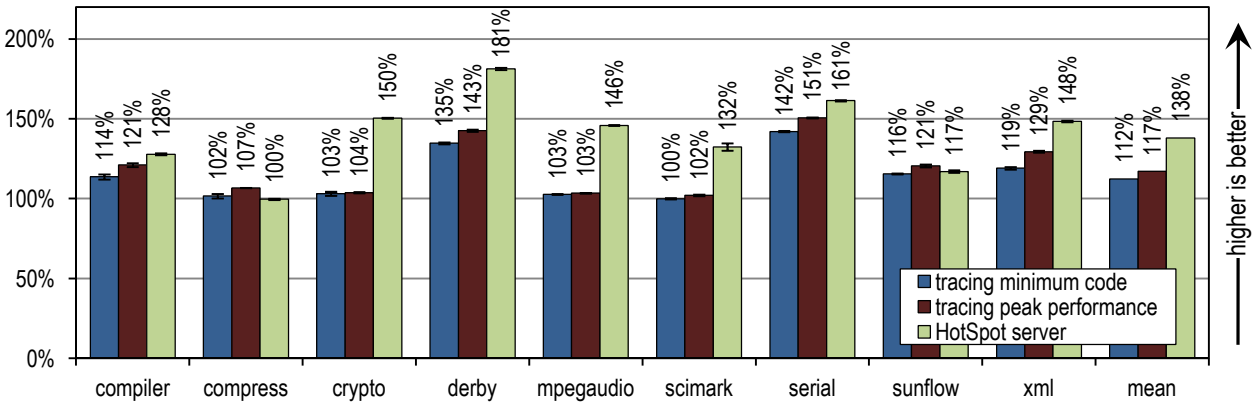


Figure 8.2: SPECjvm2008: peak performance

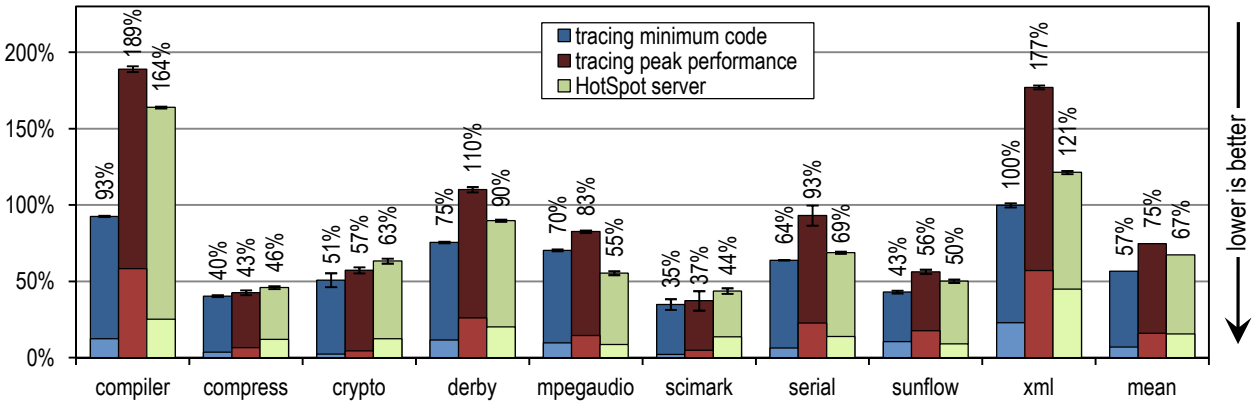


Figure 8.3: SPECjvm2008: generated machine code

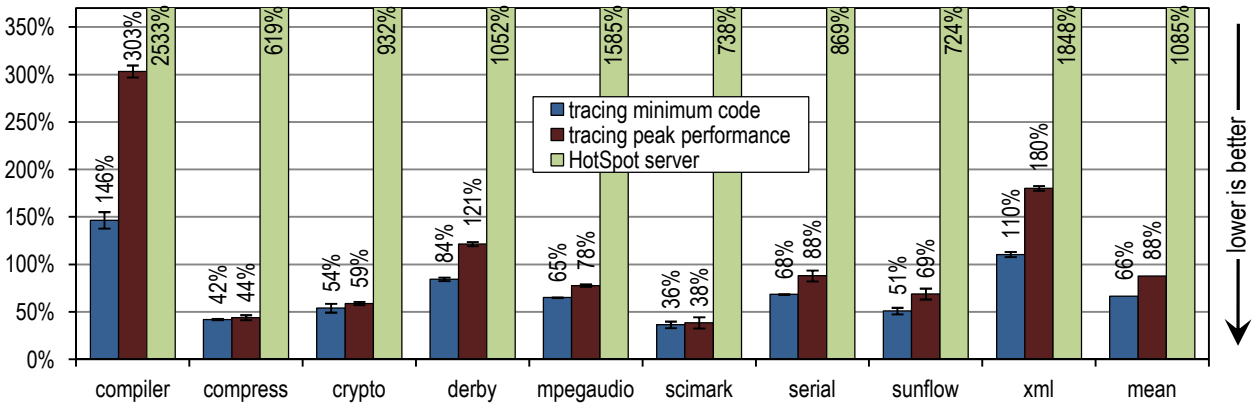


Figure 8.4: SPECjvm2008: time required for JIT compilation

optimizations during startup, so that the overspecialized code deoptimizes frequently and must be recompiled later on, which results in a fairly high amount of invalidated machine code. Small and loop-intensive benchmarks do not show a large increase in code size, even when our most aggressive trace inlining heuristic is used. On such benchmarks, a trace inlining heuristic can only be too conservative but never too aggressive.

Figure 8.4 shows the time required for JIT compilation. On average, our tracing configuration *tracing minimum code* spends 34% less time on JIT compilation than the client compiler, while our configuration *tracing peak performance* still needs 12% less time than the client compiler. The server compiler, which is explicitly designed for peak performance and which performs significantly more optimizations than our trace-based compilers, spends more than 12x as much time on JIT compilation as our configuration *tracing peak performance*.

8.2.2 SPECjbb2005

Figure 8.5 shows the peak performance, the generated machine code and the compilation time for the SPECjbb2005 benchmark. Both trace-based compiler variants outperform the client compiler significantly in terms of peak performance. More aggressive trace inlining results in a higher performance but also generates more machine code and requires a longer compilation time because of the larger size of the compilation units. The peak performance of the SPECjbb2005 benchmark clearly profits from increased trace inlining aggressiveness. In terms of compilation time and amount of generated machine code, our configuration *tracing minimum code* is especially efficient, while reaching a decent peak performance. However, the SPECjbb2005 benchmark profits heavily from some of the time-consuming optimizations of the server compiler, so that our trace-based compiler reaches only 69% of the server compiler's peak performance.

Figure 8.6 shows the SPECjbb2005 peak performance for different numbers of warehouses. The maximum peak performance is reached with 4 warehouses as every warehouse is processed by one thread and our benchmarking system has 4 cores. With a higher number of warehouses, the additional threading overhead decreases the performance for all configurations. The figure shows that our tracing configurations outperform the method-based client compiler independently of the used number of warehouses.

8.2.3 DaCapo 9.12 Bach

Figure 8.7 shows the peak performance results for the DaCapo 9.12 Bach benchmark suite. Compared to the client compiler, our configurations *tracing minimum code* and *tracing peak performance* show a similar or higher peak performance on all benchmarks. For

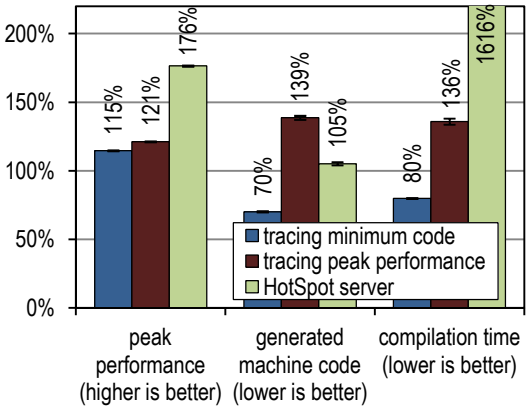


Figure 8.5: SPECjbb2005 results

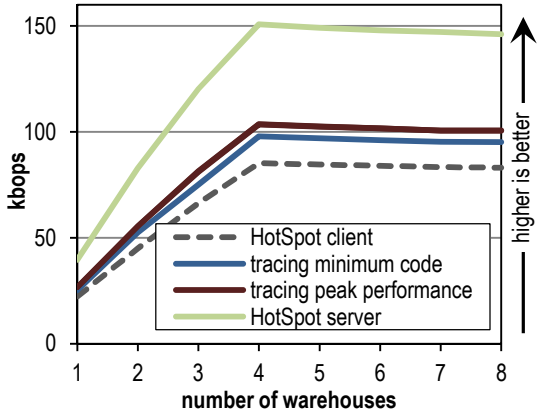


Figure 8.6: SPECjbb2005 peak performance for different numbers of warehouses

the benchmarks *luindex*, *pmd*, and *sunflow*, our configuration *tracing peak performance* even outperforms the server compiler. The benchmarks *luindex* and *sunflow* profit from the aggressive trace inlining and the context-sensitive type information, while the high performance of *pmd* is the result of trace inlining and the good exception handling of our trace-based compiler. However, on some of the benchmarks the configuration *tracing peak performance* is already over-aggressive so that more machine code is generated without a measurable change in peak performance.

In general, our tracing configurations achieve a peak performance between the client and the server compiler, where *tracing peak performance* is on average 20% faster than the client compiler and only 6% behind the server compiler. Compared to the client compiler, we achieve the highest speedup on the benchmark *jython*, which executes a large number of virtual calls that can be inlined by our compiler. This benchmark also profits from trace inlining and the larger compilation units of our trace-based approach.

The dark bars in Figure 8.8 show the total amount of generated machine code, while the light bars indicate the amount of machine code that was invalidated because optimistic optimizations deoptimized too frequently. Our configuration *tracing minimum code* generates less machine code than the client compiler, while the configuration *tracing peak performance* generates more machine code. The HotSpot server compiler invalidates significantly less machine code than our configuration *tracing peak performance*. This indicates that our trace-based compiler uses more optimistic optimizations.

