

OPTIMIZING SOFTWARE–HARDWARE INTERPLAY
IN EFFICIENT VIRTUAL MACHINES

by

Gregory B. Prokopski

School of Computer Science
McGill University, Montreal

February 2009

A THESIS SUBMITTED TO MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
DOCTOR OF PHILOSOPHY

Copyright © 2009 by Gregory B. Prokopski

Abstract

To achieve the best performance, most computer languages are compiled, either ahead of time and statically, or dynamically during runtime by means of a Just-in-Time (JIT) compiler. Optimizing compilers are complex, however, and for many languages such as Ruby, Python, PHP, etc., an interpreter-based Virtual Machine (VM) offers a more flexible and portable implementation method, and moreover represents an acceptable trade-off between runtime performance and development costs. VM performance is typically maximized by use of the basic *direct-threading* interpretation technique which, unfortunately, interacts poorly with modern branch predictors. More advanced techniques, like *code-copying* have been proposed [RS96,PR98,EG03c,Gag02] but have remained practically infeasible due to important safety concerns. On this basis we developed two cost-efficient, well-performing solutions.

First, we designed a C/C++ language extension that allows programmers to express the need for the special safety guarantees of code-copying. Our low-maintenance approach is designed as an extension to a highly-optimizing, industry-standard GNU C Compiler (GCC), and we apply it to Java, OCaml, and Ruby interpreters. We tested the performance of these VMs on several architectures, gathering extensive analysis data for both software and hardware performance. Significant improvement is possible, 2.81 times average speedup for OCaml and 1.44 for Java on Intel 32-bit, but varies by VM and platform. We provide detailed analysis and design guidelines for helping developers predict and evaluate the benefit provided by safe code-copying.

In our second approach we focused on alleviating the limited scope of optimizations in code-copying with an ahead-of-time-based (AOT) approach. A source-based

approach to grouping bytecode instructions together allows for more extensive cross-bytecode optimizations, and so we develop a caching compilation server that specializes interpreter source code to a given input application. Although this requires AOT profiling, it further improves performance over code-copying, 27% average for OCaml and 4-5% on selected Java benchmarks.

This thesis work provides designs for low maintenance, high efficiency VMs, and also demonstrates the large performance improvement potentially enabled by tailoring language implementation to modern hardware. Our designs are both based on understanding the impact of lower-level components on VM behavior. By optimizing the software-hardware interplay we thus show significant speed-up is possible with very minimal VM and compiler development costs.

Résumé

Pour avoir une meilleure performance, la plupart des langages de programmation sont compilés, soit avant leur exécution et statiquement, ou dynamiquement, pendant leur utilisation, l'aide d'un compilateur Just-in-Time (JIT). Cependant, les compilateurs avec des fonctionnalités d'optimisation sont complexes, et plusieurs langages, tel que Ruby, Python, PHP, profitent mieux d'une solution flexible et portable tel qu'une machine virtuelle (MV) interprète. Cette solution offre un change acceptable entre la performance d'exécution et les coûts de développement. La performance de la MV est typiquement maximisée par l'utilisation de la technique d'interprétation direct threading, qui, malheureusement, interagit mal avec les prédicteurs de branches moderne. Des techniques plus avancées, tel que code-copying ont été proposées [RS96, PR98, EG03c, Gag02], mais ne sont pas applicables en pratique cause de préoccupation de sécurité. C'est sur les bases suivantes que nous avons développé deux solutions cot-efficace qui offrent une bonne performance.

Premièrement, nous avons développé une extension au langage C/C++ qui permet aux programmeurs d'exprimer le besoin pour des garanties spéciales pour la technique de code-copying. Notre technique, qui requiert très peu de maintenance, est développée comme une extension à un compilateur qui a non seulement des fonctionnalités d'optimisation très laborieuses mais qui est aussi un standard d'industrie, le GNU C Compiler (GCC). Nous pouvons alors appliquer cette technique sur les interpréteurs Java, OCaml et Ruby. Nous avons évalué la performance de ces MV sur plusieurs architectures, en collectant de l'information pour analyser la performance logiciel et matériel. La marge d'amélioration possible est très grande, une accélération d'ordre 2.81 pour OCaml et 1.44 pour Java sur l'architecture Intel 32-bit. Il est important de noter que cette

marge est différente selon les MV et les architectures. Nous fournissons une analyse détaillée et des normes de développement pour aider les programmeurs à prédire et valuer les bénéfices possibles d'une utilisation sécuritaire de la technique de code-copying.

Notre deuxième solution vise à réduire les limitations sur la portée des optimisations de la technique de code-copying basée sur une méthode ahead-of-time (AOT). Une méthode basée sur le code source qui regroupe les instructions de bytecode permet des optimisations cross-bytecode plus laborieuses. Nous avons donc développé un serveur de compilation à mémoire cache qui se spécialise dans l'interprétation de code source donné comme entrée à une application. Quoique cette solution nécessite un profilage AOT, elle améliore les performances de la technique de code-copying par 27% pour le langage OCaml et par 4-5% sur certains tests de performance Java.

Cet ouvrage fournit des modèles pour des MV nécessitant peu de maintenance et hautement performant. De plus, il démontre le grand potentiel d'amélioration possible avec des techniques qui personnalisent l'implémentation selon le matériel. Tous nos modèles sont basés sur la compréhension des impacts des composants de bas-niveau sur le comportement de la MV. En optimisant les interactions entre le matériel et le logiciel, nous démontrons que des améliorations importantes sont possibles avec très peu de coût de développement pour le compilateur et la MV.

Acknowledgments

This journey and this thesis would not be possible without the help of many people. Foremost, I would like to thank my adviser, Professor Clark Verbrugge, for his guidance, patience, enthusiasm in all kinds of topics, the drive to focus our efforts on what is truly important, and for his financial support. I wish to thank Professor Laurie Hendren for her excellent teachings in the field of compilers, and that together with Professor Clark Verbrugge they had faith in me in difficult times. It is thanks to Professor Etienne M. Gagnon, that I had the opportunity to pursue studies at McGill University, and I am thankful for his support in personal and financial matters, and for seeding the initial idea of this thesis. I owe special thanks to my dear friend Ladan Mahabadi for her brutal honesty and ability to listen. I will always remember the help, warmth, and great laughs we shared with many members of the administrative and system staff of the School Of Computer Science. I would like to thank my dear Liza for her companionship, care, and invigorating energy. I am grateful to my parents, who always encouraged my passion for technology, and my brave sister, Krystyna, who unknowingly kept reminding me that a balanced, happy life can be created in many different ways, one for each of us.

This research was supported in part by NSERC and FQRNT.

Contents

Abstract	i
Résumé	iii
Acknowledgments	v
Contents	vi
List of Figures	ix
List of Tables	xiii
1 Introduction	1
1.1 Virtual Machines	2
1.2 Efficiency of Software-hardware Interplay	4
1.3 Contributions	8
1.4 Thesis Outline	9
2 Background	11
2.1 Hardware Architectures	11
2.2 Virtual Machines	14
2.3 3 Kinds of Interpreters	15
2.4 Code-copying Technique	17
2.4.1 Mechanism of Code-copying	18
2.4.2 Safety Concerns	19

2.5	Source code comparison	20
2.6	AOT-based Solution and Runtime Traces	21
2.7	Virtual Machines, Benchmarks, and Machines Used	22
3	Related Work	26
3.1	Interpreters	27
3.2	Compilation techniques	29
3.3	Virtual Machines	31
3.4	Compilation Servers and Resource Sharing	32
4	Static Compiler (GCC) Enhancement	34
4.1	Problem of Compiling for Code-copying	35
4.2	Architecture of a Compiler	37
4.3	Design	39
4.3.1	Generation of safely copyable code	40
4.3.2	GCC modifications	41
4.4	Experimental Results	53
4.5	Notes on Related Work	55
4.6	Conclusions	56
5	Application to Multiple Virtual Machines, Multiple Architectures	58
5.1	Application to Java, OCaml and Ruby	59
5.2	Execution Properties	60
5.2.1	Architectures and Virtual Machines	61
5.2.2	VM and Language Characteristics	61
5.2.3	Analysis	65
5.3	Code-Copying Results	68
5.3.1	Profiling	69
5.3.2	Dynamic behavior of copied code	73
5.3.3	General Findings	78
5.3.4	Overhead	92
5.3.5	Further Performance Factors	94

5.4	Notes on Related Work	97
5.5	Conclusions	98
6	Virtual Machine Specialization with Caching Compilation Server	101
6.1	System architecture	103
6.2	The source of performance improvement	105
6.3	Creation of the source code	109
6.3.1	Separate per-instruction source code	110
6.3.2	The profiling subsystem	110
6.3.3	Choice and creation of superinstructions	113
6.4	Specialized optimization	114
6.5	Compilation of the optimized VM	118
6.6	Performance results and metrics	119
6.6.1	Generated code comparison	123
6.6.2	Quick Comparisons of VMs Performance	126
6.6.3	Compilation Overhead and Break-even	127
6.7	Notes on related work	127
6.8	Conclusions and future directions	129
7	Conclusions and Future Work	131
7.1	Conclusions	131
7.2	Future Work	133

List of Figures

1.1	A Virtual Machine is the intermediate layer that can be compared to glue joining an application written in a programming language with the hardware it is executed on.	2
1.2	The taxonomy of virtual machines execution techniques.	3
1.3	Our goal is to achieve better runtime performance while keeping the costs of initial development and maintenance of a virtual machine low.	5
2.1	Virtual Machine is the intermediate layer that translates architecture-independent bytecode into operations directly executable on a target architecture.	12
2.2	Switch interpreter mechanism and its overhead sources.	15
2.3	A comparison of code execution in a traditional, direct-threaded (left) vs. a code-copying (right) code interpreter. Each non-dotted arrow is a jump; the total number is reduced by code-copying and the existing jumps are more predictable, mostly due to multiple copies of the same instruction.	18
2.4	Addressing the effective code of LCMP Java bytecode instruction in a) switch-threaded, b) direct-threaded, and c) code-copying interpreter.	21
4.1	Optimizing compiler can relocate less likely executed code to the outside of labels bracketing code used by code-copying.	35
4.2	Execution of a superinstruction containing a code chunk with a missing part or a call using relative addressing might cause VM crash.	36

4.3	A largely simplified view of a C/C++ compiler based on GCC internal structure.	38
4.4	To produce copyable code with minimal changes to the internal structure of the compiler we inserted several well isolated passes.	42
4.5	Pragma directives are placed around the code that will be used by code-copying engine at runtime.	43
4.6	At an early compilation stage volatile statements are automatically inserted by the compiler around the <i>end label</i> to ensure that the <i>target</i> basic block will remain intact throughout optimizations.	45
4.7	To ensure absolute addressing a <i>goto</i> to outside of a copyable area is replaced with a specially crafted <i>computed goto</i>	46
4.8	Initial marking of basic blocks right after parsing.	48
4.9	From the marking of only two basic blocks, <i>start</i> and <i>target</i> , the complete marking can be restored by following the edges of the control flow graph. Once the marking is restored it is possible to rearrange the basic blocks of a marked copyable area.	50
4.10	Performance comparison of SableVM with standard direct-threaded engine, unsafe code-copying engine and safe code-copying engine using the GCC copyable-code enhancement.	53
4.11	Metrics of code modified and added to GCC.	54
5.1	Percentage of time spent in the interpreter loop of the direct-threaded interpreters for x86_64, and of bytecodes loaded that were potentially copyable, for a) SableVM, b) OCaml, and c) Yarv.	63
5.2	Performance (speedup) results for a) SableVM (Java), b) OCaml, and c) Yarv (Ruby). Note in this figure we measured the overall VM performance, as opposed to other results in this chapter that measure VM behavior only within the interpreter function.	79
5.3	Average lengths (in bytecodes) of <i>executed</i> superinstructions for a) SableVM, b) OCaml, c) Yarv.	80

5.4	Hardware counter results for SableVM JVM running the SPEC compress benchmark. Subfigures a) and b) present branch-related results. Subfigure c) shows the number of i-cache and d-cache misses per 1M (1000000) CPU cycles.	84
5.5	Hardware counter results for SableVM JVM running the SPEC jack benchmark. Subfigures a) and b) present branch-related results. Subfigure c) shows the number of i-cache and d-cache misses per 1M (1000000) CPU cycles.	85
5.6	Hardware counter results for OCaml running the quicksort.fast benchmark. Subfigures a) and b) present branch-related results. Subfigure c) shows the number of i-cache and d-cache misses per 1M (1000000) CPU cycles.	86
5.7	Hardware counter results for OCaml running the almabench benchmark. Subfigures a) and b) present branch-related results. Subfigure c) shows the number of i-cache and d-cache misses per 1M (1000000) CPU cycles.	87
5.8	Hardware counter results for Yarv Ruby VM running the nsieve benchmark. Subfigures a) and b) present branch-related results. Subfigure c) shows the number of i-cache and d-cache misses per 1M (1000000) CPU cycles.	88
5.9	Hardware counter results for Yarv Ruby VM running the pentomino benchmark. Subfigures a) and b) present branch-related results. Subfigure c) shows the number of i-cache and d-cache misses per 1M (1000000) CPU cycles.	89
5.10	Overhead of VMs with <i>pragmas</i> inserted around bytecode instructions used by code-copying and with copied code prepared but not executed, over standard direct-threaded VMs. Part a) shows OCaml, b) Yarv, and c) averages.	93
6.1	Overview of the specialized compilation server using the enhanced GCC with two extended virtual machines.	102

6.2	Order of operations initiated by a virtual machine cooperating with our compilation server.	106
6.3	One of the obvious expected advantages of inter-bytecode optimization was the removal of unnecessary store and read operations for temporary values.	107
6.4	Profiling of superinstruction binary code can be turned on and off by simple <code>memcpy</code> and <code>memcpy</code> operations, given that the necessary memory space has been allocated in advance.	112
6.5	Source code optimization for stack accesses. One write and one read are eliminated. Others can be optimized by a regular C compiler. An actual <i>hot</i> superinstruction used by SPEC_compress benchmark. . . .	115
6.6	Summary of the results achieved with the compilation server by SableVM (Java). Y-axis values have been rescaled so as to allow for comparison of several metrics.	120
6.7	Summary of the results achieved with the compilation server by OCaml VM. Y-axis values have been rescaled so as to allow for comparison of several metrics.	121
6.8	Assembly of an OCaml superinstruction (<code>ACC4 + PUSHACC3 + LTINT + BRANCHIFNOT</code>) constructed by a) code-copying, b) sources concatenation c) sources concatenation with no <code>#pragmas</code>	125

List of Tables

5.1	Average size of an instruction (in lines of code) and fraction of copyable instructions in the 3 considered VMs. For SableVM the number in parenthesis is the number of lines after m4 macros expansion of its interpreter loop source code. Also, for the SableVM interpreter loop LOC count we did not include instruction initialization code which, in this particular VM, happens to be interleaved with instruction code. For Yarv the best performance was obtained when the maximum size of copyable instructions was limited to 100-150 bytes, lowering the number of copyable instructions from potential 71 to 61 actually used.	62
5.2	Average speedups of total execution time of code-copying over direct-threading measured across virtual machines and architectures running all used benchmarks. Note this table presents the overall VM performance, as opposed to other results in this chapter that measure VM behavior only within the interpreter function.	73
5.3	Dynamic bytecode execution metrics for SableVM (Java) on x86_64.	75
5.4	Dynamic bytecode execution metrics for OCaml on x86_64.	76
5.5	Dynamic bytecode execution metrics for Yarv (Ruby) on x86_64.	77
5.6	Direct-threaded and code-copying implementation branch misprediction percentages across architectures, for best and worst performing benchmark on each VM. For ia32 and x86_64 the numbers represent the ratio of mispredicted branches to total branches, and for PowerPC the numbers represent the ratio of instruction pipeline flushes due to branch misprediction to total branches.	78

5.7	Absolute runtimes (in seconds) of Java benchmarks presented in Figure 5.2 for the direct-threaded engine of SableVM.	81
5.8	Absolute runtimes (in seconds) of OCaml benchmarks presented in Figure 5.2 for the direct-threaded engine of OCaml.	82
5.9	Absolute runtimes (in seconds) of Ruby benchmarks presented in Figure 5.2 for the direct-threaded engine of Yarv.	82
5.10	Average branch misprediction percentages across benchmarks for ia32 on each VM, direct-threaded and code-copying.	83
6.1	Two cases of stack access optimization from SPEC-mpegaudio benchmark. a) Removes 3 reads and 3 writes to the stack. b) removes 4 reads and 4 writes to the stack. Others can be optimized by a regular C compiler. Removed read-write pairs are marked with an index number for each pair.	116
6.2	Correlation between the speedup achieved by using source-optimized superinstructions and the values of selected metrics from Figures 6.6 and 6.7. If any global correlation actually existed it would be visible on all architectures (horizontally). Stronger correlations are marked with double underline. Weaker correlations are marked with single underline.	123
6.3	Comparison of the relative performance of a single benchmark (nsieve) on both OCaml and Java VMs and different execution engines.	126
6.4	Compilation times (overhead), average speedups and resulting break-even runtime of optimized VMs.	126

Chapter 1

Introduction

Virtual machines are used as a target compilation architecture by many languages. The most widely known example is Java, but the same is true of OCaml, Ruby, Python, PHP, Perl6, Forth, and many others. In our work we are concerned with improving the efficiency of virtual machines understood as the return on investment in their development and maintenance measured in terms of the resulting performance. To this end we choose, from a number of available VM engine solutions, a simple but well-performing technique known as *code-copying*. We develop the C/C++ compiler support this technique needs to become a practical solution, we test its application to multiple programming languages and virtual machines, on a variety of hardware, and finally we use it as the basis for further VM optimization using code specialization, caching, and *ahead-of-time compilation*. By understanding, at each step, where the performance improvement is possible in the interaction between software and hardware we develop our system in the direction of increased performance but without incurring large development and maintenance costs.

Below we first present the concept of virtual machines, discuss the efficiency of software-hardware interplay in the context of our work, and finally present the contributions and outline of the entire thesis.

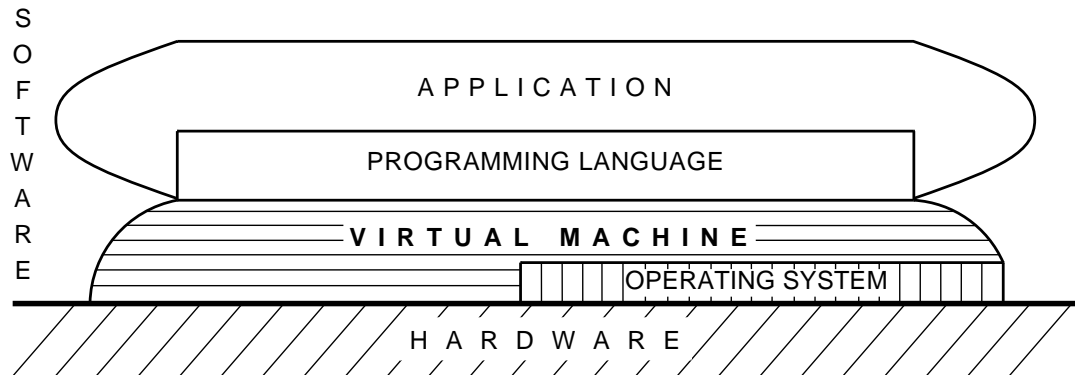


Figure 1.1: A Virtual Machine is the intermediate layer that can be compared to glue joining an application written in a programming language with the hardware it is executed on.

1.1 Virtual Machines

Virtual machines (VMs) are the intermediate layer between the actual hardware and an application written in a particular programming language. As illustrated in Figure 1.1, a VM is the layer that mediates, with the help of an operating system, between the hardware and the application written in a particular programming language. Over last two decades virtual machines have gained popularity because they enable a number of possibilities that are either not available or more difficult to exploit in more traditional solutions:

- Independence from a particular hardware and operating system platform allowing programmer to use cleaner interfaces and more universal libraries thus lowering development and porting costs.
- Transparent runtime optimization for a particular platform using all hardware and operating system-specific elements, e.g. using CPU instructions available only on a particular CPU model.
- Transparent runtime optimization based on the behavior of a particular application e.g. by focusing the optimization efforts on the most often executed code

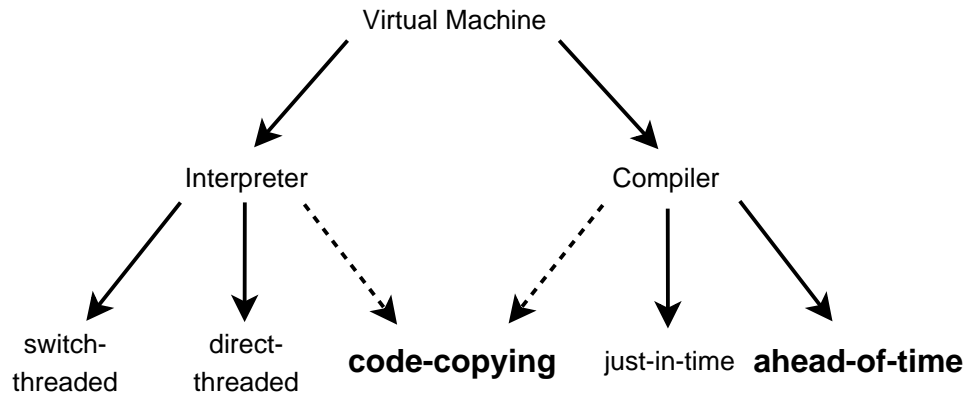


Figure 1.2: The taxonomy of virtual machines execution techniques.

(a.k.a. *hot spots*) [Sun, IBM].

- Flexibility in experimentation with new language designs and extensions to existing languages, e.g. the introduction of Java generics [NW06] required no changes to the VM, nor did Jython [Jyt] which allows for compilation and execution of Python programs on the Java platform.
- Implementation of security policies, e.g. Java web applets are run inside a *sandbox* environment with limited privileges.

A virtual machine is often a complex application that is charged with many tasks such as: efficient execution of *bytecode* (main priority), memory management (often employing automated *garbage collector*), concurrency (threading) support, dynamic code loading, exposing OS functionality, facilitating access to language libraries. Each language targeting a virtual machine uses a virtual assembly, usually called *bytecode*, to encode mostly simple operations performed on a virtual machine. The choice of the operations represented by the bytecodes and the construction of a virtual machine differ for each language and, as we will show later in this work, have profound impact on the VM design and runtime behavior.

Because of the differences between languages, resources available to developers, and their priorities, there exist a number of approaches to constructing virtual machines and, in particular, bytecode execution engines they employ. Figure 1.2 shows a rough taxonomy of the different kinds of execution techniques used by virtual machines. We will be looking at their specifics in the next chapter, in Section 2.3. Some VMs emulate the fetch-decode-execute cycle for each bytecode (interpreter-based) which provides a reasonable performance at a very low cost. Other VMs first translate (compile) a larger number of bytecodes (e.g. whole method, function, or whole application) into the underlying hardware language and then execute it (compiler-based). The compilation can take place either *ahead-of-time* (AOT), *before* the VM starts executing an application, or *just-in-time* (JIT), *while* an application is being executed. Mixed designs are also popular [SYK⁺05, MK00]. Building and maintaining a highly-optimizing compiler for a language is a very time-consuming and overall extremely costly process, and advanced research techniques such as *code-copying* offer significant promise by providing a cheap development route to better performance. Overall, we can view the different approaches to bytecode executions on a continuum, where we trade off performance for lower development and maintenance costs.

1.2 Efficiency of Software-hardware Interplay

Heuristically, development cost and performance are directly related. As a rough measure then, we can define VM efficiency as the ratio between the resulting performance and the costs of the initial development and maintenance:

$$VM\ efficiency = \frac{\text{runtime performance}}{\text{initial development cost} + \text{maintenance cost}}$$

In this sense an *optimization* is any change to a system that improves VM efficiency, as per the above definition. This is motivated by the fact that for many environments performance remains important, but the development and maintenance costs of an optimizing compiler may be outweighed by the simplicity and rapid development time of an interpreter-based VM.

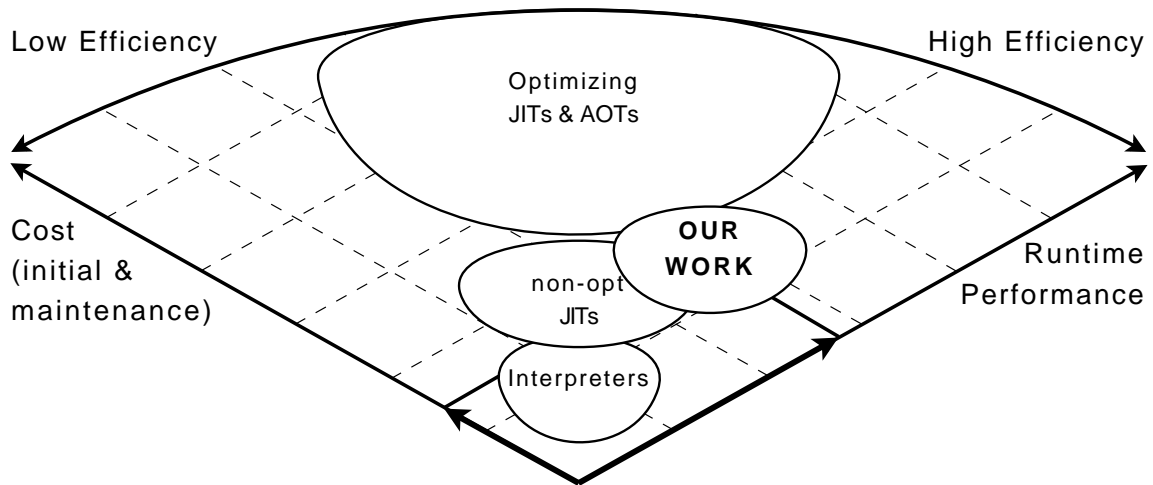


Figure 1.3: Our goal is to achieve better runtime performance while keeping the costs of initial development and maintenance of a virtual machine low.

Our work here approaches optimization from this perspective. We take two broad views on improving VM efficiency, one based on extending a known runtime optimization technique, code-copying, with important and practical safety guarantees, and one which investigates the extent to which simple Ahead of Time technique can be used to further improve performance at low cost. These two directions are intended to complement each other, showing it is possible to develop cost-efficient, well-performing solutions that come close to bridging the gap between interpreter and optimizing compiler-based virtual machines. Figure 1.3 shows the relative positioning of *our work* in relation to the VM efficiency of various executions designs. Motivation for the two major improvements of our work include:

1. Code-copying is a promising solution that provides performance better than interpreters and non-optimizing JITs while remaining simple and cheap to implement. Code-copying has been found to be very efficient in a variety of environments [RS96, PR98, Gag02, ETK06]. For example, early work in Java showed it offers an almost 2 times better performance than most popular *direct-threading* technique, and as we show almost 3 times better performance for OCaml.

2. AOT and optimized code solutions can offer better efficiency if development costs can be reduced. In our solution we make use of an existing, highly-optimizing static compiler, GCC, as the main part of a specialized caching compilation server. We create a system that operates hands-free, where the source code of VM interpreter loop is optimized for the execution of the most often occurring sequences of bytecodes that in code-copying were previously executed unoptimized. This composition of several tools and techniques provides performance even better than code-copying, over 4% average improvement on selected benchmarks for Java, and 27% average improvement in OCaml.

Below we give further motivation and detail on each of these two approaches. We organize our work around three milestones in this respect: 1) ensuring safety in code copying, 2) analyzing code-copying performance and determining relevant factors that guide its best use, and 3) extending performance further through the low-cost compilation server design.

Prerequisite of safety

Code-copying is a very attractive solution because of its simplicity and performance. Although interpreter-based, it does create code dynamically and can be thought of as a partial JIT solution (as shown in Figure 1.2). While we will describe the specifics of code-copying later, we want to make a point that code-copying, unfortunately, comes with one important, and a nearly fatal issue.

With currently available tools it is almost impossible to guarantee the execution safety of code created by code-copying, especially with variety of compilers and hardware architectures. Without such guarantee code-copying offers outstanding and cheap performance that is useless in practice.

Code-copying makes strong assumptions about the underlying compiler behavior that are not always true for modern compilers and machines. Primarily for this reason, to the best of our knowledge, there exist no production systems that use this technique,

despite its simplicity and performance advantages. This motivates the first of the 3 milestones presented in this work, described in Chapter 4, where we focus on ensuring the required conservative correctness in code optimizations and transformations. We handle the safety issue by extending the C/C++ standard and implementing this extension in a highly-optimizing, industry-standard GNU C Compiler (GCC). Despite the unusual nature of our changes we manage to follow the best compiler design practices and ensure long-term maintainability of our modifications within the GCC framework.

Broad and detailed analysis and prognostic of code-copying

The 3 interpreter-based techniques mentioned previously, in the order of increasing performance: *switch-threading*, *direct-threading*, and *code-copying*, perform differently largely because of their interactions with the underlying hardware. It is important to fully understand where the improvement comes from and why, in order to choose an appropriate technique.

Not all virtual machines, programming languages, and hardware architectures benefit the same from the use of code-copying. Before deciding whether code-copying is the right solution we want to be able to understand the reasons behind the expected future performance and be able to assess it before implementing.

This can only be achieved by gaining a broader and deeper view into how programming language features and virtual machine architectures influence the performance of code-copying on various hardware architectures. This issue is the core of the second of the 3 milestones presented in this thesis. This milestone is described in Chapter 5, where we apply code-copying techniques to virtual machines of 3 very different languages (Java, OCaml, and Ruby), on 3 different architectures (Intel 32-bit, x86.64, and PowerPC 64-bit), and perform extensive experiments to measure and analyze both, the language execution properties, and the interactions of each virtual machine with different hardware architectures.

Further optimization with a compilation server

A natural step to further improve VMs performance, beyond what code-copying is able to offer is to optimize larger portions of machine code.

The simplicity of code-copying, as well as its conservative correctness requirements mean that some opportunities for further optimization are missed. Some of them are already partially handled in hardware, by the use of caches and other means to minimize the incurred penalties, but it is much more advantageous to perform these optimizations in software. For applications that are run repeatedly the overhead of these missed optimizations accumulates.

Building yet another specialized optimizing compiler, would mean incurring a very significant cost thus, in our opinion, lowering the *efficiency* of the system. In our third and last milestone, described in Chapter 6, we propose a solution based on a caching compilation server that improves performance of VMs for repeated execution scenarios. We extend two virtual machines (for Java and OCaml) that already support code-copying to use this server to create specialized versions of virtual machines optimized for a particular application. In the server we employ an existing highly-optimizing static compiler, GCC. With this AOT-based approach we are able to further improve the performance (about 2-5% improvement over code-copying on Java and 27% on OCaml) while maintaining a good balance between the resulting performance and the overall costs of the solution, as well as ensuring an efficient, portable design.

1.3 Contributions

With our work we make the following specific contributions:

- We develop a safe and practical code-copying technique appropriate for a high-performance interpreter based on a portable extension to GCC. This provides previously elusive safety guarantees for the code-copying technique.

- Our approach ensures a maintainable design within the context of GCC while also demonstrating a simple and effective path for supporting code-copying in general in an optimizing compiler. Ensuring safety in code-copying could be performed by large, invasive efforts at nearly all levels of compilation; instead our technique minimizes the impact on other GCC components by reducing the impact to insertion of few, well-separated *passes*.
- We provide an attractive, single-compiler solution for code-copying and demonstrate implementations of this technique for three distinct virtual machines (languages: Java, OCaml, Ruby), supported on three machine architectures (Intel 32-bit, x86_64, PowerPC 64-bit).
- Using static and dynamic software metrics and hardware performance counters we provide a detailed analysis of code-copying under all combinations of our investigated languages and environments. This includes comparisons between direct-threaded and code-copying approaches. Based on our analysis we provide guidance on the expected performance of code-copying, prior to actual implementation.
- Finally, we present a system composed of multiple virtual machines that use a caching compilation server. We show how by concatenating C source code for groups of bytecodes (*superinstructions*) a virtual machine can be specialized for an application. By caching the resulting specialized binaries we provide a platform that leverages a static compiler and allows us to amortize the cost of compilation over multiple runs of an application. This provides a substantial performance improvement with minimal maintenance and development costs of the system.

1.4 Thesis Outline

In the next chapter we provide detailed background information on interpreter and virtual machine architectures followed by an overview of the related work in Chapter 3.

1.4. Thesis Outline

The first milestone, the design and implementation of our GCC compiler modifications is found in Chapter 4. The description of our second milestone regarding a detailed analysis of code-copying technique application to Java, OCaml, and Ruby on Intel 32-bit, x86_64, and PowerPC 64-bit architectures is found in Chapter 5. A presentation of our third and last milestone involving construction of a compilation server for Java and OCaml VMs is found in Chapter 6. Finally, notes on future directions and conclusions of our work are presented in Chapter 7.

Chapter 2

Background

In this chapter we focus on highlighting the concepts and tools that are essential for understanding our work. We first discuss the effects of hardware features on the performance of interpreters, then we highlight the reasons for using VMs and different approaches to their design, and then demonstrate the differences between the 3 main kinds of virtual machine interpreters, followed by a more in-depth explanation of the code-copying technique, and an overview of our AOT-based approach. We close this chapter with a summary of the benchmarks and description of the machines used throughout this work.

