ANNOTATING MOBILE CODE FOR PERFORMANCE

Draft Version
April 29, 2002

BY

JOEL JONES

B.A., University of Tennessee, 1986
M.S., Arizona State University, 1989

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2002

Urbana, Illinois

Table of Contents

List of Tables

List of Figures

1 Introduction

Java is unquestionably a success. In the space of less than ten years, it has become widely used, widely supported, and widely studied academically. While Java introduced no truly novel ideas, it packaged many good ideas in a well-engineered language system. Given the ubiquity of Java, it is particularly important that Java programs execute quickly in a range of environments. In particular, the initial success of Java was as a language for mobile code, namely in web browsers. Here, the machine-independence and network distribution of Java makes the implementation of an environment to execute Java programs quickly non-trivial.

In its network-portable form, Java programs are packaged as .class files, in which programs are compiled into a collection of files, each representing a single class. The executable code of a class's methods is represented as the bytecodes of a stack-based virtual machine, called the Java Virtual Machine (JVM). When the class files arrive on the client machine, several approaches can be taken to execute the program. One approach is to interpret the bytecodes. This approach is the simplest to implement, but suffers from mediocre performance. Another approach is to translate the bytecodes into the native machine code of the client machine. This approach — known as "just-in-time," or "JIT" compilations — is now routinely used to improve the performance of Java programs.

The implementation of the translation approach presents many choices. A simple mapping of bytecodes into fixed sequences of machine code is relatively straight-forward to implement, but exploits little of the potential benefit of dealing directly with machine dependent code, such as using machine registers instead of the stack. A more complex approach treats the bytecodes much as a batch compiler treats its intermediate representation (IR). This IR is analyzed, and optimizing transformations are performed on it, eventually producing machine code that is executed by the hardware. This approach yields the best execution time for the generated machine code, but there is a high cost. First, optimizing compilers are complicated and expensive to implement, and a compiler embedded in a JVM is no different. Second, there is a run-time cost for the compilation activity. Many analyses have super-linear cost in time and/or space. Also, more sophisticated analyses require more sophisticated intermediate representations, which add an additional linear overhead over simpler representations, both in creation and in manipulation.

This raises the question: "Can JVMs execute Java programs quickly without

an optimizing compiler?" The answer to this question is "no". Another question is: "Can JVMs execute Java programs quickly without an optimizing compiler *in the VM*?" The answer to this question is "yes". This surprising answer arises from the observation that many machine-dependent optimizations rely on expensive analyses that are primarily machine-independent.

This thesis describes a particular set of optimizations that show a division of labor between machine-independent and machine-dependent parts. In particular, we show optimizations which demonstrate a phenomenon we call "super-linear analysis and linear exploitation." The super-linear analysis is performed off-line in machine-independent fashion. The linear exploitation is machine-dependent and is performed in the JVM. By using this division, we can achieve the effects of an optimizing compiler with essentially no run-time cost.

We realize the division of labor in the following way. The Java .class file format permits the addition of optional attributes. We use these optional attributes to annotate the machine-independent bytecodes with the distillation of information from an optimizing compiler. These annotations are used by an extended JVM to linearly exploit this optimization information in a machine dependent fashion.

The constraints of the network-portable, or mobile code form, drive our research. In Figure 1.1, we see the traditional compilation environment. The source

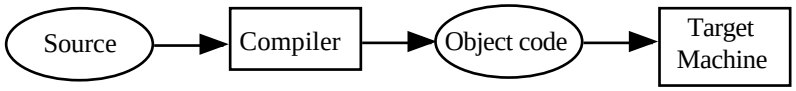

Figure 1.1: Traditional Compilation Environment

file is the input to the compiler which produces object code for a particular target machine. The compiler is unconstrained in the amount of time that it may spend in performing the compilation process, performing analysis and optimization for as long as necessary. The target machine is known, so the compiler will perform machine-dependent optimizations to maximize the speed of the object code on the target machine.

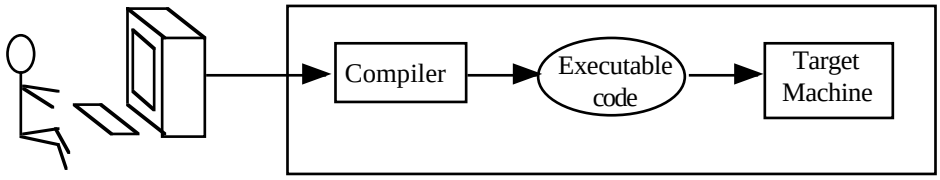

Figure 1.2: Interactive Compilation Environment

In an interactive programming environment, as illustrated in Figure 1.2, the situation is slightly different. The user enters programs interactively and demands

good interactive response. The latency — the time from when the user enters the code to when it executes — must be minimized to obtain interactive responsiveness. The compiler produces executable code, but must do it quickly. The target machine is known, so machine-dependent optimization can be performed. Further, profile information is readily available, and may be used to guide compilation.

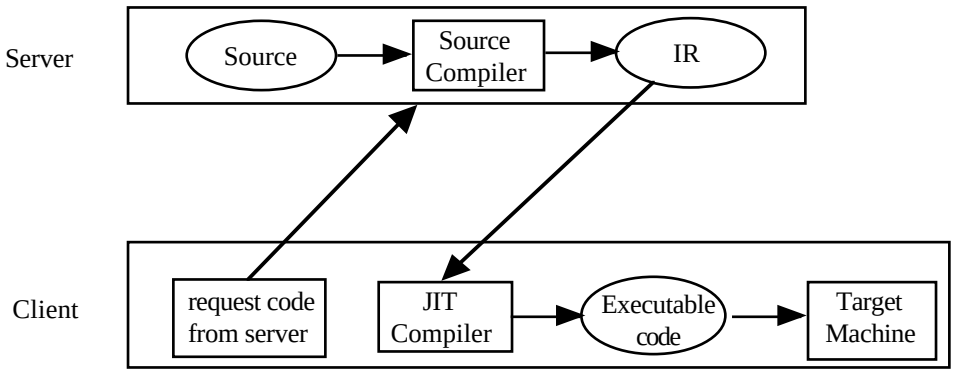

Figure 1.3: Mobile Code Compilation Environment

The mobile code environment, which we are concerned with, is shown in Figure 1.3. In this environment, there is a server that compiles source code into an intermediate form. This intermediate form can then be delivered on demand to a mobile client across a network. The compilation to the intermediate form is unconstrained in the amount of time that it can use. However, it does not know what the target machine will be and therefore cannot do any machine dependent optimizations. The intermediate form is also constrained in its size, as it will be transmitted across a network connection, while the user waits. The client in a mobile code environment has a similar model to the interactive programming environment. There is a compiler from intermediate form into machine code in the mobile client. This compiler also must work quickly to maintain interactive responsiveness. The Java VM is one instance of this mobile code model. Our annotations are added by the compiler in the server and exploited by the compiler in the client.

We demonstrate the usefulness of this annotation-based approach to virtual-machine-based mobile code by examining annotations related to the efficient exploitation of registers. The annotations which we have developed are:

- register assignment
- register spilling
- copies

These annotations allow programs for a stack-based abstract machine to be efficiently executed on a register-based machine.

We show the effectiveness of this approach by showing information gathered from standard Java benchmark suite, the $SPEC_{JVM}$, [13] where reductions in load and store instructions generated range from 7% to 18% when these annotations are used.

Chapter 2 describes in more detail related work on virtual machine implementation and compiler optimizations. Chapter 3 describes in detail the annotations that we have investigated. We take up the topic of how well the annotations perform in Chapter 4. We continue by discussing further extensions of this work in Chapter 5. We conclude the main part of the thesis in Chapter 6 with discussion of the contributions of this research. In the appendix, we give details on the exact VR annotations for each bytecode.

2 Related Work

Mobile execution environments such as Java's present unique challenges and opportunities. We have seen a hint of that in Figure 1.3 and its accompanying description. These challenges and opportunities can be partially addressed by drawing on relevant prior work. In this chapter we will discuss the following areas of related work. First, we will discuss virtual machines. Then we will examine compiler optimizations, in particular the division of labor within a compiler between machine-independent and machine-dependent optimizations and also the problem of register allocation. Next we will discuss JIT compilation, focusing on the tradeoff between complex analysis and interactive response. We conclude by summarizing and foreshadowing our research by contrasting machine-independent analysis with machine-dependent transformation.

2.1 Virtual Machines

A virtual machine is a software structuring technique where programs are expressed in a form that is independent of any particular hardware and which is not directly executable by a machine. This form is at a low-level of abstraction, consisting of sequences of primitive instructions which are close to a machine language. A common execution strategy for these primitive instructions is interpretation, where the instruction is read from memory, dispatched to an appropriate piece of the interpreter code, and executed, the interpreter repeating the "fetch-dispatch-execute" cycle until the program is finished executing.

2.1.1 Rationale for Virtual Machines

Virtual machines are used for various reasons. In the case of Java and UCSD Pascal, they are used to achieve portability and compactness. Java requires compactness to minimize the network transmission time, whereas UCSD Pascal required compactness to fit on the limited memory machines of the late 1970s era. Virtual machines are used in interactive systems because VM code can be generated more quickly than machine code.

The use of virtual machines dates back to the early days of computing, when IBM used a microcode translation system to implement the IBM 1401 instruction set on a completely different underlying hardware architecture, the IBM/360. The

ability to interpret programs originally written for the 1401 allowed IBM's customers to preserve their investment in software, while allowing IBM to explore and deploy different implementations of the IBM/360 instruction set with higher performance for certain classes of programs.

Today, the term "virtual machine" is generally used to describe instruction sets specifically designed as portability bases—a contract between programs written in a source language and the hardware that will eventually interpret it. This meaning first appeared in the 1970's in the Smalltalk and UCSD Pascal P-code systems, and has been popularized in the 1990's with the Java virtual machine. (The term abstract machine is used synonymously, primarily in the logic programming literature.)

Our research is focused on implementation techniques for virtual machines, not virtual machine design. The common design element of all of the virtual machines to be discussed is that the instructions follow a "bytecode" design. In a bytecode design, the instruction encoding has the format of an opcode followed by zero or more operands. These operands are references into a table or to offsets into a run-time stack. This is in contrast to tree- or graph-based systems, which will not be discussed further.[16] In this chapter, we sketch the implementation techniques used in Smalltalk, Self, and Java, emphasizing the transition away from repeated re-interpretation of the VM instructions to execution by compilation to native code.

Our work falls squarely into the "compilation to native code" approach. It differs from the work presented in this chapter primarily in the balance it seeks to strike: whereas the systems presented here either perform aggressive, traditional compilation at potentially high cost or perform fast compilation without much optimization, our work assumes the core information required during optimization can be computed off-line but exploited on-line quickly. Subsequent chapters show how we achieved this balance, and evaluate its performance.

The fundamental problem in implementing virtual machines is that of efficiency. The separation from the target machine that is needed for portability can prove a detriment to performance.

2.1.2 Virtual Machines and Translators

Before discussing the details of VM implementation with just-in-time compilation, we will cover some of the other uses of virtual machines and translators. One example of the use of a bytecode virtual machine is a version of the functional language ML, Objective Caml. [30] Another is Prolog, a logic programming language, which was popularized by an innovative VM design, the Warren Abstract Machine (WAM).[45, 1] Many implementations have been based upon the WAM,

including:

- Aquarius - a hardware based implementation of a WAM-like instruction set.[43]
- Zephyr-an implementation of an extended Prolog with interoperability with other languages.[4]
- Chare-an implementation for parallel environments. [35]

Another related thread of work is binary translation, also known as object-code translation. The goal here is portability of old machine code to new machines. In these systems, the machine-code of one processor is translated at run-time to the machine-code of the current processor. Below are some original-to-host pairs.

- Tandem's Non-stop to MIPS [3]
- Hewlett-Packard's HP3000 to PA-RISC [7]
- Apple's Motorola 68000 to PPC [40]
- Digital's Vax to Alpha [36]
- Transmeta's Intel x86 to Crusoe VLIW [29]

All of these owe much to the Deutsch/Schiffman Smalltalk-80 VM implementation technique of translating to machine code, discussed below. Earlier systems that rehosted machine code relied on hardware and microcode, e.g. IBM 1401 to IBM 360.[6]

Somewhat similar to binary translation is the use of a low-level intermediate language to hide even lower-level details of the processor. This approach is used in many traditional batch compilers which target multiple RISC targets. One of the earliest descriptions of such an approach is the Mahler intermediate language.[44] This language hides such details as the number of registers and delayed branches.

2.2 Compiler Optimization

Compilers in a traditional compilation environment perform extensive optimizations to decrease the execution time of the program on the target machine. Here we discuss the distinction between machine-independent and machine-independent optimizations, the time complexity of optimizations, and intermediate representations. We then examine in detail the register allocation optimization, which is the focus of our research.

2.2.1 Machine-independent and Machine-dependent Optimizations

Compiler optimizations can be divided into two categories: machine-independent and machine-dependent.

A machine-independent optimization is one that can be performed without knowledge of the target machine. Such optimizations transform the program in a way that improves performance without requiring information such as the number of registers or how the execution of one instruction relies on particular hardware resources. One common machine-independent optimization is loop-invariant code motion. This optimization moves code from inside the body of a loop to outside the body of a loop. The code to be moved is analyzed for any dependencies upon values that change during the execution of the loop. If no such dependencies exist, then the code motion is legal. This analysis and transformation do not rely on information specific to the target machine.

A machine-dependent optimization is one that can be performed only with information about the target machine. Such optimizations use information about the resources of the target machine to decide how to transform the code. Such resources might include the number of registers or how instructions are dispatched across multiple functional units. One common machine-dependent optimization is register allocation. This optimization assigns values to hardware registers by analyzing which values are in use at various points in the program and choosing when and where each value should reside in a register. This optimization requires knowledge of the number of registers in the machine and any constraints on the type and size of values that may be placed in each register.

An important point to note is that even machine-dependent transformations will use machine-independent analyses. For example, a common technique for doing register allocation uses an analysis to determine which values are used at which points in the program, called *liveness analysis*, which is machine-independent. It also contains a graph-coloring phase based on with a machine-independent graph representation. This graph contains nodes representing the values of the program and nodes representing the registers of the target machine, so the process is a mixture of machine-dependent and machine-independent aspects.

2.2.2 Cost of Optimizations

Both machine-independent and machine-dependent optimizations consist of two parts: the analyses to determine the applicability of the optimization and the transformation to change the code from its current state to an improved state. The analysis part of optimization is the more expensive of the two. The transformation of the code is linear in the size of the code to be transformed and involves changes

to simple data structures. The analyses for some optimizations are not linear. For example, the common optimization strategy used for register allocation is to view the problem as a variant of graph coloring. [10, 9] The analysis is potentially $O(N^2)$ in the number of values of the program, although it is typically $O(N \lg N)$ in practice, which is much more expensive than the corresponding transformation that changes the code to refer to the assigned physical registers. [9] Another property of analyses is that the data structures they build and use are complicated and therefore, while potentially linear, have a high linear factor. Such analyses include different varieties of data-flow analysis, which tend to use computations over long bit-vectors, and other expensive constructions.

2.2.3 Intermediate Representations

Inside an optimizing compiler, code is represented in several forms. The initial representation is in the form of abstract syntax trees which are built as a result of lexical and syntactic analysis of the source program. The final representation of the code in a traditional compilation environment is the machine code for the target machine. As neither of these representations is suitable for performing optimizations, typically another form is used, called an intermediate representation or IR. These IRs can take many forms, based upon the source language, the target machine, and the type of optimizations that are performed.

Intermediate forms are mostly machine-independent, for two reasons. First, an intermediate form can be tailored to the optimizations to be performed. For example, in our invariant code motion optimization mentioned above, it is useful to have an IR which represents loops directly as a construct in the IR. Such a loop representation would be used by many optimizations which need to analyze code contained in loops.

Second, machine-independent IRs make the compiler useful for more than one target machine, i.e. for portability. By having the optimizing compiler perform most of its work on a form that is machine-independent, the effort to add a new target machine to the compiler is reduced.

As hinted at by the “loop” construct, IRs can be varied and much richer than the instructions of a virtual machine. The IR can take the form of a graph structure with edges directly represented as pointers in the compiler’s implementation language. Such representations are not directly representable in VM bytecodes. The fundamental problem is that such graph structures are not necessarily directly executable, i.e. there may not be an explicit starting point or branching instruction to guide the VM in executing the program as represented by the graph.

2.2.4 Register Allocation

One of the most important optimizations that a batch compiler can perform is register allocation. As mentioned above, register allocation is an optimization that determines which values should reside in machine registers at every point in the program. We present a discussion of two different graph-coloring register allocators, continue with a set of heuristics that can be added to the graph-coloring register allocation algorithms, and conclude with a discussion of another approach to register allocation. This discussion will describe why each analysis is either machine-independent or machine-dependent.

We begin by describing the Chaitin graph coloring register allocator, which forms the basis for our allocator. [10] We will then continue by describing the Briggs enhancement of the basic Chaitin algorithm. [9] Although our algorithm most closely resembles the unenhanced Chaitin algorithm, we will see later that some of our work is based upon ideas from Briggs.

The following definitions are from Muchnick.[32] They will prove useful in the further discussion of register allocation.

reaches A particular definition (i.e. assignment of a value to a variable) is said to *reach* a given point in a procedure if there is an execution path from the definition to that point such that the variable may have, at that point, the value assigned by the definition. Calculating reachability is a machine-independent analysis that is $O(N \cdot A)$ where N is number of basic blocks and A is the number of back edges in a depth-first search of the control-flow graph. Building the control-flow graph and finding back-edges is also machine-independent.

du-chain The du-chain ("definition-use chain") for a variable connects a definition of that variable to all the uses it may flow to, i.e. reaches. This information is machine-independent.

web A web consists of a set of definitions and a set of reached uses. Webs are formed by merging together du-chains that share a use until a fixed point is reached. A web represents the allocatable object for register allocation. Building webs is a machine-independent construction.

liveness For a given variable (here, a web) and a given point in the program, liveness is a predicate that tells whether there is a use of that variable along some path from the point to the exit. Calculating liveness is a machine-independent analysis that is $O(N \cdot A)$ where N is number of basic blocks and A is the maximal number of back edges in the depth-first search of the control-flow graph.

live A value is considered live if at a given point in the program, there is a use of that variable along some path from the point to the exit.

live range A live range consists of the program points where a value is live—it has been defined and there is at least one subsequent use. A web has an associated live range which indicates whether or not the web's value is live at a particular point in the program.

Chaitin Allocator

Chaitin's innovation was viewing register allocation as a graph coloring problem. An *interference graph* is an undirected graph where each node is a web from the program and the edges are between webs which are simultaneously live. The goal is to find a *k*-coloring of the interference graph, where *k* is the number of physical registers. Chaitin's algorithm uses a heuristic to find a *k*-coloring for the original interference graph. If a *k*-coloring cannot be made using the coloring heuristic (the simplify step below), then the graph is modified by removing a node and its incident edges from the graph, thereby "spilling" the web's value, which relegates the value to being stored outside of a register. The heuristic then continues, trying to find a *k*-coloring for the smaller graph. If a *k*-coloring is found, then the coloring of the interference graph indicates which values should be assigned to which physical registers. This correspondence between physical registers and colors is obtained by inserting the physical registers in the interference graph in addition to the webs. Edges connecting all of the physical register nodes to each other are also added, indicating that a physical register is not equivalent to any other physical register. When the algorithm finishes, the physical registers will all be assigned a color. A web will be allocated to the physical register whose color matches its own.

Building the interference graph is a machine-independent operation, as it uses knowledge about the number and type of registers of the target machine. Building the interference graph takes $O(N \cdot E)$ time, where N is the number of webs and physical registers and E is the number of interference edges.

Chaitin Allocator Stages The following description of the Chaitin algorithm is due to Briggs. [9] See Figure 2.1 for the details of the control-flow of Chaitin's algorithm.

Renumber Find all the live ranges in a method and construct the webs based upon whether the live range has been spilled or not. Live ranges are used to find webs on the first execution of this stage. Subsequent executions may "renumber" the webs to correspond to the current number of webs that are

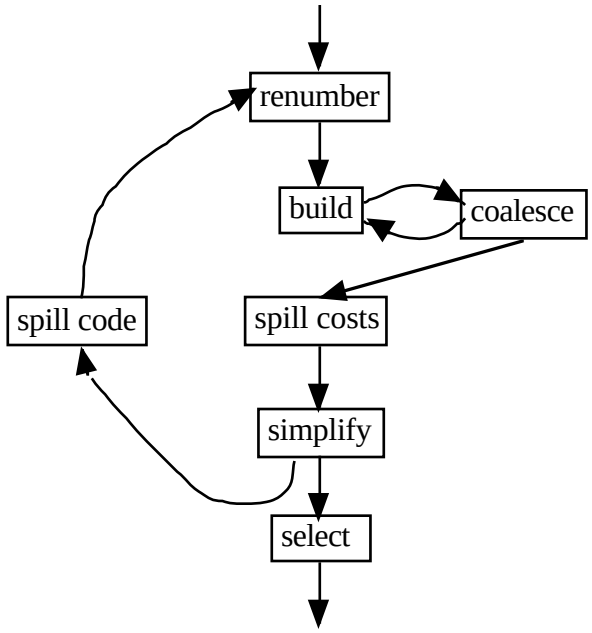

Figure 2.1: Chaitin's Allocation Algorithm

being allocated to registers. Spilling may have occured due to the execution loop as seen in Figure 2.1

Build Construct the interference graph.

Coalesce Remove unneeded copies. A copy is an instruction whose effect is a simple assignment of the value of one web to another. That instruction is not needed if both webs have been assigned the same color.

Spill Costs For every live range, calculate an estimate of the cost of the load and store instructions that would be required to spill it.

Simplify Repeatedly remove nodes from the interference graph with degree $< k$. As each node is removed, place it on a stack. If at any point, no node has degree $< k$, then one of them is marked for spilling and removed from the interference graph.

Select Pop each node from the stack, and insert it back into the interference graph, giving it a color distinct from its neighbors.

Spill Code Any nodes which were marked for spilling in the Simplify stage have code generated for loading and storing around the represented live range.