Figure 8.9 shows that our configuration *tracing minimum code* requires 25% less time for JIT compilation than the client compiler, while *tracing peak performance* requires 14% more time than the client compiler. The server compiler performs significantly more

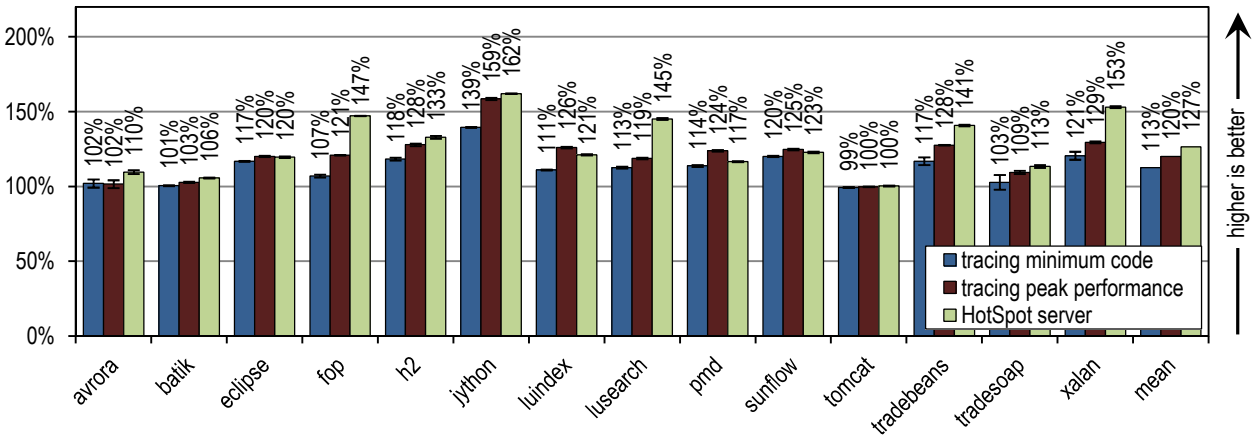


Figure 8.7: DaCapo 9.12 Bach: peak performance

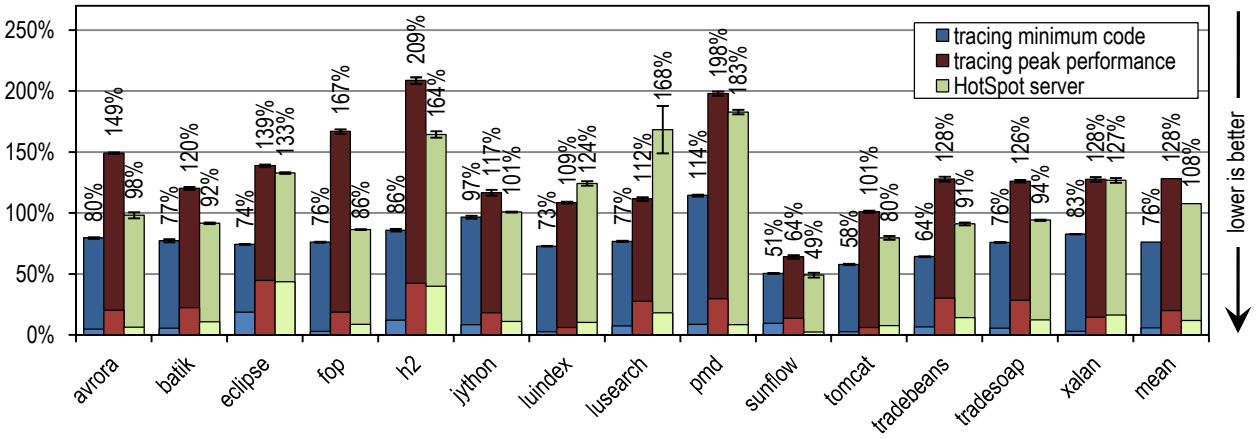


Figure 8.8: DaCapo 9.12 Bach: generated machine code

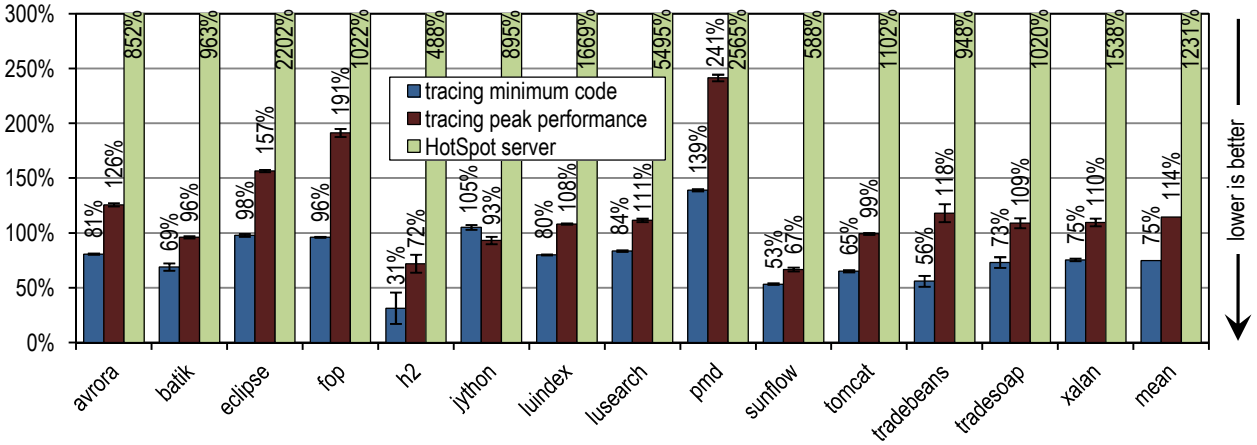


Figure 8.9: DaCapo 9.12 Bach: time required for JIT compilation

optimizations and spends more than 12x as much time on JIT compilation as the client compiler.

8.2.4 Startup Performance

The Java HotSpot client and server compilers as well as our trace-based JIT compiler are all designed for multi-threaded background compilation. So, we evaluate the startup performance in the following scenarios:

- The first scenario executes *1 application thread*, while the VM uses up to 4 JIT compiler threads. So, on our 4 core benchmarking system, up to 3 cores can be exclusively used for JIT compilation.
- In the second scenario, *4 application threads* are executed while the VM uses up to 4 JIT compiler threads. On our 4 core benchmarking system, the JIT compiler threads compete with the application threads. The Java HotSpot VM assigns a higher priority to JIT compiler threads as early JIT compilation has a positive effect on startup performance.

All presented results are normalized to the performance of the Java HotSpot client compiler with 1 JIT compiler thread. We do not present any startup performance results for the SPECjbb2005 benchmark as this benchmark is only designed to measure peak performance so that first results are obtained after 30 seconds where all configurations are already close to their peak-performance.

For the SPECjvm2008 benchmark suite, we measured the startup performance by executing one operation for each benchmark. Figure 8.10 shows the scenario when JIT compilation can be offloaded to otherwise idle cores. There, the HotSpot server compiler shows the best startup performance because the SPECjvm2008 benchmark suite contains several small benchmarks where there is little to compile and which almost reach their peak performance after compiling the innermost benchmark loop. This is the ideal case for the server compiler which does optimize loops especially well so that its peak performance advantage also affects the startup performance results if idle cores are available for compilation. Figure 8.12 shows the scenario where the compilation threads compete with the application threads. There, the HotSpot client compiler and our trace-based compiler achieve good results as both spend little time on JIT compilation and are therefore less affected by the increased number of application threads.

The figures also show that the number of compiler threads hardly affects the client compiler or our trace-based compiler because both JIT compilers require little time for compilation. The server compiler requires much more time for compilation and therefore greatly profits

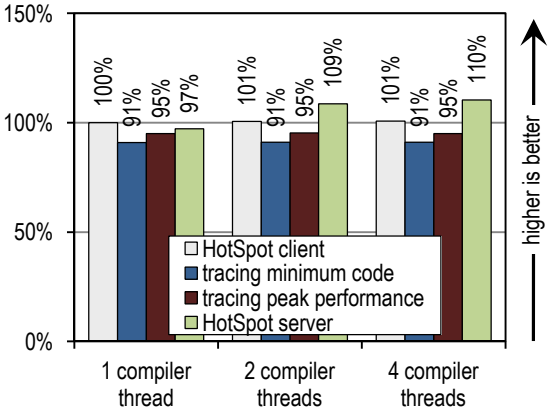


Figure 8.10: SPECjvm2008 startup performance with 1 application thread

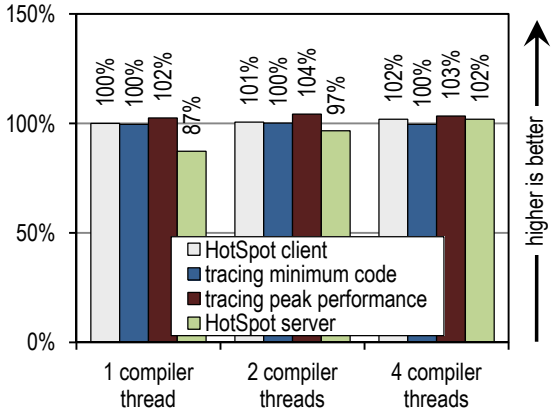


Figure 8.12: SPECjvm2008 startup performance with 4 application threads

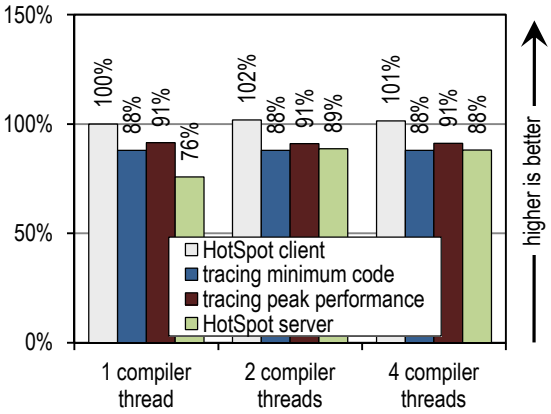


Figure 8.11: DaCapo 9.12 Bach startup performance with 1 application thread

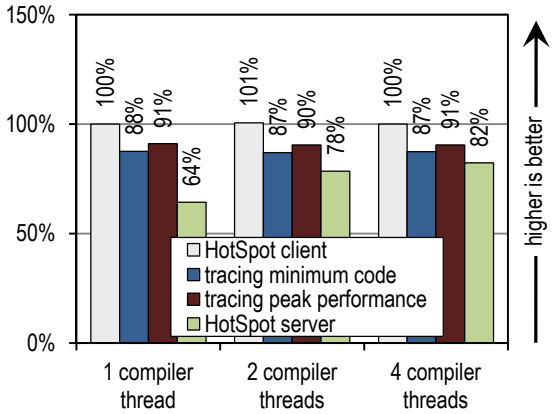


Figure 8.13: DaCapo 9.12 Bach startup performance with 4 application threads

from more than one compilation thread, especially if there are idle cores that can be used.

For the DaCapo 9.12 Bach benchmark suite, we measured the startup performance by executing one iteration for each benchmark. The benchmark results are shown in Figure 8.11 and Figure 8.13. Again, the number of compiler threads hardly affect the client compiler or our trace-based compiler because those require too little time for compilation. In contrast to that, the server compiler profits when more than one thread is used for JIT compilation. However, our configuration *tracing peak performance* always shows a higher startup performance than the server compiler, even if the server compiler can use idle cores for JIT compilation. This is the case because the DaCapo 9.12 Bach benchmarks are significantly more complex than the SPECjvm2008 benchmarks [37] so that the JIT compilation performance is the dominating factor for startup performance.

So, on the DaCapo 9.12 Bach benchmarks, our trace-based configurations show a roughly 10% slower startup performance than the HotSpot client compiler. While our trace-based compiler often requires even less time for JIT compilation than the HotSpot client compiler, it shows a lower startup performance because of the additional overhead for trace recording and deoptimization that mainly incur during startup. Furthermore, when our trace-based compiler compiles traces for the first time, it may happen that not all relevant paths have been recorded yet. This is a common problem for trace compilation, as different parts of a method might be hot during different execution phases of the application [69]. However, this drawback is outweighed by the significantly improved peak performance. Unlike the HotSpot server compiler, our trace-based compiler does not rely on idle cores to achieve a good startup performance.

8.2.5 Importance of Exception Handling

Exceptions are often seen as rare events that hardly have an effect on performance, no matter how efficient the exception handling mechanism is implemented. We evaluated the following configurations to show the importance of exception handling for Java applications:

- Our baseline is the unmodified method-based Java HotSpot client compiler from the early access version b12 of the upcoming JDK 8 because our trace-based JIT compiler builds on the client compiler. All results are normalized to the results of this configuration which is only shown implicitly in the figures as the 100% mark.

- The configuration *tracing peak performance* is the same configuration as we used to evaluate our tracing implementation in terms of peak performance in the previous chapter. It uses aggressive trace inlining and all exception handling mechanisms described in this thesis.