2.1 Hardware Architectures

The performance of an interpreter often heavily depends on the particularities of a hardware architecture it is compiled for and executed on. Some of the most important factors influencing the performance include number of registers, branch prediction capabilities, construction of the pipeline, cache size, speed of main memory.

In practice interpreters writers often hand-optimize register use by assigning a specific register to an often-used variable (like program counter of the VM) [SGBE05]. Certain interpretation techniques, e.g. *stack caching*, demand the use of one or more registers solely for the purpose of speeding up the interpretation [PWL04, Ert95].

2.1. Hardware Architectures

	Language		Bytecode instruction set	
			Bytecode dispatch intensity	
<i>Java</i>	Architecture		Bytecode sizes and complexity	
<i>OCaml</i>			Language features (memory, concurrency,...)	
<i>Ruby</i>	-----			
<i>PHP</i>	Virtual		Direct-threaded	Ahead-of-time compiler
<i>Perl</i>			Switch-threaded	
...	Machine		Code-copying	Just-in-time compiler
<i>Software</i>			Source-optimized	...
<hr/>				
	Hardware		CPU architecture	
<i>Intel 32-bit</i>	Hardware		Instruction set	
<i>x86_64</i>			Branch prediction capabilities	
<i>PowerPC</i>	Architecture		Pipeline length	
<i>ARM</i>			Available registers	
<i>MIPS</i>			Instruction and Data cache performance	
...			Memory performance	

Figure 2.1: Virtual Machine is the intermediate layer that translates architecture-independent bytecode into operations directly executable on a target architecture.

While a deeper analysis of the impact of number of registers on interpreters performance is beyond the scope of this work we shall note that, for example, the lower number of registers on Intel 32-bit architecture is often a concern when implementing interpretation techniques like stack caching. Higher number of registers allows an interpreter to keep more of its often accessed data (program counter, stack frame pointer, etc.) in registers instead of in memory thus improving the overall performance.

Branch prediction accuracy has a tremendous effect on interpreter performance. In modern hardware architectures the execution of each instruction is divided into, roughly, between 10 and 100 pipelined stages. While such design allows a CPU to

2.1. Hardware Architectures

execute more instructions per cycle it relies on CPU's long pipeline that will keep the silicon circuits implementing all the stages busy at all times. In the ideal case of linear code execution this would pose no problem but real-life code contains a multitude of control flow changes (jumps), possibly conditional. In order to keep the pipeline full such a CPU usually speculatively executes code beyond (conditional or indirect) branches and cancels the speculatively executed operations in case a branch destination turned out to be different. For such an approach to be successful it needs to rely on a branch predictor. The role of a branch predictor is to predict with as great accuracy as possible the destination of each branch. The prediction of unconditional branches is trivial. Most branch predictors focus on the prediction of destination of conditional branches but not necessarily indirect (computed) branches. The exact details vary between hardware architectures and branch prediction usually remains part of the unpublished and vaguely documented works that are critical for each vendor to gaining performance advantage over others. Usually a branch predictor can only predict a single destination of a branch and hence is rather unsuccessful when applied to indirect (computed) branches that may have multiple destinations which are the usual method of instruction dispatch in interpreters.

The size and speed of cache and main memory can also importantly impact the performance of a virtual machine. In particular, if the working body of code of a VM is larger than the size of instruction cache and the speed of the main memory is significantly lower than that of cache a noticeable slowdown can be expected. This applies, in particular, to architectures where interpreters can actually be faster than compilers because an interpreter can fit fully inside a limited amount of fast memory, while compiled code can not.

In general different hardware architectures are expected to interact somewhat differently with various approaches to interpretation and compilation in a virtual machine.

2.2 Virtual Machines

The purpose of the kind of virtual machines we are concerned with in this work is to provide a high-level abstraction of the underlying machine and its resources. In particular, VMs provide an abstraction layer with elements such as: available instruction set, architecture of the virtual CPU, memory model, object model (if any), calling convention, and many others. There exist two basic kinds of virtual machines: *system* and *process*. System virtual machines are concerned with sharing the underlying hardware resources between different operating system instances (e.g. VMWare, Xen, plex86, UML [VMW, BDF⁺03, Ple, Hos06]). These virtual machines usually offer the instruction set, memory model and other features identical to the actual hardware they are running on. This results in lower overhead than *process VMs* and very rarely need compilation-based solutions to improve the performance. We describe several existing *system* virtual machines in more detail in Section 3.3.

The other kind, the one we are concerned with in this work, are the *process* virtual machines to which we simply refer throughout this work as *virtual machines*. This kind of virtual machine is by itself an application of an operating system. As we illustrate in Figure 2.1, a virtual machine is the layer between a language architecture and hardware architecture. Just as there exist many languages, there exist many kinds of hardware. Typically a virtual machine supports one programming language and multiple hardware platforms (e.g. Java, P-Code interpreters), although of course it is possible to prepare a VM to support multiple languages.

The most important function of a virtual machine is to optimally translate the operations of a programming language (bytecode instructions) into instructions understood by hardware (machine code). As we showed earlier in Figure 1.2, page 3, the translation is usually done using a compiler or interpreter. It is important to note that in the process of translating instructions both the source (language) and target (hardware) architecture features have important influence on the resulting performance. In our work we are most interested in solutions based on an interpreter or ahead-of-time compiler.

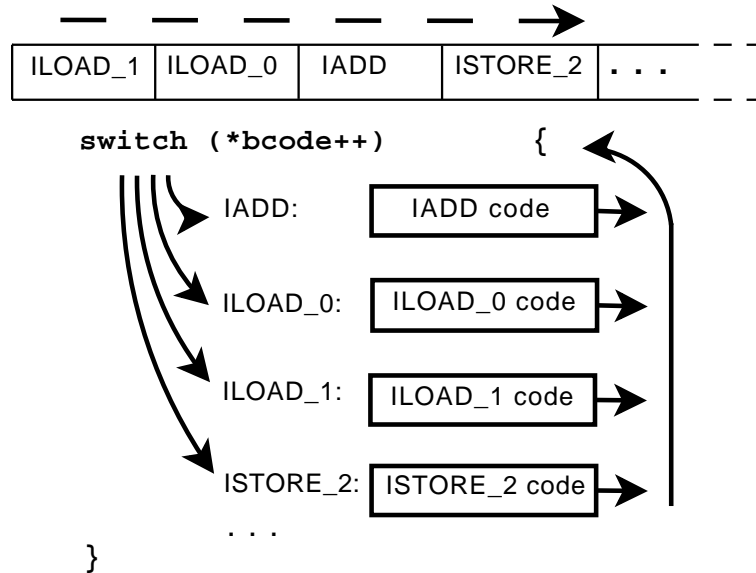


Figure 2.2: Switch interpreter mechanism and its overhead sources.

2.3 3 Kinds of Interpreters

Interpreters have the advantage of simplicity compared to compiler-based approaches. They are easier to write, maintain, and modify which allows for more rapid development and innovation. Their structure can be quite modular, making it easier to experiment with different designs for *garbage collectors*, threading models, execution engines, etc. Here we present 3 execution techniques for interpreters that build on one another to provide improved performance.

Switch-threaded

A *switch-threaded* interpreter simulates a basic fetch, decode, execute cycle. Program bytecode is fed as a stream and an interpreter repeatedly fetches the next bytecode to be executed. A large *switch-case* statement is then used to branch to the actual VM code implementing the behavior of that particular bytecode. The execution cycle

for a single bytecode in this kind of interpreter can be described as *test-branch-execute-branch-back* as illustrated in Figure 2.2. This process is straightforward and building a switch-threaded interpreter is conceptually easy. Unfortunately if, as in Java, bytecodes often encode only small operations, the overhead of fetching and decoding an instruction can be proportionally high, making the overall design quite inefficient.

Direct-threaded

A *direct-threaded* interpreter is a more advanced interpreter that minimizes the decoding overhead inherent in the switch-threaded case. This kind of interpreter requires an extension offered by some C compilers known as *labels-as-values* in order to be able to reference runtime code addresses. The C language is of particular importance since many operating systems and VMs are written in C or its close derivatives. Normally, in C a *goto* instruction can only target a statically specified *label*. With the labels-as-values extension it is possible to take an address of a label and store it in a pointer type variable. This variable can then be used as the argument to a *goto* instruction, allowing indirect control transfer. A direct-threaded interpreter makes use of this capability by replacing a stream of bytecodes with a stream of *labels* targeting the corresponding bytecode implementations. With this mechanism the interpreter can execute an indirect *goto* to immediately jump to the implementation of next bytecode. Optimization is implied by reducing the repeated decoding of instructions. The repeated *test-branch-execute-branch-back* for each bytecode execution is traded for a one-time preparatory action where a stream of bytecodes is translated into a stream of addresses. With this mechanism the operations required for one bytecode execution are simplified to a much faster *branch-execute* process thus removing a substantial amount of overhead.

Direct-threading is state-of-the-art for interpreter designs, used in GCJ, Sun, IBM, OCaml interpreter [GCJ,Sun,IBM,OCa], and many other virtual machines. It is important to notice that the speed advantage of a direct-threaded interpreter over a

2.4. Code-copying Technique

switch-threaded interpreter already comes with the requirement of additional, specialized support from the compiler used to compile the interpreter.

Code-copying

Code-copying is a further optimization to interpreter design, albeit one which makes relatively strong assumptions about compiler code generation. The basic idea behind code-copying is to make use of the compiler applied to the VM to generate binary code for matching bytecodes. The main source of performance improvement is the vast improvement application of this technique has on the branch prediction success rate in modern CPUs using BTB (Branch Target Buffer) tables. Similar to the direct-threaded approach, a preparatory step is used that translates a stream of bytecodes into a stream of addresses. Different from direct-threading, however, the preparatory process actually *copies* internal bytecode implementations. This copied code forms *superinstructions* effectively producing a *branch-execute-execute-...* execution pattern, also eliminating many internal branches. We describe this technique in greater detail in the next section.

2.4 Code-copying Technique

In a sense, and as indicated in Figure 1.2, code-copying bridges interpreter and compiler-based VM implementation approaches, and can be thought of as a „partial-JIT” technique. It creates code dynamically, but is usually considered a further optimization to interpreter design. Early works in code-copying showed significant performance improvement is possible, at least if essential safety concerns can be addressed [RS96, PR98, Gag02, PGA07].

Below we describe basic interpreter designs and implementation concerns, as well as the nature of safety considerations for code-copying.

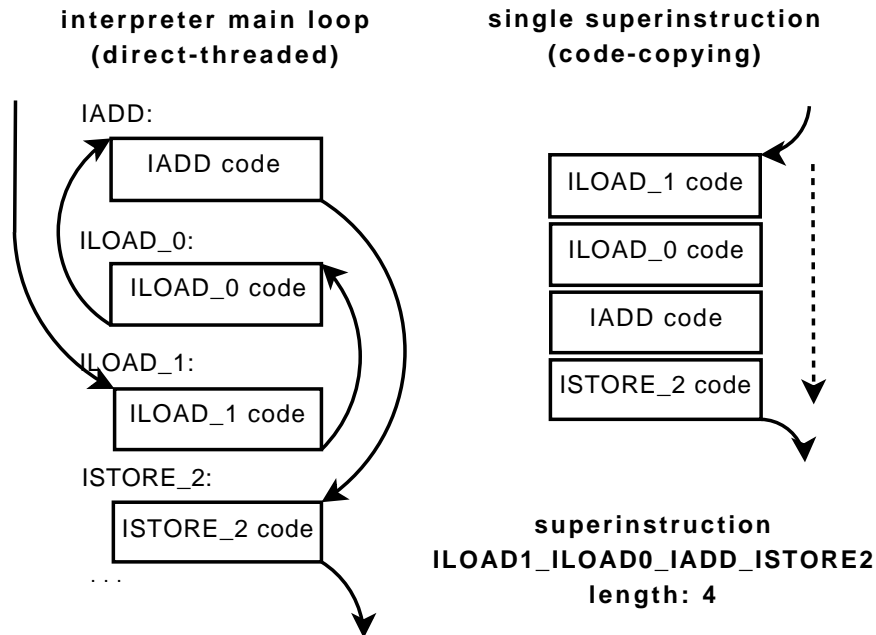


Figure 2.3: A comparison of code execution in a traditional, direct-threaded (left) vs. a code-copying (right) code interpreter. Each non-dotted arrow is a jump; the total number is reduced by code-copying and the existing jumps are more predictable, mostly due to multiple copies of the same instruction.

2.4.1 Mechanism of Code-copying

The basic idea behind code-copying is to make use of the compiler applied to the VM to generate binary code for matching bytecodes. The *chunks* of VM code used to implement the behavior of each bytecode are identified in the source code using the same labels-as-values extension as direct-threading, and then copied and replicated to generate a more efficient instruction stream. At runtime, the code-copying interpreter uses the addresses of pairs of labels to find the necessary code chunks. Each pair of labels encloses and defines the VM code actually used to implement the behavior of a bytecode. By *copying* the code between such labels and concatenating it with other code chunks from subsequent instructions, a *superinstruction* can be generated elsewhere in memory. This also eliminates branches between bytecodes in the same

superinstruction as shown in Figure 2.3.

Not all bytecodes should or can be copied in this way. Copying large bytecodes does not bring performance improvement and only causes increased memory usage and is usually avoided. Certain bytecodes that attempt unusual or special operations (the exact definition depends on the OS and VM) are also not copied. More detailed descriptions of bytecode selection techniques for code-copying and creation of superinstructions can be found in [ETK06, Gag02, PR98, RS96].

Depending on the application and other factors the code-copying technique can provide substantial performance gains over direct-threading technique. For example [Gag02] showed 1.2 to 3 times speedups for a Java interpreter. In this work we will show that by applying this code-copying to other languages the speedups over direct-threading technique can be even greater.

2.4.2 Safety Concerns

An inherent difficulty with code-copying is ensuring the integrity of the copied code. A chunk of copied code must not have dependencies on its initial location in memory, or its execution after being copied into a new place in memory will differ from the original. Early works in this area assumed a fairly naive compiler model where equivalence of the copied code and original code is straightforward. Unfortunately, this does not remain true in the presence of aggressively optimizing compilers. The C standard does not contain any semantics that would allow us to express and impose the necessary restrictions on selected parts of code. For instance the bracketing *labels* placed before and after source code chunks and used to address them do *not*, in any way, guarantee contiguity of the resulting binary code chunks, nor do they place restrictions on the use of relative as opposed to absolute addressing methods. General compiler optimizations, essential to good VM performance, may relocate basic blocks within a chunk outside of the bracketing labels, and efficient code-generation makes best use of short, relative addressing instructions. At VM runtime this will result in incomplete copies of such a code chunk, the use of relative addressing of jump or call targets outside of a code chunk, for instance, will make the copies of such a chunk

contain jumps or calls to invalid addresses. These and other related serious issues have to be handled, otherwise virtual machine crashes or undefined behavior are to be expected. To the best of our knowledge there is no production-quality solution that would ensure creation of code chunks by an optimizing C compiler that can be safely copied and executed.

Without guaranteed safety of code-copying an interpreter cannot practically, reliably make use of this powerful technique. Previous implementations used hand-done examination, trial-and-error, and manual porting combined with specialized test suites [PGA07] in attempts to ensure safety. The large effort required, and the lack of a fully verified result motivated our design presented in Chapter 4.

2.5 Source code comparison

To better demonstrate the practical differences between interpreter designs Figure 2.4 illustrates the VM code used in each of these designs. The same Java bytecode, *LCMP*, is shown as it is handled in the 3 types of interpreters. In a switch-threaded VM, we see statements `case INSTRUCTION_LCMP:` and `break` that are part of `switch` statement and together they bracket the code implementing the behavior of *LCMP*¹ instruction. In the direct-threaded code we see an `INSTRUCTION_START_LCMP:` label precedes the code. The addresses of these labels represent the the bytecode instructions in the input instructions stream, allowing each bytecode implementation to make an indirect jump, `goto *(pc++)`, to the next instruction to be executed.

In code-copying we see two labels `COPIEDCODE_START_LCMP:` and `COPIEDCODE_END_LCMP:` bracketing the implementation. At runtime, the code-copying interpreter uses the addresses of such pairs of labels to find the necessary code chunks. Overall, the implementation code remains largely the same even for more complicated instructions, while the bracketing code changes depending on the interpreter engine used.

¹LCMP compares two long (64-bit) precision scalar types.

a)

```
case INSTRUCTION_LCMP:
{ /* instruction body */
  jlong value1 = *((jlong *) (void *) &stack[stack_size - 4]);
  jlong value2 = *((jlong *) (void *) &stack[stack_size - 2]);
  stack[(stack_size -= 3) - 1].jint =
    (value1 > value2) - (value1 < value2);
}
break;
```

b)

```
INSTRUCTION_START_LCMP:
{ /* instruction body */
  jlong value1 = *((jlong *) (void *) &stack[stack_size - 4]);
  jlong value2 = *((jlong *) (void *) &stack[stack_size - 2]);
  stack[(stack_size -= 3) - 1].jint =
    (value1 > value2) - (value1 < value2);
}
goto *(pc++);
```

c)

```
COPIEDCODE_START_LCMP:
{ /* instruction body */
  jlong value1 = *((jlong *) (void *) &stack[stack_size - 4]);
  jlong value2 = *((jlong *) (void *) &stack[stack_size - 2]);
  stack[(stack_size -= 3) - 1].jint =
    (value1 > value2) - (value1 < value2);
}
COPIEDCODE_END_LCMP:
```

Figure 2.4: Addressing the effective code of LCMP Java bytecode instruction in a) switch-threaded, b) direct-threaded, and c) code-copying interpreter.

2.6 AOT-based Solution and Runtime Traces

Another source for further VM performance improvement is to advance in our approach towards actual optimizing compilation. This can be especially important for programs that execute repeatedly or over long periods of time, where the cost of compilation will be amortized. Best results can be obtained if the optimization tool is *aware* to some extent of program behavior. This allows code to be tailored or specialized to the actual execution and usually requires some form of code analysis

along with increasingly important profiling information. In the case of JITs the profile can be collected dynamically at runtime. The AOT approach requires multiple runs, where the data pertaining to one run is collected and stored to be used in subsequent runs. Typically such optimizations are heavily machine-dependant and require a massive effort to produce a full JIT or AOT compiler. In Chapter 6 we present our AOT-based solution where we leverage an existing static compiler and show how single counter-based profile data can be used to produce a low-maintenance, low-development-cost solution.

2.7 Virtual Machines, Benchmarks, and Machines Used

The approaches investigated in this thesis focus on practical results that emphasize the low development and maintenance costs before performance. We thus include extensive experiments, using a wide variety of benchmarks and architectures. Here we discuss 3 main experimental VM environments as well as the hardware architectures on which we run them throughout our work. These VMs, architectures, and benchmarks are as follows.

- *Java*

We use the SableVM Java virtual machine interpreter (v1.13) [Gag02]. This VM was chosen because of its state-of-the-art design and the fact that it already supports all 3 interpreter designs, including code-copying (originally without safety guarantees).

Benchmarks used are the standard SPECJvm98 benchmark suite [Sta] which provides a large spectrum of benchmarks with different behaviors. To further extend the range of tested applications we use two large, object-oriented in-house benchmarks SableCC (a parser generator) [GH98] and Soot (analysis and optimization framework for Java) [PQVR⁺01], and several benchmarks (bloat, fop, luindex and pmd)² from the DaCapo suite (v2006-10-MR2) [BGH⁺06].

²Not all DaCapo benchmarks are able to execute on the version of SableVM we used.

- *OCaml*

We use the OCaml interpreter (v3.10.0) [OCa] which out-of-the-box supports both switch-threaded and direct-threaded engines, and which we extended to also support (safe) code-copying in a way analogous to SableVM support. OCaml was also formerly used in research works on other techniques similar to code-copying and direct-threading [BVZB05, ZBB05].

Formal benchmarks suites, comparable to those for Java, do not exist for OCaml. A large number of benchmarks we thus use come from the interpreter itself (in the *test* subdirectory of the sources). These are small to medium-sized programs implementing and testing well-known data structures, algorithms, as well as OCaml language features. Again, to extend the range of tested applications we further added several benchmarks from the Debian Shootout suite [Deb].

- *Ruby*

Ruby interpreter before version 1.9 (release candidate at the time of this writing) used an interpreter engine largely incompatible with direct-threading or code-copying and hence unsuitable for our purposes. Instead, we base our investigation on the Yarv (Yet Another Ruby VM) interpreter (v0.4.1) [Sas05], which has since become the official engine for Ruby 1.9 due to its better design and performance. Yarv not only supports direct-threading but it also contains analysis of bytecode that have the potential to make code-copying more efficient in some cases.

Ruby is typically used as a tool for development of short scripts and web applications [Rub], and performance-based benchmarks for Ruby are even less common than for OCaml. With growing acceptance of Ruby comes interest in improving its performance which makes Ruby an interesting, industry-relevant research target. We selected the larger (mainly in the terms of runtime) of the benchmarks that are distributed with the VM source code (in the *benchmark* subdirectory) and added several benchmarks from Debian Shootout. Because most of those Ruby benchmarks had relatively small size compared to Java or

OCaml benchmarks we decided to also include Yarv's supplied self-test program *test-all* with a substantially larger total code size.

Our work includes investigating and optimizing VM design, particularly as it relates to the underlying hardware architectures. We thus employ several machine types in attempt to fully assess the impact of different hardware designs. The test configurations in this work used the following hardware as representatives of the most popular hardware configurations.

- *ia32* – 32-bit Intel. This is a typical desktop machine.
Pentium 4 3GHz with hyperthreading. L2 cache size of 1MB, L1 data cache size of 16kB, L1 instruction cache (trace cache) 12K ops. This machine had 1GB RAM installed and ran Debian GNU Linux 4.0 ("Etch") using Linux kernel version 2.6.18 with SMP support, optimized for i686 machines.
- *xeon* – 32-bit Intel. This is a typical server machine.
Intel Xeon 2.4GHz with hyperthreading. L2 cache size 512kB. This machine had 2GB RAM installed and was running Debian GNU/Linux 4.0r2 (codename "Etch").
- *x86_64* – 64-bit x86_64 – This is a modern 64-bit machine that can serve as a desktop or small server.
AMD64 (Athlon64) Dual Core 3800+ 2GHz. L2 cache size was 512kB per CPU (1GB total), L1 data cache 64kB per CPU and L1 instruction cache 64kB per CPU. This machine had 4GB RAM installed and ran Ubuntu 7.10 (codename "Gutsy") using Linux kernel version 2.6.22 with SMP support.
- *ppc* – 64-bit PowerPC. This is a 64-bit PowerPC machine with a RISC CPU (all other machines are CISCs). Can be used as a desktop or server.
Power Macintosh G5 with two CPUs 970 running at 1.8GHz. L2 cache size was 512kB per CPU (1GB total), L1 data cache 32kB per CPU and L1 instruction cache 64kB per CPU. This machine had 1.5GB RAM installed and was running Mac OS X Server version 10.4.11.

2.7. Virtual Machines, Benchmarks, and Machines Used

In the next chapter we present an overview of existing works related to the main areas of our work.

Chapter 3

Related Work

The work presented in this thesis is within several areas of research, mainly: byte-code interpretation methods, code optimization, static and just-in-time compilers, virtual machines, and compilation servers. In this chapter we present a representative selection of the most relevant works in these areas. Note that this chapter only contains descriptions of the related works and does *not* discuss the differences between these works and our work. To make the understanding of the content of this thesis easier the discussion of differences is located in a designated section in each of the chapters discussing the 3 milestones of our work (Sections 4.5, 5.4, and 6.7).

Certain related works mentioned below could be potentially assigned to more than one section, therefore our placement is somewhat subjective. We decided that the techniques that *modify* the machine code, and/or that need specialized knowledge about an architecture, or perform machine code generation (copying memory is not code generation in our understanding) would fall into the category of *Compilation Techniques*. In the *Virtual Machines* section the aim was to gather related work pertaining to *complete execution environments*, some of which might be using compilation, or interpretation, or both, or other techniques. We also included *hardware virtualization* systems in that section. Compilers, virtual machines and relevant solutions aiming at resource sharing were given a separate section on *Compilation Servers and Resource Sharing*.

3.1 Interpreters

The most basic (and slowest) bytecode interpreter design is known as *switch-threaded*, basically consisting of a *while* loop containing a large *switch* statement where each bytecode is processed by branching to the appropriate internal functionality. While simple in concept and implementation such interpreters suffer from an extremely costly dispatch overhead, and it is much better to use the more effective *direct-threaded* technique. These approaches have been compared and analyzed by Ertl and Gregg in [EG01,EG03a] and are explained in more detail in the next chapter.

Much of our work is centered around the technique of *code-copying*, also (somewhat confusingly) known as *code inlining*, *selective inlining*, or (more meaningfully) *dynamic superinstructions*. Code-copying is a technique originating from direct-threaded interpretation. It was first described by Rossi and Sivalingam [RS96] and later in a better known work by Piumarta and Riccardi on, what they called, selective inlining [PR98]. Compilers used at the time were not too aggressive in their optimizations and thus these works did not face many of the challenges that the use of code-copying poses today, due to common use of highly-optimizing compilers. Still, we have to note that their solutions, while delivering substantial performance improvements, did not mention the concern for, and did not provide safety guarantees.

The most important reason why code-copying is significantly faster than other interpretation techniques is its positive influence on the success rate of branch predictors commonly used in today's hardware containing branch target buffers (BTB). Application of the code-copying technique to GForth has been analyzed in a few studies by Ertl et al. [EG03c,ETK06,EG03b], showing that the majority of performance improvement is due to improvements in branch prediction. Ertl et al. also compared code-copying (called in these works *dynamic superinstructions*) to other techniques like *dynamic* and *static instruction replication* and *static creation of superinstructions*. They demonstrated that all of these techniques (often combined together) can also bring significant performance improvements. Their results also demonstrated, once again, that speedup due to branch prediction improvements expectedly outweighs other negative effects, such as a slight increase in instruction-cache misses.

3.1. Interpreters

Peng et al. [PWL04] described an interpreter using a *stack caching technique* which exploits the fact that in stack-based virtual machines the top stack elements are the ones accessed most often. By forcing C compiler to use registers as a cache for these elements and creating multiple versions of certain instructions (for accommodating different arrangements of stack elements in the cache) while maximizing code reuse this technique achieved visible speedups of about 13%. Analyzes of stack caching application in interpreters can also be found in works by Ertl et al. [EG03b, Ert95].

Berndl, Zaleski et al. [BVZB05, ZBB05] introduced a new technique they called *context threading*. This technique leverages the existing hardware prediction mechanisms for *call* and *return* assembly instructions to remove about 95% of mispredictions. They show it is possible to further improve the speed of their technique by selective use of code-copying for very small bytecode instructions. Vitale et al. [VZ05] analyzed applicability of this technique to a Tcl interpreter characterized by large bytecode bodies and, for most applications and benchmarks, little time spent in interpreter dispatch. On a selected set of dispatch-intensive benchmarks they achieved an almost 10% speedup.

Gagnon was the first to use the code-copying technique in a Java interpreter, SableVM [Gag02]. This implementation solved some important problems specific to the interpretation of Java bytecode by analyzing the bytecode before execution and, for example, splitting certain operations into multiple, VM-specific bytecodes. The code-copying engine it featured required manual tuning that could not give guarantees of safe execution and therefore could not be regarded as a production-ready solution. Interestingly, SableVM provided very good performance. For example, experiments with a simple, non-optimizing portable JIT for SableVM (SableJIT [B04]) showed that such a JIT was only barely able to achieve speeds comparable to the code-copying engine. This demonstrated once again that code-copying is a very attractive solution, save only for its lack of safety.

3.2 Compilation techniques

A solution similar to a code-copying engine is a JIT compiler using code generated by a C compiler, as developed by Ertl and Gregg [EG04] for Gforth. In this solution the resulting binary code is created by concatenation of binary code chunks of the VM itself that are found by placing labels at their beginning and end. The main difference between this solution and code-copying is that the resulting concatenated code is actually modified (patched) on the fly, so as to contain immediate values and remove the need for the instruction counter. The patching process requires architecture-specific knowledge about the machine code, hence we consider this solution a code-generation technique. A very similar solution has been applied to TCL interpreter by Vitale and Abdelrahman [VA04] but the resulting performance gains were much smaller than for Gforth because bytecodes in TCL are much larger and thus the dispatch overhead to be removed is much smaller.

Other solutions in this area include systems like DyC [GPM⁺04]. DyC dynamically recompiles programs during their execution so as to enable the use of run-time values. Such an approach benefits from allowing for optimizations based on partial evaluation. A similar solution was presented by Consel et al. [CLLM04] in a C program specialization system. It included run-time specialization based on C source code templates compiled ahead of time with a static compiler and then used at run-time. The specialization technique was applied to Java by Masuhara and Akinori [MY01] and Java and OCaml by Thibault et al. [TCL⁺00] with good results. Of course, moving towards a full, optimizing JIT represents a significant resource commitment, out of the reach of many scripting or experimental languages. Zaleski, Brown, and Stoodley developed Gradually Extensible Trace Interpreter (YETI) [ZBS07] that couples interpreter with a trace-oriented JIT compiler to allow for gradual development of the JIT compiler and lowering the cost of the initial development. There also exist portable JITs like GNU Lightning (used e.g. in OCamlJIT by Starynkevitch [Sta04]), but these often come with support for limited number of platforms and their own limited set of code primitives.

Specialized interpreters are another route to optimized performance. In *Vmgen*

the VM system can be trained on a set of programs to detect the most often occurring small sequences of bytecodes. The source of an interpreter is then automatically modified to combine these sequences into superinstructions, optimized the next time the interpreter is recompiled [EGKP02]. Such a generalized solution can be used for rapid development of systems that normally include an interpreter, as done by Palacz et al. [PBF⁺03]. Somewhat similar in spirit is a proposal by Varma and Shuvra [VB04] for a system where Java bytecode is translated into custom-generated C code, including Java-specific optimizations, and then compiled using a standard C compiler. While the speed benefits of these solutions are indisputable, they still require non-automated training, selection of the set of training programs and interpreter recompilation.

Profile-based tuning has also been used to improve bytecode execution. An optimization based on exploitation of frequently occurring bytecode sequences was shown by Stephenson and Holst under the name of *multicode substitution* [SH03]. In this work hot sequences of bytecodes are discovered off-line by ahead of time profiling and new code is created (either by a JIT or ahead of time in an interpreter) with aggregated, compact instructions representing larger execution sequences. Stephenson showed that to limit the total number of instructions (including those created by the optimization itself) such an approach must be combined with careful selection of sequences based on how well a sequence of bytecodes can be optimized.

Hybrid and other approaches to bytecode execution are of course also possible, which are discussed in the next section on complete virtual machine solutions. Taking a compiler-centered view of these solutions we shall note two approaches demonstrated by Bothner [Bot03] in GCJ and Lattner and Adve [LA04] in LLVM. GCJ is a GCC-based Ahead-Of-Time compiler including a direct-threaded interpreter for dynamically loaded code. GCJ takes as its input either Java source or Java bytecode (class files) and compiles them to an architecture-specific executable. Its main use is in embedded devices where memory footprint is critical and competitive JIT-based solutions are either too large for a small device or too costly due to their licensing. LLVM is a compilation framework created for lifelong program analysis that features its own code representation, own compiler and other tools that make it extendable and

reusable. Almost every performance-oriented virtual machine features a just-in-time optimizing compiler able to perform method inlining, branch-optimization and other optimizations. Such a compiler often permits several optimization levels chosen depending on how often a section of code is executed. Ma and Pirvu [MP08] presented an interesting work on improving VM startup time by using ahead-of-time (AOT) compilation. Dynamo [BDB00] is an advanced system that employs an interpreter of native code to detect common execution paths and a highly specialized optimizing compiler to optimize future code execution along these paths. Its focus is on runtime recompilation and optimization of native code.

3.3 Virtual Machines

Virtual Machines are complete vehicles that internally can use different solutions to achieve their goals. Here we present several popular Virtual Machines for Java that differ significantly in their design. The most common approach is to use an interpreter engine or non-optimizing (e.g. template-based) compiler for the early execution of code and rarely executed code (a.k.a *mixed-mode* execution). Then, in performance-oriented solutions, to gather an execution profile and use an optimizing just-in-time compiler, as described earlier. The design of IBM's Java VM [IBM] and Sun Microsystem's HotSpot Java VM [Sun] uses an interpreter and a highly optimizing JIT compiler (with several optimization levels) used only for frequently executed code. A different approach has been taken by the architects of Kaffe [Kaf], which offers a standard *direct-threaded* interpreter on many architectures, and employs a good optimizing JIT compiler on selected few due to the limited resources of the project. SableVM [GH01] is a Java virtual machine that features 3 kinds of interpreters (switch, direct, inline-threaded) built from a single set of definitions of bytecodes, is highly portable, and focuses on maintainability and quality of code. JikesRVM [Jik] is written in an extended Java, does not make use of an interpreter, but employs a JIT compiler with 3 levels of optimizations. Its focus is on delivering high performance. The Parrot Virtual Machine [Par] is a relatively new project that

3.4. Compilation Servers and Resource Sharing

created a VM capable of executing bytecode of multiple programming languages like Perl6 or Python. The VM is a register-based VM, combined with a translator that inputs a stream of bytecodes of a language and translates it into a stream of bytecodes understood by the VM.

