UNIVERSITY OF CALIFORNIA,
IRVINE

Efficient Bytecode Verification and Compilation in a Virtual Machine

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Andreas Gal

Dissertation Committee:
Professor Michael Franz, Chair
Professor Nikil Dutt
Professor Stephen Jenks

2006

The dissertation of Andreas Gal
is approved and is acceptable in quality
and form for publication on microfilm:

Committee Chair

University of California, Irvine
2006

Dedicated to the memory of

Dr. Imre Gál.

TABLE OF CONTENTS

LIST OF FIGURES

Acknowledgements

Professor Franz supervised the research work described in this dissertation. I feel deeply indebted to Professor Franz for the privilege of being allowed to work with him. For the past 5 years he was not only my advisor, but also my mentor and role model. I admire his vision, his dedication to his work and his unconditional support for his graduate students, and I strive to apply the same standards when working with students myself.

Professor Schröder-Preikschat was my advisor at the University of Magdeburg. His research philosophy deeply influenced the way I think about and approach research problems. Professor Schröder-Preikschat helped me get accepted to UCI, and even after I left his research group, he continued to support me and helped me deal with the uncertainties and difficulties of pursuing a doctorate degree.

Professor Goulet initiated the dual degree program of the University of Wisconsin, Stevens Point, which I participated in, and he also advised me during my year at UWSP. Based on Professor Goulet's recommendation and support I decided to pursue a graduate degree in the US. I have no doubt that without him this dissertation would not exist.

Over the years I have worked with many colleagues, and I am thankful for their ideas and feedback. In particular, I would like to thank my two close collaborators Olaf Spinczyk and Christian Probst. With Olaf I wrote my first paper at the University of Magdeburg, and his research and writing style are still visible in my papers today. Christian Probst has been my most important co-author and collaborator over the past years. I greatly benefited from having the opportunity to work with him, and I also greatly enjoyed it. Much of the work in this thesis was joint work with Christian.

Last but not least, I would like to thank my family and friends for their constant support, friendship, love, patience, and sacrifices.

This research effort was partially funded by the National Science Foundation (NSF) under grants CCR-0105710, CNS-0205712 and CNS-0615443, by the Office of Naval Research (ONR) under agreement N00014-01-1-0854, by the Air Force Office of Scientific Research (AFOSR) under agreement FA8750-05-2-0216, and by Microsoft Research and the University of California under grant 04-032 as part of the University of California Microelectronics Innovation and Computer Research Opportunities (UC MICRO) program. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science foundation (NSF), the Office of Naval Research (ONR), or any other agency of the U.S. Government.

CURRICULUM VITAE

Andreas Gal

Date & Place of Birth

May 22nd, 1976 (Szeged, Hungary)

Education

1996–2000	Computer Science Department University of Magdeburg, Germany Department of Mathemics and Computing University of Wisconsin, Stevens Point (Dual Degree Program)	B.Sc. (cum laude)
2000–2001	Computer Science Department University of Magdeburg, Germany	M.Sc. (summa cum laude)
2001–2006	Department of Computer Science University of California, Irvine	Ph.D.

Awards

1999–2000	Dual Degree Fellowship, German Academic Exchange Office (DAAD)
1999–2000	Dean's List, University of Wisconsin, Stevens Point
2000	B.Sc. cum laude, University of Wisconsin, Stevens Point
2001	M.Sc. summa cum laude, University of Magdeburg, Germany

Abstract of the Dissertation

Efficient Bytecode Verification and Compilation in a Virtual Machine

By

Andreas Gal

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2006

Professor Michael Franz, Chair

Applications written in modern dynamic languages such as Java or .NET are shipped in the form of high-level intermediate bytecode. A virtual machine (VM) is used on the target machine to verify and execute the code. Many VMs use just-in-time compilation to speed-up the execution of critical code areas for which interpretation alone is not efficient enough.

In this dissertation, we explore existing verification and dynamic compilation techniques. These traditional approaches do not work well in resource-constrained environments such as PDAs and cell phones. We propose alternative verification and compilation techniques that were specifically designed for efficient runtime code generation. Our prototype virtual machine can generate code 350 times faster than existing desktop VMs, and is 30 times more memory efficient, yet achieves execution performance that is much higher than that of existing embedded VMs and closer to the performance of heavyweight desktop systems.

CHAPTER 1

INTRODUCTION

Applications written in modern dynamic languages such as Java or .NET are shipped in the form of high-level intermediate bytecode. This offers two distinct advantages over shipping programs directly as compiled machine code. On the one hand the bytecode is architecture independent and can be executed on different target systems using different native instruction sets and software frameworks (i.e. operating systems). This makes bytecode programs *portable* across target platforms. Since interpretation is often too inefficient for frequently executed ("hot") code areas, dynamic compilation is used to translate parts of the bytecode program to directly executable native machine code.

On the other hand, high-level intermediate bytecode can also be *verified* with respect to certain safety properties by the code consumer. The Java Virtual Machine Language (JVML) specification, for example, requires the code receiver to check that code is well typed before it is permitted to execute on this virtual machine. This verification step provides certain safety guarantees and eliminates common security concerns such as buffer overflows by ensuring the type and memory safety of all programs prior to execution.

Bytecode *verification* and *dynamic compilation* have been studied extensively from a correctness perspective, and a large body of prior work exists that discusses how to ensure that bytecode verification is actually safe, and dynamic compilation generates correct and efficient native machine code for a bytecode fragment. These traditional approaches do not work well in resource-constrained environments such as PDAs and cell phones. *It is not sufficient to verify code and to compile it to efficient machine code. Both bytecode verification and compilation also have to be efficient themselves.*

1.1 Dissertation Outline

The remainder of this dissertation is organized as follows. In Chapter 2 we introduce the Java Bytecode Verification algorithm, and show that the worst-case complexity of this algorithm for certain pathological bytecode arrangements is significant. We conclude that it is not sufficient for a bytecode verification algorithm to deliver the correct result. Instead, it has must be efficient, in particular in environments with limited resources such as mobile devices, cell phones, and PDAs. Based on this observation, in Chapter 3 we develop a more efficient bytecode verification algorithm that uses the Static Single Assignment (SSA) form instead of the traditional iterative data-flow analysis. Chapter 4 extends the idea of providing novel more-efficient algorithms for implementing virtual machines to the dynamic compilation and code generation pipeline. We present a just-in-time approach that operates on the level of code traces instead of compiling entire methods. Related work is discussed in Chapter 5. The dissertation ends with conclusions and an outlook on future work in Chapter 6.

1.2 Relevant Academic Publications

Our analysis of the complexity of Java Bytecode Verification (Chapter 2) was presented at the 2005 New Security Paradigms Workshop (NSPW 2005) [PGF05] in Lake Arrowhead, CA. The subsequent work on an improved verification algorithm which we describe in Chapter 3 was presented at the First International Workshop on Abstract Interpretation (AIOOL 2005) [GPF05a] and the 4th International Workshop on Compiler Optimization Meets Compiler Verification (COCV 2005) [GPF05b] in Edinburg, Scotland, and has been accepted (with revisions) for publication in the ACM Transactions on Programming Languages and Systems (TOPLAS). Our work on trace-based compilation (Chapter 4) was presented at the 2nd International Conference on Virtual Execution Environments (VEE '06) [GPF06a] in Ottawa, Canada. A paper describing our work on trace-trees and their compilation was submitted to the 2007 Conference on Programming Language Design and Implementation (PLDI 2007) and is currently under review.

CHAPTER 2

THE COMPLEXITY OF JAVA BYTECODE VERIFICATION

Java bytecode verification has been studied extensively from a correctness perspective, and several vulnerabilities have been found and eliminated in this process. However, there are avenues to an attack that do not depend on correctness at all. In this chapter we show how to construct Java code that is *correct*, but that requires an excessive amount of time for verification. We explain how this could be exploited for denial-of-service attacks on JVM-based services and devices.

In contrast to previously discovered flaws in the bytecode verifier, the enabling property for our exploit lies in the verification algorithm itself, and not in its implementation. As a consequence, it is much more difficult to correct. [GPF03, PGF05]

2.1 MOTIVATION

The bytecode verifier is an integral component of the Java Virtual Machine (JVM) [LY99] and is based on the idea of data-flow analysis (DFA) [Qia00]. The abstractions of values and their types are tracked along the edges of the control-flow graph and the verifier checks that no rules of the type system are violated.

The verifier is a key security factor; even minor engineering mistakes can compromise safety. As a consequence, most JVM implementors today use the verifier implementation provided by Sun Microsystems. Even those JVMs that do not use Sun's code verbatim (e.g. the Jikes RVM [AAB$^+$00] and Kaffe [Wil06]) use in principle the same data-flow algorithm.

Hence, bytecode verification by data-flow analysis is an established and widely

```
1: $todo \leftarrow true$
2: **while** $todo = true$ **do**
3:   $todo \leftarrow false$
4:   **for all** $i$ in all instructions of a method **do**
5:     **if** $i$ was changed **then**
6:       $todo \leftarrow true$
7:       check whether stack and local variable
         types match definition of $i$
8:       calculate new state after $i$
9:       **for all** $s$ in all successor instructions of $i$
         **do**
10:        **if** current state for $s \neq$ new state derived
           from $i$ **then**
11:          assume state after $i$ as new entry state
             for $s$
12:          mark $s$ as changed
13:        **end if**
14:      **end for**
15:    **end if**
16:  **end for**
17: **end while**
```

Figure 2.1: The standard verification algorithm found in Sun Microsystem's JVM implementations.

deployed approach. For average Java programs, verification effort seems to be negligible: for shorter programs it seems to be somewhat quicker; for larger programs it seems to take somewhat longer.

Java developers and users seem to take for granted that Java bytecode verification effort scales in some acceptable fashion with program size. This assumption, as we will show in the course of this chapter, is unsubstantiated.

We will demonstrate that the perception of linear scaling of the JVM verification effort with program length is not only wrong, but that JVM programs can be constructed that can keep the verifier busy for so long as to constitute a denial-of-service attack. This has grave consequences when using Java in mobile code environments such as applets and agent-based systems. The effect may be even worse when such a program is sent to a server virtual machine, where there is no user to kill and restart the VM.

$$
\begin{aligned}
&\texttt{Types} = \{\texttt{I}, \texttt{A}, \ldots, \top\} \cup \texttt{Classes}\\
&\textit{LocVar} : \mathbb{N} \to \texttt{Types}, \textit{Stack} = (\texttt{Types} \cup \{\texttt{Err}\})^{n \geq 0}\\
&\textit{push} : \texttt{Types} \times \textit{Stack} \to \textit{Stack}, \textit{pop} : \textit{Stack} \to \textit{Stack}\\
&\textit{State} = \textit{Stack} \times \textit{LocVar}, \phi : \textit{State} \times \texttt{Instr} \to \textit{State}
\end{aligned}
$$

$$
\begin{aligned}
&\phi\left((S, L), \texttt{iconst\_}n\right) = (\langle \texttt{I}, \widehat{S}_i \rangle \sqsubseteq \widehat{S}_{i+1} S, L)\\
&\phi\left((S, L), \texttt{aconst\_null}\right) = (\langle \texttt{A}, \widehat{S}_i \rangle \sqsubseteq \widehat{S}_{i+1} S, L)\\
&\phi\left(((\texttt{I}, \texttt{I}, S), L), \texttt{iadd}\right) = (\langle \texttt{I}, \widehat{S}_i \rangle \sqsubseteq \widehat{S}_{i+1} S, L)\\
&\phi\left(((\texttt{I}, S), L), \texttt{istore\_}n\right) = (S, L[n \leftarrow \texttt{I}])\\
&\phi\left((S, L), \texttt{iload\_}n\right) = (\langle L(n), \widehat{S}_i \rangle \sqsubseteq \widehat{S}_{i+1} S, L), \text{ if } L(n) = \texttt{I}\\
&\phi\left(((\tau, S), L), \texttt{astore\_}n\right) = (S, L[n \leftarrow \tau]), \text{ if } L(n) \in \texttt{Classes}\\
&\phi\left((S, L), \texttt{aload\_}n\right) = (\langle L(n), \widehat{S}_i \rangle \sqsubseteq \widehat{S}_{i+1} S, L), \text{ if } L(n) \in \texttt{Classes}\\
&\phi\left(((\alpha_n, \ldots, \alpha_1, S), L), \texttt{invokestatic}\ C.m.\textit{sig}\right)\\
&\qquad = \left(\langle \beta, \widehat{S}_i \rangle \sqsubseteq \widehat{S}_{i+1} S, L\right) \text{ if } \textit{sig} = \beta(\beta_1, \ldots, \beta_n)\\
&\qquad\qquad \text{and } \forall i = 1 \ldots n : \alpha_i \text{ is subtype of } \beta_i
\end{aligned}
$$

Figure 2.2: Internal state and selected rules for the type-level abstract interpreter.

The remainder of this chapter is organized as follows. In Section 2.2 we will analyze the Java bytecode verification algorithm and attempt to construct worst-case scenarios as far as completion time and resource consumption are concerned. Section 2.3 contains verification time benchmarks for the code examples constructed in the preceding section, and in Section 2.4 we discuss existing and possible future countermeasures to the described denial-of-service attack scenario. Section 2.5 discusses the wider implications of this exploit for the entire dynamic code optimization and compilation pipeline.

2.2 Java Bytecode Verification

This section gives an overview of the Java bytecode verification algorithm. As already pointed out, the whole security concept of the JVM is centered around verification. The bytecode verifier [Ler01a, SS01] checks a piece of code for type consistency and other type-safety relevant properties. Leroy [Ler03] lists the least conditions for bytecode to be accepted by the verifier:

- *Type correctness*. Bytecode instructions are typed and must receive arguments of corresponding types.
- *No stack overflow or underflow*. A method must never pop a value from the empty stack or push a value onto the maximal stack specified for that method.
- *Code containment*. The program counter must always stay within the code limits of the currently active method and must always point to the beginning of a valid instruction.
- *Local variable initialization*. No variable may be loaded that has not been initialized first.
- *Object initialization*. Whenever an object of a class C is created, one of the class' constructors must be called.

One possibility to ensure that bytecode obeys to all these restrictions is to check it dynamically at runtime [Coh97]. However, this is expensive and is not advisable for all applications since it slows down execution significantly.

In order to eliminate runtime checks, Yellin, Lindholm et al. introduced *bytecode verification* [Yel95, LY96], where properties are checked statically before execution. The basic ingredient of every bytecode verifier is an abstract interpreter for Java Virtual Machine Language (JVML) instructions. Unlike JVM, its stacks and virtual registers store *types*, rather than *values*. Thus, the interpreter translates instructions into operations that execute on types. Therefore, the program's class types are presented as a graph, where the nodes are types and the edges represent sub-typing relations. Figure 2.3 shows an example program. The corresponding type graph is shown in Figure 2.4. The *main* method first stores some integer values into local variables and then instantiates one object of type B and one object of type C. In two different control-flow paths, these objects are stored in the same local variable location.

```
  class A {
    void m() {}
  }
  class B extends A {
    void m() {}
  }
  class C extends A {
    void m() {}
  }
  class App {
    void main() {
/*1*/ iconst 1
/*2*/ istore 1
      iload 1
      iconst 1
      iadd
/*3*/ istore 2
      iload 2
/*4*/ iconst 2
      if_icmple goto L1
      new B
      dup
      invokespecial B/B()
/*5*/ astore 3
      goto L2
L1:
      new C
      dup
      invokespecial C/C()
/*6*/ astore 3
L2:
      aload 3
/*7*/ invokevirtual A/m()
    }
  }
```

Figure 2.3: Example Java program in bytecode form.

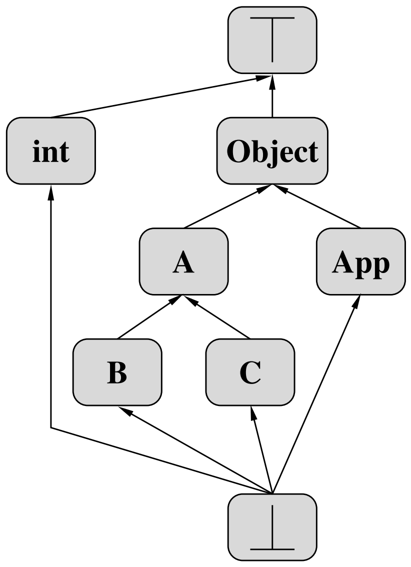


Figure 2.4: Type graph for the example Java program in Figure 2.3.

Figure 2.2 shows the definitions for the internal state as well as selected rules of an abstract JVML interpreter. *push* and *pop* have the usual definition on stacks; a stack overflow or underflow generates the `Err` state. Note that exceptions do not add to the behavior of the abstract interpreter and are hence ignored in this step. The rules describe the preconditions for the stack and the register component of the internal state. If there is no applicable definition for ϕ, an error occurs. It is noteworthy that the interpretation of method calls such as `invokestatic` does not actually call the method. Instead, it assumes that the method's effect is to push an object of type β on the stack as described by the method's signature.

The data-flow algorithm used in Sun's implementation of bytecode verification is shown in Figure 2.1. This algorithm is performed separately for every method in the Java program. For each method, it iterates over all instructions of that method until no more operand type changes are observed.

For each instruction i, the verifier checks whether the abstract data associated with i has changed. If so, it checks whether the current abstract local variable and stack content allows the execution of i and computes the new local variable and stack content. Finally,

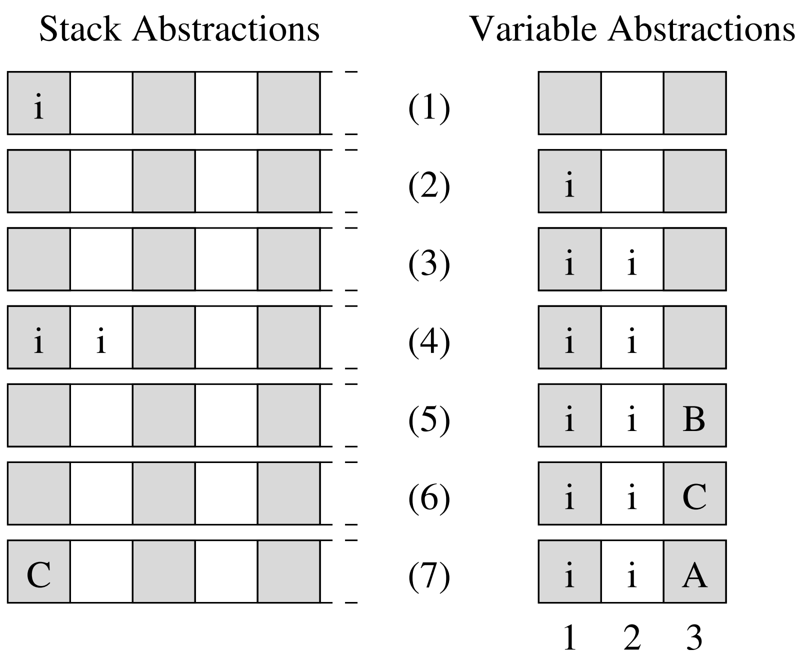


Figure 2.5: The stack and variable states for method `main`

this new abstract state is propagated to all successors of i.

The analysis of straight-line code is inexpensive, since the abstract interpreter only propagates type information through the instructions and computes the abstract stack state after each instruction. Figure 2.5 shows the computed stack and local variable states for method `main` in the example program after 7 selected instructions.

The first part of the example code in Figure 2.3 is simple to verify. From (1) to (4) the code is simply executed at the symbolic level using types instead of values.

The runtime of such a data-flow analysis is significantly increased if the code contains branches, exception handlers, and subroutines, which introduce forks and joins in the control-flow graph. When separate control flows are merged together, an instruction's predecessors may have different abstract stack or variable types. In the example in Figure 2.3, the call to the method `m` based on local variable 3 may be called on either a `B` or a `C` object. The verifier adapts the abstract type of the variable slot containing either `B` or `C` to be the smallest common ancestor of the two classes in the type graph (`A`). After merging the state information, the data-flow analysis has to be repeated for all instructions

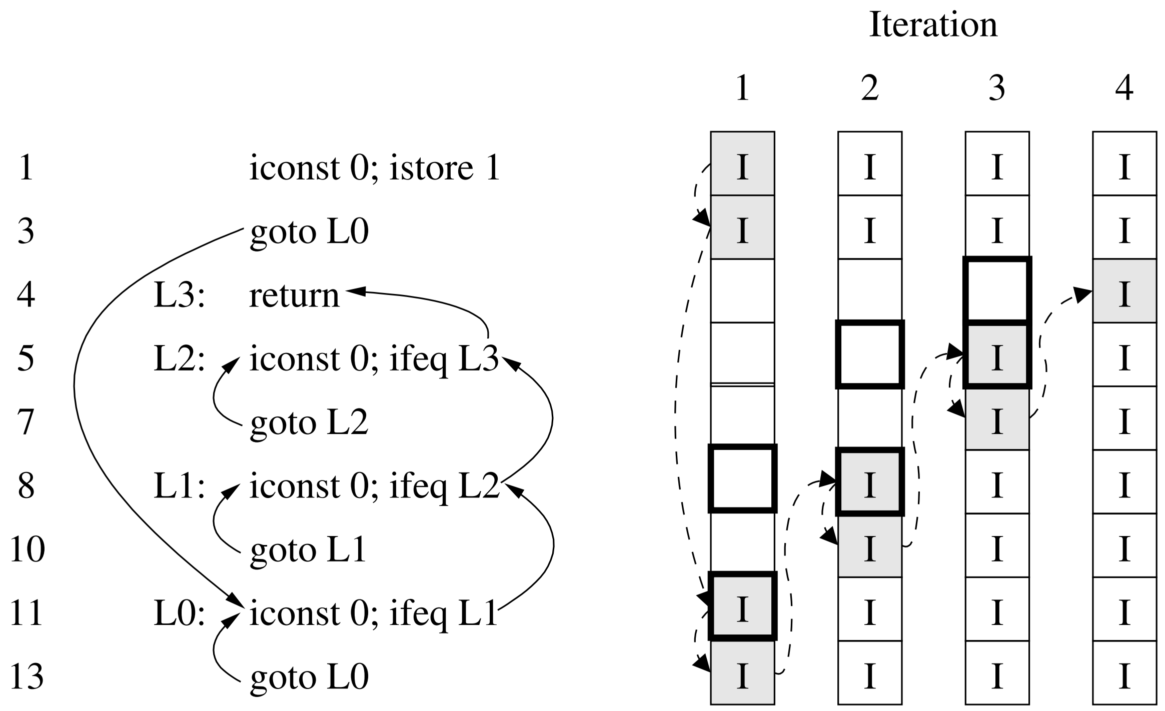


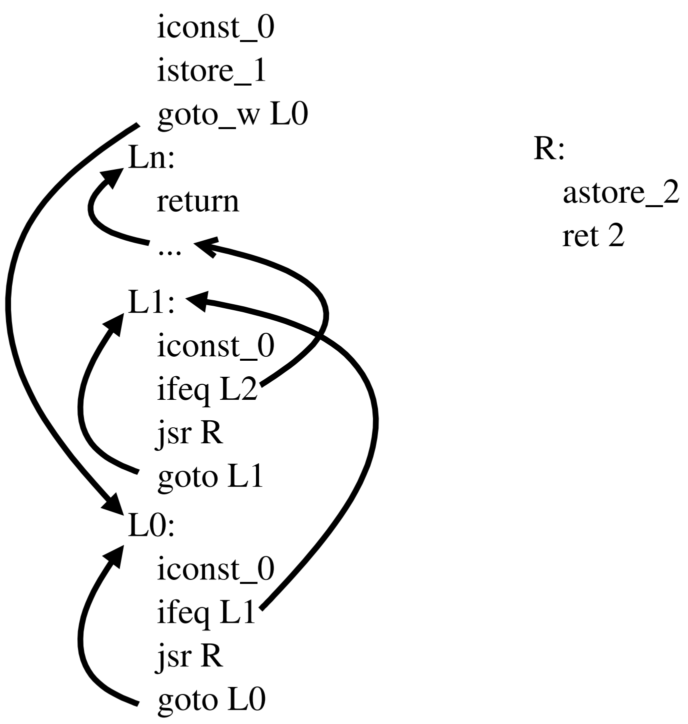
Figure 2.6: Java bytecode program that takes n iterations to be verified using Sun's standard DFA verifier approach.

which are reachable from this point in the control flow of the method. For simplicity, the Java verifier repeats the entire data-flow analysis for every instruction of a method every time there is a change.

For average Java programs, the verifier algorithm quickly reaches a fixed point after only a few iterations. It is obvious, that –*in theory*– the Java verifier could need up to n iterations over the method, with n being the number of instructions in the method. As for each iteration the verifier might have to visit all instructions, the overall complexity is $O(n^2)$.

However, such quadratic runtime behavior does not only exist in theory. We will show in the remainder of this section how simple Java programs can be constructed which expose the worst case scenario in practice.

Studying the pseudo code of the verifier algorithm in Figure 2.1 reveals that newly computed type information is immediately available for *downstream* instructions, but can be propagated *upstream* only during the next iteration of the DFA. This property is given



```
  iconst_0
  istore_1
  goto_w L0
Ln:                         R:
  return                      astore_2
  ...                         ret 2
L1:
  iconst_0
  ifeq L2
  jsr R
  goto L1
L0:
  iconst_0
  ifeq L1
  jsr R
  goto L0
```

Figure 2.7: Increasing the verification using subroutine calls.

by the order in which the algorithm iterates over the instructions in each method. Once an instruction was visited for a particular iteration, it will not be visited again, even if new information about the operand types of that instruction was computed. Thus, if we manage to order N instructions in such a way that each depends on the completion of the verification of the *successor* instruction, we effectively force the verifier to repeat the data-flow analysis N times.

Consider the Java bytecode in Figure 2.6. The right hand part of Figure 2.6 shows several stages during verification, namely the computed abstract type for local variable 1 ($LV1$) in each iteration.

At the entry point of the method, an integer constant is loaded into ($LV1$) and the control is transfered to $L0$. The verifier will actually not follow the branch instruction to the target, but continue to check instructions in sequence. Once reaching $L0$, the verifier remembers that $L0$ was already the target of a jump instruction with $type(LV1) =$ *integer*. The first two instructions after $L0$ constitute a conditional branch.

While the branch is actually statically predictable in this example, the verifier does not perform value folding and thus considers the *ifeq* instruction as conditional. The verifier records that this *ifeq* could transfer the control flow to $L1$ with $type(LV1) =$ *integer*.

The unconditional *goto* at the end of this code block transfers control back to $L0$. This computation is repeated for each of the basic blocks in the program. When the information for the `return` statement has been determined to be `I` the verifier terminates.

The number of basic blocks arranged in this fashion determines how often the verifier has to iterate over the code. For N basic blocks the verifier will have to iterate at least N times over the code, because the length of the longest path information has to flow along backwards is N.

To achieve an even greater slowdown, each basic block could jump to an empty subroutine using the `jsr` instruction (Figure 2.7). While not increasing the theoretical complexity of the verification, the practical verification time is indeed significantly higher as we will see in the benchmarks in the following section.

To evaluate the worst case efficiency of the Java bytecode verifier, in the following section we will run the verifier on Java methods containing code constructed using the control-flow pattern we developed in this section.

It is noteworthy that, while the proposed denial-of-service attack is based on the specific properties of Sun's verifier, similar attacks can be constructed for other implementations.

2.3 BENCHMARKS

We have measured the verification time for the two example programs presented in the previous section using the Sun Microsystems Java 2 HotSpot Client VM[1]. As we have

[1] Java 2 Runtime Environment, Standard Edition (build 1.4.1.02-b06), Java HotSpot Client VM (build 1.4.1.02-b06, mixed mode), running on a Dell Dimension 8250, 2.53GHz P4, 512MB RAM, RedHat Linux 9.

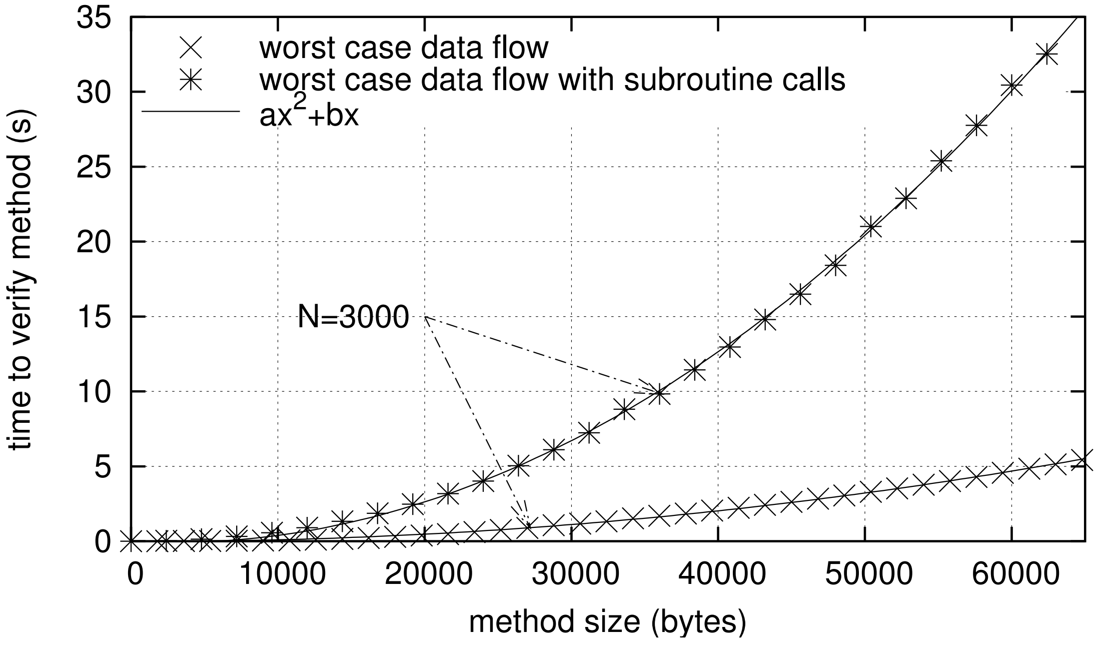


Figure 2.8: Verfication time for a worst-case data-flow scenario.

mentioned above, not all JVM implementors are using exactly the verifier implementation offered by Sun. We have repeated our tests with a number of JVMs from other vendors. While slight performance advantages or disadvantages can be observed, we are not aware of any verifier implementation that does not expose quadratic runtime behavior for the discussed test cases.

Figure 2.8 shows the verification time for a single method containing bytecode with an increasing maximum data-flow path of length N. This time includes only the time it takes the verifier to prove safety. The code is never actually executed or compiled to executable code. The first curve shows the for verification time for the basic example shown in Figure 2.6.