- The configuration *tracing abort on exception* is also optimized for peak performance but aims to illustrate how important good exception handling is for the peak performance of complex applications. When an exception is thrown, this configuration aborts trace recording and omits the currently recorded traces. This simple approach is commonly used in literature [12, 31, 32] but has the disadvantage that those instructions which always throw an exception are never compiled. We also tried another variant that stops recording the current trace when an exception is thrown so that the instruction that throws the exception and the corresponding exception handler are never within the same compilation unit. This approach was for example used by [11] and [47] but shows almost identical performance as aborting the traces so that we omit detailed results.

Figure 8.14 shows the peak performance for those SPECjvm2008 and DaCapo 9.12 Bach benchmarks where the changed exception handling has an impact on peak performance. The SPECjbb2005 benchmark is not shown in the figure, because changing the exception handling has no impact on peak performance there. Most SPECjvm2008 benchmarks are rather small so that there are only few places where exceptions are thrown. So, changing the exception handling only has a modest impact on the two largest benchmarks *compiler* and *xml*. However, several of the DaCapo 9.12 Bach benchmarks use exceptions so that the bad exception handling of the configuration *tracing abort on exception* significantly affects the performance. For some benchmarks, such as *lusearch* and *pmd*, good exception handling is crucial. Ignoring exception handling during trace recording, as done in most of the related work, leads to unacceptable slowdowns for some applications and is therefore not suitable for production use. Our tracing of thrown exceptions as well as our trace inlining approach ensure that frequently thrown exceptions are compiled together with the corresponding exception handler. This significantly increases the performance of benchmarks with lots of exception handling. Compiling frequently executed exception throwing bytecodes and their corresponding exception handlers has a hardly measurable effect on the size of the generated machine code and the compilation time so that we omit detailed results.

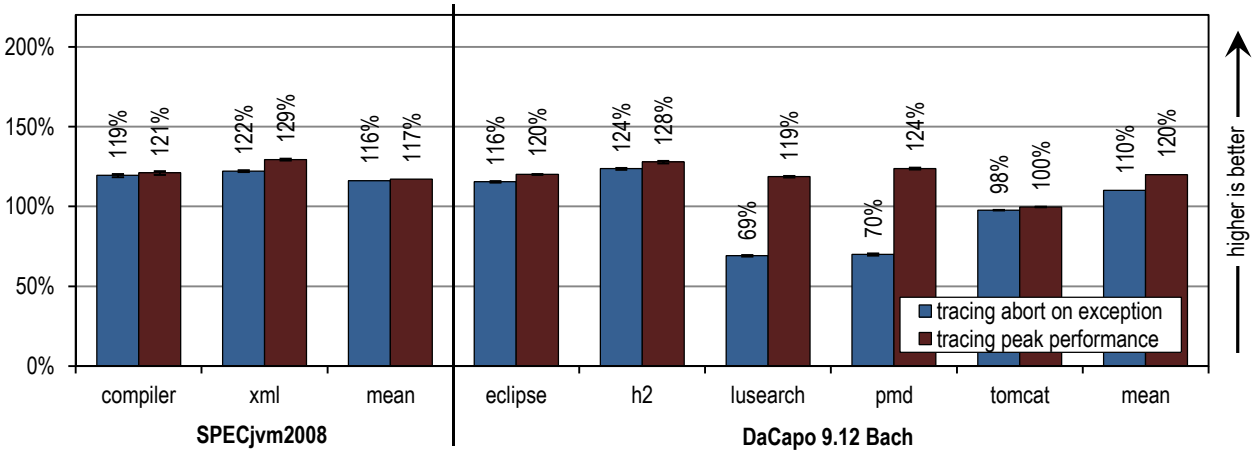


Figure 8.14: Importance of exception handling for peak performance

8.2.6 Effect of Larger Compilation Scope

Similar to method-inlining, trace inlining is an optimization that has positive effects on other compiler optimizations due to the larger compilation scope. Figure 8.15 compares the impact on peak performance for different high-level optimizations that are used by both the method-based HotSpot client compiler and our trace-based compiler. Depending on the benchmarks the individual optimizations show different gains. Overall, we see the

highest increase in effectiveness for *canonicalization*, which performs simple optimizations such as constant folding and dead code elimination. The larger compilation scope results in more values being constant which has a positive effect.

	client compiler speedup			tracing compiler speedup (peak perf.)		
	SPECjbb2005	SPECjvm2008	DaCapo 9.12	SPECjbb2005	SPECjvm2008	DaCapo 9.12
canonicalization [1]	2%	3%	1%	4%	4%	2%
conditional expression elimination [2]	1%	0%	-1%	0%	0%	0%
block merging [1,2]	0%	0%	-1%	0%	1%	1%
value numbering [1,2]	0%	2%	0%	0%	2%	1%
load/store elimination [1]	0%	1%	0%	0%	0%	0%
null-check elimination [2]	2%	2%	1%	4%	2%	1%
all optimizations [1,2]	**5%**	**7%**	**4%**	**9%**	**8%**	**6%**

[1] local optimization that is applied while building the high-level intermediate representation

[2] global optimization that is applied after building the high-level intermediate representation

Figure 8.15: Impact of high-level optimizations on peak performance

Figure 8.16 shows how the peak performance of the DaCapo 9.12 Bach benchmark suite and of its benchmarks *batik* and *jython* is affected when changing the amount of trace inlining. The benchmark *jython* is an example for an application that profits from aggressive trace inlining, i.e., the peak performance increases when inlining up to 250 bytecodes per call site. On the other hand, the benchmark *batik* hardly shows an increase in peak performance when inlining more than 50 bytecodes per call site. The average performance of the DaCapo benchmarks increases up to 200 inlined bytecodes per call site.

Figure 8.17 and Figure 8.18 show how trace inlining affects the amount of generated machine code and the time required for JIT compilation. The benchmark *batik* and the mean over all DaCapo 9.12 Bach benchmarks show a steady increase for both metrics when we increase the amount of inlining. However, the benchmark *jython* behaves differently. There, both the amount of generated machine code and the time required for JIT compilation decrease when more than 150 bytecodes are inlined per call site. This occurs because our trace inlining policy avoids inlining traces that were already compiled separately and resulted in large amounts of machine code. In that case, we assume that the previous compilation unit was already sufficiently large to allow good optimizations and we do not do the inlining to avoid code bloat. So, when more than 150 bytecodes are inlined per call site, different compilation units are chosen because of this heuristic. A more detailed analysis of different trace inlining heuristics can be found in [38] and [39].

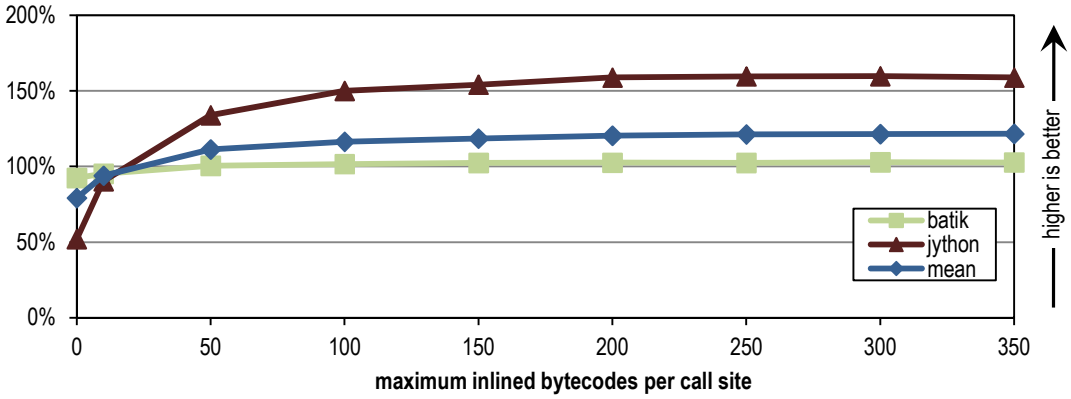


Figure 8.16: DaCapo 9.12 Bach: effect of trace inlining on peak performance

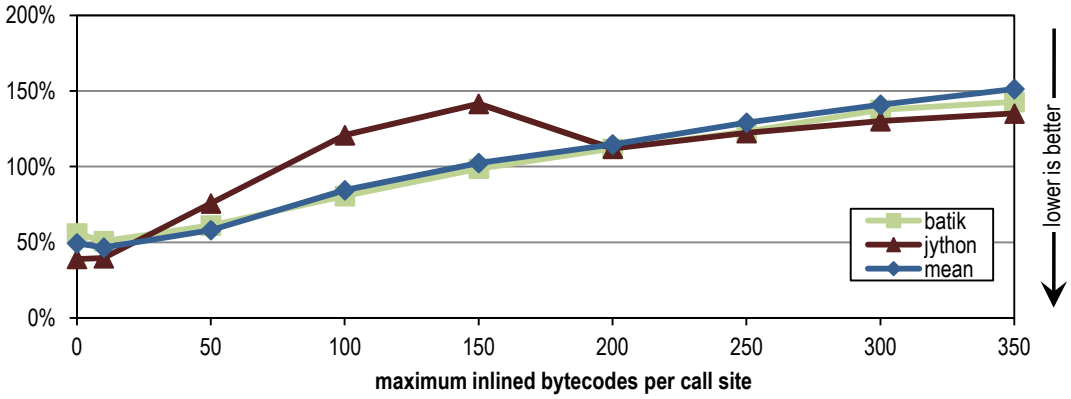


Figure 8.17: DaCapo 9.12 Bach: effect of trace inlining on the amount of generated machine code

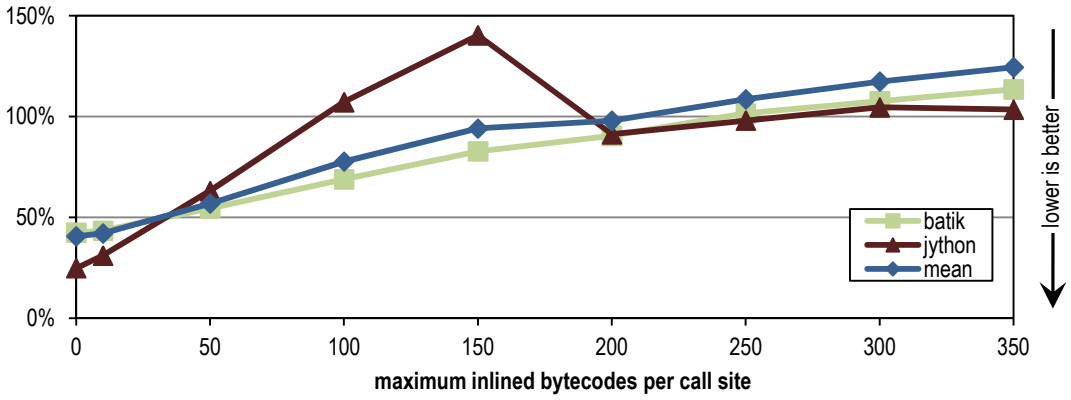


Figure 8.18: DaCapo 9.12 Bach: effect of trace inlining on the compilation time

8.2.7 Trace Transitioning

Figure 8.19, Figure 8.20, and Figure 8.21 show the average trace transition frequencies when the benchmark suites SPECjvm2008, SPECjbb2005, and DaCapo 9.12 Bach are executed with the configuration *tracing minimum code*. Most transitions occur from compiled code to compiled code, which is the fourth category according to Figure 5.6 on page 36. This is the case because all relevant code parts have been compiled when the application reaches its peak performance. Furthermore, the number of transitions to compiled method traces is significantly larger than the number of transitions to compiled loop traces. This has two reasons. First, compiled loops are invoked less frequently as execution stays within loops for a longer time than within methods because the loop body is typically executed multiple times. To directly compare those two numbers, the loop invocations would have to be scaled by the average number of loop iterations. Second, it is disadvantageous to focus solely on compiling loops as aggressive trace inlining might result in huge compilation units before inlining reaches the next loop. Furthermore, the compiler would still lack information about the values that flow into the loop because those values are passed in from the enclosing method.