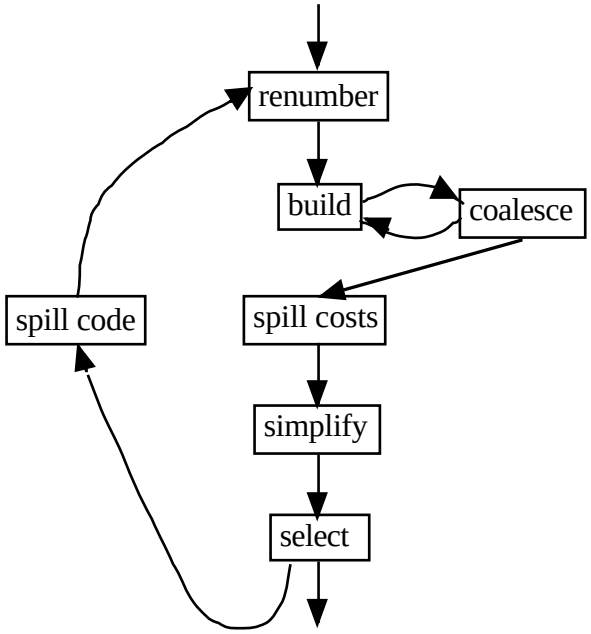

Figure 2.2: Briggs Allocation Algorithm

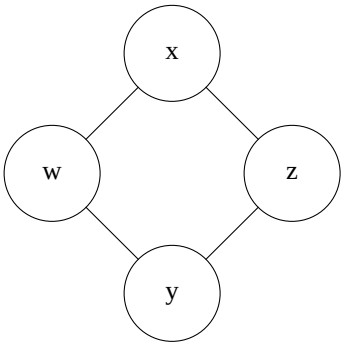

Figure 2.3: Simple Graph Requiring Two Colors

Briggs Allocator

The Briggs allocator has a similar structure to the Chaitin allocator, (see Figure 2.2).

One of the primary problems that the Briggs allocator attacks is the inability of the Chaitin allocator to find a *k*-coloring for the original graph in certain kinds of situations. The two problems are as follows:

1. A simple diamond graph (see Figure 2.3) will not receive a 2-coloring, even though one is possible by inspection. The problem here is that the Chaitin allocator approximates "find a color for node x" with "degree(x) $<$ k." This is a sufficient but not necessary condition.

2. Lives ranges may be chosen to be spilled non-productively. Spilling is a decision that cannot be retracted. Therefore, once a live range is spilled, it remains spilled.

Briggs addresses these two problems of the Chaitin allocator by making modifications to the basic algorithm. Briggs makes the following changes to two stages:

Simplify Repeatedly remove nodes from the interference graph with degree $< k$. As each node is removed, place it on a stack. If at any point, no node has degree $< k$, then one is chosen and it is optimistically placed on the stack, marked for spilling and removed from the interference graph.

Select Pop each node from the stack, and insert it back into the interference graph, giving it a color distinct from its neighbors. If any nodes are left un-colored, then the allocator inserts spill code for the corresponding live ranges, rebuilds the interference graph, and tries again.

These two changes allow more graphs to be k-colored, thereby improving the register allocation of the underlying code.

Another area in which Briggs improves on the work of Chaitin is by the introduction of aggressive live range splitting. The extent of a live range extends from its def(s) to its use(s). As a value's live range may extend across regions where the value is not accessed, it can be profitable to split a single live range into two or more pieces, thereby allowing other values an opportunity at the split range's registers. We will see in our description of the swap annotation how we use the ideas behind live range splitting to generate the swap annotation.

Although both Chaitin's and Briggs' graph-coloring register-allocators can take $O(N^2)$ time in the worst case, Briggs measured the time complexity as $O(N \lg N)$, where N is the number of nodes in the interference graph.

Haifa Heuristics

In any register allocation algorithm, a critical decision is which live range to spill when there is a choice among many. In work done at IBM's Haifa, Israel facility, spill candidates are chosen by calculating the cost associated with spilling using three heuristics and choosing the best of the three.[8] Below are definitions for the

heuristics:

$$
\begin{aligned}
defwt &- \text{cost of a def instruction} \\
cost(w) &- \text{cost of spilling a web } w \\
depth(def) &- \text{loop nest of definition } def \\
degree(w) &- \text{degree of web } w \text{ in interference graph} \\
width(I) &- \text{number of live ranges at instruction } I \\
cost(w) &= defwt \cdot \sum_{def \in w} 10^{depth(def)} \\
h_1(w) &= \frac{cost(w)}{degree(w)^2} \\
area(w) &= \sum_{I \in inst(w)} (width(I) \cdot 5^{depth(I)}) \\
h_2(w) &= \frac{cost(w)}{area(w) \cdot degree(w)} \\
h_3(w) &= \frac{cost(w)}{area(w) \cdot degree(w)^2}
\end{aligned}
$$

The goal of these heuristics (h_1, h_2, h_3) is to capture both the importance of a particular value (cost) and to quantify its interference with other values (area and degree). To better understand the process, here is the pseudocode for their register allocation algorithm:

```
for each heuristic h_i do
  while G is not empty
    if there is an X with degree < r then
      choose that X with largest degree
      delete X
    else
      choose X with Min h_i(X)
      add X to spill_list
    end
    restore G
  end for
choose heuristic h_i with smallest Cost(spill_list)
```

In addition to using this algorithm, the heuristics can be substituted into the *simplify* step of Chaitin's algorithm for choosing which web to spill.

These heuristics are machine-dependent, as they rely on *defwt*, which is the cost of a machine instruction, and on *cost* which is the cost of spilling a web. The heuristics have time complexity of $O(E \lg N)$ — E is the number of edges and N is

the number of basic blocks, derived from the need to find loops in calculating *cost*.

Linear Scan Allocator

Other register allocators exist which are not based upon the graph-coloring model. As we will see below, the linear scan register allocator is chosen by some JVM implementors. The linear scan allocator is a good choice, because it is fast and reasonably effective.

As described by Poletto and Sarkar, the linear scan register allocation algorithm takes as input the number of registers to be allocated and the live interval for every variable. A strict ordering is imposed on the pseudo-instructions of the intermediate form.[34] A *live interval* is a conservative estimate of the corresponding live range, expressed as an interval over the ordering of the intermediate form.

The algorithm proceeds by taking each live interval in increasing start-time order and considering the number of active live ranges at that program point. If there are not enough registers to contain the current set of live intervals, then the interval whose endpoint is furthest away is heuristically chosen to be spilled.

This algorithm works in linear time in the number of live intervals, assuming that the number of registers is fixed.[1] It also achieves results comparable to that of a graph-coloring based register allocator. The algorithm is machine-dependent, as it relies on knowing the number of registers the target machine has. It also relies on having performed liveness analysis.

2.3 JIT compilation

One of the mostly widely used solutions to the efficiency problem of virtual machines is to use a translator of the virtual machine's instructions to the machine code of the target machine. This technique, recently labeled as "just-in-time" or JIT compilation, can substantially improve performance as compared to an interpreted approach.

We begin by describing one of the oldest examples of a bytecoded VM which uses just-in-time compilation, the Xerox PARC Smalltalk system. We then discuss another virtual machine's dynamic compilation system, that of the Self system. We then proceed to discuss some of the current implementations of the Java virtual machine which use dynamic, optimized code-generation technology. The previous systems are then used to guide a discussion of the trade-offs in JIT compilation. We will also make a comparison of VM instructions (bytecode) and the intermediate representations (IR) used in traditional compilation environments.

[1] The algorithm is $O(n \ln k)$ where n is the number of live ranges and k is the number of registers.

2.3.1 Smalltalk

The Smalltalk programming language was developed between 1971 and 1980 at the Xerox Palo Alto Research Center (Xerox PARC). Co-developed with many innovative features of modern computer usage such as laser printers, local-area networking, and the graphical-user-interface, the Smalltalk system also helped to pioneer object-oriented programming languages.

Several versions of the Smalltalk system were designed and implemented.

Smalltalk-76

The Smalltalk-76 system was implemented using a bytecode architecture on the Dorado.[25] The Dorado was an $80,000 single-user micro-coded minicomputer. There are two innovative implementation techniques of the Smalltalk-76 system we are interested in: compact object code and message handling.

Compact Object Code The machine micro-architecture of the Dorado is that of a writable microcode store. The macro-architecture consists of a compact, high-level bytecode of Smalltalk specific instructions. The bytecode consists of a small variety of loads, message sends, jumps, and control statements (return, pop, and store). An escape bytecode also exists for doing such things as extended address loads. Here we see an example of a hybrid of machine code and microcode. Most bytecodes are handled by microcode, while the escape bytecode causes machine code instructions to be executed.

Message Handling To achieve performance while maintaining the "everything is an object" paradigm, certain message sends are implemented as bytecodes. For example, the "+" message is assigned a special bytecode. The receiver and argument are both checked to insure they are instances of SmallInteger, and if not, normal message dispatch is done. This is an example of software based bytecode specialization.

Smalltalk-80

Smalltalk-80 is a microprocessor based implementation of the Smalltalk-80 system, described by Deutsch and Schiffman.[14] The three main innovations in this system are:

- dynamic bytecode to machine code translation
- caching to reduce the cost of method lookups

	Hardware		Software	
type checking	tagged integers two-tone instructions	26% 16%		
interpretation	byte insert/extract instructions	33%	compiling to RISC	100%
procedure calls	register windows fast shuffle	46% 11%	in-line cache	33%
storage management			direct pointers generation scavenging	20% ?%
Total Improvement		132%		153%

Table 2.1: SOAR Performance Contributions: Performance measurements are percentage speed increase when features are considered in isolation. For example, a system with tagged integers performs 26% faster than a system that is identical except for the absence of tagged integers. Total improvement is for all features being present.

- multiple representations of contexts (activation records)

This VM implementation was the first to use translation to machine code without hardware assists. The machine code translator was largely macro driven, with each bytecode translating into a fixed sequence of machine code instructions. The stack nature of bytecodes was emulated by using the machine's memory in a stack fashion. To improve performance, the machine code translator generated code to keep the top element of the execution stack in a register. Another performance enhancing technique was to in-line the generated code rather than treating each bytecode as a method call and generating a jump-to-subroutine sequence. The machine-code was re-generated as necessary and was not paged-out.

SOAR

One of the more influential pieces of research work involving the compilation of bytecodes into machine code was the "Smalltalk On A RISC" (SOAR) project at the University of California at Berkeley.[33] Building on the success of machine code generation in the Smalltalk-80 VM implementation, this research group performed a complete system design process incorporating RISC processor design as well as run-time system software development. Table 2.1, adapted from Ungar's Ph.D. dissertation, summarizes the contribution of various components to the overall performance of the system.[42]

As can be seen by the summary row, both software and hardware features contributed significantly to the performance of SOAR. Here is a brief description of

hardware and software features respectively. First, the hardware.

tagged integers special opcodes for doing arithmetic operations on tagged quantities

two-tone instructions the encoding of an instruction determines whether or not it is constrained by tags.

byte insert/extract instructions needed due to the absence of byte-addressing.

register windows large, on-chip register files, arranged as a set of overlapping windows. Calls and returns manipulate a base-register into the register file, rather than manipulating the in-memory stack.

fast shuffle encoding the entire address for a call in the instruction itself, allowing for one-cycle calls.

Here we see an example of the direct hardware execution model of VM implementation.

Second, the software.

compiling to RISC compilation of Smalltalk-80 bytecodes to SOAR machine-code. As the system described in Ungar's thesis is a simulator, this was done offline.

in-line cache in-line method lookup caches, where the results of searching for the implementor of a polymorphic message send is cached in line, from Schiffman and Deutsch.[14]

direct pointers removal of the object table, so that object references are the addresses of the objects they refer to.

generation scavenging a garbage-collection scheme that creates new objects in a "new" generation and tenures those objects from one generation that are live into the next older generation. (This line in Table 2.1 is marked with a question mark, as the algorithm had not been implemented at the time of Ungar's thesis.)

Due to the unavailability of the SOAR chip, a static translation process was used to convert a normal Smalltalk image into one with the bytecodes translated to SOAR machine code. This machine-coded image was then run through a SOAR simulator to obtain the performance numbers above. This work is most influential in two ways. First is the introduction of register windows, which can be seen in the SPARC and EPIC architectures. Second is the generational garbage-collector.

Other Smalltalk Implementations

Software implementation techniques other than direct machine code generation have been used for Smalltalk VMs. Most notable is the Typed Smalltalk work done at the University of Illinois by Ralph Johnson and his students. In this work, source is either translated into bytecodes and interpreted or translated into more traditional compilation environment internal representations, optimized, and translated into machine code.[27] They solve the problem of maintaining interactive response by having optimized compilation being done on explicit request by the programmer and/or by a background thread when the system is idle. They achieve portability by having a framework that manages machine-dependencies. Although the decision to apply optimizations is under direct programmer control, this is really the same as dynamic optimization. Work was also done to support variations in the run-time system.[15] This allows the VM implementor to easily experiment with changes in the run-time structures.

Squeak is another, more recent implementation of Smalltalk.[24] This implementation follows closely the implementation as documented in the "Blue Book." [18] There, the Smalltalk VM, including a bytecode interpreter, is described in Smalltalk. To gain efficiency, this is typically a normative description of the semantics, with the actual implementation language being assembly or C. In Squeak, this Smalltalk code is actually used. This code is written in a subset of Smalltalk which is automatically translated into C and compiled. In addition, a variant of Ungar's generation scavenging garbage collector is used, along with elimination of the object table by using direct pointers and optimized representation of object headers to decrease per object storage costs. Although static, the automatic translation of Smalltalk code into C is similar to dynamic optimization.

2.3.2 Self

Self is a dynamic object-oriented language. Like Smalltalk, Self is highly integrated into its development environment. In his 1995 Ph.D. dissertation, Urs Hölzle describes the implementation of a Self system.[21] Like the JVM, compiled Self programs are defined in terms of a set of bytecodes for a virtual machine. However, in Hölzle's version of Self, the bytecodes are translated into another intermediate form after performing high-level optimizations such as type feedback-based inlining and splitting. The resulting intermediate code is used from there on.

In this Self system, the execution environment provides for dynamic recompilation. A dynamic recompilation system may generate machine code several times for the same bytecodes. In each recompilation, more optimizations are applied in generating machine code. For Self, the primary optimization is polymorphic

in-line caching (PIC). Object-oriented languages have several choices when generating polymorphic message sends. Languages with a more static nature, such as C++ and Java, typically use vtables, where each message send is accomplished by jumping indirectly through a fixed offset into a table of method pointers. More dynamic environments like Smalltalk or Self, where compile time and execution are intermingled, must use "method lookup," where the actual method to be executed is searched for at every message send by traversing the inheritance hierarchy. Like Smalltalk, Self reduces the need for this search by caching the results of a previous message lookup.

A useful variant of this technique is to inline the called method. One advantage of inlining the called method is to eliminate the overhead of doing a function call. However, the biggest win is by increasing the effectiveness of traditional optimizations.

When a method is inlined, the code for the calling method becomes larger, and more information about the called method is exposed. The Self compiler can then perform optimizations on the expanded method, potentially exposing transformations previously hidden by the message send. Below are the optimizations the Self system makes and a brief explanation of why they are particularly useful in the context of inlining:

- copy-propagation – argument(s) from the caller can be directly substituted as an in the callee, without making a copy.
- dead-code elimination – information exposed by inlining on the values of expressions used in conditional control-flow can lead to unreachable code that can be eliminated.
- register allocation – spills of arguments and locals of the caller can be reduced.

To perform these optimizations, the Self compiler does a definition/use analysis and calculates loop nesting. The register allocator used is a mixed allocator, with local allocation of registers being followed by a global allocation on any remaining values. Usage counts and loop nesting are used to do the register allocation.

Many of the implementation techniques first explored in Self have made their way into Java VM implementations, notably, Sun's HotSpot. [39]

2.3.3 Java

As described in Chapter 1, Java is "hot." In addition to the reasons outlined, another measure of Java's importance is the number of commercial entities that have

produced their own implementations of Java for research and/or commercial purposes. This list includes IBM, Microsoft, Apple, Intel, Compaq (Digital), Sun (of course!), and Hewlett-Packard. Why did these companies find Java implementation a worthy endeavor, when other VM designs have failed to take hold? Briefly, Java is interesting for the following technical reasons.

- "wire" or network format, which allows interoperability across distributed systems
- verifiability
- concurrency
- breadth of class libraries
- simpler language than C++

We will examine one of the open source VMs, as well as some of the commercial VMs. We will begin by describing the Java Virtual Machine in isolation of any particular implementation.

Java Virtual Machine

A Java virtual machine must perform the following tasks:

initializing starting the virtual machine and creating and initializing appropriate fundamental classes, such as `Object`, `Thread`, `ThreadGroup`, `String`, etc.

class loading conversion of a name into a class. The VM searches the class path and reads the named class from persistent storage into memory. Preliminary verification of the structure of the `.class` file is part of this task.

verifying assuring the class meets the constraints defined by the JVM specification. [31] The majority of the work of verification is assuring that the Java bytecodes for each method type-check, and that the operand stack does not over- or underflow.

native code calling providing access to resources of the host environment. The virtual machine mediates access to services such as I/O to the executing program.

executing the run-time support for language features. Such features include memory management, thread management, object initialization, error handling, etc.

accessing standard class libraries the `java.lang` and `java.io` libraries, etc.
The virtual machine mediates access to the standard set of class libraries.

Most importantly, a virtual machine must execute the user's program! The execution mechanism for Java code is what most distinguishes one implementation from another. A JVM with a Just-in-time (JIT) compiler translates the Java bytecode into machine code for the processor on which the VM is implemented. Kaffe is a minimal JIT implementation of a JVM. Minimal is used here to describe a JVM that converts Java bytecodes directly into machine code, without conversion into an intermediate form or any kind of optimization. As we shall see below, some virtual machines have quite elaborate optimizing-compiler-like mechanisms for performing this translation process. Kaffe does not.

Kaffe

The open source virtual machine which we use to implement our work is Kaffe, from Transvirtual.[41] This VM can be deployed either as an all-interpreted system or an all-JIT system. The JIT system does a weak local register allocation, and no other optimizations. This VM build-time decision is a striking example of the VM implementation choice between strict interpretation and machine code generation.

Other Features Kaffe supports almost any machine and operating system in its interpreter only mode. There are also a large number of different processors supported in the all-JIT mode. When possible, Kaffe maps Java threads to native threads. Due to its targeting to the embedded systems market, Kaffe has fairly weak support for multi-processor systems in terms of lightweight synchronization mechanisms. Kaffe uses a rather crude incremental stop-and-copy garbage collection system. Kaffe does not inline methods. More details of the code generation process are covered in Section 3.5.1.

HotSpot

Sun's "Java HotSpot^{TM} performance engine" utilizes dynamic compilation.[39] In dynamic compilation, increasing levels of optimization are applied to code that is determined to be performance-critical. This is determined by the run-time behavior of the program. The first phase of HotSpot's execution of a particular method is purely interpretive. As execution proceeds, the next step is to generate native machine code. With continued execution, HotSpot employs an optimizing compiler on code that has been determined to be worth the expenditure of effort.

One of the advantages of delaying machine code generation is that profile information is gathered about the running program. This information, such as caller-

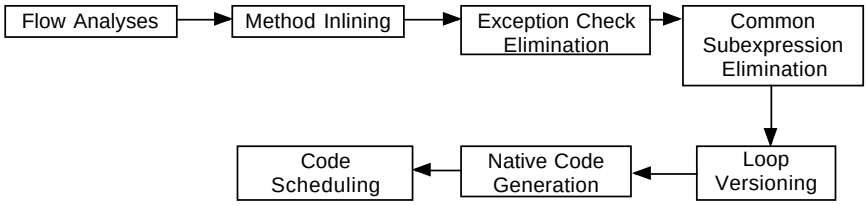

Figure 2.4: IBM Tokyo JVM Compilation System Flow

callee relationships for virtual method sends and branch direction frequency, can be used to guide the generation of machine code. In particular, method-inlining is guided by the caller-callee information. HotSpot makes much use of method-inlining. First, method-inlining reduces the cost of method invocations. Second, and most important, method inlining exposes the code of the inlined method to further optimizations. Because of the dynamic semantics of Java, wherein loading a class may necessitate undoing the previously generated code, HotSpot must perform dynamic deoptimization. This mainly consists of undoing any inlining, should subsequent performance measurements indicate the need.

The optimizations that HotSpot performs include dead-code elimination, loop invariant code hoisting, common-subexpression elimination and constant propagation. In addition to these language neutral optimizations, it also performs transformations oriented towards Java programs. These optimizations include null-check and range-check elimination. In addition, HotSpot uses a global graph-coloring register allocator.

IBM Tokyo

Another implementation of the Java Virtual Machine is one developed at IBM's Tokyo research lab.[37] Their implementation takes a tack typical of commercial JVMs—initial interpretation of bytecodes followed by an optimizing translation into machine code for those methods that are frequently executed.

In Figure 2.4, we see the compilation phases of the JVM. Below we will describe each of the phases in turn. However, as the code scheduling is conventional, we will not cover it. We take this detailed look to illustrate the optimizations and supporting analyses that were chosen by its implementors to be useful for a Java VM implementation. We use this system as an example, rather than HotSpot, since its optimizations are more fully described.

Flow Analyses The input to the Flow Analyses phase is the normal Java byte code. This phase does the typical control and data flow analyses, and additionally transforms the bytecode into an extended form. Instructions for dealing with the

interior addresses of objects and arrays are added to the normal Java bytecode. The extended bytecode is used as the input to the remaining phases, until the Code Scheduling phase.

Method Inlining Unlike Smalltalk, where any message send can be polymorphic, Java has specific bytecodes for doing non-polymorphic message sends. Sends to these methods are evaluated to be potential inlining sites by the IBM Tokyo JVM using the usual criterion used in more static languages. Such criteria include size of the method to be inlined, register pressure at the call site, whether the inlined method contains a loop or the call site is contained in a loop, etc. One area peculiar to Java is the higher frequency of empty methods. These are generated implicitly by the Java source compiler for zero-argument object constructors.

For inlining polymorphic message sends, the IBM Tokyo JVM takes advantage of the fact that a method may be marked "final", i.e. may not be overridden. Like the Deutsch-Schiffman inlining, one method is inlined with a check for its validity. However, the check is against the addresses of the current method and the non-inlined version of the method that is inlined. These addresses are easily obtained through the vtable. A similar optimization is used for message sends through an interface.