The x-axis indicates the length of the method bytecode in bytes, which is proportional to the number of basic blocks N used to construct the code. The second curve in the graph shows the maximum flow path problem with an added subroutine call in each code block. Both curves clearly show quadratic growth. The second curve grows much faster than the basic example due to the poor implementation of the verifier. The increased steepness is

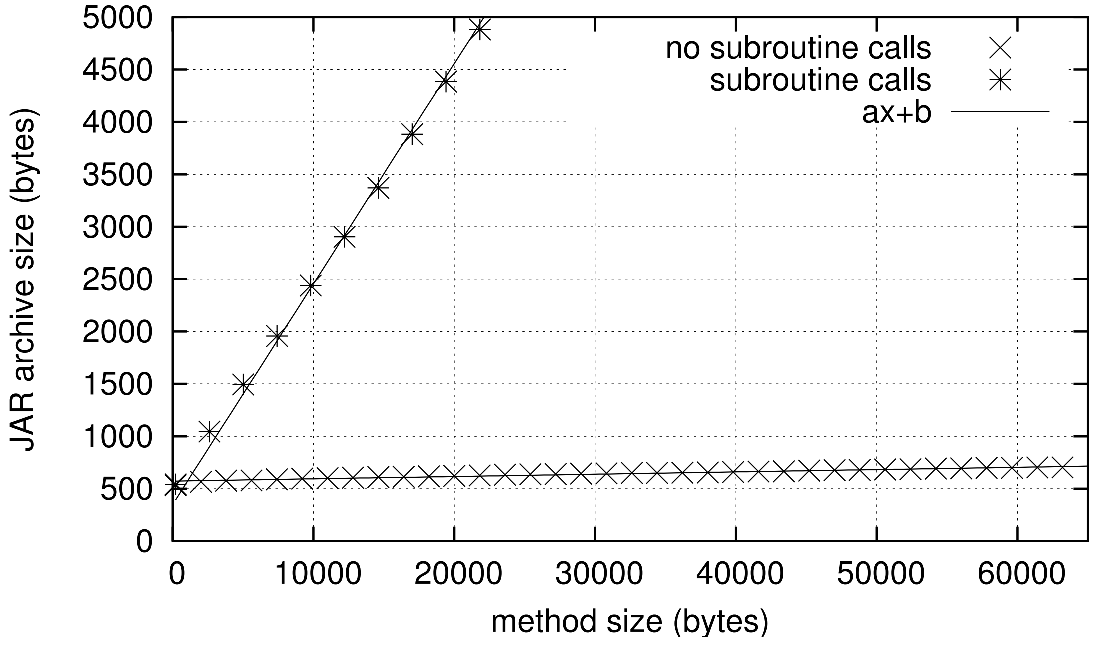


Figure 2.9: Compressibility of the exploit code.

not caused by a symptomatic problem. We have merely included this second curve to make the quadratic behavior more visible. The arrows indicate for comparison purposes the code size for the maximum path length $N = 3000$ for each of the examples.

All measurements were taken on a 2.53 GHz P4 running Linux and the Sun HotSpot VM 1.41. The maximum verification time we observed on this machine for a single method was approximately 40 seconds. The maximum basic block count N we could reach was $N = 7280$ for the simplified scenario and $N = 5460$ for the test case with subroutine calls. This stems from the 65,536 bytes limit for method code in the JVM. To achieve even longer verification times, an attacker could hide more than just one of these methods in the code. Just including 20 methods instead of one would already increase the verification time to approximately 15 minutes on a 2.54 GHz P4.

The standard JAR archive format used by Java can be used to drastically reduce the apparent size of the malicious code. But not only method repetitions can be compressed well using this approach. The code patterns used in the presented scenarios lend themselves for compression due to their very regular structure. Figure 2.9 indicates the compressed size for different problem lengths N for each of the two approaches. The basic scenario can be compressed much denser using the standard JAR algorithm, because each block consists at the bytecode level of exactly the same instructions. This is guaranteed because Java uses relative addressing for jump instructions. The branch instructions in each block thus use the same relative offset over and over again.

For the scenario using subroutine calls, the JAR archive format of the JVM is less favorable. The subroutine invocation has to reference the same subroutine location from different program counter locations, creating a different offset for each subroutine call instruction.

To evaluate the practical impact of the shortcoming of the JVM verification discussed in this chapter, we have set up a website containing a Java applet, which is automatically executed when the browser displays the website. The applet (`time.class`) loads a second class file (`paralyse.class`) using the Java `Class.forName()` API call and measures the time it takes the JVM to load the requested class. Both class files reside in a JAR archive whose size is less than 3kB.

On our test machines the execution time for the applet verification when navigation to the website with a browser ranged from minutes to hours, depending on the machine and JVM used.

Short of disabling Java applets, the user cannot prevent or interrupt the loading of this applet. In fact, many browsers do not even allow the user to interrupt the verification because the browser implementor never considered the verification time to be long enough. Other browsers, including some versions of the Microsoft Internet Explorer, allow the verifier to continue the verification silently and continue to *hog* the CPU in the background if the user leaves a website containing an applet which takes an excessive amount of time to verify.

2.4 COUNTERMEASURES

In contrast to security flaws previously discovered in the JVM [MF97], the enabling property for this vulnerability of the JVM verifier is an *inherent property* of the algorithm used and not merely some *faulty code* that could be exchanged.

Rewriting the verifier algorithm to iterate over the code in some other order, or the introduction of a work list algorithm, would not significantly improve the situation. Each of these algorithms would still expose quadratic runtime behavior for some worst case code scenarios.

However, a number of mitigating factors exist. First, current JVMs limit the code size per method to 65,536 bytes. On high-end desktop systems this limits the maximum verification time we were able to achieve using a single method to approximately 40s. This (probably accidental) ceiling prevents the construction of worst case scenarios with near-infinite verification time.

Further shortening the maximum method length of Java methods is not an option, since long Java methods are not uncommon. Some compilers emitting Java bytecode generate even very long methods. This includes some XML transformation tools and parser generators. It would be not surprising if Sun decided to remove the current code size limitation in future versions of the Java Virtual Machine.

It seems unlikely that one could establish a clear set of rules to detect this class of malicious programs. Just rejecting a program because it takes more than a certain number of iterations to verify would be arbitrary. On the other hand, trying to detect patterns such as the ones described in this chapter would not eliminate the problem as more complex and less obvious examples can be easily constructed. It would also get us back to the *pattern matching* approach used in virus detection tools, something that bytecode verification was supposed to free us from.

The impact of the exploit described in this chapter can be increased by shipping a large number of malicious methods to the verifier. While this increases verification time by only a linear factor, in conjunction with compressed archives (JAR), verification times in the magnitude of minutes and hours are achievable. The corresponding JAR archive would still be only a few kilobytes in size. Detecting this exploit is easier than the single method approach exploit. For agent-based systems or applets, restricting the overall code volume is probably acceptable. With such code size restrictions, the verification time could be limited as well.

Adding resource monitoring to the verification process could be used to counter this attack. However, bytecode verification is deeply embedded into the JVM. Introducing the possibility to abort a running verification from the outside would requires invasive changes to JVM implementations. Similar to all other previously discussed approaches, resource monitoring introduces arbitrary abort conditions for the verifier and might prevent an important and desirable Java applet or agent to run just because it takes longer than expected to verify the code.

Instead of performing the expensive DFA on the code consumer side, it has been proposed to supply the code consumer with the fixed point of the DFA. The consumer then only has to check whether the supplied fixed point indeed satisfies the data-flow equations, which can be done in linear time. This and other related works are discussed in Section 5.1.

2.5 Discussion

In this chapter, we showed that the worst-case verification effort of the Java bytecode verifier can be problematic. By carefully analyzing the data-flow algorithm underlying the Java verification approach, we were able to construct Java byte codes that, while correct, consume inordinate CPU resources during their verification. In conjunction with the JAR format, this means that relatively short class files can be constructed that require minutes or even hours to verify, even on high-end desktop systems. This in turn could be used in a denial-of-service attack.

We addressed some of the possible countermeasures to such denial-of-service attacks and argued that this exploit is hard to counter without also accepting the occasional spurious rejection of non-malicious programs—unless one is willing to switch to a completely different verification scheme.

From a more general perspective, the vulnerability described in this chapter demonstrates the need, when dealing with mobile code, for algorithms that are not only correct, but also *efficient*. This is not necessarily restricted to the verification problem: an adversary that knows the inner workings of a just-in-time compiler could construct an attack based on the worst-case behavior of an algorithm used in a dynamic code generator or optimizer. This observation forms the basis for this dissertation. In the next chapter we will develop a new algorithm to efficiently verify Java bytecode, and from there on move on to provide a new dynamic compilation method that permits efficient code generation with near-linear time worst case performance.

CHAPTER 3

PROOFING: SSA-BASED JAVA BYTECODE VERIFICATION

Traditionally, JVMs perform verification using an iterative data-flow analysis. This is necessary since the locations of temporary variables in the JVM are not statically typed. In this chapter we introduce an alternative bytecode verification algorithm that verifies bytecode via Static Single Assignment (SSA) form [CFR⁺91] construction. The resulting SSA representation can immediately be used for optimization and code generation. We show that our SSA-based verifier accepts and rejects the same class of programs as the standard verifier algorithm. Our prototype implementation transforms bytecode into SSA and verifies it in less time than it takes Sun's verifier to merely confirm the validity of Java bytecode. [GPF05a, GPF05b, GPF06b]

3.1 INTRODUCTION

The Java Virtual Machine (JVM) pioneered the concept of *code verification*, by which a receiving host examines each arriving mobile program to rule out potentially malicious behavior even before starting execution. All existing Java bytecode verifiers are based on an iterative data-flow analysis with quadratic worst-case behavior. We present an alternative verification mechanism.

Instead of verifying bytecode directly, we first annotate it such that the flow of values between instructions becomes explicit rather than going through the operand stack. We call this first intermediate step *annotated JVML (JVML$_{\mathcal{A}}$)*. In a second step, we transform further into a *Static Single Assignment variant of JVML (JVML$_{\mathcal{SSA}}$)*. Programs in JVML$_{\mathcal{SSA}}$ can be verified in near linear-time and without requiring an iterative data-flow analysis.

Our benchmarks indicate that the aggregate time required for first transforming JVML via JVML$_{\mathcal{A}}$ into JVML$_{\mathcal{SSA}}$ and then verifying the JVML$_{\mathcal{SSA}}$ representation is still less than the time needed for performing the standard verification algorithm directly on JVML.

While our actual implementation covers the complete JVM language, length restrictions prevent us from formally showing that JVML and JVML$_{\mathcal{SSA}}$ are equivalent. Hence, in this paper we only show the semantics and transformations for JVML$_{\mathcal{S}}$, a representative subset of JVML. Conversely, the performance benchmarks towards the end of the paper refer to the full JVM language.

The remainder of this chapter is organized as follows. In Section 3.2 we introduce JVML$_{\mathcal{S}}$, a representative subset of the Java bytecode language. Section 3.3 shows the semantics-preserving transformation from JVML$_{\mathcal{S}}$ via JVML$_{\mathcal{A}}$ into JVML$_{\mathcal{SSA}}$. In Section 3.5 we compare our method to Sun's standard verifier. Section 3.6 discusses the results of our work.

3.2 JVML$_{\mathcal{S}}$

For the remainder of this paper we use JVML$_{\mathcal{S}}$ to reason about properties of the Java bytecode language. While very compact, JVML$_{\mathcal{S}}$ is complete enough to explain a number of difficulties that occur during the verification of JVML.

We do not support the JVML subroutine construct [LY96] in JVML$_{\mathcal{S}}$. It is a significant complication when dealing with Java bytecode [SA98, FM99] and has been shown to be not a very effective way of reducing code size [Fre98]. We have studied numerous bytecode applications including the Eclipse framework, different Java APIs, and the SPEC benchmarks. Of approximately 5.4 million instructions we only found 0.24% to be in subroutines. The average size of a subroutine was 7 instructions and it was only called 2 times.

A JVML$_{\mathcal{S}}$ program is a sequence of instructions (Figure 3.1). For ease of presentation

$$\begin{array}{rcl}
\textit{program} & ::= & \textit{instruction}^* \\
\textit{instruction} & ::= & \textit{core} \mid \textit{dataflow} \\
\textit{core} & ::= & \texttt{iconst\_}n, \text{where } -1 \leq n \leq 5 \mid \\
 & & \texttt{lconst\_}l, \text{where } l \in \{0, 1\} \mid \\
 & & \texttt{iadd} \mid \\
 & & \texttt{ladd} \mid \\
 & & \texttt{ifeq}\ L, \text{where } L \in \mathbb{N} \mid \\
 & & \texttt{goto}\ L, \text{where } L \in \mathbb{N} \mid \\
 & & \texttt{return} \\
\textit{dataflow} & ::= & \texttt{pop} \mid \\
 & & \texttt{pop\_2} \mid \\
 & & \texttt{dup} \mid \\
 & & \texttt{dup\_2} \mid \\
 & & \texttt{istore\_}x, \text{where } x \in \mathbb{N} \mid \\
 & & \texttt{iload\_}x, \text{where } x \in \mathbb{N} \mid \\
 & & \texttt{lstore\_}x, \text{where } x \in \mathbb{N} \mid \\
 & & \texttt{lload\_}x
\end{array}$$

Figure 3.1: Instructions in JVML$_{\mathcal{S}}$.

we assume that each basic block ends with a `goto` to its successor. Thus, each conditional jump is followed by an unconditional one. Following [SA99], we treat programs as partial maps from the address space `ADDR` to instructions `INSTR`. Addresses are positive integers, the entry point of a method is 1. *Dom*$(P) \subset$ `ADDR` is the set of valid addresses in P. If m is a map, then *Dom*(m) is the domain of m and $m[x]$ with $x \in$ *Dom*(m) is the value of m at x. $m[x \mapsto y]$ is the map with the same domain as m defined by $(m[x \mapsto v])[y] = m[y]$ if $y \neq x$ and v if $y = x$.

Equality on maps is defined by
$f = g \Leftrightarrow \textit{Dom}(f) = \textit{Dom}(g) \wedge \forall x \in \textit{Dom}(f) : f[x] = g[x]$.

JVML$_{\mathcal{S}}$ supports object types and two scalar types. T is the set of all object types referenced by P. $\top$ is the common supertype of all types, including Java's Object type, $\perp$ is the uninitialized type. The scalar types are integers (`I`) and long integers. Similar to

$$\begin{array}{lcl} \alpha & \in & T \\ \mathcal{T} & ::= & \{\mathtt{I}, \mathtt{L}, \mathtt{L}', \alpha, \top, \bot\} \\ \mathcal{V} & ::= & \textit{integer} \,\dot\cup\, \textit{long} \,\dot\cup\, \textit{long'} \,\dot\cup\, T \end{array}$$

Figure 3.2: Types and values.

JVML, long integers occupy two consecutive stack locations and variables. Therefore, we divide long integers into two halves: the bottom half is of type `L`, while the top half is of type `L`'. Accordingly, long integer values are divided into the two sets *long* and *long*'.

All type names are unified to the set $\mathcal{T}$; $\mathcal{V}$ is the corresponding disjoint union of values of the types in $\mathcal{T}$(Figure 3.2).

Instructions operate on an operand stack. Additionally, values can be stored in variables. Variables are non-negative integers that correspond to local variables in JVML. In the static semantics, stacks and local variables store types. Stacks are modeled as vectors of items, variable environments are mappings from variables to items.

For the static semantics that reasons about types this results in $\widehat{\mathtt{STACK}} ::= \mathcal{T}^{n,0\leq n}$ and $\widehat{\mathtt{ENV}} ::= \mathtt{VARS} \rightarrow \mathcal{T}$.

In contrast, the operational semantics works on actual *values*, resulting in $\mathtt{STACK} ::= \mathcal{V}^{n,0\leq n}$ and $\mathtt{ENV} ::= \mathtt{VARS} \rightarrow \mathcal{V}$.

$\widehat{\sigma}_i \in \widehat{\mathtt{ENV}}$ and $\sigma_i \in \mathtt{ENV}$ are variable environments, mapping variables to the type respectively value stored in them. $\widehat{S}_i \in \widehat{\mathtt{STACK}}$ and $S_i \in \mathtt{STACK}$ are stacks, written as $\langle s_{sd}, s_{sd-1}, \cdots, s_0\rangle$, with $0 \leq sd$, where sd is the stack depth. We use $\tau \in \mathcal{T}$ and $\nu \in \mathcal{V}$ as type respectively value variables.

Based on the Java class hierarchy and its subtype relation, using standard lattice theory variable maps and stacks form lattices with the ordering relations $\sqsubseteq_{\widehat{\sigma}}$ and $\sqsubseteq_{\widehat{S}}$ according to the Java type rules and least upper bound operators $\sqcup_{\widehat{\sigma}}$ and $\sqcup_{\widehat{S}}$. We will omit the index whenever it is clear from the context which is meant. It is noteworthy that $\sqcup_{\widehat{S}}$ is only defined for arguments that abstract stacks with the same stack depth. As required by the

$$[\texttt{iconst\_}n] \quad \frac{\begin{array}{c} P[i] = \texttt{iconst\_}n \wedge i+1 \in \textbf{\textit{Dom}}(P) \\ \widehat{\sigma}_i \sqsubseteq \widehat{\sigma}_{i+1} \wedge \langle \texttt{I}, \widehat{S}_i \rangle \sqsubseteq \widehat{S}_{i+1} \end{array}}{\widehat{\sigma}, \widehat{S}, i \vdash P}$$

$$[\texttt{iadd}] \quad \frac{\begin{array}{c} P[i] = \texttt{iadd} \wedge i+1 \in \textbf{\textit{Dom}}(P) \\ \widehat{\sigma}_i \sqsubseteq \widehat{\sigma}_{i+1} \wedge \widehat{S}_i = \langle \texttt{I}, \texttt{I}, \widehat{S'} \rangle \wedge \langle \texttt{I}, \widehat{S'} \rangle \sqsubseteq \widehat{S}_{i+1} \end{array}}{\widehat{\sigma}, \widehat{S}, i \vdash P}$$

$$[\texttt{lconst\_}l] \quad \frac{\begin{array}{c} P[i] = \texttt{lconst\_}l \wedge i+1 \in \textbf{\textit{Dom}}(P) \\ \widehat{\sigma}_i \sqsubseteq \widehat{\sigma}_{i+1} \wedge \langle \texttt{L}, \texttt{L}', \widehat{S}_i \rangle \sqsubseteq \widehat{S}_{i+1} \end{array}}{\widehat{\sigma}, \widehat{S}, i \vdash P}$$

$$[\texttt{ladd}] \quad \frac{\begin{array}{c} P[i] = \texttt{ladd} \wedge i+1 \in \textbf{\textit{Dom}}(P) \\ \widehat{\sigma}_i \sqsubseteq \widehat{\sigma}_{i+1} \wedge \widehat{S}_i = \langle \texttt{L}, \texttt{L}', \texttt{L}, \texttt{L}', \widehat{S'} \rangle \\ \langle \texttt{L}, \texttt{L}', \widehat{S'} \rangle \sqsubseteq \widehat{S}_{i+1} \end{array}}{\widehat{\sigma}, \widehat{S}, i \vdash P}$$

$$[\texttt{ifeq } L] \quad \frac{\begin{array}{c} P[i] = \texttt{ifeq}\, L \wedge (i+1), L \in \textbf{\textit{Dom}}(P) \\ \widehat{\sigma}_i \sqsubseteq \widehat{\sigma}_L \wedge \widehat{\sigma}_i \sqsubseteq \widehat{\sigma}_{i+1} \wedge \widehat{S}_i = \langle \texttt{I}, \widehat{S'} \rangle \wedge \widehat{S'} \sqsubseteq \widehat{S}_{i+1} \wedge \widehat{S'} \sqsubseteq \widehat{S}_L \end{array}}{\widehat{\sigma}, \widehat{S}, i \vdash P}$$

$$[\texttt{goto } L] \quad \frac{\begin{array}{c} P[i] = \texttt{goto}\, L \wedge L \in \textbf{\textit{Dom}}(P) \\ \widehat{\sigma}_i \sqsubseteq \widehat{\sigma}_L \wedge \widehat{S}_i = \langle \texttt{I}, \widehat{S'} \rangle \wedge \widehat{S'} \sqsubseteq \widehat{S}_L \end{array}}{\widehat{\sigma}, \widehat{S}, i \vdash P}$$

Figure 3.3: Static semantics for JVML$_\mathcal{S}$ core instructions.

Java verifier description, this ensures that at control flow joins each incoming edge has the same stack depth.

The instruction set of JVML$_\mathcal{S}$ consists of two kinds of instructions. *Core* instructions operate on values stored on the operand stack, while *dataflow* instructions facilitate the flow of values between core instructions by manipulating the state of the operand stack and exchanging values between operand stack and variables.

Next, we introduce the static semantics of JVML$_\mathcal{S}$. A program P is well-typed if there exists a vector $\widehat{\sigma}$ of maps from variables to types and a vector $\widehat{S}$ of vectors of types that satisfy the judgment $\widehat{\sigma}, \widehat{S} \vdash P$.

[istore_x]
$$\frac{\begin{array}{c} P[i] = \texttt{istore\_}x \wedge i+1 \in \textit{\textbf{Dom}}(P) \\ \sigma_i[x \mapsto \texttt{I}] \sqsubseteq \widehat{\sigma}_{i+1} \wedge \widehat{S}_i = \langle \texttt{I}, \widehat{S}' \rangle \wedge \widehat{S}' \sqsubseteq \widehat{S}_{i+1} \\ x \in \texttt{VARS} \end{array}}{\widehat{\sigma}, \widehat{S}, i \vdash P}$$

[iload_x]
$$\frac{\begin{array}{c} P[i] = \texttt{iload\_}x \wedge i+1 \in \textit{\textbf{Dom}}(P) \\ \widehat{\sigma}_i \sqsubseteq \widehat{\sigma}_{i+1} \wedge \langle \texttt{I}, \widehat{S}_i \rangle \sqsubseteq \widehat{S}_{i+1} \\ x \in \textit{\textbf{Dom}}(\widehat{\sigma}) \wedge \widehat{\sigma}_i(x) = \texttt{I} \end{array}}{\widehat{\sigma}, \widehat{S}, i \vdash P}$$

[lstore_x]
$$\frac{\begin{array}{c} P[i] = \texttt{lstore\_}x \wedge i+1 \in \textit{\textbf{Dom}}(P) \\ \sigma_i[x \mapsto \texttt{L}', (x+1) \mapsto \texttt{L}] \sqsubseteq \widehat{\sigma}_{i+1} \wedge \widehat{S}_i = \langle \texttt{L}, \texttt{L}', \widehat{S}' \rangle \wedge \widehat{S}' \sqsubseteq \widehat{S}_{i+1} \\ x, x+1 \in \texttt{VARS} \end{array}}{\widehat{\sigma}, \widehat{S}, i \vdash P}$$

[lload_x]
$$\frac{\begin{array}{c} P[i] = \texttt{lload\_}x \wedge i+1 \in \textit{\textbf{Dom}}(P) \\ \widehat{\sigma}_i \sqsubseteq \widehat{\sigma}_{i+1} \wedge \langle \texttt{L}, \texttt{L}', \widehat{S}_i \rangle \sqsubseteq \widehat{S}_{i+1} \\ x, x+1 \in \textit{\textbf{Dom}}(\widehat{\sigma}) \\ \widehat{\sigma}_i(x) = \texttt{L}' \wedge \widehat{\sigma}_i(x+1) = \texttt{L} \end{array}}{\widehat{\sigma}, \widehat{S}, i \vdash P}$$

[pop]
$$\frac{\begin{array}{c} P[i] = \texttt{pop} \wedge i+1 \in \textit{\textbf{Dom}}(P) \\ \widehat{\sigma}_i \sqsubseteq \widehat{\sigma}_{i+1} \wedge \widehat{S}_i = \langle \tau, \widehat{S}' \rangle \wedge \widehat{S}' \sqsubseteq \widehat{S}_{i+1} \\ \tau \neq \texttt{L}' \end{array}}{\widehat{\sigma}, \widehat{S}, i \vdash P}$$

[pop_2]
$$\frac{\begin{array}{c} P[i] = \texttt{pop\_2} \wedge i+1 \in \textit{\textbf{Dom}}(P) \\ \widehat{\sigma}_i \sqsubseteq \widehat{\sigma}_{i+1} \wedge \widehat{S}_i = \langle \tau_1, \tau_2, \widehat{S}' \rangle \wedge \widehat{S}' \sqsubseteq \widehat{S}_{i+1} \\ \tau_1 \neq \texttt{L}' \end{array}}{\widehat{\sigma}, \widehat{S}, i \vdash P}$$

[dup]
$$\frac{\begin{array}{c} P[i] = \texttt{dup} \wedge i+1 \in \textit{\textbf{Dom}}(P) \\ \widehat{\sigma}_i \sqsubseteq \widehat{\sigma}_{i+1} \wedge \widehat{S}_i = \langle \tau, \widehat{S}' \rangle \wedge \langle \tau, \tau, \widehat{S}' \rangle \sqsubseteq \widehat{S}_{i+1} \\ \tau \neq \texttt{L}' \end{array}}{\widehat{\sigma}, \widehat{S}, i \vdash P}$$

[dup_2]
$$\frac{\begin{array}{c} P[i] = \texttt{dup\_2} \wedge i+1 \in \textit{\textbf{Dom}}(P) \\ \widehat{\sigma}_i \sqsubseteq \widehat{\sigma}_{i+1} \wedge \widehat{S}_i = \langle \tau_1, \tau_2, \widehat{S}' \rangle \wedge \langle \tau_1, \tau_2, \tau_1, \tau_2, \widehat{S}' \rangle \sqsubseteq \widehat{S}_{i+1} \\ \tau_1 \neq \texttt{L}' \end{array}}{\widehat{\sigma}, \widehat{S}, i \vdash P}$$

Figure 3.4: Static semantics for JVML$_\mathcal{S}$ dataflow instructions.

PC	Instruction	StackDepth	Stack $\widehat{S}$						Vars $\widehat{\sigma}$	
			5	4	3	2	1	0	1	0
1	`lconst_0`	0					`L`	`L'`		
2	`lconst_1`	2			`L`	`L'`	`L`	`L'`		
3	`iconst_1`	4		`I`	`L`	`L'`	`L`	`L'`		
4	`ifeq` L	5			`L`	`L'`	`L`	`L'`		
5	`goto` 6	4			`L`	`L'`	`L`	`L'`		
6	`dup_2`	4	`L`	`L'`	`L`	`L'`	`L`	`L'`		
7	`ladd`	6			`L`	`L'`	`L`	`L'`		
8	`goto` 9	4			`L`	`L'`	`L`	`L'`		
9	L: `ladd`	4					`L`	`L'`		
10	`lstore_0`	2							`L`	`L'`

Figure 3.5: An example program and the abstraction for stack and variable states. Each instruction is labeled with the stack depth prior to its execution.

The evaluation of a program P starts with the initial rule

$$\frac{\begin{array}{c}\widehat{S}_1 = \langle\,\rangle \wedge \forall x \in \texttt{VARS} : \widehat{\sigma}_1[x] = \textit{undefined} \\ \forall i \in \textit{Dom}(P) : \widehat{\sigma}, \widehat{S}, i \vdash P\end{array}}{\widehat{\sigma}, \widehat{S} \vdash P}$$

The rules for the local judgments $\widehat{\sigma}, \widehat{S}, i \vdash P$ are given in Figure 3.3 (core instructions) and Figure 3.4 (dataflow instructions).

For `dup` and `pop`, the topmost stack cell must not be of type `L` as otherwise only one half of a long integer would be duplicated or deleted. It is not necessary to explicitly exclude `L`' here, as core instructions never leave a `L`' as topmost stack value. For `dup_2` and `pop_2`, two reference types or integers are allowed as topmost stack cells, or a properly formed long integer pair $(\texttt{L}, \texttt{L}')$.

Figure 3.5 shows a simple JVML$_S$ example. The right-hand part of the figure gives the abstractions computed for the states of stacks and variables.

Figure 3.6 and Figure 3.7 define the small-step operational semantics of JVML$_S$. Execution states are modeled as tuples $[\![pc, \sigma, S]\!]$, where the program counter pc is an element of `ADDR`, the state of local variables $\sigma \in \texttt{ENV}$, and the state of the operand stack

$$[\texttt{iconst\_}n] \quad \frac{P[pc] = \texttt{iconst\_}n}{P \vdash [\![pc, \sigma, \langle s\rangle]\!] \rightarrow [\![pc{+}1, \sigma, \langle n, s\rangle]\!]}$$

$$[\texttt{iadd}] \quad \frac{P[pc] = \texttt{iadd}}{P \vdash [\![pc, \sigma, \langle n_1, n_2, s\rangle]\!] \rightarrow [\![pc{+}1, \sigma, \langle (n_1 + n_2), s\rangle]\!]}$$

$$[\texttt{lconst\_}l] \quad \frac{P[pc] = \texttt{lconst\_}l}{P \vdash [\![pc, \sigma, \langle s\rangle]\!] \rightarrow [\![pc{+}1, \sigma, \langle (l, l'), s\rangle]\!]}$$

$$[\texttt{ladd}] \quad \frac{P[pc] = \texttt{ladd} \wedge (l_3, l_3') = (l_1, l_1') + (l_2, l_2')}{P \vdash [\![pc, \sigma, \langle (l_1, l_1'), (l_2, l_2'), s\rangle]\!] \rightarrow [\![pc{+}1, \sigma, \langle (l_3, l_3'), s\rangle]\!]}$$

$$[\texttt{ifeq } L] \quad \frac{P[pc] = \texttt{ifeq}\, L}{P \vdash [\![pc, \sigma, \langle 0, s\rangle]\!] \rightarrow [\![pc{+}1, \sigma, \langle s\rangle]\!]}$$

$$[\texttt{ifeq } L] \quad \frac{P[pc] = \texttt{ifeq}\, L \wedge n \neq 0}{P \vdash [\![pc, \sigma, \langle n, s\rangle]\!] \rightarrow [\![L, \sigma, \langle s\rangle]\!]}$$

$$[\texttt{goto } L] \quad \frac{P[pc] = \texttt{goto}\, L}{P \vdash [\![pc, \sigma, \langle s\rangle]\!] \rightarrow [\![L, \sigma, \langle s\rangle]\!]}$$

Figure 3.6: Dynamic semantics of JVML$_{\mathcal{S}}$ core instructions.

$S \in$ `STACK`. Execution starts from the state $[\![1, \sigma_1, \langle\rangle]\!]$. The initial mapping σ_1 maps all variables in `VARS` to $\bot$.[1] The execution stops as soon as a `return` instruction is reached.

3.3 Verification in Static Single Assignment form

The central responsibility of the Java bytecode verifier is to ensure that stack locations and local variables are used in a type-safe manner. This is the case if the definitions and uses of values have compatible types. To ensure this, the verifier algorithm has to determine the types of all stack locations and variables for each instruction in P.

[1]JVML$_{\mathcal{S}}$ can easily be extended to support JVML-style method arguments by initializing σ_1 accordingly. For simplicity, we exclude method arguments in this description.

[pop] $$\frac{P[pc] = \texttt{pop}}{P \vdash [\![pc, \sigma, \langle v, s\rangle]\!] \rightarrow [\![pc{+}1, \sigma, \langle s\rangle]\!]}$$

[pop_2] $$\frac{P[pc] = \texttt{pop\_2}}{P \vdash [\![pc, \sigma, \langle v_1, v_2, s\rangle]\!] \rightarrow [\![pc{+}1, \sigma, \langle s\rangle]\!]}$$

[dup] $$\frac{P[pc] = \texttt{dup}}{P \vdash [\![pc, \sigma, \langle v, s\rangle]\!] \rightarrow [\![pc{+}1, \sigma, \langle v, v, s\rangle]\!]}$$

[dup_2] $$\frac{P[pc] = \texttt{dup\_2}}{P \vdash [\![pc, \sigma, \langle v_1, v_2, s\rangle]\!] \rightarrow [\![pc{+}1, \sigma, \langle\ v_1, v_2, v_1, v_2, s\rangle]\!]}$$

[iload_x] $$\frac{P[pc] = \texttt{iload\_}x}{P \vdash [\![pc, \sigma, \langle s\rangle]\!] \rightarrow [\![pc{+}1, \sigma, \langle \sigma(x), s\rangle]\!]}$$

[lload_x] $$\frac{P[pc] = \texttt{lload\_}x}{P \vdash [\![pc, \sigma, \langle s\rangle]\!] \rightarrow [\![pc{+}1, \sigma, \langle\ (\sigma(x), \sigma(x+1)), s\rangle]\!]}$$

[istore_x] $$\frac{P[pc] = \texttt{istore\_}x}{P \vdash [\![pc, \sigma, \langle n, s\rangle]\!] \rightarrow [\![pc{+}1, \sigma[x \mapsto n], \langle s\rangle]\!]}$$

[lstore_x] $$\frac{P[pc] = \texttt{lstore\_}x}{P \vdash [\![pc, \sigma, \langle (l, l'), s\rangle]\!] \rightarrow [\![pc{+}1,\ \sigma[x \mapsto l, (x+1) \mapsto l'], \langle s\rangle]\!]}$$

Figure 3.7: Dynamic semantics of JVML$_{\mathcal{S}}$ dataflow instructions.