from frame \ to frame	interpreted method traces	interpreted loop traces	compiled method traces	compiled loop traces
interpreted method traces	5030	same frame	8303	356
interpreted loop traces	5274	same frame	23026	0.1
compiled method traces	715	165	1494019	156765
compiled loop traces	91	0.4	463860	39832

Figure 8.19: Transition frequencies in thousands for the SPECjvm2008 benchmarks

from frame \ to frame	interpreted method traces	interpreted loop traces	compiled method traces	compiled loop traces
interpreted method traces	2847	same frame	226194	190
interpreted loop traces	0	same frame	46	0
compiled method traces	289	0.3	3597458	103074
compiled loop traces	9	0	3795197	0

Figure 8.20: Transition frequencies in thousands for the SPECjbb2005 benchmark

from frame \ to frame	interpreted method traces	interpreted loop traces	compiled method traces	compiled loop traces
interpreted method traces	13427	same frame	52715	528
interpreted loop traces	3	same frame	22	1
compiled method traces	2655	2	1288883	83227
compiled loop traces	286	0.1	504250	38755

Figure 8.21: Transition frequencies in thousands for the DaCapo 9.12 Bach benchmarks

8.2.8 Further Evaluations

Varying the heap size only results in a slight performance difference due to the different garbage collection overhead. This is the case because no tracing-specific modifications were necessary to the JVM garbage collectors.

We also experimented with even more aggressive trace inlining than that used in our configuration *tracing peak performance*. If the maximum inlining size is increased further, the size of generated machine code increases without an additional positive effect on the peak performance. This seems to be the case because of two reasons. First, for many benchmarks the configuration *tracing peak performance* already results in large compilation units so that good optimizations are possible. Second, deoptimizing to the interpreter gets more expensive with an increasing trace inlining depth because more values are alive and must be saved during deoptimization. This also results in more interpreter frames that must be created and filled with the saved values.

Our bytecode preprocessing step, which is performed once for every method, is already implemented efficiently but could be optimized further. While executing the Dacapo 9.12 Bach benchmark *tradesoap*, the class loader processes classes with a total of more than 80,000 methods. This is the highest number used in any of the benchmarks and stresses our bytecode preprocessing step that took 170 msec on our benchmarking system in that case. By only processing actually executed methods, we could reduce this time and improve the startup performance.

For most benchmarks, 1% to 2% of the compiled traces must be recompiled because deoptimization is required too frequently. With nearly 13% recompilation, the benchmark *xml.validation* of the SPECjvm2008 benchmark suite is a worst-case example.

8.2.9 Discussion of Results

Our tracing configurations perform especially well on large applications such as the DaCapo 9.12 Bach benchmarks, where aggressive trace inlining results in large compilation units and better optimized machine code. On some of the benchmarks, our configuration *tracing peak performance* even outperforms the server compiler due to our aggressive trace inlining and optimizations that use the context-sensitive type information that is recorded in the traces. This is especially interesting as the server compiler is designed for best peak performance and implements significantly more optimizations than our trace-based compiler. Our results show that a fairly simple trace-based compiler can achieve an excellent peak performance by performing only basic traditional optimizations and aggressive trace inlining. Depending on the target platform, we also think that our configuration *tracing*

minimum code is interesting as it combines a good peak performance with little generated machine code and an excellent compilation time.

Our trace-based compiler does not do any loop optimizations such as loop unrolling or loop-invariant code motion, so that there is only a small performance gain for small and loop-intensive benchmarks. On those benchmarks, the server compiler is especially strong as it does perform sophisticated loop optimizations. However, this also shows that our trace-based compiler still has plenty of potential as it performs only basic traditional optimizations so far.

8.3 Code Coverage

We evaluated two configurations of our trace-based code coverage system in comparison to the state-of-the-art code coverage framework JaCoCo:

- The baseline is our unmodified trace-based JIT compiler and all results are normalized to the results of this configuration. In the figures, this configuration is only shown implicitly as the 100% mark.
- The configuration *JaCoCo* shows the performance of the JaCoCo [6] code coverage tool in the version 0.62 when used with our unmodified trace-based runtime system. We use JaCoCo's online bytecode instrumentation mechanism as this greatly simplifies instrumenting the benchmark suites which do a significant amount of class loading at run time.
- The configuration *trace-based coverage untuned* uses the trace-based compiler and our modified runtime system to record code coverage information. So, the trace recording interpreter records traces from the beginning. Eventually, the recorded traces are compiled to optimized machine code.
- The configuration *trace-based coverage tuned* also uses the trace-based compiler and our modified runtime system to record code coverage information. However, we reduced the compilation thresholds as described in Chapter 7.5 on page 68 so that the startup performance improves because traces are compiled earlier.

Figure 8.22 shows a comparison between the features of JaCoCo and our approach. One important difference between JaCoCo and our approach is that JaCoCo does not record code coverage information for the classes on the boot classpath, while our approach records code coverage information for all loaded classes. JaCoCo also excludes all classes defined in the rt.jar file such as `String` and `ArrayList` which are used frequently. Code coverage information for those classes is usually not particularly interesting but it can, for example, be used to identify the JDK parts that are used by an application. Besides that, we

tried to stress the code coverage instrumentation as much as possible so that we did not exclude any application classes from the collection of code coverage information. For many applications it would be possible to further reduce the amount of instrumentation done by JaCoCo by limiting code coverage to a minimum set of interesting classes.

	JaCoCo	our approach
instrumentation technique	online or offline bytecode instrumentation[1]	no additional instrumentation
line coverage	yes	yes
detects partially covered lines	yes	yes
edge coverage	yes	yes
path coverage	no	partially[2]
supports explicit exceptions	yes	yes
supports implicit exceptions	no	yes
records execution counts	no	partially[3]
can instrument all classes	excludes classes on the boot classpath	yes

[1] we use online instrumentation as it allows us to instrument the application transparently
[2] some information is lost for partial traces
[3] only recorded while code is executed in the interpreter

Figure 8.22: Features of the evaluated tools

8.3.1 SPECjvm2008

Figure 8.23 shows the performance results for the SPECjvm2008 benchmark suite. In this diagram, the startup and the peak performance for each benchmark category are shown on top of each other. The startup performance was measured by executing each benchmark for one benchmark operation in a new VM. Both runs are shown relative to the fastest run of the baseline.

In comparison to the unmodified trace-based compiler, both trace-based code coverage configurations succeed in preserving full peak performance. This is the case because the compiled machine code does not contain any instrumentation so that there is no overhead when executing the compiled code. In terms of startup performance, our configuration *trace-based coverage tuned* reaches the same performance level as the unmodified trace-based JIT compiler (not shown in the diagram). Tuning the compilation thresholds reduces the impact of trace recording on startup performance so that the startup performance of all benchmarks improves in comparison to the configuration *trace-based coverage untuned*.

When the bytecodes of the benchmarks are instrumented using JaCoCo, the added probes have a significant negative effect on the peak performance because they are compiled to machine code and also executed in compiled code. One exception is the benchmark *serial*,

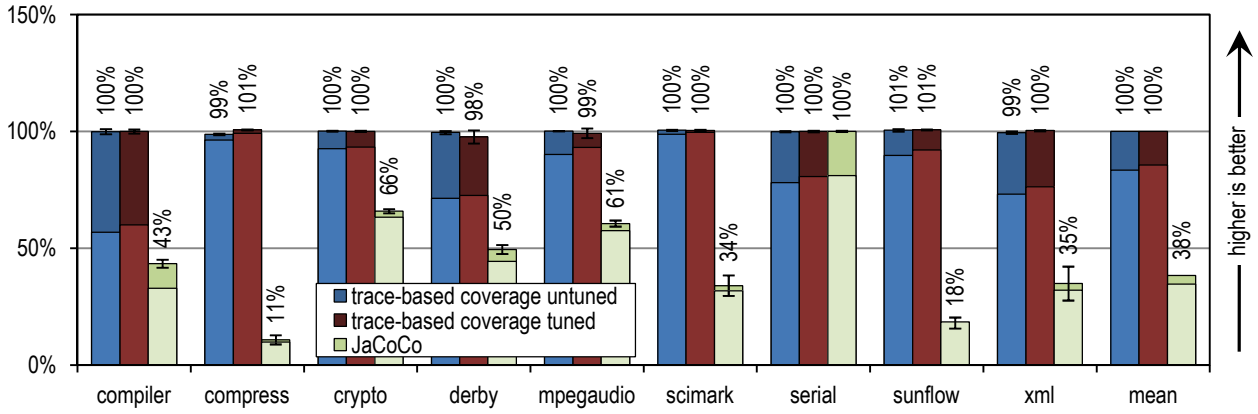


Figure 8.23: SPECjvm2008: startup and peak performance

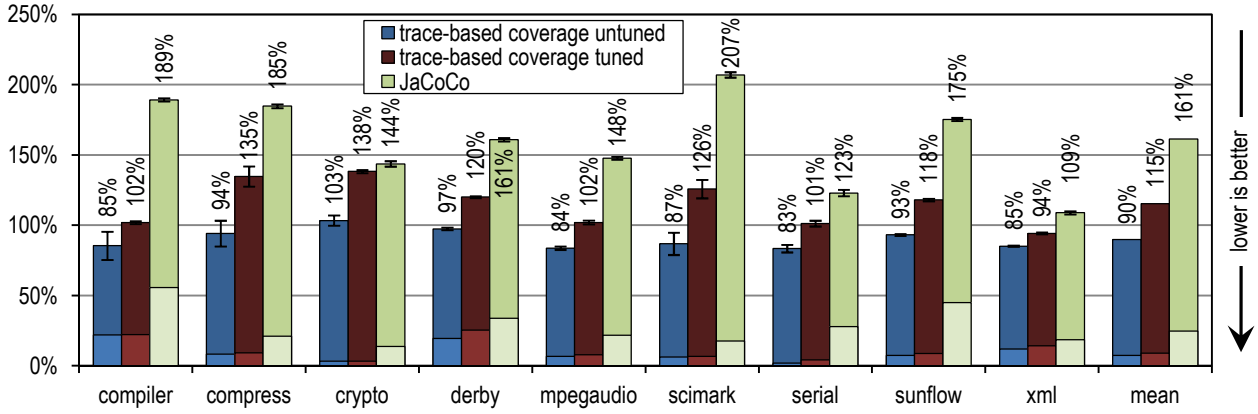


Figure 8.24: SPECjvm2008: amount of generated machine code

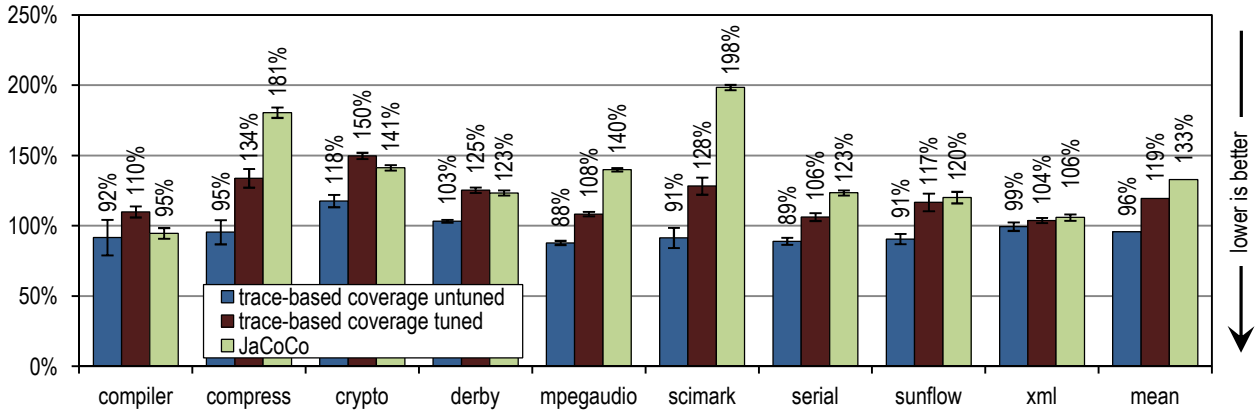


Figure 8.25: SPECjvm2008: time required for JIT compilation

which mainly executes JDK methods in its performance-critical code parts. JDK methods are not instrumented by JaCoCo and the probes within the actual benchmarking logic do not impact the peak performance of this benchmark.