Exception Check Elimination There are two types of exception checks that potentially can be eliminated by this phase. The first is null pointer check elimination. For most processors, the majority of cases are handled by the memory management subsystem. However, for those bytecodes which may throw the exception, but which do not actually dereference the pointer, checks may have to be added. One such bytecode is athrow which should generate an exception if the reference to the exception object is null. To eliminate the number of these inserted checks, forward data-flow analysis is done to find all subsequent uses, and the checks are inserted only if the pointer is used. The second situation for exception check elimination is for array bounds checks. Here the set of potential array index values is calculated and propagated forwards and backwards along the data-flow path. Non-constant changes to the index variable invalidate the index-set. If the analysis shows an in-bounds set of indices, then the array-bounds check can be omitted.

Common Subexpression Elimination Common subexpression elimination, (or CSE) is only performed on code contained in loops. This is to focus the limited optimization time on the code which impacts performance the most. Additionally, simplified forms of CSE are done, also to save time. Scalar replacement is one of these. Here, a subscripted variable is replaced with a local variable reference. An-

other is common effective address generation. This CSE specialization transforms expressions referencing the middle of an array into a pointer to the interior of the array. The last CSE specialization is partial redundancy elimination, which is used to reduce the number of accesses to instance variables. Multiple accesses on the same execution path are turned into local variable accesses after the first access. All of these optimizations are tested for loop-invariance and hoisted out of the loop body if possible.

Loop Versioning The particular variety of loop versioning used by this JVM is one tailored to languages with exact exception semantics. A loop body may contain exception checks that could be removed, but only using information known at run-time. Such loops can be rewritten into two versions of the loop, one with the exception checks, and one without. Additionally, code is emitted to check the exception conditions. If the loop will not generate exceptions, then the quicker, no exception check version is jumped to. Otherwise, the slower version of the loop, with exception checks, is jumped to. This preserves the exact exception semantics.

Native Code Generation This phase is responsible for register allocation and generation of processor-specific machine code. Again, analyses are restricted due to the tight time constraints: "We consider expensive register allocation algorithms, such as graph-coloring, to be inappropriate, owing to the just-in-time nature of the JIT compiler."[37, p. 183] HotSpot amortizes the higher cost of a graph-coloring register allocator by deferring a global register allocator longer than the IBM Tokyo JVM does.

Accordingly, a less time-complex register allocation algorithm is used. The registers are divided into three sets—stack variables, permanent cached local variable, and temporary cached local variables. The physical registers are then dedicated to values in that order. In addition, registers in a particular set are allocated in a circular fashion, to reduce the potential for instruction scheduling conflicts.

In generating machine code from the extended bytecodes, a set of 80 different "idioms" are recognized, supplementing the one bytecode $\rightarrow$ one-or-more machine instruction translation scheme. These idioms help to mask the inefficiencies of the stack semantics, in particular the direct stack manipulation bytecodes.

One optimizing transformation not covered by the "idioms" is the generation of type-inclusion tests. Java has specific bytecodes for doing run-time type checking. Here, the machine code for the type-inclusion tests are handled in a fashion similar to inlined methods. If the object reference is null, then execution resumes at the next bytecode. Otherwise, if the type of the object is the same as the last successful type-check, then execution resumes at the next bytecode. If neither of the above

conditions is true, a call to a run-time library routine is called, and if successful, the object type is recorded, and execution resumes at the next bytecode. If the routine fails, an exception is thrown.

As stated earlier, the memory subsystem handles many of the possible exception checks. However, this is not the complete story. When an exception is thrown, the appropriate exception handler must be found, after which the stack is unwound, and the exception handler is executed. To reduce overhead, only those methods that contain an exception handler will have code generated for handling exceptions. This is in contrast to some implementations of exception handling where every method must have code generated for handling the unwinding of the stack during exception handling. To decrease the likelihood of instruction-cache misses, exception handlers are moved to the bottom of the method. Because this movement makes a simple mapping of locations to exception handlers difficult, a local variable is used to keep track of the current record for exception handling information. This record is threaded to similar records deeper in the call stack to facilitate finding the appropriate exception handler.

Summary The focus on performance in VM implementations has been illustrated by the foregoing discussion. Specific specialized techniques, such as loop versioning to maintain exception semantics, and partial inlining of type checks, are examples of how VM implementations achieve performance while maintaining the semantics required by the VM design. Further, we have seen careful tailoring of traditional compiler algorithms to meet the interactive response constraints.

Jalapeño

Another JVM implemented by an IBM research group is the Jalapeño system developed at IBM's T.J. Watson Laboratories.[2] It is designed for servers, in particular those with symmetric multi-processing (SMP.) A novel features of this JVM is that it is written in Java, except an extremely small kernel of functions written in C and assembly. This non-Java code includes a C-binding for accessing OS services, a boot loader, and two signal handlers for software traps and timer interrupts. Like the compile-only build of the Kaffe system, all Java bytecode is translated into native code, rather than being interpreted. The compilation system consists of three different compilers. (The technique of using a language for its own VM implementation, known as self-hosting, is a common technique. Many implementations of the Pascal P-system VM did the same.)

Compilers The Jalapeño system has three compilers that convert bytecodes to machine code. The *baseline compiler* is used for system bring-up and focuses

Table 2.2: Relative Performance of Jalapeño vs. IBM Development Kit VM

	Compilation Time	Execution Time
IBM-DK Interpreter		4-40
IBM-DK JIT	30-45	1
Jalapeño Baseline	1	2-20
Jalapeño Optimizing	30-45	1

on being transparently correct. The *quick compiler* is used for situations where machine code must be generated quickly. It uses no intermediate representation (IR) but does do copy propagation to remove the stack semantics of Java bytecode. Somewhat ironically, the quick compiler does have a graph-coloring register allocator, which may be used in addition to a simpler, faster register allocator. The *optimizing compiler* is used on those sections of code it is worthwhile to optimize. It uses multiple intermediate forms, with the "atoms" of each IR being an operator with an n-tuple of arguments. Each of these representations maintains type information. As the compiler is otherwise conventional, it will not be discussed further. Currently, no comprehensive strategy exists for choosing which compiler to use.

Performance Because important parts of Jalapeño were still under construction when the article was written, the following performance numbers are preliminary. The IBM Developer Kit (IBM-DK) compiler was produced by IBM Tokyo, as reviewed in section 2.3.3. In Table 2.2, we see the relative performance of compilation time and execution time. The compilation time and execution time are not comparable, as the times were reported only as ratios to the basline system, not in direct time measures. These numbers are for micro-benchmarks from Symantec. On the SPEC_{jvm98} benchmarks, similar results are found. It is important to note relative amounts of compilation time spent by the Jalapeño baseline and the optimizing version. Here we see the cost of performing optimizations as compared to a translation to machine-code which does no optimizations.

2.3.4 JIT Compilation Tradeoff

The preceding discussion highlights some of the trade-offs that occur in the mobile code environment, particularly when using JIT compilation. There is a limit to the amount of time that can be spent executing optimization algorithms while maintaining responsiveness, so less costly optimizations are used when available and expensive ones deferred until necessary. Note, however, that none of the systems discussed above saw memory footprint as a concern. That becomes an issue in environments that are tightly constrained in memory. Such JVMs do not have the

Table 2.3: Comparison of VM Implementations

Language	Implementation	Hardware	Interpretation	Compilation
Smalltalk	ST-76	✓	✓	
	ST-80		✓	✓
	SOAR	✓		✓
Self	Hölzle		✓	✓
Java	Kaffe		✓	✓
	Hot Spot		✓	✓
	IBM Tokyo		✓	✓
	Jalapeño			✓
	AJIT			✓
	Press Pot			✓
Prolog	(WAM)		✓	
	Aquarius	✓		✓
	Zephyr			✓
	Chare		✓	
IBM 1401	IBM 360	✓	✓	
IBM 370	VM/370	✓		
Tandem	MIPs		✓	✓
x86	Crusoe	✓		✓

memory to store the code for doing optimizations and most interpret the bytecodes.

We have also seen that the Java bytecode, while similar to the intermediate representation of optimizing compilers, is not sufficient to do many optimizations. As discussed above, particularly in the discussion of the IBM Tokyo JVM, the delivered JVM bytecodes are used to create graph-based IRs , and optimizations are performed on the IRs rather than the bytecodes.

2.4 Summary

Table 2.3 summarizes the VMs discussed in this chapter. The first column indicates which language is being implemented by the particular VM implementation indicated by the second column. The third column, labeled “Hardware,” is checked if the VM implementation uses hardware specifically designed for VM implementation. The fourth column, labeled “Interpretation,” is checked if the VM implementation uses interpretation to execute the user’s program. The fifth column, labeled “Compilation” is checked if the VM implementation uses compilation to machine code to execute the user’s program.

The systems which employ both interpretation and compilation usually employ dynamic compilation, i.e. interpreting first with potential compilation later. However, Kaffe makes this decision when it is built, so that a given implementation either always interprets or always compiles. We will discuss AJIT, a JVM similar to ours, in Section 3.7.2. Our system, PressPot, is discussed in Chapter 3. The Tandem to MIPs translation system employs interpretation only when maintaining the semantics of the input makes compilation too difficult. VM/370 is marked as employing hardware, as certain hardware "assists" are added to the machine to aid in hosting multiple virtual machines. Similarly, Crusoe is marked as employing hardware, as features were added to the processor specifically to support the semantics of compiling x86 code for the Crusoe VLIW.

In Section 2.2, we raised the point that there is an opportunity for pre-processing to be used to do machine-independent analysis on the server. When considering that in conjunction with Table 2.3, we see that *all* of the Java VM implementations use compilation as an execution strategy. This indicates that any work that could be performed by the server, transmitted in `.class` files, and used by a JVM when doing translation to machine code, would be useful. Such a system, to be most useful, must maintain the portability and safety of unmodified Java `.class` files.

In the next chapter, we will see how such a system can be designed and implemented.

3 Annotations

Annotations form the basis of our research. In this chapter we describe how annotations can be used to take advantage of the division of labor between the server and the client in a mobile code environment. We describe which compiler optimizations our annotations address and how we transmit optimization information from the server to the client. We then describe each of our annotations and the environment they operate in. The description of our system continues with how annotations guide code generation and how the annotations are generated. We conclude with a description of two closely related register allocators.

3.1 Introduction to Annotations

The division of labor in the mobile code environment between the server and the code generator can be exploited to improve the performance of code generated in the mobile client. As we saw in Chapter 2, most JVMs perform extensive analyses. If these analyses could be performed by the server and exploited inexpensively by the client, mobile code would execute more quickly and run-time costs for code generation would be reduced. The server is not time-constrained, so we can preprocess the mobile code and express the analysis information as annotations to the Java `.class` files. We focus on register allocation, as proper use of registers is critical to performance. This is because registers typically have an access cost that is at least an order of magnitude less than that of accessing memory. Our annotations use as a machine model a register-based machine with an infinite number of registers.

We add information about register allocation for this machine model to the Java `.class` file. The `.class` file has an annotation mechanism which is used to store most of the information about the class. For example, each Java method is represented in the `.class` file with a structure containing a reference to the method's name, a reference to the method's signature, and a variable number of named "attribute_info" structures. The bytecodes for the method are stored as an attribute_info structure named "Code." In our subsequent discussion, we will refer to these attribute_info structures as annotations.

In Figure 3.1 we see the environment of our annotated `.class` file. A client makes a request for some Java code, packaged either as a `.class` file or as a

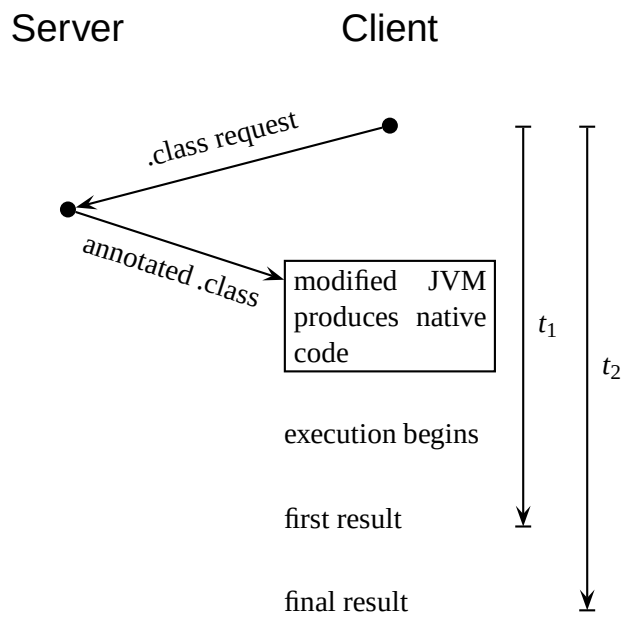

Figure 3.1: Java Virtual Machine Execution. t_1–time from client requests a Java `.class` file to time that the client first sees results; t_2–time from the `.class` file request to computation completion.

`.jar` file containing a `.class` file. The server sends the file to the client. This annotated code is used by our modified JVM to produce machine code, as in any Just-in-time (JIT) compilation system. We are interested in minimizing both t_1 and t_2. If we were interested in minimizing just t_1, then we could use an interpreter over the bytecodes and see our first results quickly. If we were interested in minimizing just t_2, then for a program which was compute intensive, we could tolerate compiler analysis being done by the JVM.

The client-side code generator could perform the same control and data flow analysis that the server-side annotator does. But our approach is to do as much work up front as possible. Many optimizations can be divided into a super-linear analysis phase and a linear exploitation phase. By performing the expensive analysis phase on the server and recording the results as annotations, we can then very quickly produce high-quality code in the code generator. In addition to the reduction of time needed to produce code, annotations allow the JVM implementation to be less complex.

3.2 Annotating JVM Code

Annotations are the addition of information to source code to guide or give hints to code generator. These annotations typically take the form of statements in the textual representation of the source code, supplied by a programmer, and are meant

to give guidance to the code generator in producing more efficient code. Our annotations are information added to the "source" of the code generator doing native code generation in a mobile code environment. The source here is Java bytecodes and our annotations are expressed as information added to the Java `.class` file. We implement our system, Press Pot, as a class file transformation. This class file transformation runs on the server in advance of any client requesting a `.class` file. We take a `.class` file and perform analyses on the code for each method, and write the results of these analyses as an annotated `.class`, where the results are represented as attribute_info structures for the method. Because the `.class` files already use annotations, extending the format of the `.class` files was not required.

The JVM specifies two relevant requirements for annotations. JVMs are required to ignore any annotations that that are not recognized. Annotations are not allowed to change the meaning of a program, as this would subvert the safety guarantees of the JVM.

These requirements constrain the implementation of our system. The requirement that annotations not change the meaning of a program restricts us from changing the bytecodes to a form that has an explicit register allocation. It also means that the annotated class files must work on JVMs that are not annotation-aware. To meet these requirements, we do not modify the Java bytecodes, but add information expressing our analysis information.

Java bytecodes are also verified to insure certain safety properties. The relevant safety property for Press Pot is that values must be type-safe. In other words, any use of a value must conform to the type it had at its definition. If a bytecode produces an integer value, then a bytecode that consumes it must be defined to take integer values as arguments. This restrains our system, as our annotations must maintain this type-safety property and be *verifiable* that they do.

Note that adding annotations does not preclude the VM from doing additional optimizing transformations on the bytecodes. It merely changes the intermediate representation from bytecodes for a stack-based virtual machine to bytecodes for an infinite register virtual machine.

The first part of our system, the annotator, takes as input `.class` files and produces as output annotated `.class` files (Figure 3.2).

These annotations are produced by using the Java bytecodes as an intermediate representation and applying traditional optimizing compiler algorithms to them. We then add the results of these algorithms as annotations to the output `.class` file. The size of these annotations must not be overly large, as this would increase the amount of time needed to transfer the annotated `.class` file across a network.

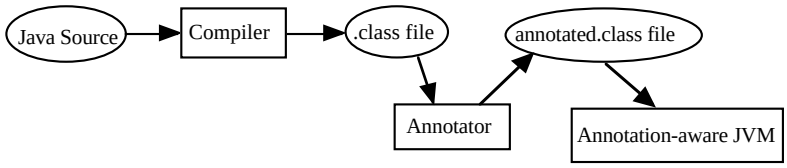

Figure 3.2: Annotation Process, showing flow of annotated code

3.3 Register Allocation Annotations

The system of annotations we have developed is used to specify how physical registers should be used by an annotation aware JVM. We have three annotations which contain information about register usage: virtual registers, copies and swaps. These annotations are associated with a particular Java method.

The base of our system of annotations is the virtual register (VR) annotation. The VR annotation is the means to obtain a register allocation (as opposed to a register assignment) without performing it on the client. A register allocation is the division of values into a set of equivalence classes. A register assignment is the assignment of a specific register to each equivalence class. For example, if we have five values, (A, B, C, D, E), then a register allocation may divide these into two equivalence classes, $\{$A, B$\}$ and $\{$C, D, E$\}$ without there being an assignment of these equivalence classes to a particular register of a machine. A register assignment for the above might be ($\{$A, B$\} \rightarrow$ r0) and ($\{$C, D, E$\} \rightarrow$ r1), where r0 and r1 are registers for some machine.

It is not possible to give a register *assignment* in a portable way. Each type of CPU has a different number of registers, and some registers may be limited in the values they can hold or in which instructions can use them. Traditional register allocation has access to this information, as the target machine is known. This is not true in the mobile code environment. Instead, we prioritize our register allocation, allowing the code-generator for the target machine to make register assignment decisions. The use of a verifiable register allocation permits the code-generator to quickly generate machine code which makes more frequent use of registers to hold values, as opposed to keeping values in memory. As access to registers is at least an order of magnitude faster than access to memory, efficient use of registers is critical to producing high-performance machine code.

We have an additional annotation, the copy, whose use is required in conjunction with the VR annotation. The copy annotation is used to allow an important optimization by the code-generator by compensating for certain aspects of the Java bytecode's stack-based nature.

Our final annotation is the swap, which is used to tailor the VR annotation for regions of a method. If different areas of a method have a significantly different

use of VRs, then the swap annotation is used to temporarily raise the priority of VRs which are heavily used in an area of the method. If the VR whose priority is being raised is not currently allocated to a physical machine register, then the code generator can "swap" a higher-priority but temporarily unused VR to memory and place the VR whose priority is being raised into a register. This compensates for situations where there are more values than registers by adjusting to the areas where values are used. Placing values into registers when they are active improves performance by avoiding accesses to memory.

3.3.1 Simple Example

Before going into detail about how annotations are generated and the exact nature of the mechanisms in the code generator to exploit them, let us examine a simple example illustrating the VR annotation. As our prototype implementation is for the SPARC, all examples are for the SPARC. In some examples, we will use a reduced-register SPARC machine to illustrate points about machines with a smaller register set.

The SPARC architecture divides the general-purpose register set into four parts, the global registers, labeled `%g0-%g7`, the input registers, labeled `%i0-%i7`, the output registers, labeled `%o0-%o7`, and the local registers, labeled `%l0-%l7`. In SPARC assembly language, the destination of an instruction is the rightmost argument.

The Java bytecodes operate on stack values, with many instructions taking as implicit arguments one or more elements from the top of an operand stack and placing the results onto the top of the operand stack. Other Java bytecodes load or store "slots" for holding values. These slots are part of the activation record for the method. These slots are referenced by number, with bytecodes set aside for accessing lower-numbered slots implicitly. Other bytecodes have operands that are part of the bytecode stream, just like operands for a register-based real machine.

In Figure 3.3, we have an example of the Java source for a "for" loop. This loop has three values, `sum`, `i`, and the constant `3`. Below it, we have the Java bytecodes corresponding to the source. The Java bytecode is formatted with the first column the bytecode location, the second column the Java bytecode and any arguments, and the third column the "VR" annotation. The third column is a register assignment for the bytecode to its left. We call this register assignment "virtual registers" or VRs. In the fourth column, we have the SPARC machine code generated from the bytecodes using the VR annotation. The last column has the SPARC machine code generated from the bytecodes using Kaffe's built-in local register allocator. Kaffe causes values active in a basic block to be spilled (stored) at the end of each

```
int sum = 0;
for (int i = 0; i < 3; i++) {
    sum += 1;
}
```

VM PC	ByteCode	VR	SPARC Instruction (w/VR)	SPARC Instruction (wo/VR)
0	iconst_0	1	mov %g0, %l1	clr %l0
1	istore_0	1		mov%l0, %l1
2	iconst_0	0	mov %g0, %l0	clr %l0
3	istore_1	0		mov %l0, %l2
4	goto 13	–	b	st %l1, [%fp - 96]
			nop	st %l2, [%fp - 92]
				b
				nop
7	iinc 0,1	1	add %l1,1,%l1	ld [%fp - 96], %l3
				add %l3,1,%l3
10	iinc 1,1	0	add %l0,1,%l0	ld [%fp - 92], %l4
				add %l4,1,%l4
				st %l3, [%fp - 96]
				st %l4, [%fp - 92]
13	iload_1	0		ld [%fp - 92], %l5
				mov %l5, %l6
14	iconst_3	2	mov 3, %l2	mov 3, %l7
15	if_icmplt 7	0,2	cmp %l0, %l2	cmp %l6,%l7
			bl	bl
			nop	nop

Figure 3.3: Simple Example, Shows relationship of Java source, Java bytecodes, VRs, and SPARC instructions for “for” loop with accumulator

basic block and loaded as needed. Basic blocks are separated by double horizontal lines.

The virtual register $\texttt{vr}_0$holds `sum`, the virtual register $\texttt{vr}_1$holds `3`, and the virtual register $\texttt{vr}_2$holds `i`. Also, the assignment of physical registers for virtual registers starts at `%l0`. [1] The first piece of machine code starts with a SPARC machine idiom of using global register `%g0`, which is always zero, to initialize a register with zero. At VM PC 1, we see an empty slot in the "SPARC instruction (w/VR)" column, indicating that no machine code was generated for the `istore_0` bytecode. The next interesting set of instructions occurs in the code for the `iinc` bytecodes. The `iinc` bytecode increments a local integer variable, indexed by its position in the stack frame, by a signed one byte quantity. The numbers after the `iinc` bytecodes indicate which local slot is being incremented, and what it is being incremented by, respectively. This is part of the normal encoding for the `iinc` bytecodes. Each `iinc` bytecode is translated into a single SPARC add instruction, since the values, referenced by both bytecodes are being stored in physical registers.