In JVML, there is no obvious link between the definition of a value and its uses. However, even if definition-use chains were available for each value in a JVML program, it would still be impossible to verify a Java program in a single pass by comparing the type of each definition with its uses.

The reason for this becomes more obvious if we consider how instructions are categorized in JVML$_{\mathcal{S}}$. In JVML$_{\mathcal{S}}$, only *core* instructions define and use values. *Dataflow* instructions merely facilitate the flow of values between core instructions. Core instructions are always *self-typed*, i.e. the expected type of each consumed operand and the type of any produced values is known statically.

In contrast, dataflow instructions are non-self-typed, i.e. they are polymorphic. It is not always possible to determine the type of the value produced by a dataflow instruction without knowing the type of its operands. The result type of a `dup` instruction, for example, depends on the type of the value on top of the stack. While local variable access instructions such as `iload_`x suggest stronger static typing, this works for scalar types only. In the JVM, object references are written and read from local variables using `astore_`x and `aload_`x, and data-flow analysis is still necessary to determine the precise type of the variables accessed.

We annotate JVML code so that the flow of values between core instructions becomes explicit instead of relying on an operand stack. By doing so, all non-self-typed instructions (dataflow instructions) can be eliminated from the code.

The final step is to transform the annotated JVML$_{\mathfrak{A}}$ format into SSA, which allows to check type safety by directly relating the type of definitions with the corresponding uses (*definition-use verification*).

In the remainder of this section, we develop an algorithm that performs definition-use verification in SSA form.

First, all reachable instructions are annotated with the depth of the stack before the instruction is executed. At the same time all dataflow instructions are resolved to `move` instructions. Next we compute the Iterative Dominance Frontier (IDF) for all stack cells and local variables and place ϕ-nodes in the control-flow graph (CFG) accordingly. In step 3 we traverse the CFG in dominator tree-order, assigning a unique name to all stack and local variable definitions and recording the type for each definition. `move` instructions are eliminated through copy propagation. Finally, in step 4 we merge the types in all ϕ-instructions, iterate over all instructions and match the expected operand types to the actual definition types. Whenever a type error occurs, verification fails.

Step 1 transforms the program so that the operand stack is no longer necessary. For this, each instruction is annotated with the depth of the incoming stack, so instructions no

$$\frac{P[pc] = \texttt{iadd} \wedge P_{\mathcal{A}}[pc] = \texttt{iadd}^{sd} \wedge P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc+1, sd-1\rangle}{P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc, sd\rangle}$$

$$\frac{P[pc] = \texttt{iconst\_}n \wedge P_{\mathcal{A}}[pc] = \texttt{iconst\_}n^{sd} \wedge P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc+1, sd+1\rangle}{P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc, sd\rangle}$$

$$\frac{P[pc] = \texttt{ladd} \wedge P_{\mathcal{A}}[pc] = \texttt{ladd}^{sd} \wedge P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc+1, sd-2\rangle}{P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc, sd\rangle}$$

$$\frac{P[pc] = \texttt{lconst\_}l \wedge P_{\mathcal{A}}[pc] = \texttt{lconst\_}l^{sd} \wedge P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc+1, sd+2\rangle}{P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc, sd\rangle}$$

$$\frac{P[pc] = \texttt{ifeq}\, L \wedge P_{\mathcal{A}}[pc] = \texttt{ifeq}\, L^{sd} \wedge P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc+1, sd-1\rangle}{P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc, sd\rangle}$$

$$\frac{P[pc] = \texttt{ifeq}\, L \wedge \texttt{SD}[L] \overset{?}{=} sd-1 \wedge P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle L, sd-1\rangle}{P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc, sd\rangle}$$

$$\frac{P[pc] = \texttt{goto}\ L \wedge P_{\mathcal{A}}[pc] = \texttt{goto}\ L \wedge \texttt{SD}[L] \overset{?}{=} sd \wedge P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle L, sd\rangle}{P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc, sd\rangle}$$

$$\frac{P[pc] = \texttt{return} \Rightarrow P_{\mathcal{A}}[pc] = \texttt{return}}{P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc, sd\rangle}$$

Figure 3.8: Transforming core instructions of JVML$_{\mathcal{S}}$ into JVML$_{\mathcal{A}}$.

longer depend on the stack to connect operands to their definitions.

We define a new map $\widehat{\texttt{R}}_i : \{\Sigma, \Lambda\} \times \mathbb{N} \rightarrow \mathcal{T}$ that for program point i maps stack cells respectively local variables to types, as well as its operational equivalent $\texttt{R}_i : \{\Sigma, \Lambda\} \times \mathbb{N} \rightarrow \mathcal{V}$ for values. Each $\texttt{R}_i$ and $\widehat{\texttt{R}}_i$ unify two maps—Λ maps a stack position to the value v and the type of v stored in that position. Accordingly, Σ maps a variable x to the value v and the type of v stored in x. We write Σ^{sd} for the topmost stack cell at program counter i and Σ^{sd-n} for the cell at offset $-n$.

We combine the maps $\texttt{R}_i$ and $\widehat{\texttt{R}}_i$ for all the program points to a vector $\texttt{R}$ and $\widehat{\texttt{R}}$.

Figure 3.8 and Figure 3.9 define the transformation from JVML$_{\mathcal{S}}$ into JVML$_{\mathcal{A}}$.

$$\frac{P[pc] = \texttt{pop} \wedge P_{\mathcal{A}}[pc] = \texttt{skip} \wedge P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc + \mathit{1}, sd - \mathit{1} \rangle}{P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc, sd \rangle}$$

$$\frac{P[pc] = \texttt{pop\_2} \wedge P_{\mathcal{A}}[pc] = \texttt{skip} \wedge P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc + \mathit{1}, sd - \mathit{2} \rangle}{P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc, sd \rangle}$$

$$\frac{\begin{array}{c} P[pc] = \texttt{dup} \wedge P_{\mathcal{A}}[pc] = \texttt{move}_1^{\epsilon} \left(\Lambda^{sd-1}\right) \left(\Lambda^{sd}\right) \wedge \\ P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc + \mathit{1}, sd + \mathit{1} \rangle \end{array}}{P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc, sd \rangle}$$

$$\frac{\begin{array}{c} P[pc] = \texttt{dup\_2} \wedge P_{\mathcal{A}}[pc] = \texttt{move}_2^{\epsilon} \left(\Lambda^{sd-2}, \Lambda^{sd-1}\right) \left(\Lambda^{sd}, \Lambda^{sd+1}\right) \wedge \\ P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc + \mathit{1}, sd + \mathit{2} \rangle \end{array}}{P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc, sd \rangle}$$

$$\frac{\begin{array}{c} P[pc] = \texttt{iload\_}x \wedge P_{\mathcal{A}}[pc] = \texttt{move}_1^{\epsilon} \left(\Sigma^{x}\right) \left(\Lambda^{sd}\right) \wedge \\ P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc + \mathit{1}, sd + \mathit{1} \rangle \end{array}}{P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc, sd \rangle}$$

$$\frac{\begin{array}{c} P[pc] = \texttt{lload\_}x \wedge P_{\mathcal{A}}[pc] = \texttt{move}_2^{\epsilon} \left(\Sigma^{x}, \Sigma^{x+1}\right) \left(\Lambda^{sd}, \Lambda^{sd+1}\right) \wedge \\ P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc + \mathit{1}, sd + \mathit{2} \rangle \end{array}}{P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc, sd \rangle}$$

$$\frac{\begin{array}{c} P[pc] = \texttt{istore\_}x \wedge P_{\mathcal{A}}[pc] = \texttt{move}_1^{sd-1} \left(\Lambda^{sd-1}\right) \left(\Sigma^{x}\right) \wedge \\ P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc + \mathit{1}, sd - \mathit{1} \rangle \end{array}}{P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc, sd \rangle}$$

$$\frac{\begin{array}{c} P[pc] = \texttt{lstore\_}x \wedge P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc + \mathit{1}, sd - \mathit{2} \rangle \wedge \\ P_{\mathcal{A}}[pc] = \texttt{move}_2^{sd-2, sd-1} \left(\Lambda^{sd-2}, \Lambda^{sd-1}\right) \left(\Sigma^{x}, \Sigma^{x+1}\right) \end{array}}{P, P_{\mathcal{A}}, \texttt{SD} \vdash \langle pc, sd \rangle}$$

Figure 3.9: Transforming dataflow instructions of JVML$_{\mathcal{S}}$ into JVML$_{\mathcal{A}}$.

Dataflow instructions are replaced with `move` instructions that transport values between the maps for stack and local variables.

The transformation is started with $P_{\mathfrak{A}}$ being undefined for all program counters, P being a JVML$_{\mathfrak{S}}$ method, $pc = 1$, and $sd = 0$. Instructions are visited depth first and for each basic block entry the field `SD` stores the stack depth. Initially, `SD` is only defined for the entry program counter ($\texttt{SD}[1] = 0$) and maps all other program counter values to undef. If a second jump to an already visited basic block is encountered, it is checked whether this stack depth is the same as those seen before. We define a special comparison operator $\overset{?}{=}$:

Definition 3.3.1 *($\overset{?}{=}$). Let $\overset{?}{=} \subseteq (\mathbb{N} \cup \{\mathit{undef}\} \times \mathbb{N}) \rightarrow \{\mathit{stop}, \mathit{err}, \mathit{true}\}$. The $\overset{?}{=}$ operator is defined by*

$$a \overset{?}{=} b = \begin{cases} \mathit{true} & \text{, if } a = \mathit{undef} \\ \mathit{stop} & \text{, if } a = b \\ \mathit{err} & \text{, if } a \neq b \end{cases}$$

In the case $a = \mathit{undef}$, as a side effect of the comparison a is assigned the value b.

The operator $\overset{?}{=}$ is used to guide the transformation. It ensures that basic blocks are only visited once by `SD` only being defined when the basic block has already been entered. If the basic block is visited again with the same stack depth, the comparison result is stop, causing the transformation to not visit the basic block again. If the comparison result is err a basic block has been visited with two differing stack depths and the transformation will fail.

While core instructions are annotated, dataflow instructions are replaced by the `move` instruction. Its semantics is to move values between the maps for stack cells and variables.

During the transformation all instructions are annotated with a possibly empty sequence of stack addresses $sd^1, \cdots, sd^i \in \mathbb{N}, i \geq 0$. These annotations are needed since the operational semantics of JVML$_{\mathfrak{A}}$ no longer involves a stack. The purpose of the annotation depends on the kind of instruction being annotated: for core instructions, the

PC		P	**StackDepth**	$P_{\mathcal{A}}$
1		`lconst_0`	0	`lconst_0`^{0}
2		`lconst_1`	2	`lconst_1`^{2}
3		`iconst_1`	4	`iconst_1`^{4}
4		`ifeq` L	5	`ifeq` L^{5}
5		`goto` 6	4	`goto` 6
6		`dup_2`	4	`move`$_{2}^{\epsilon}$ (Λ^2, Λ^3) (Λ^4, Λ^5)
7		`ladd`	6	`ladd`^{6}
8		`goto` 9	4	`goto` 9
9	L:	`ladd`	4	`ladd`^{4}
10		`lstore_0`	2	`move`$_{2}^{0,1}$ (Λ^0, Λ^1) (Σ^0, Σ^1)

Figure 3.10: The example program from Figure 3.5 before and after the transformation.

annotation is the stack depth computed for that instruction. This annotation allows the semantics of JVML$_{\mathcal{A}}$ to model the stack by accessing the correct entries in the Λ map. For the `move` instruction, the annotation is a (possibly empty) list of stack cells that need to be undefined in the static semantics, since they would have been popped from the operand stack in JVML$_{\mathcal{S}}$.

Instructions operating on long integers push and pop pairs onto and from the operand stack. E.g., `lload_`x^{sd} transfers two values from variable x to the stack. It is replaced by `move`$_{2}^{\epsilon}$ $(\Sigma^{x}, \Sigma^{x+1})$ $\left(\Lambda^{sd}, \Lambda^{sd+1}\right)$, indicating that the pair of values stored in the variable x (that is in slots x and $x+1$ of the Σ map) must be copied to the two consecutive stack cells sd and $sd+1$ (that is in the Λ map).

Figure 3.10 shows the result of transforming the example code from Figure 3.5 to JVML$_{\mathcal{A}}$. The core instructions have been annotated with the stack depth that they would be executed with, and dataflow instructions have been replaced with `move`, annotated with the stack cells that must be undefined in the static semantics. The last instruction moves the computed result from the stack map to the variable map.

Figure 3.11 and Figure 3.12 introduce the static and operational semantics of JVML$_{\mathcal{A}}$, respectively. The main difference to JVML$_{\mathcal{S}}$ is that the new semantics reasons about the $\widehat{\mathtt{R}}$

and R map instead of stack and local variables. In the static semantics, values that have been popped from the stack must be reset to $\perp$, to make sure that the comparison against the stack of the following instruction is still valid. E.g., while in the JVML$_{\mathcal{S}}$ semantics `iadd` implicitly pops its two arguments from the stack (c.f. Figure 3.3), the JVML$_{\mathfrak{A}}$ rule must use $\widehat{\mathtt{R}}_i[\Lambda^{sd-1} \mapsto \perp]$ instead.

Theorem 3.3.1 *The static semantics of JVML$_{\mathfrak{A}}$ is sound and correct with respect to the static semantics of JVML$_{\mathcal{S}}$.*

By simple inspection of the rules and checking of the dataflow between stack and local variables in JVML$_{\mathcal{S}}$ and between the Σ and the Λ map in $\widehat{\mathtt{R}}$ in JVML$_{\mathfrak{A}}$, it is obvious that the flow of values between definition and uses is equivalent. However, because dataflow instructions have been eliminated, some of the typing rules enforced by them no longer apply. E.g., the following JVML$_{\mathcal{S}}$ program will be rejected by the Java verifier, but is valid in JVML$_{\mathfrak{A}}$:

```
1: lconst_0
2: istore_1
3: iload_1
4: lstore_2
```

In Line 1 a long integer is pushed onto the stack as a pair of halves $(\mathtt{L}, \mathtt{L}')$. Partially storing the long integer in an integer register (Line 2) is rejected by the traditional verifier. In contrast, since our verifier does not consider the typing rules of data-flow instructions, it accepts this code fragment, because the $(\mathtt{L}, \mathtt{L}')$ pair pushed in Line 1 is restored on the stack before it is used in Line 4. It is important to note that this program, while rejected by the JVM, is perfectly safe when executed.

During the transformation into JVML$_{\mathfrak{A}}$, dataflow instructions have been replaced by $\mathtt{move}_n$ instructions that make the dataflow between stack and local variables explicit.

$$[\texttt{iconst\_}n] \quad \frac{P_{\mathcal{A}}[i] = \texttt{iconst\_}n^{sd} \wedge i{+}1 \in \textbf{\textit{Dom}}(P_{\mathcal{A}}) \qquad \widehat{\mathtt{R}}_i[(\Lambda, sd) \mapsto \mathtt{I}] \sqsubseteq \widehat{\mathtt{R}}_{i+1}}{\widehat{\mathtt{R}}, i \vdash P_{\mathcal{A}}}$$

$$[\texttt{lconst\_}l] \quad \frac{P_{\mathcal{A}}[i] = \texttt{lconst\_}l^{sd} \wedge i{+}1 \in \textbf{\textit{Dom}}(P_{\mathcal{A}}) \qquad \widehat{\mathtt{R}}_i[(\Lambda, sd) \mapsto \mathtt{L}, (\Lambda, sd{+}1) \mapsto \mathtt{L}'] \sqsubseteq \widehat{\mathtt{R}}_{i+1}}{\widehat{\mathtt{R}}, i \vdash P_{\mathcal{A}}}$$

$$[\texttt{iadd}] \quad \frac{P_{\mathcal{A}}[i] = \texttt{iadd}^{sd} \wedge i{+}1 \in \textbf{\textit{Dom}}(P_{\mathcal{A}}) \qquad \widehat{\mathtt{R}}_i(\Lambda^{sd-2}) = \widehat{\mathtt{R}}_i(\Lambda^{sd-1}) = \mathtt{I} \wedge \widehat{\mathtt{R}}_i[\Lambda^{sd-1} \mapsto \bot] \sqsubseteq \widehat{\mathtt{R}}_{i+1}}{\widehat{\mathtt{R}}, i \vdash P_{\mathcal{A}}}$$

$$[\texttt{ladd}] \quad \frac{P_{\mathcal{A}}[i] = \texttt{ladd}^{sd} \wedge i{+}1 \in \textbf{\textit{Dom}}(P_{\mathcal{A}}) \qquad (\widehat{\mathtt{R}}_i(\Lambda^{sd-4}), \widehat{\mathtt{R}}_i(\Lambda^{sd-3})) = (\widehat{\mathtt{R}}_i(\Lambda^{sd-2}), \widehat{\mathtt{R}}_i(\Lambda^{sd-1})) = (\mathtt{L}, \mathtt{L}') \qquad \widehat{\mathtt{R}}_i[\Lambda^{sd-2}, \Lambda^{sd-1}] \mapsto \bot] \sqsubseteq \widehat{\mathtt{R}}_{i+1}}{\widehat{\mathtt{R}}, i \vdash P_{\mathcal{A}}}$$

$$[\texttt{goto } L] \quad \frac{P_{\mathcal{A}}[i] = \texttt{goto } L \wedge L \in \textbf{\textit{Dom}}(P_{\mathcal{A}}) \qquad \widehat{\mathtt{R}}_i \sqsubseteq \widehat{\mathtt{R}}_L}{\widehat{\mathtt{R}}, i \vdash P_{\mathcal{A}}}$$

$$[\texttt{ifeq } L] \quad \frac{P_{\mathcal{A}}[i] = \texttt{ifeq } L^{sd} \wedge (i{+}1), L \in \textbf{\textit{Dom}}(P_{\mathcal{A}}) \qquad \widehat{\mathtt{R}}_i(\Lambda^{sd-1}) = \mathtt{I} \wedge \widehat{\mathtt{R}}_i[(\Lambda^{sd-1}) \mapsto \bot] \sqsubseteq \widehat{\mathtt{R}}_{i+1}, \widehat{\mathtt{R}}_L}{\widehat{\mathtt{R}}, i \vdash P_{\mathcal{A}}}$$

$$[\texttt{move}_n] \quad \frac{P_{\mathcal{A}}[i] = \texttt{move}_n^{sd^{1\cdots l, l\geq 0}}\ (x_1, \cdots, x_n)\ (y_1, \cdots, y_n) \wedge i{+}1 \in \textbf{\textit{Dom}}(P_{\mathcal{A}}) \qquad 1 \leq j \leq n : \widehat{\mathtt{R}}_i(y_j) = x_j \wedge \widehat{\mathtt{R}}_i[\Lambda_j^{sd}) \mapsto \bot]_{1\leq j<l} \sqsubseteq \widehat{\mathtt{R}}_{i+1}}{\widehat{\mathtt{R}}, i \vdash P_{\mathcal{A}}}$$

Figure 3.11: Static semantics of JVML$_{\mathcal{A}}$.

$$\frac{P_{\mathcal{A}}[pc] = \texttt{iconst\_}n^i}{P_{\mathcal{A}} \vdash \langle pc, \texttt{R} \rangle \rightarrow \langle pc{+}1, \texttt{R}[\Sigma^i \mapsto n] \rangle}$$

$$\frac{P_{\mathcal{A}}[pc] = \texttt{lconst\_}l^i}{P_{\mathcal{A}} \vdash \langle pc, \texttt{R} \rangle \rightarrow \langle pc{+}1, \texttt{R}[\Sigma^i \mapsto l, \Sigma^{i+1} \mapsto l'] \rangle}$$

$$\frac{P_{\mathcal{A}}[pc] = \texttt{iadd}^i}{P_{\mathcal{A}} \vdash \langle pc, \texttt{R} \rangle \rightarrow \langle pc{+}1, \texttt{R}[\Sigma^{i-2} \mapsto (\texttt{R}(\Sigma^{i-2}) + \texttt{R}(\Sigma^{i-1}))] \rangle}$$

$$\frac{P_{\mathcal{A}}[pc] = \texttt{ladd}^i \wedge (l, l') = (\texttt{R}(\Sigma^{i-4}) + \texttt{R}(\Sigma^{i-2}), \texttt{R}(\Sigma^{i-3}) + \texttt{R}(\Sigma^{i-1}))}{P_{\mathcal{A}} \vdash \langle pc, \texttt{R} \rangle \rightarrow \langle pc{+}1, \texttt{R}[\Sigma^{i-4} \mapsto l, \Sigma^{i-3} \mapsto l'] \rangle}$$

$$\frac{P_{\mathcal{A}}[pc] = \texttt{ifeq}\, L^i,\ \texttt{R}(\Sigma^{i-1}) = 0}{P_{\mathcal{A}} \vdash \langle pc, \texttt{R} \rangle \rightarrow \langle pc{+}1, \texttt{R} \rangle}$$

$$\frac{P_{\mathcal{A}}[pc] = \texttt{ifeq}\, L^i,\ \texttt{R}(\Sigma^{i-1}) \neq 0}{P_{\mathcal{A}} \vdash \langle pc, \texttt{R} \rangle \rightarrow \langle L, \texttt{R} \rangle}$$

$$\frac{P_{\mathcal{A}}[pc] = \texttt{goto}\ L}{P_{\mathcal{A}} \vdash \langle pc, \texttt{R} \rangle \rightarrow \langle L, \texttt{R} \rangle}$$

$$\frac{P_{\mathcal{A}}[pc] = \texttt{move}^{sd}_n\ (x_1, \cdots, x_n)\ (y_1, \cdots, y_n)}{P_{\mathcal{A}} \vdash \langle pc, \texttt{R} \rangle \rightarrow \langle pc{+}1, \texttt{R}[(y_i) \mapsto x_i] \rangle}, \text{ where } 1 \leq i \leq n,\ x_i, y_i \in \{\Sigma, \Lambda\} \times \mathbb{N}$$

Figure 3.12: Operational semantics of JVML$_{\mathcal{A}}$

After computing the Iterative Dominance Frontier IDF for all defined entries of $\widehat{\texttt{R}}$ (that is used variables and stack cells), the next step is to uniquely label all definitions of values.

Function DEFINE maintains a counter *values* (initialized to zero) and records the type of each definition in $type$:

```
DEFINE($t$)
1  $n$ = value++;
2  $type[n] \leftarrow t$;
3  return $n$;
```

$$\frac{\begin{array}{c} n = \mathrm{DEFINE}(\mathtt{I}) \\ \mathtt{V}[(\Lambda^{sd}) \mapsto n] \\ P_{\mathscr{SSA}}[i] = \mathtt{iconst\_}n^{sd} \end{array}}{\mathtt{OUT}, \mathtt{IN}, P_{\mathscr{A}}, P_{\mathscr{SSA}} \vdash \langle i, \mathtt{V} \rangle} \; P_{\mathscr{A}}[i] = \mathtt{iconst\_}n^{sd}$$

$$\frac{\begin{array}{c} \mathtt{IN}[i] = \{\mathtt{V}(\Lambda^{sd\text{–}2}), \mathtt{V}(\Lambda^{sd\text{–}1})\} \\ n = \mathrm{DEFINE}(\mathtt{I}) \\ \mathtt{V}[(\Lambda^{sd\text{–}2}) \mapsto n] \\ \mathtt{OUT}[i] = \{n\} \\ P_{\mathscr{SSA}}[i] = \mathtt{iadd}^{sd} \end{array}}{\mathtt{OUT}, \mathtt{IN}, P_{\mathscr{A}}, P_{\mathscr{SSA}} \vdash \langle i, \mathtt{V} \rangle} \; P_{\mathscr{A}}[i] = \mathtt{iadd}^{sd}$$

$$\frac{\begin{array}{c} n_1 = \mathrm{DEFINE}(\mathtt{L}); n_2 = \mathrm{DEFINE}(\mathtt{L}') \\ \mathtt{V}[(\Lambda^{sd}) \mapsto n_1, (\Lambda^{sd+1}) \mapsto n_2] \\ P_{\mathscr{SSA}}[i] = \mathtt{lconst\_}l^{sd} \end{array}}{\mathtt{OUT}, \mathtt{IN}, P_{\mathscr{A}}, P_{\mathscr{SSA}} \vdash \langle i, \mathtt{V} \rangle} \; P_{\mathscr{A}}[i] = \mathtt{lconst\_}l^{sd}$$

$$\frac{\begin{array}{c} \mathtt{IN}[i] = \{\mathtt{V}(\Lambda^{sd\text{–}4}), \mathtt{V}(\Lambda^{sd\text{–}3}), \mathtt{V}(\Lambda^{sd\text{–}2}), \mathtt{V}(\Lambda^{sd\text{–}1})\} \\ n_1 = \mathrm{DEFINE}(\mathtt{L}); n_2 = \mathrm{DEFINE}(\mathtt{L}') \\ \mathtt{V}[(\Lambda^{sd\text{–}4}) \mapsto n_1, (\Lambda^{sd\text{–}3}) \mapsto n_2] \\ \mathtt{OUT}[i] = \{n_1, n_2\} \\ P_{\mathscr{SSA}}[i] = \mathtt{ladd}^{sd} \end{array}}{\mathtt{OUT}, \mathtt{IN}, P_{\mathscr{A}}, P_{\mathscr{SSA}} \vdash \langle i, \mathtt{V} \rangle} \; P_{\mathscr{A}}[i] = \mathtt{ladd}^{sd}$$

Figure 3.13: Rename definition and uses of stack cells and variables (arithmetic instructions).

Each ϕ-node inserted into the program is assigned a new number by DEFINE. Since this phase is well understood we assume that our program $P_{\mathscr{A}}$ contains sufficient ϕ-nodes at the beginning of each basic block. Each ϕ-node is annotated with the entry of $\widehat{\mathtt{R}}$ it stands for and the unique label assigned by DEFINE.

Uses of variables are rewritten to reference the unique name assigned to the corresponding definition using three data structures (Figure 3.13 and Figure 3.14). $\mathtt{V} \subseteq \{\Sigma, \Lambda\} \times \mathbb{N} \times \mathbb{N}$ maps variables and stack cells to the unique name assigned to their last definition along the current path while traversing the CFG in dominator tree-order to perform SSA variable renaming. At the end of each basic block, the ϕ-nodes in all

$$\frac{\begin{array}{c}\mathtt{IN}[i] = \{\mathtt{V}(\Lambda^{sd-1})\}\\ \forall\phi\text{-instructions } p \text{ for entry } r \text{ at basic block } L\text{:}\,\mathtt{IN}[p] = \mathtt{IN}[p] \cup \mathtt{V}[r]\\ P_{\mathcal{SSA}}[i] = \mathtt{ifeq}\, L^{sd}\end{array}}{\mathtt{OUT}, \mathtt{IN}, P_{\mathcal{A}}, P_{\mathcal{SSA}} \vdash \langle i, \mathtt{V}\rangle}\; P_{\mathcal{A}}[i] = \mathtt{ifeq}\, L^{sd}$$

$$\frac{\begin{array}{c}\forall\phi\text{-instructions } p \text{ for entry } r \text{ at basic block } L\text{:}\,\mathtt{IN}[p] = \mathtt{IN}[p] \cup \mathtt{V}[r]\\ P_{\mathcal{SSA}}[i] = \mathtt{goto}\, L\end{array}}{\mathtt{OUT}, \mathtt{IN}, P_{\mathcal{A}}, P_{\mathcal{SSA}} \vdash \langle i, \mathtt{V}\rangle}\; P_{\mathcal{A}}[i] = \mathtt{goto}\, L$$

$$\frac{\begin{array}{c}\mathtt{V}' = \mathtt{V}\\ \forall 1 \leq j \leq n : \mathtt{V}(y_j) = \mathtt{V}'(x_j)\\ P_{\mathcal{SSA}}[i] = \mathtt{nop}\end{array}}{\mathtt{OUT}, \mathtt{IN}, P_{\mathcal{A}}, P_{\mathcal{SSA}} \vdash \langle i, \mathtt{V}\rangle}\; P_{\mathcal{A}}[i] = \mathtt{move}_n^{sd}\ (x_1, \cdots, x_n)\ (y_1, \cdots, y_n)$$

$$\frac{\begin{array}{c}\mathtt{V}[(\mathit{entry}) \mapsto \mathit{label}]\\ P_{\mathcal{SSA}}[i] = \phi_{\mathit{label}}^{\mathit{entry}}\end{array}}{\mathtt{OUT}, \mathtt{IN}, P_{\mathcal{A}}, P_{\mathcal{SSA}} \vdash \langle i, \mathtt{V}\rangle}\; P_{\mathcal{A}}[i] = \phi_{\mathit{label}}^{\mathit{entry}}$$

Figure 3.14: Rename definition and uses of stack cells and variables (branches and data flow instructions).

successor blocks in the CFG are updated with the current name of the ϕ-operands. `IN` $\subseteq \mathbb{N} \times \wp(\mathbb{N})$ assigns to each instruction i the ordered set of unique names of definitions that i uses. `OUT` $\subseteq \mathbb{N} \times \wp(\mathbb{N})$ assigns to each instruction i the ordered set of unique names of definitions that i produces. `move` instructions are eliminated through copy propagation. For core instructions, the unique name of their operands is tracked in `IN` and the name of any defined values in `OUT`.

By retracing the effect of `move` instructions on values stored in local variables and stack cells, this step also performs copy propagation. After having performed the necessary assignments, `move` instructions are deleted from the program, transforming it into a JVML$_{\mathcal{SSA}}$ program.

The numbering scheme used by DEFINE is crucial for the identification of proper use of long integers. E.g., for the `lconst_`l^{sd} instruction, DEFINE is invoked twice consecutively, resulting in the two partial long values being labeled with two consecutive

```
ResolvePhi$(w, visited)$
1  $t \leftarrow \bot;$
2  $\forall x \in \mathtt{IN}[w], \neg\textsc{IsPhi}(x)$ :
3      $t \leftarrow t \sqcup \mathit{type}[x];$
4  $\mathit{visited} = \mathit{visited} \cup \{w\};$
5  $\forall x \in \mathtt{IN}[w], \textsc{IsPhi}(x) \wedge x \notin \mathit{visited}$
6      $t \leftarrow t \sqcup \textsc{ResolvePhi}(w, visited)$
7  return $t$;

Check$(P)$
1  $\forall\phi$-nodes $w$
2      $\mathit{type}[w] = \textsc{ResolvePhi}(w, \emptyset)$
3  $\forall i \in P_{SS\mathcal{A}}$
4      $\forall \mathit{arguments}\ a_j\ \mathit{of}\ i$
5          $(\mathit{type}[\mathtt{V}(a_j)] \sqcup \mathit{StaticType}[j]) \in \{\bot, \top\}$
6            $\Rightarrow \textsc{Fail}$("Type mismatch.")
7          $\mathit{type}[\mathtt{V}(a_j)] = \mathtt{L} \wedge \mathtt{IN}[i]_j \neq \mathtt{IN}[i]_{j+1}$
8            $\Rightarrow \textsc{Fail}$("Invalid L integer.")
```

Figure 3.15: Simplified algorithm for definition-use verification.

numbers. When the code is checked for type correctness, this property allows to check that two long integer halves were actually defined together.

