LEABHARLANN CHOLÁISTE NA TRÍONÓIDE, BAILE ÁTHA CLIATH | TRINITY COLLEGE LIBRARY DUBLIN
Ollscoil Átha Cliath | The University of Dublin

Terms and Conditions of Use of Digitised Theses from Trinity College Library Dublin

Copyright statement

All material supplied by Trinity College Library is protected by copyright (under the Copyright and Related Rights Act, 2000 as amended) and other relevant Intellectual Property Rights. By accessing and using a Digitised Thesis from Trinity College Library you acknowledge that all Intellectual Property Rights in any Works supplied are the sole and exclusive property of the copyright and/or other IPR holder. Specific copyright holders may not be explicitly identified. Use of materials from other sources within a thesis should not be construed as a claim over them.

A non-exclusive, non-transferable licence is hereby granted to those using or reproducing, in whole or in part, the material for valid purposes, providing the copyright owners are acknowledged using the normal conventions. Where specific permission to use material is required, this is identified and such permission must be sought from the copyright holder or agency cited.

Liability statement

By using a Digitised Thesis, I accept that Trinity College Dublin bears no legal responsibility for the accuracy, legality or comprehensiveness of materials contained within the thesis, and that Trinity College Dublin accepts no liability for indirect, consequential, or incidental, damages or losses arising from use of the thesis for whatever reason. Information located in a thesis may be subject to specific use constraints, details of which may not be explicitly described. It is the responsibility of potential and actual users to be aware of such constraints and to abide by them. By making use of material from a digitised thesis, you accept these copyright and disclaimer provisions. Where it is brought to the attention of Trinity College Library that there may be a breach of copyright or other restraint, it is the policy to withdraw or take down access to a thesis while the issue is being resolved.

Access Agreement

By using a Digitised Thesis from Trinity College Library you are bound by the following Terms & Conditions. Please read them carefully.

I have read and I understand the following statement: All material supplied via a Digitised Thesis from Trinity College Library is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of a thesis is not permitted, except that material may be duplicated by you for your research use or for educational purposes in electronic or print form providing the copyright owners are acknowledged using the normal conventions. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone. This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

Data and Type Optimizations in Virtual Machine Interpreters

Ph.D. Thesis

June 3rd, 2010

Kevin Williams, B.A. (Mod.)

Declaration

This thesis has not been submitted as an exercise for a degree at this or any other university. It is entirely the candidates own work. The candidate agrees that the Library may lend or copy the thesis upon request. This permission covers only single copies t to normal conditions of acknowledgement.

6/10

LEGE
010
LIBRARY DUBLIN
THESIS
8937

Acknowledgements

I would like to thank my supervisor Dr. David Gregg for his support and tuition throughout my undergraduate and postgraduate studies. His knowledge, experience and input have been a crucial factor in the completion of this work.

I would also like to thank the members of my research group; Bob, Jason, Nicholas, Paul and Scott. Debates and discussions of all matters science, pseudo-science and pure nonsense have made my three years of study more enjoyable. My co-authors deserve much credit for their efforts and contributions to our collaborations.

I would like to thank Sarah for her patience and encouragement over the last three years, without it I may never have completed this work. Finally, I would like to thank my parents for the support they have given me through my many years of education, this work is dedicated to their efforts.

Abstract

Virtual machines provide a portable platform for the execution of programming languages. They are popular amongst language designers as they simplify the task of developing a language and its features. Programmers benefit from virtual machines as a single implementation of a program can be run on many platforms. Virtual machine interpreter implementations are commonly slower than an equivalent native implementation. The optimization of virtual machine interpreters is the goal of many virtual machine research projects. In this dissertation, we investigate optimizations for two different interpreters implementing the programming languages Java and Lua.

In our Java interpreter, we find that the fetching of virtual instructions and their operands is a bottleneck to performance. While for Lua, a dynamically typed language, we find the run-time resolution of variable types to be a bottleneck. We evaluate existing techniques and present new approaches to the efficient interpretation of these languages.