In this example, each VR corresponds directly to a physical register. The usefulness of the VR annotation is that it provides a register assignment at no cost. If there were always a sufficient number of physical registers, the register assignment problem would be completely conventional. In situations where the number of physical registers is not sufficient, our VR annotation keeps the most frequently used values in registers. We shall see later how our swap annotation further aids in keeping frequently used values in registers.

3.3.2 Virtual Registers

As noted above the VR annotation is the means to obtain a register allocation without performing it on the client. We give characteristics of the VR annotation itself, then briefly describe its benefit.

The VR annotation is a mapping from bytecode operands to virtual register numbers. The number of virtual registers per bytecode varies. For example, a binary arithmetic operator will have three virtual registers—two for the input and one for the output. A method call bytecode will have a variable number of virtual registers—one for the object, one for each argument to the method, and one for the return value.

Lower-numbered VR's have higher priority. VRs are assigned their priority based upon their importance and to minimize the number of distinct virtual registers. VR annotations are minimal assignments—we use no more VRs than the maximum number of simultaneous live ranges. A live range is a set of points in

[1] "ell zero", not "one zero."

the program where a value is "live," i.e. there is a use of the value along some path starting at the point. VRs are assigned based on the importance of a given VR staying in a physical register. Disjoint live ranges of the same type may be assigned to the same virtual register. This makes each VR *monotyped* (i.e. a VR can only "carry" one type of value throughout the entire method). This includes object reference types. For example, if vr_0 is used as a reference to an object of class A at one point in a program, it may not later be used as an integer or even as a reference to an object of class B, unless class A and class B have a type-compatible super class. Two object references are type-compatible if they have a common ancestor and accesses to them are appropriate to the ancestor. If one class is an ancestor of the other, then the ancestor is the common super class. The bytecode verification procedure calculates this nearest common ancestor, which is the least-upper bound along the inheritance and interface hierarchies. [31]

This annotation uses an unsigned one byte quantity for each virtual register. The values 0–254 indicate a valid virtual register number and the value 255 indicates no virtual register assignment. More information on the VR usage for every bytecode can be found in Appendix A.

3.3.3 Copies

The VR annotation by itself is not sufficient to ensure a proper register allocation. This can be characterized as an "impedance mismatch" between the stack orientation of the Java bytecode and the three-address code nature of the VR annotation. This mismatch occurs because local load/store bytecodes do not have machine code generated for them. We generate machine code on a per-bytecode basis and the local load and store bytecodes do not have any machine code generated for them. With the exception of the situation described below, this is possible, since any bytecode that defines a value will result in machine code that places that value in a physical location (a register or in the stack) and any use will reference that location. However, in some situations values are placed on the operand stack in a way that doesn't match the simple consumption model, resulting in incorrect programs if not compensated for. To correct the problem, we have the copy annotation. The copy annotation deals with the situation where the value of a slot is loaded on the Java stack and used in such a way that simply referencing the corresponding VR would be incorrect. One such case is when the loaded value is simply stored into another slot. The simplest example occurs with a Java source statement like:

```
h = k;
```

Which results in Java bytecode like:

```
iload_0
istore_1
```

This bytecode sequence does the following. The first bytecode, `iload_0`, pushes onto the stack the value in slot 0 of the method's activation record. The second bytecode `istore_1`, pops from the stack a value which is placed into slot 1 in the method's activation record. If no code is generated for these bytecodes, then the assignment effectively doesn't take place. We will see other situations where copies must be generated in Section 3.6.2. A copy is, quite simply, an annotation to indicate the program point where the code generator should generate a copy from one virtual register to another. The source VR is determined by examining the VR for the bytecode at the source PC. The source VR is the VR that is being copied from—the value that is being generated to compensate for the absence of the code for local loads and stores.

A copy annotation consists of a triple of <sourcePC, targetPC, targetVR>. The physical location of the source and target VRs of the copy can be in a register or in memory.

3.3.4 Swaps

When every variable is used with about the same frequency throughout a program, deciding which are most important to keep in physical registers (or equivalently in our case, to assign to lower numbered virtual registers) does not matter much. Similarly, if there are enough physical registers, then the priority does not matter. The reason is that no matter what the assignment, an equal number of operations will result in accesses to memory. The point is that the non-uniform use of a variable throughout a method opens up opportunities for that variable to be spilled and allow a more important value to have a chance at a physical register.

Using the swap annotation to guide changes in physical register assignment is crucial in achieving our goal of having machine-independent annotations. There is a great disparity in the number of registers available in popular microprocessors. Those in the RISC family, such as the SPARC, have a large number of registers, typically 32 or greater. In the CISC family, there is a great deal of variety. The Motorola 68000 has only 16 registers, and the most popular non-embedded processor, the Intel x86 family, has even fewer.[2] On machine witha sufficient number of physical registers, swaps will have no effect—i.e. generate no code at all, if the swap is between two VR's that are already assigned to physical registers. Non-uniform register usage is related to portability in the following way. A non-uniform register

[2] Stating an exact number is problematic, due to the non-orthogonal nature of register usage, i.e. some instructions can only be used with certain registers.

usage will result in lower performance if the number of physical registers is insufficient to hold all the values and if locally important values are not swapped into physical registers to replace less important values. In terms of register allocation, the distinguishing difference between different processors is the number of physical registers. The number of registers that a RISC processor possesses is typically sufficient to hold all values if a global register allocator is used. This is not true of the x86 or similar low register count machines. Therefore, to have portable register allocation, low register count machines must be accommodated.

Given the disparity in the number of registers between different processors, swap annotations provide a means for improving the quality of the register assignment. Higher quality is achieved by permitting the generation of improved spill code in a machine-independent fashion.

To do this, we mark regions of the program with "swaps" between two virtual registers. This marking indicates that until another swap is encountered, the roles of the two registers are exchanged. In a traditional global register allocator, if the number of registers is insufficient to contain all live ranges, then code is inserted to "spill" values from registers into memory and load values from memory into registers. The swap annotation has the effect that spill code generation has in a traditional allocator. As covered in more detail in Section 3.5.3, the presence of a swap annotation may or may not result in spill code being generated by the code generator. If there are sufficient physical registers, no spill code will be generated.

3.4 Overall Environment

To keep things concrete, we will continue using the SPARC as our example machine in the following sections, when we need a specific machine for expository purposes. With the exception of its register windows, the SPARC is like most modern RISC architectures. Furthermore, since all architectures perform better when registers are used as much as possible, most of the principles guiding the SPARC implementation hold everywhere. [3]

Figure 3.4 gives the overall structure of the generated code's environment. Values kept in registers are shown in the block labeled "Registers." This block represents the physical registers on the machine. In the figure, the machine has six physical registers, of which only two are available for storing virtual registers. The first two physical registers, r0 and r1, are reserved for storing the frame pointer (fp) and the stack pointer (sp), respectively. The next two physical registers, r2 and r3, are reserved for storing temporary values that are used during execution of the machine code. One use for these temporary registers is for storing values for

[3] Including the popular Intel x86 and the Motorola 68K.

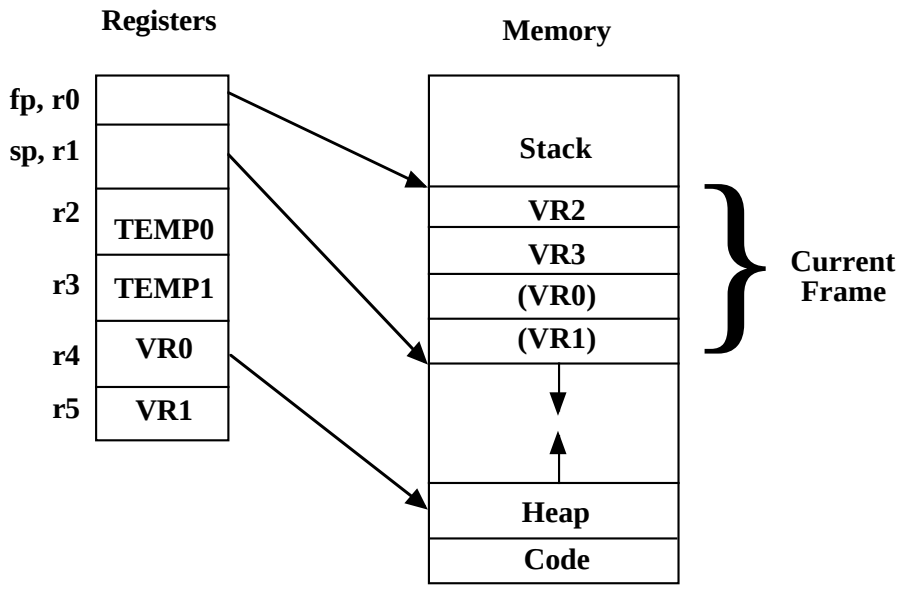

Figure 3.4: Run-time Environment. The stack location of vr_0 is reserved for spilling register vr_0; it may or may not contain the same value at other times.

virtual registers that are not allocated in physical registers. Values kept in memory are shown in the block labeled "Memory." This block represents the memory of the physical machine. Within that memory, an area is set aside for the runtime stack, which holds the frames of the currently active methods. The current stack frame is shown at the top of the stack. Within the current stack, the VRs which are not allocated to physical registers, $vr_2$and vr_3, are shown. At the opposite end of memory from the stack is the heap. Objects and arrays are kept in the heap and references to them are kept in registers or on the stack. An example of a VR for a reference to the heap is show in r4, where $vr_0$is either an array reference or an object reference. The following sections give details related to the environment sketched in Figure 3.4.

3.4.1 Simple Properties

Our central annotation specifies a register allocation. This annotation, the Virtual Register annotation (VR), decorates each Java bytecode with a set of virtual registers, which correspond to the operands of the bytecode. The code generator uses the VR annotation to generate a register assignment for the machine instructions. These VRs are arranged in *priority* order, meaning that the lower the VR is, the more likely it is to be assigned to a physical register in the code generator.

Below are some simple properties that guide our code generation process. We use the term "physical register(s)" to refer to the machine registers and "physical

location" to refer to some location on the machine, either in a physical register or in memory on the stack. We will also use the term "primarily stored" to indicate the physical location where a VR's value can be found most of the time. If this location is a register, there may still be times when the value is stored in the area set aside for it on the stack. This is done when making method calls for registers whose values are not preserved across method calls.

- All virtual registers have at least one physical location. Virtual registers which are allocated to physical registers have two physical locations, their physical register and their stack location. Other virtual registers will reside only in one physical location, on the stack. We will see the need for all values to have a place on the stack when we deal with swaps.

- Constants are either folded into the machine instructions or are statically allocated alongside the code for each method. The memory for the code and constants is allocated in the heap, but is not subject to garbage collection, so fixed addresses are used by our code generator.

- The bytecode operand stack is not mimicked. Rather than loading operands and storing results around every bytecode, physical registers are used as much as possible. If the VRs that are arguments to a bytecode aren't assigned to physical registers, then they are loaded from the stack into temporary registers. If the VR that is the target of a bytecode isn't assigned to a physical register, the result of the bytecode is computed into temporary register, then stored to the stack. For example, the integer add bytecode requires three operands; two for the input and one for the output. If one of the inputs has a VR which is not allocated to a physical register, then it is loaded into a temporary register. This temporary register is then used as an operand to the add machine instruction. A more detailed description of this process is given in the next section.

- A simple code-generation time data structure, the VR location table, provides the mapping from VRs to physical locations. Conceptually, the code generator keeps two sets of mappings, one for virtual registers residing only in memory and one for virtual registers additionally residing in a physical register. The VR location table is static and is built before code generation begins. [4] Imagine we have a machine where only two physical registers remain after setting aside registers for use as temporaries. Further, assume that the method we are generating code for has five VRs, numbered 0–4. An example of this data structure can be seen in Figure 3.5. This indicates that

[4] Ignoring for the moment our swap annotation.

virtual register	physical register	memory
0	%l0	[%fp - 12]
1	%l1	[%fp - 16]
2	–	[%fp - 0]
3	–	[%fp - 4]
4	–	[%fp - 8]

Figure 3.5: VR Location Table; MAXREGS = 2

virtual registers 0 and 1 will be primarily stored in physical registers `%l0` and `%l1` respectively and that virtual registers 2, 3, and 4 will be primarily stored on the stack at offsets 0, 4, and 8 from the frame pointer.

3.4.2 Non-register Resident Values

To manage situations where the number of physical registers is insufficient to contain all the virtual registers, the excess virtual registers are placed on the stack. There are several things to note.

- Several physical registers are reserved for temporary use. The lifetime of the values placed into these registers typically does not extend beyond the span of the machine code generated for one Java bytecode.
- For a given virtual register, its location on the stack is at a fixed offset from the frame pointer.
- On machines which allow the target of an instruction to be the same as one of its operands (e.g. `add %l0,%l1,%l0`), only two temporary registers are needed (actually, two temporary registers for integral values and another two for floating-point values.)

The heap is used for instance variables and class variables. The JVM bytecodes include instructions for accessing these kind of variables. For those bytecodes, there is an associated VR which holds the value that is either loaded from or stored to the variable. The VR represents this copy of the field value and there is no allocation of VRs to represent the field itself.

3.4.3 Constraints Imposed by Verification

Virtual registers are *monotyped*, meaning that the values held by a given virtual register are of the same type. This makes the calculation of the VR location table possible. If the type of the virtual register was not fixed, then determining the number and class of physical register to allocate to the VR would be difficult. Some types supported by the JVM require 64 bits to represent, which requires two

physical registers on some machines. Other types are most beneficially allocated to floating-point registers. The type of a VR is one of the Java primitive types, such as int or float, or an object reference type. Object reference types are distinct from each other and from the primitive types. For example a VR may not contain values that are of primitive type "`int`" and object type "`String`." As another example, a VR may not contain values that are of object type "`String`" and object type "`Exception`," unless the values are only used in a way that is compatible with their nearest common ancestor in the inheritance hierarchy, the object type "`Object`."

3.5 The Code Generation Process

In this section we discuss the code generation process, starting with the process for generating machine code from bytecodes that are annotated with VR information. Then we discuss the code generation for the copy annotation and then conclude with a discussion of the code generation for the swap annotation.

3.5.1 The Overall Code Generation Process and VRs

The general scheme for generating machine code is as follows, doing the steps below for each bytecode:

1. For all VRs used as input to the bytecode which are not assigned a physical register according to the current state of the VR location table, generate a load from their stack location into a temporary physical register.

2. If the VR(s) used for output from the bytecode is not assigned a physical register, set aside a temporary physical register.

3. Generate the appropriate machine instruction using temporary and/or permanent registers

4. If the output VR for the bytecode is not assigned a permanent physical register, generate a store from the temporary physical register to its location on the stack.

The VR location table is determined by the code generator using information gathered during the bytecode and VR annotation verification process. The verification step is used by the JVM to ensure that the bytecodes of a downloaded program do not corrupt the virtual machine. We have modified this step to additionally verify the VR annotation associated with each method. As a side effect of this process, the type of each virtual register is determined. This type is one of the Java primitive

types, such as integer or float, or an object reference type. For machines with a split general purpose register set and floating point register set, object references and integral types are assigned to the general purpose register set and floating point types are assigned to the floating point register set. There may be some VRs which will not be assigned to physical registers. Any virtual registers not assigned to physical registers are only assigned locations on the stack. This verification process is also responsible for gathering information needed for generating procedure entry prologues.

The calculation of the VR location table begins by finding a physical register of the appropriate type for $\mathtt{vr}_0$ and marking that physical register as allocated. The process continues with $\mathtt{vr}_1$ and proceeds until a physical location is determined for every VR.

The verification process allows us to make an important optimization. The Java Virtual Machine associates each method with a frame. These frames have two fixed-size components. The first is an operand stack for holding the operands to the bytecodes and for their results. The second is a set of indexed slots for holding local variables. The JVM has bytecodes that reference the frame slots by either loading from a slot onto the stack or popping from the stack and storing into a slot. Since the verification process ensures that every use of a JVM frame slot (or equivalently a local variable) is preceded along all paths by a definition of that slot, we can logically eliminate load/store bytecodes that reference locals.

As an example of the machine code generated using this scheme, consider an `iadd` bytecode and its corresponding annotation. The `iadd` bytecode is a zero-address instruction, which in the semantics of the stack-oriented JVM, removes the top two integer operands off the bytecode operand stack, adds them, and pushes the resulting integer result onto the stack. The `iadd` bytecode annotated as follows: "`iadd` 2,3→4". signifies that the `iadd` bytecode is annotated with three virtual registers, 2, 3, and 4. Our VR annotation has the semantics: take the values contained in virtual registers 2 and 3, add them, and place the result in virtual register 4. If we have a machine with a SPARC-like instruction set, but with only two allocatable physical registers for holding integers, then we generate the following, assuming the mapping from Figure 3.5:

```
! VRs > vr1 live in the activation record
     ld [%fp - 0], %g1  ! copy vr2 -> g1
     ld [%fp - 4], %g2  ! copy vr3 -> g2
     add %g1,%g2,%g1    ! compute result
     st %g1, [%fp - 8]  ! copy result to vr4
```

virtual register	physical register	memory
0	%l0	[%fp - 4]
1	%l1	[%fp - 8]
2	%l2	[%fp - 12]
3	%l3	[%fp - 16]
4	–	[%fp - 0]

Figure 3.6: Mapping from VRs to Physical Registers; MAXREGS = 4

(We are using `%g1` and `%g2` as temporary registers.) If we can allocate four physical registers to virtual registers, then the mapping would become that shown in Figure 3.6. We would generate:

```
! VRs > vr3 live in the activation record
     add %l2,%l3,%g1      ! compute result
     st %g1, [%fp + 36]   ! copy result to vr4
```

3.5.2 Code Generation for Copies

Generating code for a copy is straightforward. The copy annotation, is composed of a sequence of triples of the form: <sourcePC, targetPC, targetVR>, An individual triple is called a copy. The target of the copy is the targetVR of the copy. The source of the copy is the VR that is defined by the bytecode at sourcePC of the copy. The targetPC is the PC of the consumer of the copy. This targetPC is not currently used.

To generate the machine code for the copy the following procedure is used. The process for generating machine code is to iterate through the bytecodes and generate machine code for them, one bytecode at a time. At each iteration of this loop, the copy annotation is checked for a copy for the current bytecode PC. If one is found, then machine code for the copy will be generated. Using the VR location table, we generate the machine code to copy the value from the sourceVR to the targetVR. Either of these VRs may be in memory or in a register. Therefore either a register-to-register, memory-to-register, register-to-memory, or memory-to-memory move is generated.

3.5.3 Code Generation for Swaps

The swap annotation consists of information indicating the PC where the swap is located and which virtual registers are having their priorities swapped. What the code generator does with the swap annotation is dependent upon the machine architecture, particularly the number of registers available to be allocated. It is summarized in Figure 3.7.

A	B	action
in reg	in reg	none
in reg	in mem	store reg(A) and load mem(B), change loc(B) to be reg(A), loc(A) to be mem(A)
in mem	in reg	none
in mem	in mem	none

Figure 3.7: Swap of Virtual Registers A and B Versus Their Physical Registers, B is being "promoted"

```
int sum1 = 0, sum2 = 0;
for (int i = 0; i < 3; i++) {
    sum1 += 1;
}
for (int i = 1; i < 4; i++) {
    sum2 += 1;
}
return sum1 + sum2;
```

Figure 3.8: Java source used in illustrating swap annotation utility

In Figure 3.7, "in reg" means that the corresponding virtual register resides in a physical register, "in mem" means that the corresponding virtual register resides in memory, reg(A) means the physical register assigned to $\mathtt{vr}_\mathtt{A}$, mem(A) means the memory location assigned to $\mathtt{vr}_\mathtt{A}$, and loc(A) means the current physical location of a virtual register A. B is the VR whose priority is being raised. The second line of the table in Figure 3.7 specifies a change in these mappings, as well as a physical exchange of values between a register and memory. Since $\mathtt{vr}_\mathtt{B}$ is currently not in a register, but $\mathtt{vr}_\mathtt{A}$ is, $\mathtt{vr}_\mathtt{A}$ is stored to memory and $\mathtt{vr}_\mathtt{B}$ is loaded into the physical register previously used for $\mathtt{vr}_\mathtt{A}$.

One can see a simple example of the utility of swaps in Figures 3.8, 3.9, and 3.10. In Figure 3.8 is the body of a Java function. In Figure 3.9 is the corresponding annotated Java bytecode and SPARC instructions, with registers allocated as if the number of allocable registers were three. In Figure 3.10 we see the situation when swaps are used. In both Figure 3.9 and Figure 3.10, a starred ("*") location indicates the position of a swap. As a summary, here is the VR location table as it is at the beginning of the code genration process, along with the corresponding Java frame slot:

VM PC	ByteCode	VR	SPARC Instruction
0	iconst_0	0	mov %g0, %l0
1	istore_0	0	
2	iconst_0	3	mov %g0, %g2
			st %g2, [%fp+4]
3	istore_1	3	
4	iconst_0	1	mov %g0, %l1
5	istore_2	1	
6	goto 15		b xxx
			nop
9	iinc 0 1	0	addcc %l0,1,%l0
12	iinc 2 1	1	addcc %l1,1,%l1
15	iload_2	1	
16	iconst_3	2	mov 3, %l2
17	if_icmplt 9	1,2	cmp %l1, %l2
			bl xxx
			nop
20	iconst_1	1	mov 1, %l1
21	istore_3	1	
22*	goto 31		b xxx
			nop
25	iinc 1 1	3	ld [%fp+4], %g2
			addcc %g2,1,%g2
			st %g2, [%fp+4]
28	iinc 3 1	1	addcc %l1,1,%l1
31	iload_3	1	
32	iconst_4	2	mov 4, %l2
33	if_icmplt 25	1,2	cmp %l1, %l2
			bl xxx
			nop
36*	iload_0	0	
37	iload_1	3	
38	iadd	$0,3 \rightarrow 0$	ld [%fp+4],%g2
			add %l0, %g2, %l0
39	ireturn	0	ret
			restore%g0, %l0, %o0

Figure 3.9: Java bytecodes, VRs, and SPARC instructions showing utility of swaps (without swaps)

VM PC	ByteCode	VR	SPARC Instruction
			⋮

22*	`goto 31`		`st %l0, [%fp+8]`
			`ld [%fp+4], %l0`
			`b xxx`
			`nop`

			⋮
25	`iinc`	3	`addcc %l0, 1,%l0`
			⋮
36*	`iload_0`	0	`st %l0, [%fp+4]` `ld [%fp+8], %l0`
			⋮

Figure 3.10: Java bytecodes, VRs, and SPARC instructions showing utility of swaps (with swaps)

slot	VR	location
0	0	%l0 ([%fp+8])
3,2	1	%l1
–	2	%l2
1	3	[%fp+4]

The value corresponding to `sum1` is assigned to $\texttt{vr}_0$, and the value for `sum2` is assigned to $\texttt{vr}_3$. Although $\texttt{vr}_3$ is not used in the first for loop and $\texttt{vr}_0$ is not used in the second, we cannot assign either to the same virtual register since they are simultaneously live or interfere. If we used only our VR annotation, then $\texttt{vr}_3$ would not be assigned to a physical register. This is demonstrated by the load and store for the `iinc` bytecode at PC 25. However, we can improve the performance of this code by annotating PC 22 and PC 36 by saying that the relative priority of $\texttt{vr}_0$ and $\texttt{vr}_3$ are swapped when doing code generation. We annotate at these locations since they are natural loop boundaries. This is done by modifying the VR location table (discussed in Section 3.4.1). The change is seen in Figure 3.10. This has resulted in removing a memory load and store from the body of the second loop. By having swaps, we have managed to obtain almost the same performance with a three register machine as with a four-or-more register machine. This was accomplished at the cost of a load and a store, plus transmitting the swaps.