Figure 3.16 shows the state of IN, OUT, V and *type* for the example code from Figure 3.10. In contrast to JVML$_{\mathcal{A}}$, basic blocks now contain ϕ-nodes. The `move` instructions have been replaced by `nop` instructions and are only shown for completeness.

The next phase is to determine the type for each definition in $P_{SS\mathcal{A}}$. The actual type-checking is performed by function CHECK (frefproofing-code4). It determines the type for each definition in $P_{SS\mathcal{A}}$. At this stage, all instructions are self-typed, except for ϕ-nodes. The return type of ϕ-nodes depends on the type of their operands, which in turn can be ϕ-nodes again. The return type of ϕ-nodes is resolved through a depth-first search in RESOLVEPHI.

Next, the remaining core instructions are visited and the definition type of their uses is match to the expected operand type. For each instruction i the function *StaticType* : $\mathbb{N} \rightarrow \mathcal{T}$ gives the expected type of its operands. The definition type of each

PC	**code**	IN	OUT	V	*type*$[v]$
1	`lconst_0`^{0}		2,3	$\Lambda^0 \to 2$ $\Lambda^1 \to 3$	$0 \to \bot, 1 \to \bot$ $2 \to \mathtt{L}, 3 \to \mathtt{L}'$
2	`lconst_1`^{2}		4,5	$\Lambda^2 \to 4$ $\Lambda^3 \to 5$	$4 \to \mathtt{L}, 5 \to \mathtt{L}'$
3	`iconst_1`^{4}		6	$\Lambda^4 \to 6$	$6 \to \mathtt{I}$
4	`ifeq` L^5	6			
5	`goto` 6				
6	$\mathtt{move}_2^{\epsilon}$ (Λ^2, Λ^3) (Λ^4, Λ^5)			$\Lambda^4 \to 4$ $\Lambda^5 \to 5$	
7	`ladd`^{6}	4,5,4,5	7,8	$\Lambda^2 \to 7$ $\Lambda^3 \to 8$	$7 \to \mathtt{L}, 8 \to \mathtt{L}'$
8	`goto` L				
9	L:$\phi_0^{\Sigma^2}$	4,7	0	$\Lambda^2 \to 0$	$0 \to \mathtt{L}$
10	$\phi_1^{\Sigma^3}$	5,8	1	$\Lambda^3 \to 1$	$1 \to \mathtt{L}'$
11	`ladd`^{4}	2,3,0,1	9,10	$\Lambda^0 \to 9$ $\Lambda^1 \to 10$	$9 \to \mathtt{L}, 10 \to \mathtt{L}'$
12	$\mathtt{move}_2^{0,1}$ (Λ^0, Λ^1) (Σ^0, Σ^1)			$\Sigma^0 \to 9$ $\Sigma^1 \to 10$	

Figure 3.16: `IN`, `OUT`, and `V` for the example code from Figure 3.10.

operand a_j is matched to the expected static operand type (*StaticType*$[j]$) of that instruction. The additional condition *type*$[\mathtt{V}(a_j)] = \mathtt{L} \wedge \mathtt{IN}[i]_j \neq \mathtt{IN}[i]_{j+1}$ guarantees that halves of long integers are used consistently and no separately defined long integers are combined.

3.4 Exceptions, Arrays, and Object Initialization

JVML$_S$ does not model exceptions. However, extending our verifier algorithm to support exceptions is straightforward. In JVML, exception handlers are regular code fragments. A list of exception handlers specifies what code areas a particular handler guards. If an exception of matching type is thrown, control is transfered to the handler, otherwise it is rethrown in the outer scope. While the stack is cut if an exception occurs, exception handlers can access local variables. Thus, to ensure that the Iterative Dominance Frontier is calculated properly along regular control flow edges as well as exception edges, instructions that can throw exceptions have to be connected to all potential exception handlers within the same method. This means that during stack depth annotation and for the calculation of the dominator tree, all instructions that can throw exceptions have to be treated as implicit branch instructions. As the JVML specification does not guarantee that code regions guarded by an exception handler must contain instructions that can actually raise that kind of exception, the resulting CFG is not always fully connected. This has to be considered during calculation of the dominator tree.

JVML uses the `aaload` instruction to load values from an array. Multi-dimensional arrays are modeled as arrays of array objects. For example, a number is read from a two-dimensional integer array ($int[][]$) in two steps. An `aaload` instruction returns a reference to an array of integers ($int[]$) and an `iaload` instruction returns the actual value of type int. While `aaload` and `iaload` obviously are core instructions, they do not fit the definition of JVML$_{\mathcal{S}}$ core instructions, because they are non-self-typed. With a small extension to our verifier algorithm, however, it is possible to support the proper array semantics of JVML.

Programs in JVML$_{\mathcal{SSA}}$ allow to determine the precise type of each array instruction using a simple depth-first search. Starting at any `aaload` instruction we track its array operand to its definition, which is trivial due to the SSA-form and the copy propagation performed in Step 3. If the definition points to another `aaload` instruction, its return type has to be resolved first. Otherwise, the return type of the original `aaload` instruction can be derived from the type of the definition that it refers to. After visiting all `aaload` instructions, they are annotated with their precise return type and become self-typed. It is always possible to resolve such chains of `aaload` instructions, as typing rules forbid any circular flow of array operands between `aaload` instructions.

Another difficulty for Java bytecode verification is to ensure proper object

	# of methods	**method size [avg / max]**	**stack depth [avg / max, cells]**	**local variables [avg / max, bytes]**
java/*	6490	41.36 / 4065	2.74 / 14	2.47 / 37
java/io	1213	38.12 / 1295	2.39 / 8	2.35 / 15
java/lang	1336	38.41 / 4065	2.32 / 10	2.17 / 37
java/math	405	72.67 / 3041	3.16 / 8	3.73 / 29
java/nio	2096	26.80 / 417	3.05 / 11	2.31 / 15
java/util	2359	49.21 / 2916	2.64 / 14	2.62 / 25

Table 3.1: Characteristics of the test set used to compare the run time of the SSA-based verifier with the run time of the traditional verifier.

initialization. Traditionally an alias analysis is performed during the data-flow analysis to make sure that all object allocated with `new` are subsequently initialized using the appropriate constructor.

Our verifier uses a different approach. We extend the static semantics of `new` to produce two return values: the regular return value, which is a reference to the newly allocated object, and an *initialization guard*. Guard variables are void in the dynamic semantics and do not incur any runtime overhead. In the renaming array `V`, this guard variable initially maps to $\perp$, indicating that it has not been defined yet. If a constructor call is encountered, a new definition is entered in `V` for the associated guard variable. The static semantics for all instructions operating on object references enforce that the guard associated with their operand must be undefined in `V`.

To accommodate partial initializations, for example only along one case of an *if/else* branch, during renaming the definition of guard variables is implicitly reverted to $\perp$ in the iterative dominance frontier of the constructor invocation instruction.

3.5 BENCHMARKS

To evaluate the performance of our SSA-based verifier we have implemented a prototype verifier based on the algorithm presented in this paper. The prototype is able to verify the entire JVML, except for subroutines. On the one hand, the subroutine construct in JVML

	verify (DFA)	verify (SSA)	DOM	IDF
java/*	145.92	124.03	12.31	31.86
java/io	28.20	24.41	2.41	5.90
java/lang	27.98	26.20	2.49	6.90
java/math	17.11	12.46	0.99	3.42
java/nio	31.32	28.78	3.07	7.25
java/util	66.90	53.48	4.74	12.83

Table 3.2: Performance comparison of the traditional verifier and our SSA-based verifier using parts of the Java Runtime Libraries (JDK) as the test sets.

is obsolete and will probably be removed in future versions of the Java virtual machine. On the other hand, our current algorithm depends on the fact that the control-flow graph can be recovered quickly for JVML code. In the presence of subroutines, this is not always the case as returning edges from subroutines are not explicit.

We compare the total run time of our SSA-based verifier to the run time of the DFA-based verifier found in Sun's Hotspot Virtual Machine v5. For this benchmark, only the actual method verification time is considered. Disk I/O and class file parsing are not measured. Unfortunately, there is currently no established set of benchmarks to test the performance of verifiers. Benchmark suites such as SPECjvm are designed to evaluate the performance of code execution, *not code verification*. Thus, we have decided to use various parts of the Java Runtime Libraries (JDK 1.4.2) as the test set (Table 3.1). Measurements were conducted on a Pentium4 2.53GHz CPU with 512MB of RAM, running under RedHat Linux 9.

Table 3.2 and Figure 3.17 compare the total run time of the traditional DFA-based verifier with our SSA-based verifier. Verification in SSA-form is approximately 15% faster than the traditional algorithm when comparing the total run time. Not considering the time spent to calculate DOM and `IDF`, SSA-based verification is approximately 45% faster.

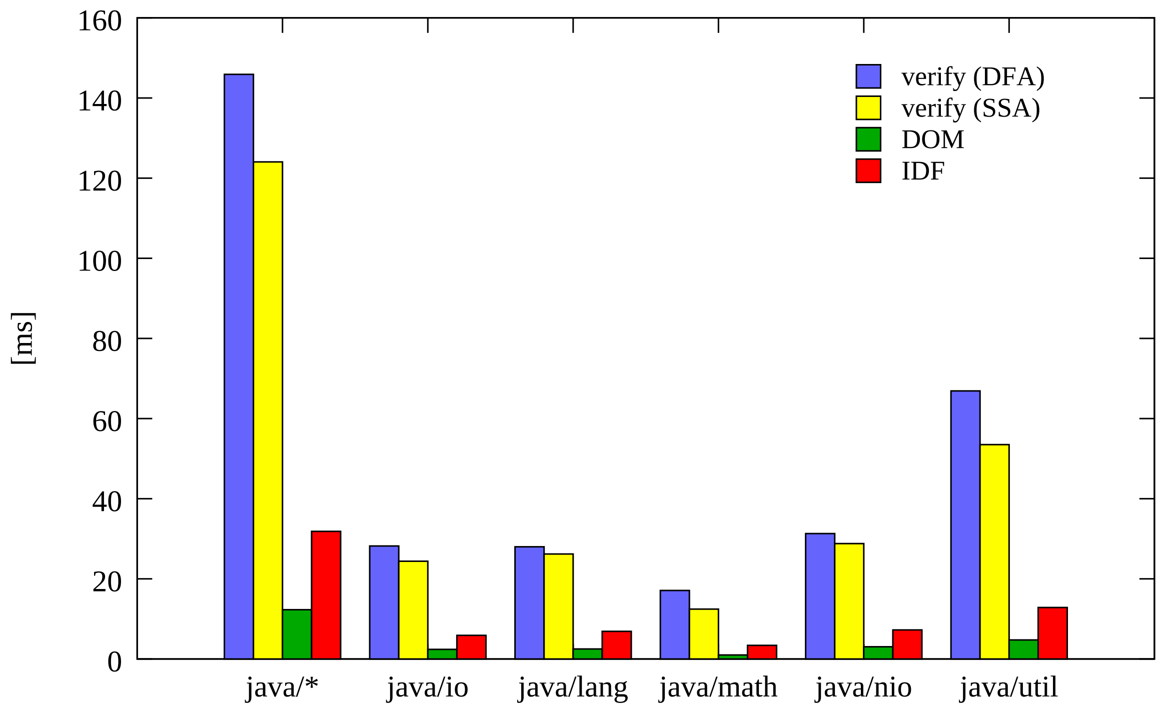


Figure 3.17: Performance comparison of the traditional verifier and our SSA-based verifier using parts of the Java Runtime Libraries (JDK) as the test sets.

3.6 Discussion

Existing JVML verifiers perform substantial data-flow analysis but do not preserve the results of this analysis for subsequent code generation and optimization phases. We have presented an alternative verifier that not only is faster than the standard Java verifier, but that additionally also computes the dominator tree and brings the program into Static Single Assignment form. As a result, the respective computations need not be repeated in subsequent stages of the dynamic compilation pipeline.

In the larger context of verifiable mobile code, our results indicate that verification should not be practiced in isolation "up front," but integrated with the rest of the client-side mobile code pipeline. Hence, we expect our approach to be applicable to other mobile-code systems besides the JVM, such as Microsoft's .NET platform.

When considering the overall complexity of the proposed verifier, it is apparent that verification complexity (and implicitly cost) is distributed very unevenly. Straight-line code is trivial to verify since each instruction has exactly one predecessor and thus the verifier state can be easily updated in a single step. Dealing with control-flow merges where instructions have several possible predecessors is much more complex, and it necessitates an iterative data-flow analysis in case of the standard SUN verifier, or the use of Static Single Assignment form in case of our verifier.

In the next chapter we will propose a dynamic compiler that eliminates control-flow merges from the program through a novel use of trace scheduling. By doing so we can exploit the advantage of Static Single Assignment form for program optimization and code generation where transforming into SSA is cheap (sequential code), without having to deal with the complexity of control flow merges (where constructing SSA is difficult).

CHAPTER 4

TRACE-BASED JUST-IN-TIME COMPILATION

In this chapter we present an approach for dynamic code generation in a virtual machine. While traditional just-in-time compilers are modeled after static compilers and use a control flow graph-based intermediate representation, our system uses a dynamic program representation based on a tree-shaped data structure that is discovered and updated lazily on-demand while the program is executed by the virtual machine. The resulting representation consists of a series of frequently executed bytecode traces (which may span several basic blocks across several methods), and our dynamic compiler compiles them to executable machine code using a specialized variant of Static Single Assignment (SSA) form. Our novel use of SSA form in this context allows using SSA-form based code optimization at much lower cost than traditional control flow graph-based SSA form, making our system suitable for embedded systems where memory is a scarce resource. The overall memory consumption (code and data) of our system is only 150 kBytes, yet benchmarks show a speedup that in some cases rivals heavy-weight just-in-time compilers.

4.1 MOTIVATION

Traditional dynamic compilers are often not much different from static compilers, except for being invoked at runtime, and thus many optimizing just-in-time compilers use the same underlying static intermediate representation as their static counterparts: annotated control-flow graphs.

To translate a bytecode program to machine code, the compiler first discovers the control-flow graph through basic block recovery analysis, and—in the case of an optimizing compiler—, performs a series of data-flow analyses such as obtaining definition/use information and determining the liveness of variables.

In [GPF06a] we have shown that most of these analyses have in common that they are cheap for simple linear code sequences, but expensive for control-flow graphs because the analysis result along multiple incoming edges to a basic block have to be reconciled.

Consider Static Single Assignment (SSA) form [CFR$^+$91], for example. Transforming a program into SSA (which is a step performed by many optimizing compilers) is trivial, except for basic blocks where several control-flow paths merge since ϕ-instructions have to be inserted into these blocks. Determining the location of these ϕ-instructions is the main reason transforming into SSA form is so costly. In a representation where control-flow never merges, transformation into SSA form could be done at much less cost, and with a much less complex implementation.

For static compilers, compilation cost is not necessarily a significant issue. The programer can wait a reasonable amount of time for the program to be translated to machine code. In a virtual machine (VM) setting, however, a slow compiler creates a noticeable compilation latency, which negatively impacts overall performance.

This problem is even more critical in embedded virtual machines, which have significantly less resources available for the compilation task as desktop systems. Compilers for such embedded VMs must be small and fast. Traditional SSA-based optimizing compilers are neither.

We propose a novel intermediate representation that is specifically geared towards the properties of mixed-mode interpretation and dynamic code generation. In contrast to the traditional control-flow graph, our representation is discovered and extended dynamically at runtime, which eliminates the need for static analysis to discover the control-flow graph before a bytecode region can be compiled to native code.

Our representation is also a more natural abstraction of the program structure, containing only those control-flow edges that are actually relevant at runtime, whereas control-flow graphs contain all possible edges, no matter how infrequent or irrelevant an edge might be. Because we represent only relevant edges, our representation can be compiled to native code faster, and with less effort, since the compiler is not burdened with the added constraints of "irrelevant" code blocks and control-flow edges between them.

In our novel representation, we represent performance-critical ("hot") areas of a program as a set of related traces. Through its trie-like shape, our representation eliminates control-flow merges except for returning loop edges to the anchor node, which dramatically simplifies program analysis since expensive control-flow merges are rare and all follow the same rigid structure. SSA form, for example, can be constructed without having to calculate the location of ϕ-instructions. The rigid shape of our representation prescribes that they can only appear in nodes that are an immediate predecessor of the anchor node.

In Section 4.2 we introduce the traditional control-flow graph model which was initially coined in the static compilation context, but is also used for dynamic compilers. Section 4.3 discusses our intermediate representation, and its construction. Section 4.3.2 describes the on-demand extension of trace-trees as more traces are discovered. Compiling trace-trees is explained in Section 4.4. In Section 4.5.2 we discuss our prototype implementation of a dynamic compiler based on trace-trees and show the trace-trees it generates for a series of benchmark programs. The chapter ends with a discussion of the presented research.

4.2 THE TRADITIONAL CONTROL FLOW GRAPH MODEL

The traditional control flow graph model represents a program as $G = (\mathcal{B}, \mathcal{E})$ where G is a *directed graph*, $\mathcal{B}$ is the set of *basic blocks* $\{b_1, b_2, \ldots, b_n\}$ in G, and $\mathcal{E}$ is a set of *directed edges* $\{(b_i, b_j), (b_k, b_l), \ldots\}$. Figure 4.1 shows the graphical representation of such a graph. Since *methods* can be understood as sub-programs, we can use the terms program and methods interchangeably in this context.

Each basic block $b \in \mathcal{B}$ is a linear sequence of instructions. A basic block is always entered at the top (first instruction), and always continues until the last instruction is reached. After executing all instructions in a basic block b_i, execution continues with an *immediate successor block* of b_i.

The existence of a direct edge from b_i to such a successor block b_j is indicated through an ordered pair (b_i, b_j) of nodes. Note that blocks can succeed themselves (a tight loop consisting of a single basic block), and thus the elements of said pair can be identical.

The set of all immediate successor blocks of a block b_i is characterized through a successor function $\Gamma_G^1(b_i) = \{b_j | (b_i, b_j) \in \mathcal{E}\}$, and it can be empty only for the *terminal node* $x \in \mathcal{B}$, which terminates the program: $\Gamma_G^1(x) = \emptyset$.

The set of *immediate predecessors* of b_j (the set of basic blocks that can branch to b_j) is characterized through the inverse of the successor function: $\Gamma_G^{-1}(b_j) = \{b_i | (b_i, b_j) \in \mathcal{E}\}$. It can be empty only for the *entry node* $e \in \mathcal{B}$, which is the first block executed when running P: $\Gamma_G^{-1}(e) = \emptyset$.

A *path* P along edges of a graph G can be expressed a sequence of nodes $(b_1, b_2, \ldots, b_n)$ of G. Each node in the sequence is an immediate successor of the predecessor node in the sequence: $b_{i+1} \in \Gamma_G^1(b_i)$. A path does not have to consist of distinct blocks and can contain the same blocks (and implicitly the same edges) repeatedly.

Figure 4.1 shows a sample program consisting of an `if-then-else` condition inside a `do/while` loop and the corresponding control flow graph with $\mathcal{B} = \{1, 2, 3, 4, 5, 6\}$ and edges $\mathcal{E} = \{(1,2), (2,3), (2,4), (3,5), (4,5), (5,6)\}$. In this example, both $(1,2,3)$ and $(1,2,4)$ are valid paths, and so is $(1,2,3,5,2,3)$ since paths are allowed to contain the same node multiple times. $(1,2,5)$ on the other hand is not a valid path, because $(2,5) \notin \mathcal{E}$.

A *cycle* C in a control flow graph is a path $(b_1, b_2, \ldots, b_n)$ where $b_1 = b_n$. In Figure 4.1, $(2,3,5,2)$ and $(2,4,5,2)$ are both valid cycles, and so is $(2,3,5,2,3,5,2)$.

```
1:  code;
2:  do {
        if (condition) {
3:          code;
        } else {
4:          code;
        }
5:  } while (condition);
6:  code;
```

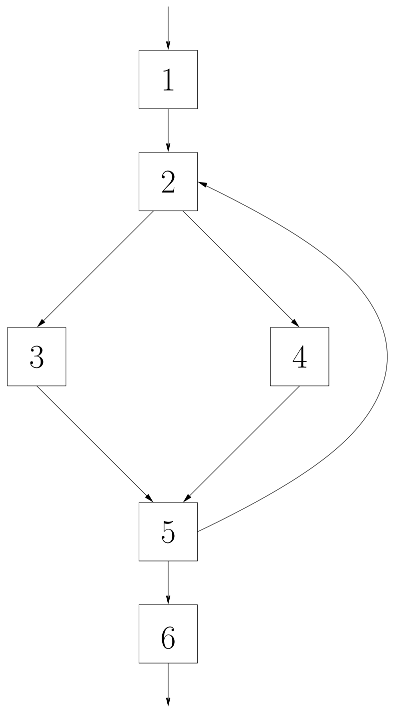


Figure 4.1: A sample program and its corresponding control flow graph. The starting point of each basic block is indicated through a corresponding label in the program code.

Cycles in a control flow graph correspond to loops in the original program. $(2, 3, 5, 2)$ and $(2, 4, 5, 2)$ are in fact two different cycles through the same loop and the `if-then-else` construct in it.

For the purpose of this paper, we will assume that no unconnected basic blocks exist in control flow graphs. This means that for every node b_i of a graph G there exists a valid path P of the form $(e, \ldots, b_i)$ that leads from the entry node e to b_i.

4.3 Representing Partial Programs as Trace Trees

A *trace tree* $TT = (\mathcal{N}, \mathcal{P})$ is a directed graph representing a set of related cycles (*traces*) in the program, where $\mathcal{N}$ is a set of nodes (*instructions*), and $\mathcal{P}$ is a set of directed edges $\{(n_i, n_j), (n_k, n_l), \ldots\}$ between them. Each node $n \in \mathcal{N}$ is labeled with an *operation* from the set of valid operations defined by the (virtual) machine language we are compiling. For the purpose of this paper it is sufficient to consider two operations: *bc* and *op*, with *bc* representing conditional branch operations (for example `ifeq` or `lookupswitch` in case of JVML [LY96]), and *op* representing all other non branching operations.

Each directed edge (n_i, n_j) in $\mathcal{P}$ indicates that instruction n_j is executed immediately before instruction n_i executes. Thus, we also refer to an edge $(n_i, n_j) \in \mathcal{P}$ as *predecessor edge*.

As we did for the control-flow graph, we define a successor function $\Gamma^1_{TT}(n_j) = \{n_i | (n_i, n_j) \in \mathcal{P}\}$, which returns all instructions that have a predecessor edge pointing to n_j, and thus can execute immediately *after* instruction n_j. Instructions labeled with *op* have at most one successor instruction: $\forall n \in \mathcal{N} \wedge \texttt{label}(n) = \textit{op} : |\Gamma^1_{TT}(n)| \leq 1$. These correspond to non-branching instructions in the virtual machine language (such as *add* or *mul*). Instructions labeled with *bc* can have an arbitrary number of successor instructions, including no successors at all. This is the case because edges are added to the trace tree lazily as it is built. If only one edge is ever observed for a conditional branch, all other edges never appear in the trace tree.

A node can have no successors if the control flow is observed to always return to the anchor node a (defined below), since this back-edge is considered implicit and does not appear in the predecessor edge set $\mathcal{P}$. Such instructions that have an implicit back-edge to a are called *leaf* nodes, and we denote the set of leaf nodes of a trace tree TT as *leaf set* $\mathcal{L}$. Leaf nodes can be branching (*bc*) or non-branching (*op*). A non-branching leaf node implies an unconditional jump back to the anchor node following the leaf node. A branching leaf node corresponds to a conditional branch back to the anchor node.

The predecessor function $\Gamma^{-1}_{TT}(n_i) = \{n_j | (n_i, n_j) \in \mathcal{P}\}$ returns the set of instructions an instruction n_i has predecessor edges pointing to. There is exactly one node a such that $\Gamma^{-1}_{TT}(a) = \emptyset$, and we called it the *anchor node*. All other nodes have exactly one

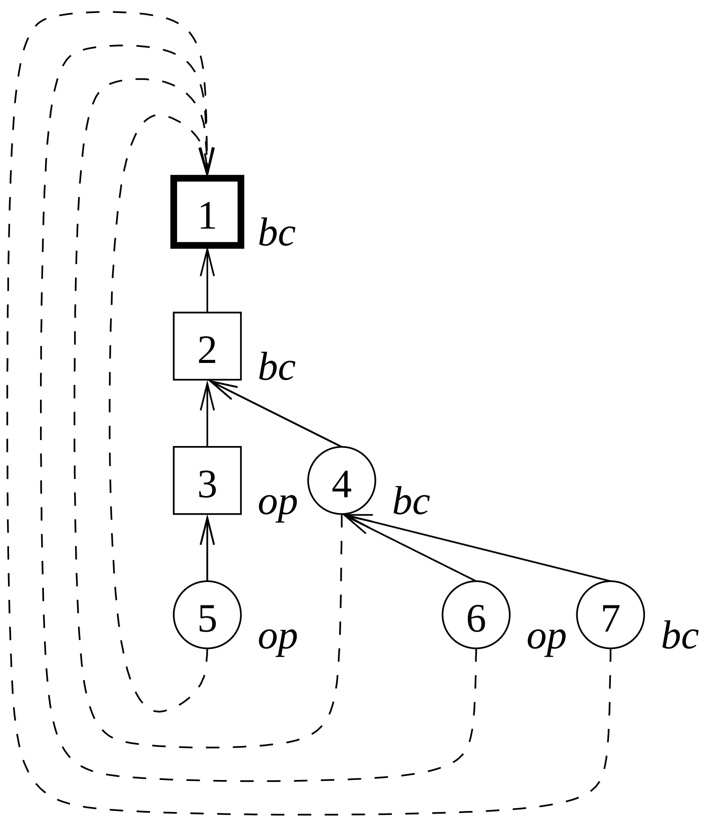


Figure 4.2: A sample trace tree.

predecessor node, and thus the predecessor function returns a set containing exactly one node: $\forall n \in \mathcal{N} \wedge n \neq a : |\Gamma_T^{-1}(n)| = 1$. This gives the directed graph the structure of a *directed rooted tree*, with the anchor node a as its root.

Each leaf node of the trace tree represents the last node in a cycle of instructions that started at the anchor node and ends at the anchor node, and we call each of these cycles a *trace*. The anchor node is shared between all traces, and the traces split up in a tree-like structure from there.

Figure 4.2 shows the visual representation of an example trace tree with the nodes $\mathcal{B} = \{1, 2, 3, 4, 5, 6, 7\}$ and predecessor edges $\mathcal{P} = \{(2, 1), (3, 2), (4, 2), (5, 3), (6, 4), (7, 4)\}$. Only solid edges are true predecessor edges. Dashed edges denote implicit back-edges, and are not part of the trace tree. The anchor node of the trace tree is 1 and is shown in bold. Leaf nodes $\mathcal{L} = \{4, 5, 6, 7\}$ are drawn as circles. Each instruction is labeled with its operation (*bc* or *op*).

4.3.1 Constructing a Trace Tree

In contrast to control flow graphs, our IR does not require complete knowledge of all nodes (basic blocks) and directed edges between them. Thus, no static analysis of the program is performed. Instead, our IR is constructed and extended lazily and on demand while the program is being executed by an *interpreter*.

Once a *trace graph* has been constructed, and every time it is extended, the graph is compiled (or recompiled) and subsequent executions of the covered program region are handed off to the compiled code instead of interpreting it.

Since loops often dominate the overall execution time of programs, the trace tree data structure is designed to represent loop regions of programs, and loop regions of programs only.

The first step to constructing a trace graph is locating a suitable *anchor node* $a \in \mathcal{B}$. Since we are interested in loop regions, *loop headers* are ideally suited as anchor nodes since they are shared by all cycles through the loop.

To identify loop headers, we use a simple heuristic that first appeared in Dynamo [BDB00], a framework for dynamic runtime optimization of binary code.

Initially, our virtual machine interprets the bytecode program instruction by instruction. Each time a backwards branch instruction is executed, the destination of that jump is a potential loop header. The general flow of control in bytecode is forward, and thus each loop has to contain at least one backward branch.

To filter *actual* loop headers from the superset of *potential* loop headers, we track the invocation frequency of (potential) loop headers. After the execution frequency of a potential loop header exceeds a certain threshold, our VM marks the instruction as an anchor node and will start recording bytecode instructions. Recording stops when a cycle is found, and the resulting trace is added to the tree.

Not all instructions and edges are suitable for inclusion in a trace tree. Exception edges, for example, indicate by their very semantics an exceptional (and usually rare) program state. Thus, whenever an exception occurs while recording a trace, the trace is voided, the trace recorder is disengaged and regular interpretation resumes.

Similarly, certain expensive instructions abort trace recording. Memory allocation instructions, for example, often take hundreds of cycles to execute, and thus the cost of the actual memory allocation operation by far exceeds the potential runtime savings that can be realized through compiling that code region to native code. This is often not a real limitation for performance critical code, since programmers are aware of the cost of memory allocation and tend to use pre-allocated data structures in-place in performance critical loops.

The third possible abort condition for trace recording is an overlong trace. This is necessary because in theory it would be possible to cover the entire program within a single trace tree, at the expense of creating a huge trace tree that would grow exponentially. This is clearly not what we strive for, and thus we limit traces to a certain length before they must either cycle back to the anchor node or be aborted.

Figure 4.3 shows a potential initial trace that could be recorded for the control flow graph in Figure 4.1. The trace starts at node 2, which is the anchor node for this new trace tree (which currently consists of a single trace). An alternative initial trace would have been $(2, 4, 5)$ instead of $(2, 3, 5)$ which is shown in the figure, since both start at the same anchor node $a = 2$. Which trace is recorded first solely depends on the conditional branch following the anchor node. If any particular cycle dominates a loop (i.e. the same path is taken through a loop most of the time), statistically the first recorded trace is likely to be that dominating cycle.

Nodes in Figure 4.3 labeled with *s* symbolize potential *side exits* from the trace tree. A side exit is a path originating from a node in the trace tree that is not covered by the trace tree itself. Every branching node (*bc*) that has fewer incoming edges in the trace graph

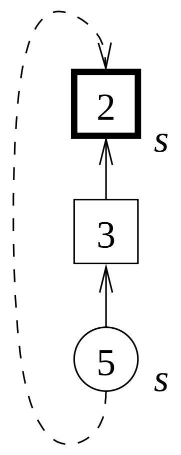


Figure 4.3: An initial trace recorded for the loop construct shown in Figure 4.1.

than possible successor nodes in the corresponding control flow graph is thus a potential side exit.

In case of the side exit in node 2 this is the edge to node 4, which is an alternative cycles through the loop. Since this path is also a cycle through the loop, it will be added to the trace tree if it is ever taken. The side exit edge originating at node 5, however, leaves the loop, and since it does not circle back to the loop edge, we are not interested in considering it as part of this trace tree and it will remain a side exit edge.

When a trace tree is compiled to native code, side exit edges generate code that restores the virtual machine state and then resumes interpretation of bytecode instructions by the virtual machine. The rationale of this approach is that we want to compile only frequently executed *loop code*, while infrequently executed non-loop code is interpreted by the virtual machine.

Once a trace tree has been constructed (initially containing only a single trace), it is compiled to native code (which we will discuss in more detail in Section 4.4). For subsequent executions starting from the anchor instruction a, the compiled native code is executed instead of interpreting a.