The dark bars in Figure 8.24 show the total amount of generated machine code, while the light bars indicate the amount of machine code that was invalidated because optimistic optimizations deoptimized too frequently. The configuration *trace-based coverage untuned* generates less machine code than the unmodified trace-based compiler because it does record traces for a longer time before compiling them. So, the recorded trace information is more accurate and it is less likely that the compiled machine code deoptimizes and has to be invalidated and recompiled because of too frequent deoptimization. In contrast to that, the configuration *trace-based coverage tuned* generates more machine code than the unmodified trace-based compiler because of two factors. Traces are recorded fewer times so that the trace information is less accurate and deoptimization and invalidation of machine code occurs more frequently. Second, the SPECjvm2008 benchmarks are loop intensive and due to the lower compilation threshold for loops, more loops are compiled during startup. Some of this generated code becomes redundant later when the enclosing method traces are compiled, and the loop traces may get inlined.

When the benchmarks are instrumented with JaCoCo, the amount of generated machine code increases heavily because the instrumentation code is also compiled. A similar behavior can also be seen in Figure 8.25, which shows the amount of time required for JIT compilation. However, the increase in compilation time is not as significant because the instrumentation code is simple and hardly complicates the compilation.

8.3.2 SPECjbb2005

Figure 8.26 shows the peak performance, the generated machine code, and the compilation time for the SPECjbb2005 benchmark. Both trace-based code coverage configurations significantly outperform the case when JaCoCo is used to instrument the benchmark. However, the trace-based configurations also lose several percent in terms of peak performance because traces are now also recorded during startup so that the trace-based JIT compiler uses different trace information to guide its optimizations. The additional trace information that is recorded during startup, does not represent the warmed up benchmark behavior which has a negative impact on optimistic type-specific optimizations and trace inlining. So, the amount of generated machine code decreases due to the less aggressive trace inlining, while the compilation time increases because of the additional trace information that must be processed.

However, the instrumentation that is added by JaCoCo has a significantly higher impact on peak performance. The instrumentation hinders trace inlining because it increases the

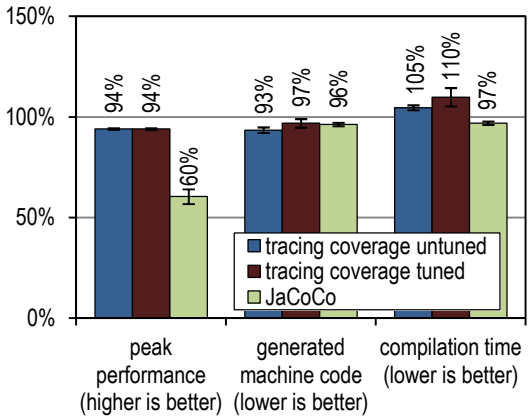


Figure 8.26: SPECjbb2005 results

size of the recorded traces and larger traces are less likely inlined. Furthermore, the instrumentation is also present in the generated machine code where it reduces performance.

8.3.3 DaCapo 9.12 Bach

Figure 8.27 shows the performance results for the DaCapo 9.12 Bach benchmark suite. In this diagram, the startup and the peak performance for each benchmark category are shown on top of each other. Both runs are shown relative to the fastest run of the baseline.

On average, both code coverage configurations that use our trace-based runtime system nearly achieve the same peak performance as the baseline. However, for a few benchmarks such as *avrora*, *jython*, and *luindex* our approach for recording code coverage information has some impact on peak performance. This happens because different parts of the execution are trace recorded due to the changed thresholds. So, the recorded trace information also covers code that is only required during startup. Those differences in the trace information may result in less effective compiler optimizations which reduces the peak performance. However, the JaCoCo instrumented run is still far slower despite not recording any coverage information for classes on the boot classpath. The main exception is the benchmark *tomcat* which spends most of its time in native functions where neither JaCoCo nor our approach places any instrumentation.

In terms of startup performance, the configuration *trace-based coverage tuned* reaches the same level as the unmodified trace-based JIT compiler (not shown in the diagram). The benchmark *tradesoap* is the only one that does not benefit from the startup performance tuning. Here, the reduced compilation threshold results in slightly more frequent deoptimization which reduces the startup performance again.

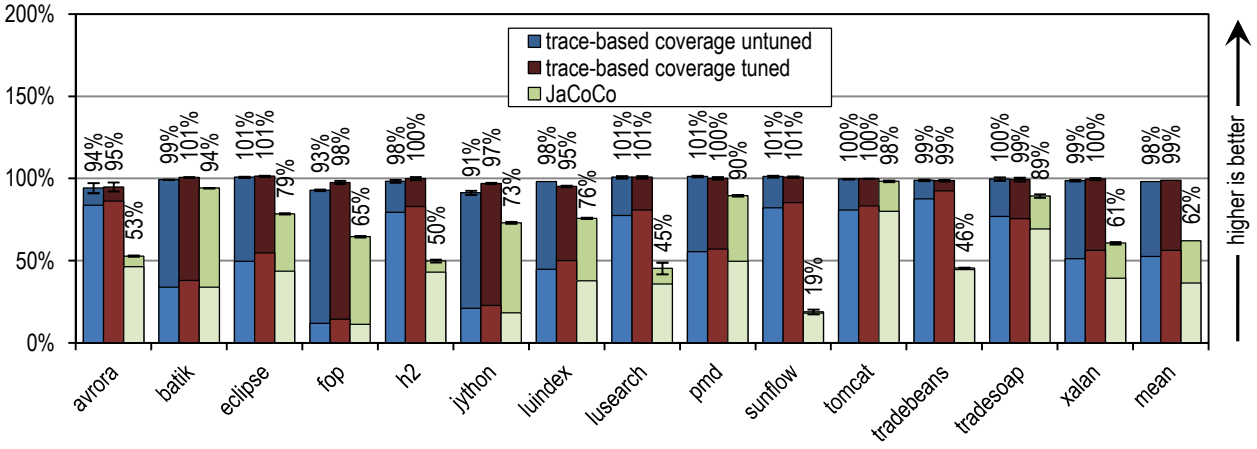


Figure 8.27: DaCapo 9.12 Bach: startup and peak performance

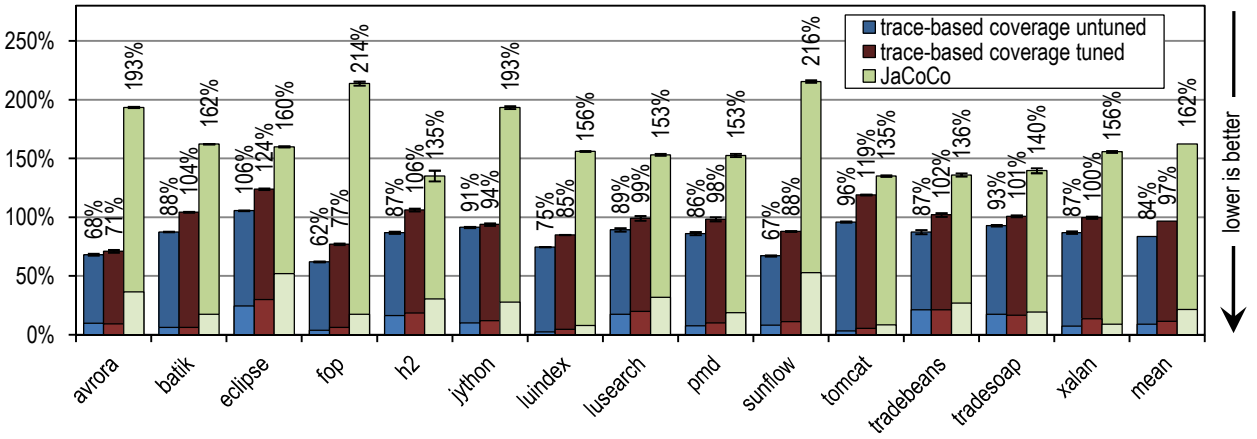


Figure 8.28: DaCapo 9.12 Bach: amount of generated machine code

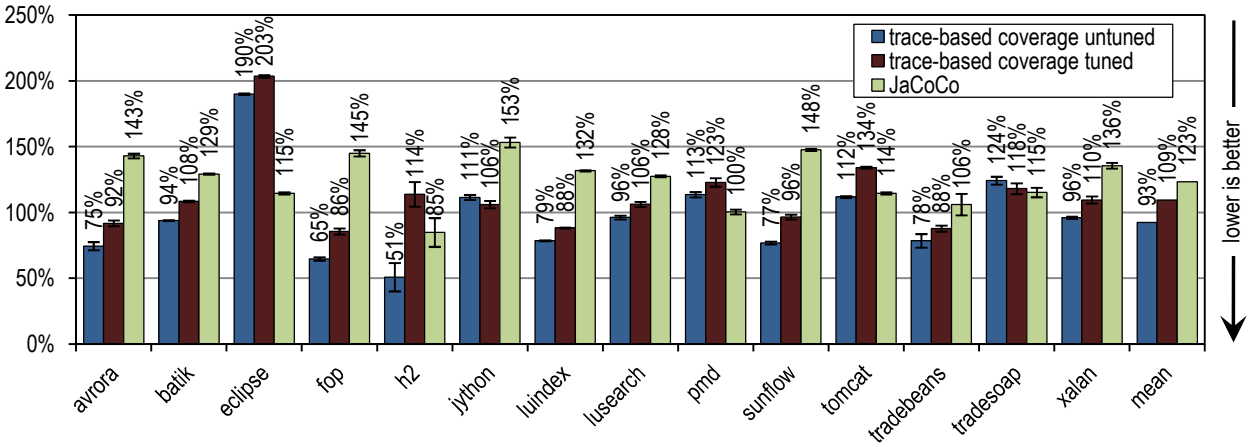


Figure 8.29: DaCapo 9.12 Bach: time required for JIT compilation

The dark bars in Figure 8.28 show the total amount of generated machine code, while the light bars indicate the amount of machine code that was invalidated because optimistic optimizations deoptimized too frequently. The configuration *trace-based coverage untuned* generates less machine code than the unmodified trace-based compiler because the recorded trace information is more accurate so that less code has to be invalidated and recompiled. By decreasing the threshold for JIT compilation, the configuration *trace-based coverage tuned* increases the startup performance but also generates more machine code. However, due to the more accurate trace information (traces are still recorded 3 to 4 times more often before compilation), deoptimization and recompilation of traces is required less frequently than in the unmodified trace-based JIT compiler. The configuration *JaCoCo* generates by far the largest amounts of machine code because the instrumentation code is also compiled.