One aspect of the swap annotation not covered in the code generation description above is the placement of the machine code for doing the swap. In the following discussion, it is assumed that the VR location table indicates that code for performing a swap is to be generated. There are three different types of swaps,

SWAP_BEFORE, SWAP_AFTER, and SWAP_EARLY. The relative ordering of the code for the copy and the annotated bytecode as well as changes to the VR location table depend upon the type of the swap. If the swap has type SWAP_BEFORE, then the swap code is generated before the code for the bytecode at the swaps's PC and the VR location table changed. If the swap has type SWAP_AFTER, then the swap code is generated after the code for the bytecode at the swap's PC and the VR location table is changed. If the swap has type SWAP_EARLY, then the situation is slightly more complicated. If the annotated bytecode is not a conditional branch, then the code for the swap is generated after the code for the bytecode at the swap's PC, and the VR location table is not changed. If the annotated bytecode is a conditional branch, then the swap code should only be executed when the branch is taken. Therefore, we generate code for the branch, but with the sense inverted, and the target of the branch set to the code for the next bytecode, thereby continuing execution in the loop. The code for the swap is generated after this branch, followed by an unconditional branch to the target of the annotated conditional branch. The VR location table is not changed.

3.6 The Annotation Process

The generation of annotations is done off-line and therefore is not time constrained. We discuss the process of generating our three annotations: virtual registers, copies, and swaps.

3.6.1 Generating VR Annotations

The process of generating our VR annotation is similar to that of other register allocators. We use standard graph-coloring techniques, with an important distinction — we do not know the number of physical registers. Therefore, our algorithm for finding a register allocation consists of the standard algorithm modified to operate without this vital piece of information.

Instead, we find a coloring which includes all values and then prioritizes the resulting virtual registers. Additionally, to meet the mobile code need for verification, we force registers to be monotyped.

Prioritizing the virtual registers compensates for the lack of knowledge about the number of physical registers. In the normal Chaitin allocator (see Chapter 2.2.4), the physical register which holds a live range does not matter. In our prioritized VR scheme, on the other hand, we need to decide beforehand which registers should be spilled first, and it is simplest to use the register number to indicate its priority. Thus there is no difference between using register 1 and using register 10

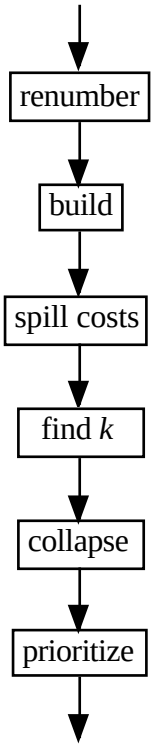

Figure 3.11: Presspot Register Allocator

if the machine has ten registers, but there is a difference if the machine has fewer registers and we are forced to spill one; register 10 has the lower priority, so it will be the one spilled. Therefore, we use the so-called "Haifa heuristics" developed at IBM's Haifa Israel laboratory by Bernstein et al. to order our colors so the colors that hold the most important live ranges are numbered lower.[8] These heuristics take into account the number of uses of the live range and how many other live ranges are simultaneously live with this live range. They were originally developed to determine which live range to spill if a k-coloring wasn't possible in a traditional compilation environment. Once this prioritization has been calculated, the VR annotation is added to the `.class` file.

Our algorithm is based upon the Chaitin work. In the following list, starred items are those phases of the algorithm which we have modified. Each of these phase descriptions will begin with the Chaitin version followed by our modification, where applicable.

Renumber Coalesce the def-use chains, creating the set of webs for which registers are allocated.

Build * Construct the interference graph. Edges are added between those webs which are simultaneously live. *To support verification in the client, addi-*

tional edges are added between those webs which are of incompatible types. For example, if we have webs 0 and 1 of type integer and webs 2 and 3 of type float, the following edges are added if they do not already exist due to simultaneous liveness: $\{(0,2),(0,3),(1,2),(1,3)\}$.

Spill Costs * Spill costs are calculated for each web based on an estimate of the cost of the load and store instructions that would be required to spill a particular web. The cost of each bytecode that uses a web is weighted by 10^d where d is the instruction's loop nesting depth. *We use an estimate of the cost of spilling an instruction that is machine independent, such that the cost of a load or store is some multiplier (> 1) of the cost of non-load/store instructions.*

Find *k* ***** Set *k* to the number of physical registers. *Our modification is to set k to the max degree of all nodes in the interference graph.*

Simplify * Remove all nodes with degree less than *k*. If at any point there is a node with degree greater than, or equal to, *k*, choose the node which has the lowest spill cost and remove it from the graph. *This test is unnecessary in our version of the algorithm. Our algorithm doesn't need to choose values to spill, as all values will be colored.* Nodes are colored by choosing the lowest color that does not interfere with any neighbors already colored. This helps to minimize the number of colors.

Collapse Merge webs that have identical colors. This minimizes the number of virtual registers, and is used to aggregate the definitions and uses associated with each VR when calculating priorities in the next stage.

Prioritize * Assign priorities to the webs based upon the Haifa heuristics. [8] *The prioritization in our algorithm works over the merged VRs, unlike the simplify stage which works on the spill costs of the webs. These heuristics consist of three functions that calculate the cost of keeping a particular value in a register. For each of the three functions, calculate the sum of the value of the function on each web. Choose the function which produced the lowest total cost and sort the webs based upon the cost for that function. Assign the priorities (VR assignment) for each web. Write out the annotation to the* `.class` *file.*

The prioritization phase deserves more explanation. The input is an assignment of non-interfering webs to VRs. The output of this phase is also such an assignment, but with the most important values (webs) placed in lower numbered VRs in a contiguous integer progression.

We use the Haifa heuristics, described in Section 2.2.4, modified to be machine-independent. We take as our spill list the entire set of "colors," (i.e. the register assignment for all the webs) and derive an ordering for this VR assignment using the following algorithm:

```
for every heuristic h_i
  for every web w
    costs[h_i][w.color] += h_i(w)
    perHcost[h_i] += h_i(w)
minH = index of MIN(perHcost)
sort web using cost[minH]
```

In other words, we calculate the priority for all VRs under each of the three heuristics. We then choose the heuristic which has the lowest total cost and use that heuristic in ordering the VRs (i.e. choosing their priorities).

3.6.2 Generating Copy Annotations

Copies are generated in three situations. We will deal with them in order of increasing complexity. The copy annotation is generated in exactly three circumstances.

Variable-to-variable Assignment

The simplest situation occurs when the bytecode contains the implementation of a Java source level variable to variable assignment statement. That is:

```
h = k;
```

would cause something similar to the following to be generated:

```
0 iload_0  5
1 istore_1 5
```

The first bytecode loads the integer value at slot 0 and pushes it onto the stack. The second bytecode pops the integer value from the top of the stack and stores it into slot 1. Both bytecodes are annotated with the VR 5. These bytecodes have the effect of copying the value in slot 0 to slot 1. But with no machine code being generated for such "local" loads and stores in our annotation-aware JVM, this copy will not take place unless some other annotation is used. Both instructions are annotated with VR 5, as the algorithm for generating VRs tracks values as the are pushed onto the stack and move to and from slots. To compensate, a explicit direction to the JVM in the form of a copy annotation. The copy <1,6,?> is generated to indicate that the bytecode at PC 1, (`iload_1`) is to have a copy generated

```
14 dup       9
15 istore_0  9
16 istore_1  9
```

Figure 3.12: Example of Duplicating Stack Manipulation

for it. [5] The result will be a copy from VR 5 to VR 6. In this situation, no new VRs are introduced, as the copy is generated with a VR (6) that is used elsewhere in the JVM code. The generation of a copy annotation is based upon a simple peephole analysis of the JVM code, searching for load/store pairs.

Stack Manipulation Used for Multiple Assignment Expressions

Using just the copy annotation for variable-to-variable assignment is not sufficient. Stack-manipulation bytecodes *may* generate new VRs, otherwise errors would result for situations like the bytecode in Figure 3.12. The `dup` bytecode pops the top value off the stack and pushes two copies of the value onto the stack. This has the effect of making two copies of the value on the top of the stack. Again, no machine code is generated for the `dup` bytecode or for either of the store bytecodes. Therefore, two copies must be inserted. In this case the two copies <15,10,?> and <16,11,?> would be generated. This would result in machine code for a copy from VR 9 to VR 10 at PC 15 and in machine code for a copy from VR 9 to VR 11 at PC 16.

Definitions Across Basic Block Boundaries

In Figure 3.13, the input to the bytecode at 23 comes from two possible definitions, one at 6 and one at 18. Simply annotating the bytecode at 23's input VR as the def VR of either 6 or 18 would be incorrect. [6] We introduce a copy annotation for every instruction that pushes a value that crosses a basic block boundary. Such values can occur as the result of compiling a Java source expression with the ternary operator, "?:". This copy will have as its target VR a VR which is a "temporary" that is shared between the other def instruction(s) and the use instruction in the succeeding basic block.

For basic blocks which have greater than two in-edges, we keep track of the VR generated on previous analysis and generate the corresponding copy.

[5] The third element of the triple, the targetPC is not currently used.

[6] A def VR is the VR for the value defined (the target) by a bytecode

```
public void example(boolean boolVar) {
    double d;
    int i = 1, j = 2;
    d = boolVar ? j++ : i++ ;
}
```

results in the following JVM code:

```
   0 iconst_1
   1 istore 4
   3 iconst_2
   4 istore 5
   6 iload_1
   7 ifeq 18

  10 iload 5
  12 iinc 5 1
  15 goto 23

  18 iload 4
  20 iinc 4 1

  23 i2d
  24 dstore_2
  25 return
```

Figure 3.13: Multiple Defs Across Basic Block Boundary

3.6.3 Generating Swaps

In devising an algorithm for generating swaps, the intuition provided by the aggressive live range splitting algorithm of Briggs was used. [9] There, static single assignment form is used to determine regions in the program where live ranges should be split. A live range is split by inserting code to store a value from a register into memory and to load value from memory into a register. This code is inserted around areas where the value isn't used. Similarly, loops are used in our system as the code regions around which to generate live range splitting.

The swap annotator is run after the virtual register allocator. All of the loops in the method are examined, looking for virtual registers that are of high-priority (low VR number), which are not used in the loop. If there is any VR which is not used in the loop (a "hole") and there is a lower-priority VR (higher VR number) that is used in the loop, then a swap annotation is generated to change the priority of the lower-priority VR to that of the higher-priority VR. Furthermore, the number of accesses for every VR in the loop is calculated, and those VRs with more accesses are assigned to those holes with higher priority. Swaps are limited to be between VRs of compatible types.

Once the VRs and regions are determined, the swap annotation is generated. There are three distinct types of swaps, BEFORE swaps, AFTER swaps, and EARLY_EXIT swaps. The differ in the placement of the movement code with respect to the machine code for the annotated bytecode. The first bytecode for the loop header is annotated with a BEFORE swap, which places the movement code (generated by the annotation-aware JVM) before any code in the the loop. The last bytecode for the loop is annotated with an AFTER swap, which places the movement code on the exit from the loop. Both the BEFORE and AFTER swaps result in changes to the VR mapping table. The EARLY_EXIT swap is placed on any branch bytecodes in the loop whose target is outside of the loop. The EARLY_EXIT swap does not result in changes to the VR mapping table.

3.7 Other Fast Allocators

In this section, we discuss two fast register allocators. The first is the linear scan register allocator by Poletto and Sarkar. The second is the AJIT system by Azevedo et al., which is an annotation-based JVM like ours.

3.7.1 Linear Scan Allocator

In Section 2.2.4, we discussed the linear scan register allocator by Poletto and Sarkar. It runs in linear time and achieves results comparable to that of a graph-

coloring allocator. This makes it a good choice for JVMs which are not annotation aware, because such JVMs are already performing substantial analysis, including the required liveness analysis. Thus, the quick code generation time and good performance of generated code make implementing the linear scan algorithm the right choice for such environments. However, our situation is different. We try to avoid doing "heavy-duty" analysis in our JVM, preferring to stick to the simplest possible exploitation of possible annotation information. Without calculating liveness information, we cannot make use of the linear scan register allocator in our back-end. For our front-end, where we can afford more analysis and total compilation time, the linear scan algorithm is still not the best choice. In their experiments, Poletto and Sarkar reported no benchmark where the performance of the code using the linear scan algorithm was better than that for the code generated by the graph coloring algorithm.[34] Therefore, the choice of using the graph coloring approach in our front-end is appropriate.

3.7.2 AJIT

Most closely related to our work is the AJIT system. [22, 5] In this system, annotations are added to the `.class` file, and then use an "annotation-aware" JVM to generate machine code. The more complicated Java bytecodes are broken into suboperations (e.g. `iaload`, load an element from an array of integers) and produce annotations specific to the suboperations. Like our system, AJIT uses the Kaffe JVM for their underlying virtual machine. Notably, they maintain (and extend with annotations) the Kaffe intermediate representation, whereas we convert bytecodes directly into machine code. They use the annotation feature of the Java VM design to open-up a communication path between the front-end and their VM implementation.

Annotation Semantics The AJIT system describes a virtual register assignment (VRA) which is similar to our VR annotation. However, due to the use of the Kaffe IR and a loosening of the safety requirements, their annotation takes on different semantics from ours. [7] We use an example from Azevedo to give a feel for the nature of VRA.[5]

In Figure 3.14 we see an example of an annotated java bytecode, `iaload` and the corresponding intermediate form. The `iaload` bytecode loads an element out of an array of integers. The `iaload` bytecode takes two arguments from the stack, the address of the array and the index to be accessed. In the AJIT system, there are two different cases for this particular bytecode. [8] In the first, the array element

[7] See Appendix A for detailed semantics of our VR annotation.

[8] N.B. The code for checking array bounds is not shown.

Case 1: Array element address calculation and array load

Bytecode	Java IR
iaload	V0 holds array address V1 holds index 1 : ishl V1, "ishift", V2 2 : iadd V2, "arraySizeOffset", V2 3 : aadd V0, V2, V3 4 : ild (V3), V4

Annotated Bytecode

opcode	SRC	SRC	EXTRA	EXTRA	DEST
iaload	V0	V1	V2	V3	V4

Case 2: Array load

Bytecode	Java IR
iaload	V3 holds array element address 1 : ild (V3), V4

Annotated Bytecode

opcode	SRC	DEST
iaload	V0	V1

Figure 3.14: Example of VRA annotations for iaload operation (from Azevedo).[5] Bytecodes are standard Java bytecodes and Java IRs are AJIT's custom intermediate representation.

address has not been previously calculated. This is shown in Figure 3.14 as “Case 1.” The corresponding Java IR multiplies the index (`V1`) by the size of an integer, by shifting (`ishl`), placing the result into `V2`. Then the total offset of the array element is calculated by adding (`iadd`) the offset of the desired element to the offset of the first element of the array (`arraySizeOffset`). Then the address of the array element is determined by doing an address add (`aadd`) of the address of the array (`V0`) to the total offset of the array element (`V2`). This address (`V3`) is used to do an indirect load (`ild`) into the result register (`V4`). In the second case, the array element address has been calculated, presumably by code similar to the the first case. This is shown in Figure 3.14 as “Case 2.” In this case, a simple indirect load is generated (`ild`) using the array element address (`V0`) and the virtual register of the destination (`V1`).

Like our PressPot system, the AJIT VM takes advantage of the fact that the Java VM design included a mechanism for communication from the front-end to the back-end. However, the nature of the communication, while appearing similar on the surface, is substantially different in philosophy. AJIT trades safety for performance. From a semantic viewpoint, the AJIT annotations introduce new primitives into the bytecode language of the JVM. For example, when annotating an array load bytecode, their system breaks the array load into its component machine-like operations, which will include a new “bytecode”, load from a precalculated address. This is used to remove the overhead of checked array accesses. By being able to express unchecked array accesses, the AJIT system can move bounds-check elimination analysis to the front-end. However, given the absence of analyses in the AJIT VM implementation to verify the safety of such accesses, the safety semantics of the Java VM design are violated.

4 Performance

In this chapter we compare the performance of our annotation-aware Java virtual machine against the underlying virtual machine implementation, Kaffe, as well as against a global register allocator. We first discuss the VR and copy annotations. These annotations were effective in improving the performance of programs in the SPEC benchmarks suite. We next discuss the swap annotation, which was effective for some benchmarks.

4.1 Virtual Registers and Copies

To measure the effectiveness of our VR and copy annotations, we modified the Kaffe JVM to accept our annotated `.class` files. Kaffe was modified as little as possible, to make the distinction between annotated and unannotated versions of the JVM as focused as possible. We begin by describing our experimental setup, present our results and then discuss our findings.

4.1.1 Methodology

In the following section we describe the specifics of our JVM implementation, architectural details of the SPARC, the execution environment, and the benchmark programs used to test our annotations.

Kaffe

Kaffe performs the Java Virtual Machine tasks described in Section 2.3.3, plus providing JIT translation into native machine code. For both the annotated and unannotated cases, machine code is generated in a macro-expansion fashion, with each Java bytecode considered in isolation from those around it and macros being responsible for generating the machine code. The code generation process begins by doing some set-up and performing verification.

The standard version of Kaffe works with unannotated bytecodes. When generating machine-code from unannotated bytecodes, the first stage is a `switch` statement over the bytecodes for the JVM. This `switch` statement is embedded in a loop over the bytecodes of the method being translated. The branches of the switch statement contain macros which generate an array of structures containing

function pointers. These macros implement the semantics of each Java bytecode in a machine independent fashion.

The second stage of the JIT process occurs as a `switch` statement over the operators of the structure array. The intermediate form contains pointers to functions which invoke macros that emit machine code to an in-memory array. After the code is generated, it is copied to its permanent location and labels are resolved. Any virtual method tables that refer to this method are now changed to point to the generated code, rather than to the function for starting code generation.

Our modification to Kaffe works with annotated bytecodes. The annotation-aware code generator follows the above process closely. The first stage is a loop over all the bytecodes of the annotated method. Within this loop is a `switch` statement over the Java bytecodes. However, in the annotated version, the individual branches of the `switch` contain macros for directly generating machine code, in our current implementation for the SPARC. Labels are constructed where needed, which are resolved as in the unannotated code-generator. The generated code follows the code generated by the non-annotation-aware version as much as possible.

The unannotated version of the Kaffe JIT does local register allocation rather than directly emulating the stack-based nature of the Java bytecodes. Such local register allocation results in a load instruction at the first occurrence of a use of a value in each basic block. Correspondingly, there is a store of every value defined in a basic block at the end of the block. In contrast, the annotation-aware code generator does not generate such loads and stores.

Relevant SPARC Features

The SPARC has several architectural features which are relevant when doing code-generation.

The most important feature is that the SPARC is a load-store architecture, with no machine instructions which operate on operands in memory, other than loading a value from memory into a register or storing a value from a register into memory.

The next most important architectural feature is register windows. The SPARC organizes its general purpose register set into a collection of register windows. General purpose registers hold integer values and addresses. There is a floating point register set, an indexed collection of thirty-two 32-bit registers. A register window in the SPARC contains twenty-four 32-bit registers. Each window is associated with an invocation of a method. The register window is divided into three different eight register sets. The input registers are used for passing the first six words of parameters to the current method. The last two words of the input reg-

isters are used for recording information about the stack-based activation record. The eight local registers are strictly for the use of the current method. The eight output registers are used for passing arguments to and from the current method to any method which it calls. Our annotation-aware code-generator properly accesses incoming arguments by generating references to the input registers.

Execution Environment

For our performance experiments, we used a Sun Ultra-1 workstation with 128 megabytes of RAM. We ran the benchmarks on a quiescent system.

The $\text{SPEC}_{\text{JVM98}}$ Benchmarks

To measure the performance of our system, the $\text{SPEC}_{\text{JVM98}}$ benchmarks from the Standard Performance Evaluation Corporation (SPEC) were used. [13] This corporation is a consortium of vendors and academics who develop standard programs, called benchmarks, for use in comparing the performance of different systems. The $\text{SPEC}_{\text{JVM98}}$ suite consists of eight programs, covering a range of different application areas. The benchmarks are actual applications, not synthetic benchmarks. The eight benchmarks are described below:

200_check not strictly a benchmark, this tests the overall correctness of the JVM implementation. It checks for such things as proper implementation of exceptions, array bounds checking, sub-classing and virtual methods, integer and floating-point math, etc.

201_compress a text compression and decompression algorithm based upon the Modified Lempel-Ziv method.

202_jess an expert system shell. This benchmarks reads in a problem definition in an expert system language and executes the program.

209_db a simple database program. Mainly tests input and output, but also performs in-memory sorting.

213_javac a version of a Java source to Java `.class` compiler. This is based upon an early version of Sun's JDK javac compiler.