As long as the control flow follows the previous recorded trace $(2, 3, 5)$, the program will execute as native code without further interaction with the VM. In other words, as long as only explicit and implicit edges of the previous compiled trace tree are taken, the compiled code keeps executing.

4.3.2 Extending Trace Trees

If trace $(2, 3, 5)$ is the sole actual cycle through the loop (i.e. 2 always branches to 3 because its branch condition is invariant), the trace tree does not have to be extended any further to achieve good dynamic compilation results. If $2 \rightarrow 3$ is a true edge, however, we will see a side exit at 2 as soon as this edge is taken.

As discussed in Section 4.3.1, instructions that can cause a side exit are compiled in such a way that the virtual machine state is restored and interpretation resumed at the corresponding instruction if such a side exit occurs.

Since we detect anchor nodes by monitoring their execution frequency, we know that they are located in *hot* program code that is executed frequently. If we record a trace starting at such an anchor node a, it is likely that the entire trace consists of frequently executed program code. Thus, if a side exit occurs along such a cycle, such a side exit is likely going to occur frequently. Since side exits are expensive to perform it is desirable to extend the trace tree to include such alternative traces. The trace tree is then recompiled and subsequent encounters of the edge that triggered this side exit will no longer result in aborting native code execution.

Upon resuming from a trace tree side exit, the virtual machine immediately re-enters recording mode. The recorded trace is only added to the current trace tree if it cycles back to the anchor node of the trace tree without becoming overly long or executing any instructions and edges that are never included in a trace (such as exception edges).

Figure 4.4 shows the extended trace tree resulting from adding the trace $(2, 4, 5)$ to the initially recorded trace tree on the left. The new trace shares all instructions “upstream” (all instructions between the side exit node and the anchor node) of the (former) side exit node 2, which is in this particular example only the side exit node and anchor node 2

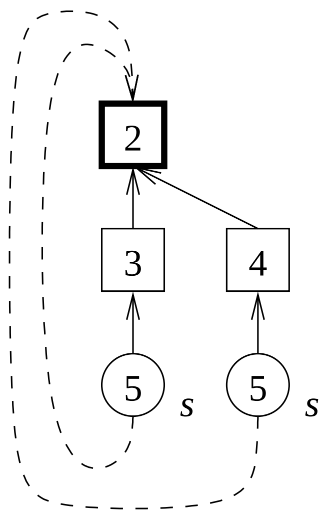


Figure 4.4: Lazy extension of the trace tree.

itself. Instructions are never shared “downstream”. Instead, duplicates of such instructions are added to the tree. Node 5 is an example for this. It appears in both traces, but since it is located after the side exit, it is not shared but duplicated and appears twice in the trace tree. This node duplication gives the trace tree its characteristic *tree*-like shape, and allows it to be transformed into SSA form and analyzed quickly.

The former side exit node 2 is no longer labeled as a side exit in Figure 4.4, because all possible successor edges are now part of the trace tree. Thus, once this trace tree is recompiled, subsequent executions of the corresponding native code will no longer cause the VM to be resumed, no matter whether $2 \rightarrow 3$ or $2 \rightarrow 4$ is executed at runtime.

4.3.3 Nested Loops

Trace graphs are not restricted to representing simple (non-nested) loops. Two interesting effects can be observed when recording a trace graph for a nested loop. First, the loop header of the inner loop tends to be selected as anchor node for the trace tree. This is intuitive, since the inner loop is executed more frequently than the outer portions of the loop, and thus the inner back-edge returning to the header instruction of the inner loop first crosses the anchor node threshold. And second, since trace trees have only one anchor node, only one loop header is treated as anchor node. The outer parts of the loop—including the outer loop header—are recorded until the control-flow arrives back at the inner loop header (which is the anchor node). Effectively, we have turned the loop inside out, treating the inner loop as the actual trace loop, and the outer loop as mere cycles originating from the anchor node and going back to it. This dramatically simplifies loop analysis, since our representation automatically simplifies all nested loop constructs to simple loops.

Figure 4.5 shows the control flow graph for a sample nested loop. Traces are shown in the order they would be recorded (left to right). For simplicity, implicit back-edges back to the anchor node $a = 2$ are drawn as simple edges to a copy of the anchor node drawn with a circle around it. This representation is equivalent to showing the implicit back-edge as dashed edge, and neither the copy of the anchor node nor the edge actually appear in the internal representation. The mere fact that the nodes from which the edge originates are labeled as leaf nodes is sufficient to indicate the existence of this implicit back-edge.

In the example shown in Figure 4.5, the left-hand side of the inner loop was recorded first as trace $(2, 3, 5)$. The next trace is $(2, 3, 5, 7, 1, 6, 7, 1)$. It consists of the continuation of the initial trace at the side exit at node 5, and then two iterations through the outer loop, because the outer loop header branched to 6 instead of 2 which would have terminated the trace earlier. The next trace is $(2, 3, 5, 7, 1)$. It shares all but the final edge to $1 \rightarrow 2$ with the previous trace. This trace must have been recorded after the previous one, because it splits off at node 1, which is part of the previous trace.

We always show the successor instructions of an instruction that was recorded first (i.e. is part of the *primary* trace) as a straight line. $(5, 7, 1, 6, 7, 1)$ for example is a secondary trace continuation that merges with the primary trace $(2, 3, 5)$ in 5. The naming of primary and secondary traces is recursive in the sense that the back-edge $2 \rightarrow 1$ is a secondary trace to $(5, 7, 1, 6, 7, 1)$, because it was recorded following a side exit from it. Such secondary traces that were recorded later following a side exit are shown coming in at an

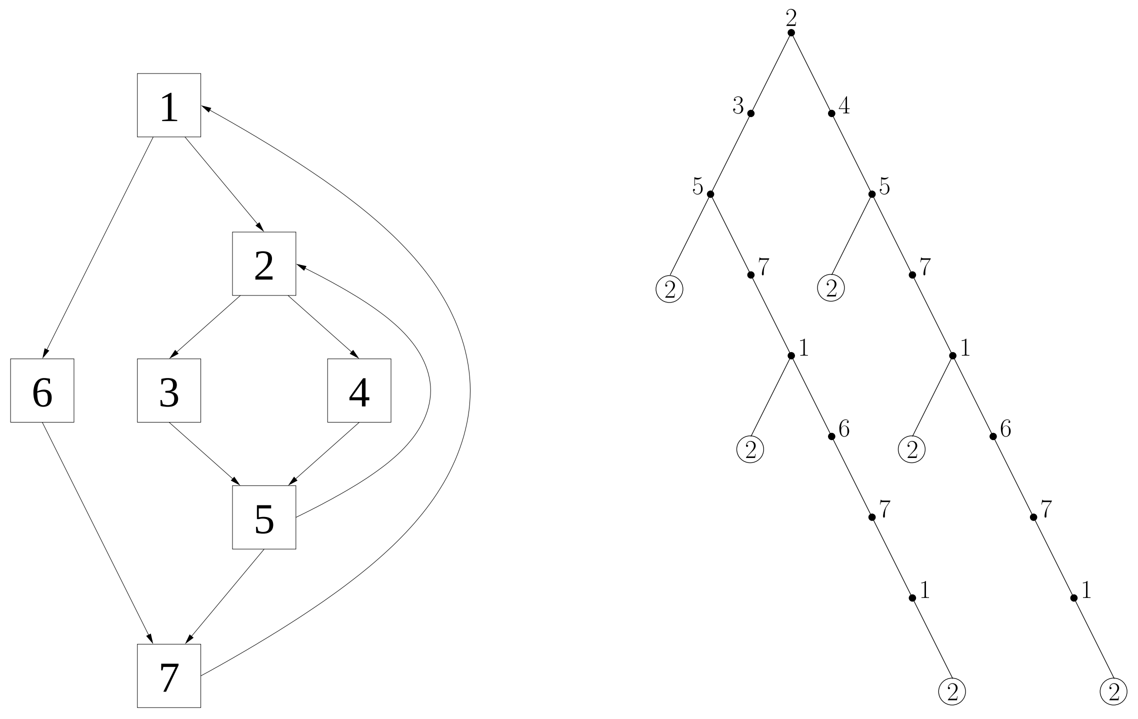


Figure 4.5: Example of a nested loop and a set of suitable traces through the nested loop.

angle.

The trace tree in Figure 4.5 is extended further with additional cycles through the right side of the inner loop, and a cycle through the outer loop branching of from this alternative path through the inner loop.

Since trace trees are assembled and extended dynamically, the exact order in which traces are added to the tree is not deterministic, but as we have discussed above, secondary traces are always recorded *after* their primary trace, forming a topological ordering of the traces.

4.3.4 Bounding Trace Trees

Trace trees can obviously grow indefinitely. Instead of branching back to the anchor $a = 2$, the outer loop in Figure 4.5 could, for example, enter a cycle $(6, 7, 1)$ and never return to 2, or return only after a large number of iterations. In essence we would repeatedly inline and unroll the outer loop, hoping that at some point the control flow returns to the anchor node $a = 2$.

To limit the growth of such malformed trace tree, in addition to a maximal trace length we also limit the number of allowed backward branches during trace recording. Since each such back-edge is most likely indicative of an iteration of some outer loop, by limiting backward branches we essentially limit the number of times we permit the outer loop to be unrolled before the control must return to the anchor node or we abort recording. In our current prototype implementation we permit 3 back-edges per trace, which is sufficient to generate trace-trees of triply-nested loops as long as the outer loops immediately return to the inner anchor. This is usually the case for most regular loops.

If a trace recording is aborted due to an excessive number of back-edges, we record this in the side-exit node and allow a certain number of additional attempts to record a meaningful trace. The rationale behind this is that potentially we only encountered an unusual irregular loop iteration, and future recordings might reveal a direct path back to the anchor node. Our prototype permits a second record attempt, but we have observed only one loop in our benchmarks that ever required such a second recording attempt.

4.3.5 Method Calls

Similar to outer loops, method calls are inlined into a trace tree instead of appearing as an actual method invocation instruction. Whereas a *static* method invocation has exactly one target method that will be inlined, *virtual* and *interface* method dispatches can invoke different methods at runtime, depending on the actual type of the receiver object.

During trace recording, we convert static and virtual method invocations into a conditional branch depending on the type of the receiving object. While in theory such a conditional branch instruction sometimes could have hundreds of potential successor nodes, most programs rarely invoke more than two or three specific implementations at single virtual method call site. Each of these target methods can potentially be inlined into the trace graph, as long as the abort conditions described above are not violated (i.e. not too many back-edges are encountered).

Experiments have shown that simple methods can easily be inlined into the trace tree of the invocation site since such methods often contain no or very few conditional branches. Method calls that invoke a method that itself contains a loop, however, are frequently rejected due to an excessive number of back-edges, in particular if the invoked method contains a nested loop. Especially for the latter, we believe that this is not really a problem. Instead of inlining the method into the surrounding scope, the method itself will at some point be recognized as a hot spot and a trace tree will be recorded for it. The slowdown resulting from interpreting the outer scope will not significantly impact overall performance, since the method does contain an expensive nested loop construct and thus optimizing the method by far outweighs the cost of interpreting the surrounding scope (if this was not the case, the method would have been inlined in the first place).

An additional restriction we apply to inlining method calls is that we only permit downcalls, i.e. the anchor node must always be located in the same or a surrounding scope as all leaves (which correspond to tails of traces). This means that we do not follow `return` statements and abort traces that encounter them in scope 0. This does not restrict the maximal trace tree coverage, because for every trace tree we disallow (i.e. a trace tree with the anchor node growing outwards) there is always another possible trace tree that does grow downward (with an anchor outside the method and only the traces reaching inside the method).

This restriction has the added benefit that it simplifies the handling of side exits inside inlined methods. Each side exit node is annotated with an ordered list of scopes that have to be rebuilt in case the traces abruptly ends at that point. By limiting the growth of trace trees in one direction, we always only have to add scopes to the top of the virtual machine stack when recovering from a side exit, and we never have to deal with removing method frames from the stack because a side exit happens in a scope further out than the anchor node (where the trace was entered).

The treatment of side exits in our system significantly differs from trace-based native-code to native-code optimization systems such as Dynamo [BDB00]. When inlining code from invoked methods we do not create a stack frame for the invoked method call at each iteration. Instead, the additional local variables are allocated into machine registers, just as local variables in the outermost scope of the trace tree. When a side exit occurs inside code inlined from a method, we materialize the additional method frames on the stack before writing back the stack and local variable state.

4.4 Compiling Trace Trees

To compile trace trees, we use a variant of Static Single Assignment (SSA) form [CFR+91]. In traditional SSA, multiple definitions of variables are renamed such that each new variable is written to exactly once. ϕ-instructions are inserted in control-flow merge points where multiple values of an original variable flow together. Figure 4.6 shows an example for transformation into SSA form in case of a loop containing an `if-then-else` statement. A ϕ-instruction has to be inserted in the basic block following the `if-then-else` statement to merge the values of x, which now has been split up into x_1 and x_2 for the left and right branch respectively. Another ϕ-instruction is inserted into the loop header (node 1) to merge the values for i, which can either be the initial value of i when the loop was entered (for the first iteration), or the value from the previous iteration for all subsequent cycles through the loop.

4.4.1 Trace Static Single Assignment Form

To transform traces into SSA, we perform stack deconstruction during trace recording. References to the Java stack are replaced with direct pointers to the instruction that pushed that value onto the stack. Since each trace corresponds to a cycle through a non-nested

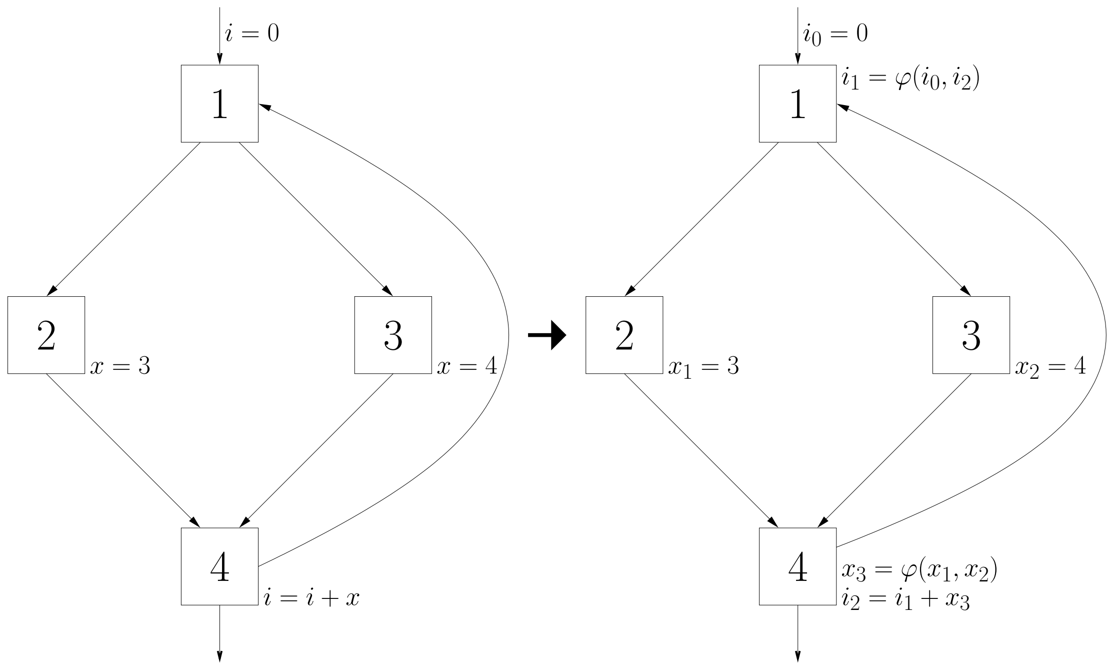


Figure 4.6: Traditional Static Single Assignment form.

loop[1], such pointers either point to an instruction upstream in the trace (closer to the anchor node), or they point to placeholder pseudo-instructions that represent values that were on the stack when the trace started at the anchor node, and local variable references. These pseudo-instructions are generated for all live stack locations and local variables, and are shared by all traces in a trace tree.

Since traces only follow exactly one control flow, they do not contain ϕ-instruction *except* for the anchor node which has two potential predecessor states: the initial state at trace entry, and the state from the previous trace iteration in case of returning to the anchor node from a completed iteration. To distinguish this special form of SSA from its common form, this form is also called *Trace Static Single Assignment* (TSSA) form [GPF06a].

Figure 4.7 shows the TSSA form for the example trace in Figure 4.3. If a variable x is assigned a different value in left side of the `if-then-else` statement (3) than the right side (4), we still would not have to insert a ϕ-statement, because this trace only consists of

[1]Even in case of nested loops a trace is always a single cycle through it, inlining the outer scopes into the non-nested innermost loop.

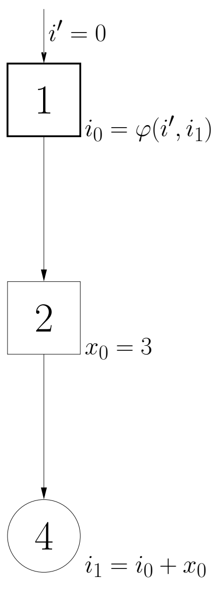


Figure 4.7: Trace Static Single Assignment (TSSA) form for the initial trace shown in Figure 4.3.

the instructions following the left side of the `if-then-else` statement. A ϕ-statement has to be inserted for the loop variable i, however, to update it after a successful iteration through the trace.

It is important to note that TSSA can be applied here only because we consider each trace separately, even thought traces often share instructions "upstream". Thus, we do not insert a ϕ-statement after branching nodes that have several potential successor instructions. Instead, each of those traces is considered a mere linear sequence, and is converted into TSSA separately. We will discuss in Section 4.4 the register coalescing algorithm that we use to ensure that register assigned to overlapping traces match up and values are always in the same location along all possible successor paths of a branching instruction.

4.4.2 TRACE TREE STATIC SINGLE ASSIGNMENT FORM

Treating each trace individually for SSA-form purposes of course creates a collision of their respective ϕ-instructions in the anchor node, since each trace would try to place a

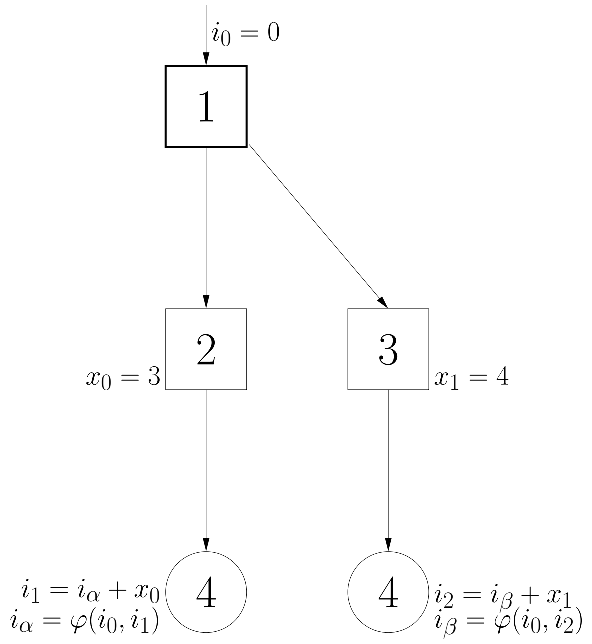


Figure 4.8: Trace Tree Static Single Assignment (TTSSA) form for the trace tree shown in Figure 4.4.

ϕ-instruction into the anchor node for each loop variable or value that it uses from the surrounding virtual machine content. To avoid this collision, when constructing SSA form for an entire trace tree—which we refer to as *Trace Tree Static Single Assignment Form* (TTSSA)—, ϕ-instructions are always attached to the leaf node of the trace. Since each trace has its unique leaf node, no collision can ever occur.[2]

Figure 4.8 shows the final TTSSA form for the trace tree we constructed in the previous section. References to the surrounding context (which are represented by placeholder pseudo-instruction) are always labeled with a zero index. Variable i_0 thus corresponds to the initial value of loop variable i when the trace tree was entered. Since all traces share the same pseudo-instructions when accessing the surrounding context, such *context* references are also shared by all traces. Variable assignments inside traces are denoted with a unique numeric index, whereas ϕ-instructions are denoted with a unique letter index (i.e. i_α and i_β).

[2]Traces can contain inlined copies of leaf nodes of other traces, but they still have their own leaf node further "downstream".

4.4.3 Code Generation and Register Allocation

Every time a trace tree is created or extended, we recompile the entire tree and re-generate the associated native code from scratch. While this seems to be wasteful and slow at first, it is important to realize that we only have to recompile a set of sequences of linear instructions, which can be done quickly and in linear time. Thus, a compiler using a trace tree representation is extremely fast.

When compiling a trace tree, the first step is to identify loop variables. A loop variable is any value in the surrounding context (stack or local variables) that is modified inside the loop. The presence of a ϕ-instruction updating a context variable indicates a loop-variable. If only one trace has a ϕ-instruction for a variable, its sufficient to make it a loop-variable, because all other traces have to respect the register assignment made for that variable, otherwise its value could be overwritten unwittingly.

The rationale for recompiling the entire tree every time a trace is added is that when a new trace is added, it might reveal additional loop variables, which for efficiency reasons we might want to allocate into a dedicated hardware register. For this effect, that hardware register must not be used along any possible trace path. In addition, such new traces can also reveal additional instructions that can be hoisted out of the loop, and thus again need a dedicated register, potentially invalidating previous register assignment assumption.

When compiling traces, we want to emit shared code for instructions that are shared "upstream" between traces. For this, we start compiling traces at their leaves, and ascend the trace tree until we hit an instruction for which we are *not* the primary successor, which means that the instruction we just came from was not the instruction that initially followed that instruction when it was added to the tree. This approach has a number of advantages for code generation and instruction pattern matching.

4.4.4 Compiling Traces

The code generator then starts to emit code, starting at the last instruction, and moving backwards. Correspondingly, the machine code is also generated backwards, which has the subtle advantage that the target address of conditional branches is always encountered before the actual branch instruction, eliminating the need to fix up target addresses in a second pass.

During code generation, we first try to perform constant folding for each operand reference of the instruction that is currently being compiled. Traditionally, constant folding is performed at compile time by executing the dynamic semantics of constant instructions. In a mixed-mode interpreter, this leads to significant code duplication between the compiler and the virtual machine, because both essentially perform the same task (executing bytecode instructions). In our system, instead of folding constants in the compiler, we rely on the bytecode interpreter to record the value calculated by each instruction in the trace. The compiler merely tracks a bit whether operands are constant or not, and if all operands of an instruction are constant, the result previously recorded by the virtual machine is assumed to be the correct result value of the instruction.

The only JVML instruction that warrants special treatment is the `IINC` instructions. In contrast to all other JVML instructions it does not store the value it generates on top of the stack and thus our stack recording mechanism cannot capture the value. Instead, it directly pushes the new value into a local variable. Our recording code is aware of this irregularity and records the value of the local variable instead of the top of the stack for `IINC` instructions.

Only if constant folding fails we assign a register to the instruction defining the operand. As code generation continues, we will eventually reach the instruction we just assigned a register to, and generate the necessary code to produce a value into that register. Code is only generated for instructions that were previously assigned a register. If we are able to satisfy all uses statically through constant folding, the instruction will have

```
IADD:
  if constant(son[0])
    swap(son[0], son[1])
  if constant(son[1]) && imm16(son[1])
    emit(addi, reg(son[0]), value(son[1]))
  else
    emit(add, reg(son[0]), reg(son[1]))

constant:
  IADD,ISUB,IMUL,etc:
    if constant(son[0]) && constant(son[1])
      value = OP(son[0],son[1])
    return TRUE
```

Figure 4.9: Bottom-up code generation with pattern matching.

no register allocation when we arrive at it, and it is considered dead.

Besides simplifying register allocation, bottom-up code generation also enables us to perform efficient pattern matching to emit specialized machine instruction forms. As shown in Figure 4.9, the backend iterates over the code bottom-up, which allows us to use simple pattern-matching for emitting specialized instruction forms. The generator function for the *IADD* bytecode instruction, for example, first checks whether its left operand is constant, and swaps its operands in that case. This is to ensure that if one operand is constant, this is always the right one, as expected by the *addi* machine instruction. Only operands that cannot be folded into a constant are assigned a register using the *reg* function. The corresponding code for the definition will be emitted later, once the code generator arrives at the defining instruction.

4.4.5 MERGING TRACES

If we start compiling at leaf 1 in Figure 4.10, for example, we would not stop emitting code until we hit the anchor node, because we only follow the primary trace (which is symbolized here through a straight line). When compiling leaf 2, however, we would stop the compilation run when hitting the primary trace (which started at 1), because we leave

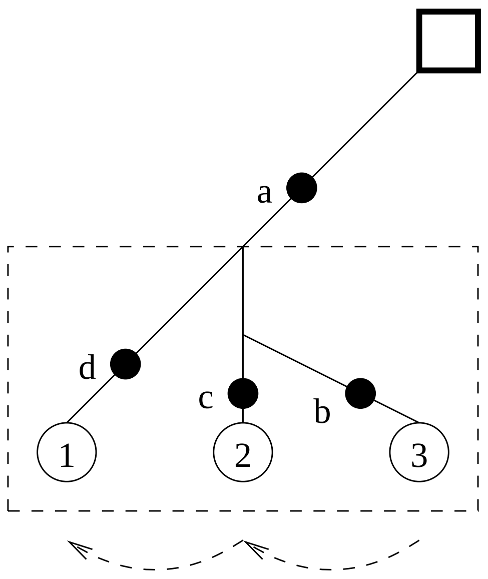


Figure 4.10: Bottom-up traverse of a trace tree during code generation in reverse recording order of the leaves of the tree.

the upstream code to be compiled when ascending from the leaf of the primary trace itself.

When ascending the trace tree along a trace path, we assign registers to all operands of every instruction we encounter. Due to the structure of our tree, all instructions that execute before the instruction being compiled are located "upstream" in the tree. In this sense the trace tree could amongst others also be considered a dependence graph. Thus, it is guaranteed that for a single trace, when compiling from bottom, we will always see all uses of an instruction before we see its actually definition.

Every time we approach an actual definition, we check whether a register was assigned to it previously. If this is the case, some "downstream" instruction uses the value generated by it, and we generate code for the definition. If no "downstream" instruction assigned a register to the definition (and it is side-effect free), we do not generate code for it, which in essence is equivalent to performing dead code elimination.

The order in which we compile traces is crucial, however. While it is guaranteed that in any particular *trace* we will see all uses of a definition before we encounter a definition, this guarantee does not hold for the entire trace tree. Consider the example in Figure 4.10. If we start compiling at leaf 1 and then compile leaves 2 and 3, we would encounter the definition site (a) right after passing over the downstream use d, but before we have visited uses b and c. This is the case because when starting at leaf 1 we do not stop ascending the tree at the merge point between a and d since the corresponding merge point is the primary continuation of the path coming from 1.

To guarantee that *all* uses (b, c, d) are seen before the definition (a), we have to find an ordering of the leaves such that all code downstream of a is generated before a itself is visited. In Figure 4.10, this is symbolized by a dashed box around the corresponding trace parts.

While not the only valid ordering, the reverse recording order of traces always fulfills this requirement, because traces can only be extended once they have been recorded. Thus, in the recording order of traces, a primary trace always occurs before any trace that depends on it. A secondary trace could never be inserted into the tree if its corresponding primary trace isn't present, because there is no place to attach it to.

Reversing this ordering guarantees that all dependent traces are visited before we consider the primary edge into the final merge point closest to the definition. In Figure 4.10, this means that compilation would start at leaf 3, stop when hitting its primary trace that originates at leaf 2, at which point code generation continues starting at leaf 2 until it is stops again at the merge point with the primary trace 1, which then finally is compiled in one sweep without interruption (since we are ascending the trace along its primary edge in the merge point).

When compiling the tree by ascending it in reverse recording order, registers are always blocked by the register allocator as soon as use is encountered, and the register is not freed again until the definition is reached (which means that all uses were visited). This implicitly also correctly blocks loop variables for the entire code, since as soon as the first use of a loop variable is encountered, that register will be blocked until the anchor node is compiled—which always happens last.

An important restriction for registers assigned to loop variables is that they cannot be used *anywhere* in the generated code *except* for a loop variable as long as the loop variable is not dead along a trace. This is the case because if we would use a register r_1 in a trace and then subsequently assign r_1 to a loop variable further up in the tree, the already generated code would unknowingly overwrite r_1 because when it was initially compiled, r_1 wasn't blocked yet.

To prevent this problem, every time a hardware register is used within a trace tree, we mark it as *dirty* and this flag is sticky even after the register is released again. While such dirty registers can be re-used for non-loop variables, loop-variables are only allocated from the pool of *virgin* registers that have never been used before. Our prototype implementation assigns registers in ascending order $(1, 2, 3, \ldots)$ to non-loop variables, and in descending order to loop variables $(31, 30, 29, \ldots)$. If there are not sufficient hardware registers (the compiler runs out of virgin registers), the compilation run is restarted and virtual registers which are mapped to memory through spilling are added to the pool. A more sophisticated approach would involve splitting live ranges and making loop registers available for the general pool once they become dead along a particular trace.

4.5 Trace Trees in Practice

We have implemented a prototype trace tree-based dynamic compiler for the JamVM [Lou06] virtual machine. JamVM is a small virtual machine suitable for embedded devices. The JIT compiler itself consists of roughly 2000 lines of C code, of which 800 lines are used by the front end that records traces and deconstructs the stack. TTSSA transformation, liveness analysis, common subexpression elimination and invariant code analysis are contained in another 200 lines of C code. This remarkable code density is possible, because linear sequences of instruction traces are much simpler to analyze than corresponding control-flow graphs. The remaining 800 lines of code consist of the PowerPC backend, which transforms our trace tree into native PowerPC machine code.

Compiled to native code, our JIT compiler has a footprint of approximately 150 kBytes (code plus static and dynamic data, including the code buffer), which is noteworthy for a compiler that implements a series of aggressive optimizations. To evaluate the performance of our prototype dynamic compiler, ideally we would like to compare it to existing commercial embedded Java virtual machines. Unfortunately, all commercial embedded JVMs (such as Sun's CLDC Hotspot VM system [Sun05]) are available to OEMs only, and we were unable to obtain a license to perform comparative benchmarks. A small number of free JVM implementations exist that target embedded systems, including the JamVM virtual machine that we used as basis platform. However, most of these do not have a dynamic compiler and thus are of limited use for a direct performance comparison.

An additional complication arises from the fact that embedded systems are not nearly as homogenous as the desktop computing market. A number of different processor architectures compete for the embedded systems space, including StrongARM/XScale, PowerPC, SPARC, SH and others. Not all embedded VMs are available for all platforms, further complicating comparative benchmarks. In addition to the interpreter-only JamVM, we use Kaffe [Wil06] and Cacao [KG97] as representatives for embedded virtual machines with just-in-time compilation. While both systems were not primarily designed for embedded use, both contain a baseline JIT-compiler that does not perform aggressive optimization, which is very similar to the state of the art in embedded dynamic compilation. Thus, even though a direct comparison with commercial embedded VMs would be preferable, the comparison with Kaffe and Cacao should provide an estimate on how our trace-based compiler performs in comparison to traditional space-saving and complexity-saving dynamic compilers.

Another small Java virtual machine suitable for embedded systems that we used in our benchmarks is SableVM [GH03], which uses a very simple template-based code generator. SableVM is small and space efficient, and faster than pure interpeters, but its performance still lags behind optimizing dynamic compilers.