Virtual machines and virtualization matters are not limited to only execution of code for virtual architectures. There exist a number of solutions used to virtualize actual hardware, known as *system virtual machines*. System virtual machines are concerned with sharing the underlying hardware resources between different operating system instances. The *VMWare* [VMW] emulator recompiles native **x86** architecture code on the fly to execute both user-level and kernel-level code within a user-level program, and emulate the standard PC hardware. It allows for multiple virtual PC machines executing on a single physical machine with a single *host* operating system. A similar approach is used by *plex86* [Ple]. Another solution known as User Mode Linux [Hos06] that allows a modified Linux kernel to be executed in userspace as an unprivileged application that can then be used as a virtual Linux machine to execute other applications. All these solutions require that a single *host* OS is installed and running on a physical machine, and that the virtual machines are run as clients (applications) on top of the host OS layer. A recently developed solution is known under the name *Xen* [BDF⁺03], with the goal of running multiple equivalent operating system instances, with no *host* OS on a single machine. This solution is more lightweight but it requires slight modifications to the kernels of operating systems running on a virtualized physical machine to include more advanced mechanisms of hardware resources sharing.

3.4 Compilation Servers and Resource Sharing

In some environments, due to imbalance of resources available to different devices, or due to opportunities for avoiding performing identical tasks by multiple agents, sharing of certain resources (compilation services, compiled code, execution environments) and optimizing ways of transferring the resources (e.g. minimizing network

3.4. Compilation Servers and Resource Sharing

traffic, transferring running programs or live objects) can bring tangible benefits. Below we present a selection of works using these techniques.

Lee et al. presented a compilation server [LDM04] based on JikesRVM. This system uses a central, powerful machine running JikesRVM as a compilation server that serves smaller, embedded devices connected to the server via relatively slow links. They achieved a significant improvement of execution time, pause time and memory allocation on the client virtual machines. Another server-based solution targeted at handheld devices was presented by Palm et al. Its goal was to reduce power consumption [PLDM02]. Franz proposed a different system for centralized code generation [Fra97]. It uses a specialized, platform-independent software distribution format twice as dense (i.e. taking half the space to carry the same information) as Java bytecode. In this system the generated code is translated into unoptimized native code at the destination VM, at load time. The native code, including libraries used by it, is profiled and optimized using otherwise idle system cycles. These systems demonstrate that a compilation server is a useful tool that can improve certain characteristics of VM execution by sharing resources among VMs.

There were other attempts at using shared resources to improve performance of VMs. Cabri et al. implemented a mechanism [CLQ06] to capture the state of a running thread and restore it on a different JVM. Such mechanisms can be used to parallelize multithreaded programs and balance the computational load among multiple physical machines. A review of the research on a related mechanism – object persistence in Java – can be found in a work by Lunney and McCaughey [LM03]. Joisha et al. [JMSG01] modified a VM to share the binary executable code among multiple VMs. Importantly, with this technique they were able to reduce the amount of writable memory used.

As the basis for our later work on fast interpretation of multiple programming languages and implementation of a VM-oriented compilation service, in the next chapter we present an enhancement to GNU C Compiler (GCC) that enables safe code-copying.

Chapter 4

Static Compiler (GCC) Enhancement

The main advantage of using an interpreter is its simplicity that results in low costs of development and maintenance. The performance of interpreters, however, has always been their weakest point.

The code-copying technique provides a vast performance improvement at a very low engineering cost. The biggest problem with this technique is the difficulty of providing safety guarantees. Without guaranteed safety of code-copying an interpreter cannot practically, reliably make use of this powerful technique. Previous implementations used examination by hand, trial-and-error, and manual porting combined with specialized test suites [PGA07] in attempt to ensure safety. The large effort required, and the lack of a fully verified result motivates the work presented in this chapter.

As we discussed previously, the most simplistic, switch-threaded interpreter was the basis for a faster design known as a direct-threaded interpreter. Direct-threading is a technique that used a *compiler extension* known as labels-as-values. Specialized support of direct-threading in GCC and other compilers permitted this technique to become the most commonly used one by interpreter developers. In regard to code-copying we believe that providing a similar support in an industry-standard compiler will make this technique much more attractive in practical applications.

In this chapter we present our enhancement to an industry-standard GNU C Compiler (GCC). As we said previously, our goal was to create a system where the superior performance and simplicity of code-copying technique can be exploited, while

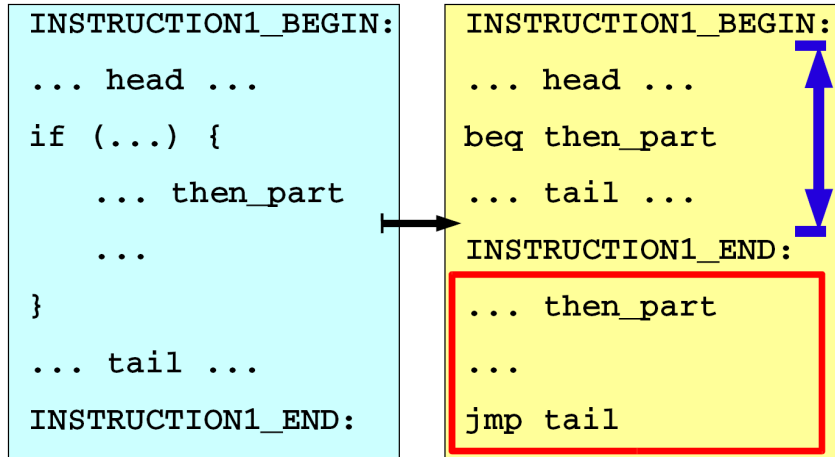


Figure 4.1: Optimizing compiler can relocate less likely executed code to the outside of labels bracketing code used by code-copying.

providing the necessary safety guarantees. We demonstrate that with this enhancement code-copying Java VM (SableVM) can safely achieve speedup up to 2.7 times, 1.5 on average, over the direct interpretation, thus proving that this maintainable enhancement makes the code-copying technique reliable and thus practically usable.

Below we first present the design of our enhancement, the experimental results, discussion of related work and finally the conclusions.

4.1 Problem of Compiling for Code-copying

As numerous studies have shown the performance gains from using code-copying technique are clear [ETK06,EG03c,Gag02,GH01,PR98]. However, one of the biggest problems faced by the developers of code-copying interpreters is ensuring that the fragments of the code chunks copied to construct superinstructions are still fully functional in their new locations and as parts of superinstructions. In an optimizing compiler, like GCC, there are a number of possible issues caused mostly by a certain few optimizations.

- *Basic blocks partitioning.* Optimizing compilers divide basic blocks into likely

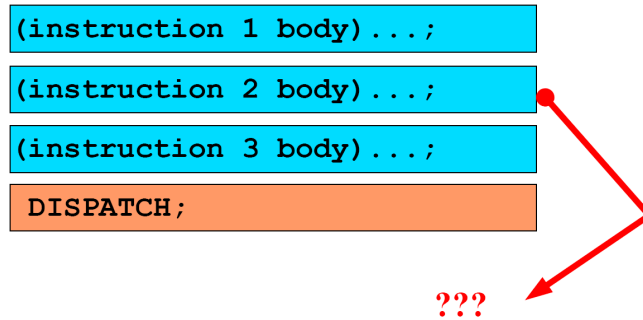


Figure 4.2: Execution of a superinstruction containing a code chunk with a missing part or a call using relative addressing might cause VM crash.

executed ones (hot) and not likely executed (cold). Blocks belonging to each group are put together, so as to improve cache efficiency. Unfortunately this optimization often moves a basic block belonging to the internal control flow of a code chunk to the outside (usually far away) of the bracketing labels of the code chunk thus making it unusable for code copying (see Figure 4.2).

- *Most often executed path optimization.* As illustrated in Figure 4.1 an optimizing compiler can relocate code that is less likely to be executed, like null pointer checks (common in many bytecodes) to the outside of pair of labels bracketing the code chunk. If this happens, such a code chunk can not be used for code-copying. This is because the only code that is copied is the code between the two bracketing labels. There is no easy and portable way that a VM can even detect such unusable code chunks at runtime. What is worse, when such a code chunk is used (see Figure 4.2) in code-copying and the less likely execution path is encountered then the relocated part of code is missing from superinstruction an undefined behavior will occur resulting most likely a segmentation fault.
- *Jumps using relative addressing.* A regular C `goto` to a label can be translated by a compiler into an instruction using a relative or absolute addressing. If a relative addressing method is used and the target is outside of the copied code chunk then such a bytecode is not suitable for code-copying. This is because the

target of the jump is dependant on the position of the code, and this position is changed when the code is copied. Again, there is no easy and portable way to detect this problem in a VM and using such a bytecode makes a VM unreliable.

- *Calls using relative addressing.* On many popular architectures, e.g. on Intel, the target address of a call is specified using an address relative to the currently executed instruction (unless the call is to a far target, which is rarely the case). A code chunk containing such call can not be used for code-copying for the same reasons as in the case of a relative jump.

Previous works [PGA07] attempted to remedy some of these problems by modifications to the C source code or disabling some of the compiler optimizations. The usual result was mostly a lower likelihood of encountering the above problems, but not an actual guarantee. These attempts prompted us to solve the problem at the source, that is, in the compiler itself.

4.2 Architecture of a Compiler

The modifications to GNU C Compiler (GCC) presented later in this chapter are easier to understand when viewed over a draft of a general compiler architecture. In Figure 4.3 we present a simplified data-flow oriented structure of a C compiler. For the C language the process of compilation is preceded by processing of C source code by the C preprocessor (`cpp`) and its output becomes the input for a C compiler. The compiler first parses the source code and translates it into an *intermediate representation* (IR). For GCC this intermediate representation is a data structure known as `tree`. This representation is then processed by multiple *compilation passes* that incrementally analyze and transform it in a way that improves the quality of the code. In this process additional data structures are built around the intermediate representation, like *basic blocks* (BB) and *control flow graph* (CFG). Basic block is a unit of code that has one entry point, one exit point, no jumps within the unit (with the exception of the last operation in the unit) and no outside jumps that jump to the middle of the unit. Basic blocks form vertices (or nodes) of the control flow

4.2. Architecture of a Compiler

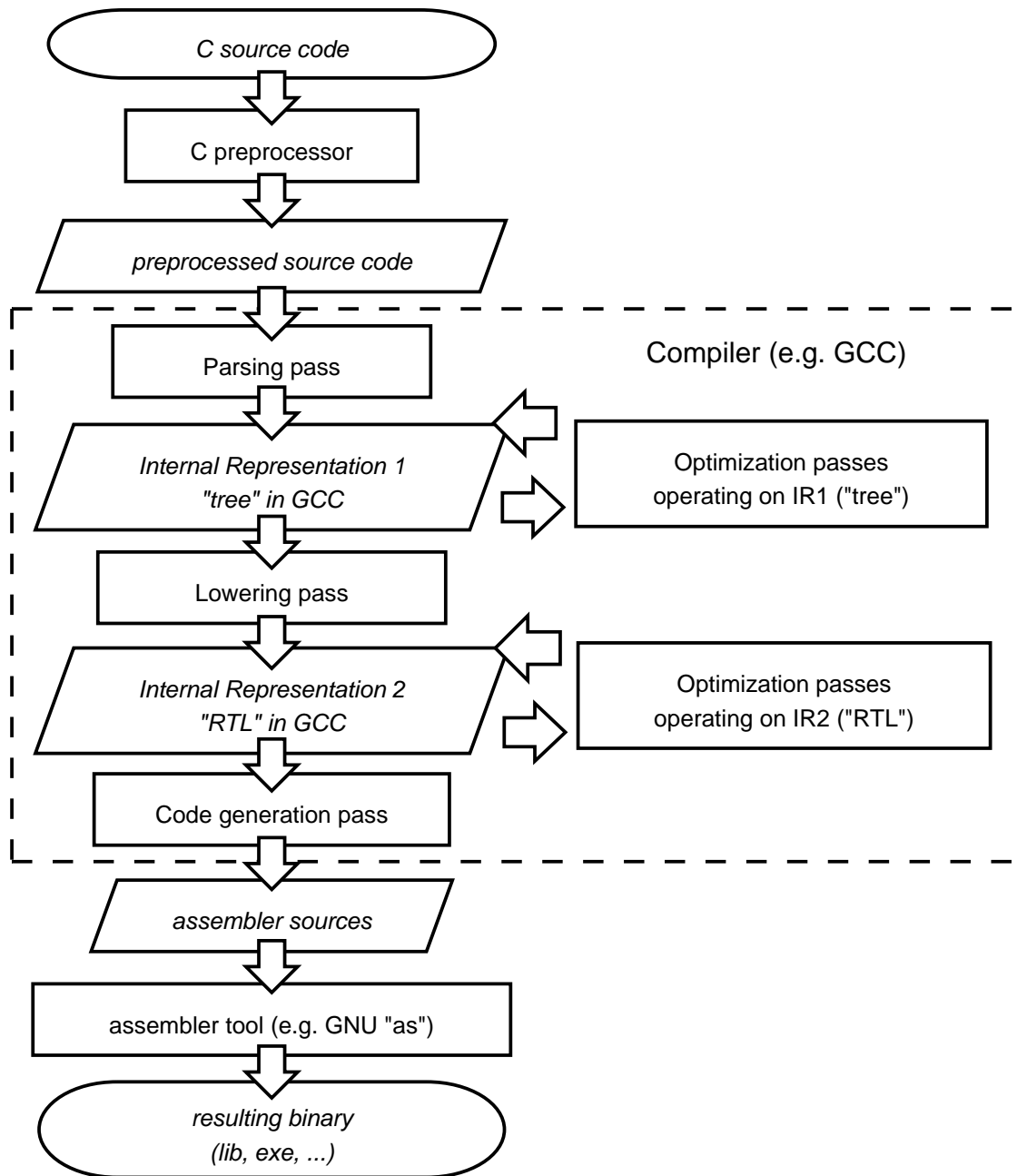


Figure 4.3: A largely simplified view of a C/C++ compiler based on GCC internal structure.

graph. The control flow graph is a representation of all control flow paths that might be visited during an execution of the code. After all passes operating on the `tree` IR are completed¹ that representation is transformed into another one, in the case of GCC known as *Register Transfer Language* (RTL). Register Transfer Language is a lower-level representation that is very close to the actual assembly of a specific hardware architecture². Again, multiple passes process this representation of the code analyzing it and incrementally transforming it to ensure better quality resulting code. In GCC the total number of passes applied to all intermediate representations (IRs) depends on optimization options used and is usually greater than 50. The last pass of the compiler is the code generation pass. This pass translates the lower intermediate representation, RTL, into textual assembly output. This output is then used by an external, architecture-specific assembler tool (e.g. GNU `as`) to create the resulting binary object (e.g. an executable or library file).

It is important to note that given the general structure of compilers, and C/C++ compilers in particular, the solution we propose in this chapter is not a special case based on particularities of a specific compiler (GCC). Rather, it is based on the principles on which compilers are built and is expected to be just as applicable to other compilers.

4.3 Design

Many virtual machines are implemented in C, and problems arise from the fact that there is nothing in the C standard nor in most C compiler capabilities that guarantees that part of binary code copied from one place in memory to another will be functionally equivalent to its original image. Code optimization can rearrange instructions outside of expected, source-based limits, and without specific optimization guarantees it is difficult to ensure the correctness of the final implementation. A more in-depth overview of the code-copying technique can be found in Section 2.4.1. Here,

¹The actual mechanism is more complicated but the explanation is omitted to improve clarity.

²Definitions of Basic Block, Control Flow Graph, and Register Transfer Language based on Wikipedia [Wik] content.

we immediately continue to the design of our specialized support for this technique.

Our approach is to use manual specification by source annotation in the form of the well-known `#pragma` operator. This operator is used to surround and thus help identify copyable code chunks. The bulk of our design effort is in ensuring safety for code copying, a result guaranteed by a small set of well-specified additional passes within GCC. Below we first detail requirements for code to be *relocatable* and thus suitable for code-copying, followed by a description of the GCC modifications, including the final verification phase.

4.3.1 Generation of safely copyable code

There are specific requirements that a chunk of code has to meet so it could be copied to another location in memory, concatenated with other chunks and safely executed. A code chunk can only be safely copied if its copy is *functionally equivalent*, i.e. chunk of code $C_{baseaddr\alpha} \equiv C_{baseaddr\beta}$ where $\alpha \neq \beta$.

We thus define a chunk of code C to be *copyable* if all of the following conditions ensuring functional equivalence are true:

- C occupies a single contiguous space in memory that starts and ends with two distinct code labels specified by a programmer.
- Natural control flow enters C only at its “top” and exits only at its “bottom.”
- Any jump from inside of C to code outside of C (e.g. to an exception handler) uses an absolute target address.
- Any jump from the inside of C to another place inside C uses a relative target address.
- Any function call from inside of C uses an absolute target address.
- At C boundaries registers must be used consistently with other code chunks boundaries (this is already ensured by GCC’s computed goto extension).

4.3.2 GCC modifications

Our goal was to modify a highly optimizing C compiler, such as GNU C Compiler 4.2, to selectively generate code that meets these requirements, thereby ensuring functional equivalence of selected code chunks.

As we have shown before in Section 4.2 a regular optimizing C compiler like GCC applies over 50 passes to process the code. These passes modify the code in ways that are usually supposed to improve the speed of the resulting code or its other parameters. Given the size, complexity and continuous development of the optimization passes it is not feasible to modify and maintain all of these passes to selectively generate code conforming to our requirements. Instead, we modify the compiler to:

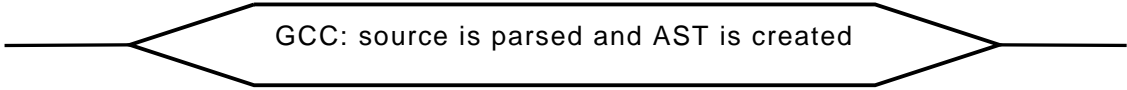
- preserve the information about which parts of the code have to be treated specially—from the moment the source code is parsed to the moment the final assembly is generated,
- allow (almost) all of the optimizations to execute without modifications and then at certain selected points of the compilation process use additional passes that modify the code in a manner that makes selected code chunks copyable.

One can think of it as a cross-cutting approach that works by finding, preserving, and processing data across multiple compilation passes.

The overall set of modifications is divided into separate passes that collectively track or restore information throughout the whole compilation process; a general description is shown in Figure 4.4. Depending on the representation of the code at each stage of compilation this information is tracked in a different form. In the source code it exists as *#pragma* lines, then as special flags of selected AST elements, later we attach it to basic blocks and *computed goto*'s, and eventually it is inserted in the form of *notes* into the assembly (RTL). Tracking this information turned out to be the most difficult part of our work. It is because of all the aggressive optimizations that might duplicate, remove, and move parts of the code in which we are interested that ensuring copyable code is non-trivial. We ensure that this information is not

4.3. Design

I. Register pragma locations start/end during parsing



GCC: source is parsed and AST is created

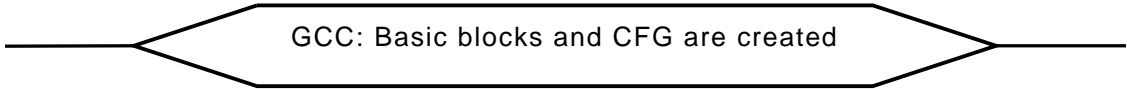
II. Scan the tree (twice)

SCAN 1:

- ensure each pragma location is followed by a label
- flag these label statements as BEGIN & END
- insert volatile assembly around END labels

SCAN 2:

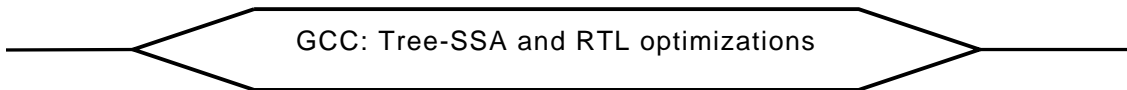
- modify gotos within the copyable areas to use absolute addressing (via register) if the target is outside of an area
- modify calls within areas to use absolute addressing (call via register)



GCC: Basic blocks and CFG are created

III. Insert permanent marking and ensure areas are solid

- initial permanent marking of BEGIN/TARGET basic blocks
- restore marking of copyable areas using BEGIN, TARGET and computed gotos as boundaries (reusable pass)

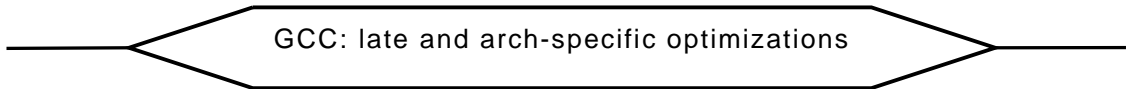


GCC: Tree-SSA and RTL optimizations

IV. Correct ordering of basic blocks in copyable areas

- restore marking of copyable areas (reusable pass)
- reorder basic blocks of copyable areas

V. Insert RTL markers of copyable areas boundaries



GCC: late and arch-specific optimizations

VI. Verify RTL of copyable areas

- ensure the copyable-code properties hold

Figure 4.4: To produce copyable code with minimal changes to the internal structure of the compiler we inserted several well isolated passes.

4.3. Design

```
case SVM_INSTRUCTION_LCMP:
  { /* instruction initialization */
    vm->instructions[instr].param_count = 0;
    vm->instructions[instr].copyable_code = &&COPYABLE_START_LCMP;
    env->vm->instructions[instr].copyable_size =
      ((char *) &&END_LCMP) - ((char *) &&COPYABLE_START_LCMP);
    break;
  }

#pragma copyable begin
COPYABLE_START_LCMP:
  { /* instruction body */
    jlong value1 = *((jlong *) (void *) &stack[stack_size - 4]);
    jlong value2 = *((jlong *) (void *) &stack[stack_size - 2]);
    stack[(stack_size -= 3) - 1].jint =
      (value1 > value2) - (value1 < value2);
  }
#pragma copyable end
END_LCMP:
```

Figure 4.5: Pragma directives are placed around the code that will be used by code-copying engine at runtime.

lost, misplaced or mangled by separating it from structures accessed by optimization passes, where possible, and by employing multiple sanity checks in each of our passes that use this information.

Below we discuss in great detail our modifications to GNU C Compiler as presented in Figure 4.4. For an overview of the data-flow structure of a C compiler in which these changes have been implemented please see Figure 4.3.

Phase I: Register pragma locations

Figure 4.5 illustrates a fragment of interpreter source code for a single instruction. The code of an instruction (bytecode) is surrounded by the special *copyable #pragma* statements that mark the beginning and end of the copyable chunk. To make the compiler recognize and accept the pragmas we reuse the existing standard mechanisms for handling pragmas inside GCC.

Phase II: Scan the tree (1)

To ensure chunks are properly identified and separated an initial pass is performed to check starting and ending conditions. Each location of *#pragma copyable begin* and *end* registered during parsing is checked to ensure it is followed by a label. These *start* and *end* labels have then their special *start* and *end* flags set accordingly. Finally the code is modified by artificially inserting into the stream of statements two empty *volatile assembly* instructions around the *end* label.

The volatile assembly code acts as a barrier to code movement, and is used to ensure the basic blocks directly following areas, the *target* blocks, are preserved and act as the sole and unique exits of the natural control flow from a copyable area. Our tests showed that otherwise some optimizations would attempt to remove or merge *target* blocks. In principle a similar concern applies to the first basic block of a copyable area, the *starting* block. However in our tests the compiler would never try to remove or duplicate this block. We did not investigate it further, but if it ever became a problem such issue could always be handled the same way as in *target* blocks. We implemented sanity checks that would fail if blocks marked *start* or *target* were removed or duplicated.

Phase II: Scan the tree (2)

In most architectures control flow jumps can be *relative* or *absolute*. Relative jumps have the advantage of being (usually) smaller instructions, but having a machine-specific limitations on the distance for which they are useful. Absolute jumps are often longer instruction sequences since the complete target address must be encoded, not just the relative displacement. As mentioned in Section 4.3.1 for control flow that goes outside of the copyable area *absolute* jumps are required to ensure the code behaves the same once copied. Similarly, jumps within a copyable region must use relative addressing to guarantee a copy will behave in an equivalent fashion.

Our second phase thus includes a pass to convert control flow statements that go outside of a copyable area (and not to the *target block*) to use absolute addresses for their targets. There are two cases of such control flow: a *goto* and a function

4.3. Design

Original source code:

```
#pragma copyable end
    END_LCMP:
```

Is changed into:

```
__volatile__ __asm__ (""::"memory");
    END_LCMP:
__volatile__ __asm__ (""::"memory");
```

Figure 4.6: At an early compilation stage volatile statements are automatically inserted by the compiler around the *end label* to ensure that the *target* basic block will remain intact throughout optimizations.

call, both complicated by the fact that GCC itself does *not* produce the final binary code, rather it uses an external, platform-specific assembler program. It is in fact the assembler's role to choose the addressing mode for each call or jump; typically the shortest addressing mode to reach the target is chosen, but there is no general and relatively platform-agnostic way to specify in the assembler input that a jump or a call is to use absolute addressing. Below we describe how we ensure absolute jumps are used through the use of *computed gotos*, and then how we process the code chunk to ensure control flow is safe for copying.

To force selected jumps and calls to use absolute addressing we modify the code of these instructions to make jumps and calls via a register. As shown in Figure 4.7, in C these instructions are represented respectively by a *computed goto* and a function call using a *function pointer*. A *computed goto* is a special feature of the *labels-as-values* extension of GCC used by direct-threaded engine. It is a *goto* whose argument is not a label but a variable containing the address of a label (or any other address). Using a register to hold the destination address may have a negative impact on the performance that will vary from platform to platform, or even CPU type. Here the benefits of maintainability and safety are paramount, and as we will show in Section 4.4 our solution is efficient in practice. Nevertheless, more portable ways of expressing absolute addressing could slightly improve performance.

4.3. Design

Original code within a copyable area:

```
goto NullPointerException_label; /* label outside of the copyable area */
```

Is automatically replaced during early compilation stages with:

```
{
  void *address = &NullPointerException_label;

  /* this assembly prevents constant propagation */
  __asm__ __volatile__ (" : "=r" (address) : "0" (address) : "memory");

  goto *address; /* computed goto uses absolute addressing */
}
```

Figure 4.7: To ensure absolute addressing a *goto* to outside of a copyable area is replaced with a specially crafted *computed goto*.

Our current system assumes that code chunks are small enough that the compiler will use optimal, relative jumps within the code of instructions found in a region. While it does not attempt to ensure intra-area jumps are relative, an appropriate pass could easily be added. The reason this assumption is valid is that the relative jumps are *preferred* by GCC and external assembler whenever possible because of their smaller size and possibly faster execution. The only case when an absolute jump target addressing is used is when the target of a jump is beyond the scope of relative addressing, which on contemporary architectures is about +/- 32kB or more. Since the application of code copying only makes sense for small code chunks (less than 1kB, most of the time about 100B) relative addressing is the only one used, and thus this assumption is always valid.

Phase III: Mark and ensure areas are solid

Rather than modifying a large part of GCC to ensure the properties of copyable code regions are preserved at all subsequent compilation stages, by all compilation passes, we instead inserted two additional passes. The first pass modifies the code in a way that ensures a minimum of information about copyable code regions is always preserved. The second (reusable) pass uses this information and is capable of finding all the basic blocks belonging to copyable areas after arbitrary optimizations. Both passes include sanity checks mentioned earlier ensuring the additional information on code chunks is not lost or mangled.

After the source code is parsed into the stream of statements the compiler creates descriptions of basic blocks. Each such description contains pointers to the first and the last instruction that a basic block contains. We found that a basic block is a convenient unit to carry the additional information about the copyable code. It gives an easy access to smaller components of the code, like each particular instruction, while also being easily accessible via higher-level structures, e.g. the control flow graph. We extended the data structure describing a basic block to store the unique id of the copyable area a block belongs to and to store a field of utility flags. The initial marking of basic blocks is straightforward. We scan the stream of statements

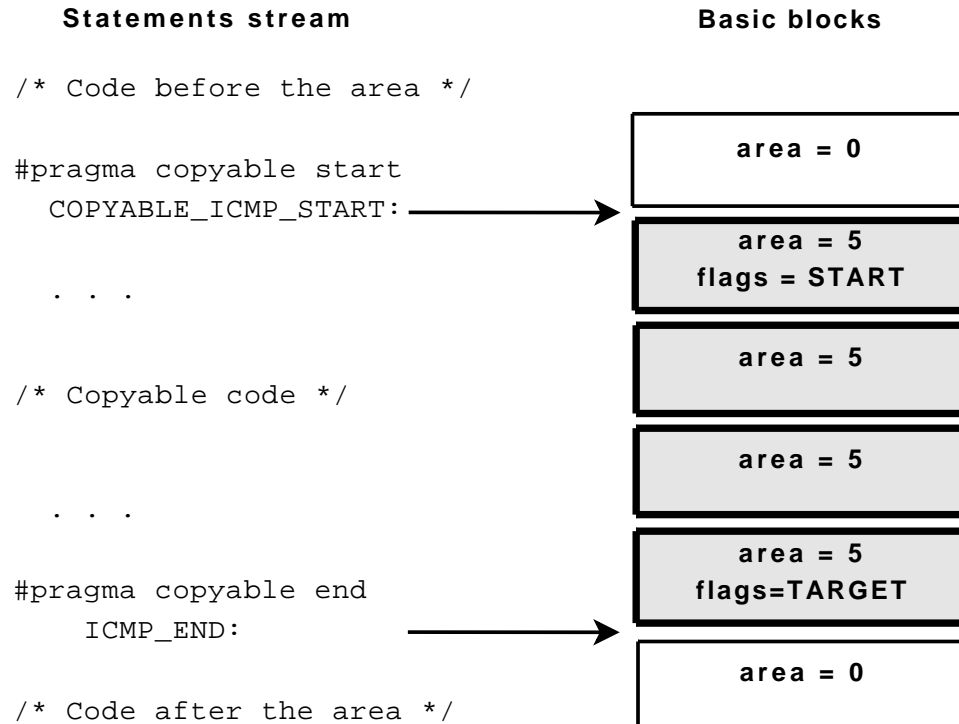


Figure 4.8: Initial marking of basic blocks right after parsing.

for labels earlier marked as *start* and *end*, and mark basic blocks located between pairs of such statements with corresponding flags, as shown in Figure 4.8.

In general, optimizations can create new basic blocks, move or split existing ones. One of the possible results is that some basic blocks that functionally are part of a copyable area might no longer be placed between the *start* and *target* basic blocks of this area and might not carry the initial marking. To recover marking after optimizations we rely on the preservation of the *start* and *target* blocks, which in turn is ensured with sanity checks. Area marking restoration can then be done through simple propagation along the control flow graph, from the *start* block of each area until the *target* block and jumps via computed goto's. It is critical that the compiler had earlier modified all the jumps to outside of copyable areas to use computed goto's. This way it is possible to always find the limits of copyable areas.

Importantly, our approach *does not use a heuristic* and is guaranteed to properly restore the list of blocks belonging to each copyable area. We still included extensive sanity checks that in practice should never be triggered. This is because, for instance, we earlier inserted volatile assembly around chunks end labels (see Figure 4.5) and disabled *cross-jump* optimization (see below). With these measures in place previously executed optimizations should not have inserted or deleted *start* or *target* blocks or cause the control flow graphs of different code chunks to interfere.

The one optimization that is nearly guaranteed to cause interference between control flow sub-graphs of different code chunks is, in GCC, called a *cross-jump*. It is currently the only optimization that has to be disabled for a function that contains copyable code. It attempts to find parts of code within a function that are identical and then share a single copy of the code among all the places in the function where this code is used, reducing overall code size. This optimization clearly conflicts with the need of the code-copying engine to use self-contained code chunks and has therefore always been useless in this context. Selective per-function disabling of this optimizations does not change the way all the rest of code of virtual machine is compiled.

Phase IV: Correct basic blocks ordering

The main reason for our basic block reordering pass is an optimization performed by GCC by default, *basic block partitioning*. This pass does two things. It divides the set of basic blocks of a function into those that are expected to be executed frequently (hot blocks) and those that are expected to be executed rarely (cold blocks). In the final assembly all the hot blocks of each function are located contiguously in the upper part of the code, and the cold blocks are located below the hot blocks. This optimization also reorders basic blocks to ensure that fall-thru edges are used for the most often encountered control flow. These are heuristic techniques for improving instruction cache hit rate and simplifying control flow, and this optimization can in practice improve the performance of a virtual machine by several percent, therefore we want to allow for it.