222_mpegaudio an MPEG audio decompressor. MPEG is a standard method of compressing audio data. This benchmarks decompresses input files.

227_mtrt a ray tracer. This program reads in a three-dimensional scene description and uses a ray-tracing algorithm to calculate the rasterization of the resulting image.

228_jack a parser generator. This program reads in the description of a context free language and builds the corresponding tables for a parser for the language.

For each of the benchmarks, we produced annotated versions of all the `.class` files involved in the benchmarks. In addition to the `.class` files associated directly with the benchmark, we also annotated the SPEC testing framework and Kaffe's version of the standard class libraries. The standard class libraries are normally kept in `.jar` files, an archive format based upon ZIP for holding multiple logical files in a single physical file. The `.class` files for the system classes, such as `java.lang.String` were kept in `.jar` files, for both the annotated and unannotated versions. The `.class` files for the benchmark classes, such as `spec.benchmarks._200_check.Main` were kept in their `.class` file format as files in the file system, for both annotated and unannotated versions.. This configuration is how the SPEC benchmarks are configured to run.

We gathered one static measure: the count of the number of loads and stores generated by the unannotated benchmarks and the annotated benchmarks. Kaffe was modified to count the number of load and store machine instructions generated. These counts are used to measure the amount of load/store removal due to the annotations. This produces a slightly more dynamic measure of loads and stores than is seen in other research, as it counts only those loads and stores that occur in methods that are actually executed. The count does not indicate whether the individual loads and stores are actually executed.

Dynamic performance measures were also made. Each benchmark was executed six times: with and without annotations at problem sizes 1, 10, and 100. Each benchmark run was done in a separate instance of a JVM. The VM was instructed to exploit only the VR and copy annotations, although it still had to read and process the VR, copy, and swap annotations. The problem size specifies the amount of input to be processed or the number of iterations to be performed, depending upon the benchmark. Several different times were measured and are presented below. The most important is the total time, which includes all time from the beginning of the execution of the JVM to the very end, including any time spent doing I/O, including I/O done during VM initialization, during the benchmark's execution, and during the reporting of results by the benchmark framework. The next is the user time, defined as the amount of time spent executing the entire benchmark, from the very start of JVM to the end, excluding any time spent performing I/O. The last is the benchmark time, the time spent from the beginning of the execution of the benchmark to the end of the benchmark. The benchmark time does not include the time spent initializing the JVM or the amount of time spent reporting the results. It does however include the amount of time spent doing I/O during the execution of

Table 4.1: Load/Store Removal

Benchmark	Unannotated Load/Store Count	Annotated Load/Store Count	Ratio
200	91432	82206	0.90
201	88342	82537	0.93
202	104871	85857	0.82
209	89516	83236	0.93
222	151969	131200	0.86
227	99318	85778	0.86
228	116237	101751	0.88

the benchmark.

4.1.2 Static Measures: Results and Discussion

Annotations were appropriate for 11,463 of 13,303 methods (the remainder were abstract or native methods, or did not use virtual registers). Of these, 1,293 methods (11%) were not annotated due to implementation restrictions. From available measures, these unannotated methods did not represent a significant percentage of execution time.

In Table 4.1 we see the effect of our annotations on the number of load and store machine instructions generated. The fourth column, "Ratio," is the ratio of loads and stores in annotated code to loads and stores in unannotated code. A ratio lower than 1 indicates removal of loads and stores due to the use of annotations. Calculating percent reduction over all benchmarks,

$$1 - \frac{\text{total annotated count}}{\text{total unannotated count}}$$

gives an overall percent reduction of 12%.

As compared to other work in register allocation, where reductions in static counts of loads and stores of 59% are seen, our reduction of 12% is disappointing. [11] Their machine, the MIPS, had a larger number of available registers for allocation than our SPARC, which allows more values to stay in registers, thereby reducing the number of spills necessary. Also, the environment in that work was a traditional batch compiler, so safety and its attendant need for verification did not increase the number of conflicting live ranges, giving the register allocator more flexibility in assigning live ranges to registers. Additionally, their count is against what they labeled as "no register allocation," but it is unclear if that means that no values are kept in registers at all or if a local register allocation similar to Kaffe's is done. If no values are kept in registers, this would be similar to the example on

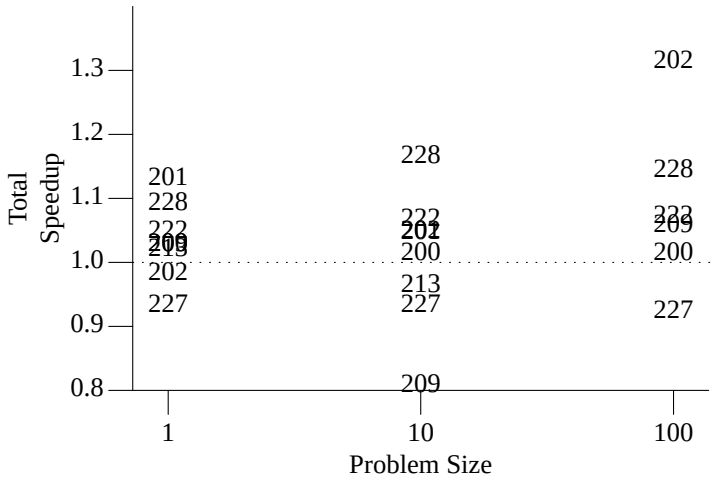

Figure 4.1: Total Speedup Versus Problem Size

page 45 where virtual registers 2, 3, and 4 are non-register resident. If their register allocator was being compared against a scheme where no values were kept in registers, the number of loads and stores eliminated would be higher than if it was compared against a local register allocation scheme.

4.1.3 Dynamic Performance

As described above, we used the $\text{SPEC}_{\text{JVM98}}$ benchmarks.[1] We will examine the speedups achieved in total run-time, total time spent not doing I/O, and executing just the benchmark. We will also examine the number of bytecodes and methods executed that were annotated and the amount of time spent by the annotated and unannotated JIT compiler.

First, we must give a definition for "speedup." Drawing from Hennessy and Patterson, we use the following: [20]

$$speedup = \frac{time\ spent\ executing\ by\ unannotated\ version}{time\ spent\ executing\ by\ annotated\ version}$$

This gives the best measure of the effect a user can expect when using our annotations.

Dynamic Performance Results

In Figure 4.1, we see the speedup our annotations produce on the total amount of time spent by each benchmark. On the x axis, we present the problem size on a

[1]The numbers we report do not strictly follow the SPEC guidelines for reporting benchmarks numbers. Specifically, our annotator modifies the `.class` files, which is prohibited under the guidelines. However, none of our annotations are specifically tailored to these benchmarks, so we do follow the spirit of the guidelines.

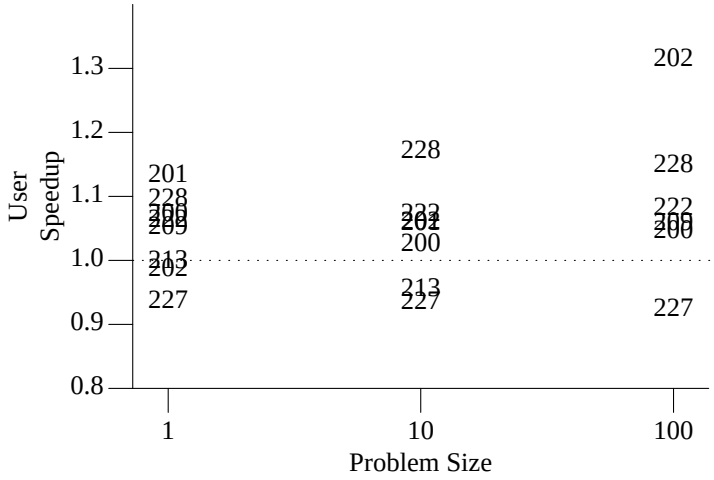

Figure 4.2: User Speedup Versus Problem Size

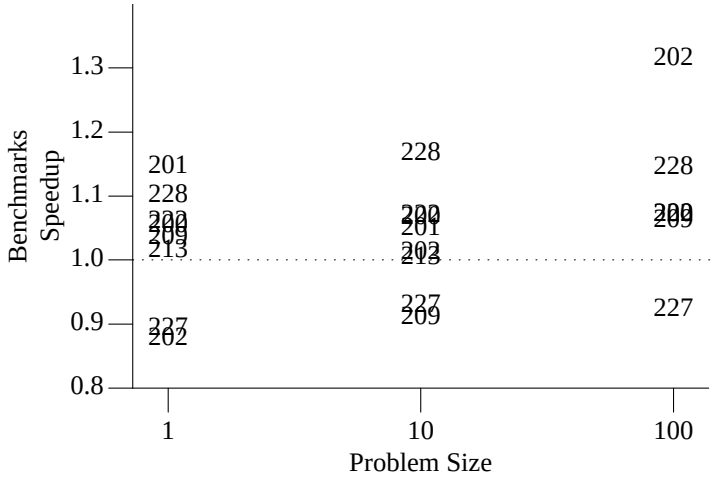

Figure 4.3: Benchmarks Speedup Versus Problem Size

logarithmic scale, for convenience. The y axis shows the speedup. Each three-digit number shows the speedup for the given problem size and benchmark. Note that for all problem sizes, we see speedups on most benchmarks, with one, 202, showing over 30%. For all problem sizes, we note that benchmark 227 experiences a slowdown. This will be explained below in our discussion of the benchmark and user speedups. Also note that benchmarks 201 and 213 are missing from problem size 100. They are not present, as both the unannotated and annotated versions experience "out of memory" errors when executed at problem size 100.

Figures 4.2 and 4.3 show the user time speedup and benchmark time speedup, respectively. These graphs use the same design as Figure 4.1. From these measures, we see a similar portrait of our annotations. We also measured the relationship between the ratio of methods and bytecodes annotated against the problem size.

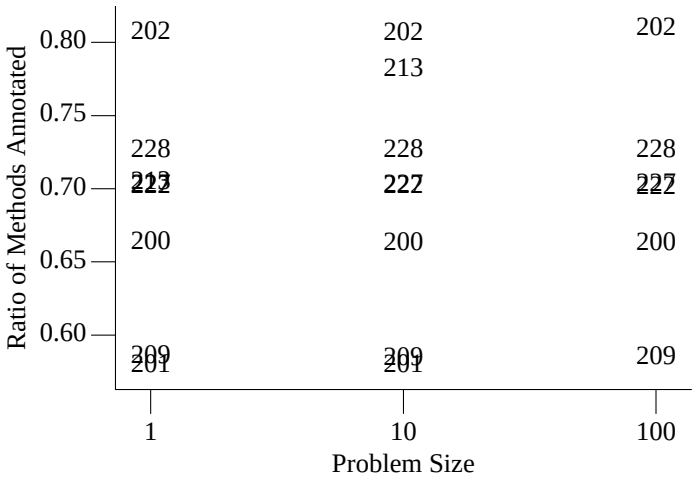

Figure 4.4: Ratio of Methods Annotated Versus Problem Size

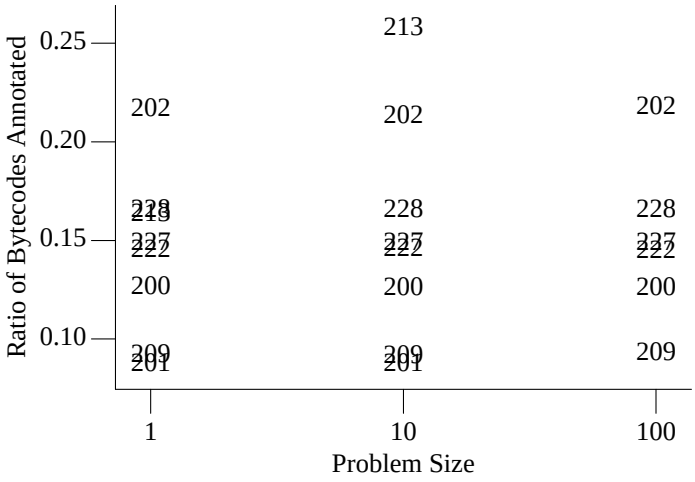

Figure 4.5: Ratio of Bytecodes Annotated Versus Problem Size

Table 4.2: Speedup Summary

Benchmark	Problem Size	Total Speedup	User Speedup	Benchmark Speedup
200	1	1.03	1.07	1.05
200	10	1.01	1.02	1.07
200	100	1.01	1.04	1.07
201	1	1.13	1.13	1.15
201	10	1.05	1.06	1.05
201	100			
202	1	0.98	0.99	0.88
202	10	1.05	1.06	1.01
202	100	1.31	1.31	1.31
209	1	1.03	1.05	1.03
209	10	0.81	0.80	0.91
209	100	1.06	1.06	1.06
213	1	1.02	1.00	1.01
213	10	0.96	0.95	1.00
213	100			
222	1	1.05	1.06	1.06
222	10	1.07	1.07	1.07
222	100	1.07	1.08	1.07
227	1	0.93	0.94	0.89
227	10	0.93	0.93	0.93
227	100	0.92	0.92	0.92
228	1	1.09	1.09	1.10
228	10	1.16	1.17	1.17
228	100	1.14	1.15	1.14

We modified the JVM to gather information about methods that were executed. For each run of Kaffe, if the "-jitSummary" flag was given, the number of different methods which executed and the total number of their bytecodes was recorded. Further, the number and size for annotated methods was also kept. The results can be seen in Figures 4.4 and 4.5. Table 4.2 gives the numbers for the speedups seen in the graphs in Figures 4.1, 4.4, and 4.5. The aggregate percent reduction in total time, over all benchmarks and problem sizes was 5.4% calculated using the geometric mean of all percent reductions.

Dynamic Results Discussion

Our 5.4% reduction in total execution time can be compared to work done in other global register allocators. Chow and Hennessy achieved a 28% execution time reduction. As discussed above, their system was a batch compilation environment where the target machine was known. Also, safety concerns did not limit

the choice of registers available to the register allocator. If we modified our verification algorithm to do a rudimentary form of liveness analysis, the performance might improve. By improving the verification algorithm so that VRs are not required to be monotyped to be verified, values of different types could reside in the same VR. This would lower the number of registers needed for a method, thereby improving performance. We would also to have to modify our annotator to insert edges into the interference graphs to limit VR sharing to those live ranges that have the same "machine type." A machine type would be one of the following types: one word (32 bit) integral, object reference, two word (64 bit) integral, one word (32 bit) floating, or two word (64 bit) floating. These collapsed types would allow the appropriate machine register to be used for a VR.

In Figures 4.1, 4.2, 4.3 we see significant slowdowns for benchmark 227. This raises the question, "What distinguishes this benchmark from the others?" In a work on removable array bounds checks for Java, Yessick and Jones discovered that this benchmark was one of two that had a significantly higher percentage of their loops which had removable bounds checks.[46] The other benchmark identified there was 222. A removable array bounds check in that paper was an array reference whose VRs could be easily determined to be a linear induction variable and would remain in bounds. We postulate from this result that the benchmark 227 has a higher density of array references and that this results in some kind of interaction with the data cache.

In an attempt to further explain these slowdowns, we examined the impact of not having 100% coverage for our annotations. Comparing Figures 4.4 and 4.5 to the speedup graphs in Figures 4.1, 4.2, 4.3, we can see that problem size has little effect on either the ratio of methods annotated or number of bytecodes annotated. Also, there is not a correlation between the slowdowns discussed above and the dynamic annotation ratios. The dynamic bytecodes annotated ratios (Figure 4.5) does indicate that the annotator's shortcomings seem to be have a higher impact on longer methods. This is not surprising, as longer methods are more likely to have problematic code.

Our VR and copy annotations are effective. In our earlier work, we showed the effectiveness of our annotations on small problems, such as quicksort and on the code from Chaitin's paper. [28, 10] Our work here shows that these annotations are effective for larger problems. Our current implementation however, does not capture as much code as possible. Removing the shortcomings in our annotator, as outlined above, should only improve the performance of our system.

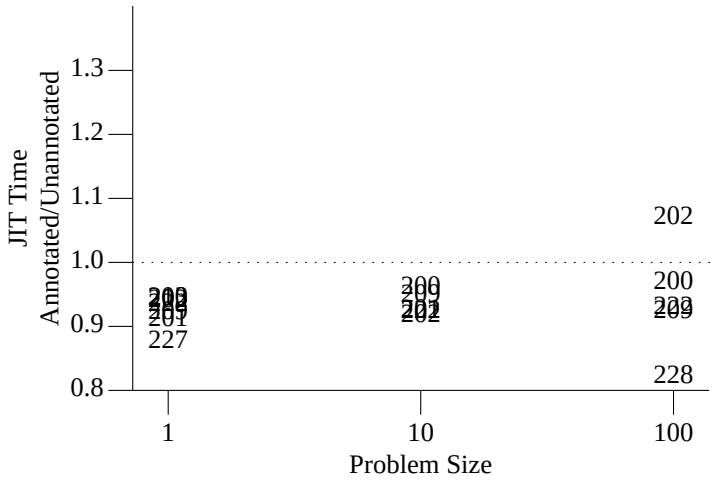

Figure 4.6: Ratio of Total Time in JIT: Annotated and Unannotated Versus Problem Size

4.1.4 JIT cost

We measured the relative cost of exploiting annotations by measuring the amount of time spent in the JIT by the annotation-aware and un-annotation-aware code-generator.

JIT cost results

In Figures 4.6 and 4.7 we have two different comparisons of the annotation-aware JIT and the simple JIT. In Figure 4.6 we compare the amount of time spent in the JIT when running the benchmarks when they were annotated versus when they were unannotated.

In Figure 4.7 we compare the amount of time spent in the annotated version of the JIT versus the amount of time spent in the JIT when the files are annotated.

JIT Time Discussion

In Figure 4.6 we see that the annotated version actually is faster than the unannotated version. In Figure 4.7, again we see that the annotated version is an improvement in the performance of the JIT. From this we conclude that the cost of exploiting virtual register annotations to generate a register allocation is lower on average than doing even local register allocation.

4.2 Swaps

The swap annotation, as described in Section 3.3.4, was intended to improve the performance of code on machines which have a small number of registers. In

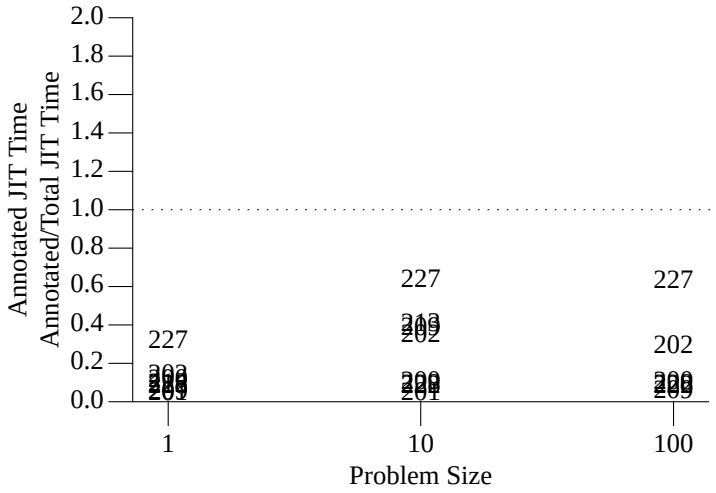

Figure 4.7: Ratio of Annotated Time in JIT and Total Time in JIT Versus Problem Size

particular, this was intended to be useful for the Intel x86 family of microprocessors. However, the swap annotation, as originally formulated, was not effective in achieving this goal. We will describe how we simulated the effect of the swap annotation on a small register machine using a large register machine implementation and show the performance results. We show the results of our first formulation of swaps and then an improved formulation where the type restrictions are loosened.

4.2.1 Swap Performance Methodology

To measure the effectiveness of the swap annotation, the same set of benchmarks as described above, the $\text{SPEC}_{\text{JVM98}}$ suite, was run. Note that the swap annotation depends upon the presence of the VR and copy annotations. Therefore, a comparison is made of whether or not the swap annotation provides any improvement over the VR and copy annotations, rather than comparing against the unannotated case. The SPARC is not a machine with a small number of registers. Therefore, the SPARC implementation of the annotation-aware JVM was modified to test the hypothesis that the swap annotation will be beneficial for small register number machines. A register limiting mechanism was implemented in the SPARC implementation to simulate a machine with fewer registers. This was rather straight-forward, as the JVM already had to deal with handling values which did not fit in a register and were instead stored on the stack. In Section 3.5.1, the process of loading a value temporarily into a register and also storing results from a register was described. The register limiting mechanism was placed in the code for calculating the VR location table. This code is called before generating machine instructions for each method. Machine registers were marked as already being allocated, thereby lim-

iting the number of registers available when developing the mapping from virtual registers to physical registers, thereby forcing more values to be stored on the stack. A command-line flag was added ("-intReg *n*") to Kaffe to set the number of physical registers available to the allocator.

Limiting registers in this fashion captures the essential difference under examination: do swaps improve performance on machines that have a small number of registers? Other aspects of a machine's implementation, such as the size, design, and number of caches, the speed of memory accesses, etc. can all vary within the same processor family. By holding these factors constant while varying the number of registers, the hypothesis can be tested directly.

Some modifications to this register limiting scheme were necessary to account for the argument passing conventions of the SPARC. As discussed above, the first six words of method arguments are passed in the input registers. Since Kaffe treats calls to Java and native code uniformly, the argument passing conventions were not changed, even when limiting the number of physical registers available to be allocated. Therefore, any virtual registers which were used for passing arguments in the first six input registers were retained in those physical registers. Suppose that the effect of swaps on a CPU that has only two registers available for allocation is being simulated. Any method that has more than two arguments will not be simulated faithfully, as *all* of the arguments that are passed in registers in the host environment, the SPARC, will remain in registers, thereby increasing the number of values in registers above the number that are being simulated. This has the effect of losing simulation fidelity for those methods which pass more arguments than there are simulated physical registers.

4.2.2 Swap Performance Results

In Tables 4.3 and 4.4, we see the total speedups of annotated code with swaps over annotated code without swaps, when using "strict swaps". A strict swap is a swap that is limited to be between VRs of the same type. For example, if VR 0 is a String object and VR 1 is an Exception, then no strict swap will be generated between VR 0 and VR 1.