In addition to embedded VMs (or to be more precise two VMs representative for this class of systems), we also compare our prototype to Sun's heavyweight JVM Hotspot systems. JVM Hotspot is clearly not designed for embedded use, and the code size of the JIT compiler alone is over 3.5 MBytes, which is more than 20 times the total size of our system. The goal of our system is to deliver performance close to JVM Hotspot, at a much lower cost.

We will first show the control-flow graph structure of a set of benchmark programs from the Java Grande [MCH99] benchmark suite, and discuss the traces our prototype compiler recorded for them. Then we will evaluate the impact of certain trace recording and trace tree extension parameters such as maximum number of traces in a tree on the overall performance of the generated code. As last evaluation step, and to demonstrate that the resulting trace trees can be compiled to efficient native code, we provide benchmark numbers comparing our prototype dynamic compiler with traditional embedded Java interpreters and a traditional industrial-strength desktop JIT compiler that uses control-flow graphs for program representation and optimization.

4.5.1 Example Traces

The control flow graph for the IDEA benchmark and the traces our trace compiler recorded for it are shown in Figure 4.11. The IDEA benchmark implements a block cypher with an inner loop processing individual blocks, and an outer loop passing the message pushing the entire message through the block cypher. Node 1 is the header of the outer loop, and node 3 the header of the inner (block cypher) loop. As discussed previously, the inner loop header 3 is detected first since it is "hotter" than the surrounding loop code. The trace $(3, 4)$ is recorded first (red trace, in case your printout is in color.) A second trace (red) is recorded starting at side exit node 4 once the inner loop is left, and cycles back to 3 eventually.

Figure 4.12 shows the control-flow graph and traces for the SOR benchmark, which consists of a triple-nested loop. Trace $(5, 6)$ is recorded first (red), followed by trace $(5, 8, 3, 4, 5)$ after a side exit in 5 (blue). Block 5 appears twice in this cycle due to the `do/while` structure of the loop, which caused node 6 to be recognized as anchor node and innermost loop header. The final trace (green) is recorded from side exit 3, and as in case of IDEA the trace tree achieves full coverage of the critical loop code.

Examples for an incomplete coverage are shown in Figure 4.13 and Figure 4.14. The graphs show the FFT benchmark and LU benchmark respectively. Both consists of a complex series of inner loops and branches, inside an overarching outer loop. Due to the back-edge abort condition our dynamic compiler is unable to extend a single trace to cover all possible paths, and instead several partial traces are collected and compiled, each accelerating the execution of a part of the outer loop. The outer loop itself continues to be interpreted.

4.5.2 BENCHMARKS

Table 4.1 and Figure 4.15 show the benchmark results for running the Java Grande [MCH99] benchmark suite on a 1.8 GHz PowerPC G5 with 1GB RAM using the virtual machines outlined above. All virtual machines are compared against the JVM Hotspot virtual machine running in interpretation-only mode (`-Xint`).

With a speedup of 7 and 10 respectively, our Trace Tree Compiler (TTC) and JVM Hotspot significantly outperform all other VMs in all benchmarks. Only Cacao is able to achieve comparable performance for two benchmarks (LUFact and SOR).

The result for the Series benchmark kernel significantly diverges from the results for the other benchmark programs. This is due to the fact that the Series benchmark spends most of its runtime in a tight loop invoking trigonometric functions, which for our platform (PowerPC) all VMs implement as native machine code library function. Thus,

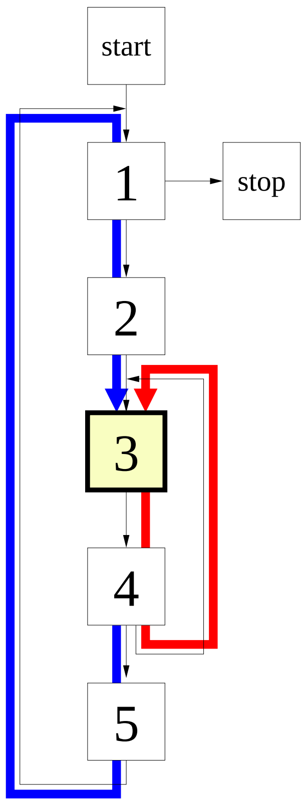


Figure 4.11: Control-flow graph and complete trace tree for the IDEA benchmark kernel.

the Series benchmark mostly measures the performance of the Java Native Interface (JNI) as well as the performance of the implementation of the trigonometric functions. None of the VMs is able to extract a significant performance gain through dynamic bytecode compilation. The Series benchmark is only meaningful for evaluating the capabilities of the JIT compiler on platforms where trigonometric functions are implemented in hardware (i.e. x86), permitting the JIT compiler compile the entire critical region to machine code without having to resort to JNI method invocations within the generated code. For the remaining graphs in this chapter we continue to include the Series benchmark program, but it is labeled with an asterisk (*) to indicate that the results for this benchmark have limited meaning for our platform (PowerPC).

While JVM Hotspot still out outperforms our system (speedup 10 vs speedup 7), this additional performance cost comes at a runtime price that is unaffordable for most embedded system. The runtime compilation costs of our system in comparison to the

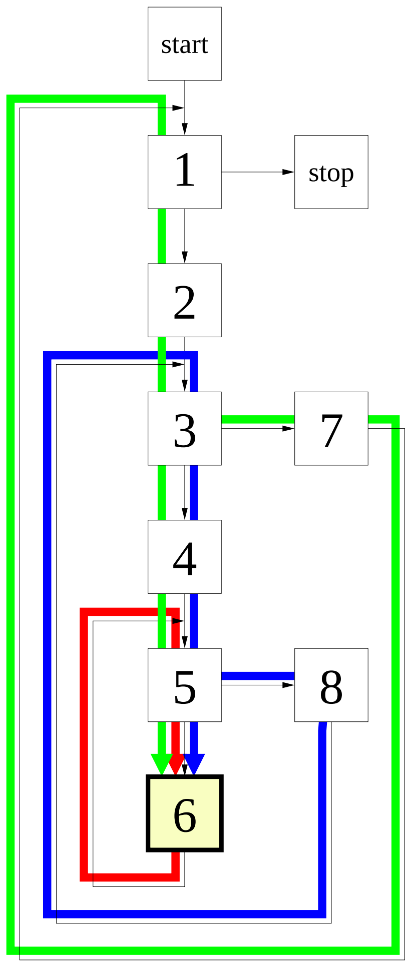


Figure 4.12: Control-flow graph and complete trace tree for the SOR benchmark kernel.

Hotspot compiler are shown in Figure 4.16 and Figure 4.17. On average, our JIT compiler is 350 times faster than the Hotspot compiler, and emits 30 times less native code.

In addition to the native code buffer our system also has to maintain the intermediate representation in memory across compiler runs since each additional trace has to be merged with the previously built trace graph. Figure 4.18 shows the aggregate memory consumption for the intermediate representation in comparison to Hotspot. Our system is 7 times more memory efficient than Hotspot.

It is important to note that the intermediate representation has to be held in memory only for a very short period of time. Once the first trace has been recorded for a certain "hot" code region, other relevant traces are added to the trace tree in rapid succession,

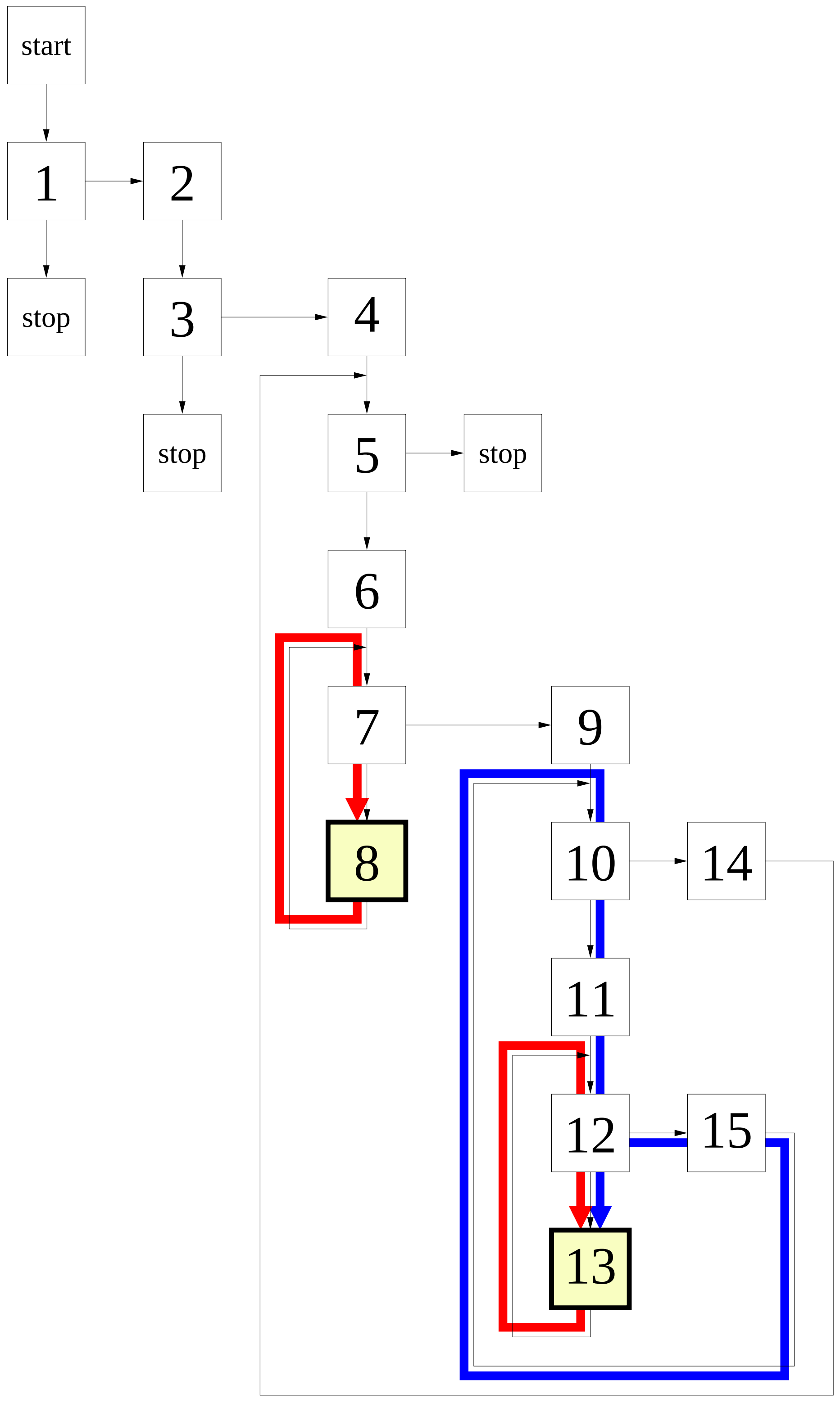


Figure 4.13: Control-flow graph and partial trace tree for the FFT benchmark kernel.

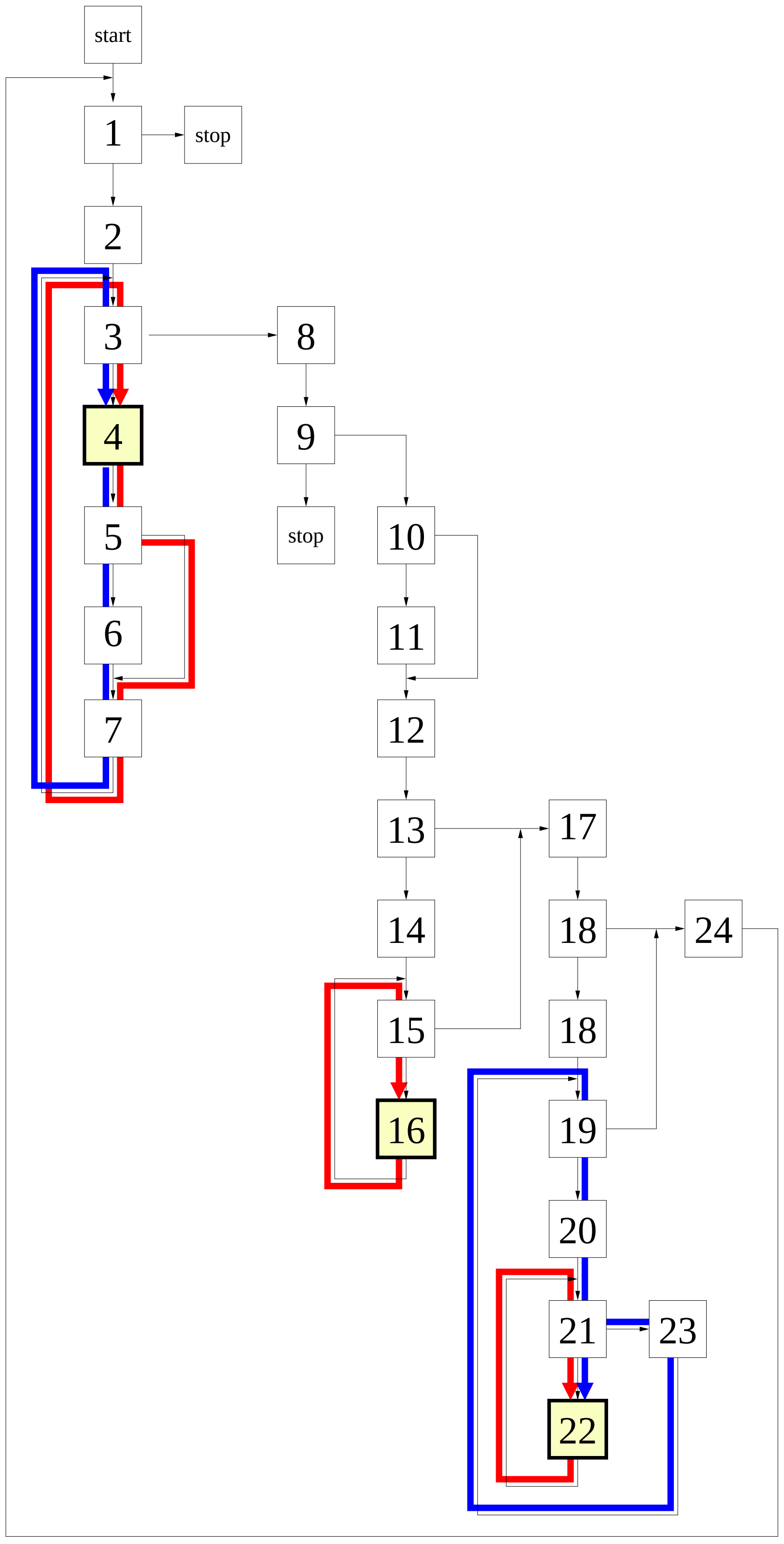


Figure 4.14: Control-flow graph and partial trace tree for the LU benchmark kernel.

	TTSSA	JVM Hotspot	JamVM 1.4.3	SableVM 1.13
Series	381.841	145.252	322.108	595.428
LUFact	6.667	3.359	70.604	39.244
HeapSort	16.328	11.207	58.521	34.592
Crypt	11.44	8.902	61.769	42.951
FFT	101.88	83.374	285.936	175.296
SOR	8.333	5.276	91.231	34.252
SparseMatMult	18.844	18.551	54.067	39.824

	Cacao 0.96	JVM Hotspot (interpreter)
Series	653.248	282.837
LUFact	8.518	83.526
HeapSort	29.62	86.772
Crypt	69.382	72.823
FFT	234.507	297.19
SOR	15.402	81.539
SparseMatMult	53.125	54.683

Table 4.1: Relative application performance of our system in comparison to existing JVMs.

quickly reaching a point where additional traces do not significantly improve performance any more.

Figure 4.19 shows the performance impact of bounding the number of traces that can be added to a trace tree. For this we modified the dynamic compiler to no longer extend trace trees when a side exit occurs once they contain a certain number of traces. The graph shows the speedup for each benchmark programs when compiled with trace trees consisting of a single trace only up to a maximum of 10 traces. For most benchmark programs no significant performance increase can be observed beyond the 4th trace in a tree.

Figure 4.19 thus confirms our observation that trace trees can be bounded to limit the compilation effort without incurring a significant performance penalty. This property of trace trees can be used to further reduce the resident memory footprint of our system by deallocating the trace tree data structure once a reasonable number of traces have been added to a tree since we no longer expect performance to increase significantly for future tree extensions.

Disposing of the IR, however, does not automatically preclude further optimization of

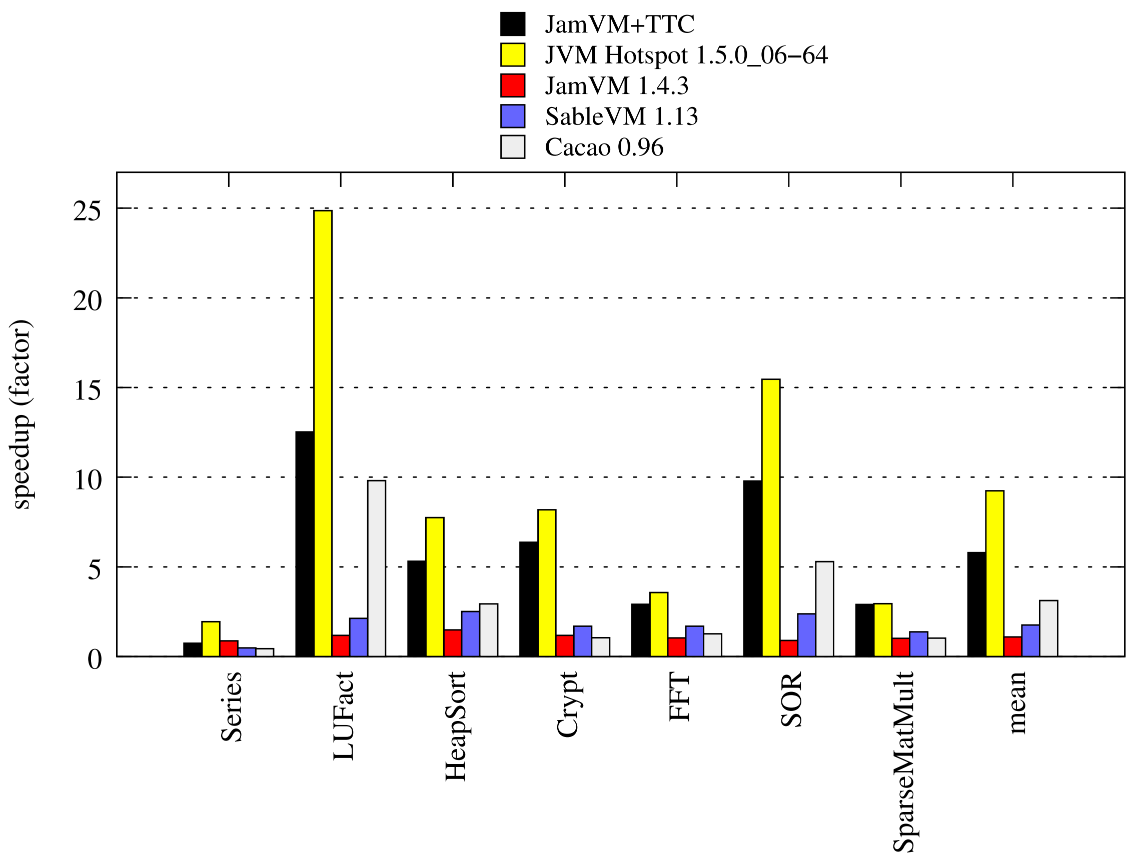


Figure 4.15: Relative application performance of our system in comparison to existing JVMs.

the code area. It only means that any additional optimization has to start “from scratch” by recording a fresh initial trace, followed by subsequent extensions of the trace tree. It is conceivable that program phase shifts cause excessive side exits after a tree was completed, and the prototype system could be extended to discard the optimized code and re-record the code area. Even repeated recompilations should not cause a significant performance problem considering the vast performance advantage of our system in comparison to existing heavy weight compilers.

Bounding the number of traces in a tree only limits how many *complete* traces are processed by the compiler. The trace recorder does not always succeed in recording a continuation trace after a side exit. Such *incomplete* traces occur when a loop region is left, or when a trace through a loop region is of excessive length. Since such traces do not

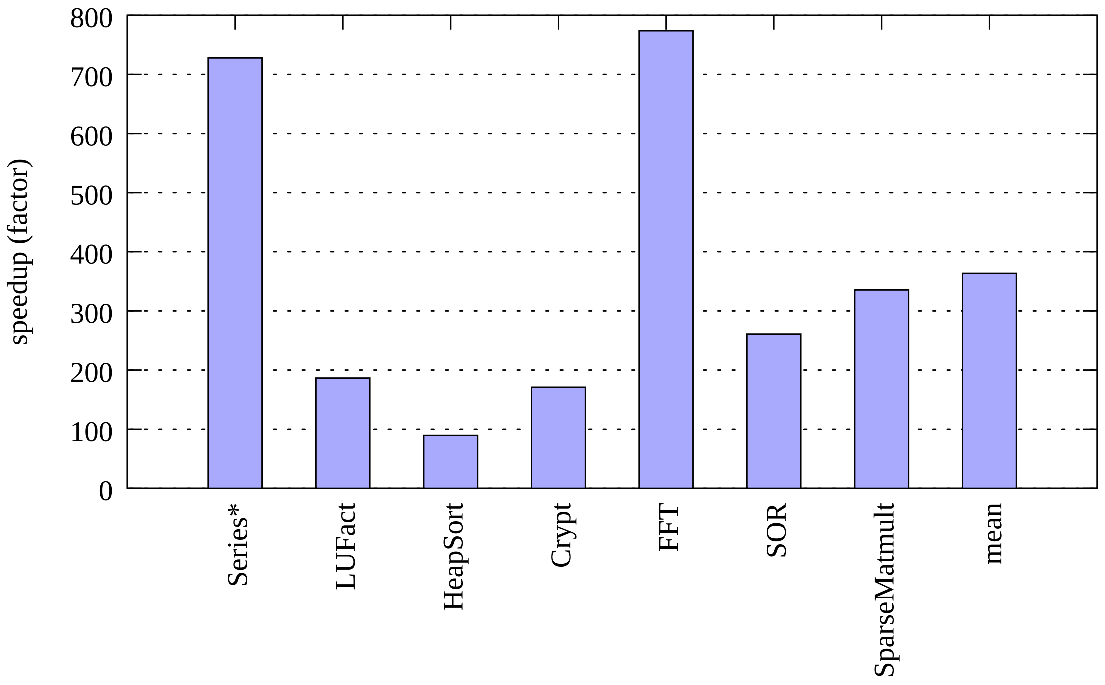


Figure 4.16: Just-in-time compiler performance (speedup) of our system relative to JVM Hotspot 1.5.0.

return to the loop header or only return to the loop header after executing too many instructions, they are aborted and do not count towards the maximum traces per tree limit.

Figure 4.20 shows that it is not sufficient to only extend a trace tree the first time a side exit is encountered along that path. The FFT benchmark, for example, contains side exits that require 3 attempts to record a complete trace before the trace tree could be successfully extended to cover the entire "hot" code region as evident from the additional speedup. The reason for this becomes more obvious when considering the trace tree graph of the FFT benchmark (Figure 4.13). Some of the loop edges of the FFT benchmark program are too long to be recorded and thus do not produce complete traces. When we encounter a side exit (i.e. node 21), the resulting tracing is only successful if the control flow stays "near" the code area the trace tree represents. Depending on the program state during the iteration, the control flow sometimes diverges (i.e. to node 24 after node 19 instead of following node 20 back to the loop header) and causes the trace to be aborted. Such a situation is not fatal. We simply attempt to extend the trace tree during the next

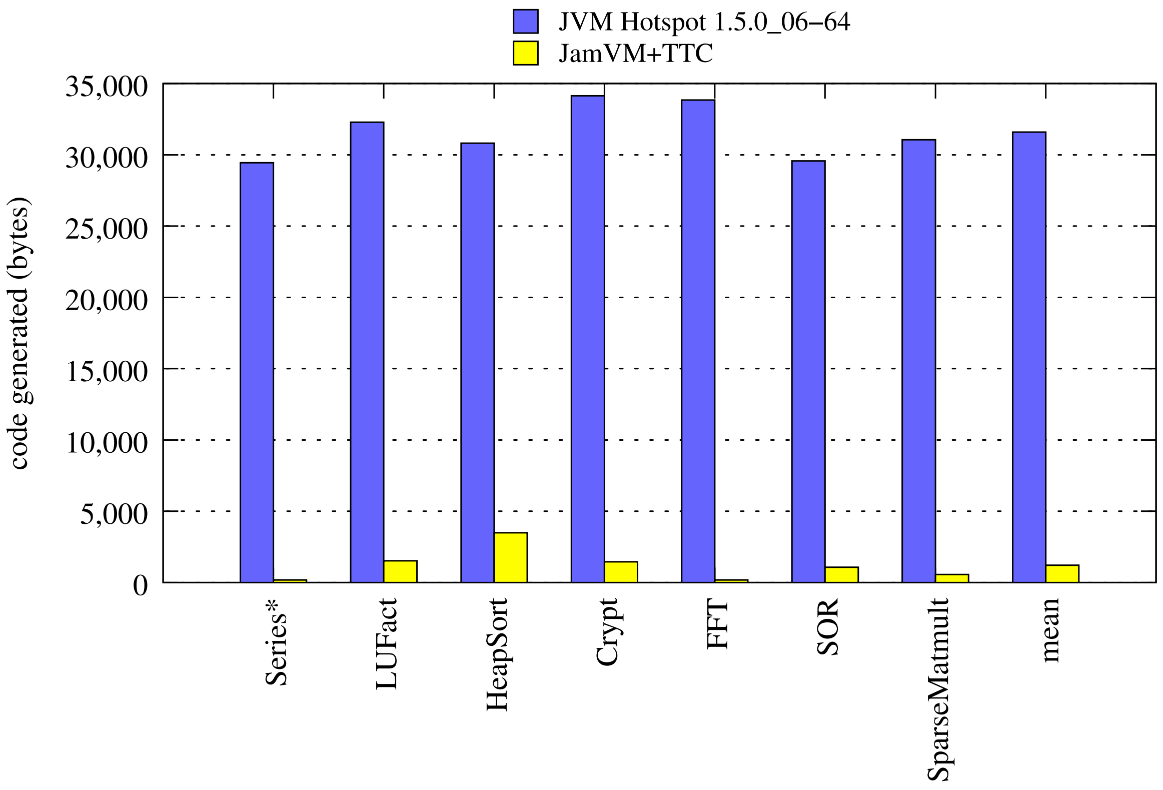


Figure 4.17: Native code emitted by our system and JVM Hotspot 1.5.0

occurrence of the side exit. If the path (and thus the side exit) is performance relevant, it will occur again and eventually complete. In our benchmarks none of the programs required more than 3 side exit tracing attempts to achieve the maximum speedup.

The overhead of recording traces that eventually will not complete can be reduced by limiting the number of permissible back-edges. Such back-edges are often loop edges returning to a loop header. Since we inline outer loops, we expect them to occur in side-exit traces to a certain extent, however, Figure 4.21 shows that the number of permissible back-edges in a trace can safely be limited to 2 without incurring a performance penalty. This allows for an early detection of traces that leave the area represented by the trace tree, and thus reduced the overall tracing overhead.

We have also analyzed the impact of the threshold value we use to trigger the recording of initial traces that form new trace trees. Just as side exits do not always produce the same secondary trace, one could suspect that the trace tree might be impacted by the specific

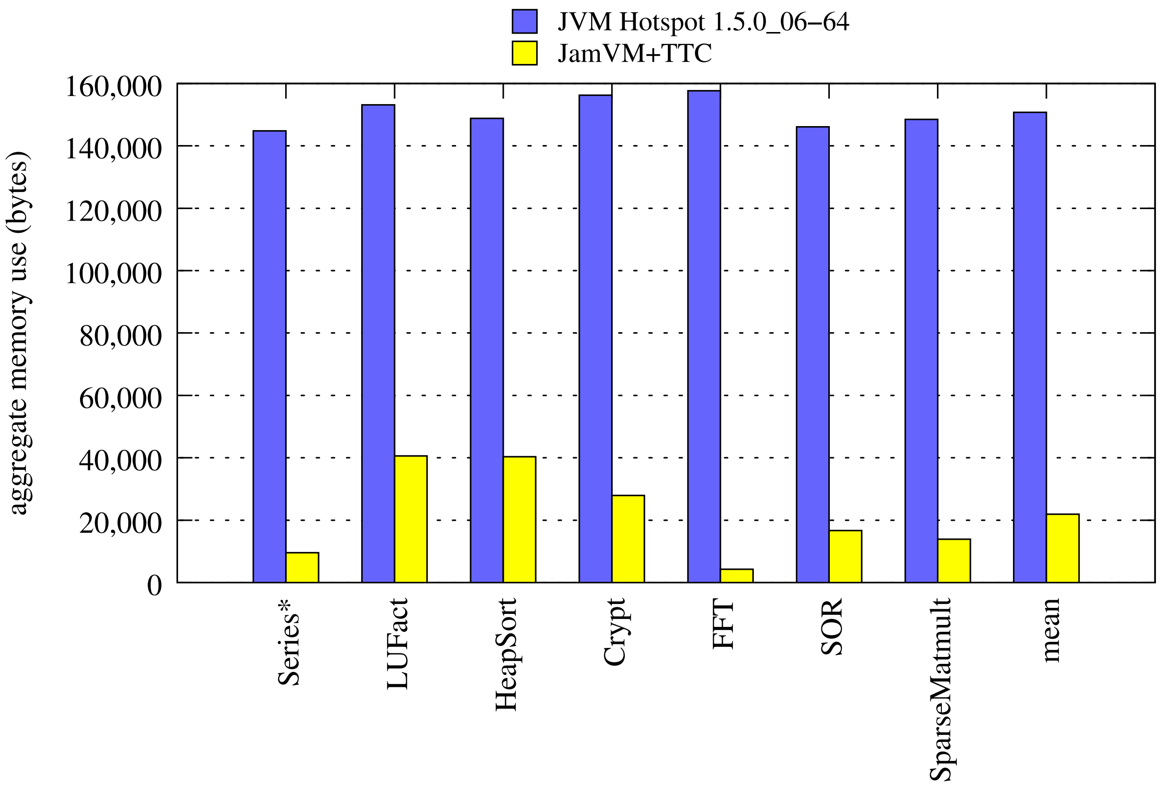


Figure 4.18: Comparison of the aggregate memory usage for the intermediate representation.

loop iteration it was created by. The SOR benchmark, for example, consists of two nested loops. The trace tree shown in Figure 4.12 was recorded for a loop iteration where the control-flow returned to the loop header (node 3) after passing node 4. However, when choosing a different threshold frequency, and thus recording the initial trace for a different loop iteration, the first trace could conceivably be $(3, 4, 5, 1, 2, 3)$ instead.

To evaluate the performance impact of starting trace trees with different initial traces we modified the benchmark programs to make sure that such a critical "fall over" point falls within the threshold value range of 1951 and 1960. Each of the benchmark programs produces at least two different trace trees for the 10 threshold values shown in Figure 4.22. Even though the underlying trace trees differ slightly (i.e. in the order traces were added and are being compiled), overall performance is not affected.