For our Java interpreter, we break the execution of virtual instructions into stages and pipeline their execution. We dramatically reduce both data stalls and virtual instruction dispatch stalls. We present experimental results that show our techniques to increase performance by up to 20%.

For our Lua interpreter, we implement a novel intermediate representation that is constructed and executed at run-time by our interpreter. The representation defines both the control-flow of the program and also the type of all local variables at every point in execution. All virtual instructions are therefore specialized to their operand types. For applications with many type checks, our interpreter produces a speedup of up to 3x over standard interpretation.

Contents

Acknowledgements i

Abstract ii

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 Thesis . . . 3
1.2 Contributions . . . 3
1.3 Relevant Academic Publications . . . 4
1.4 Dissertation Outline . . . 5

2 Background 6

2.1 Architecture Features . . . 6
2.1.1 Pipeline . . . 6
2.1.2 Superscalar and VLIW . . . 8
2.1.3 Branch Prediction . . . 8
2.1.4 Cache . . . 10
2.2 Virtual Machine Languages . . . 11
2.3 Java . . . 12
2.3.1 JavaVM . . . 12
2.3.2 CellVM . . . 15
2.4 Lua . . . 16
2.4.1 Lua Virtual Machine . . . 18
2.4.2 The Lua Instruction Set . . . 20

3 Literature Survey **23**
3.1 Interpreters 23
3.1.1 Investigating Interpreter Performance 24
3.1.2 Dispatch Techniques 25
3.1.3 Stack versus Register 26
3.1.4 Stack Caching 27
3.1.5 Interpreter Pipelining 28
3.1.6 Superinstructions 29
3.1.7 Instruction (De)Specialization 30
3.2 Run-Time Optimization 30
3.2.1 Functions, Loops and Traces 31
3.2.2 Self 32
3.2.3 Prolog 34
3.2.4 Scripting Language Specialization 35

4 A Cell BE Java Interpreter **40**
4.1 Introduction 40
4.2 Background 41
4.2.1 CellVM 41
4.3 Architectural Features 42
4.3.1 Local Store 42
4.3.2 Branch Predictors 43
4.3.3 Register Files 44
4.3.4 Dual Issue Pipeline 44
4.4 Design Considerations 44
4.5 Experimental Setup 45
4.6 Optimizations 46
4.6.1 Interpreter Pipelining 47
4.6.2 Operand Pipelining 47
4.6.3 JVM Branch Prediction 50
4.6.4 Top Of Stack Caching 52
4.6.5 Merged Operators 53
4.6.6 Superinstructions 57
4.7 Discussion 58

5 A Portable Mixed-Mode Lua virtual machine 59
5.1 Introduction . . . 59
5.1.1 Advantages of JITing with C . . . 60
5.1.2 Contributions . . . 60
5.2 Background . . . 61
5.2.1 Mixed-mode Interpretation & Dynamic Compilation . . . 61
5.2.2 Dynamic Typing . . . 61
5.2.3 Compilation Units . . . 62
5.3 Profiling Architecture . . . 62
5.3.1 Profile Signatures . . . 63
5.3.2 VM Execution Flow . . . 63
5.4 Data Flow Analysis . . . 65
5.4.1 Flow Graph . . . 65
5.4.2 Type Inference . . . 65
5.4.3 Library Function Analysis . . . 66
5.5 Optimizations . . . 66
5.5.1 Control Flow and Interpreter Overhead . . . 67
5.5.2 Type Check Removal . . . 67
5.5.3 Lua Variables to Primitive C Variables . . . 68
5.5.4 Library Inlining . . . 69
5.5.5 Lua Inlining . . . 69
5.6 Experimental Setup . . . 70
5.7 Experimental Evaluation . . . 70
5.7.1 Micro Benchmarks . . . 70
5.7.2 Benchmark Introduction . . . 72
5.7.3 Type Inference Performance . . . 73
5.7.4 Performance . . . 74
5.8 Related Work . . . 77
5.9 Discussion . . . 77