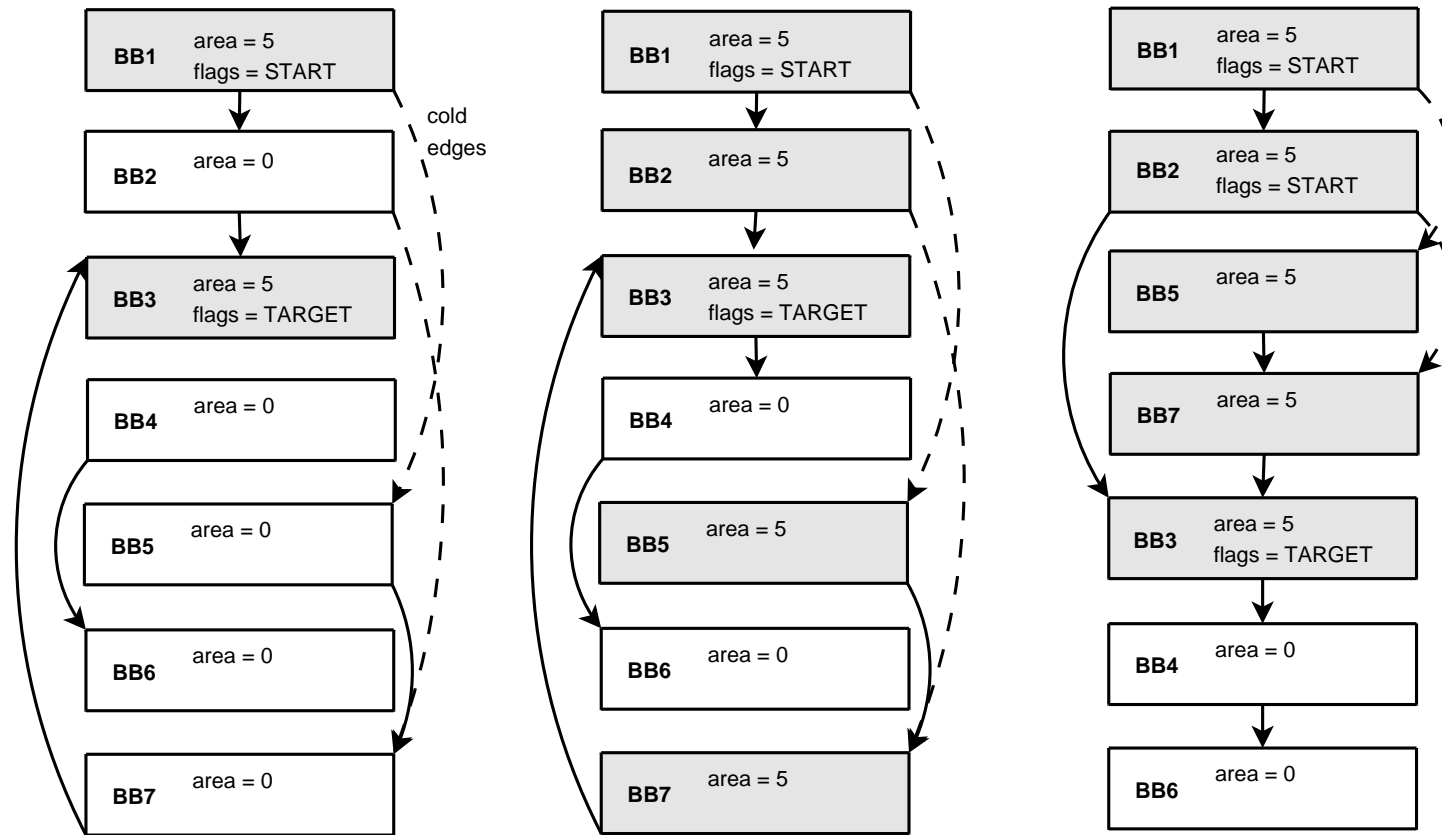


Figure 4.9: From the marking of only two basic blocks, *start* and *target*, the complete marking can be restored by following the edges of the control flow graph. Once the marking is restored it is possible to rearrange the basic blocks of a marked copyable area.

Alternatively we had an option to disable this optimization on a per-function basis. This was deemed unsatisfactory for two reasons. First, we perceive the fall-thru edges optimization as a welcome attempt to improve the quality of the resulting code later used for code-copying. Secondly, we have to be aware that there are other optimization passes that can also relocate basic blocks. Therefore with or without block partitioning we had to create a solution that would be able to deal with any kind of relocation of basic blocks.

For a chunk of code to be copyable the compiler has to restore the order of basic blocks so that the marked code is self-contained. In this case the goal is to move basic blocks to ensure that the *start* basic block of the copyable area is followed by all other blocks belonging to it, which are then followed by the *target* basic block of the same copyable area. After the marking of basic blocks belonging to all areas is restored (as described in the previous section) it is relatively easy to move all basic blocks belonging to an area into the wanted positions, as can be seen in Figure 4.9. Positions of basic blocks that are part of the control flow graph of a copyable area and are initially located between *start* and *target* blocks are left untouched. Basic blocks that are part of the control flow graph of a copyable area but are initially *not* in between *start* and *target* blocks are moved to immediately precede the *target* block. The *ordering* of these basic blocks is preserved. Positions of other basic blocks, not belonging to copyable areas, are left unchanged. This means that almost all optimizations of fall-through edges and others that reposition basic blocks are preserved which minimizes the negative effect this reorganization might have on performance. In fact, in some cases, this reorganization actually improves VM performance, regardless of the use of code-copying, as we will show in the case of OCaml in Chapter 5, Figure 5.10 on page 93.

Phase V and VI: RTL markers and final verification

The additional passes described above modify the structure of code based on up-to-date information about the boundaries of basic blocks, construction of the control flow graph, and other data. During the final compilation passes the GCC compiler

discards some of this information or does not keep it up to date. In our tests we found that these last optimization passes do not change the structure of the code enough to invalidate the properties of copyable code. Nonetheless, this was not sufficient for the safety guarantees we required and another solution was needed. We therefore added two simple passes.

During the compilation process, not long before the information about basic blocks and control flow graph becomes unavailable, an additional pass inserts into the program representation (*RTL* stream) special (untouchable by other passes) *notes* that mark the start and end of copyable areas, including the ID of an area. The *notes* is an existing GCC mechanism for RTL code annotations. The second pass is then a simple verification pass that uses only a minimum of information. It is executed just before the final assembly is sent to an external assembler. With the *notes* inserted by our previous pass it is possible to verify all the necessary properties of copyable areas when the code is final. The verification algorithm traverses the RTL instruction stream and ensures that:

- all copyable areas are present,
- copyable areas do not interleave with one another,
- jumps from a copyable area A to a symbol (i.o.w. to a label, thus assumed relative) are either to a target within A or to this area's target label, i.e. the label that begins the target basic block (the relative nature of the jump is trivially ensured, as explained in the description of *Phase II: Scan the tree*(2) on page 45),
- jumps to the outside of an area are made via register and not a symbol (thus are absolute),
- all calls from within areas are made via register and not a symbol (thus are absolute).

A verification error at this point is uncorrectable and is treated as an internal compiler error. This guarantees that if source code compiles properly then the copyable

4.4. Experimental Results

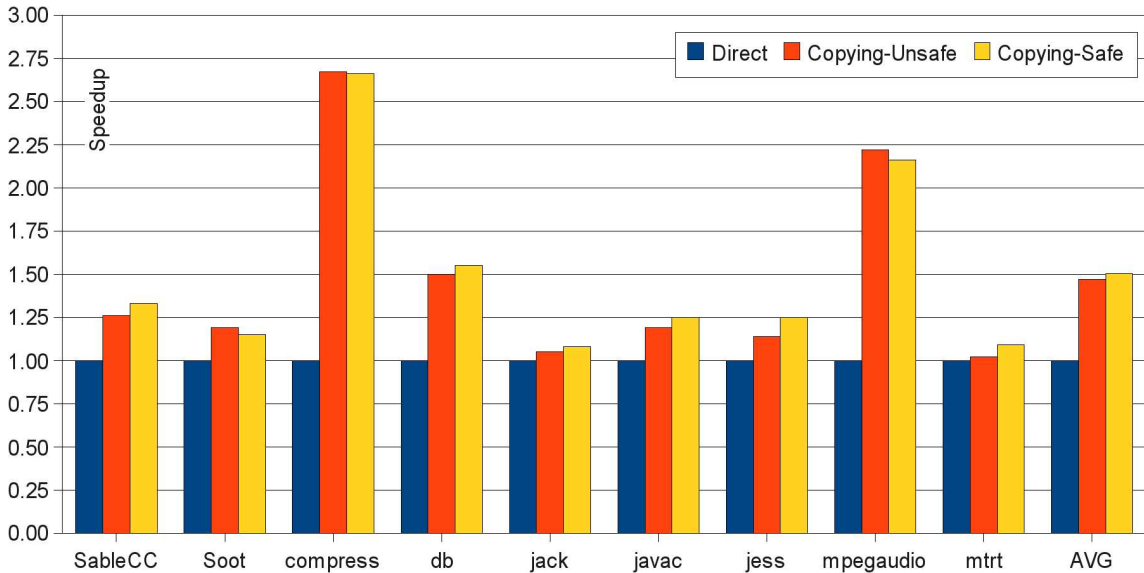


Figure 4.10: Performance comparison of SableVM with standard direct-threaded engine, unsafe code-copying engine and safe code-copying engine using the GCC copyable-code enhancement.

chunks of binary code will be safe to copy and execute in a code-copying VM. Sanity checks in all our passes ensure proper flow of the information on code chunks which allows the final verification to function reliably. In our experience we have not yet encountered a case where the verification pass would fail when all the former passes executed properly.

4.4 Experimental Results

To examine practicality of our design we modified a Java Virtual Machine, SableVM [Gag02], to use our enhanced GCC. In SableVM source we marked code chunks with our *copyable #pragma*. Code-copying was already supported in SableVM, but required globally disabling block reordering in GCC and did not provide safety guarantees. During preparations we used our enhanced GCC to verify the unsafe code

4.4. Experimental Results

Metric	#
Data structures modified	4
Fields added to data structures	6
Data structures added	3
Functions added to existing files	4
Function calls/hooks inserted	8
Code lines added or modified	139
Code lines in new files	1500

Figure 4.11: Metrics of code modified and added to GCC.

formerly used by SableVM’s code-copying engine and found several cases where execution of a less likely control flow path in a bytecode would result in a VM crash due to a function call using relative addressing.

The goal of our main experiments was thus to demonstrate that our new approach allows the code-copying strategy to be realistically and more reliably used while maintaining the performance. The results shown in Figure 4.10 have been gathered using machine *ia32* (machine specifications and benchmark descriptions can be found in Section 2.7, page 22). The SPEC and in-house benchmarks were executed with their default settings (`-S 100`), the resulting executions times were averaged over 10 runs, and performance is shown normalized to the speed of the direct-threaded engine as a baseline for comparison.

The benefits of code-copying are clear. We are able to achieve approximate parity with the unsafe code-copying approach. More surprising perhaps is that the performance of SableVM version 1.13 modified to use our GCC extensions actually improved over the manual code-copying design in most cases. We attribute the general improvement to the fact that previously SableVM had to globally disable basic block reordering for the code-copying engine to work at all. With the added GCC support for code-copying this useful optimization was enabled. We also note that the performance of two SPEC benchmarks that benefit the most from code-copying, as well

as *Soot* slightly decreased, about 2-3%. We suspect that this effect is caused by the memory barriers inserted into the code in places where the special *#pragma* is used. These barriers might be inhibiting some of the optimizations. It is also the case that inadvertent changes to instruction cache behavior cause significant performance variations (as much as 10% [GVG06,GVG04]). More detailed analysis of performance gains and losses is thus warranted, but certainly the magnitude of correlation in Figure 4.10 is sufficient to demonstrate the general success of our compiler-facilitated approach. Overall, the effect is clear: our modifications efficiently enable code-copying as a safe technique for VM interpreter design.

One of our goals was to minimize the impact of our changes to GCC on GCC maintenance. Figure 4.11 shows the results of our impact measurements in terms of required changes to code and data structures. In a truly large project such as GCC we see these numbers as indicators that our extension has minimal impact on the existing GCC code and its maintenance. A major upgrade of our enhanced GCC, porting our modifications from GCC v. 3.4 to v. 4.2 (about 2 years of GCC development) took only a few hours and consisted mostly of renaming via search/replace to account for changed names of fields in GCC data structures and testing. We believe this validates our claim that a relatively simple compiler modification can help improve the performance of dynamic execution environments.

4.5 Notes on Related Work

The first uses of code-copying technique by Rossi and Sivalingam [RS96] and soon after by Piumarta and Riccardi [PR98] did not, in practice, have to face the issues resulting from the use of highly-optimizing compilers. Compilers at that time were vastly less aggressive than today and there existed a much closer relation between the source code and the resulting binary code. Most of the works that used code-copying also silently ignored the issue, with the exception of Ertl's work [EG04] on retargettable JIT using code generated by C compiler (building on code-copying) which faced similar issues. Ertl's solution did include automated tests to detect code

chunks that were definitely not copyable, but it was not guaranteed to find all such chunks (see Figure 6 in [PGA07] for an example) and thus did not ensure safety. In Gagnon’s work within SableVM [Gag02] by-hand and trial-and-error methods were used for detecting non-copyable code. As a later improvement to SableVM an actual database of copyable bytecodes per-architecture, and per-compiler (and its version) was added. Prokopski et al. [PGA07] described a specialized bytecode testing suite to speedup the process of constructing this database for new architectures or compiler versions. Despite the extensiveness of the test suite its use still did not and could not provide proper execution safety guarantees.

4.6 Conclusions

For a variety of reasons, including simplicity and dynamic support, many modern languages are based on virtual machine (VM) designs. Efficiency and ease of design are key features for rapidly evolving languages and associated execution environments.

Code-copying interpreters offer a good trade-off between performance and maintenance, but were previously limited by the lack of critical safety guarantees, as well as maintenance concerns with respect to the modifications to the static compiler. Copyable code must behave functionally the same when copied, and while conceptually trivial these guarantees are simply not provided by current compilers or C language extensions.

With our work we demonstrate that it is possible to make code-copying safe and practical. Our approach to GCC modifications demonstrates viability of our technique for ensuring the safety properties essential to code-copying. We show how this technique can be relatively easily integrated with a modern C compiler, while keeping the changes relatively isolated and making only limited assumptions about the inner workings of a compiler, thus ensuring long-term maintainability.

The implementation of a code-copying GCC extension on which we based the work presented in this chapter was focused on supporting the i386 architecture. On

4.6. Conclusions

other architectures there might be additional issues with delay slots (e.g. MIPS), relative addressing of externs and globals (e.g. x86_64), or relative-jump span limitations (e.g. PowerPC). Addressing these hardware architecture-specific issues a deeper performance analysis, further determining the source of our gains in using code-copying applied to other VM architectures, are the core of our work presented in the next chapter.

Chapter 5

Application to Multiple Virtual Machines, Multiple Architectures

In the previous chapter we presented a C compiler enhancement implemented within GNU C Compiler (GCC) that provided the support necessary for safe execution of virtual machines that use the code-copying technique. This enhancement allows a programmer to use a simple C *pragma* to mark chunks of VM source code that will be used at runtime for code-copying. The compiler then ensures that the code chunks generated from these parts of the source code are contiguous in memory and can be safely copied at runtime, while also ensuring as many code optimizations as possible are still applied. This approach makes code-copying practically usable, and was essential in allowing us to scale our development and investigation further.

Due to varying language features and virtual machine design, however, not all languages benefit from code-copying to the same extent. We consider here properties of interpreted languages, and in particular bytecode and virtual machine construction that enhance or reduce the impact of code-copying. We implemented code-copying and compared performance with the original direct-threading virtual machines for three languages, Java (SableVM), OCaml, and Ruby (Yarv), examining performance on three different architectures, ia32 (Pentium 4), x86_64 (AMD64) and PowerPC (G5). Best speedups are achieved on ia32 by OCaml (maximum 4.88 times, 2.81 times on average), where a small and simple bytecode design facilitates improvements to

branch prediction brought by code-copying. Yarv only slightly improves over direct-threading; large working sizes of bytecodes, and a relatively small fraction of time spent in the actual interpreter loop both limit the application of code-copying and its overall net effect. We are able to show that simple ahead of time analysis of VM and execution properties can help determine the suitability of code-copying for a particular VM before an implementation of code-copying is even attempted.

Below we first discuss the application of code-copying to 3 different VMs, then we discuss properties of different VMs and languages backed by several metrics. Later we present and discuss the experimental results, comment on related works, and finally draw conclusions.

5.1 Application to Java, OCaml and Ruby

Our experimentation is based on examining code-copying in several environments. SableVM already supported code-copying (in fact all three interpreter designs [Gag02]). Our modifications to SableVM were thus mainly to modify the VM to use the *gcc* enhancements described previously, obviating the existing system for verifying correctness of copied code.

OCaml and Yarv (Ruby)¹ use only direct-threading, and so changes were more extensive. For both OCaml and Yarv VMs we used the same general scheme for the code-copying implementations. At a high level this involved:

- a) adding a small set of functions and data structures supporting the management of superinstructions,
- b) modifying the interpreter loop, either through C macros (OCaml) or changes to the C code generator (Yarv) to use the special *pragmas* to identify the copyable code regions implementing individual bytecodes, and

¹The current version of the popular Ruby interpreter used, at the time of this writing, a different interpretation technique, not readily compatible with code-copying. It is also the case that Yarv contains analyzes that can potentially simplify bytecode and improve opportunities for code-copying.

- c) modifying the function responsible for bytecode preparation to construct and use copied code, storing the revised code in a modified code array.

To identify copyable sequences several language-specific bytecode analysis are required, making changes to bytecode preparation the most complicated step. For example, before attempting the creation of copied code all bytecodes changing control flow must be found and their jump targets identified. In the case of SableVM other analyzes are necessary for detecting potential class loading, identifying GC points, and other language-specific concerns.

5.2 Execution Properties

Basic properties essential to good code-copying performance can be gathered ahead of time, through simple dynamic metrics and examination of the virtual machine bytecode design. This allows an assessment of the suitability of a virtual machine for code-copying prior to implementation; our results here can also serve as a heuristic guide during bytecode design for maximizing the performance of code-copying.

Performance and behavior of three different virtual machines, on three hardware platforms are examined. These cases illustrate a spectrum of implementations and potential performance. Each of a Java virtual machine, an OCaml interpreter, and a Ruby interpreter are examined running on 32-bit Intel (*ia32*), 64-bit AMD64 (*x86_64*), and 64-bit PowerPC (*ppc*) hardware. Each virtual machine was tested on each architecture in two configurations: one using the standard direct-threading technique and one using our safe code-copying technique. For each VM we selected a set of benchmarks that would run properly across all architectures. We measured the behavior of each benchmark in 3 ways: its performance, its interaction with the hardware as indicated by hardware counters and dynamic characteristics of creation and usage of code created using code-copying technique.

5.2.1 Architectures and Virtual Machines

Performance of any program is obviously closely tied to the machine architecture on which it runs. Variability due to different hardware designs can be magnified when investigating the performance of a code-copying virtual machine—the benefit of code-copying is largely produced by better enabling branch prediction, improving code locality, etc. These are factors that can vary significantly on different computer hardware. Our analysis thus compares performance on three popular architectures: Intel 32-bit *ia32*, 64-bit *x86_64*, and 64-bit PowerPC *ppc* (the exact descriptions of machines can be found in Section 2.7).

External performance influences were eliminated or reduced as much as possible. On all machines all used software was installed on a local hard drive, and unnecessary processes and network activity were stopped. To reduce noise due to multiprocessing on *ia32* and *x86_64* we ensured through the *cpu affinity* functionality in `glibc` and the Linux kernel that all VM threads execute on a single CPU only. Mac OS X, however (*ppc* machine), intentionally does not provide such functionality, leaving the decision of CPU assignment to the OS.

For each virtual machine and on each architecture we examine performance for a variety of language-specific benchmarks, as described in Section 2.7

All of these virtual machines are (primarily) implemented in C so the GCC enhancement presented previously was directly applicable. To measure the performance we timed 7 runs of each benchmark and averaged the results. We computed standard deviation, and for almost all cases it was less than 0.01 of the average measured value, and never greater than 0.04.

5.2.2 VM and Language Characteristics

The different bytecode sets offered by different virtual machines can have a large impact on code-copying performance. More complex bytecode instructions perform more “work” per bytecode, while smaller bytecodes will tend to have more relative branching overhead, and thus greater opportunity for improvement through code-copying. The extent to which copyable sequences can be exploited is also driven

5.2. Execution Properties

	interpreter	instructions	LOC per	copyable instructions	
	loop kLOC	total	instruction	#	%
SableVM	(15.0) 3.4	319	(57) 25	194	61%
OCaml	1.0	133	8	96	72%
Yarv	1.2	121	10	(71) 61	(59%) 50%

Table 5.1: Average size of an instruction (in lines of code) and fraction of copyable instructions in the 3 considered VMs. For SableVM the number in parenthesis is the number of lines after m4 macros expansion of its interpreter loop source code. Also, for the SableVM interpreter loop LOC count we did not include instruction initialization code which, in this particular VM, happens to be interleaved with instruction code. For Yarv the best performance was obtained when the maximum size of copyable instructions was limited to 100-150 bytes, lowering the number of copyable instructions from potential 71 to 61 actually used.

by characteristics of VM workload. The programs which exercise copyable sequences more often will naturally see greater improvements. This is in relation to non-copyable sequences, but in order to account for whole program improvement also in relation to the fraction of whole program time spent in the *interpreter loop*. If execution depends on expensive library or runtime services then again the relative improvement provided by code-copying will be reduced. Below we discuss these concerns and present dynamic data from our three virtual machines.

The relative complexity of operations executed by a bytecode is a measure of how much work (in terms of CPU time) each bytecode does, including work done by functions called from within a bytecode. A reasonable expectation is that the more work an average bytecode does the less positive the effect of using the code-copying technique. As previously mentioned, much of the improvement shown by code-copying is due to reduced overhead in jumping between bytecodes, removing jumps and simplifying branch prediction. Complex bytecodes mean this optimization operates on a smaller runtime overhead per bytecode, and so has less overall impact.

The presence of consecutive groups of simple or small bytecodes (doing little

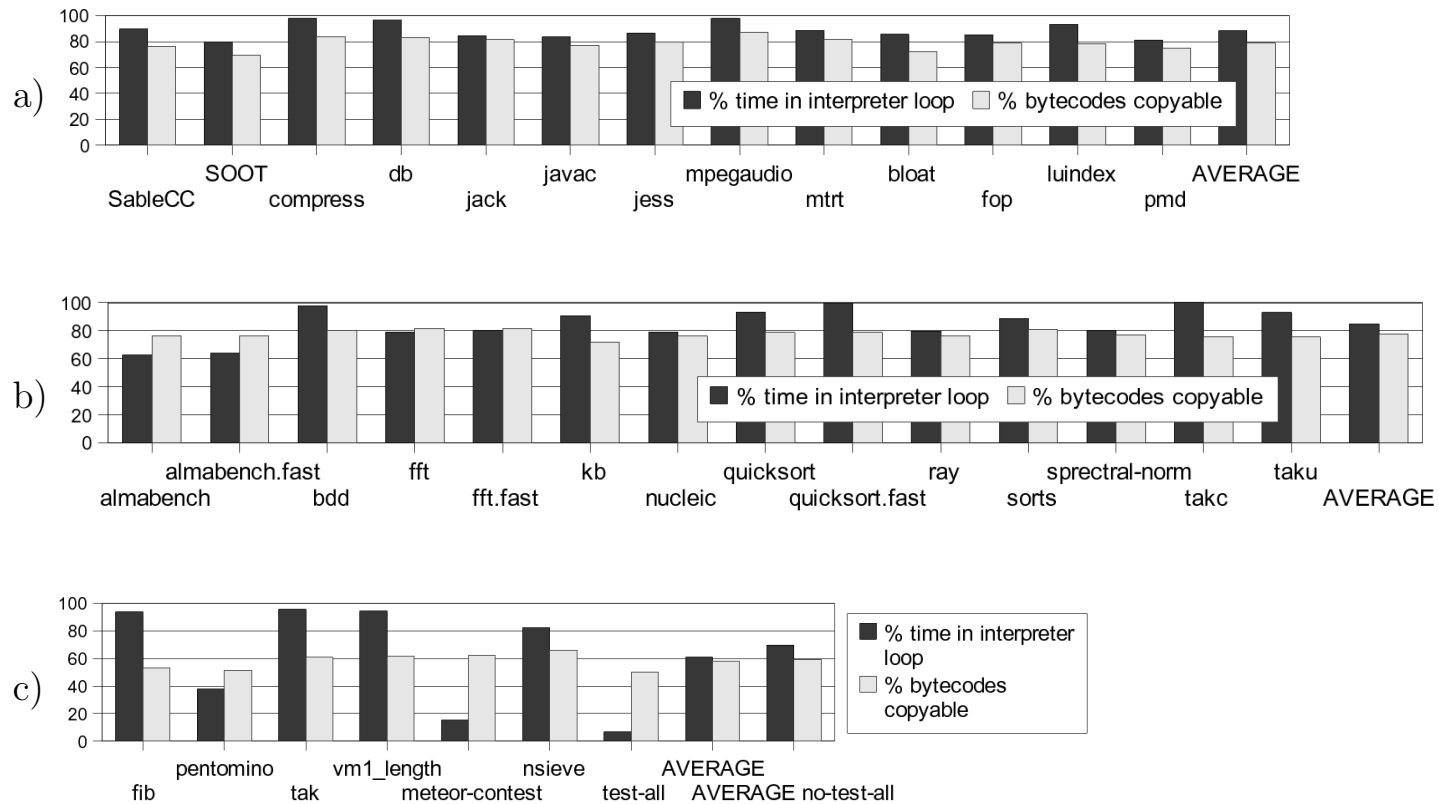


Figure 5.1: Percentage of time spent in the interpreter loop of the direct-threaded interpreters for x86_64, and of bytecodes loaded that were potentially copyable, for a) SableVM, b) OCaml, and c) Yarv.

“work”) not containing control flow branches or targets (from the bytecode point of view, not the internal implementation) is also an important factor. Such consecutive groups are candidates to become superinstructions during runtime, and so a greater percentage of execution spent in such groupings implies greater benefit from code-copying. Arithmetic operations tend to be expressed with small bytecodes, and so programs including significant amounts of direct computation, not too densely interleaved with control flow changes, will show a greater benefit. Note that in our implementation we currently require that the copied code of a superinstruction contains at most one control-flow changing bytecode, and it must be the last one in a superinstruction. This is not, however, an inherent limitation of code-copying. If all bytecode instructions were copyable it would equate a superinstruction span to a basic block but since not all bytecode instructions are copyable superinstructions are usually smaller.

Static measurements for our three VMs are shown in Table 5.1: the overall size of each actual interpreter loop, the number of distinct bytecodes, the size of instructions, and the relative percentage of the number of copyable bytecode instructions. Copyable instructions were manually identified for this data. Lines-of-code is of course a very approximate measure, not taking into account usage of C macros or inlined functions. SableVM for example does not use C macros, but does generate C code using a preprocessor (`m4`); the number of lines is thus higher than in other interpreters even for similar bytecodes. Coupled with the complexity of any internal functions called by bytecode implementations this, as we will show later in comparison to dynamic results, makes the average static size of bytecodes a fairly *poor* indicator of runtime complexity. Relative number of copyable bytecodes is more useful; while it is also a static indication of performance, suggesting the proportion of execution time that may benefit from code-copying, it reflects runtime performance quite well.

Dynamic results are shown in Figure 5.1. The actual execution time is measured, and the relative amount of runtime spent in the interpreter loop is shown, along with the relative number of bytecodes loaded at runtime that were statically identified as copyable. This data is from the x86_64 architecture, but other machines demonstrate comparable results. It is important to note that these statistics can be easily gathered

early in the VM design process, and importantly *before* any effort is invested in implementing code-copying.

5.2.3 Analysis

Analysis of these simple measurements already provides strong indicators for the potential performance of code-copying. Here we separately consider first Java, then OCaml, and finally Ruby. Java is an example of a good environment for code-copying; OCaml serves as an example where code-copying excels, and Ruby of an environment in which code-copying will not tend to provide significant improvements.

Analyzing SableVM's execution behavior is made more complex due to the use of some internal code optimizations. Java, for instance, requires dynamic loading and linking of classes at first runtime reference, and so many otherwise simple bytecodes include more complex implementation behavior to handle this special case. SableVM (and other Java VMs) optimizes this by splitting such instructions into two versions, a slower version to handle initialization, and a faster version for repeated execution. Let us take an example of `PUTFIELD` bytecode. The purpose of this bytecode is to set the value of a field in an object. The opcode of this bytecode is followed by a 16-bit parameter that is an index into constant pool. The element in the constant pool is a symbolic reference to a field in a class. If one were to always assume the pessimistic case and follow the chain of informations until the memory address (offset) of an actual field is found the interpretation process would be incredibly slow. Instead, it is on the first execution of a specific occurrence of `PUTFIELD` bytecode that the pessimistic case is followed. This might result in class loading and static initialization. At the least it is necessary to calculate the address (offset) of the wanted field within a class of objects. Once a specific `PUTFIELD` occurrence is executed, however, class loading will not be required, and once the relative address is calculated it can be cached and used by subsequent executions of this occurrence of `PUTFIELD`. Other optimizations include using specialized bytecodes in place of instructions that operate on generic classes of primitives (short, byte, int, etc). For a more complete explanation of the design see Gagnon [Gag02].

Figure 5.1 shows that SableVM spends almost 90% of execution time executing code within the interpreter loop (not counting code executed in functions called from the loop), and that almost 80% of dynamically loaded bytecodes are potentially copyable. With respect of overall execution time, there is certainly significant room for code-copying to achieve very good performance. From Table 5.1, however, the average size of an instruction is relatively large (47 lines). As mentioned, SableVM code, unlike the other two VMs, has expanded macros and moreover many very complex operations related to class loading are implemented directly in the interpreter loop source. The impact of the apparent complexity of bytecodes here can be further discounted by comparing static and dynamic results. 61% of bytecode instructions were statically found to be copyable in the interpreter loop, whereas almost 80% of bytecodes loaded at runtime were copyable. Larger, non-copyable bytecodes seem to be found at execution time less often than the shorter, copyable bytecodes. These characteristics suggest a great positive impact of code-copying application and as previous works have already noted (e.g. Gagnon [Gag02]) Java bytecode is in fact a great candidate for code-copying, even in the presence of many bytecodes executing complex operations.

OCaml bytecode in many ways resembles that of Java, without much of the complexity brought by Java's Object model, VM and class library interface, and other factors. From Table 5.1 a quick comparison of the source code of SableVM and OCaml reveals that implementations of OCaml instructions are several times smaller than Java instructions (in SableVM), only about 8 lines of code. The fraction of bytecodes defined in interpreter loop that are copyable is actually higher than SableVM (72% vs. 61%) while the copyable bytecodes constitute a similar ratio, almost 80% of the loaded bytecodes (Figure 5.1). In this case bytecodes are more evenly sized, and so dynamic measurement matches static better. The OCaml interpreter spends about 85% of execution time in the interpreter loop, also giving it a vast space for improvement. With similar overall runtime characteristics, and less complex code size (even accounting for lack of macro expansion) these initial measurements suggest that the OCaml interpreter may be an even better candidate for code-copying than SableVM.

The size of the Ruby interpreter loop and the average size of an instruction in

terms of lines of code are comparable to that of OCaml (Table 5.1). Still, overall Yarv has the overall weakest positive indicators. Although 59% of bytecodes are statically copyable, later implementation results showed that applying code-copying to larger bytecodes (more than 100–150 bytes; values in this range produced similar results) reduced performance, and a threshold of 50% of bytecodes provided the best performance (this lower setting is used for actual code-copying experiments later in this chapter). Unlike SableVM or OCaml the time spent in the interpreter loop of Yarv varied greatly between different benchmarks; Figure 5.1 shows an average of slightly less than 70%, but ranging between 7% and 96%. The number of potentially copyable bytecodes loaded was also much lower, only about 60%, compared to almost 80% in case of SableVM and OCaml. An inspection of the source code for bytecode instructions further reveals that many of the bytecodes that are small in terms of code size use internal VM function calls, making the bytecodes fairly “heavy” at runtime, and heuristically limiting the impact of code-copying. These properties suggest that code-copying may not bring nearly as much general performance improvement to Yarv as to the other VMs.

Although coarse, for each of the three VMs these observations are sufficient to at least give a relative ranking of expected performance benefit. OCaml has almost all positive attributes, while SableVM suffers from more complex instructions, and Yarv much less runtime opportunity for its even smaller set of copyable instructions. This gives three widely spread data points for examining code-copying performance. By considering VMs serving as both positive and negative examples we hope to validate our ahead of time analysis, and to allow future researchers and practitioners to understand where opportunities and pitfalls lie in terms of implementing or optimizing a code-copying VM. In the next section we examine actual performance of code-copying implementations of all of these VMs on different architectures.

5.3 Code-Copying Results

Deeper analysis of code-copying is performed by making detailed measurements of code-copying versions of all three VMs on our different architectures. These implementations all make use of the basic GCC support for code-copying described in Chapter 4. Results from analyzing these implementations demonstrate the value of code-copying in terms of speedup, as well as providing good evidence of how and why code-copying achieves the performance it does, given different VM and architectural characteristics. In our wide performance testing across 3 architectures and 3 virtual machines we used a variety of benchmarks, as described previously. We have made use of hardware counter information for detailed and low-overhead profiling, although this necessarily exposes machine differences and profiling limitations.