While this experiment does not guarantee total independence of speedup and picking

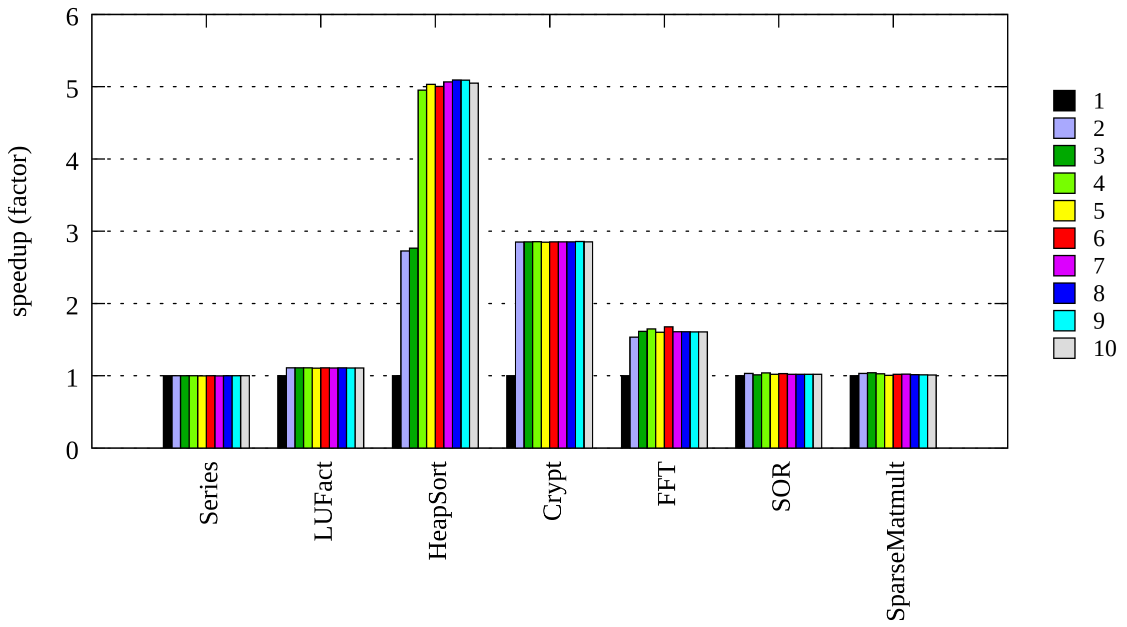


Figure 4.19: Impact of bounding the number traces that can be added to a trace tree.

the “right” threshold value, it does underline that trace trees are largely independent from the particular threshold value that was used to record them. Using different threshold values thus may produce different trace trees, but these trees seem to be equivalent as far as suitability for optimization and code generation is concerned.

4.6 Discussion

We have presented a novel intermediate representation for dynamic compilers based on a tree-shaped data-structure representing a group of closely related traces. We have discussed the mechanics of identifying a suitable anchor node that is shared by all traces in a trace tree, and we illustrated the dynamic extension of trace trees with additional traces everytime a suitable side-exit is encountered. To demonstrate the practical applicability of our approach, we discussed the prototype implementation of our trace-tree based dynamic compiler, we have shown the trace-tree representations it produces for a number of benchmark programs, and provided benchmark results comparing the our

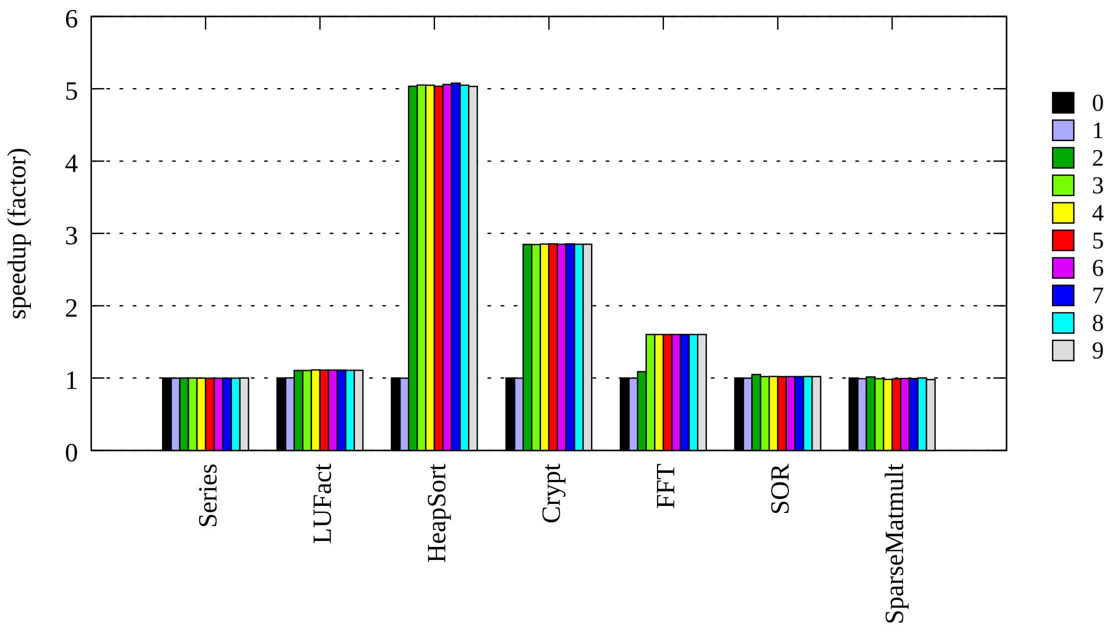


Figure 4.20: Impact of bounding the number attempts to extend a trace tree.

dynamic compiler to an existing dynamic compiler that uses a traditional control-flow graph based representation.

We believe that the main contribution of our work is finally advancing dynamic compilation beyond mere static compilation at runtime. When programs execute dynamically, the actual set of visited basic blocks and control flow edges often vastly differs from those seen in the static control flow graph. Our novel trace-tree representation captures this difference, and provides a representation that solely addresses "hot" code areas and "hot" edges between them. Any other basic blocks and instruction are not only not compiled, they are not even part of the intermediate representation and thus do not burden the analysis and compiler phases.

Our approach significantly diverges from traditional dynamic compilation systems which target entire methods and compile all code contained in them no matter how irrelevant a particular path in that code might be. The resulting increase in compilation time and the additional cost to generate efficient code might be acceptable in the desktop domain, but can be a significant burden in an embedded context. Our system only

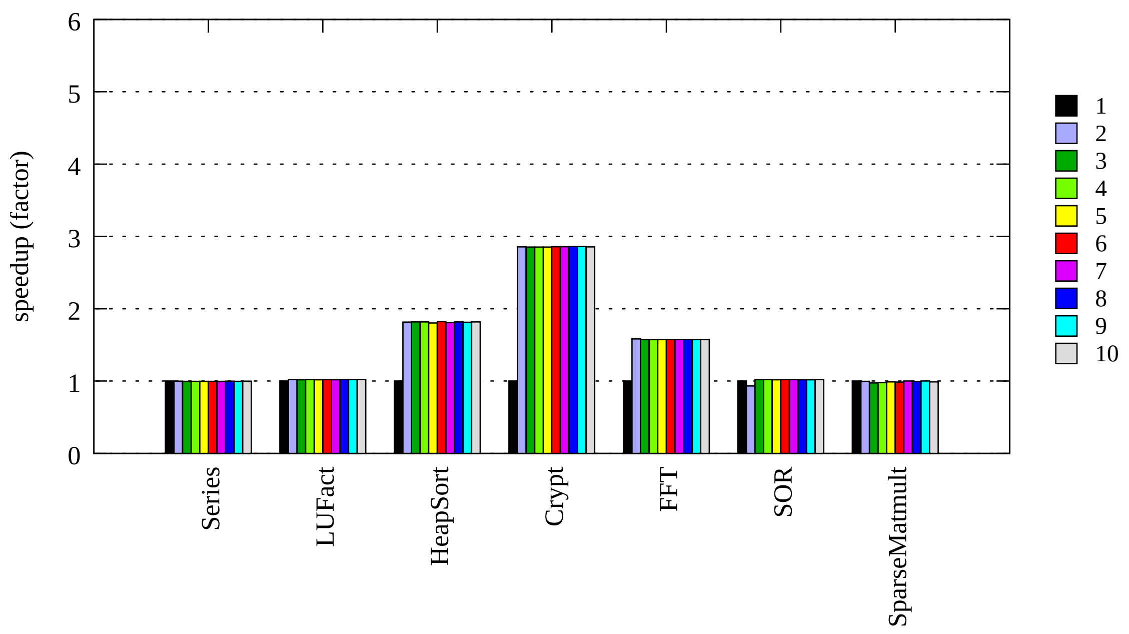


Figure 4.21: Limiting the number of permissible back-edges when extending a trace.

processes a small subset of the code, narrowly focusing on relevant “hot” code areas, which produces the significant compilation time and memory utilization improvements we have observed in the benchmark section.

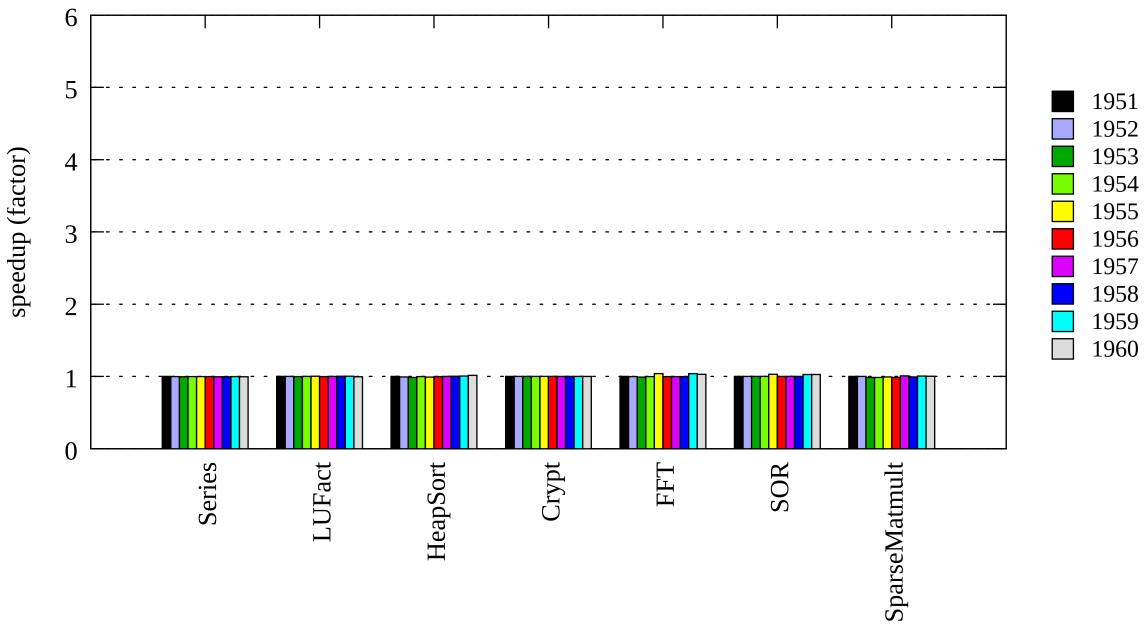


Figure 4.22: Impact of varying the trace recording threshold frequency.

CHAPTER 5

RELATED WORK

5.1 JAVA BYTECODE VERIFICATION

Java bytecode verification [Ler01a, Ler03] has been explored extensively. At the time of the introduction of Java Sun only provided an informal specification of the bytecode verification algorithm [Yel95, LY96]. Since then a number of formal specifications for the Java Virtual Machine Language (JVML) and its verification have been proposed [Gol98, FM99, Ler01a, FM03].

A number of vulnerabilities and type holes were discovered in Sun's implementation of the Java verifier [MF97]. Coglio et al. used formal analysis to show that not just the implementation but in fact also the informal specification was incomplete, and proposed a series of improvements to close type holes [Cog01].

Basin et al. first succeeded in proving soundness of bytecode verification using model checking [BFPV99]. Pusch et al. [Pus99] used automated reasoning using the popular Isabelle/HOL [NPW02, KN03] generic theorem proving environment.

Stark et al. showed that the Java bytecode verification is not complete [SS01, SS03], and provided examples of Java source code programs that cannot be compiled to verifiable Java bytecode. They proposed changes to the specification to remedy this problem.

Java bytecode subroutines are particularly difficult to specify formally and have been of particular interest to the research community. Stata et al. were the first to propose a type system for Java subroutines [SA98]. Other formal specifications of the Java subroutine construct include [FM99, O'C99, Qia99, KW03].

Ironically, Freund et al. found that subroutines in fact offer very little space saving, considering the complexity they add to the Java bytecode language [Fre98]. Coglio et al.

proposed to limit subroutine usage to a certain easily verifiable subset [Cog04], which is very similar to our treatment of subroutines in case of Static Single Assignment form-based bytecode verification [GPF05a, GPF05b, GPF06b]. While neither Coglio nor we can handle all possible uses of subroutines, both approaches capture all relevant and practically useful forms and thus do not limit the expressibility of the compiler.

Traditional bytecode verifiers use a form of iterative data-flow analysis [Qia00] to decide type-safety and thus have a quadratic worst-case runtime complexity, which we have shown to be exploitable by malicious programs [GPF03, PGF05].

Cohen proposed his Defensive Java Specification [Coh97] to entirely rely on runtime checks instead of static analysis and data-flow analysis to ahead-of-runtime verify the code. While feasible from a type-safety perspective, the resulting runtime overhead is prohibitive for most application.

Necula proposed Proof-Carrying Code (PCC) [NL96, NR01] to enable a code receiver to prove certain safety properties of mobile programs. Instead of proving the safety of the program at runtime, the code producer already annotates the mobile program with a safety proof. All the code receiver has to do is to check that the annotated proof actually holds. Necula's initial proof checker was fairly complex. Appel significantly reduced the the runtime system and in his Foundational Proof-Carrying Code work [App01], which is defined as proof-carrying code verification from the smallest possible set of axioms, using the simplest possible verifier and the smallest possible runtime system. Due to the small size of the axiom set, however, Appel's proofs are significantly more complex (in terms of size of the annotations) than those used by Necula.

Rose applied Necula's PCC to bytecode and proposed *lightweight bytecode verification* [Ros98, Ros03] as an alternative to performing an iterative data-flow analysis at runtime. Instead, a data-flow analysis is run at compile time and the Java bytecode is (sparsely) annotated with the fixed-point of that analysis. With the fixed-point already available, all the runtime verifier has to do to prove type safety is to confirm the fixed-point, which can be done in linear time. The *split verifier* used by Sun's Kilobyte Virtual Machine (KVM) [Sun99, Sun01] is based on an Rose's work.

Leroy implemented a verifier for Java Card systems [Ler01b, Ler02] that requires off-target bytecode analysis, but instead of using annotations transforms the bytecode in a way that allows linear time verification at runtime. The transformed bytecode has to conform to three specific requirements: a) the operand stack must be empty after every branch instruction, b) each JVM register must have exactly one type, and c) on method entry all non-parameter register must be initialized with `null`. The latter is done to ensure that any attempt to use an uninitialized register for array or object access will fail since this condition cannot be verified without data-flow analysis.

Leroy's approach is closely related to our work on structural encoding of static single assignment form for Java bytecode [GPF05b]. Similar to Leroy's verifier, we transform and rearrange the Java bytecode such that an aware verifier can infer SSA-format without further code analysis, while a traditional verifier can still verify the code using the standard verifier algorithm.

A bytecode verification approach that is very similar to our SSA-based verifier was proposed by League et al [LTS01]. λJVM, a functional representation of Java bytecode, makes data flow explicit, just like the verifier described in this thesis. Similarly to our work the authors split verification in two phases: a phase to construct the λJVM code, followed by a simple type checking phase. In contrast to our work, however, League et al. initially perform a regular data-flow analysis to infer types for the stack and local variables at each program point, whereas we split the verification in two phases to avoid the initial data-flow analysis all together.

5.2 Static Single Assignment Form

Building upon Shapiro et al.'s work on pseudo assignments [SS70], Cytron et al. [CFR+91] proposed Static Single Assignment (SSA) form as an intermediate program representation during optimization and code generation. By renaming all variables in the program such that each variable is written to exactly once, the representation implicitly contains definition/use information, which is a necessary prerequisite for many common code optimizations.

Transforming into SSA form requires the insertion of ϕ nodes at points where the control-flow merges to reconcile values along multiple incoming control-flow edges. Placing such ϕ nodes in *every* merge point and for every variable is trivial [AH00], but inefficient since it unnecessarily increases the size of the representation. Instead, most compilers use the minimal SSA form instead, which represents a program with the fewest number of ϕ nodes while maintaining its semantics. Minimal SSA form can be further enhanced by deleting ϕ nodes that merge the values of dead variables. This form is also refered to as *pruned SSA*.

To transform a program into pruned SSA, the compiler has to calculate the Iterated Dominance Frontier (IDF) set for every variable, and perform liveness analysis. Briggs et al. proposed semi-pruned SSA [BCHS98] as a less expensive alternative.

Bilardi et al. [BP99, BP03] and Sreedhar et al. [SGL94, SG95, SJGS99] independently proposed algorithms to calculate IDF sets in linear time. While Bilardi's algorithm is often faster in practice [BP03], Sreedhar's DJ-graph algorithm seems to be more popular and is used by several research virtual machines, including IBM's Jikes RVM [AAB+00] and Microsoft's Marmoth virtual machine and dynamic compiler [FKR+00]. Both algorithms are linear per variable only and have to be repeated for every variable in the program.

Amme et al. proposed to directly ship Java programs in a referentially safe SSA form-based format instead of transforming it into SSA-form at runtime and defined the SafeTSA [ADvRF01, vR05] format. SafeTSA eliminates the need for verification as mobile code is stored in a self-consistent format that cannot represent anything but well-formed and well-typed programs. Our approach and SafeTSA have in common that they both make the code available to the JIT in SSA-form, which can be used to speed up code generation.

Other approaches to make Java bytecode easier to optimize and compile by redefining the transport format include the Jimple bytecode [VRH98] used by the Soot framework [VRCG+99]. Jimple is not based on SSA form. Both SafeTSA and Jimple are incompatible with the Java bytecode format. Jimple is in fact an annotation that is embedded in standard Java class files and thus each class file contains two independent program representations that cannot be compared or verified against each other.

SafeTSA and Jimple are both related to our work on structural encoding of static single assignment form for Java bytecode [GPF05b]. Similar to SafeTSA we transport the code in SSA-form, but unlike SafeTSA (and Jimple) we do not require additional annotations or a new class file format because SSA-form is encoded structurally while maintaining full compatibility with the original bytecode format.

5.3 Trace Scheduling

Trace-based dynamic compilation is closely related to *trace scheduling*, which was proposed by Fisher [Fis81] to efficiently generate code for VLIW (very long instruction word) architectures. VLIW architectures encode a number of independent and parallel operations in a single instruction word. The TRACE VLIW architecture [CNO+88], for example, is capable of executing up to 28 instructions in parallel.

To take advantage of this extensive instruction-level parallelism the compiler has to properly arrange sequential program code into VLIW machine instructions. Since single basic blocks often do not contain sufficient parallelizable instructions, Fisher instead schedules instructions from entire traces through microprograms to fill the VLIW instruction slots. Moving and reordering instructions across basic block boundaries can impact other (off-trace) paths in the program, and often *compensation code* has to be inserted at each side-exit and the trace end point. [FGL94]

Fisher's work, and its subsequent improvements [SDJ84, Fis93] created the foundation for a number of trace scheduling systems, including the Bulldog compiler by Ellis et al. [Ell84, Ell86], the Horizon compiler by Howland et al. [HMS87, MDSW88], and the Multiflow system by Lowney et al. [LFK+93].

All these system follow Fisher's original trace scheduling approach when translating programs to microcode. The compiler first determines the most common paths in a program (i.e. using a heuristics, profile information, or programmer annotations), and then compiles those paths to machine code. The compiler then iterative identifies the next most likely paths, compiles them, and attaches them to the previous generated code, until the entire program has been translated.

The trace scheduling approach differs from our system in so far that a trace scheduling compiler always compiles all code in a program, whereas our dynamic compiler exclusively focuses on frequently executed ”hot” program regions, and leaves it to the virtual machine interpreter to execute compensation code and rarely executed code regions.

Compiling only partial "hot" program traces was first introduced in the Dynamo system by Bala et al. [BDB99, BDB00]. Dynamo is a transparent binary optimizer that records frequently executed traces and optimizes instructions in that trace. While the Dynamo work differs significantly in its target domain and implementation (Dynamo both records and emits machine code traces, while our system records bytecode traces and compiles them to machine code using SSA), its trace selection and recording mechanism is very closely related to our work.

Chen et al.'s Mojo [CLCG00] system is also a machine-code to machine-code binary optimizer. In contrast to Dynamo, it targets Intel x86 executables, while the original

Dynamo work was designed for the HP PA RISC CPU. Due to its similarity to Dynamo, Mojo is also closely related to our work.

Abandoning the traditional use of methods as compilation units for dynamic compilation in embedded systems was first proposed by Bruening et al. [BD00, BDA01]. The authors made the observation that compiling entire methods can be inefficient even when only focusing on ”hot” methods. Instead, they propose compiling based on hot traces and loops.

Whaley [Wha01] also found that excluding rarely executed code when compiling a program significantly increases compilation performance and code quality. The author discusses two examples of code optimizations that benefit from such rare path information: a partial dead code elimination path, and a rare-path-sensitive pointer and escape analysis. Compensation code is inserted to fall back to the interpreter when a rarely taken path (which has not been not compiled) is executed. A direct performance comparison of Whaley's work and our system is not possible, because Whaley didn't implement an actual dynamic compiler but instead simulated the results using profiling and off-line code transformation to benchmark the expected performance of the proposed system.

Berndl et al. [BH02, BH03] also investigated the use of traces in a Java Virtual Machine. However, while providing a mechanism for trace selection, the authors do not actually implement any code compilation or optimization techniques based on those traces.

Bradel et al. [Bra04, BA04, BA05] propose using traces for inlining methods calls in a Java Virtual Machine [Bra04, BA04, BA05]. Similar to our work they use trace recording to extract only those parts of a method that are relevant for the specific caller site.

Recently, the GNU C Compiler (GCC) framework introduced a tree-shaped SSA-based intermediate representation [Nov03, Nov04]. Similar to our work, the control flow is simplified to reduce optimization complexity. However, in contrast to our work, the author uses static analysis to generate this Tree SSA representation, whereas we dynamically record and extend trace trees. Novillo's Tree SSA approach also does not transform nested loops into simple loops, forcing the compiler to deal with the added complexity of SSA in the presence of nested loops and complex control-flows. This might be appropriate in a static compilation setting such as GCC, but would be too time consuming in the context of dynamic compilation.

5.4 Traces in Hardware

Traces are also used at the hardware level, i.e. to predict program branch behavior [JRS97] or improve fetch and issue policies [FPP97]. However, existing works in this area go far beyond mere hardware-based trace profiling, or caching of frequently executed instruction traces [RBS99, PFP99]. For example, Friendly et al. propose to use the fill unit of the processor to perform trace-based code optimization. [FPP98] The fill unit is particularly well suited for this purpose, because it is located outside the most critical paths of the processor since its processing speed is limited by the bandwidth of the memory hierarchy. Thus, the fill unit can dedicate significant time and effort to optimize instruction traces without creating any additional visible latency.

5.5 Superblocks

When a trace scheduling compiler selects a trace, it disregards all but the favored execution path through the basic blocks involved in that trace. While the favored path is the most frequently executed path (if the predictions are accurate), the control flow can still unexpectedly diverge from a trace (*side exit*), or unexpectedly branch into a trace (*side entry*). Side exits can be dealt with little overhead. If code was scheduled past a side exit, for example, the same code can be duplicated along the side exit path to ensure that it is always executed–even in case of a side exit. Side entry points on the other hand require a significant bookkeeping overhead to generate the required compensation code, since the compensation code can differ depending from where the side entry originates.

Based on this observation, Hwu et. al proposed superblocks as an alternative to trace scheduling. A superblock is a trace (or a set of traces) that has no side entries. A simple method for creating superblocks is to first collect regular traces, and then eliminate side entries along the trace path using tail duplication [CMH91]. Static program analysis has been proposed as well for superblock formation [HMB$^+$93].

Similar to superblocks, our trace trees have only a single entry point and no side entries. When extending trace trees, we follow side exits if they reconnect to the trace tree entry point, which could be considered an extension of Chang et al.'s tail duplication algorithm.

5.6 Region-based Compilation

Trace scheduling was the first technique to part from the scope of functions as compilation units, which has been (and often still is) the case for most traditional back-end frameworks. In subsequent work. Chang et al observed that not all parts of a program are equally profitable when comparing compilation and optimization effort (e.g. time and space), and the resulting speedup. Even the smallest machine code enhancement can produce a significant speedup within a frequently executed loop, while in rarely executed code (e.g. startup code) even dramatic enhancements would likely not produce a significant overall speedup. Thus, in their IMPACT system [CMC$^+$91] Chang et. al focus the compilation effort on certain profitable program regions, and do not heavily optimize code regions that are deemed unprofitable. Similar systems were proposed by Hank et al. [HHR95, Han96], and Liu et al. [LZQcJ03].

While most region-based compilation systems focus on static code generation (based on the argument that some aggressive optimization algorithms are so expensive that their cost is unacceptable for the whole program even in a static compilation setting), previous work also exists on region-based dynamic compilation. Suganuma et al. [SYN03, SYN06] for example proposed a region-based just-in-time compiler for Java. It uses runtime profiling to select code regions for the compilation and uses partial method inlining to inline profitable parts of method bodies only. The authors observed not only a reduction in compilation time, but also achieved better code quality due to rarely executed code being excluded from the analysis and optimization.

Our system is similar to region based compilation in that we focus on certain ”hot” (and thus frequently highly profitable) code regions of a program, excluding rarely executed code from compilation, analysis and optimization. In particular the partial method inlining approach proposed by Suganuma et al. is very similar in effect to our method of recording frequently executed traces across method calls. The most significant difference between trace-based compilation and region-based compilation is that region-based compilation reduces compilation time through excluding rarely used code, while trace-based compilation reduces compilation time further by significantly simplifying the complexity of compilation by working with linear sequences of instructions instead of control flow graphs.

5.7 Path Profiling

Unlike regions, traces do not only improve code quality by localize the optimization problem. Traces are also an ordered sequence that adds contextual information to the optimization equation. A basic blocks can be optimized differently depending on which basic blocks proceeded it, and since basic blocks can appear multiple times in a trace, the code can be actually specialized specifically for every individual occurrence. This approach of exploiting context information about basic block sequences has been studied in the context of *path profiling*.

Path profiling was first proposed by Ball et al. [BL96, BL00]. To collect profiling information about program paths, the author use a path encoding that produces unique index numbers for program paths. Profiling code is injected into the code such that a unique index number is generated when a certain sequence of basic blocks (path) is executed, and a profiling counter is associated with each such index number that counts the execution frequency.

Young [You98] proposed a path profile collection algorithm that is nearly as efficient as point profiling. For this, the authors limit the length of paths using a history cutoff length. With such a maximum path length limit each path is identically to the previous path, except for the latest basic block (that was just added to the list of basic blocks in the current path), and the earliest basic block (which was just dropped from the list). As a result, at every branch instruction only those path profiles have to be updated where the corresponding path ends at the current basic block. All other sub-path were already updated in previous iterations. In essence, profiling information is only tracked for paths of maximum lengths. All other paths are suffixes of such maximum length paths, and their profile information can be derived through post-processing. Using a pre-calculated lookup table that points to the maximum length successor path for the current path, the runtime overhead can be further reduced to a single table lookup and counter update per branch instruction.

Path profile information can be applied to enhance the result of a multitude of optimization algorithms, i.e. dead code elimination [GBF97], branch prediction [You98], global scheduling [YS98], and path-based partial redundancy elimination [GBF98].

Path profiling can also be used to perform data flow analyses more efficiently. Ammons et al. [AL98] proposed to construct a subset control flow graph that only contains hot paths, which they call the hot path graph (HPG). The analysis is then performed on this subset graph, and similarly to region-based optimization the analysis result is improved because less frequently executed paths are disregarded for the analysis.

To avoid having to insert compensation code before the analysis can be applied back to the control flow graph, Ammons et al. duplicate code covered by the HPG that is invoked from paths outside the HPG, which is very similar to tail duplication [CMH91].

Our trace-based compilation approach is similar to path profiling in that we perform path specific optimization along each trace. Also, the iterative extension of trace trees can be seen as a form of dynamic hot path graph construction. The most significant difference is that we do not incur the actual path profiling runtime overhead, because path specificity is an inherent property of our intermediate representation. Thus, we also do not have to deal with compensation code generation, because we already implicitly perform tail duplication as we record traces.

5.8 FEEDBACK DIRECTED OPTIMIZATION

Dynamic compilation with traces uses dynamic profile information to identify and record frequently executed code traces (paths). By dynamically adding them to a trace tree, and thus iteratively extending that trace tree as more traces are discovered, we perform a form of feedback directed optimization, which was first proposed by Hansen [Han74]. Feedback-directed optimization was used heavily in the SELF system [CUL91, Cha92] to produce efficient machine code from a prototype-object based language. Feedback directed optimization was subsequently also used in conjunction with other highly dynamic languages such as Scheme [Bur97, BDC98], the object-oriented parallel language Charm++[Kri96], Smalltalk [GR83], and Java.

Hölzle later extended the SELF system to not only use profile-guide compilation, but also adaptive code re-compilation [Höl94]. Kistler et al. [Kis99, KF01, KF03] further extended this idea and proposed a *continuous* program optimizer for the Oberon system that continuously re-schedules program instructions and dynamically rearranges the layout of objects during execution to maximize performance.

A problem frequently encountered by feedback-directed re-optimizers is how to substitute a newly compiled code region for its currently running counterpart. A long-running loop inside a method, for example, can only be replaced with an optimized version if the runtime system knows how to replace the currently running code including all the associated state with the newly compiled version. Such an exchange of code versions is often called *on-stack replacement* [HCU92].

Similar to traditional feedback-oriented and continuous compilers, our trace compiler compiles an initial version of a trace tree consisting of a single trace, and then iteratively extends the trace tree by recompiling the entire tree. A key advantage of our trace-based compiler over traditional frameworks is the simplicity of the on-stack replacement mechanism. Instead of having to handle machine-code to machine-code state transitions (which are highly complex), trace-trees always write back all state onto the Java stack when a side-exit is encountered. Recording restarts at such side exit points, and once the trace tree has been extended it is compiled and entered the next time execution arrives at the header node. In essence, in our system on-stack replacement comes for free because we never deal with methods in the first place. We can change the code layout at every side-exit from a loop.

5.9 JAVA JUST-IN-TIME COMPILATION

With the advent of Java [AG98, LY99], object-oriented languages have become ubiquitous, and today most workstation or server class runtime systems use some form of feedback-directed optimization to execute Java programs. [Ayc03] Besides countering the complexity increase on the software side, feedback-directed optimization also simplifies dealing with the increased complexity of modern processor architectures. Instead of trying to predict and model the behavior of a processor, dynamic profiling is used to measure the behavior at runtime, and adjust the program code accordingly. [Smi00]

A popular research compiler for Java is IBM's Jikes (formerly Jalapeño) Virtual Machine. [BWC+99, AFSS00, AFG+00, AAB+00, AHR02]. It uses dynamic profiling to optimize code at runtime. Since the compiler uses methods as compilation units, on-stack replacement is performed to substitute new code versions for the currently running copy.

Other optimizing research Java just-in-time compilers include the Intel's Open Runtime Platform [CLS02, CEG+05] and the StarJIT compiler [ATBC+03], and Microsoft's Marmot system [FKR+00].

Since dynamic optimization is so crucial for the efficient execution of Java code, commercial just-in-time compilers such as Sun's HotSpot VM [Sun02] and IBM's production VM [SOT+00, SOK+04] often feature the same aggressive dynamic optimization and on-stack replacement techniques as their research compiler counterparts.