In Tables 4.5 and 4.6, we see the total speedups of annotated code with swaps over annotated code without swaps, when using "permissive swaps". A permissive swap is a swap that is limited to be between VRs of the same "collapsed" type. A collapsed type is the result of mapping Java's primitive and reference types into five categories: one word integral types (byte, char, short, int), two word integral types (long), one word floating types (float), two word floating types (double), and reference types. For example, if VR 0 is a String object and VR 1 is an Excep-

Table 4.3: Strict Swap Speedup Summary, size = 1

Problem Size	Benchmark	Number of Registers	Total Speedup
1	200	10	0.97
1	200	8	0.98
1	200	6	0.97
1	200	4	0.97
1	200	2	0.97
1	200	0	1.00
1	201	10	0.93
1	201	8	0.99
1	201	6	0.96
1	201	4	1.01
1	201	2	1.00
1	201	0	0.99
1	202	10	0.98
1	202	8	1.01
1	202	6	0.98
1	202	4	1.01
1	202	2	0.99
1	202	0	1.00
1	209	10	0.98
1	209	8	1.01
1	209	6	0.99
1	209	4	1.00
1	209	2	0.98
1	209	0	1.00
1	222	10	0.99
1	222	8	1.00
1	222	6	1.01
1	222	4	1.00
1	222	2	0.97
1	222	0	1.00
1	227	10	1.00
1	227	8	1.02
1	227	6	1.00
1	227	4	1.00
1	227	2	1.07
1	227	0	1.00
1	228	10	1.00
1	228	8	1.03
1	228	6	1.04
1	228	4	0.97
1	228	2	1.03
1	228	0	0.99

Table 4.4: Strict Swap Speedup Summary, size = 10

Problem Size	Benchmark	Number of Registers	Total Speedup
10	200	10	0.98
10	200	8	0.97
10	200	6	0.99
10	200	4	0.98
10	200	2	0.99
10	200	0	1.00
10	201	10	1.00
10	201	8	0.99
10	201	6	1.01
10	201	4	1.01
10	201	2	0.96
10	201	0	1.01
10	202	10	1.00
10	202	8	0.99
10	202	6	1.02
10	202	4	1.01
10	202	2	0.98
10	202	0	0.96
10	209	10	0.52
10	209	8	1.00
10	209	6	1.17
10	209	4	0.99
10	209	2	1.17
10	209	0	1.01
10	222	10	1.03
10	222	8	0.99
10	222	6	1.01
10	222	4	1.02
10	222	2	1.01
10	222	0	1.00
10	227	10	1.11
10	227	8	1.31
10	227	6	0.97
10	227	4	0.58
10	227	2	0.91
10	227	0	0.88
10	228	10	1.00
10	228	8	1.01
10	228	6	1.04
10	228	4	0.98
10	228	2	1.02
10	228	0	0.97

Table 4.5: Permissive Swap Speedup Summary, speed = 1

Problem Size	Benchmark	Number of Registers	Total Speedup
1	200	10	1.00
1	200	8	1.00
1	200	6	1.00
1	200	4	1.00
1	200	2	0.99
1	200	0	0.99
1	201	10	1.04
1	201	8	1.02
1	201	6	1.01
1	201	4	1.01
1	201	2	1.01
1	201	0	0.93
1	202	10	0.99
1	202	8	0.99
1	202	6	0.99
1	202	4	0.98
1	202	2	1.01
1	202	0	0.97
1	209	10	0.99
1	209	8	1.00
1	209	6	1.03
1	209	4	0.99
1	209	2	0.99
1	209	0	0.98
1	222	10	0.98
1	222	8	1.00
1	222	6	1.00
1	222	4	1.01
1	222	2	1.00
1	222	0	0.98
1	227	10	1.10
1	227	8	1.10
1	227	6	1.10
1	227	4	1.00
1	227	2	1.02
1	227	0	0.93
1	228	10	0.99
1	228	8	1.00
1	228	6	0.99
1	228	4	1.01
1	228	2	0.99
1	228	0	1.02

Table 4.6: Permissive Swap Speedup Summary, size = 10

Problem Size	Benchmark	Number of Registers	Total Speedup
10	200	10	1.01
10	200	8	0.99
10	200	6	0.99
10	200	4	0.98
10	200	2	0.97
10	200	0	1.00
10	201	10	1.04
10	201	8	1.07
10	201	6	1.00
10	201	4	1.01
10	201	2	1.00
10	201	0	1.01
10	202	10	1.03
10	202	8	0.99
10	202	6	0.98
10	202	4	1.03
10	202	2	1.00
10	202	0	1.00
10	209	10	0.75
10	209	8	1.04
10	209	6	0.60
10	209	4	1.16
10	209	2	1.17
10	209	0	0.86
10	222	10	1.00
10	222	8	0.98
10	222	6	0.98
10	222	4	1.02
10	222	2	0.99
10	222	0	1.00
10	227	10	1.06
10	227	8	1.01
10	227	6	1.14
10	227	4	1.06
10	227	2	1.01
10	227	0	0.89
10	228	10	1.02
10	228	8	1.02
10	228	6	0.99
10	228	4	1.00
10	228	2	1.00
10	228	0	1.01

tion, then a permissive swap may be generated between VR 0 and VR 1, as both Exception and String collapse to the reference type. There is no loss of safety by using collapsed types, as each VR still has its own storage on the stack that is used when a swap is applied, so no coercion between types is necessary. The collapsed types are used to match the mapping of physical registers to VRs for modern machines. Most modern machines have separate integer and floating-point registers sets. Also, many modern processors do not yet have 64-bit registers, so 64-bit quantities must be split across two registers.

4.2.3 Swap Performance Discussion

It is clear that strict swaps do not provide a definite benefit, resulting in slowdowns of roughly the same order as speedups. However, for one of the benchmarks, 227_mtrt, swaps provide a benefit for both problem sizes when the register count is greater than zero. Strict swaps reduce the number of potential swaps by preventing swaps between values that can be stored in the same physical register without losing any of Java's safety guarantees.

A simple model for the behavior of swaps indicates that any swap that is applied cannot result in a loss of performance. Any VR which is the "raiseVR" of an applied swap will have at least one load associated with it if the swap were not applied. This load will be from memory into a register and will occur in the body of the loop. By adding one load and one store to memory on the boundaries of the loop and eliminating any associated loads and stores in the body of the loop, a simple model of program execution would indicate that a swap will always in a speedup.

This simple model does not account for the slowdowns. In the following section, the reason for this behavior will be postulated, based upon the construction of a code example for which swaps do provide benefit.

Permissive swaps do provide a definite benefit for the same benchmark, plus one other, 201_compress. Additionally, the number of cases where swaps result in slow downs is reduced. As will be discussed in Section 5.2.3, a profiler could be used to guide the insertion of permissive swaps. If profiling indicates that a program will benefit from swaps, then permissive swaps can be inserted. If profiling indicates that swaps will slow down the program or provide no benefit, swaps can be omitted.

As there is no loss in verifiability from using permissive swaps, they should always be used in preference to strict swaps when adding swaps annotations. From our experience with permissive swaps, we expect to derive benefit from deriving a new form of VR annotation which uses a more sophisticated verification algo-

```
public static int testWithArrays() {
     int sum1 = 0, sum2 = 0, sum3 = 0;
     int sum4 = 0, sum5 = 0, sum6 = 0;
     int[] a1 = u;
     int[] a2 = v;
     int[] a3 = w;
     int[] a4 = x;
     int[] a5 = y;
     int[] a6 = z;
     int i;
     for (i = 0; i < a1.length; i++) {
         sum1 += a1[i];
         sum2 += a2[i];
         sum3 += a3[i];
     }
     for (i = 0; i < a4.length; i++) {
         sum4 += a4[i];
         sum5 += a5[i];
         sum6 += a6[i];
     }
      return sum1 + sum2 + sum3 + sum4 + sum5 + sum6;
}
```

Figure 4.8: Example Where Swaps Are Not Beneficial

rithm which will not have a requirement for monotyping. This should improve performance by allowing more values to reside in registers.

An Example for Which Strict Swaps Do Benefit

Given the performance numbers above for strict swaps, it is natural to ask the question, "Is it possible to construct an example where strict swaps are beneficial?" The answer, of course, is yes. Further, a discussion of failed attempts at constructing an example are beneficial to understanding why the final example does show benefit, but more importantly, when swaps of either type can improve performance in general.

In Figure 3.8, code with two loops is given. These loops each contain a set of references to distinct variables, which are live across both loops. This code has too small a loop trip count for swaps to have a measurable impact. In scaling up the size of a problem involving loops, it is natural (perhaps) to introduce loops containing references to arrays. An attempt at producing an example of a beneficial use of swaps is seen in Figure 4.8. There are several things to note about this example.

Most importantly, the use of strict swaps on this example did not produce notable speedups.

There are other things to note. First, each of the for loops accesses eight VRs, three for the "sum" variables, three for the array addresses, one for the induction variable, and one shared by the array length and array values. Further, the "sum" variables are live across both loops, as they are set by their initializers (def), have uses and defs in the for loops, and have uses in the return statement. On a minor note, the variables "a1" through "a6" are used to avoid two shortcomings of the Java compiler and our register limiting mechanism. The array variables "u" through "z" are class variables initialized in code not shown here. The Java source to bytecode compiler does not allocate their addresses to slots, instead making all accesses to these array variables through use of bytecodes for accessing class variables. The class variables are moved into slots by using such statements as "`int[] a1 = u;`." Since both versions of Kaffe's JIT always produces machine code for loading class variables from memory, the memory traffic in the loops is removed by moving the array addresses into slots. Because values held in slots are eligible for allocation into physical registers, this eliminates one memory access per array reference, if there are enough physical registers. The passing of array variables as parameters to the function is avoided, as these arguments would not be limited by our register limiting mechanism.

Several attempts were made at manipulating the minor aspects of the program structure to produce a marked difference between the annotated version with swaps and the annotated version without swaps. Changing the size of the arrays did not make a difference. Increasing the number of array references in the loops did not make a difference. Modifying the simulated number of physical registers did not make a difference. From this, it was concluded that something other than access to registers or the stack was affecting the results. Since no I/O was being performed, this left traffic to main memory as the primary culprit.

To test this hypothesis, the motivating example of Figure 3.8 was returned to. There, all variable references inside the loops were to slots. The code in Figure 4.9 was written and measured. This code has no heap accesses in its inner loops. The effect of these changes is shown in Figure 4.10. When the number of registers is sufficient, i.e. when greater than eight, there are no speedups. With a lower number of registers, the swaps become quite effective until the number of physical registers is insufficient to hold enough active values.

```
public static int testNoArrays() {
     int sum1 = 0, sum2 = 0, sum3 = 0;
     int sum4 = 0, sum5 = 0, sum6 = 0;
     int i;
     int start = 1;
     for (i = 0; i < arraySize; i++) {
         sum1 = i + start;
         sum2 = i + start + 1;
         sum3 = i + start + 2;
     }
     for (i = 0; i < arraySize; i++) {
         sum4 = i + start + 3;
         sum5 = i + start + 4;
         sum6 = i + start + 5;
     }
      return sum1 + sum2 + sum3 + sum4 + sum5 + sum6;
}
```

Figure 4.9: Example Where Swaps Are Beneficial

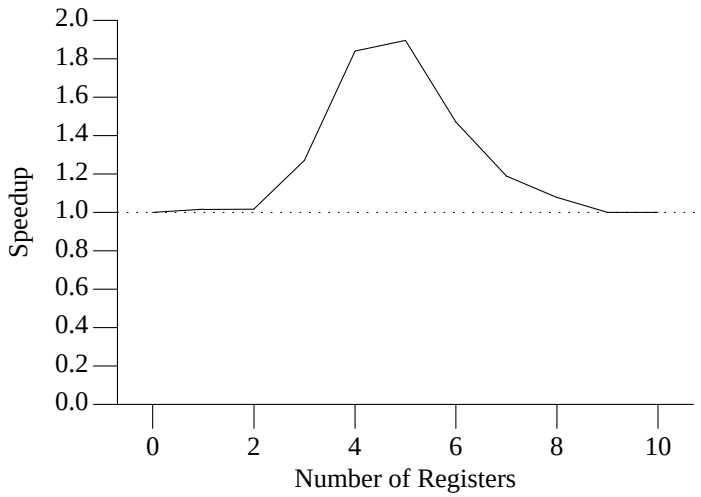

Figure 4.10: Contrived Swap Effectiveness

4.2.4 Summary of Swaps

Swaps improved the performance of code, but only when using permissive swaps and only for some benchmarks. It is possible that the swap annotation could become more generally effective in the presence of more aggressive optimizations such as redundant load store elimination. Use of a profiler to determine when to add permissive swaps would provide the balance between transmission time and execution time.

5 Future Work

We have shown that it is possible to compute low-level optimizations that are machine-independent, communicate them as annotations, and then exploit them. How does this effect the implementation of mobile code and the JVM in particular? The use of annotations broadens the design space of virtual machines, as high-performance no longer depends on implementing complicated optimizers in every VM. As one of the primary goals behind choosing virtual machines is to lower the entry barriers to new devices, annotations such as we have developed lower the performance cost that must be paid. The development of annotations covering more low-level optimizations will further improve the situation.

5.1 Immediate Questions

While we have shown the potential for a useful technique in the implementation of portable virtual machines, there are important immediate questions still open.

1. Are the virtual register assignments worth the transmission cost versus doing global register allocation in the JVM?

2. How much does monotyping a virtual register cost in terms of lost opportunities to allocate the register for typical Java functions?

3. How effective are these techniques on machines with fewer and more restricted registers than the SPARC?

We intend to aggressively address these issues in the near future as our implementation matures. It is our intention to answer these questions without compromising the mobile code constraints of portability and security.

5.2 Long Term Work

Dealing with the top of the memory hierarchy, registers, is certainly attacking the low-hanging fruit of low-level optimizations. Doing a good job of register allocation is critical for achieving high performance in compiled code. That out of the way, it is time to build a ladder and pluck other optimizations off the tree. Some of these optimizations deal with the other parts of the memory system—the caches

and the virtual memory system. Other optimizations deal with garbage collection, instruction fetching, thread scheduling, and unnecessary synchronization removal. The ladder that is needed to pick these optimization fruits is an effective run-time profiling system that transmits concise information from the client to the server.

We will begin the exposition of potential future work by describing simple optimizing transformations for addressing the above issues and how annotations can be used to communicate from the front-end to the back-end. We will then describe the requirements of our profile system and segue into details of potential future work.

5.2.1 Backend Transformations

When one examines the past two decades of compiler research, one finds a plethora of analysis techniques, optimization heuristics, and micro-architectural enhancements. A careful examination of the transformations that are performed upon the instructions and data of a program that are the result of the application of these sophisticated ideas results in a much simpler picture. Many transformations that occur can be described as moving, copying, or placing items (data or instructions) in memory so as to improve performance.

For example, trace (or superblock) scheduling is the placement of program instructions in such a fashion that the most commonly executed trace (path) through a program is efficiently fetched by the CPU's instruction decoding unit.[23] This is a placement and copying transformation on the original instructions of a program. Instructions from various basic blocks are placed together in a linear fashion. These basic blocks are also copied off-trace to handle the less frequent paths through the program.

Another example is proper alignment and chunk-sizing of arrays. [17] Scientific codes are full of tight loops over arrays. The manner in which an array is laid out in memory and the way the elements of an array are accessed strongly interact with the data cache of a processor. By properly aligning an array in memory and potentially extending the size of its rows, significant improvements in the cache miss rate of the data cache can be made.

The proper exploitation of both of these transformations depend upon both sophisticated compiler analyses and on specific machine-dependent information, much as our existing register allocation work does. Like our previous work, we propose the development of a set of annotations that assign the sophisticated analysis to the server and the exploitation of this analysis to the client. The data and instruction transformations can even be combined in an annotation describing how data and instructions should be placed for threaded programs on a multiprocessor

system.

5.2.2 Transformation Annotations

Let us examine in a little more detail how we might express transformations as annotations. Our current virtual register (VR) annotation is an array of unsigned bytes, where sequences of VRs in this array specify the register assignment for a bytecode in the "Code" attribute of a method. The assignment of VRs is dense, as most bytecodes are assigned one or more VRs, so no explicit tagging of VRs with program counters is used to associate VRs to bytecodes. Our annotation for manipulating VR priority, the swap, is different. A swap annotation consists of a sequence of structures, each element representing an individual swap. Each swap contains a program counter (PC) where the swap is to be applied, along with two bytes, one each for the VRs whose priorities are being swapped. This encoding is used because swaps are infrequent. Our proposed set of annotations will be closer in encoding to the swap annotation than to the VRs.

How would we encode a trace scheduling annotation? The JVM as part of its verification procedure finds the set of basic blocks of a method. We could encode a trace schedule for a method by listing the leader PCs of the basic blocks that form the trace. When the JVM sets about generating machine code, it would attempt to place the machine code associated with the trace in linear sequence and place the off-trace code some place out of this sequence. If the code on the trace has a control-flow structure that is too complicated for a particular JVM to handle, then it can ignore the annotation and simply proceed as normal.

The decoupling of annotation generation from annotation exploitation is important. First, by separating the analysis algorithms needed for annotation generation from their use, new optimization techniques can be used without necessarily changing the implementation in the JVM. Second, JVM implementors can choose which annotations are worth implementing for their particular application. Thus, a JVM implementor for a machine which has an extremely small instruction cache would properly decide not to implement a trace scheduler or would place a tight upper bound on the code size of any potential trace. This is a clear example of our separation of machine independent analysis from machine dependent exploitation.

5.2.3 Instruction and Data Profiler

The size of traces has always been a problem in doing profile-guided optimization. Separating the execution environment from the front-end consumer of the profile information with a network exacerbates the problem. One solution worth examining is to selectively send profile information only for those methods that are "hot,"

i.e. frequently executed. Another part of the solution to examine is compression of the profile results.

Once such a profiler has been built, then several possible optimizations become feasible.

5.2.4 Trace Scheduler

One possible optimization is trace scheduling. In the front-end, one would use the profile information of edge and/or path frequencies to reconstruct the most frequent traces. In the back-end, one would build a code placement and replication facility to build the traces and the corresponding off-trace "fix-up" code. One part of such work would be to obtain dynamic information about the instruction cache. An examination of instruction scheduling within the trace, using analysis in the back-end, would also be useful.

5.2.5 Redundant Load/Store and Synchronization Removal

In the Java Virtual Machine instruction set architecture, there are bytecodes devoted to performing loads and stores of heap-allocated data structures. In our current code-generation scheme, these loads and stores are mandatory, i.e. they always result in an actual load or store in the generated machine code. If Java programs were strictly sequential or communicated with other threads using some limited communication channel, then traditional load/store elimination could be done in the server and either transmitted to the client through a relatively simple annotation or by rewriting the bytecode before it is sent. However, Java uses a shared memory model of thread communication, which precludes the use of traditional load/store elimination optimization techniques. One solution worth investigating is whole program analysis with timestamping of the `.class` files. If the timestamps of all the `.class` files for a program do not match, then the load/store elimination annotation is ignored. Unnecessary synchronization removal is handled similarly. One problem with the approach is that the standard packages which the user program is analyzed against are not necessarily the same as the packages in the JVM where the program will be executed. One could devise an annotation that states the requirements that the user program makes on the standard packages. If the client JVM's packages do not meet these requirements, then the load/store elimination annotation is ignored. This should be fairly inexpensive to perform at run-time, as the requirements of the user program will be determined by the front-end when the user's program is annotated. The standard packages for the JVM in the client will be analyzed and their constraints recorded when the JVM is built. The situation here is precarious—many of the potential problems that an incorrect

analysis can induce can only occur (under the current understanding of the Java Memory Model) in systems with more than two processors.[38]

5.2.6 Thread Scheduling and Garbage Collector Hints

A technique for improving the performance of multi-threaded programs in a multiprocessor system is to do "gang-scheduling" of processes that communicate with each other. In gang-scheduling, threads that should be run together are assigned separate processors and scheduled to run concurrently. Also, heap data should be properly distributed to reside on the same processor as the thread that will use it the most. We suggest an annotation for the invocation of thread constructors that both identifies those threads that belong together and also separates them onto different processors. Threads should be brought into execution at the same time if they communicate with each other. Separate threads should be separated onto different processors if they can operate in parallel. If two or more threads operate as coroutines, then they should be placed onto the same processor if possible. It would also be useful to place annotations on heap memory allocations so that memory is clustered on the same processor with the thread the most frequently accesses it, particularly if the thread doing the allocation isn't the thread that uses the memory being allocated. The work would involve obtaining information dynamically about the number of processors.

Another potential optimization is to generate hints for the garbage collection (GC) system. One frequently used GC technique is generation scavenging. In this technique, objects are allocated in an initial area which is frequently scavenged. Objects that survive their first scavenging are "tenured" and moved into an area that is less frequently scanned. Information about allocations which result in long-lived objects can be communicated as annotations on the heap-allocation bytecode. Such an object can then be immediately tenured and thereby avoid being scanned and copied. The work here would involve changing the memory allocation system to use the provided hints from the annotation.

5.2.7 Array-based Data Cache Optimization

A final suggested optimization is to improve data-cache performance for array codes. There is a body of complex analysis techniques for determining the cache-miss behavior of programs loops over arrays. These analysis techniques combine information about the structure of the loop code with information about the cache size. The result of this analysis is to determine the precise alignment and padding for the array(s) that will result in the fewest number of cache misses. Alignment is the determination of the beginning address of the array, expressed as the offset

from some power-of-two boundary. Padding is the determination of how many additional elements should be added to the rows of an array. Again we see an instance of super-linear analysis and linear exploitation. One could annotate an array allocation with two equations, one expressing the alignment in terms of the cache size(s) (total size and cache line size), the other expressing the padding in similar terms. The back-end would merely plug the machine-dependent values (the cache sizes) into the results of machine-independent analyses (the equations) to obtain the appropriate alignment and padding. This work would involve modifying the run-time system to allow control over the alignment and row sizes of allocated arrays. Also, information must be obtained dynamically about the data cache.

5.3 Broader Questions

This work can also be expanded in a less linear fashion by examining different verification methods, different system architectures and different language models. Below are itemized some possible expansions, followed by a brief explanation of each of the extensions.