The presentation and analysis of results are organized as follows. We first discuss our low-level profiling strategy, along with some of its attendant complexities followed by the most general overall performance results summary in Table 5.2. We then present detailed performance results for each VM in Figure 5.2 with the absolute runtimes available in Tables 5.7, 5.8, and 5.9. An overview of branch misprediction rates, which we found to be tightly bound to the performance results, is reported in Tables 5.6 and 5.10. This is followed by a closer examination of the best and worst performing benchmark for each VM; precise identification of hardware and VM features that impact performance helps understand performance and orient future optimization and design. In the end we present the results of dynamic metrics, particularly dynamic superinstruction lengths in Figure 5.3 followed by detailed results of multiple metrics for all VMs and benchmarks in Tables 5.3, 5.4, and 5.5. From this we are able to show that the basic ahead of time analysis predictions are validated, and that several potential contributors to code-copying performance do indeed reflect runtime performance.

We are looking for answers to the following question: what are the features of a language, its bytecode construction and thus its virtual machine construction that improve the overall runtime performance, in particular when using the code-copying technique.

5.3.1 Profiling

To assess the interaction of the benchmarked VMs with hardware architectures we used hardware performance counters to do online profiling. On the *ia32* and *x86_64* machines we used OProfile [OPr] versions 1.9.2 and 1.9.3, respectively. On PowerPC 970 (G5) we used the standard OS X tool Shark 4.5. The goal was to measure the following values for the interpreter loop and copied code:

- CPU time (copied code measured separately),
- number of branches encountered (conditional branches only, whenever possible),
- number of branches mispredicted (or pipeline stalls and other negative events due to branch mispredictions),
- cache misses (treating L2 and L1 separately whenever possible).

Acquiring counter data is straightforward in concept, but poses a number of technical challenges. For instance, because of hardware limitations it is not possible to gather all counter data in a single program execution; only a limited set (and certain combinations) of counters can be used at any one time. We therefore ran each benchmark once as a warm-up with hardware counters disabled, then ran the same benchmark several times with different configuration of hardware counters until all desired data was collected. On *ia32* and *x86_64* machines we collected 7 sets of results for each benchmark. *Ppc* machine results are more limited. Available tools permit the measurement of only one hardware counter at a time (there can only be one counter set as *Trigger* in Shark) and data must be manually processed for the accurate event counts (rather than relative percentages) necessary to evaluate code-copying behavior. On PowerPC 970 (G5) we therefore limited the number of collected results to 3 sets per benchmark for 4 benchmarks per virtual machine (with the exception of Yarv where we measured 2 benchmarks only).

Variability in hardware counter behavior was also assessed. For each VM we chose 2 benchmarks (the slowest and fastest) and collected data for each counter 7 times (with the exception of PowerPC where due to the labor-intensive process of gathering

5.3. Code-Copying Results

counter data we limited the number of trials to 3 per counter). With the exception of Yarv, to further ensure any trends visible in the results are actually related to the observed performance we selected 2 more benchmarks, the second slowest and second fastest. For these benchmarks we collected data 3 times (once on PowerPC) for each hardware counter. Primary and secondary tests here correlate well, indicating reasonably stable behavior.

Processing the results collected on Pentium 4 and AMD64 we discarded the highest and the lowest count out of 7 runs then averaged the remaining 5 and computed their standard deviation. To decide on the level of trust for each result we used the standard deviation divided by the average of the values on which it was computed. This allowed us to easily assess the accuracy of hundreds of results². The values of this metric are low for the majority of counter measurements. On *ia32* and *ppc* our deviation metric had values below 0.1 and on *x86_64* below 0.2 in most cases. Generally higher values existed, as expected, for the more noisy cache-related counters. In all cases our observations and relative judgements are based on data changes dramatically larger than variance. Full data is available in the raw data results [Raw], not included here for space reasons.

On *ia32* machine we collected hardware counter results in two runs with the following settings:

- First run:
 - L2 cache misses: BSQ_CACHE_REFERENCE events with mask 0x700, trigger count 6000.
 - Branches: BRANCH_RETIRED events with mask 0xc, trigger count 6000.
 - Mispredicted branches: RETIRED_MISPRED_BRANCH_TYPE events with mask 0x1f, trigger count 6000.

- Second run:
 - CPU time: GLOBAL_POWER_EVENTS events, trigger count 100000.

²For example, for a series of values 3, 5, 7 this deviation metric is the same as for 30, 50, 70.

5.3. Code-Copying Results

On *x86_64* machine we collected hardware counter results in two runs with the following settings:

- First run:
 - L1 instruction cache misses: `INSTRUCTION_CACHE_MISSES`, trigger count 600.
 - L1 data cache misses: `DATA_CACHE_MISSES`, trigger count 600.
 - Total no. of branches: `RETIRED_BRANCH_INSTRUCTIONS`, trigger count 600.
 - Mispredicted branches: `RETIRED_MISPREDICTED_BRANCH_INSTRUCTIONS`, trigger count 600.

- Second run:
 - L2 cache misses: `L2_CACHE_MISS` events with mask `0x7`, trigger count 600.
 - Instruction fetch stalls: `INSTRUCTION_FETCH_STALL` events, trigger count 600.
 - Dispatch stall because of branch abort: `DISPATCH_STALL_FOR_BRANCH_ABORT`, trigger count 600.
 - CPU time: `CPU_CLK_UNHALTED`, trigger count 6000.

On *ppc* machine we collected each hardware counter result in a separate run:

- Flush caused by branch misprediction (presented as *branches mispredicted* in the next section): trigger count 6000.
- Total number of branches: trigger count 60000.
- CPU cycles: trigger count 600000.
- L2 data cache misses: trigger count 600.

5.3. Code-Copying Results

- L2 instruction cache misses: trigger count 600.

The following hardware counter results were also collected on PowerPC but turned insignificant or unrelated results (as can be seen in the raw data results [Raw]):

- Branch mispredictions because of: target address misprediction or valid instructions available but *Instruction Fetch Unit* held by *Branch Information Queue* or *Instruction Decode Unit*. Trigger count 6000, raw data name *BranchMispredTargetPlus*.
- Branch misprediction because of: *Condition Register* value. Trigger count: 600, raw data name *BranchMispredCR*.
- Branch misprediction because of: target address prediction events. Trigger count: 600, raw data name *BranchMispredTarget*.

On all architectures to ensure the best possible view of hardware behavior we used higher trigger counts for events that occur more frequently (like CPU cycles) and lower trigger counts for rare events (like cache misses). This is also subject to software and hardware constraints; e.g., we were not able to use trigger counters below 6000 on the Pentium 4. To obtain more comparable results for counters collected at different rates we rescaled the final values as if all hardware counters were set to trigger events at the same counter value, 10000.

Code-copying is primarily an optimization of the interpreter loop, and so in most data gathering we focus on the execution of the loop (and copied) code. Our measurements are narrowed to only those events that were registered while executing inside of the interpreter function (static code space) or in the copied code. Events registered in functions called from the interpreter function were not taken into account. In particular, in the current section unless otherwise specified *total time* values are only measured for the time spent in the interpreter loop, be it of a direct-threaded interpreter or code-copying one. *Time in copied code* values tell how much of the total time spent in interpreter loop was spent in copied code. Other hardware counter values related to branches and cache behavior were also analyzed *only* for the code executed inside of interpreter loop.

5.3. Code-Copying Results

	SableVM	OCaml	Yarv
ia32	1.44	2.81	1.14
x86_64	1.32	1.80	1.06
ppc	1.05	1.52	1.03

Table 5.2: Average speedups of total execution time of code-copying over direct-threading measured across virtual machines and architectures running all used benchmarks. Note this table presents the overall VM performance, as opposed to other results in this chapter that measure VM behavior only within the interpreter function.

5.3.2 Dynamic behavior of copied code

We have gathered both overall and detailed performance and other profiling data from code-copying implementations of all three VMs. Overall speedup is summarized in Table 5.2, showing the ratio of direct-threaded execution time to code-copying time. More detailed examination is done through a variety of metrics, presented in Tables 5.3, 5.4, and 5.5 on pages 75, 76, and 77.

The speedups in Table 5.2 show that across architectures code-copying works best on *ia32*, with high average speedups on our selected benchmarks of 1.44, 2.81, and 1.14 for SableVM, OCaml, and Yarv respectively. On *x86_64* code-copying also achieves high speedups of 1.32, 1.80 on SableVM and OCaml, and more marginal improvement with Yarv. The PowerPC implementation of code-copying is able to reach a high speedup of 1.52 only by the OCaml interpreter, with only small improvement for SableVM and Yarv. As our ahead of time analysis predicted, across virtual machines the best performance was observed on OCaml (reaching a maximum speedup of 4.88 on the *ia32* execution of the *quicksort.fast* benchmark), with good but less-improved performance on SableVM, and minimal overall improvement to Yarv.

To gain a better insight into actual dynamic usage of copied code we gathered further profiling data pertaining to bytecode and superinstruction creation and execution and translated them into several dynamic metrics. Tables 5.3, 5.4, and 5.5 summarize the following measurements for our three virtual machines. Note that we only present these metrics on *x86_64* architecture, as they are similar between

5.3. Code-Copying Results

architectures with appropriate scaling.

- Average bytecodes executed per dispatch. Calculated as the number of executed bytecodes divided by the number of dispatches (jumps) this is the length of executed superinstructions in bytecodes weighted by relative number of executions of each sequence. More directly this metric allows us to see how many dispatches are removed by code-copying and thus the length of an average executed instruction.
- Copied code memory usage. This is the amount of memory allocated and used to store copied code (superinstructions). Excessive memory usage can contribute to reduced performance, and so it is important to know the extent of code-copying memory requirements.
- Unique superinstructions. The number of superinstructions created during a VM execution measures the variety of execution, giving indications of memory requirements and potential overhead.
- Copied code memory divided by number of unique superinstructions. This is the average amount of memory used by a superinstruction; another indicator of size and resource requirements.
- Memory occupied by 90% of used copied code instructions. Heuristically, the majority of instructions will come from a small set of potential instructions. We counted the total number of executions of both simple and superinstructions, then found the subset of instructions that make up 90% of executed instructions. Out of those 90% we summed the memory used only by superinstructions (created using code-copying).
- Memory occupied by 90% of used copied code instructions divided by copied code memory usage. This is the percentage of total memory used for copied code that accounts for 90% of executed code.

5.3. Code-Copying Results

	Avg. bytecodes per dispatch [bc]	Copied code memory [kB]	Unique superinstructions	Copied c. mem./ unique sup. [B]	90% copied code mem. [kB]	90% copied c.mem./c.mem. [%]
SableCC	2.13	505	1972	262	16	3.16
Soot	2.04	975	3698	270	17	1.69
compress	4.70	294	1232	244	10	3.48
db	2.66	290	1307	227	4	1.21
jack	2.10	358	1562	235	17	4.76
javac	2.40	590	2607	232	18	2.94
jess	1.99	398	1744	233	8	2.08
mpeg.	7.96	750	1557	493	23	2.99
mtrt	1.74	359	1591	231	5	1.25
bloat	2.20	1227	4065	309	4	0.32
fop	2.35	1891	4050	478	16	0.84
luindex	2.98	961	3366	292	22	2.27
pmd	1.86	1195	3862	3175	10	0.80
Average	2.84	719	2421	288	13	2.30

Table 5.3: Dynamic bytecode execution metrics for SableVM (Java) on x86_64.

Per-benchmark performance data is further presented in Figure 5.2. Branch prediction behavior is known to be critical to code-copying performance [EG03c,ETK06], and so we present data in Tables 5.6 and 5.10. Another important property for both

5.3. Code-Copying Results

	Avg. bytecodes per dispatch [bc]	Copied code memory [kB]	Unique superinstructions	Copied c. mem. / unique sup. [B]	90% copied code mem. [kB]	90% copied c.mem./c.c.mem. [%]
almabench	7.68	114	675	172	23	20.07
almabench.fast	7.68	111	675	169	21	18.56
bdd	3.24	31	242	131	3	10.69
fft	17.34	32	171	189	9	27.02
fft.fast	17.34	32	171	189	9	27.02
kb	2.03	41	425	98	1	3.00
nucleic	2.14	113	734	157	2	2.06
quicksort	3.00	19	152	131	1	4.90
quicksort.fast	2.88	18	159	113	1	5.15
ray	2.34	93	710	134	4	4.51
sorts	3.30	214	1478	148	5	2.30
norm	5.12	82	635	132	1	1.54
Average	6.17	75	519	145	7	9.70

Table 5.4: Dynamic bytecode execution metrics for OCaml on x86_64.

understanding performance and for considering future, further optimizations is the length of superinstructions; this is shown in Figure 5.3. For the best and worst performing benchmark in each VM we give further hardware counter data, including cache behavior. This is shown for SableVM in Figures 5.4 and 5.5, for OCaml in

5.3. Code-Copying Results

	Dynamic avg. sup.ins. len. [bc]	Copied code memory [kB]	Unique superinstructions	Copied c. mem. / unique sup. [B]	90% copied code mem. [kB]	90% copied c.mem./c.c.mem. [%]
fib	1.37	0.42	16	27	0.2	46.15
pentomino	1.22	4.98	72	71	0.4	8.39
tak	1.44	0.62	16	40	0.1	15.65
meteor-contest	1.21	14.33	145	101	0.3	2.11
nsieve	1.53	2.18	41	54	0.3	15.41
Average	1.36	4.51	58	59	0.3	17.54
test-all	1.29	436.29	2328	192	2.4	0.53
Average	1.34	76.47	436	81	0.6	14.71

Table 5.5: Dynamic bytecode execution metrics for Yarv (Ruby) on x86_64.

5.3. Code-Copying Results

VM	benchmark	Direct			Copying		
		ia32	x86_64	ppc	ia32	x86_64	ppc
SableVM	compress	58.4%	37.3%	34.3%	3.8%	5.6%	17.3%
	jack	40.7%	33.0%	29.5%	20.2%	15.0%	22.2%
Ocaml	quicksort.fast	53.2%	20.9%	28.7%	4.5%	4.2%	11.3%
	almabench	38.9%	27.1%	37.0%	3.8%	6.5%	26.1%
Yarv	nsieve	12.8%	15.2%	15.5%	0.2%	2.5%	3.8%
	pentomino	9.6%	14.2%	12.8%	6.2%	6.7%	10.8%

Table 5.6: Direct-threaded and code-copying implementation branch misprediction percentages across architectures, for best and worst performing benchmark on each VM. For ia32 and x86_64 the numbers represent the ratio of mispredicted branches to total branches, and for PowerPC the numbers represent the ratio of instruction pipeline flushes due to branch misprediction to total branches.

Figures 5.6 and 5.7, and for Yarv in Figures 5.8 and 5.9. To make observations of benchmark behavior based on hardware counter data easier in Figures 5.4 to 5.9 under each bar chart of hardware counter results an additional marking showing the expected impact of the measured hardware event on the overall performance is added. The ‘++’ symbol indicates a very substantial improvement from using code-copying over direct-threaded, ‘+’ indicates a useful improvement, \simeq means a lack of meaningful change, and ‘-’ an expected degradation of performance.

5.3.3 General Findings

Overall branch prediction is, unsurprisingly, one of the main reasons for performance improvement. Reductions in the rate of branch mispredictions (or pipeline flushes due to branch misprediction) closely follows performance. Reductions in the total number of branches, as measured through average superinstruction length, also mirrors performance. Both are, however, constrained by the relative amount of time spent in

5.3. Code-Copying Results

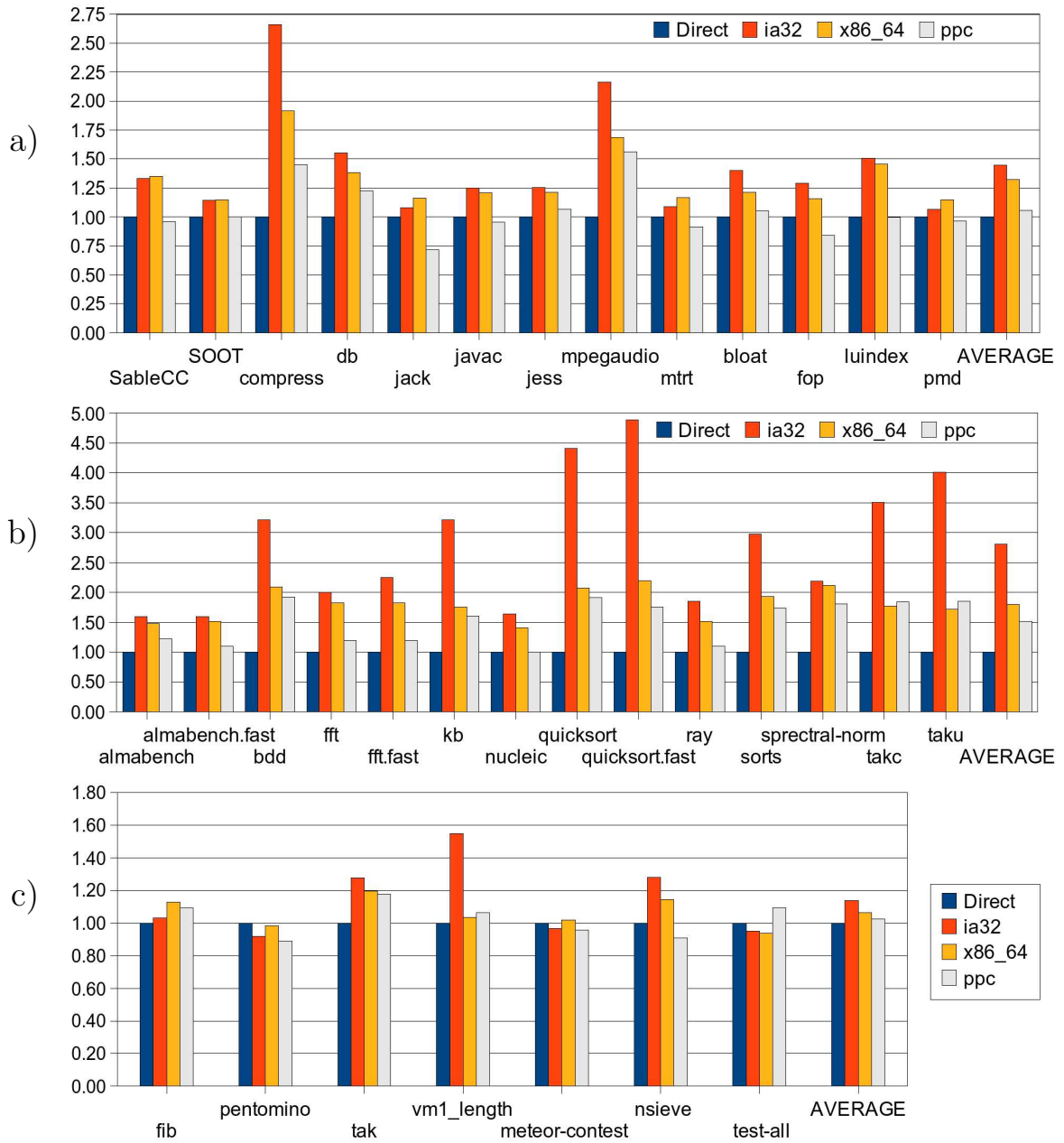


Figure 5.2: Performance (speedup) results for a) SableVM (Java), b) OCaml, and c) Yaru (Ruby). Note in this figure we measured the overall VM performance, as opposed to other results in this chapter that measure VM behavior only within the interpreter function.

5.3. Code-Copying Results

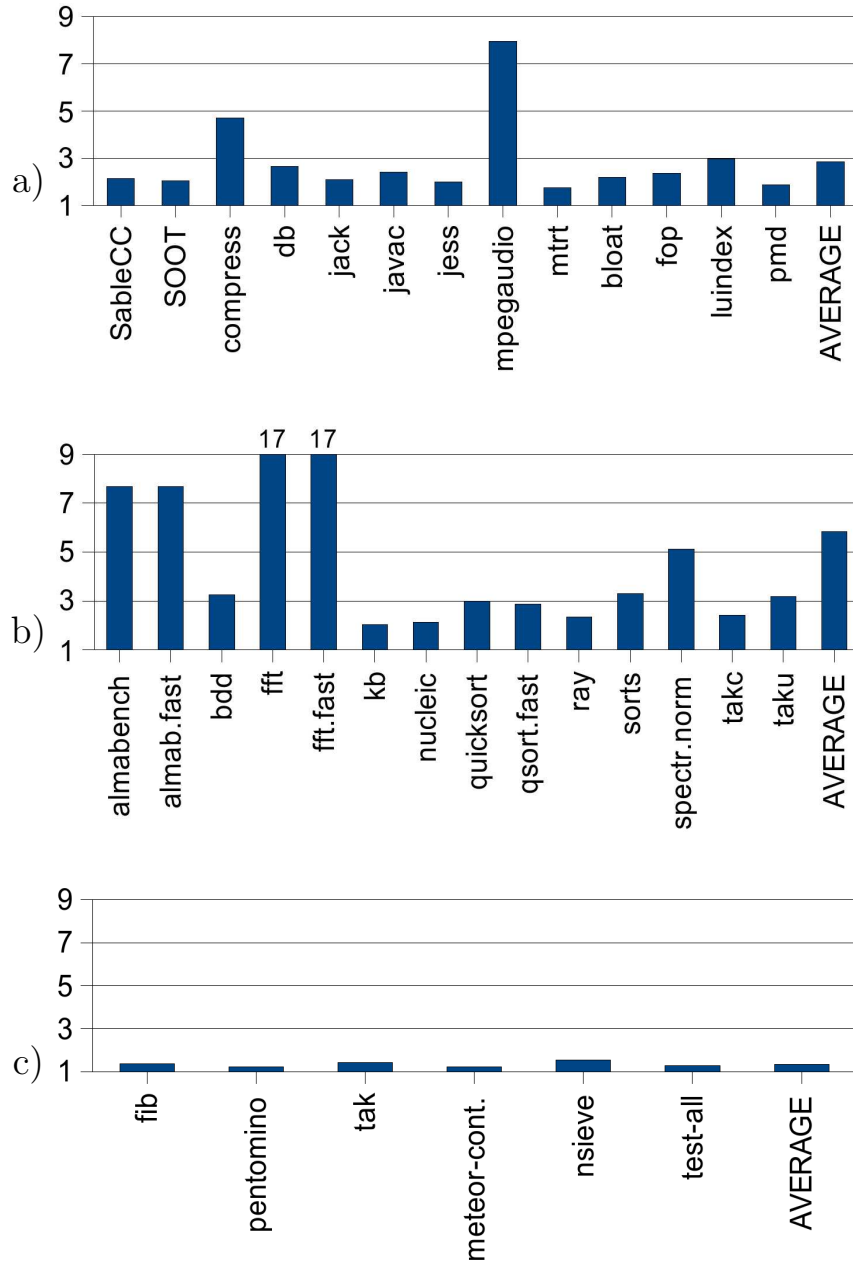


Figure 5.3: Average lengths (in bytecodes) of *executed* superinstructions for a) SableVM, b) OCaml, c) Yavv.

5.3. Code-Copying Results

benchmark	ia32	x86_64	ppc
SableCC	23.04	14.72	21.23
Soot	373.86	230.96	363.31
compress	213.35	101.70	148.96
db	76.97	48.38	74.22
jack	36.71	22.30	31.61
javac	60.26	34.14	52.49
jess	38.25	24.14	36.96
mpegaudio	178.70	84.76	126.82
mtrt	42.04	29.91	36.83
fop	247.91	159.76	237.01
luindex	2418.01	1356.92	1968.47
pmd	1385.74	884.92	1219.03

Table 5.7: Absolute runtimes (in seconds) of Java benchmarks presented in Figure 5.2 for the direct-threaded engine of SableVM.

5.3. Code-Copying Results

benchmark	ia32	x86_64	ppc
almabench	50.52	33.50	49.33
alma.fast	50.09	32.95	47.87
bdd	16.77	11.23	16.14
fft	13.01	7.57	12.33
fft.fast	14.27	7.46	12.41
kb	18.02	11.82	18.41
nucleic	28.73	18.36	25.98
quicksort	14.92	6.25	12.91
qsort.fast	15.03	5.71	11.22
ray	175.56	110.21	165.68
sorts	46.63	30.26	45.47
spec-norm	93.69	60.33	96.30
takc	15.13	6.78	13.45
taku	24.47	11.12	24.71

Table 5.8: Absolute runtimes (in seconds) of OCaml benchmarks presented in Figure 5.2 for the direct-threaded engine of OCaml.

benchmark	ia32	x86_64	ppc
fib	27.45	24.10	43.00
pentomino	43.75	40.89	69.13
tak	2.45	2.22	3.92
vm1.length	3.38	2.87	7.76
meteor-contest	26.55	27.39	46.78
nsieve	17.72	14.25	24.51
test-all	14.49	13.96	21.14

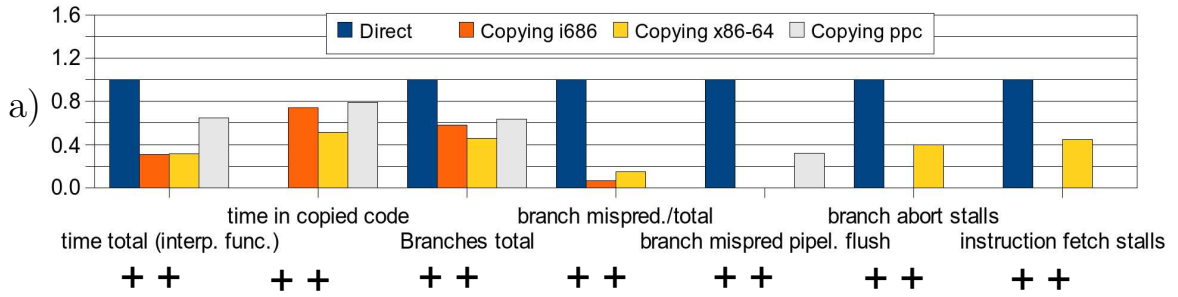
Table 5.9: Absolute runtimes (in seconds) of Ruby benchmarks presented in Figure 5.2 for the direct-threaded engine of Yarv.

5.3. Code-Copying Results

VM	Direct (ia32)	Copying (ia32)
SableVM	37%	11%
OCaml	44%	5%
Yarv	12%	8%

Table 5.10: Average branch misprediction percentages across benchmarks for ia32 on each VM, direct-threaded and code-copying.

5.3. Code-Copying Results



b)

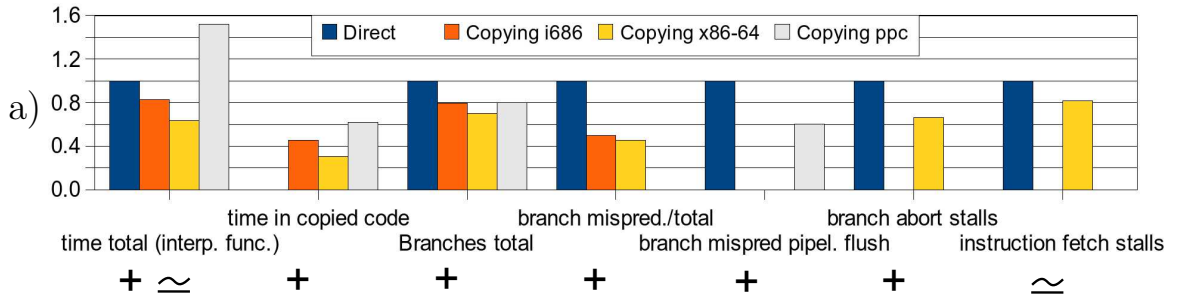
	Dir	Cpy ia32	Cpy x86_64	Cpy ppc
total time (interp. func.)	1.0	0.31	0.31	0.65
time in copied code	0.0	0.74	0.51	0.79
branches total	1.0	0.58	0.46	0.64
branches mispred. / total	1.0	0.07	0.15	-
branches mispred. flush	1.0	-	-	0.32
branch abort stalls	1.0	-	0.40	-
instruction fetch stalls	1.0	-	0.45	-

c)

	ia32		x86_64		ppc	
	Dir	Cpy	Dir	Cpy	Dir	Cpy
L1 d	-	-	3461	4349	-	-
L1 i	-	-	6	2052	-	-
L2 d	-	-	-	-	53	153
L2 i	-	-	-	-	3	8
L2 i+d	2	4	122	385	-	-

Figure 5.4: Hardware counter results for SableVM JVM running the SPEC compress benchmark. Subfigures a) and b) present branch-related results. Subfigure c) shows the number of i-cache and d-cache misses per 1M (1000000) CPU cycles.

5.3. Code-Copying Results



b)

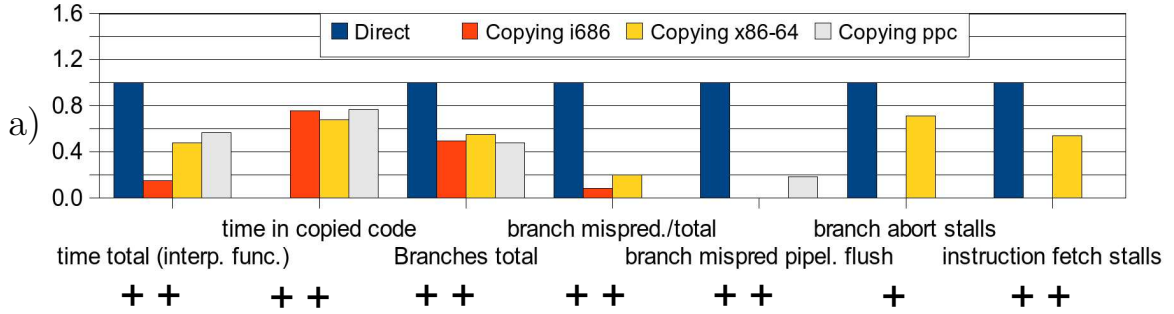
	Dir	Cpy ia32	Cpy x86.64	Cpy ppc
total time (interp. func.)	1.0	0.83	0.63	1.52
time in copied code	0.0	0.45	0.31	0.62
branches total	1.0	0.79	0.70	0.80
branches mispred. / total	1.0	0.50	0.45	–
branches mispred. flush	1.0	–	–	0.60
branch abort stalls	1.0	–	0.67	–
instruction fetch stalls	1.0	–	0.82	–

c)

	ia32		x86.64		ppc	
	Dir	Cpy	Dir	Cpy	Dir	Cpy
L1 d	–	–	9167	8721	–	–
L1 i	–	–	1450	13233	–	–
L2 d	–	–	–	–	25	212
L2 i	–	–	–	–	11	4041
L2 i+d	3	16	76	311	–	–

Figure 5.5: Hardware counter results for SableVM JVM running the SPEC jack benchmark. Subfigures a) and b) present branch-related results. Subfigure c) shows the number of i-cache and d-cache misses per 1M (1000000) CPU cycles.

5.3. Code-Copying Results



b)

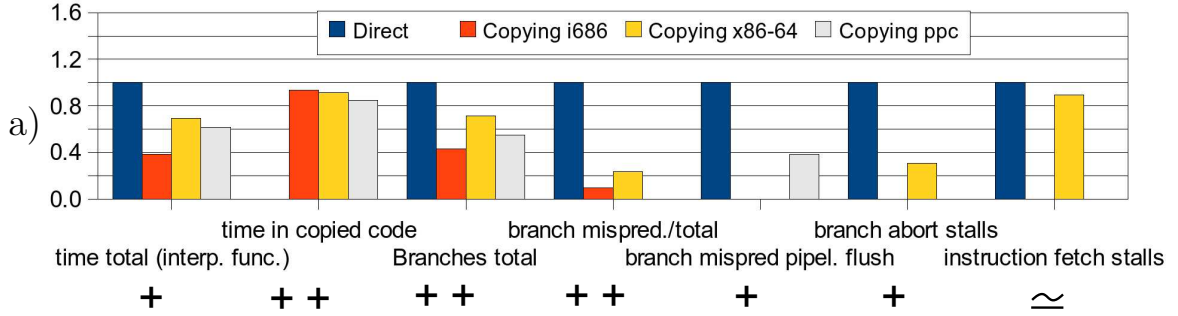
	Dir	Cpy ia32	Cpy x86_64	Cpy ppc
total time (interp. func.)	1.0	0.15	0.48	0.57
time in copied code	0.0	0.76	0.68	0.77
branches total	1.0	0.50	0.55	0.48
branches mispred. / total	1.0	0.08	0.20	-
branches mispred. flush	1.0	-	-	0.19
branch abort stalls	1.0	-	0.71	-
instruction fetch stalls	1.0	-	0.54	-

c)

	ia32		x86_64		ppc	
	Dir	Cpy	Dir	Cpy	Dir	Cpy
L1 d	-	-	522	472	-	-
L1 i	-	-	1	2	-	-
L2 d	-	-	-	-	2	2
L2 i	-	-	-	-	0	1
L2 i+d	0	1	5	10	-	-

Figure 5.6: Hardware counter results for OCaml running the quicksort.fast benchmark. Subfigures a) and b) present branch-related results. Subfigure c) shows the number of i-cache and d-cache misses per 1M (1000000) CPU cycles.