Figure 8.29 shows the time required for JIT compilation. For most benchmarks, the instrumentation added by JaCoCo increases the time required for JIT compilation. Exceptions are the benchmarks *h2* and *tradesoap* where several trace anchors cannot be compiled because the instrumentation violates invariants assumed by HotSpot, see Chapter 7.4. Especially for *h2*, this also impacts the startup and peak performance. Figure 8.30 shows for every benchmark how many hot trace anchors are not compiled because the instrumentation added by JaCoCo violates some HotSpot invariant.

SPECjvm2008	not compiled trace anchors	compiled trace anchors
compiler	0	2508
compress	0	384
crypto	0	566
derby	39	1627
mpegaudio	0	530
scimark	0	350
serial	0	736
sunflow	0	619
xml	0	1451

DaCapo 9.12 Bach	not compiled trace anchors	compiled trace anchors
avrora	5	664
batik	0	1573
eclipse	50	5377
fop	2	1325
h2	25	1059
jython	0	1648
luindex	5	618
lusearch	3	571
pmd	0	1835
sunflow	0	454
tomcat	25	3227
tradebeans	32	2018
tradesoap	63	3839
xalan	6	1345

Figure 8.30: Hot trace anchors that are not compiled because of JaCoCo

Another interesting case can be observed for the benchmark *eclipse*. Both trace-based coverage configurations require significantly more time for JIT compilation than the unmodified trace-based runtime system. This happens because the different trace information

leads to different trace inlining decisions. Therefore, some compilation units get significantly larger, which increases the compilation time by a non-linear factor. However, it also results in slightly better optimized code so that the peak performance increases.

8.3.4 Memory Usage

Figure 8.31 shows some statistics about the recorded traces when the benchmark suites are executed with the trace-based baseline and the configuration *trace-based coverage tuned*. When code coverage information is being recorded, a significantly higher number of full traces (i.e. those that start at trace anchors) is being recorded. This also increases the amount of memory that is used by the traces but on average the traces still use only a few megabytes of memory. So, it should be possible to record code coverage for large server applications without a significant increase in memory usage.

		baseline	trace-based coverage tuned
total trace size in MB	SPECjvm2008	0,30	2,68
	DaCapo 9.12 Bach	1,17	4,51
	SPECjbb2005	0,23	2,13
number of full traces	SPECjvm2008	4052	31953
	DaCapo 9.12 Bach	12651	45351
	SPECjbb2005	2050	21669
number of partial traces	SPECjvm2008	449	242
	DaCapo 9.12 Bach	861	423
	SPECjbb2005	211	92

Figure 8.31: Trace recording statistics

The figure also illustrates that the DaCapo 9.12 Bach benchmark suite contains the most complex benchmarks. For the benchmark *eclipse* more than 100,000 traces are recorded with the baseline configuration, which occupy nearly 12 MB on the heap. When executing *eclipse* with our trace-based code coverage variant, the number of recorded traces increases to 270,000 which occupy 33 MB on the heap. This is by far the highest number measured for any of the benchmarks.

8.3.5 Discussion of Results

In comparison to existing code coverage frameworks such as JaCoCo, our approach does not add any instrumentation to the application. This avoids that the generated machine code contains instrumentation that would degrade performance and unnecessarily increase the amount of generated machine code as well as the time required for JIT compilation.

Due to the changed trace recording and compilation thresholds, our trace-based configurations record different trace information. This may change the optimizations used by the JIT compiler and can affect performance. However, our approach still has a very low performance and memory overhead so that it can also be used to record code coverage in daily operation.

We also validated our JaCoCo performance results, by executing the JaCoCo-instrumented benchmarks with an unmodified version of the HotSpot server compiler. The results were similar to those seen with our trace-based JIT compiler.

Chapter 9

Related Work

The main topic of this thesis is trace-based compilation and in the following we compare our work to other trace-based compilation approaches. Because trace inlining is our most profitable optimization, we also illustrate the commonalities and differences when compared to various method inlining strategies. Furthermore, we cover some code coverage literature and compare it to our approach that uses the recorded traces to derive exact code coverage information.

9.1 Trace-based Compilation

Bala et al. [9] pioneered trace compilation in their Dynamo system for dynamic and transparent optimization of native instruction streams. They used a software interpreter to execute binary applications and to identify hot instruction sequences during execution. For those identified traces, optimized code fragments are generated and directly executed. The interpretation overhead decreases with the number of compiled traces, resulting in a speedup eventually. In contrast to our trace compilation approach, the Dynamo system assumes that every backward branch is a possible loop header. This however leads to detection of false loops [42], which may affect the performance negatively. In contrast to Dynamo, we identify traces in Java bytecode instead of in a native instruction stream, and we perform a static analysis to detect loops in the bytecode. This simplifies trace recording and avoids problems with false loops. Furthermore, we limit individual traces to at most one method and at call sites we link the recorded traces to preserve context-sensitive trace information across method boundaries. This allows us to delay the inlining decision to the time of compilation instead of doing it already during trace recording.

Rogers [62] implemented a compilation variant for Java, where frequently executed basic blocks are detected and compiled. Related blocks, which may also span multiple methods, are grouped and optimized as an entity when executed frequently. In comparison to

method-based compilation, this approach compiles up to 18% fewer bytecodes. Our system records and compiles traces and uses trace inlining to increase the peak performance. When compiling the traces, we apply general and tracing-specific optimizations before generating machine code.

Trace-based compilation is also used for compiling dynamically typed languages such as ActionScript [17], JavaScript [30], or Python [14]. In those cases, the recorded traces contain information about the encountered types so that the trace-based compiler can perform type specialization. This results in high speedups as boxed types can be replaced by primitive types so that operations such as adding two integers can be replaced by a single hardware instruction instead of a method call. Our trace-based JIT compiler targets Java, which is a statically typed language so that most type information is already available at compile time. Therefore, similar optimizations are mostly not applicable for Java. However, we use the recorded type information to perform aggressive inlining of virtual calls.

The next approaches implemented different variants of trace-based compilation for Java [12, 31, 32, 47]. However, all approaches have in common that traces may span more than one method, so that inlining must be performed during trace recording. In contrast to that, we assume that one method is the maximum scope of a trace and use trace linking to preserve call information between traces. This allows delaying the inlining decision to the time of compilation when more information is available. So, our inlining can be more selective while using simple inlining heuristics that result in increased peak performance and a reduced amount of generated machine code.

Gal et al. [31, 32] implemented trace-based compilation for Java on resource-constrained devices. Traces start at frequently executed backward branch targets and side exits of existing traces. To improve the accuracy of loop detection, a backward branch is only considered as a loop header if its execution frequency exceeds a certain threshold. Each trace may span multiple methods so that inlining is performed during trace recording. If a Java exception is thrown during trace recording, they abort trace recording and invalidate the recorded traces. After recording a trace, it is immediately compiled to machine code. When a trace is left via a side exit during execution, the side exit is recorded in a separate trace and that trace is attached to its parent. This results in tree-like structures with explicit tail duplication and without merge points. While the trace tree simplifies many optimizations, it cannot be used for applications with a complex control flow because excessive tail duplication results in code bloat. Still, this approach is fairly popular and a similar concept is used by the Dalvik VM [15, 18, 57] on Android-based mobile devices. In contrast to that, we merge individual traces into a trace graph before compilation to avoid excessive tail duplication. This addresses the problems with tail duplication and allows us to handle also complex traces efficiently.

Bebenita et al. [12] implemented trace compilation for the meta-circular Maxine VM. Instead of using an interpreter, the Maxine VM uses a non-optimizing baseline JIT compiler to generate code that is used for the initial executions. This baseline JIT compiler was modified to generate instrumentation for trace recording. When a Java exception is thrown during trace recording, they abort and invalidate the recorded traces. To avoid unnecessary tail duplication, the recorded traces are merged into *trace regions* which have explicit control flow merge points. Those trace regions are then jitted using a SSA-based optimizing compiler. The evaluation is done using a subset of the DaCapo 2006 and SPECjvm2008 benchmarks and they achieved an excellent speedup for loop-intensive benchmarks due to various loop optimizations. However, their compiler performs worse than a method-based compiler on benchmarks with fewer loops. Our work is complementary as we focus on complex applications that are not loop-intensive such as DaCapo 9.12 Bach *jython*. We achieve excellent speedups for those applications, while achiving only small speedups on loop-intensive benchmarks, because our trace-based compiler does not perform any sophisticated loop optimizations yet.

Inoue et al. [47] added a trace-based JIT compiler to the IBM J9/TR JVM by modifying the method-based JIT compiler. Similar to the Dynamo system, trace recording focuses on linear and cyclic traces where no join points except the head of cyclic traces are present. To reduce the transition overhead between interpreted and compiled code, a code sequence is generated for every potential trace exit to ensure that the stack is compatible to the interpreter. Such code sequences are also required for every bytecode that might throw an exception. When the application throws an exception during trace recording, the trace recording stops and the trace is stored. This is slightly better than aborting trace recording but still does not support the case that the exception source and the exception handler are in the same compilation unit. The implemented trace-based JIT compiler does not support all optimizations of the normal method-based JIT compiler so that a method-based JIT compiler with a reduced set of optimizations was chosen as the baseline. On average, the trace-based JIT compiler achieves 96% of the baseline performance on the DaCapo 9.12 Bach benchmarks excluding the benchmark *tradesoap*. The benchmark *jython* showed the highest speedup with 26% but the benchmarks *tomcat* and *eclipse* were approximately 20% slower than the baseline. The size of the generated machine code depended highly on the specific benchmark and ranged from 52% smaller to 390% larger. Wu et al. [71] extended that work by avoiding short-lived traces and unnecessary trace duplication. This reduced both the amount of generated machine code and the compilation time, while peak performance was not affected. This work is the closest to ours as it does also build on an existing production quality method-based JIT compiler. In contrast to their work, we limit individual traces to span at most one method so that we can delay the inlining decision to the time of compilation. This increases the peak performance while reducing compilation time and the size of the generated machine code for most benchmarks. Furthermore, we

do not need to generate compensation code for every exception-throwing bytecode, as we rely on deoptimization instead. Our trace-based compiler sometimes even outperforms the Java HotSpot server compiler, although it performs significantly fewer optimizations.

9.2 Method Inlining

Method inlining is a well-researched topic that is extensively covered in literature. The remaining related work therefore concentrates on ways to inline *method parts* instead of whole methods as this is closest to our work. Still, these approaches are complementary to trace compilation as method parts are explicitly *excluded* there, while trace recording identifies method parts that should be compiled.

Partial method compilation [28, 68] uses profiling data to detect rarely executed method parts. Those parts are then excluded from compilation, so that less bytecodes are compiled to machine code. If code must be executed that was excluded from compilation, execution falls back to the interpreter or another version of the code is compiled on demand. The approach is evaluated on various applications and on the SPECjvm98 benchmark suite. The results show a reduced compilation time and a 10% increase of the startup performance on average. Our approach is even more selective as we record and compile only frequently executed traces. Because the recorded trace information is context-sensitive, we can avoid compiling method parts that were executed frequently in total, but are not required for the current caller. Furthermore, we use the saved compilation resources for aggressive trace inlining, which increases the peak performance.