- Proof-carrying code for stronger verification
- Caching compile server
- Annotations for other VMs
- Annotations for other binary translators

5.3.1 Proof-carrying code

Our current system and the proposed extensions detailed in Section 5.2 rely on a fairly simple verification model. VRs are verified using an extension of the standard JVM verification procedure, insuring that every VR access is type-compatible with all other accesses. Our other implemented annotations also pass verification based upon the type compatibility of VRs. The annotations suggested immediately above are inherently safe, with the only potential effect of malicious annotation being performance degradation. Such simple verification techniques could be enhanced, thereby allowing the expression of optimizations through annotations that are not as easily verified. One such verification technique is proof-carrying code. [12] In this technique, proofs for doing verification are transported along with the corresponding code. The client verifies the code by checking that the proof and the code match and that the proof is correct. This would broaden the set of potential optimizations without losing verifiability.

5.3.2 Caching Compile Server

Running annotated code with an annotation-aware VM is great, but what about code that isn't annotated? If the code that is unannotated is downloaded frequently by clients with a shared infrastructure, then we can amortize the annotation cost across multiple clients. Then it would be useful to have a proxy server that handles requests for code and notices when there is duplicated code being downloaded. When such duplicate code is detected, then the code is placed on a queue of code to be compiled. In a separate thread, the code is dequeued, annotated, and cached. Subsequent requests for the code can then be answered with the annotated version.

5.3.3 Annotations for other VMs

Just as Java isn't the only language implemented using a virtual machine, it also isn't the only language whose performance could be improved with annotations. There has already been work annotating Prolog for memory optimizations. [19]

5.3.4 Annotations for Binary Translators

Occasionally, there are windows in processor evolution where, for the sake of compatibility, code is generated for one processor, even though newer, higher performance processors are used to execute the generated code. In such situations, it can be useful to annotate the code and have the annotation recognized in the execution environment. The simplest annotation would be to mark traces that would most benefit from doing processor specific instruction reordering. One example of this would be differences between versions of Intel Pentium processors. [26] The encoding of annotations could be handled in various ways—embedding the annotations in a extension section of the object file format, named tables in the initialized data part of the object file, or inside unreachable code.

6 Conclusions

Producing high-performance code has been an important part of high-level language implementation since the first FORTRAN compiler. Java is just the latest language where tradeoffs between programmer power and implementation ease had to be made. The implementation choices for Java are richer than for most modern languages. This is because Java was designed to be a mobile code system, in addition to performing as a more conventional language. Mobile code has many interesting language implementation problems. Mobile code is compiled and executed in an environment that differs from traditional batch compilation systems. We have shown that it is possible to do machine-dependent optimizations on mobile code without an optimizing compiler in the client. We review here the issues raised, how we addressed them, and how effective these efforts were in achieving our goal.

6.1 Requirements for Mobile Code

The distinguishing characteristic of mobile code is that code is designed to be stored on a server machine and delivered across a network to a computing device, potentially one that is portable. This characteristic adds additional requirements to the compiler and the run-time system beyond a desire for high-performance. Code in a mobile system must execute on devices that vary widely in their capabilities—CPU, memory, etc. Mobile code is separated from its execution environment by a wireless network, which may be several orders of magnitude slower than a typical local wired area network. Further, there is a lower level of trust between server and client. In a mobile computing environment, the separation between the server and the client is both technological and sociological. Technically, the server is separated from the client by various routers and other network connections, but most importantly by “air”, i.e. transmission across a medium where messages are easily intercepted. Sociologically, the server and the client are much more likely to belong to different organizations than in the batch compilation environment. These environmental demands create the requirements that mobile code be portable, safe, and transportable. These requirements have all affected this work.

6.2 Design for Mobile Code Annotations

Portability drove the requirement that our annotations be machine independent. For code to be portable, it must be expressed in a form that allows it to be executed in differing environments. Java's bytecodes can be interpreted, which is a well-known way of achieving portability. However, to obtain better performance, JVMs have implemented just-in-time compilers. Our annotations work with these VMs by providing machine independent optimization information. We achieved machine independence by designing our annotations for a virtual machine model that matches the needs of a just-in-time compiler, one consisting of registers and stacks.

Safety drove the requirement that our annotations be verifiable. For code to be safe, it must either have no ability to do anything unsafe or be checked and prevented from executing if it cannot be proven safe. Therefore, mobile code must be verifiable. Our annotations are verifiable and furthermore the verification is a natural byproduct of the verification done on the Java bytecodes, avoiding the potential cost of a separate verifier for the annotations. Our system has the further advantage that the verification can be done without adding any information to the `.class` file beyond that needed to communicate the optimization information. This is in contrast to other mobile code systems where the size of the information needed for verification is roughly of the same size as the code it verifies.[12] It is also in contrast to other work on annotations which were not verifiable.[5]

Transportability drove the requirement that our annotations be as small as possible. The choice of virtual registers tightly associated with bytecode is driven by this concern. As seen in our chapter on implementation, there are many analyses that are performed by an optimizer. There we saw control flow graphs, reachability, def-use chains, liveness, interference graphs, etc. Any of these could have been packaged as an annotation and beneficially used by a just-in-time compiler. However, the representation of these analysis would not be as compact as our virtual register annotation.

6.3 Mobile Code Annotation Effectiveness

The effectiveness of our approach must be analyzed in two ways. First, we must consider our annotation-aware approach against sophisticated JVMs with optimizing just-in-time compilers. Second, we must consider the effectiveness as compared to a minimal JVM which is not annotation-aware.

6.3.1 Annotation-aware versus Optimizing JITs

Comparison to more sophisticated JVMs is important, as this is the current approach taken by most commercial JVMs which implement a just-in-time compiler. As we saw in Section 4.1.4, use of our annotations adds little to the amount of time spent in the just-in-time compiler. In comparing our annotation-aware minimal JVM to more sophisticated JVMs, there are several considerations to make. We compare annotation-aware JVMs directly against non-annotation-aware JVMs, consider hybrid approaches, and examine how broadly applicable both techniques are.

First, what is the comparative performance between an annotation-aware JVM and a sophisticated JVM which does not make use of any annotations that it encounters? This largely depends upon the nature of the programs. If the programs only execute for a short amount of time, then the annotation-aware VM has an advantage, as the cost of utilizing the VR and copy annotations is only slightly more expensive than doing a minimal JIT compilation without annotations. For programs that execute for a longer period of time, the situation is murkier. Certainly, optimizing JITs can produce better code than a minimal JIT. However, the price must be paid for doing the required analyses. Theses analyses take super-linear time to perform and any time spent doing analysis is time spent not executing the user's program. Our annotation-aware gets the benefits of such analyses in only linear-time. Whether current optimizing JITs perform the proper amortization of analysis versus execution when compared to our annotation approach is difficult to decide without holding every other factor constant. Comparison would require that a similar low-level code-generator be used, that similar approaches to VM startup be taken, the same native methods be used for linking the system classes to the underlying operating system, etc. We see here that an annotation-aware JVM is the right approach for programs that run for a short amount of time and may be the right approach for others.

Second, what if the sophisticated JVM did use the annotations? A JVM that did so would have to integrate the added information provided by the annotation into its intermediate representation. All of the JVMs that we examined in Chapter 2 had a register-based intermediate form. Such an intermediate form seems ideal for integrating with our VR annotation. The copy annotation also should be easy to handle with a register-based intermediate form by generating a move instruction with the appropriate register arguments. Our swap annotation would possibly be a little more difficult, as the intermediate form early in the compilation process may not be configured to make use of detailed information about physical registers. However, as we saw in Section 2.2.4, determining where to place spill code is one

of the tasks in the Briggs-Chaitin register allocator. The information provided by the swap annotation might prove useful in this task. Here we see that annotations provide useful information even to sophisticated optimizing JITs.

Lastly, what if the client environment is memory-constrained? With Java running in devices ranging from cell phones to multi-processor mainframe computers, it is necessary to consider the implementation of a JVM in a memory-constrained environment. Every task performed by a JVM beyond the essentials adds to the amount of memory that the device must have to execute Java programs. Sun has defined several small virtual machines which only implement a subset of the Java VM bytecodes and of the Java system classes. These VM designs are driven by a desire to lower memory requirements as compared to the standard VM. Additionally, these VMs only perform interpretation, rather than using a JIT. From this, we see that there is demand for Java (or Java-like) VMs for memory constrained devices. Just as in most computing environments, higher-performance in memory-constrained environments is desirable if the price is right. Unfortunately, analyses done by sophisticated JVMs have a memory cost. There is memory needed to store the code implementing the analyses. This storage is significant, as most of the code in an optimizing compiler is dedicated to optimizations, rather than to the fundamentals of compilation. There is also memory needed to store the data for the analyses. This can also be significant, as many analyses require storage that is larger than the code being analyzed. Here we see that annotations are the proper response to the desire to obtain high-performance when sophisticated JVMs have too high a memory cost.

6.3.2 Annotation-aware versus Pure Minimal JITs

To test the effectiveness of our design, we implemented the virtual register, copy, and swap annotation in a Java virtual machine. This implementation was embedded inside the Kaffe Java virtual machine, a minimal, just-in-time compiler JVM. Kaffe was well-suited for our tests, as it allowed us to focus on the contribution that our annotations could make.

The swap annotation as presented here did not improve performance. While it did not ever significantly degrade performance, it also never produced significant speedups. Fortunately, the swap annotation is small enough that the added transmission costs would be negligible. We saw that it is possible to construct programs for which the swap annotation had tremendous effect—a 200% speedup. Therefore, more work should be done on an analysis to more accurately determine when swaps would be effective, along with changes to the verfication requirements.

The paired annotations which are the most widely applicable, the VR and copy

annotations, were benchmarked separately from the swap annotation. Our performance experiments proved that these two annotations are an effective way of improving the performance of a minimal JVM, Kaffe. There we saw total speedups for most benchmarks clustered in the range from 3% to 20%, with one case achieving over 30% speedup, and only one benchmark that saw a slowdown over all problem sizes. This proves that annotations are an effective way of improving the performance of mobile code.

A Semantics of “VR” Annotation

This chapter gives a detailed description of the “VR” annotation by describing the virtual register usage and code generation for various classes of bytecodes. This classification is based on the purpose of the bytecodes, not necessarily the similarity of virtual register usage. The virtual register annotation use an unsigned one byte quantity for each virtual register. The values 0–254 indicate a valid virtual register number, 255 indicates no virtual register assignment. The notion of virtual register here is similar to that found in the literature.

f

scalar loads {*ldc, ldc_w, ldc2_w, iload, lload, fload, dload, aload, iload_0, iload_1, iload_2, iload_3, lload_0, lload_1, lload_2, lload_3, fload_0, fload_1, fload_2, fload_3, dload_0, dload_1, dload_2, dload_3, aload_0, aload_1, aload_2,* and *aload_3*}

These bytecodes utilize one virtual register indicating where the result of the load is to be placed. The source of the load is either the constant pool (for the *ldc* and *ldc_w* instructions) or the stack. The code generator produces a load machine instruction into the physical register associated with the indicated virtual register. To see how the virtual register is translated into a physical register, see the section “Non-register Resident Values on a Three-Address Machine” above.

array loads {*iaload, laload, faload, daload, aaload, baload, caload,* and *saload*}

These bytecodes utilize three virtual registers, one indicating the address of the array, one indicating the index into the array, and the third indicating where the result of the load is to be placed. Unlike the previous class of load instructions, a load machine instruction is always generated, due to the perceived difficulty in allocating array variables to registers.

scalar stores {*istore, lstore, fstore, dstore, astore, istore_0, istore_1, istore_2, istore_3, lstore_0, lstore_1, lstore_2, lstore_3, fstore_0, fstore_1, fstore_2, fstore_3, dstore_0, dstore_1, dstore_2, dstore_3, astore_0, astore_1, astore_2,* and *astore_3*}

These bytecodes utilize one virtual register, indicating the source of the store. The code generator generates a machine language store.

array stores {*iastore, lastore, fastore, dastore, aastore, bastore, castore,* and *sastore*}

These bytecodes utilize three virtual registers, one for the array reference, one for the index, and one for the value to be stored. Unlike the previous class of store instructions, a store machine instruction is always generated, due to the perceived difficulty in allocating array variables to registers.

stack manipulations {*pop, pop2, dup, dup_x1, dup_x2, dup2, dup2_x1, dup2_x2,* and *swap*}

These bytecodes utilize no virtual registers.

binary ops {*iadd, ladd, fadd, dadd, isub, lsub, fsub, dsub, imul, lmul, fmul, dmul, idiv, ldiv, fdviv, ddiv, irem, lrem, frem, drem, ishl, lshl, ishr, lshr, iushr, lushr, iand, land, ior, lor, ixor, lxor, lcmp, fcmpl, fcmpg, dcmpl,* and *dcmpg*}

These bytecodes utilize three virtual registers, the first two for the arguments, and the third for the result.

unary ops {*ineg, lneg, fneg, dneg, i2l, i2f, i2d, l2i, l2f, l2d, f2i, f2l, f2d, d2i, d2l, d2f, i2b, i2c,* and *i2s*}

These bytecodes utilize two virtual registers, the first for the input, and the second for the result.

two argument conditional jumps {*if_acmpeq, if_acmpne, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt,* and *if_icmple*}

These bytecodes utilize two virtual registers, one for each argument.

one argument conditional jumps {*ifeq, ifne, iflt, ifge, ifgt, ifle, ifnonnull,* and *ifnull*}

These bytecodes utilize one virtual register, for the argument.

unconditional jumps {*goto, goto_w,* and *return*}

These bytecodes utilize no virtual registers.

`finally` handling jumps {*jsr, ret,* and *jsr_w*}

The *jsr* and *jsr_w* instructions are to defined to place the return address on the stack. One virtual register is used to contain this return address. The *ret* instruction has one virtual register which contains the address to return to, placed there by one of the *jsr* instructions.

value returns {*ireturn, lreturn, freturn, dreturn,* and *areturn*}

These bytecodes utilize one virtual register number, indicating what register contains the value to be returned.

function invocations {*invoke_virtual*, *invoke_special*, *invoke_static*, and *invoke_interface*}

For the *invoke_virtual*, *invoke_interface*, or *invoke_special* instruction, one virtual register for the address of the object whose method is being invoked, plus one virtual register for each argument to the method, plus one virtual register if the method has a return value. For the *invoke_static* instruction, one virtual register for every argument to the function, plus one virtual register if the function has a return value.

object creation {*new*, *newarray*, *anewarray*, and *multianewarray*}

For the *new* instruction, one virtual register number, indicating what register the address of the newly created object should be placed into. For the *newarray* and *anewarray* instructions, two virtual registers, one being the number of elements of the array to be created and the second indicating what register the address of the newly created array should be placed into. For the *multianewarray* instruction, the instruction will indicate the number of dimensions and the annotation will be that many virtual registers, indicating which register the size of each dimension of the array to be allocated will be located in, plus one more for the address of the newly created array. For all object creation bytecodes, the code generator creates a function call to the free store allocator and potentially a register-to-register move to transfer the address from the return register to the indicated register.

constants {*aconst_null*, *iconst_m1*, *iconst_0*, *iconst_1*, *iconst_2*, *iconst_3*, *iconst_4*, *iconst_5*, *lconst_0*, *lconst_1*, *fconst_0*, *fconst_1*, *fconst_2*, *dconst_0*, *dconst_1*, *bipush*, and *sipush*}

These bytecodes utilize one virtual register number, indicating which register the constant should be placed into. (NB: a peephole optimizer could possibly fold many of these into an addressing mode, particularly the integer constants.)

no operation {*nop*}

This bytecodes utilizes no virtual register.

exception throwing {*athrow*}

This bytecode utilizes one virtual register number, indicating what register contains the address of the `Throwable` object.

monitor traversal {*monitor_enter*, and *monitor_exit*}

These bytecodes utilize one virtual register, indicating the object whose monitor is to be used.

dynamic type checking {*checkcast*, and *instanceof*}

For the *checkcast* instruction, one virtual register, for the object whose type is being checked. For the *instanceof* instruction, two virtual registers, one for the object whose type is being checked and one for the result of the type check.

jump tables {*tableswitch*, and *lookupswitch*}

These bytecodes utilize one virtual register, for the value to switch on.

field manipulation {*getstatic*, *putstatic*, *getfield*, *putfield*, and *arraylength*}

For the *getstatic* instruction, one virtual register to contain the result. For the *putstatic* instruction, one virtual register to contain the value to put. For the *getfield* instruction, two virtual registers, one for the address of the object being referenced, and one for the result. For the *putfield* instruction, two virtual registers, one for the address of the object being referenced, and one for the value to put. For the *arraylength* instruction, two virtual registers, one for the address of the array being referenced and one for the length of the array.

increment {*iinc*}

This bytecodes utilizes one virtual register for the local variable being incremented.

References

[1] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. published by the author, 1999.

[2] 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, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[3] Kristy Andrews and Duane Sand. Migrating a CISC computer family onto RISC via object code translation. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 213–222, Boston, Massachusetts, October 12–15, 1992. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society. *Computer Architecture News*, 20, October 1992; *Operating Systems Review*, 26, October 1992; *SIGPLAN Notices*, 27(9), September 1992.

[4] Y. Asakawa, H. Komatsu, H. Etoh, Y. Hama, and K. Maruyama. Zephyr: Toward true compiler-based programming in Prolog. *IBM Journal of Research and Development*, 36(3):391–408, May 1992.

[5] Ana Azevedo, Alex Nicolau, and Joe Hummel. Java annotation-aware just-in-time (AJIT) compilation system. In *ACM 1999 Java Grande Conference*, 1999.

[6] C. G. Bell, A. Newell, M. Reich, and D. Sieworek. The ibm system/360, system/370, 3030, and 4300: A series of planned machines that span a wide performance range. In D. Siewiorek, C. Bell, and A. Newell, editors, *Computer Structures: Principles and Examples*, chapter 52. McGraw-Hill, 1982.

[7] Arndt B. Bergh, Keith Keilman, Daniel J. Magenheimer, and James A. Miller. HP 3000 emulation on HP precision architecture computers. *Hewlett-Packard Journal: technical information from the laboratories of Hewlett-Packard Company*, 38(11):87–89, December 1987.

[8] David Bernstein, Dina Q. Goldin, Martin C. Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter. Spill code minimization

techniques for optimizing compilers. *SIGPLAN Notices*, 24(7):258–263, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*.

[9] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, April 1992.

[10] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Journal of Computer Languages*, 6:45–57, 1981.

[11] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[12] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for java. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 95–107, 2000.

[13] Standard Performance Evaluation Corporation. Spec jvm98 benchmarks, 1998.

[14] Peter Deutsch and Alan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297–302. ACM, ACM, January 1984.

[15] Alan M. Durham and Ralph E. Johnson. A framework for run-time systems and its visual programming language. In *OOPSLA '96 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 406–420. ACM Press, 1996.

[16] Michael Franz and Thomas Kistler. Slim binaries. *Communications of the ACM*, 40(12):87–94, December 1997.

[17] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, July 1999.

[18] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[19] Gudjn Gudjnsson and William H. Winsborough. Compile-time memory reuse in logic programming languages through update in place. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):430–501, 1999.

[20] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[21] Urs Holzle. Adaptive optimization for SELF: Reconciling high performance with exploratory programming. Technical Report STAN//CS-TR-94-1520, Stanford University, Department of Computer Science, August 1994.

[22] Joe Hummel, Ana Azevedo, David Kolson, and Alex Nicolau. Annotating the java bytecodes in support of optimization. *Concurrency: Practice and Experience*, 9(11):1003–1016, November 1997.

[23] Wen-mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1–2):229–248, May 1993.

[24] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *OOPSLA '97 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 318–326. ACM Press, 1997.

[25] Daniel H. H. Ingalls. The smalltalk-76 programming system design and implementation. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona*, pages 9–16. ACM, January 1978.

[26] Intel. *Intel Architecture and Programming Manual, 245127-001*. Intel, 1999.

[27] Ralph E. Johnson, Carl McConnell, and J. Michael Lake. The RTL system: A framework for code optimization. Technical Report 1698, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1991.

[28] Joel Jones and Samuel Kamin. Annotating Java class files with virtual registers for performance. *Concurrency: Practice and Experience*, 12(6):389–406, 2000.

[29] Alexander Klaiber. The technology behind crusoe^{tm} processors: Low-power x86-compatible processorsimplemented with code morphingł software. http://www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf, January 2000.

[30] Xavier Leroy. The caml language. http://caml.inria.fr/.

[31] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1997.

[32] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., 1997.

[33] David A. Patterson. Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21, January 1985.

[34] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, September 1999.

[35] Balkrishna Ramkumar and Laxmikant V. Kale. Machine independent AND and OR parallel execution of logic programs: Part II — compiled execution. *IEEE Transactions on Parallel and Distributed Systems*, 5(2):181–192, February 1994.

[36] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Digital Technical Journal of Digital Equipment Corporation*, 4(4):137–152, Fall 1992.

[37] 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.

[38] Sun Microsystems Incorporated. JSR-133 Java ^{TM} memory model and thread specification revision. http://jcp.org/jsr/detail/133.jsp.

[39] Sun Microsystems Incorporated. The Java HotSpot^{TM} performance engine architecture: A white paper about Sun's second generation performance technology, April 1999.

[40] Randy Thelen. Under the hood: The Power Mac's Run-Time architecture: The RISC-based Power Mac uses a dramatically different application architecture that provides compatibility with past applications and future applications. *Byte Magazine*, 19(4):131–??, April 1994.

[41] Transvirtual Technologies. Kaffe OpenVM^{TM}.

[42] David Michael Ungar. The design and evaluation of A high performance smalltalk system. Technical Report CSD-86-287, University of California, Berkeley, February 1986.

[43] Peter Van Roy and Alvin M. Despain. High-performance logic programming with the Aquarius Prolog compiler. *Computer*, 25(1):54–68, January 1992.

[44] D. W. Wall and M. L. Powell. The mahler experience: Using an intermediate language as the machine description. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems-ASPLOSII*, pages 100–104, Palo Alto, CA, October 1987. Association for Computing Machinery, IEEE.

[45] David H. D. Warren. An Abstract PROLOG Instruction Set. Technical Report 309, Artificial Intelligence Center, Computer Science and Technology Division, SRI International, Menlo Park, CA, October 1983.

[46] Don Yessick and Joel Jones. Removal of bounds checks in an annotation aware jvm. In *IEEE SoutheastCon.*, 2002.