5.3. Code-Copying Results



b)

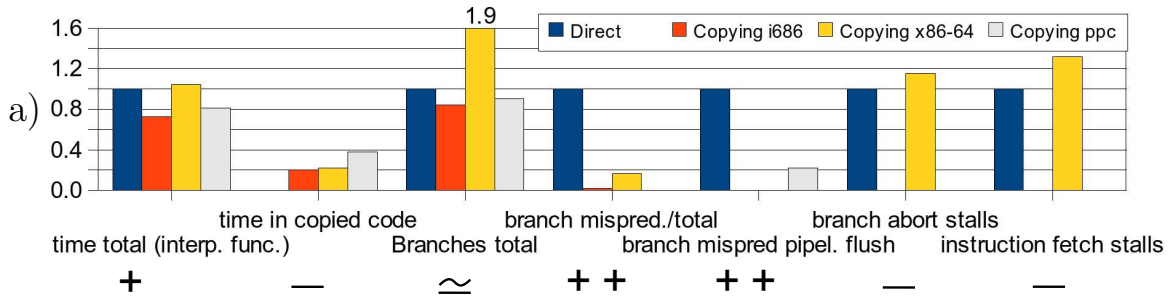
	Dir	Cpy ia32	Cpy x86_64	Cpy ppc
total time (interp. func.)	1.0	0.38	0.69	0.61
time in copied code	0.0	0.94	0.92	0.85
branches total	1.0	0.43	0.71	0.55
branches mispred. / total	1.0	0.10	0.24	-
branches mispred. flush	1.0	-	-	0.39
branch abort stalls	1.0	-	0.31	-
instruction fetch stalls	1.0	-	0.90	-

c)

	ia32		x86_64		ppc	
	Dir	Cpy	Dir	Cpy	Dir	Cpy
L1 d	-	-	10749	8979	-	-
L1 i	-	-	401	6491	-	-
L2 d	-	-	-	-	2	12
L2 i	-	-	-	-	2	9
L2 i+d	0	1	47	185	-	-

Figure 5.7: Hardware counter results for OCaml running the almbench benchmark. Subfigures a) and b) present branch-related results. Subfigure c) shows the number of i-cache and d-cache misses per 1M (1000000) CPU cycles.

5.3. Code-Copying Results



b)

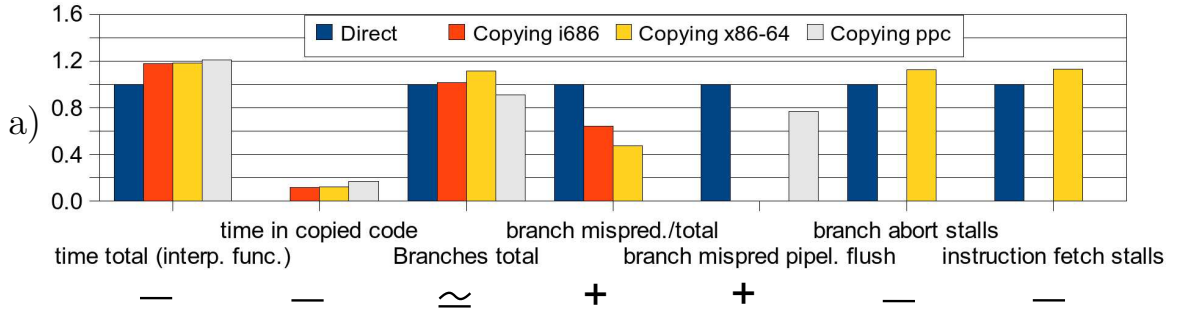
	Dir	Cpy ia32	Cpy x86_64	Cpy ppc
total time (interp. func.)	1.0	0.73	1.05	0.81
time in copied code	0.0	0.20	0.22	0.38
branches total	1.0	0.84	1.85	0.90
branch mispred. / total	1.0	0.02	0.17	–
branches mispred. flush	1.0	–	–	0.22
branch abort stalls	1.0	–	1.15	–
instruction fetch stalls	1.0	–	1.32	–

c)

	ia32		x86_64		ppc	
	Dir	Cpy	Dir	Cpy	Dir	Cpy
L1 d	–	–	2108	1914	–	–
L1 i	–	–	2	2	–	–
L2 d	–	–	–	–	6	5
L2 i	–	–	–	–	2	3
L2 i+d	428	598	522	633	–	–

Figure 5.8: Hardware counter results for Yarv Ruby VM running the nsieve benchmark. Subfigures a) and b) present branch-related results. Subfigure c) shows the number of i-cache and d-cache misses per 1M (1000000) CPU cycles.

5.3. Code-Copying Results



b)

	Dir	Cpy ia32	Cpy x86_64	Cpy ppc
total time (interp. func.)	1.0	1.18	1.18	1.21
time in copied code	0.0	0.11	0.12	0.17
branches total	1.0	1.01	1.12	0.91
branches mispred. / total	1.0	0.64	0.47	-
branches mispred. flush	-	-	-	0.77
branch abort stalls	1.0	-	1.13	-
instruction fetch stalls	1.0	-	1.13	-

c)

	ia32		x86_64		ppc	
	Dir	Cpy	Dir	Cpy	Dir	Cpy
L1 d	-	-	2340	3585	-	-
L1 i	-	-	4235	2484	-	-
L2 d	-	-	-	-	7	5
L2 i	-	-	-	-	7	6
L2 i+d	16	26	133	120	-	-

Figure 5.9: Hardware counter results for Yarv Ruby VM running the pentomino benchmark. Subfigures a) and b) present branch-related results. Subfigure c) shows the number of i-cache and d-cache misses per 1M (1000000) CPU cycles.

5.3. Code-Copying Results

copied code.

Branch misprediction (or pipeline flush due to branch misprediction) rates are found in Tables 5.6 and 5.10. A comparison between the rates for direct-threaded and code-copying VMs reveals a correlation between these results and performance. In each of the analyzed cases a significant drop in branch mispredictions due to code-copying results in a significant performance improvement, and smaller branch misprediction drops resulted in more moderate performance improvements.

The relative impact can largely be explained by considering the branch prediction capabilities of the different architectures. The Pentium 4 (Prescott core) has a 31 stage pipeline, along with 4k entries in the front-end BTB (Branch Target Buffer) table, and 2k entries in the back-end BTB. A specialized predictor borrowed from the Pentium M series is used to improve the prediction of indirect branches. Unfortunately, this predictor has serious performance problems with consecutive indirect branches, and is designed to work best when indirect branching is interleaved with direct branches, a property which is generally not true of direct-threaded code execution. Limitations such as this, the relatively small size of BTB tables and a very long pipeline mean the impact of complex branching can be large on the Pentium 4, and we conclude these as the reasons for the very high branch misprediction rates in the direct-threaded engines.

The AMD64 X2 Dual CPU (Hammer core) has a 12 stage pipeline with branch misprediction penalty being 11 cycles. It uses three branch prediction structures. The local branch history table has 2k entries, the global history table has 16k entries, and a *branch selection table* is used to decide which of these two predictors is expected to give a more accurate prediction. Additionally it also has a specialized unit called the *branch target address calculator* which diminishes the penalty caused by a wrong prediction. A short pipeline, advanced, hybrid prediction strategy, and more abundant resources allow this architecture to greatly reduce misprediction-rates over Pentium 4.

Our PowerPC G5 (970FX) uses 10 execution units per CPU with a 25 stage pipeline. Similar to the AMD64 it employs a three-part branch prediction strategy,

5.3. Code-Copying Results

although with tables allowing 16k entries each. The global history table contains 11-bit long vectors of branch execution history. Up to 2 branches can be predicted per cycle and up to 16 predicted branches can be in flight. Despite the larger tables, on a directed-threaded engine this has a roughly comparable branch-prediction behavior to AMD64.

Improvements brought by code-copying mostly correlate well with branch prediction behavior—the AMD64 machine is able to better predict branch targets for SableVM and OCaml than the Pentium 4 (ia32 architecture), and so the drop in mispredictions due to using code-copying is correspondingly smaller on x86_64 than ia32. PowerPC shows even less impact from branch prediction improvements, and this hierarchy is reflected in the overall performance of the three architectures. The code-copying implementation of the compress benchmark, for example, reduces the misprediction rate on ia32 from 0.584 to 0.038 (just 7% of direct-threading), on x86_64 from 0.373 to 0.056 (down to 15%), on PowerPC pipeline flushes due to branch mispredictions drop from 0.343 to 0.173 (32%). Similarly, (Figure 5.2) ia32 performance is greatly increased (speedup 2.66), x86_64 performance is nicely improved (speedup 1.92), and PowerPC performance improved somewhat (speedup 1.45).

The number of branches is obviously an important performance factor affected by code-copying, both through reduced branch mispredictions, and with the elision of branches within a superinstruction, through the execution of fewer instructions. The average, dynamic length of superinstructions (in bytecodes) is shown in Figure 5.3. this behavior can be directly related to the average VM performance shown in summary Table 5.2. The longest superinstructions (6.17 bytecode on average) are found in OCaml and it also delivers the best performance out of all 3 VMs. Somewhat shorter (2.84 bytecode on average) superinstructions are used by SableVM, allowing it to deliver very good performance, but not as much improved as OCaml. Yarv execution results in very short (1.36 bytecode on average) superinstructions, and comparatively poor overall performance. Unfortunately, although Yarv still has about half of its bytecodes copyable these do not tend to form large contiguous sequences at runtime.

The overall impact of benefits to branch prediction and code execution are reduced if they are not applied reasonably ubiquitously. Data in proportion to time spent in

actual copied code as opposed to non-copyable bytecode executions is shown for the individual case studies in Figures 5.4 through 5.9. In the case of OCaml benchmarks in Figures 5.6 and 5.7, for example, we see that the fraction of interpreter loop time spent in copied code ranges from about 70% to about 90%. The other extreme is again represented by Yarv, where the time spent in copied code ranges from only about 15% to about 25%. Modulo cache and other non-local effects these ratios provide an upper limit on the potential performance improvements.

5.3.4 Overhead

Overhead in a code-copying system comes from several sources. Actual copying of code increases code-preparation time, and memory and superinstruction management add additional costs, although these are one-time costs amortized over the lifetime of execution. Ongoing overhead is mainly due to changes in code-generation from the compiler enhancements that support code-copying. Some of the modifications done to the compiled code by our passes create in the code elements that can act as *code-motion barriers* (for example the *volatile asm* statements) that prevent certain optimizations across such a barrier. These addition of code-motion barriers can be expected to inhibit or alter application of some optimizations.

To measure the overall overhead we compared performance of direct-threaded VMs with the performance of code-copying VMs, where the copied code is created, but not actually used at runtime. Figure 5.10 (page 93) summarizes results; SableVM data is not gathered due to difficulties in separating the code-copying engine from direct-threaded execution. Overhead varies considerably, changing performance on average between -5% and 11%, and overall between -20% and nearly 25% for the two VMs. Negative overhead is possible due to instruction cache changes, and perhaps poor heuristic performance of basic block rearrangement or other optimizations that are (locally) prevented by the code-copying enhancement. We note, however, that for OCaml overhead is fairly low and often negative; for Yarv overhead is almost uniformly positive and sometimes large. This also partially accounts for and reflects the lower improvement experienced by Yarv due to code-copying.

5.3. Code-Copying Results

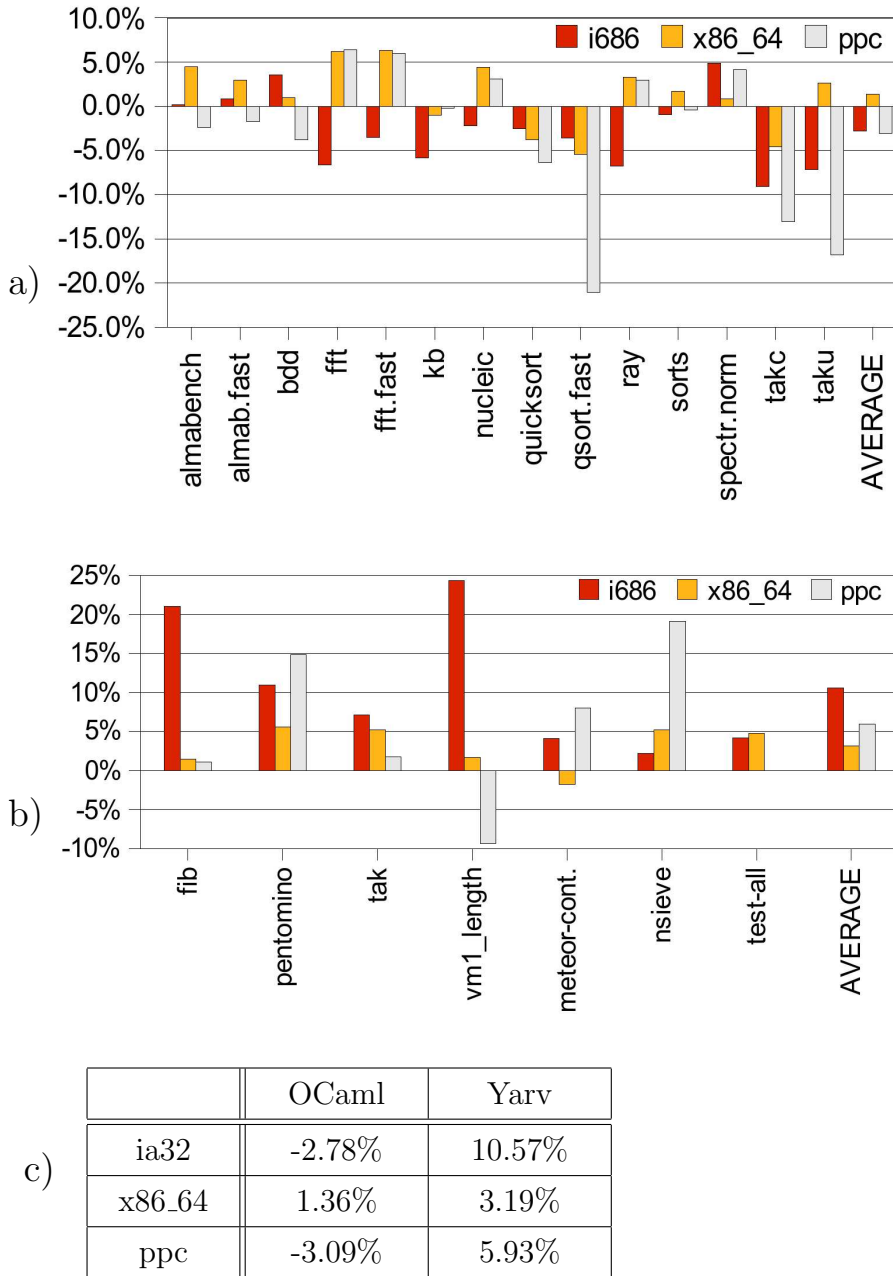


Figure 5.10: Overhead of VMs with *pragmas* inserted around bytecode instructions used by code-copying and with copied code prepared but not executed, over standard direct-threaded VMs. Part a) shows OCaml, b) Yarv, and c) averages.

5.3.5 Further Performance Factors

While branch instruction effects are primary in a general sense, individual benchmarks naturally vary, and raise a number of issues not obvious or clear from a simple consideration of the branch-reducing nature of code-copying or its basic overhead. Instruction cache behavior, for instance, is also potentially negatively affected by the size and number of superinstructions. Other, benchmark-specific and compiler-driven factors can further intrude. Below we discuss these issues in relation to our experimental data.

Instruction Cache Impact

Using any dynamic code creation technique carries the danger of creating too much code and lowering the performance (increasing miss rates) of the CPU instruction cache. Our results show that excessive code size is not a significant concern for our VMs and tests. From Tables 5.3, 5.4, and 5.5 we see that copied code usually occupies at most a few hundred bytes per superinstruction, with total size averaging less than 1MB in Java and well less than 100kB in OCaml and Ruby. Despite this large size difference, a consideration of *important* code size shows the VMs are both similar and not overly affected by too many distinct superinstructions. When looking at code within the 90% of most actively executed bytecode instructions memory requirements are much reduced, 13kB on average for Java, 7kB on average for OCaml, and never more than 23kB in any tested case.

Code size is a rough predictor of I-cache performance; better information is obtained by measuring actual cache miss behavior. Part c) of Figures 5.4 through 5.9 (pages 84 through 89) shows runtime instruction and data cache miss rates for the best and worst performing benchmarks. While there is a general, and sometimes marked (Figure 5.4) increase in L1 I-cache misses the effect is not uniform, and does not correlate well with performance. Of course L1 misses are not necessarily that expensive if caught by the L2 cache, and a more important concern with L1 usage is that an increase in I-cache pressure can cause more spillover into the L2, and thus

5.3. Code-Copying Results

greater chance of having to undergo the costly step of retrieving data from main memory. This effect is evident in the SPEC jack benchmark on PowerPC; in part c) of Table 5.5 there is a large increase in the L2 I-cache miss rate due to code-copying. We believe that this higher L2 I-cache miss rate is part of the reason for this benchmark's lower performance on PowerPC.

Improvements to instruction cache miss rates may be possible by exploiting the heavy concentration of execution in a relatively small number of superinstructions. We expect we would be able to achieve most of the performance benefits of code copying by only creating a small fraction of the code we create currently, although this requires profiling or adaptive techniques to discover the 90% of most actively used superinstructions.

Extreme Case of Code Preparation Overhead

The test-all benchmark on Yarv is the largest Ruby benchmark we used and its behavior and characteristics are very different from other benchmarks. This benchmark performs surprisingly poorly, but for other reasons than originally expected. As can be seen in the dynamic metrics in Table 5.5 it creates a large amount of copied code, over 3200 superinstructions using over 430kB of memory, an exceptionally large number of superinstructions in comparison with other Ruby benchmarks. As a code sanity test, the bulk of code in this program consists of small tests executed once, or only a few times. This results in only about 0.5% of the code created accounting for 90% of execution. Code-copying itself has little positive impact in this situation; interestingly, as seen in the actual performance of test-all (Figure 5.2) the larger overhead of copied code does not have as large a negative impact as these relative results for Yarv may suggest.

Performance degradation (on Intel-like architectures) is in fact dominated by *code preparation* costs, in which the VM initializes code for execution, adding an overhead which is not in this case amortized over multiple executions of the created code: only about 7% of execution time is spent in the interpreter loop. Since the behavior of this benchmark is unusual we present averages in Tables 5.5 with and without the

test-all benchmark included.

Extreme Case of Inhibited Compiler Optimizations

In early experiments with the code-copying version of Yarv an anomaly was found when running the nsieve benchmark on the x86_64 architecture. Paradoxically, the number of branches executed in the interpreter loop *grew* by a factor of 1.85 in the code-copying engine, and moreover disabling runtime use of code-copying and re-measuring performance produced the same result. We attribute this to the interaction of code-copying and compiler optimization, an overhead consideration in general. Non-localized disabling of branch or basic block reordering optimizations, in combination with benchmark-specific behavior would account for the increased number of branches in the interpreted code, irrespective of whether it was actually code-copying or executing code in the normal, direct-threaded fashion. This behavior did not occur with other benchmarks or on other platforms, and while it may have reduced performance it did not result in a slowdown—the overall overhead for nsieve is still small on x86_64 (Figure 5.10). In Chapter 6 we show how by using *source-optimized superinstructions* we can provide performance equal or better than that of code-copying and avoid inhibiting any compiler optimizations by removing the need for `#pragma copyable` use.

Summary

Different languages and VM (bytecode) designs have a strong impact on the performance benefit provided by code-copying. At one end of the scale languages and bytecode instruction sets like Ruby (and the particular VM we used—Yarv) introduce several problems when it comes to the application of code-copying. Outwardly small bytecodes in fact perform significant amounts of runtime work, making numerous calls to helper functions and often changing control flow of the program. This constrains code-copying, making long superinstructions unlikely. With little time overall spent in the actual interpreter loop, the main positive effect of code-copying in reducing branch costs is greatly lessened.

Dramatically better effects are shown on a language like OCaml, that has functionally small, fine-grained bytecodes containing mostly simple operations. Full implementations are also typically contained within the bytecode itself, with no need for functions called externally. Code-copying applies very well to this kind of VM design since it is possible to create many superinstructions (improving branch prediction, even for short superinstructions) and superinstructions that are longer (removing many branches). Java bytecode has qualities of both Ruby and OCaml; performance fortunately falls closer to the OCaml side of the spectrum. Further simplification of bytecode implementations, however, would likely increase the performance improvement.

All behavior is affected by the quality of hardware branch prediction. In general PowerPC shows the least gains and Intel 32-bit the most, largely reflecting the relative branch prediction capabilities of the different architectures. The effect of hardware is still not as pronounced as aspects of virtual machine design, and hardware resources are scarce, but clearly improved hardware branch prediction would be another source of optimization especially applicable to interpreter-based virtual machines.

5.4 Notes on Related Work

The related work for this chapter is mostly in the area of hardware branch predictors behavior during interpreter execution. Application of the code-copying technique to GForth has been analyzed in a few studies by Ertl [EG03c, ETK06, EG03b], showing that the majority of performance improvement is due to improvements in branch prediction. Ertl et al. also compared code-copying (called in these works *dynamic superinstructions*) to other techniques intended to improve branch prediction behavior like *dynamic* and *static instruction replication* and *static creation of superinstructions*. They demonstrated that all of these techniques (often combined together) can also bring significant performance improvements. Their results also demonstrated, once again, that speedup due to branch prediction improvements expectedly outweighs other negative effects, such as a slight increase in instruction-cache misses. Vitale

et al. [VZ05] analyzed applicability of code-copying technique to a Tcl interpreter characterized by large bytecode bodies and on a set of dispatch-intensive benchmarks achieved an almost 10% speedup.

In our work we performed a much broader analysis of hardware behavior by employing 3 hardware architectures and testing code-copying implementations for 3 vastly different programming languages. This allowed us to draw conclusions across different hardware and programming languages and this way giving a more comprehensive view of applicability of the code-copying technique. More information regarding the unique extent and direction of our work can be found in the section that follows.

5.5 Conclusions

An examination of the behavior of different languages and virtual machines is useful for any cross-context optimization. Code-copying has been prototyped in single environments before, but our work represents the first multi-language, multi-platform examination of performance, based on a safe implementation model for code-copying. There is clearly a spectrum in the performance impact of code-copying, with behavior depending on virtual machine implementation, bytecode design, and hardware considerations.

Although various factors can influence results, bytecode properties dominate. Simple bytecodes can be easily collected into superinstructions, and imply more time in the actual interpreter loop, raising the upper bound on potential impact. Virtual machine designs that stress simple as opposed to complex bytecode behavior represent a trade-off between simplicity of code generation/execution, and smaller code size. Code-copying can be applied in both scenarios, but practical limitations mean performance improvement is best for simple designs, where overhead is heavily concentrated in interpreter loop dispatch costs.

Our choice of VMs allowed us to begin with an established, existing implementation of code-copying and then extend our field of experimentation with one VM (OCaml) whose characteristics suggested it might be an even better candidate for

code-copying than the first VM (SableVM), and another VM (Yarv) whose characteristics suggested that obtaining speedups from the use of code-copying might be problematic. We believe our choice gave us an opportunity to explore both the positive and negative examples of code-copying application hence allowing future researchers and practitioners to both understand what there is to be gained and why, but also what they need to avoid.

Determining the potential impact of code-copying ahead of time benefits virtual machine designers as well as implementers contemplating the use of code-copying. Based on our experiments we are able to tie certain characteristics of bytecode and interpreters using direct-threading to the performance improvement provided by code-copying. Our experience suggests a short, step-by-step procedure for estimating the degree to which a virtual machine currently using direct-threading may improve after application of code copying.

1. Determine the bytecode instructions in the source code that can be used in code-copying.
2. Calculate the relative number of bytecodes in the interpreter source code that are potentially copyable. Check if applying an upper limit to the bytecode size, e.g. 150-200 bytes, makes a difference. Refer to Table 5.1 for evaluation. The more (and more small) bytecodes that are copyable the better for code-copying.
3. On a set of benchmarks measure how many loaded bytecodes are potentially copyable. Refer to Figure 5.1 for evaluation. The more loaded bytecodes that are copyable the better.
4. On a set of benchmarks measure how much time is spent in the interpreter loop (excluding functions called from within the loop). The more consistent the results and the more time is spent in the loop the more generally useful and effective code-copying will be.
5. On a set of benchmarks and using a given architecture measure branch prediction miss-rates within the interpreter loop. Refer to Table 5.2 for interpretation. The higher the miss-rate the more room for improvement by using code-copying.

Based on these simple measurements (or as many as can be acquired) it should be possible to gain a meaningful insight into whether an investment of time and effort into a code-copying implementation for a particular language and virtual machine is worthwhile. Note that an unpromising evaluation outcome does not necessarily mean code-copying will never be applicable to a language or a virtual machine. For example, some forms of bytecode optimization or restructuring, as was originally done for SableVM [Gag02], may improve the end result.

Our implementations of code-copying form an initial step in analyzing and optimizing performance through this technique. The typically small size of the core runtime bytecode working-set (top 90% of executed bytecodes) for all our virtual machines suggests significant improvements to runtime overhead are possible by not copying all copyable sequences executed, either through adaptive selection of the sequences to copy (e.g., using hotness counts) or by using ahead of time profiles to guide choices. This would further improve performance, and make the technique more generally attractive even when branching is not known to be a major performance bottleneck. In the next chapter we present a technique that provides performance comparable and better than code-copying and requires no C compiler extension. In this approach 99% of most often executed superinstructions are selected and a standard C compiler is used to perform inter-bytecode optimizations on these superinstructions, and by that further improve the performance of interpreters.

Chapter 6

Virtual Machine Specialization with Caching Compilation Server

In this chapter we present a specialized compilation server that provides further performance improvements to virtual machines by enabling optimizations within selected superinstructions. Our server is architecture and VM-agnostic and can be used with virtually any bytecode-based virtual machine interpreter. In the following sections we discuss the architecture of our system, the specifics of our implementation, performance results for two virtual machines, compare our system with other related works and finally present our conclusions.

The result of our work is a system composed of multiple virtual machines that use a caching compilation server service. The compilation server is used to create and store specialized VM binaries optimized for particular applications. To improve performance of a virtual machine beyond what interpreter-based solutions can provide, a compiler is required. Projects with sufficient funds usually create their own just-in-time compiler from scratch, specialized for a particular language. This costly approach is not available to smaller projects. In our solution we leverage an existing, industry-standard highly-optimizing static compiler, GCC. A profile-enabled, code-copying interpreter is used to detect frequently used (hot) groups of instructions (superinstructions) that are candidates for optimization. The biggest drawback of such solution, the long compilation time of a static compiler, is dealt with by caching

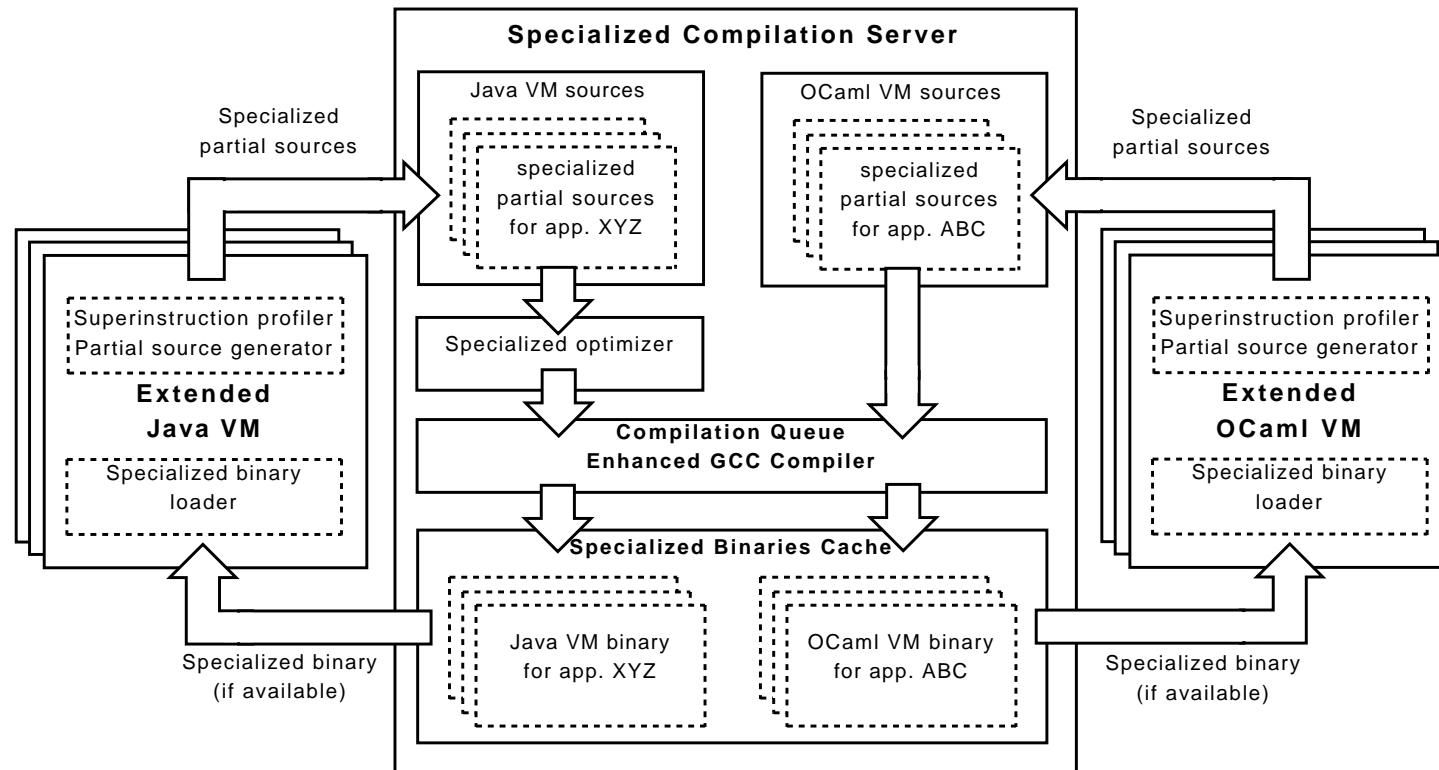


Figure 6.1: Overview of the specialized compilation server using the enhanced GCC with two extended virtual machines.

the resulting binaries. This way the compilation overhead is spread over multiple runs of a virtual machine. The performance improvement over previously evaluated purely code-copying interpreters is most visible in the case of OCaml, with average speedup of 27%, and a maximum of 81%. Java interpreter, SableVM, shows smaller benefits of 2-2.5% on average, with the top-5 benchmarks improving by 4-5%, and an 8% maximum. Our analysis of the results indicates that the performance difference between languages results from their dissimilar functional properties.

The rest of this chapter is structured as follows. We first present an overview of our design, then we explain the source of performance improvement, followed by detailed description of how specialized source code is created, including optional source optimization; we then present the process of optimized VM recompilation, our performance results, notes on selected related work, and, finally, we draw the conclusions.

6.1 System architecture

We designed our optimization system to achieve the best possible performance results while avoiding low-level, architecture-specific concerns and maximizing the reuse of existing solutions. Figure 6.1 presents an overview of the implemented system.

The main components of the compilation server are as follows:

- Virtual machine sources repository – the server stores multiple sets of *partial* sources it receives from extended virtual machines. Each set is specialized, i.e. it is generated based on the behavior of a specific application.
- Specialized optimizer – because the server has access to the specialized sources of a VM there exists an opportunity to perform additional analysis and optimizations in the source code that will improve the final result of compilation. This component is optional.
- Compilation queue – The server can receive multiple compilation requests which might need to be delayed. The server can be configured to allow for a specific

number of compilations to occur simultaneously. It could also order compilations in a specific manner, e.g. giving priority to a specific VM.

- Enhanced GCC Compiler – we used a compiler supporting safe code-copying because we wanted the resulting virtual machines to use this technique next to source-optimized superinstructions. We should note that these two techniques are separable and, while not using code-copying would slightly lower the overall performance of virtual machines in our setup, any C compiler could potentially be used.
- Cache of specialized binaries – after a compilation task is completed the resulting binary of a specialized virtual machine is stored in a file cache that is accessible to extended virtual machines.

Our compilation server requires that virtual machines are extended to cooperate with the server. We optimized the division of the tasks between the compilation server and virtual machines so as to avoid unnecessary complications to the system. By our design we perform each task in the most suitable place, where all input data is readily available and an action can be performed on this data, instead of encoding and transferring this data for later processing to a separate tool. This principle applies in particular to the specialized code generation which is performed by virtual machine itself.

Each extended virtual machine needs to be able to perform the following tasks.

- For a specific application a VM needs to search the server cache for a specialized binary and load it – if the binary is available. If this step is successful then the next tasks are generally not used in this run of the virtual machine.
- Profiling the application – a virtual machine gathers the execution profile of superinstructions used by the application.
- Partial source code generation – the profile is then used to generate the specialized source code and sent to the compilation server. We discuss profiling and source code generation in details in a later section.