A popular JVM for academic research is SableVM [GH03]. It uses a very simple template-based code generator. SableVM is small and space efficient, and faster than pure interpeters, but its performance still lags behind optimizing dynamic compilers. The CACAO virtual machine implements a just in time for the Alpha processor [KG97]. The Kaffe virtual machine [Wil06] is an ongoing project to provide a free and open source virtual machine for Java. It contains an outdated non-optimizing compiler, as well as a more recent (but less table) optimizing compiler that performs some basic local optimizations. As evident from CACAO and Kaffe, building a dynamic compiler is a Herculean task, and many academic frameworks are incomplete or cannot compete with commercial compiler systems. As a result many researchers use and extend existing open virtual machines such as Jikes to implement new dynamic compilation techniques.

Our system significantly differs from existing just-in-time compilers in that it achieves very competitive performance with much less effort. Our entire dynamic compilation framework consists of less than 2000 lines of code, and it was implemented by a single graduate student in less than a year. Yet, it achieves significantly better application performance than existing lightweight virtual machines such as SableVM or Cacao. We attribute this to our novel use of traces, which can be compiled and optimized at much less cost, and with much reduced complexity in comparison to traditional control-flow graph based compilation.

5.10 Embedded Java Virtual Machines

A number of existing virtual machines target the embedded and mobile systems domain. The Mate system by Levis et al. [LC02] provides a Java VM for very small sensor nodes. Sun's Kilobyte Virtual Machine (KVM) [Sun99, Sun01] targets 32-bit embedded systems with more than 256kB RAM. KVM is not compatible with the full Java standard and only supports a subset of the Java language and runtime libraries. JamVM [Lou06] is not strictly an embedded VM, but it is small enough to be used on small embedded devices. In contrast to KVM, it does support the full Java standard.

Sun has produced a research JIT compiler for the Kilobyte Virtual Machine, called KJIT [Sha02]. KJIT is a lightweight dynamic compiler. In contrast to our trace tree-based JIT compiler, KJIT does not perform any significant code optimization but merely maps bytecode instructions to machine code sequences. Also, KJIT seems to be an internal research project only. We have not been able to obtain it for comparative benchmarks.

Sun's current implementation of an embedded JIT compiler is called CLDC Hotspot VM [Sun05]. Unfortunately, very little is known about the internal details of this compiler. According to Sun's white papers, CLDC Hotspot performs some basic optimizations including constant folding, constant propagation, and loop peeling, while our compiler also applies common subexpression elimination and invariant code motion.

Other VM's for the embedded domain include
E-Bunny [DGK+04, DMTY05, DMT05] and Jeode EVM [Ins02]. E-bunny is a simple, non-optimizing compiler for x86 that uses stack-based code generation. It is very fast as far as compile time is concerned, but yields poor code quality in comparison to optimizing

compilers. Jeode EVM is an optimizing compiler that uses a simplified form of dynamic compilation. Unfortunately, just as with CLDC Hotspot, little is known about its internals.

Kaffe [Wil06] and Cacao [KG97] are free virtual machines for Java that are small enough to run on mobile devices. Both contain JIT compilers, but neither performs aggressive code optimization. We use Kaffe and Cacao in our comparative benchmarks because they are suitable for embedded use, and freely available.

IBM's J9 Virtual Machine [IBM06] is a modular VM that can be configured to run on small embedded devices, workstation computers, and servers. While in its smallest configuration (which is used for cell phones and PDAs) it can run on embedded platforms, that configuration only supports a very limited set of code optimizations.

CHAPTER 6

CONCLUSIONS

6.1 SUMMARY OF RESEARCH AND CONCLUSIONS

In Chapter 2 of this thesis, we showed that the worst-case verification effort of the Java bytecode verifier can be problematic. By carefully analyzing the data-flow algorithm underlying the Java verification approach, we were able to construct Java byte codes that, while correct, consume inordinate CPU resources during their verification. From a more general perspective, the vulnerability described in this chapter demonstrates the need, when dealing with mobile code, for algorithms that are not only correct, but also *efficient*.

In Chapter 3 we presented an alternative verifier that not only is faster than the standard Java verifier, but that also computes the dominator tree and brings the program into Static Single Assignment form "for free". As a result, the respective computations need not be repeated in subsequent stages of the dynamic compilation pipeline. When considering the overall complexity of the proposed verifier, it became apparent that verification complexity (and implicitly cost) is distributed very unevenly. Straight-line code is trivial to verify since each instruction has exactly one predecessor and thus the verifier state can be easily updated in a single step. Dealing with control-flow merges where instructions have several possible predecessors is much more complex, and it necessitates an iterative data-flow analysis in the case of the standard SUN verifier, or the use of Static Single Assignment form in the case of our verifier.

Based on this observation, in Chapter 4 of this thesis we proposed a dynamic compiler that eliminates control-flow merges from the program through a novel use of trace scheduling, which we call trace-based compilation. By doing so we can exploit the advantage of Static Single Assignment form for program optimization and code generation where transforming into SSA is cheap (sequential code), without having to deal with the complexity of control flow merges (where constructing SSA is difficult).

We believe that the main contribution of the work described in this chapter is finally advancing dynamic compilation beyond mere static compilation at runtime. When programs execute dynamically, the actual set of visited basic blocks and control flow edges often vastly differs from those seen in the static control flow graph. Our novel trace-tree representation captures this difference, and provides a representation that solely addresses “hot” code areas and “hot” edges between them. Code is generated for exactly those code paths that are relevant. Any other basic blocks and instruction are not only not compiled, they are not even part of the intermediate representation and thus do not burden the analysis and compiler phases.

This does not only improve upon traditional method-based bytecode compilation approaches, it offers an even better starting point for code optimization and code generation than source code since our representation reduces the program to a subset of actually relevant and frequently executed code traces which are indistinguishable in source code form due to the lack of runtime profiling information.

6.2 Future Work

While conducting the research summarized in this dissertation we have identified a number of unanswered research questions that fall outside the scope of this thesis. In this section we will discuss these unanswered research questions and outline possible future work to address them.

6.2.1 Integrated Trace-based Bytecode Verification and Compilation

In Chapter 3 we have introduced a novel efficient bytecode verification algorithm that produces a SSA-based high-level code representation, which can be reused “for free” for code optimization and code generation. We have currently not integrated this verification algorithm with our trace-based compilation framework since our trace-based compiler is only interested in a control-flow merge free “hot” code subset. However, we believe that such an integration is possible with little effort.

Possible future work in this direction would be to limit verification to such “hot” code traces. Instead of verifying the entire program prior to execution, traces would be verified only after they have been recorded. The remaining code is verified “on-the-fly” through increased runtime checking during interpretation, for example by maintaining a separate stack tracking the type of objects on the value stack.

It is important to note that it is not sufficient to perform runtime checking and then rely on the fact that every trace in the tree was already subject to runtime checking (otherwise it would not have been recorded). Traces still have to undergo static verification after they have been recorded, because runtime checking only catches actual typing violations, whereas traces have to be safe with respect to every possible type combination since compiled trace code does not contain any runtime checks.

This “mixed-verification” approach would allow to benefit verification from the reduced complexity of the trace-based approach. Similarly to code optimization, verification is much easier (and cheaper) for code that does not contain control-flow merges, or as in case of our trace tree representation only exactly one merge point (in the loop header). This does not only improve the worst-case performance characteristic of the verifier (verification would always complete in at most 2 iterations), but can also improve the fidelity of the verification results due to the reduced complexity of the verifier implementation.

However, such a “mixed-verification” environment, while safe, is not exactly equivalent to the semantics of the Java bytecode verification algorithm. For example, partially correct methods are allowed to execute as long as no actual incorrect typing is attempted or the possibility thereof is discovered through static analysis of a recorded trace.

For “cold” code this approach would certainly incur a certain performance loss, but since such “cold” code is executed infrequently by its very definition the overall impact on runtime performance should be manageable.

6.2.2 Optimizing for other Dimensions than Resource Usage

For the purpose of this thesis we mainly focused on efficient runtime resource utilization as our goal function when we re-design and evaluating the intermediate representation and the dynamic compiler.

However, we believe that our trace-based compilation approach could also be used successfully in systems where other properties than resource utilization are the key objective. In a mobile environment energy consumption is often an important priority, for example. A possible avenue for future work would be to evaluate how trace compilation would look like when the goal is to minimize energy utilization. In such an environment it is likely that trace trees should be constructed much earlier using lower threshold values to trigger the recording of initial traces. Due to the high (energy) overhead of interpretation and the low cost of compilation in our system, it is conceivable that always compiling all loops immediately after the first few iterations might produce the best overall energy balance.

Another property that is often key in embedded systems is robustness. While desktop users are accustomed to software failures and reboots, such lack of software robustness can have catastrophic consequences in mission-critical embedded systems. In the past this has mostly precluded the use of dynamic compilation in such systems since dynamic compilers introduce another layer of dynamic behavior that is hard to predict statically. Trace-based compilation might offer an acceptable “middle-ground” due to its much reduced complexity resulting from the elimination of control-flow merges. Our current dynamic optimizer prototype consists of less than 200 lines of C code. This small amount of code is much easier to verify and prove correct than the massive verifier frameworks used in traditional control-flow graph based dynamic compilers.

Merging our trace-based compiler approach with the idea of certifying compilation [SS03] could prove a particular interesting avenue for future work. Certifying compilers generate proof code on the fly that shows that the code optimization and code generation process did introduce any type errors. These properties can then be checked on the generated machine code prior to accepting it for execution. Certifying compilers have not been widely adopted in part because certification and proof checking introduce an additional runtime burden. We have shown in this thesis that trace-based compilation is significantly faster than control-flow graph based compilation. Certifying compilation is likely to experience a similar speedup which might be sufficient to make its additional runtime cost more widely acceptable.

6.2.3 Trace-specific Optimization

When compiling trace trees, we use a trace merging algorithm that permits merging individually compiled traces to globally optimized machine code. Currently we use the same compiler settings and the same compiler for all traces in a trace tree. This approach might not be ideal for all applications. Future work should explore whether varying compiler settings or even using entirely different compilers for individual traces when compiling a trace tree can yield a performance benefit, or improve the compilation result with respect to other compilation goals such as energy efficiency or robustness. With our trace merging method such individual compilation results can still be merged to a globally valid code output for the entire tree, even if separate compilers were used for each trace.

BIBLIOGRAPHY

[AAB$^+$00] B. Alpern, C.R. Attanasio, J.J. Barton, M.G. Burke, P. Cheng, J.D. Choi, A. Cocchi, S.J. Fink, D. Grove, M. Hind, et al. The jalapeño virtual machine. *IBM Systems Journal*, 39(1):211, 2000.

[ADvRF01] Wolfram Amme, Niall Dalton, Jeffery von Ronne, and Michael Franz. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 137–147, June 20–22, 2001. *SIGPLAN Notices,* 36(5), May 2001.

[AFG$^+$00] M. Arnold, S. Fink, D. Grove, M. Hind, and P.F. Sweeney. Adaptive optimization in the jalapeño jvm. *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, 2000.

[AFSS00] M. Arnold, S. Fink, V. Sarkar, and P.F. Sweeney. A comparative study of static and profile-based heuristics for inlining. *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00)*, pages 52–64, 2000.

[AG98] Ken Arnold and James Gosling. *The Java programming language (2nd ed.).* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[AH00] John Aycock and Nigel Horspool. Simple generation of static single assignment form. In *Proceedings of the 9th International Conference in Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 110–125. Springer, 2000.

[AHR02] M. Arnold, M. Hind, and B.G. Ryder. Online feedback-directed optimization of java. *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 111–129, 2002.

[AL98] G. Ammons and J.R. Larus. Improving data-flow analysis with path profiles. *ACM SIGPLAN Notices*, 33(5):72–84, 1998.

[App01] A. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 247–256. IEEE Computer Society Press, 2001.

[ATBC$^+$03] A.R. Adl-Tabatabai, J. Bharadwaj, DY Chen, A. Ghuloum, V. Menon, B. Murphy, M. Serrano, and T. Shpeisman. The starjit compiler: A dynamic compiler for managed runtime environments. *Intel Technology Journal*, 7(1):19–31, 2003.

[Ayc03] J. Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.

[BA04] Borys J. Bradel and Tarek S. Abdelrahman. The use of traces for inlining in Java programs. In *Proceedings of the 2004 International Workshop on Languages and Compilers for Parallel Computing*, pages 179–193, September 2004.

[BA05] Borys J. Bradel and Tarek S. Abdelrahman. A characterization of traces in java programs. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers*, pages 87–93, June 2005.

[BCHS98] P. Briggs, K.D. Cooper, T.J. Harvey, and L.T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software Practice and Experience*, 28(8):859–881, 1998.

[BD00] D. Bruening and E. Duesterwald. Exploring optimal compilation unit shapes for an embedded just-in-time compiler. *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 13–20, 2000.

[BDA01] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for Windows. *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001.

[BDB99] V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization: The design and implementation of dynamo. *Hewlett Packard Laboratories Technical Report HPL-1999-78. June 1999*, 1999.

[BDB00] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, 2000.

[BDC98] R.G. Burger, R.K. Dybvig, and B. Coulter. An infrastructure for profile-driven dynamic recompilation. *Proceedings of the 1998 International Conference on Computer Languages*, pages 240–249, 1998.

[BFPV99] David Basin, Stefan Friedrich, Joachim Posegga, and Harald Vogt. Java byte code verification by model checking. In *11th International Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 491–494, Trento, Italy, July 1999. Springer-Verlag.

[BH02] M. Berndl and L. Hendren. Dynamic profiling and trace cache generation for a Java virtual machine. Technical report, McGill University, 2002.

[BH03] M. Berndl and L. Hendren. Dynamic profiling and trace cache generation. *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 276–285, 2003.

[BL96] T. Ball and J.R. Larus. Efficient path profiling. *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 46–57, 1996.

[BL00] T. Ball and JR Larus. Using paths to measure, explain, and enhance program behavior. *Computer*, 33(7):57–65, 2000.

[BP99] Gianfranco Bilardi and Keshav Pingali. The static single assignment form and its computation. Technical report, Department of Computer Science, Cornell University, Jul 1999.

[BP03] G. Bilardi and K. Pingali. Algorithms for computing the static single assignment form. *Journal of the ACM (JACM)*, 50(3):375–425, may 2003.

[Bra04] B.J. Bradel. *The Use of Traces in Optimization*. PhD thesis, University of Toronto, 2004.

[Bur97] R.G. Burger. *Efficient compilation and profile-driven recompilation in Scheme*. PhD thesis, Department of Computer Science, Indiana University, 1997.

[BWC$^+$99] M.G. Burke, J. Whaley, J.D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M.J. Serrano, VC Sreedhar, and H. Srinivasan. The jalapeño dynamic optimizing compiler for java. *Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141, 1999.

[CEG$^+$05] M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. The open runtime platform: a flexible high-performance managed runtime environment. *Concurrency and Computation Practice and Experience*, 17(5-6):617–637, 2005.

[CFR+91] 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, October 1991.

[Cha92] Craig Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, April 1992.

[CLCG00] W.K. Chen, S. Lerner, R. Chaiken, and D.M. Gillies. Mojo: A dynamic optimization system. *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, pages 81–90, 2000.

[CLS02] M. Cierniak, B.T. Lewis, and J.M. Stichnoth. Open runtime platform: flexibility with performance using interfaces. *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 156–164, 2002.

[CMC+91] P.P. Chang, S.A. Mahlke, W.Y. Chen, N.J. Warter, and W.H. Wen-mei. IMPACT: an architectural framework for multiple-instruction-issue processors. *Proceedings of the 18th annual international symposium on Computer architecture*, pages 266–275, 1991.

[CMH91] P.P. Chang, S.A. Mahlke, and WW Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21(12):1301–1321, 1991.

[CNO+88] R.P. Colwell, R.P. Nix, J.J. O'Donnell, D.P. Papworth, and P.K. Rodman. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on Computers*, 37(8):967–979, 1988.

[Cog01] A. Coglio. Improving the official specification of Java bytecode verification. In *Proceedings of 3rd ECOOP Workshop on Formal Techniques for Java Programs*, June 2001.

[Cog04] A. Coglio. Simple verification technique for complex java bytecode subroutines. *Concurrency and Computation: Practice & Experience*, 16(7):647–670, 2004.

[Coh97] Richard M. Cohen. The defensive Java Virtual Machine specification version 0.5. Technical report, Computational Logic, Inc., May 1997.

[CUL91] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self, a dynamically-typed object-oriented language based on prototypes. *Higher-Order and Symbolic Computation*, 4(3):243–281, 1991.

[DGK$^+$04] Mourad Debbabi, Abdelouahed Gherbi, Lamia Ketari, Chamseddine Talhi, Nadia Tawbi, Hamdi Yahyaoui, and Sami Zhioua. A dynamic compiler for embedded Java virtual machines. In *PPPJ '04: Proceedings of the 3rd international symposium on Principles and practice of programming in Java*, pages 100–106. Trinity College Dublin, 2004.

[DMT05] Mourad Debbabi, Azzam Mourad, and Nadia Tawbi. Armed E-Bunny: a selective dynamic compiler for embedded java virtual machine targeting ARM processors. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 874–878, New York, NY, USA, 2005. ACM Press.

[DMTY05] M. Debbabi, A. Mourad, C. Talhi, and H. Yahyaoui. Accelerating embedded Java for mobile devices. *Communications Magazine, IEEE*, 43(9):80–85, 2005.

[Ell84] J.R. Ellis. *A Compiler for VLIW Architectures*. PhD thesis, Yale University, 1984.

[Ell86] J.R. Ellis. *Bulldog: a compiler for VLSI architectures*. MIT Press Cambridge, MA, USA, 1986.

[FGL94] Stefan M. Freudenberger, Thomas R. Gross, and P. Geoffrey Lowney. Avoidance and suppression of compensation code in a trace scheduling compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1156–1214, 1994.

[Fis81] J.A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, 1981.

[Fis93] J.A. Fisher. Global code generation for instruction-level parallelism: Trace scheduling-2. Technical report, Hewlett-Packard Laboratories, 1993.

[FKR$^+$00] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An optimizing compiler for Java. *Software-Practice and Experience*, 30(3):199–232, March 2000.

[FM99] S. N. Freund and J. C. Mitchell. Specification and verification of java bytecode subroutines and exceptions. Technical Report CS-TN-99-91, Standford University, 1999.

[FM03] Stephen N. Freund and John C. Mitchell. A type system for the Java bytecode language and verifier. *Journal of Automated Reasoning*, 30(3-4):271–321, 2003.

[FPP97] DH Friendly, S.J. Patel, and YN Patt. Alternative fetch and issue policies for the trace cache fetch mechanism. *Proceedings of the Thirtieth Annual*

IEEE/ACM International Symposium on Microarchitecture, pages 24–33, 1997.

[FPP98] DH Friendly, SJ Patel, and YN Patt. Putting the fill unit to work: dynamic optimizations for tracecache microprocessors. *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-31)*, pages 173–181, 1998.

[Fre98] Stephen N. Freund. The costs and benefits of Java bytecode subroutines. In *Proceedings of the Formal Underpinnings of Java Workshop at OOPSLA 1998*, oct 1998.

[GBF97] R. Gupta, D. Berson, and J.Z. Fang. Path profile guided partial dead code elimination using predication. *International Conference on Parallel Architectures and Compilation Techniques (PACT'97)*, pages 102–115, 1997.

[GBF98] R. Gupta, D. A. Berson, and J. Z. Fang. Path profile guided partial redundancy elimination usingspeculation. *Proceedings of the 1998 International Conference on Computer Languages*, pages 230–239, 1998.

[GH03] E. Gagnon and L. Hendren. Effective inline-threaded interpretation of Java bytecode using preparation sequences. *Compiler Construction, 12th International Conference, Jan*, 2003.

[Gol98] A. Goldberg. A specification of java loading and bytecode verification. *Proceedings of the 5th ACM Conference on computer and communications security*, pages 49–58, 1998.

[GPF03] Andreas Gal, Christian W. Probst, and Michael Franz. A denial of service attack on the Java bytecode verifier. Technical Report 03-23, University of California, Irvine, School of Information and Computer Science, 2003.

[GPF05a] Andreas Gal, Christian W. Probst, and Michael Franz. Integrated Java bytecode verification. In *First International Workshop on Abstract Interpretation of Object Oriented Languages*, January 2005.

[GPF05b] Andreas Gal, Christian W. Probst, and Michael Franz. Structural encoding of static single assignment form. In *Proceedings of the 4th International Workshop on Compiler Optimization Meets Compiler Verification (COCV 2005)*, Amsterdam, The Netherlands, April 2005. Elsevier Science Publishers.

[GPF06a] Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE '06)*, pages 144–153, New York, NY, USA, 2006. ACM Press.

[GPF06b] Andreas Gal, Christian W. Probst, and Michael Franz. Java bytecode verification via static single assignment form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, to appear, 2006.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1983.

[Han74] Gilbert J. Hansen. *Adaptive Systems for the Dynamic Run-Time Optimization of Programs*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, March 1974.

[Han96] R.E. Hank. *Region-based compilation*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[HCU92] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992* *conference on Programming Language Design and Implementation (PLDI)*, pages 32–43, New York, NY, USA, 1992. ACM Press.

[HHR95] R.E. Hank, W.M.W. Hwu, and B.R. Rau. Region-based compilation: an introduction and motivation. *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 158–168, 1995.

[HMB$^+$93] Richard E. Hank, Scott A. Mahlke, Roger A. Bringmann, John C. Gyllenhaal, and Wen mei W. Hwu. Superblock formation using static program analysis. In *MICRO 26: Proceedings of the 26th annual international symposium on Microarchitecture*, pages 247–255, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[HMS87] M.A. Howland, R.A. Mueller, and P.H. Sweany. Trace scheduling optimization in a retargetable microcode compiler. *Proceedings of the 20th annual workshop on Microprogramming*, pages 106–114, 1987.

[Höl94] U. Hölzle. *Adaptive optimization for self: reconciling high performance with exploratory programming*. PhD thesis, Stanford University, Department of Computer Science, 1994.

[IBM06] IBM. WebSphere Everyplace Custom Environment J9 Virtual Machine. `http://www-306.ibm.com/software/wireless/wece/`, October 2006.

[Ins02] Insignia Solutions. Jeode Platform: Java for Resource-constrained Devices, White Paper. `http://www.insignia.com/`, 2002.

[JRS97] Q. Jacobson, E. Rotenberg, and J.E. Smith. Path-based next trace prediction. *Proceedings of the 30th International Symposium on Microarchitecture*, pages 14–23, 1997.

[KF01] T. Kistler and M. Franz. Continuous program optimization: Design and evaluation. *IEEE Transactions on Computers*, 50(6):549–566, 2001.

[KF03] T. Kistler and M. Franz. Continuous program optimization: A case study. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(4):500–548, 2003.

[KG97] Andreas Krall and Reinhard Grafl. CACAO: A 64 bit Java VM just in time compiler. In *PPoPP Workshop on Java for Science and Engineering Computation*, 1997.

[Kis99] Thomas Kistler. *Continuous Program Optimization*. PhD thesis, Department of Information and Computer Science, University of California, Irvine, November 1999.

[KN03] Gerwin Klein and Tobias Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, April 2003.

[Kri96] Sanjeev Krishnan. *Automating Runtime Optimizations for Parallel Object-Oriented Programming*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[KW03] Gerwin Klein and Martin Wildmoser. Verified bytecode subroutines. *Journal of Automated Reasoning*, 30(3-4):363–398, 2003.

[LC02] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, Oct. 2002.

[Ler01a] X. Leroy. Java bytecode verification: an overview. *Computer Aided Verification, CAV*, pages 265–285, 2001.

[Ler01b] X. Leroy. On-card bytecode verification for Java card. *International Conference on Research in Smart Cards (E-smart)*, pages 19–21, 2001.

[Ler02] X. Leroy. Bytecode verification for Java smart card. *Software Practice & Experience*, 32:319–340, 2002.

[Ler03] X. Leroy. Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning*, 30(3/4):235–269, 2003.

[LFK$^+$93] P.G. Lowney, S.M. Freudenberger, T.J. Karzes, WD Lichtenstein, R.P. Nix, J.S. O'Donnell, and J.C. Ruttenberg. The multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7(1):51–142, 1993.

[Lou06] Robert Lougher. JamVM virtual machine 1.4.3. `http://jamvm.sf.net/`, May 2006.

[LTS01] Christopher League, Valery Trifonov, and Zhong Shao. Functional Java bytecode. In *Proceedings of the 5th World Conference on Systemics, Cybernetics, and Informatics*, July 2001. Workshop on Intermediate Representation Engineering for the Java Virtual Machine.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison Wesley Longman, Inc., second edition, 1999.

[LZQcJ03] Yang Liu, Zhaoqing Zhang, Ruliang Qiao, and Roy Dz ching Ju. A region-based compilation infrastructure. In *INTERACT '03: Proceedings of the Seventh Workshop on Interaction between Compilers and Computer Architectures*, page 75, Washington, DC, USA, 2003. IEEE Computer Society.

[MCH99] J. A. Mathew, P. D. Coddington, and K. A. Hawick. Analysis and development of Java Grande benchmarks. In *Proceedings of the ACM 1999 Java Grande Conference, San Francisco*, 1999.

[MDSW88] R. A. Mueller, M. R. Duda, P. H Sweany, and J. S. Walicki. Horizon: a retargetable compiler for horizontal microarchitectures. *IEEE Transactions on Software Engineering*, 14(5):575–583, 1988.

[MF97] G. McGraw and E.W. Felten. *Java security: hostile applets, holes & antidotes*. John Wiley & Sons, Inc. New York, NY, USA, 1997.

[NL96] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 229–243, Berkeley, CA, USA, 1996. USENIX.

[Nov03] D. Novillo. Tree SSA, a new optimization infrastructure for GCC. *Proceedings of the 2003 GCC Developers' Summit*, pages 181–193, 2003.

[Nov04] D. Novillo. Design and implementation of Tree SSA. *Proceedings of the GCC Developer's Summit*, 2004.

[NPW02] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL*. Springer New York, 2002.

[NR01] George C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. *ACM SIGPLAN Notices*, 36(3):142–154, 2001.

[O'C99] Rober O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 70–78, San Antonio, Texas, 1999.

[PFP99] S. J. Patel, D. H. Friendly, and Y. N. Patt. Evaluation of design options for the trace cache fetch mechanism. *IEEE Transactions on Computers*, 48(2):193–204, 1999.

[PGF05] Christian W. Probst, Andreas Gal, and Michael Franz. Average case vs. worst case: margins of safety in system design. In *NSPW '05: Proceedings of the 2005 Workshop on New Security Paradigms*, pages 25–32, New York, NY, USA, 2005. ACM Press.

[Pus99] Cornelia Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. *Lecture Notes in Computer Science*, 1579:89–103, 1999.

[Qia99] Zhenyu Qian. A formal specification of Java virtual machine instructions for objects, methods and subrountines. In *Formal Syntax and Semantics of Java*, pages 271–312, 1999.

[Qia00] Zhenyu Qian. Standard fixpoint iteration for Java bytecode verification. *ACM Transactions on Programming Languages and Systems*, 22(4):638–672, 2000.

[RBS99] E. Rotenberg, S. Bennett, and JE Smith. A trace cache microarchitecture and evaluation. *IEEE Transactions on Computers*, 48(2):111–120, 1999.

[Ros98] E. Rose. Towards secure bytecode verification on a Java Card. *Master's thesis, University of Copenhagen*, 1998.

[Ros03] E. Rose. Lightweight bytecode verification. *Journal of Automated Reasoning*, 31(3):303–334, 2003.

[SA98] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Conference Record of POPL 98: The 25TH ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, pages 149–160, New York, NY, 1998.

[SA99] Raymie Stata and Martin Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, 1999.

[SDJ84] B. Su, S. Ding, and L. Jin. An improvement of trace scheduling for global microcode compaction. *ACM SIGMICRO Newsletter*, 15(4):78–85, 1984.

[SG95] Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ-nodes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 62–73, San Francisco, California, 1995.

[SGL94] V. Sreedhar, G. Gao, and Y. Lee. DJ-graphs and their applications to flowgraph analyses. Technical Report ACAPS Memo 70, McGill University, May 1994.

[Sha02] N. Shaylor. A just-in-time compiler for memory-constrained low-power devices. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*, pages 119–126, San Francisco, CA, August 2002.

[SJGS99] V.C. Sreedhar, R.D.C. Ju, D.M. Gillies, and V. Santhanam. Translating out of static single assignment form. *Proceedings of the 6th International Symposium on Static Analysis*, 1694, 1999.

[Smi00] M.D. Smith. Overcoming the challenges to feedback-directed optimization. *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00)*, 2000.

[SOK$^+$04] T. Suganuma, T. Ogasawara, K. Kawachiya, M. Takeuchi, K. Ishizaki, A. Koseki, T. Inagaki, T. Yasue, M. Kawahito, T. Onodera, et al. Evolution of a java just-in-time compiler for ia-32 platforms. *IBM Journal of Research and Development*, 48(5/6):767–795, 2004.

[SOT$^+$00] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the ibm java just-in-time compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[SS70] R. M. Shapiro and H. Saint. The representation of algorithms. Technical Report CA-7002-1432, Massachusetts Computer Associates, Feb 1970.

[SS01] R. Stärk and J. Schmid. Java bytecode verification is not possible. In R. Moreno-Díaz and A. Quesada-Arencibia, editors, *Formal Methods and Tools for Computer Science (Proceedings of Eurocast 2001*, pages 232–234, Canary Islands, Spain, February 2001. Universidad de Las Palmas de Gran Canaria.

[SS03] R. F. Stärk and J. Schmid. Completeness of a bytecode verifier and a certifying Java-to-JVM compiler. *Journal of Automated Reasoning*, 30(3–4):323–361, 2003.

[Sun99] Sun Microsystems. KVM-Kilobyte virtual machine white paper. Technical report, Sun Microsystems, Palo Alto, CA, 1999.

[Sun01] Sun Microsystems. KVM porting guide. Technical report, Sun Microsystems, Palo Alto, CA, September 2001.

[Sun02] Sun Microsystems. The Java Hotspot Virtual Machine v1.4.1, September 2002.

[Sun05] Sun Microsystems. CLDC HotSpot Implementation Virtual Machine. `http://java.sun.com/j2me/docs/pdf/CLDC-HI_whitepaper-February_2005.pdf`, February 2005.

[SYN03] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. A region-based compilation technique for a java just-in-time compiler. *ACM SIGPLAN notices*, 38:312–323, 2003.

[SYN06] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. A region-based compilation technique for dynamic compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(1):134–174, 2006.

[vR05] Jeffrey von Ronne. *A Safe and Efficient Machine-Independent Code Transportation Format Based on Static Single Assignment Form and Applied to Just-In-Time Compilation*. PhD thesis, University of California, August 2005.

[VRCG$^+$99] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a Java bytecode optimization framework. *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, 1999.

[VRH98] R. Vallee-Rai and L.J. Hendren. Jimple: Simplifying Java bytecode for analyses and transformations. Technical report, Sable Research Group, McGill University, 1998.

[Wha01] J. Whaley. Partial method compilation using dynamic profile information. *ACM SIGPLAN Notices*, 36(11):166–179, 2001.

[Wil06] T. Wilkinson. Kaffe–a Java virtual machine. `http://www.kaffe.org/`, October 2006.

[Yel95] F. Yellin. Low level security in Java. *Proceedings of the 4th International World Wide Web Conference, Boston, Massachusetts, December*, 1995.

[You98] R.C. Young. *Path-based Compilation*. PhD thesis, Harvard University, 1998.

[YS98] C. Young and M.D. Smith. Better global scheduling using path profiles. *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 115–123, 1998.