Suganuma et al. [65, 66] implemented a region-based compiler where compilation heuristics and profiling data are used to exclude rarely executed method parts from compilation. Furthermore, method inlining is used heavily to group frequently executed code into a single compilation unit, a so called region. If a method part must be executed that was not compiled, the affected method is recompiled and on-stack-replaced so that the execution continues in the recompiled code. For SPECjvm98 and SPECjbb2000, this approach reduces the compilation time by more than 20% and increases the performance by 5% on average. Trace-based compilation does not explicitly exclude method parts from compilation but only compiles frequently executed traces that are identified using trace recording. The recorded trace information is context-sensitive so that it can be used for optimizations such as context-sensitive trace inlining and tail duplication.

Bradel et al. [16] analyzed the usage of traces for method inlining. An offline feedback-directed system was implemented for the Jikes RVM, which considers return instructions and backward branch targets as trace anchors. Then, hot call sites are identified within the recorded traces and this information is used to guide method inlining. Their evaluation

with the benchmarks SPECjvm98 and Java Grande shows a 10% performance increase, while 47% more machine code is generated. Our system records traces during execution in the interpreter and only compiles and inlines method parts covered by traces. This improves the performance and also reduces the size of the generated machine code significantly. Furthermore, we establish loops as top level compilation units so that they can be compiled and invoked independently from method traces.

Hazelwood et al. [43] implemented context-sensitive inlining for the method-based Jikes RVM compiler. Timer-based sampling and recording of call information are used to gather profiling data to guide inlining decisions during compilation. For the benchmark suites SPECjvm98 and SPECjbb2000, the size of the generated machine code and the compilation time could be reduced by 10%, without affecting the performance. We record traces in a call-graph-like data structure, which gives us even more detailed context-sensitive information. Depending on the inlining heuristic, this either increases peak performance or reduces the amount of generated machine code significantly.

Trace compilation is also suitable for compiling dynamic scripting languages as shown by Bebenita et al. [11], Bolz et al. [14], Chang et al. [17], or Gal et al. [30]. In such an environment, the recorded type information can be used to perform type specialization. This may result in high speedups as boxed types can be replaced by machine-specific types such as integer or double. Java is a statically typed language so that exact type information is available at compile time. Therefore, similar type specialization optimizations are mostly not applicable.

Another major difference between our and related work is the way how we record and inline traces. Other approaches allow traces to span more than one method, so that inlining must be performed during trace recording. We assume that one method is the maximum scope of a trace and use trace linking to preserve call information between traces. This allows delaying the inlining decision to the time of JIT compilation when more information is available so that the inlining can be performed more selectively.

9.3 Code Coverage

The approaches related to our work can be roughly divided into two categories. The first category tries to place instrumentation more selectively so that fewer probes are required. The second category removes instrumentation as soon as it is no longer needed.

9.3.1 Selective Instrumentation

Ball et al. [10] propose an efficient algorithm for path profiling. They add instrumentation at certain positions in a method so that the instrumentation computes a path identifier when executed. This path identifier exactly identifies the executed path. Our trace recording interpreter records control flow decisions and other information such as observed types while the application is being executed. This additional information is valuable for aggressive and optimistic compiler optimizations and helps us to increase the peak performance.

Agrawal [7, 8] does a variant of dominator computation to identify leaf blocks that must be instrumented to record code coverage information. Instrumenting only leaf blocks greatly reduces the amount of instrumentation. However, in case of exceptional control flow, this technique is less efficient because it requires every potentially exception throwing instruction to be instrumented. Otherwise, an exception could leave the current block or method without marking already executed code as covered. Our approach is complementary as we do not add any instrumentation to record code coverage. Instead, we use the already existing tracing infrastructure in the interpreter that records profiling data, in our case traces. This avoids any instrumentation in the compiled code and thus minimizes the impact on peak performance.

9.3.2 Disposable Instrumentation

Another approach that is related to ours is to reduce the instrumentation overhead by removing the instrumentation as soon as possible. Pavlopoulou et al. [61] instrument the bytecodes of Java applications to record coverage information. Then, the instrumented application is executed multiple times with different parameters. Whenever the application finishes running, all executed and therefore no longer needed probes are removed from its class files. This reduces the amount of instrumentation with every execution.

Tikir et al. [67] combine selective instrumentation and removal of no longer needed instrumentation to decrease the overhead of determining code coverage. They dynamically instrument binary code while it is executed to record code coverage information and - similar to Agrawal - they use dominator tree information to reduce the number of inserted probes. Their system periodically checks if there are any probes that are no longer needed and can be removed. So, the amount of instrumentation is gradually reduced without stopping the executed application.

Kumar et al. [51, 52] developed a framework for dynamic instrumentation of binary code. They also adapted their tool for the Jikes RVM so that Java applications can be instrumented on the machine code level. The machine code of the Java application is generated

by the Jikes JIT compiler which keeps track of which machine code instructions map to which bytecodes. Using this information, they can map positions within the machine code to bytecode, and also to source code if the class file contains a line number table. Probes can be added and removed at run time and are implemented as fast breakpoints [49] that branch to the instrumentation code.

Chilakamarri et al. [19, 20] use bytecode instrumentation to record code coverage information. They modified the Kaffe JVM interpreter in such a way that the instrumentation bytecodes are overwritten with `nop` instructions once they have been executed. This effectively removes the instrumentation immediately after execution and increases the performance. However, when the Kaffe JIT compiler is used, they only support method coverage which is simple to implement.

Similarly to Tikir et al, Li et al. [53] combine selective instrumentation and removal of no longer needed instrumentation to decrease the overhead of code coverage instrumentation. They propose to do an offline analysis step with their *super nested block* algorithm, which aims at reducing the number of required probes and is similar to the dominator-based approach of Agrawal. The results of this offline analysis is then used when instrumenting the application at run time, so that probes do not need to be added to every basic block. After a probe was executed, they remove it in a similar way as Tikir et al.

Dmitriev [25, 26] uses the hotswapping API of the HotSpot JVM to dynamically instrument Java applications. The hotswapping API provides mechanisms to replace methods and classes with different versions at run time. This approach can be used to add instrumentation to an application but it can also be used to remove no longer needed instrumentation. Hotswapping has a certain run-time overhead but when focusing on peak performance, the possibility to remove the instrumentation outweighs this overhead.

Forax [29] uses bytecode instrumentation to insert code coverage probes into Java applications. The probes are modeled as Java 7 *invokedynamic* bytecodes. When an invokedynamic bytecode is executed for the first time, the VM invokes a developer-defined bootstrapping method that resolves the target method. For subsequent executions, the target method is cached so that it can be invoked directly. Forax exploits this behavior and updates the code coverage information in the bootstrapping method and returns a reference to a method that does not perform any operations. This ensures that the code coverage information is updated properly, while the run-time overhead is kept small because subsequent executions invoke an empty method.

Our approach is different from all those approaches as we avoid the explicit instrumentation of applications by reusing the already existing tracing infrastructure of our trace recording interpreter. Because we do not add any instrumentation, we also do not have to take care of removing it later. Instead, hot code parts are compiled by the JIT compiler and

the compiled code never contains any instrumentation. Thus, our approach gives us code coverage information while avoiding the run-time overhead of instrumentation in compiled code.

Chapter 10

Summary

The project started in July 2010 and was funded by the Austrian Science Fund (FWF) until September 2013. From January to March 2012, the project was paused because the author of this thesis did an internship at Oracle Labs.

During the first few month of the project, we concentrated on different trace recording techniques and eventually developed the trace recording approach described in this thesis. This approach has the significant advantage of delaying the inlining decisions to the time of JIT compilation. During the main project part, we modified and extended the Java HotSpot client compiler to support trace-based compilation. The resulting first prototype was only able to compile traces that started at method entries. Loop traces always had to be inlined. Later on, we addressed this limitation and defined suitable calling conventions for invoking compiled loop traces. Then, we added further optimizations to the trace-based JIT compiler and experimented with trace inlining to increase the peak performance. Towards the end of the project, we concentrated on additional topics such as the approach to derive code coverage from the recorded traces.

10.1 Future Work

Both in trace-based compilation and in code coverage computation, the following further improvements would be possible.

10.1.1 Trace-based Compilation

When the project started, we decided to choose the Java HotSpot client compiler as the basis for our trace-based JIT compiler because the client compiler is far simpler and therefore more approachable to the changes that were necessary. However, the HotSpot server compiler performs significantly more optimizations so that it would be interesting to see

how those optimizations are affected with the increased compilation scope that is achieved due to the context-sensitive trace information. The server compiler also has a more powerful intermediate representation which might simplify some trace-specific optimizations. Transforming the server compiler into a trace-based JIT compiler would be a significant effort.

A more feasible approach would be to stay with the concept of method-based compilation but replace the method-based profiling data with the context-sensitive trace information. This would result in a method-based compiler that only compiles parts that are covered by traces. The recorded context-sensitive trace information is more accurate than the method-based profiling data so that method inlining could be replaced with our effective trace inlining technique. Loops would always be compiled with their enclosing method so that it is unnecessary to establish loops as top-level compilation units. While this would reduce the flexibility to choose certain compilation units, it would greatly simplify the implementation as the trace transitioning is one of the most complex parts of a trace-based compiler. The resulting hybrid between a method-based and a trace-based compiler should be able to combine the best parts of both worlds.

10.1.2 Code Coverage

Our approach should be applicable to most of today's high-performance VMs that have an aggressive JIT compiler and an already instrumented interpreter or baseline compiler that records profiling data. We decided to implement our approach for a trace-based JIT compiler instead of for a method-based compiler, because the recorded traces are path-based and contain more information than the profiling data recorded for a method-based compiler. The next logic step would be an implementation of our approach for Oracle's HotSpot server compiler, which is one of the most used Java JIT compilers. For this, the following steps would be necessary:

- The HotSpot interpreter is already instrumented to record profiling data for the server compiler. However, to maximize the startup performance, profiling data is only recorded after a method was executed a certain number of times. This threshold would have to be set to zero so that the profiling data is recorded from the beginning.

- In most cases, the server compiler already avoids compiling unexecuted code and unexecuted control flow edges. However, in a few cases such as when on-stack-replacement (OSR) [46] is used to optimize a long running loop, unexecuted code may get compiled. This would have to be changed so that only executed Java bytecode is compiled to machine code.

Another important field, which we hardly spent any time on, is visualizing the coverage information. All commonly used code coverage tools visualize the code coverage in a user friendly and intuitive way. While we prototypically implemented a basic visualization to validate the recorded data, it still lacks most convenience features.

10.2 Conclusions

We presented a trace recording mechanism and a trace-based JIT compiler for Java that is integrated into Oracle's production quality Java HotSpot VM. Traces have the advantage that they cover only the executed method parts. The runtime system supports traces that start at method entries or at loop headers and we employ efficient calling conventions for invoking traces from interpreted and from compiled code. Furthermore, our trace-based compiler can handle exceptional control flow efficiently, which is an important factor to achieve good peak performance for complex Java applications.

To avoid false loops, we detect loop headers in the Java bytecodes using static analysis during class loading. Our trace recording approach restricts individual traces to at most one method, while preserving context-sensitive information by linking caller and callee traces. Furthermore, we delay inlining decisions from trace recording to the time of compilation as more information is available at that time. This allows more selective trace inlining. Prior to compilation, we merge the recorded traces into a trace graph to avoid unnecessary tail duplication. During JIT compilation, we apply general and tracing-specific optimizations such as constant folding and trace inlining. The recorded traces are context-sensitive so that we can inline different method parts depending on the specific call site. This allows aggressive trace inlining while generating reasonable amounts of machine code. To reduce the amount of generated machine code, we avoid generating compensation code for side exits but rather rely on deoptimization. Furthermore, we eliminate infrequently executed traces before compilation to ensure that only the most frequently executed traces are compiled to machine code.