The compilation server and extended virtual machines cooperate in the manner illustrated in Figure 6.2. Initially the compilation server is idle and waiting to be contacted by a virtual machine. A virtual machine is about to begin an execution of an application. It searches server cache for an optimized binary. If it finds one it loads this binary and executes the application on a specialized VM. If it does not find an binary optimized for the application it is about to run it begins executing the application using a non-specialized, *vanilla* version of the VM engine. It gathers the execution profile. When enough profiling information is gathered it uses this information to generate optimized partial sources of itself and sends them to the compilation server as a compilation request. The VM continues execution of the application with no further profiling overhead. The compilation server receives the request, optimizes the partial sources (optional), and enqueues the compilation request. When the compilation is complete the resulting specialized binary is placed in server cache. From that point on every virtual machine about to execute the same application will find the specialized binary in the cache and will load it and use it to execute the application at a greater speed.

6.2 The source of performance improvement

The goal of our system was to improve the performance of virtual machines. Such improvement does not come from the use of compilation server. Rather, the server is only a convenient way of engineering a system that takes advantage of existing optimization opportunities. In our case this opportunity was the suboptimal code of superinstructions. In the previous two chapters we described the superinstructions as created by concatenating binary code from several small, contiguous memory regions into one larger contiguous memory region. While the performance improvement achieved was very substantial and added safety allowed reliable implementations in multiple virtual machines there was at least one clear optimization opportunity that was not yet exploited.

6.2. The source of performance improvement

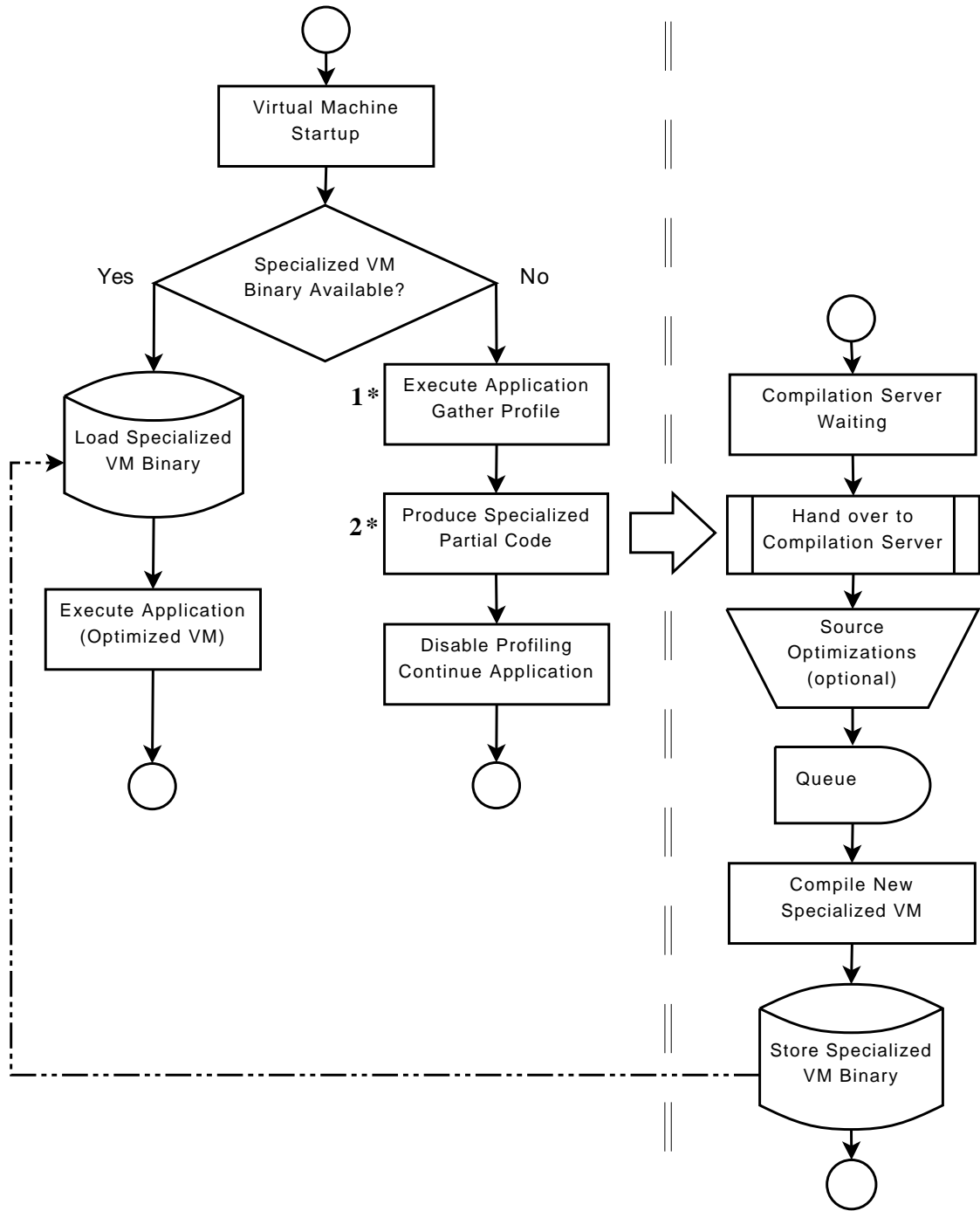


Figure 6.2: Order of operations initiated by a virtual machine cooperating with our compilation server.

6.2. The source of performance improvement

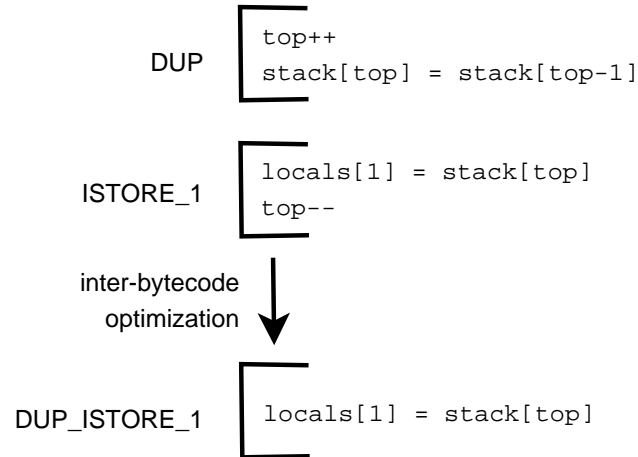


Figure 6.3: One of the obvious expected advantages of inter-bytecode optimization was the removal of unnecessary store and read operations for temporary values.

The performance improvement in our system comes mainly from performing optimizations *within* superinstructions. Figure 6.3 presents an actual case where superinstructions are clearly inefficient because they are focused on executing each bytecode, instead of focusing on the overall effect of execution of all instructions in a superinstruction. The Figure demonstrates how two bytecodes `DUP` and `ISTORE_1` perform stack *push* (meaning data store in memory and incrementation of stack top pointer), and *pop* (meaning data read from memory and decrementation of stack top pointer) that undo one another's work. This problem can be remedied by introducing optimizations that span across instructions boundaries, optimizations that work globally within a superinstruction (`DUP_ISTORE_1` in the Figure). One possible solution would be to write a specialized *machine code to machine code* optimizing compiler. Such systems exist already (e.g. Dynamo [BDB00]) and require substantial amounts of architecture-specific code and re-implementation of many generic compiler optimizations. Continuing our practice of maintaining maximum independence from specific hardware, reusing existing tools and solutions, and addressing issues at a higher level we decided on a different and, from our point of view, more advantageous solution.

The following elements are important when analyzing the sources of performance

improvement in our system:

1. *crucial inter-instruction optimizations* — Our system uses the source code of instructions as the singular elements from which *sources* of whole superinstructions are created. This is done simply by concatenation of source code of each instruction instead of their binary code as it is done in code-copying. The source code is then sent to a C compiler which, in almost all cases, can perform the wanted optimizations. As we will show later, this approach enabled the compiler to perform many crucial inter-instruction optimizations that were not possible before. For the purpose of this work we will call these superinstructions *source-optimized superinstructions*.
2. *independence of code-copying* — The use of source-optimized superinstructions is de-facto independent of the use of code-copying. In our design we have extended a pre-existing framework for code-copying to make the detection (via profiling) and creation of source-optimized superinstructions easier. At the same time it would be possible to build it on top of a switch or direct-threaded interpreter. In the system we implemented the use of code-copying provides about 1% of the speedup (over direct-threading) while about 99% of the performance improvement (over direct-threading) is due solely to the use of source-optimized superinstructions. This is because in our system we choose the 99% of most often executed superinstructions as candidates for source-based optimization. Note that in this chapter we will most often compare the performance of our system with the performance of code-copying.
3. *avoiding the use of `#pragma copyable`* — Interestingly, we were also able to improve the quality of the resulting binary code by avoiding the use of `#pragma copyable` in source-optimized superinstructions. This was possible because source-optimized superinstructions do *not* need to be copied to other places in memory to be concatenated with other instructions. The use of `#pragma copyable` in chunks of code used by code-copying engines is necessary for safety purposes. Unfortunately, as we noted in Chapter 4, by ensuring the required

safety properties this pragma also inhibits certain optimizations. As we will show later in this chapter, not using this `#pragma` allows the C compiler to produce better quality code.

4. *specialized optimizations* — In Section 6.4 we present a tool that operates on C source code and optimizes accesses to a stack structure in ways a C compiler could not optimize by itself.

Overall, the main performance improvement in our system comes from allowing optimizations across instruction boundaries.

6.3 Creation of the source code

For the purpose of optimizing superinstructions across instruction boundaries our system must be able to create source code for these superinstructions. Creation of the source code of a superinstruction is a three-step process.

1. *individual source separation (vanilla VM build time)* — The source code of *each instruction* must be separated out from the sequence of instructions defined in interpreter main loop. Source code of each instruction must be available separately so it can be concatenated as necessary with other instructions' sources.
2. *profile required (1* in Figure 6.2)* — A VM might be using a large number of superinstructions. It might not be necessary or feasible to make them all source-optimized. This is because the compilation time tends to increase (approximately) exponentially with the number of instructions defined in the main interpreter loop. The superinstructions that are worth optimizing are chosen based on application profiling information, so appropriate profiling data must be gathered.
3. *sources concatenation (2* in Figure 6.2)* — For the selected, most often used superinstructions the sources of the instructions they contain are concatenated so that they can be later optimized by a compiler.

This process is described in detail in the following sections.

6.3.1 Separate per-instruction source code

To be able to create the source code of a superinstruction the source code of each instruction it contains of must be available. The natural place where the source code of each instruction is defined is the interpreter loop. In our system at virtual machine build time the interpreter loop source code is analyzed and split into multiple files – one per instruction – containing only the source code of that instruction. To that end the source code is first pre-processed by `cpp` preprocessor and pragmas, embracing labels, dispatch code – are stripped. The resulting files only contain the *effective* code of each instruction, that is, the code that actually executes the useful operations prescribed by the definition of an instruction. The mechanism for splitting and stripping the source code slightly varies for each virtual machine. This process occurs only once at *vanilla* VM build time and is *not* repeated for specialized VMs because the split source code is already available.

6.3.2 The profiling subsystem

As we mentioned before, it might not be feasible or advantageous to optimize all superinstructions used by an application, especially if the number of superinstructions is large. This makes it necessary to *choose* which superinstructions will be optimized and this choice is based on the behavior of an application that is measured by a profiling system. The two most popular approaches to profiling and using the gathered optimization for optimizing the application are the following:

- Executing training set of applications (or the same application multiple times), then generating an optimized VM using the gathered overall profile.
- Dynamic optimization where profiling and optimization (recompilation) happen in a single run of an application. Most optimizing JITs use this technique.

Our system is more similar to the first one with the difference that we use only a single run of an application (or approximately the first 30 seconds of the run) to gather the profile. Also, we do not attempt to create a VM optimized to an "average"

application. Instead we optimize one copy of the VM for each application it runs. The optimized VM is made available to all new instances of application after the compilation of optimized VM is complete, and is not available to the already running VMs, although this could potentially be achieved via *on-stack-replacement*.

The profiling system we implemented measures the frequency with which superinstructions are executed. The choice of superinstruction as the main profiling unit is natural because the goal of the profiling subsystem was to identify the superinstructions that were executed most often (*hot* superinstructions). An additional instruction `PROFILE` was defined that is inserted at the head of each profiled superinstruction and can be removed at runtime, as shown on Figure 6.4. Because a superinstruction is a relatively small unit, registering execution of each single superinstruction causes a substantial overhead. We therefore decided to use a simple but more efficient method. For performance reasons the `PROFILE` instruction uses a global counter so that only every N-th execution of superinstruction is recorded in the profile. After a few experiments the value of N was chosen to be 10 so as to keep the overhead of profiling at around 2%. Also, after the profile is gathered the profiling of superinstructions is turned off with no further performance penalties.

While the choice of a particular data structure to hold the profile data has no bearing on the validity of our method, since we query our profile data often we choose an adaptive data structure with efficient query cost. In our system we use a *splay tree* data structure because, while its cost of an operation is only $O(\log n)$, it also has the property of keeping most often accessed elements closer to the tree root thus minimizing the access time for these elements. In our implementations of code-copying engine the descriptions of superinstructions are kept in a splay tree data structure using the integer values of bytecodes as keys. To profile superinstructions effectively we created a second splay tree holding profile data about each superinstruction using the starting memory address of each superinstruction as keys. At superinstruction creation time an element is inserted into each of these trees for every superinstruction.

Profiling of superinstructions can be easily enabled and disabled at runtime which allows us to remove the overhead of profiling after sufficient amount of profiling data has been gathered. As shown in Figure 6.4 making an instruction profiled or not

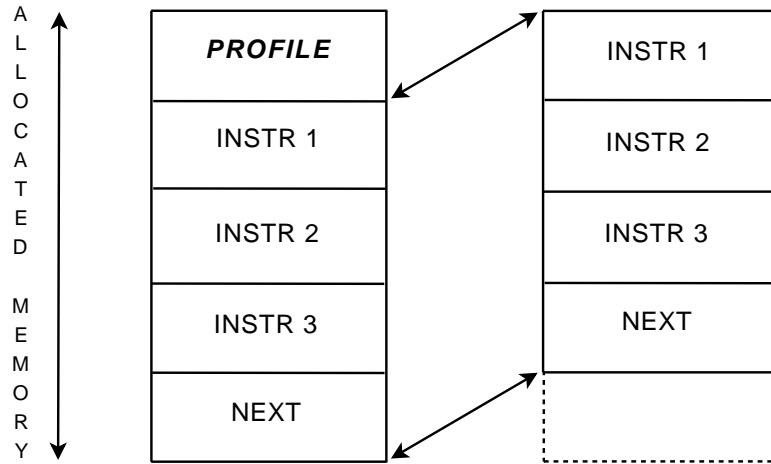


Figure 6.4: Profiling of superinstruction binary code can be turned on and off by simple `memcpy` and `memmove` operations, given that the necessary memory space has been allocated in advance.

requires only one or two simple memory copy or move operations. With this method the starting memory address of a superinstruction remains the same. This is important, because it would not be feasible to visit all loaded bytecode arrays (the content of which has been translated into memory addresses) and modify the addresses for each occurrence of a superinstruction that we want to enable or disable profiling of. In our approach the performance of superinstructions with profiling disabled is equivalent to those initially created without profiling, and the only overhead is the extra memory allocated for `PROFILE` instruction code. The function recording an execution of profiled superinstruction searches the profile splay tree using the start address of superinstruction as the key. At the time of execution of `PROFILE` bytecode this address can be found in `*(pc-1)`, since `pc` is incremented just before the jump to the start address of next superinstruction. The profiling code in SableVM ignores profile hits coming from virtual machine startup and only profiles the application. In OCaml the virtual machine startup is non-existent.

The profile is considered complete either after sufficient number of profile samples has been gathered (about 30 seconds of execution of Java VM) or the application has

exited. This imperfect method proved to be quite effective in practice. Once enough samples have been gathered the profile data can be used to choose the superinstructions that will be source-optimized.

6.3.3 Choice and creation of superinstructions

To decide which superinstructions will be *source-optimized* the profile data is used to select the superinstructions that make for 99% of all executions of superinstructions, but not more than 1000 superinstructions. As we mentioned before, the hard limit is necessary because because the compilation time tends to increase (approximately) exponentially with the number of instructions defined in the main interpreter loop. In practice the hard limit (of 1000 superinstructions) we defined was never reached by any of our benchmarks (see *source-optimized superinstructions count* in Figure 6.6) but there might exist applications where this limit would be reached.

The source code of the selected superinstructions needs to be provided to the compilation server (and later – the compiler) as regular C code, in a file. The name of the file is based on the identification of the benchmark that was being executed when the profile was gathered. For each superinstruction a header is written, then the sources of each instruction in the order they appeared in the superinstruction, then the footer is appended. Header and footer contain code that creates the labels necessary for direct-threading or code-copying engines, and optionally `#pragma copyable` if the latter engine were to use source-optimized superinstructions. At this step several other files are also created which, at compilation time, will initialize additional data structures, e.g. tables of addresses and names of the source-optimized superinstructions. All these files are handed over to the compilation server which in a later step might apply specialized optimizations to the source code (as described in the next section) and finally enqueue a compilation request.

Note that only source code for specialized superinstructions is created, since most of the VM will be left unchanged. For this reason we call this set of source code files *specialized* or *partial code*.

6.4 Specialized optimization

The partial code of VMs is stored in the compilation server before it is compiled along with the rest of VM sources. This creates an opportunity to apply additional analyses and optimizations to the source code to improve the final results of compilation.

In particular, in case of SableVM, due to initially unsatisfactory performance results, we decided to apply specialized optimization to the source code of superinstructions. We created a helper application that inputs the source code of all superinstructions, optimizes Java method stack accesses within each superinstruction, and outputs an optimized version. Our tool works directly on C source code, and understands a subset of C large enough to properly parse and analyze the source code of all instructions of SableVM.

This tool exploits one particular property of stack-based code that a standard C compiler has no way of understanding and exploiting. This is because there is no mechanism defined by the C standard (or any GCC extensions) to inform the compiler that an array structure and a pointer together describe a stack structure. Let us take an example when in stack notation a value is pushed on the stack by one instruction and then it is popped from the stack by the next one. Knowing how a stack works means understanding that the state of the stack after these two operations is the same as if the value was never written to the stack in the first place. If these two instructions are within a single superinstruction then there is no need to actually have this value ever written to memory as it can be temporarily stored in a register. Unfortunately GCC (or any C compiler we know about) will see the stack as an array that was updated and will insist on making the write actually happen.

To remedy this inefficiency our optimization tool first analyzes the source code of each superinstruction for stack accesses and stack pointer updates. The input to our tool is the source code and the names two variables that are provided manually: the variable holding the stack slots and the variable pointing to the top of the stack. The results of this analysis are then used to create a temporal map of accesses and stack sizes, as can be seen in Table 6.1. The temporal map is then used to map all input, output and temporary stack values to actual local variables in each superinstruction

6.4. Specialized optimization

```

a) #pragma stackopt begin
{
    { /* ILOAD */
      jint indx = (pc++)->jint;
      stack[stack_size++].jint =
        locals[indx].jint;
    }
    { /* ILOAD */
      jint indx = (pc++)->jint;
      stack[stack_size++].jint =
        locals[indx].jint;
    }
    { /* ISUB */
      jint value1 =
        stack[stack_size - 2].jint;
      jint value2 =
        stack[--stack_size].jint;
      stack[stack_size - 1].jint =
        value1 - value2;
    }
    { /* ISTORE */
      jint indx = (pc++)->jint;
      locals[indx].jint =
        stack[--stack_size].jint;
    }
    { /* ILOAD */
      jint indx = (pc++)->jint;
      stack[stack_size++].jint =
        locals[indx].jint;
    }
}

b)
{ /* Local stack slots. */
  jint __stack_1_jint ;
  jint __stack_0_jint ;
  /* No inputs. */
  { /* ILOAD */
    jint indx = (pc++)->jint;
    1*  __stack_0_jint = locals[indx].jint;
  }
  { /* ILOAD */
    jint indx = (pc++)->jint;
    2*  __stack_0_jint = locals[indx].jint;
  }
  { /* ISUB */
    jint value1 =
    3*  __stack_0_jint;
    jint value2 =
    4*  __stack_1_jint;
    5*  __stack_0_jint = value1 - value2;
  }
  { /* ISTORE */
    jint indx = (pc++)->jint;
    6*  locals[indx].jint = __stack_0_jint;
  }
  { /* ILOAD */
    jint indx = (pc++)->jint;
    7*  __stack_0_jint = locals[indx].jint;
  }
  /* Store outputs. */
  stack[stack_size + 0].jint =
    __stack_0_jint;
  stack_size += 1;
}

```

Stack location	Stack accesses timeline						
	1*	2*	3*	4*	5*	6*	7*
1		Write(opt)		Read(opt)			
0	Write		Read		Write	Read	Write

Figure 6.5: Source code optimization for stack accesses. One write and one read are eliminated. Others can be optimized by a regular C compiler. An actual *hot* superinstruction used by SPEC.compress benchmark.

6.4. Specialized optimization

a)

Stack location	ALOAD + ILOAD + FCONST_0 + FASTORE					
	Stack accesses timeline					
	1	2	3	4	5	6
2			W(3)	R(3)		
1		W(2)			R(3)	
0	W(1)					R(3)

b)

Stack location	ALOAD_3+ILOAD+ALOAD_3+ILOAD+FALOAD+FNEG+FASTORE+IINC											
	Stack accesses timeline											
	1	2	3	4	5	6	7	8	9	10	11	12
3				W(4)	R(4)							
2			W			R	W	R	W(3)	R(3)		
1		W(2)									R(2)	
0	W(1)											R(1)

Table 6.1: Two cases of stack access optimization from SPEC-mpegaudio benchmark. a) Removes 3 reads and 3 writes to the stack. b) removes 4 reads and 4 writes to the stack. Others can be optimized by a regular C compiler. Removed read-write pairs are marked with an index number for each pair.

code, as can be seen in Figure 6.5.

For the purpose of describing the modifications done to the source code we introduce the following definitions. A temporary stack value is one that does not exist on stack neither before nor after superinstruction is executed, but is only used internally within superinstruction. An input stack value is one that existed before superinstruction execution and is used (read) by superinstruction code. An output stack value is one that is modified (written to) by superinstruction and is left on stack after the execution of superinstruction is complete.

The source code of each superinstruction is modified in the following ways:

- New local variables are declared, one for each data type (int, float, etc.) of an input, output or temporary stack location, e.g. see *Local stack slots* section in Figure 6.5b.
- Stack accesses (both reads and writes) are replaced with accesses to these local variables, e.g. in Figure 6.5 in the first *ILOAD* section we see how `stack[stack_size++].jint` access is replaced with `__stack_0_jint` based on accessed data type and stack location 0 for the access number `1*`, as seen in the table at the bottom of the Figure.
- Code is added at the head of superinstruction to read input stack locations into appropriate local variables (the example code in Figure 6.5 has no stack inputs).
- Code is added at the tail of superinstruction to write values of appropriate local variables into output stack locations, e.g. see section *Store outputs* in Figure 6.5b, where `__stack_0_jint` variable is stored into the top location of the stack.

With data flow modified in this way a C compiler is able to avoid unnecessary read and write operations on the stack. The stack (memory) operations that can be avoided are marked as (*opt*) in Figure 6.5, and with numbers (1), (2), ... in Table 6.1. Some of the reads and writes are not marked because the compiler had all the data to optimize them out even without our optimizations to the source. This is mainly because only the last write to a stack location has to actually be carried out before the end of a chunk.

We applied this technique to a Java VM only but we shall note that we expect OCaml to be much less amenable to this technique. While both Java and OCaml bytecode instructions use a stack-based machine the set of instructions in OCaml allows for direct access not only to the top element of the stack, but basically allows for random access to several top elements of the stack. This makes the actual stack use patterns in OCaml much different to those of Java, in particular lessening the need for oft-repeated stack push and pop operations in OCaml.

6.5 Compilation of the optimized VM

The final step where the partial source code is used is the actual compilation of a specialized virtual machine. Files with specialized, partial source code are stored in a separate directory, uniquely marked by a prefix based on identification of the program that virtual machine was executing. A formerly enqueued compilation request waits until the compilation server is available. We decided to limit the number of compilations to 1, but there is nothing inherent in our design that would prevent raising this number to allow for simultaneous compilations. The reason for choosing to have a single compilation at a time is that compilation is a highly CPU- and cache-intensive process and on a single-CPU machine, especially running other applications at the same time, increasing the number of concurrent GCC instances could very easily severely decrease the overall performance of the system.

Once the server is ready to handle a compilation request the files with partial source code generated for a specific benchmark on a specific VM are copied into the directory containing complete VM sources. The sources of each VM were modified so as to compile properly and produce a non-specialized, *vanilla* VM (identical to the original code-copying VM) if these additional files with specialized source are empty. Once the files with specialized code are in place the virtual machine engine is recompiled and the resulting binary is placed in the server cache, next to the specialized partial sources from which it was generated. Note that by VM compilation we mean that only the VM engine is recompiled, as there is no need to recompile language libraries and other elements that may come as parts of a complete execution environment.

The process consisting of profiling the application, generating specialized partial code of superinstructions based on the profile, optional optimizing the source code and finally compilation of an optimized VM is now complete. The optimized VM binary is available to any virtual machine instance that will attempt to find a binary optimized for the same application.

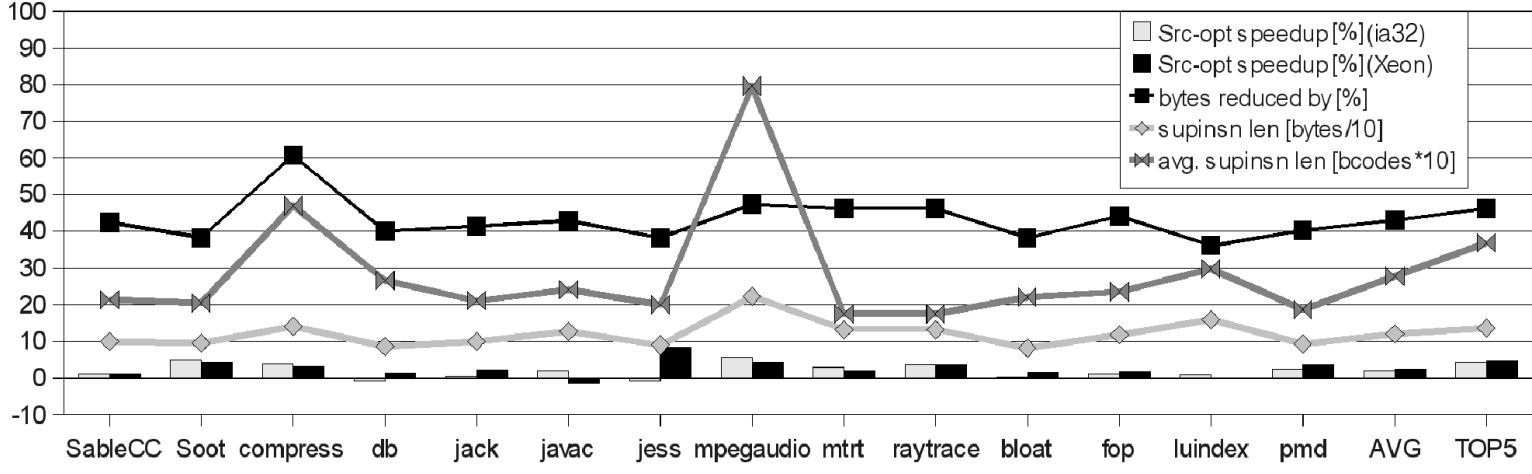
6.6 Performance results and metrics

For the purpose of testing our expectations of achieving performance improvement by using source-optimized superinstructions we extended two virtual machines, SableVM and OCaml to cooperate with our compilation server. To validate our design we performed a series of performance tests using two machines *ia32* and *xeon* (described in Chapter 2, Section 2.7 as Intel P4 3GHz and Intel Xeon 2.4GHz, respectively) to run the extended SableVM and OCaml. We executed each benchmark 10 times and took the average runtime as the final measured value. The standard deviation of runtimes for all benchmarks did not exceed 0.15 of the measured value.

The results presented in Figure 6.6 and Figure 6.7 demonstrate that the performance improvement brought by our technique of source-optimized superinstructions differs very significantly between languages and virtual machines. The performance of Java interpreter (SableVM) was slightly increased by about 2.0% and 2.5% on average, depending on the machine. For the top-5 benchmarks (Soot, compress, jess, mpegaudio and raytrace) the performance improved by a 4-5% average. Individual benchmarks performance improved by up to about 8%. The performance of OCaml interpreter increased dramatically by 27% on average, with performance of individual benchmarks improved by up to 81%.

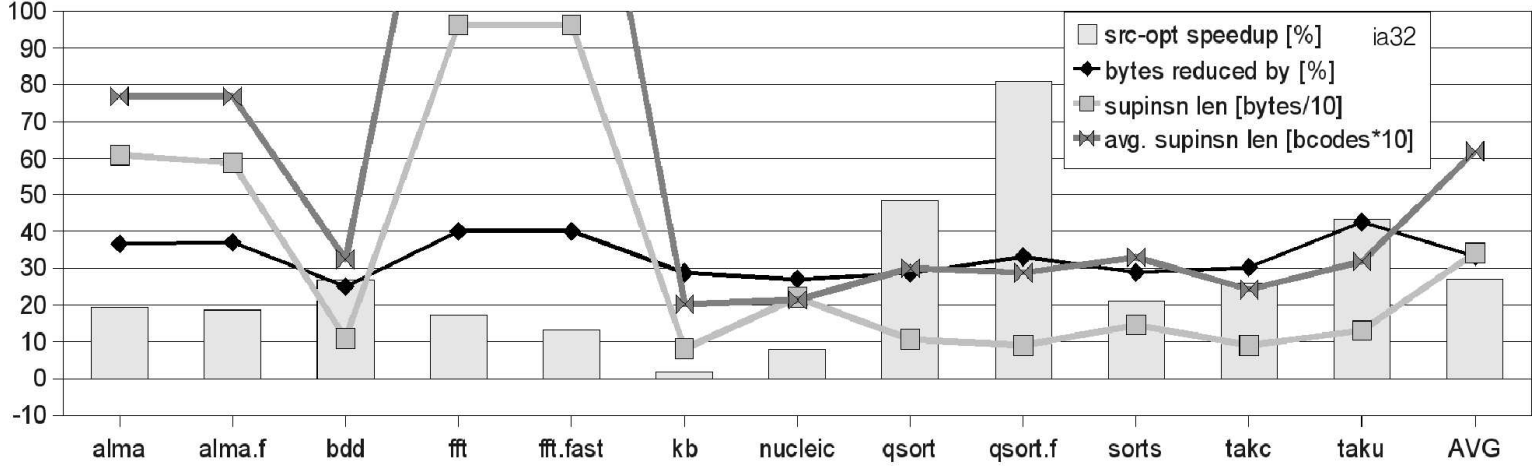
As in the previous stages of our work presented in this thesis we looked at possible metrics that would explain the difference in results for different benchmarks and virtual machines. We employed several metrics comparing code-copying and source-optimized superinstructions: superinstruction binary code length reduction factor, superinstruction length in bytes, average superinstruction length in bytecodes (weighted by executions of each superinstruction), and the number of created source-optimized superinstructions. We also looked at the data gathered from hardware counters during experiments described in the previous chapter.

Even without using more sophisticated tools it is clear there is little correlation between the values provided by metrics and the performance results. Let us attempt to look at Figure 6.7 and view the number of bytecodes in a superinstruction as a predictor of speedup, following the reasoning that a longer chunk of source code can



	SableCC	Soot	compress	db	jack	javac	jess	mpegaudio	mtrt	raytrace	bloat	fop	luindex	pmd	AVG	TOP5
Src-opt speedup (ia32)	1.1	5.0	3.9	-0.7	0.3	1.9	-0.8	5.6	2.8	3.6	0.0	1.1	0.8	2.5	1.9	4.2
Src-opt speedup (Xeon)	1.0	4.2	3.0	1.2	2.2	-1.5	8.4	4.2	1.8	3.4	1.4	1.7	-0.1	3.5	2.5	4.6
bytes reduced by [%]	42.4	38.3	60.7	40.0	41.4	42.9	38.2	47.3	46.2	46.2	38.3	44.1	36.1	40.2	43.0	46.1
supinsn len [bytes]	99	94	140	85	99	126	89	222	132	132	81	118	158	92	119	135
avg. supinsn len [bcodes]	2.1	2.0	4.7	2.7	2.1	2.4	2.0	8.0	1.7	1.7	2.2	2.3	3.0	1.9	2.8	3.7
src-opt supinsns count	133	215	10	16	118	206	57	65	112	112	129	228	138	163	122	91.8

Figure 6.6: Summary of the results achieved with the compilation server by SableVM (Java). Y-axis values have been rescaled so as to allow for comparison of several metrics.



	alma	alma.fast	bdd	fft	fft.fast	kb	nucleic	qsort	qsort.fast	sorts	takc	taku	AVG
src-opt speedup [%]	19.3	18.7	26.9	17.2	13.2	1.7	7.8	48.4	80.9	21.0	26.0	43.3	27.0
bytes reduced by [%]	36.7	37.0	25.1	40.1	40.1	28.8	27.0	28.6	33.1	28.9	30.2	42.5	33.2
supinsn len [bytes]	609	587	108	962	962	81	220	107	91	146	91	130	34.1
avg. supinsn len [bcodes]	7.7	7.7	3.2	18.3	18.3	2.0	2.1	3.0	2.9	3.3	2.4	3.2	6.2
src-opt supinsns count	43	43	45	23	23	60	71	14	19	106	5	5	38.1

Figure 6.7: Summary of the results achieved with the compilation server by OCaml VM. Y-axis values have been rescaled so as to allow for comparison of several metrics.

be better optimized. We find the evidence to the contrary, for example, by comparing the results and metrics for `fft` and `fft.fast` with `quicksort` and `quicksort.fast` benchmarks. The first pair (Fast Fourier Transform) has an average executed superinstruction lengths of about 18 bytecodes amounting to almost 1kB of binary code but resulting in only a moderate speedup of 13% and 17%. The second pair (Quicksort) has an average executed superinstruction lengths of only about 3 bytecodes amounting to only about 100 bytes of binary code that result in extraordinary speedups of 48% and 81%. Similar examples contradicting the expected correlations are found for all presented metrics. These observations suggest that a more thorough approach is warranted and a calculation of an actual correlation coefficient is necessary.