Compared to the Java HotSpot client compiler we achieve up to 59% speedup (20% on average) for complex benchmarks such as DaCapo 9.12 Bach *jython* and on some of the benchmarks we even outperform the Java HotSpot server compiler. Furthermore, we also showed that trace inlining achieves larger compilation scopes that increase the effectiveness of common compiler optimizations and eventually result in a better peak performance. These results are promising as we show that a fairly simple trace-based compiler, that uses only basic traditional optimizations, can achieve an excellent peak performance that sometimes even outperforms one of today's best optimizing JIT compilers for Java.

We also presented a novel runtime system which allows us to obtain code coverage information without explicitly instrumenting an application. Instead, we derive exact code coverage information from the profiling data that is already recorded for an optimizing JIT compiler. This gives us coverage information almost for free while avoiding instrumentation in the compiled code parts and thus keeping the run-time overhead to a minimum. While we implemented our approach for a variant of the HotSpot VM that uses a trace-based JIT compiler, our approach is general enough to be also applicable to most other modern VMs. Measurements showed that our approach hardly affects the performance of applications so that it can also be used in daily operation systems.

List of Figures

Bibliography

[1] EMMA: a free Java code coverage tool, 2005. `http://emma.sourceforge.net/`.

[2] Cobertura, 2010. `http://cobertura.sourceforge.net/`.

[3] CodeCover, 2011. `http://codecover.org/`.

[4] Clover, 2013. Atlassian, Inc. `http://www.atlassian.com/software/clover/`.

[5] EclEmma, 2013. `http://www.eclemma.org/`.

[6] JaCoCo, 2013. `http://www.eclemma.org/jacoco/`.

[7] Hira Agrawal. Efficient Coverage Testing Using Global Dominator Graphs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 11–20. ACM Press, 1999.

[8] Hiralal Agrawal. Dominators, Super Blocks, and Program Coverage. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–34. ACM Press, 1994.

[9] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12. ACM Press, 2000.

[10] Thomas Ball and James R. Larus. Efficient Path Profiling. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, pages 46–57. IEEE Computer Society, 1996.

[11] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. SPUR: A Trace-Based JIT Compiler for CIL. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 708–725. ACM Press, 2010.

[12] Michael Bebenita, Mason Chang, Gregor Wagner, Andreas Gal, Christian Wimmer, and Michael Franz. Trace-Based Compilation in Execution Environments without Interpreters. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 59–68. ACM Press, 2010.

[13] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190. ACM Press, 2006.

[14] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In *Proceedings of the Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM Press, 2009.

[15] Dan Bornstein. Dalvik VM Internals. Presented at the Google I/O developer conference, 2008. `http://sites.google.com/site/io/dalvik-vm-internals`.

[16] Borys J. Bradel and Tarek S. Abdelrahman. The Use of Traces for Inlining in Java Programs. In *Proceedings of the International Conference on Languages and Compilers for High Performance Computing*, pages 179–193. Springer-Verlag, 2005.

[17] Mason Chang, Edwin Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Tracing for Web 3.0: Trace Compilation for the Next Generation Web Applications. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 71–80. ACM Press, 2009.

[18] Ben Cheng and Bill Buzbee. A JIT Compiler for Android's Dalvik VM. Presented at the Google I/O developer conference, 2010. `https://www.youtube.com/watch?v=Ls0tM-c4Vfo`.

[19] Kalyan-Ram Chilakamarri and Sebastian Elbaum. Reducing Coverage Collection Overhead With Disposable Instrumentation. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 233–244. IEEE Computer Society, 2004.

[20] Kalyan-Ram Chilakamarri and Sebastian Elbaum. Leveraging Disposable Instrumentation to Reduce Coverage Collection Overhead. *Software Testing, Verification & Reliability*, 16(4):267–288, 2006.

[21] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape Analysis for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 1–19. ACM Press, 1999.

[22] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[23] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 77–101. Springer-Verlag, 1995.

[24] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-First Garbage Collection. In *Proceedings of the International Symposium on Memory Management*, pages 37–48. ACM Press, 2004.

[25] Mikhail Dmitriev. Application of the HotSwap Technology to Advanced Profiling. Technical report, Sun Microsystems Laboratories, USA, 2002.

[26] Mikhail Dmitriev. Profiling Java Applications Using Code Hotswapping and Dynamic Call Graph Revelation. In *Proceedings of the International Workshop on Software and Performance*, pages 139–150. ACM Press, 2004.

[27] Evelyn Duesterwald and Vasanth Bala. Software Profiling for Hot Path Prediction: Less is More. *SIGPLAN Notices*, 35:202–211, 2000.

[28] S.J. Fink and Feng Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 241–252. IEEE Computer Society, 2003.

[29] Rémi Forax. JSR 292 Goodness: Fast code coverage tool in less than 10k, 2011. `https://www.java.net/blog/forax/archive/2011/02/12/jsr-292-goodness-fast-code-coverage-tool-less-10k`.

[30] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based Just-in-Time Type Specialization for Dynamic Languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 465–478. ACM Press, 2009.

[31] Andreas Gal and Michael Franz. Incremental Dynamic Code Generation with Trace Trees. Technical report, Donald Bren School of Information and Computer Science, University of California, Irvine, USA, 2006.

[32] Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: An Effective JIT Compiler for Resource-constrained Devices. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 144–153. ACM Press, 2006.

[33] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering, 2nd Edition.* Prentice Hall, 2003.

[34] Google. V8 JavaScript Engine, 2013. `https://code.google.com/p/v8`.

[35] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java® Language Specification, Java SE 7 Edition.* Addison-Wesley, 2013.

[36] Robert Griesemer. Generation of Virtual Machine Code at Startup. In *OOPSLA Workshop on Simplicity, Performance, and Portability in Virtual Machine Design.* Sun Microsystems, Inc., 1999.

[37] Christian Häubl and Hanspeter Mössenböck. Trace-based Compilation for the Java HotSpot Virtual Machine. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 129–138. ACM Press, 2011.

[38] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. Evaluation of Trace Inlining Heuristics for Java. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1871–1876. ACM Press, 2012.

[39] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. Context-sensitive Trace Inlining for Java. *Computer Languages, Systems and Structures*, 39:123–141, 2013.

[40] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. Deriving Code Coverage Information from Profiling Data Recorded for a Trace-based Just-in-time Compiler. In *Proceedings of the International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 1–12. ACM Press, 2013.

[41] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. Trace Transitioning and Exception Handling in a Trace-based JIT Compiler for Java. *ACM Transactions on Architecture and Code Optimization*, 2013. Accepted for publication.

[42] Hiroshige Hayashizaki, Peng Wu, Hiroshi Inoue, Mauricio J. Serrano, and Toshio Nakatani. Improving the Performance of Trace-based Systems by False Loop Filtering. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 405–418. ACM Press, 2011.

[43] Kim Hazelwood and David Grove. Adaptive Online Context-Sensitive Inlining. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 253–264. IEEE Computer Society, 2003.

[44] David Hiniker, Kim Hazelwood, and Michael D. Smith. Improving Region Selection in Dynamic Optimization Systems. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 141–154. IEEE Computer Society, 2005.

[45] Urs Hölzle, Craig Chambers, and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–43. ACM Press, 1992.

[46] Urs Hölzle and David Ungar. Optimizing Dynamically-dispatched Calls With Runtime Type Feedback. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 326–336. ACM Press, 1994.

[47] Hiroshi Inoue, Hiroshige Hayashizaki, Peng Wu, and Toshio Nakatani. A Trace-based Java JIT Compiler Retrofitted from a Method-based Compiler. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 246–256. IEEE Computer Society, 2011.

[48] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* Chapman & Hall/CRC, 2011.

[49] Peter B. Kessler. Fast Breakpoints: Design and Implementation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 78–84. ACM Press, 1990.

[50] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot Client Compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1):7:1–7:32, 2008.

[51] Naveen Kumar, Jonathan Misurda, Bruce R. Childers, and Mary Lou Soffa. FIST: A Framework for Instrumentation in Software Dynamic Translators. Technical report, University of Pittsburgh, USA, 2003.

[52] Naveen Kumar, Jonathan Misurda, Bruce R. Childers, and Mary Lou Soffa. Instrumentation in Software Dynamic Translators for Self-Managed Systems. In *Proceedings of the ACM SIGSOFT Workshop on Self-Managed Systems*, pages 90–94. ACM Press, 2004.

[53] J. Jenny Li, David M. Weiss, and Howell Yee. An Automatically-Generated Run-Time Instrumenter to Reduce Coverage Testing Overhead. In *Proceedings of the International Workshop on Automation of Software Test*, pages 49–56. ACM Press, 2008.

[54] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java® Virtual Machine Specification, Java SE 7 Edition.* Addison-Wesley, 2013.

[55] Sun Microsystems. Memory Management in the Java HotSpot™ Virtual Machine, 2006. `http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf`.

[56] Mozilla. IonMonkey, 2013. `https://wiki.mozilla.org/IonMonkey/Overview`.

[57] Hyeong-Seok Oh, Beom-Jun Kim, Hyung-Kyu Choi, and Soo-Mook Moon. Evaluation of Android Dalvik Virtual Machine. In *Proceedings of the International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 115–124. ACM Press, 2012.

[58] Oracle Corporation. *JVM Tool Interface*, 2007. `http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html`.

[59] Oracle Corporation. *Java Platform, Standard Edition 8 Developer Preview Releases*, 2013. `http://jdk8.java.net/download.html`.

[60] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot Server Compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, pages 1–12. USENIX, 2001.

[61] Christina Pavlopoulou and Michal Young. Residual Test Coverage Monitoring. In *Proceedings of the International Conference on Software Engineering*, pages 277–284. ACM Press, 1999.

[62] Ian Rogers. *Optimising Java Programs Through Basic Block Dynamic Compilation.* PhD thesis, Department of Computer Science, University of Manchester, 2002.

[63] Standard Performance Evaluation Corporation. *The SPECjbb2005 Benchmark*, 2005. `http://www.spec.org/jbb2005/`.

[64] Standard Performance Evaluation Corporation. *The SPECjvm2008 Benchmarks*, 2008. `http://www.spec.org/jvm2008/`.

[65] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. A Region-Based Compilation Technique for a Java Just-In-Time Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 312–323. ACM Press, 2003.

[66] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. A Region-Based Compilation Technique for Dynamic Compilers. *ACM Transactions on Programming Languages and Systems*, 28:134–174, 2006.

[67] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Efficient Instrumentation for Code Coverage Testing. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 86–96. ACM Press, 2002.

[68] John Whaley. Partial Method Compilation using Dynamic Profile Information. *SIGPLAN Notices*, 36:166–179, 2001.

[69] Christian Wimmer, Marcelo S. Cintra, Michael Bebenita, Mason Chang, Andreas Gal, and Michael Franz. Phase Detection using Trace Compilation. In *Proceedings of the International Conference on Principles and Practice of Programming in Java*, pages 172–181. ACM Press, 2009.

[70] Christian Wimmer and Hanspeter Mössenböck. Optimized Interval Splitting in a Linear Scan Register Allocator. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, pages 132–141. ACM Press, 2005.

[71] Peng Wu, Hiroshige Hayashizaki, Hiroshi Inoue, and Toshio Nakatani. Reducing Trace Selection Footprint for Large-scale Java Applications without Performance Loss. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 789–804. ACM Press, 2011.

[72] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array Bounds Check Elimination for the Java HotSpot™ Client Compiler. In *Proceedings of the International Symposium on Principles and Practice of Programming in Java*, pages 125–133. ACM Press, 2007.

All URLs were last accessed on 14-December-2014