Table 6.2 contains correlation coefficients calculated between the speedups achieved using source-optimized superinstructions in each of the benchmarks and a value of each of the metrics. The correlations presented were calculated using Spearman's rank correlation coefficient formula [rcc]. The results from applying this formula can be explained in the following manner. Possible values range from -1.0 to 1.0, where 0.0 means no correlation, 1.0 means complete correlation and -1.0 means reverse correlation (e.g. if values in the base series increase and the values in the correlated series always decrease).

We have to report that overall we found no meaningful correlations between these metrics and performance improvements. In a few cases we found correlations for a *specific* architecture and VM. For SableVM on ia32 (P4 machine) the performance seems to be correlated to some degree with the reduction in superinstructions code length (coeff. 0.68 – the more reduction the bigger the speedup), and the length of superinstructions in bytes (coeff. 0.63 – the longer the code in bytes the bigger the speedup). For OCaml on ia32 machine the performance seems to be somewhat correlated with the number of superinstructions constituting 99% of superinstruction executions (coeff -0.58 – the less superinstructions necessary to cover the 99% of executions the bigger the speedup). Most of the correlation coefficients, however, are much lower. In particular, a strong correlation for any particular metric would be visible on all or most architectures, which is not the case. In our opinion there is no substantial evidence that the performance is bound to any subset of the properties

6.6. Performance results and metrics

	OCaml (ia32)	SableVM (ia32)	SableVM (Xeon)
src-opt speedup	1.00	1.00	1.00
bytes reduced by [%]	-0.03	<u>0.68</u>	0.05
supinsn len [bytes]	<u>-0.41</u>	<u>0.63</u>	-0.05
avg. supinsn len [bcodes]	-0.07	0.03	<u>-0.37</u>
src-opt supinsns count	<u>-0.58</u>	0.09	<u>-0.26</u>

Table 6.2: Correlation between the speedup achieved by using source-optimized superinstructions and the values of selected metrics from Figures 6.6 and 6.7. If any global correlation actually existed it would be visible on all architectures (horizontally). Stronger correlations are marked with double underline. Weaker correlations are marked with single underline.

we measured.

Overall, the results of the above analysis and the evident lack of substantial correlation are the motivation for a deeper investigation of the reasons behind the observed results.

6.6.1 Generated code comparison

In our opinion the actual performance improvement is due to compiler optimizations that were enabled by concatenation of the source code. These new opportunities depend heavily on the actual instructions involved. Optimization opportunities differ between bytecodes for different virtual machines, hence the observed variance of results.

Figure 6.8 presents the assembly of a single superinstruction created in three different ways. The a) version is a superinstruction created by concatenating binary code created by GCC compiler using formerly described `#pragma copyable` extension to ensure the necessary properties of the code. The b) version is the assembly of the same superinstruction but this time created by using concatenated source and

compiled again with the enhanced GCC, still using `#pragma copyable`. The c) version is the same as b) but with `#pragma copyable` removed from the code. Besides the obvious and clearly visible difference in lengths of these three binary code chunks a deeper analysis reveals that the compiler was able to apply several optimizations that could not be used previously, or apply optimizations with better results.

The most important optimizations now applied by the C compiler were:

- load-store-load optimization – keeping more data in registers without writing them back to memory and loading back (several eliminated memory accesses are marked in Figure 6.8 with \odot symbol),
- common subexpression elimination – once a value is computed it is kept for reuse by the code that follows,
- low-level, machine-dependant optimizations – different (better) choice of CPU instructions (e.g. compare CPU instructions used by LTINT bytecode instruction to instructions used in b) or c) version in Figure 6.8),
- optimizations of common execution path – by basic blocks reordering, code reorganization and inversion of conditionals (e.g. see jump to rarely executed code at `label_2` in Figure 6.8c marked with \otimes symbol).

We note that in the c) version the actual number of assembly instructions is almost the same as in b) but code that was expected to be less often executed was moved further away so as to avoid interference with the top-down control flow. Also note that because the c) version does not use `#pragma copyable` it can not be copied and executed elsewhere in memory. This limitation is largely unimportant because the superinstruction is already fully constructed and there is no reason to create copies or concatenations with other instructions. The c) version was the one used during the performance experiments while the b) version was the one used to compute metrics on binary code size. In our opinion the difference in optimization opportunities resulting from the source code of superinstructions and thus the quality of resulting binary code is the main cause for the observed differences in performance of different benchmarks and virtual machines.

```

a) - - - - ACC4:
   mov 0x10(%edi),%eax
   ◉mov %eax,0xfffffeb4(%ebp)
   - - - - PUSHACC3:
   ◉mov 0xfffffeb4(%ebp),%ecx
   mov %edi,%eax
   lea 0xffffffff(%edi),%edx
   mov %edx,%edi
   mov %ecx,0xffffffff(%eax)
   mov 0xc(%edx),%edx
   ◉mov %edx,0xfffffeb4(%ebp)
   - - - - LTINT:
   ◉mov 0xfffffeb4(%ebp),%eax
   cmp %eax,(%edi)
   setg %al
   add $0x4,%edi
   movzbl %al,%eax
   lea 0x1(%eax,%eax,1),%edx
   ◉mov %edx,0xfffffeb4(%ebp)
   - - - - BRANCHIFNOT:
   ◉cmpl $0x1,0xfffffeb4(%ebp)
   je <label_2>
   add $0x4,%esi
label_1:
   jmp <label_3>
label_2:
   mov %esi,%eax
   mov (%esi),%esi
   lea (%eax,%esi,4),%esi
   jmp <label_1>
   - - - - NEXT:
label_3:
   mov (%esi),%ecx
   add $0x4,%esi
   jmp *%ecx

b) mov 0x10(%edi),%eax
   lea 0xffffffff(%edi),%ecx
   mov %esi,%edx
   mov %eax,0xffffffff(%edi)
   cmp 0xc(%ecx),%eax
   jle <label_2>
   lea 0x10(%esi),%esi
   movl $0x3,0xfffffeb4(%ebp)

label_1:
   lea 0x4(%ecx),%edi
   jmp <label_3>

label_2:
   mov 0xc(%esi),%esi
   movl $0x1,0xfffffeb4(%ebp)
   lea 0xc(%edx,%esi,4),%esi
   jmp <label_1>

label_3:
   mov (%esi),%ecx
   add $0x4,%esi
   jmp *%ecx

c) mov 0x10(%edi),%eax
   lea 0xffffffff(%edi),%ecx
   mov %esi,%edx
   mov %eax,0xffffffff(%edi)
   cmp 0xc(%ecx),%eax
   ⊗jle <label_2 (outside)>
   lea 0x10(%esi),%esi
   movl $0x3,0xfffffeb4(%ebp)

label_1:
   lea 0x4(%ecx),%edi

label_3:
   mov (%esi),%ecx
   add $0x4,%esi
   jmp *%ecx

```

Figure 6.8: Assembly of an OCaml superinstruction (ACC4 + PUSHACC3 + LTINT + BRANCHIFNOT) constructed by a) code-copying, b) sources concatenation c) sources concatenation with no #pragmas.

6.6. Performance results and metrics

nsieve	OCaml [s]	Java [s]	relative speed OCaml/Java
direct-threaded	13.5	7.24	0.54
code-copying	4.12	2.38	0.58
src-optimized	3.49	1.85	0.53

Table 6.3: Comparison of the relative performance of a single benchmark (nsieve) on both OCaml and Java VMs and different execution engines.

	SableVM	SableVM-top5	OCaml
Compilation [s]	73.4	73.4	88.3
Speedup [%]	2.5	4.6	27.1
Break-even runtime [min]	49.9	26.4	5.4

Table 6.4: Compilation times (overhead), average speedups and resulting break-even runtime of optimized VMs.

6.6.2 Quick Comparisons of VMs Performance

Experimentation with two VMs of different design but using the same interpretation techniques is a good opportunity for comparison of performance also across different VMs. We used two implementations of *nsieve* benchmark, one for Java and one for OCaml to solve the same task. We measured their performance on different execution engines for both virtual machines and present the results in Table 6.3. While we do not claim that a single-benchmark comparison allows us to draw final conclusions our results do demonstrate that with different execution engines the relative speed of this benchmark implemented in Java and OCaml remained similar, with Java being almost twice as fast in all 3 evaluated cases. This suggests that Java bytecode is more efficient as an input to an interpreter than OCaml bytecode is.

6.6.3 Compilation Overhead and Break-even

Using a static compiler to recompile a complete VM engine is a relatively expensive task even though our system uses a cache to minimize the number of compilations. It is therefore important to determine what actual gains can be expected. Taking into account the compilation overhead and the resulting speedup we computed basic *break-even* expectations that are found in Table 6.4. SableVM, on average, would need to accumulate a total runtime of an application of almost one hour (or less than 30 minutes for the top-5 benchmarks) to start gaining on the investment in compilation. OCaml would start benefiting from source-optimized superinstructions after only about 5 minutes of accumulated runtime of an application. It shall be noted that we refer to the *accumulated runtime*, meaning the total runtime an application executed on one or more instances of an optimized VM. These times may seem substantial but are expected because of the use of highly-optimizing static C compiler (GCC). The existence of optimized binaries cache helps offset these long necessary runtimes by allowing the gains to accumulate over a longer period of time, across multiple executions of the same program.

6.7 Notes on related work

The design of work presented in this chapter relies on our advanced safe code-copying technique and previous work on its application to interpreters discussed in Chapters 4 and 5.

Other have investigated the application of source-optimized superinstructions, or, in general, optimizing sources created from lists of bytecodes. In the system proposed by Varma [VB04], for example, Java bytecode is translated into custom-generated C code including Java-specific optimizations and then compiled using a standard C compiler. Ertl created a system known as Vmgen [EGKP02] that generates source for superinstructions based on an execution profile, which is an analogous idea to our source-optimized superinstructions but differs from our system which provides efficient code reuse and caching costs. In comparison to Varma's work our system

is only concerned with generating C source code for small groups of bytecodes, not for whole applications. One of differences between our work and both Varma's and Ertl's work is that our system offers a *hands-free operation* without user intervention (although one can imagine Ertl's system as a basis for a hands-free implementation similar to ours). Our system is centered around a compilation server working in conjunction with a cache system and supporting multiple virtual machines for different languages. In our system there is also no need to choose a training set of applications since every application is optimized separately. Both our system and Vmgen are able to perform source-optimization of stack accesses, and while Vmgen expects the stack operations to be defined in a simple, yet specialized language, our Java bytecode source optimizer understands a subset of C. We also pursued a different path of analysis for determining the performance improvements brought by source-optimized superinstructions.

Our design for a VM-based compilation server makes use of code cache for resource sharing with later invocations of an application. There are many works in the area, here we list the main approaches and how they differ from ours. In several solutions we find a powerful machine used as a compilation server for mobile devices or devices with limited resources, e.g. Franz et al. [Fra97] or Palm et al. [PLDM02]. In a similar approach Lee et al. [LDM04] presented a compilation server based on JikesRVM JIT. There were attempts at using code caching and static ahead-of-time compilation to improve startup time of IBM JVM [MP08]. These approaches worked on larger codebases, optimizing parts of complete applications, as opposed to our approach where the basic source-optimization unit is a superinstruction. Also they did not use C code as their intermediate representation and did not employ a static but JIT compiler. Joisha et al. [JMSG01] modified a VM to share the binary executable code among multiple VMs to reduce the amount of writable memory. In the area of distributed compilation and caching compiler servers two prominent solutions exist known as CCache [CCa] and DistCC [Dis]. Their focus is on improving the compilation process itself either by distributing compilation or caching parts of the compiled works for future reuse and as such are complementary to our work and could potentially be used in place of the compiler in our server solution. In our case the

sharing takes place on disk and is done to avoid recompilation of the same source code, and to spread the substantial overhead of compilation with a static compiler over multiple runs of an application.

Overall, in contrast with our work, other solutions were focused on reusing existing, highly optimizing and specialized JITs, or translating and optimizing larger bodies of code, or focused on distribution and caching of the compilation process itself. They targeted other environments, or attempted to apply other optimization techniques, not necessarily taking into account reusability of the solution, or the implementation and maintenance costs. We summarize the properties of our solution in the next section.

6.8 Conclusions and future directions

In this chapter we presented a source-optimized superinstructions technique applied to Java and OCaml interpreters. Our solution used a static compiler, GCC, as the main optimization tool thereby ensuring portability and avoiding costly development of a JIT. While our implementation used a code-copying-enabled VM as the basis, the technique itself is, in fact, *independent of code-copying* and can be implemented without the specialized compiled support required for safe code-copying. The performance improvement over previously evaluated purely code-copying interpreters is most visible in the case of OCaml, with speedup of 27% on average, and 81% maximum. Java interpreter, SableVM, shows smaller benefits of 2-2.5% on average, with the top-5 benchmarks improving by 4-5%, and 8% maximum. We attribute the great improvements in OCaml to enabling better optimization opportunities for the compiler. At the same time we shall note that even with this improvement the OCaml interpreter does not seem to beat Java bytecode interpretation performance in performing an identical task, although more evaluation points are necessary to make this statement conclusive.

Our work can be further extended in several directions. An interpreter able to perform an on-stack-replacement could be used so that the optimized interpreter

6.8. Conclusions and future directions

function can be loaded at VM runtime. We also did not attempt to optimize GCC compilation flags which could be used to speed-up the compilation process and still regain the majority of performance improvement.

Overall, we achieved a very good performance improvement with this technique using relatively few resources and building upon existing solutions like GCC and the support for safe code-copying.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

While highly-optimizing compilers will always offer better performance than any interpreter-based solution there will also always be groups of language developers and users for whom such compilers are much too expensive to write and maintain. These groups are on the lookout for cheaper solutions offering the best possible performance at a lower cost. Our goal in this work has been to develop and improve methods for maximizing the performance of virtual machines in ways that require relatively little effort from a VM programmer.

Part of this goal can be achieved by employing solutions that improve the hardware-software interplay. We have added compiler support for source-specified optimization barriers solving the long-standing problem of safety in code-copying virtual machine design. Our low-maintenance approach to this important safety problem offers a good trade-off between performance and development costs and allows CPU branch predictors to work with the VM design instead of against it.

In the first milestone of our work we designed a C/C++ language extension that allows programmers to express the need for special safety guarantees of code-copying. We implemented this extension in a highly-optimizing, industry-standard GNU C Compiler (GCC). Despite the unusual nature of our changes we managed to follow

the best compiler design practices and ensure long-term maintainability of our modifications within the GCC framework (see Section 4.6 for details). This makes it more likely that the extension will gain a wider industry support in the future.

In the second milestone of our work we implemented safe code-copying in Java, OCaml, and Ruby to practically demonstrate the applicability of our compiler extension and performance gains delivered by code-copying. To comprehensively test and analyze the performance and the issues of software-hardware interplay we gathered extensive data on 3 largely different architectures Intel 32-bit, x86_64, and PowerPC 64-bit. This shows the varying nature of the performance improvement with respect to architectures and the importance of considering both language-VM design and the hardware-software interplay issues. OCaml on Intel 32-bit was 2.81 times faster on average, and up to 4.8 times maximum. The performance of Ruby, however, improved only slightly by 1.03 to 1.14 average factor, depending on the architecture. Java improved significantly by an average factor of 1.32 on x86_64, and 1.44 on Intel 32-bit, but only 1.04 on PowerPC 64-bit. Our investigation included a detailed analysis of the causes of such vast performance differences and found several factors influencing it, most important among them being the average size of language bytecodes and the behavior of branch predictors on each hardware platform. We believe these results and the detailed conclusions our work contains (see Section 5.5) will provide guidance to developers evaluating the potential use of safe code-copying in their virtual machines.

In the third milestone of our work, to further improve the performance of our system, we implemented an ahead-of-time-based approach with a compilation server. This solution focuses on specializing interpreter source code to each application and optimizing source code of groups of bytecode instructions. By using a cache to store specialized VM binaries the overhead of compilation is spread over multiple runs of a virtual machine. With this system we achieved speedups of average 27% for OCaml and 4-5% on selected Java benchmarks *over the safe code-copying technique* (not direct-threading). The technique itself is in fact completely independent of code-copying and can be implemented as an alternative VM engine with no need for the specialized compiler support required for code-copying.

In summary, our work presents two largely different approaches, both offering great performance, and applicable to variety of virtual machines. Code-copying has the advantage of being 100% dynamic and adaptable, thus is suitable for nearly any VM, but to ensure execution safety requires specialized compiler support. Our AOT-based approach works with any compiler, but requires on-disk cache and is most suitable for often-run or long-running applications. Both solutions offer performance far better than the industry standard direct-threading, and the choice depends on particularities of the system they will enhance.

7.2 Future Work

The work presented in this thesis can be further extended in several directions. It would be advantageous to extend the safe code-copying support we implemented in GNU C Compiler (GCC) to more architectures and ensure the design generalizes. Also, the mechanism used within GCC to implement the special guarantees for code-copying demonstrates a more general approach to providing a variety of localized changes to the generated code. Other uses for such an approach include specialized optimization barriers, turning selected optimizations on and off for selected blocks of code, disabling optimizations for the sections of code that will be debugged while keeping the rest highly-optimized, etc. Such features are not currently available in GCC because they seem difficult to create and maintain but our approach to code-copying can be used as the basis for their implementation.

While we gathered extensive performance results, software and hardware behavior metrics for multiple VMs and hardware platforms evaluation of performance on current CPUs is non-trivial and certain behaviors observed could still use a better explanation. We have attributed the disappointing improvement on the PowerPC platform to branch predictor design, but fuller detail on the specific branch prediction features and operation, and how they relate to our design would give a more complete explanation of performance. Such detail, unfortunately, is not readily available for commercial and proprietary CPUs.

Currently, in our AOT-based solution, when an optimized VM binary is created it can be used by the *next* invocation of VM while the currently executing instances cannot take advantage of the optimized version. This could be helped by performing on-stack-replacement (OSR) so that the optimized version of VM could be loaded at VM runtime. On-stack-replacement is a mechanism where the state of currently executing function (or method) can be saved and the execution state transferred to a new implementation of a function (or method) to continue its execution. In the case of an interpreter this would allow us to replace an interpreter engine, currently executing the function containing the main interpreter loop, with a new, optimized (specialized) one. Additionally, to further reduce the cost of compilation the GCC flags used could be optimized so as to only include the core optimizations and thus speed-up the compilation process while still regaining the majority of performance improvement.

Ruby VM (Yarv) contains many bytecodes that are currently unfit for code-copying due to their large sizes. We suspect that this situation could be improved by bytecode splitting and specialization. For example, more detailed traces could be used to track the behavior of bytecodes themselves and then, after analysis, to automatically produce simpler *specialized bytecodes* where necessary. In certain cases the existing bytecode can be transformed and specialized at load time, as is currently done in the Java interpreter we used.

We also hope that the gains from the use code-copying and the advantage provided by our GCC extensions will be recognized and the support necessary for this technique will be one day available in some of the major compilers.

Bibliography

- [AK02] Randy Allen and Ken Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers, 2002.
- [AR02] Matthew Arnold and Barbara G. Ryder. Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 498–524, London, UK, 2002. Springer-Verlag.
- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [B04] David Bélanger. SableJIT: A retargetable just-in-time compiler. Master’s thesis, McGill University, August 2004.
- [BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming Language Design and Implementation*, pages 1–12, New York, NY, USA, 2000. ACM Press.
- [BDF⁺03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and

- the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.
- [BGH⁺06] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.
- [BK08] Derek Bruening and Vladimir Kiriansky. Process-shared and persistent code caches. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 61–70, New York, NY, USA, 2008. ACM.
- [BKGB06] Derek Bruening, Vladimir Kiriansky, Timothy Garnett, and Sanjeev Banerji. Thread-shared software code caches. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 28–38, Washington, DC, USA, 2006. IEEE Computer Society.
- [Bot03] Per Bothner. Compiling Java with GCJ. *Linux Journal*, 2003(105):4, 2003.
- [BVZB05] Marc Berndl, Benjamin Vitale, Mathew Zaleski, and Angela Demke Brown. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. *Code Generation and Optimization (CGO), IEEE/ACM International Symposium on*, 0:15–26, 2005.
- [CCa] CCache. <http://ccache.samba.org>.
- [CLLM04] C. Consel, J.L. Lawall, and A.-F. Le Meur. A tour of Tempo: A program specializer for the C language. *Science of Computer Programming*, 2004.

- [CLQ06] Giacomo Cabri, Letizia Leonardi, and Raffaele Quitadamo. Enabling Java mobile computing on the IBM Jikes research virtual machine. In *PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java*, pages 62–71, New York, NY, USA, 2006. ACM Press.
- [Deb] Debian Shootout. <http://shootout.alioth.debian.org/>.
- [Dis] DistCC. <http://www.distcc.org>.
- [EG01] M. Anton Ertl and David Gregg. The behavior of efficient virtual machine interpreters on modern architectures. In *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 403–412, London, UK, 2001. Springer-Verlag.
- [EG03a] M. Ertl and D. Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25, 2003.
- [EG03b] M. Anton Ertl and David Gregg. Implementation issues for superinstructions in Gforth. In *EuroForth 2003 Conference Proceedings*, 2003.
- [EG03c] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *SIGPLAN '03 Conference on Programming Language Design and Implementation*, 2003.
- [EG04] M. Anton Ertl and David Gregg. Retargeting JIT compilers by using C-compiler generated executable code. In *Parallel Architecture and Compilation Techniques (PACT' 04)*, pages 41–50, 2004.
- [EGKP02] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. Vmgen: a generator of efficient virtual machine interpreters. *Software-Practice and Experience*, 32(3):265–294, 2002.
- [Ert95] M. Anton Ertl. Stack caching for interpreters. *SIGPLAN Not.*, 30(6):315–327, 1995.

- [ETK06] M. Anton Ertl, Christian Thalinger, and Andreas Krall. Superinstructions and replication in the Cacao JVM interpreter. *Journal of .NET Technologies*, 4:25–32, 2006.
- [Fra97] M. Franz. Run-time code generation as a central system service. In *HOTOS '97: Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, page 112, Washington, DC, USA, 1997. IEEE Computer Society.
- [FSF06] FSF. *GNU Compiler Collection Internals*. GNU Project, 2006.
- [Gag02] Etienne M. Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, McGill University, 2002.
- [Gag03] Etienne M. Gagnon. Porting and tuning inline-threaded interpreters. In *CASCON 2003 workshop reports*, 2003.
- [GCJ] GCJ. <http://gcc.gnu.org/java/>.
- [GH98] Etienne M. Gagnon and Laurie J. Hendren. SableCC, an object-oriented compiler framework. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 140, Washington, DC, USA, 1998. IEEE Computer Society.
- [GH01] Etienne Gagnon and Laurie Hendren. SableVM: A research framework for the efficient execution of Java bytecode. In *Java Virtual Machine Research and Technology Symposium*, 2001.
- [GH03] Etienne Gagnon and Laurie Hendren. Effective inline-threaded interpretation of Java bytecode using preparation sequences. In *Compiler Construction, 12th International Conference*, 2003.
- [GPM⁺04] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. A retrospective on: “an evaluation of staged run-time optimizations in DyC”. *SIGPLAN Not.*, 39(4):656–669, 2004.

- [GVG04] Dayong Gu, Clark Verbrugge, and Etienne Gagnon. Code layout as a source of noise in JVM performance. In *Component And Middleware Performance workshop, OOPSLA 2004*, 2004.
- [GVG06] Dayong Gu, Clark Verbrugge, and Etienne M. Gagnon. Relative factors in performance analysis of Java virtual machines. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 111–121. ACM Press, 2006.
- [Hos06] Matthew E. Hoskins. User-mode linux. *Linux J.*, 2006(145):2, 2006.
- [IBM] IBM JIT. <http://publib.boulder.ibm.com/infocenter/javasdk/v1r4m2/index.jsp?topic=/com.ibm.java.doc.diagnostics.142/html/underthejvm.html>.
- [Jik] JikesRVM. <http://jikesrvm.org>.
- [JMSG01] Pramod G. Joisha, Samuel P. Midkiff, Mauricio J. Serrano, and Manish Gupta. A framework for efficient reuse of binary code in Java. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 440–453, New York, NY, USA, 2001. ACM Press.
- [Jyt] Jython. <http://www.jython.org>.
- [Kaf] Kaffe. <http://www.kaffe.org>.
- [KWM⁺08] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the java hotspotTMclient compiler for java 6. *ACM Trans. Archit. Code Optim.*, 5(1):1–32, 2008.
- [LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

- [LDM04] Han B. Lee, Amer Diwan, and Eliot B. Moss. Design, implementation, and evaluation of a compilation server. Technical Report CU-CS-978-04, University of Colorado, 2004.
- [LM03] Tom Lunney and Aidan McCaughey. Object persistence in Java. In *PPPJ '03: Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 115–120, New York, NY, USA, 2003. Computer Science Press, Inc.
- [MK00] Geetha Manjunath and Venkatesh Krishnan. A small hybrid JIT for embedded systems. *SIGPLAN Not.*, 35(4):44–50, 2000.
- [MP08] Kenneth Ma and Marius Pirvu. AOT compilation in a dynamic environment for startup time improvement. In *7th Workshop on Compiler-Driven Performance*, 2008.
- [Muc97] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, 1997.
- [MY01] Hidehiko Masuhara and Akinori Yonezawa. Run-time bytecode specialization. *Lecture Notes in Computer Science*, 2053, 2001.
- [NW02] Matt Newsome and Des Watson. Proxy compilation of dynamically loaded Java classes with MoJo. In *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 204–212, New York, NY, USA, 2002. ACM Press.
- [NW06] Maurice Naftalin and Philip Wadler. *Java Generics and Collections*. O'Reilly Media, Inc., 2006.
- [OCa] OCaml. <http://caml.inria.fr>.
- [OPr] OProfile. <http://oprofile.sf.net/>.
- [Par] ParrotVM. <http://www.parrotcode.org/>.

- [PBF⁺03] K. Palacz, J. Baker, C. Flack, C. Grothoff, H. Yamauchi, and J. Vitek. Engineering a customizable intermediate representation. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 67–76, New York, NY, USA, 2003. ACM.
- [PGA07] Gregory B. Prokopski, Etienne M. Gagnon, and Christian Arcand. Byte-code testing framework for SableVM code-copying engine. Technical Report SABLE-TR-2007-9, Sable Research Group, School of Computer Science, McGill University, Montréal, Québec, Canada, September 2007.
- [PLDM02] Jeffrey Palm, Han Lee, Amer Diwan, and J. Eliot B. Moss. When to use a compilation service? In *LCTES/SCOPEs '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 194–203, New York, NY, USA, 2002. ACM Press.
- [Ple] Plex86. <http://plex86.sourceforge.net/>.
- [PQVR⁺00] Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, and Clark Verbrugge. A framework for optimizing java using attributes. In *CASCON '00: Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, page 8. IBM Press, 2000.
- [PQVR⁺01] Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, and Clark Verbrugge. A framework for optimizing Java using attributes. In Reinhard Wilhelm, editor, *Proceedings of the 10th International Conference on Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science (LNCS)*, pages 334–354, April 2001.
- [PR98] Ian Piumarta and Fabio Ricciardi. Optimizing direct threaded code by selective inlining. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 291–300, New York, NY, USA, 1998. ACM Press.
- [PV07] Gregory B. Prokopski and Clark Verbrugge. Towards GCC as a compiler for multiple VMs. In *GCC Developers' Summit, 2007*.

Bibliography

- [PV08a] Gregory B. Prokopski and Clark Verbrugge. Analyzing the performance of code-copying virtual machines. In *OOPSLA 2008*, pages 403–422, 2008.
- [PV08b] Gregory B. Prokopski and Clark Verbrugge. Compiler-guaranteed safety in code-copying virtual machines. In *Compiler Construction, 17th International Conference*, LNCS. Springer, 2008. to appear.
- [PWL04] Jinzhan Peng, Gansha Wu, and Guei-Yuan Lueh. Code sharing among states for stack-caching interpreter. In *IVME '04: Proceedings of the 2004 workshop on Interpreters, virtual machines and emulators*, pages 15–22, New York, NY, USA, 2004. ACM.
- [Raw] Raw results used for this publication. <http://www.sable.mcgill.ca/~gproko/gcc/multi-08-raw-results.pdf>.
- [rcc] Spearman's rank correlation coefficient. http://en.wikipedia.org/wiki/Spearman's_rank_correlation_coefficient.
- [RS96] Markku Rossi and Kengatharan Sivalingam. A survey of instruction dispatch techniques for byte-code interpreters. Technical Report TKO-C79, Faculty of Information Technology, Helsinki Univeristy of Technology, May 1996.
- [Rub] Ruby On Rails. <http://rubyonrails.org/>.
- [Sas05] Koichi Sasada. YARV: yet another RubyVM: innovating the Ruby interpreter. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 158–159, New York, NY, USA, 2005. ACM.
- [SGBE05] Yunhe Shi, David Gregg, Andrew Beatty, and M. Anton Ertl. Virtual machine showdown: Stack versus registers. In *Virtual Execution Environments (VEE '05)*, pages 153–163, 2005.

- [SH03] Ben Stephenson and Wade Holst. Multicodes: optimizing virtual machines using bytecode sequences. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 328–329, New York, NY, USA, 2003. ACM Press.
- [SOT⁺00] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time compiler. *IBM Systems Journal*, 39(1):175–193, 2000.
- [Sta] Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. <http://www.spec.org/jvm98>.
- [Sta04] Basile Starynkevitch. OCAMLJIT – a faster just-in-time OCaml implementation. In *Proceedings of MetaOCaml 2004 workshop*, 2004.
- [Sto06] Mark Stoodley. Avoiding live lock when patching code in real-time execution environments. In *CASCON '06: Proceedings of the 2006 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 2006.
- [Sun] Sun HotSpot. <http://java.sun.com/products/hotspot/whitepaper.html>.
- [SYK⁺05] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. Design and evaluation of dynamic optimizations for a Java just-in-time compiler. *ACM Trans. Program. Lang. Syst.*, 27(4):732–785, 2005.
- [SZ08] Yu Sun and Wei Zhang. Efficient code caching to improve performance and energy consumption for java applications. In *CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 119–126, New York, NY, USA, 2008. ACM.

- [TCL⁺00] S. Thibault, C. Consel, J. Lawall, R. Marlet, and G. Muller. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation*, 13(3):161–178, September 2000.
- [VA04] Benjamin Vitale and Tarek S. Abdelrahman. Catenation and specialization for tcl virtual machine performance. In *IVME '04: Proceedings of the 2004 workshop on Interpreters, virtual machines and emulators*, pages 42–50, New York, NY, USA, 2004. ACM.
- [VB04] Ankush Varma and Shuvra S. Bhattacharyya. Java-through-C compilation: An enabling technology for Java in embedded systems. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 30161, Washington, DC, USA, 2004. IEEE Computer Society.
- [VMW] VMWare. <http://www.vmware.com>.
- [VZ05] Benjamin Vitale and Mathew Zaleski. Alternative dispatch techniques for the tcl vm interpreter. In *Proceedings of 12th Annual Tcl/Tk Conference*, October 2005.
- [Wik] Wikipedia. <http://en.wikipedia.org/>.
- [ZBB05] Mathew Zaleski, Marc Berndl, and Angela Demke Brown. Mixed mode execution with context threading. In *CASCON '05: Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 305–319. IBM Press, 2005.
- [ZBS07] Mathew Zaleski, Angela Demke Brown, and Kevin Stoodley. Yeti: a gradually extensible trace interpreter. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 83–93, New York, NY, USA, 2007. ACM.
- [ZCS05] Shukang Zhou, Bruce R. Childers, and Mary Lou Soffa. Planning for code buffer management in distributed virtual execution environments. In

Bibliography

VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments, pages 100–109, New York, NY, USA, 2005. ACM.