6 A Type-Specialized Lua Interpreter 80
6.1 Introduction . . . 80
6.2 Dynamic Intermediate Representation . . . 81
6.2.1 Standard Node . . . 82
6.2.2 Conditional Node . . . 82
6.2.3 Type-Directed Node . . . 83

6.2.4 Call Node 83
6.2.5 Return Node 84
6.2.6 Instruction Specialization 85
6.3 Motivating Example 86
6.3.1 SIR 86
6.3.2 DIR 88
6.4 Experimental Evaluation 90
6.4.1 Experimental Setup 90
6.4.2 Machine Instructions Issued 90
6.4.3 Branch Instructions 90
6.4.4 Cache Access 93
6.4.5 Processor Stalls 95
6.4.6 Memory 95
6.4.7 Performance 98
6.5 Scope for Optimization 100
6.6 General Applicability 104
6.7 Related Work 105
6.8 Discussion 106

7 Final Thoughts **107**
7.1 Thread-level Interpreter Optimization 107
7.2 Scripting Language Architecture 108
7.2.1 A New Approach 110
7.3 A Lua 6 Proposal 112
7.4 Conclusions 114

Bibliography **116**

List of Tables

2.1 Lua 5 instruction format 18
2.2 Lua 5 instruction set 22

5.1 Table of Lua opcodes grouped by type. 72

6.1 Intermediate representation memory usage 97

7.1 Scripting Language Implementations 108
7.2 The Lua5 versus Lua6 basic types 113
7.3 The Lua 5 versus *'Lua 6'* instruction format 113
7.4 The *'Lua 6'* specialized instruction set 120

List of Figures

2.1 Example processor pipeline 7
2.2 State machines for dynamic branch predictors 9
2.3 Classic Memory Hierarchy 10
2.4 JVM Architecture 13
2.5 CellVM Architecture 16
2.6 Structures representing Lua's 'tagged values' 19
2.7 Structures representing Lua's Tables 20

3.1 A comparison of sequential and pipelined interpreter execution 28

4.1 Pipeline schedule for virtual instruction IADD 42
4.2 PPE interpreter versus SPE interpreter running times 45
4.3 Pipeline schedule for virtual instruction IADD with PC fetch pipelining 46
4.4 Pipelining performance increases 49
4.5 JVM Branch Prediction Evaluation 51
4.6 Stack caching performance increases 53
4.7 Pipeline schedule for virtual instruction IADD with stack caching of one variable 54
4.8 Superinstruction and Instruction merging performance increases 55
4.9 Pipeline schedule for merged-instruction IADD_ISUB with program counter and operand pipelining 56

5.1 Fields and structure of profile signatures 63
5.2 A flowchart showing the flow of execution between JIT and profile-interpreting execution 64
5.3 A comparison of run-time performance. 71
5.4 Frequency of Lua opcodes use per benchmark 73
5.5 Relative opcode overhead for opcode groups 74
5.6 Total computation overhead for virtual machine components 75

5.7 Run-time operand type knowledge for Lua operands as a result of our type inference 75
5.8 Total number of type check removals achieved as a result of our code specialization 76
5.9 Source of unknown operand types from our type inference algorithm . . 76
5.10 A comparison of run-time performance. 78

6.1 Node structures and interpreter dispatch techniques for standard nodes 82
6.2 Node structures and interpreter dispatch techniques for conditional nodes 82
6.3 Node structures and interpreter dispatch techniques for type-directed nodes 83
6.4 Node structures and interpreter dispatch techniques for call nodes . . . 84
6.5 A Lua implementation of the Sieve of Eratosthenes algorithm. 86
6.6 An *'imaginary'* control flow graph for the static interpretation of the Sieve of Eratosthenes algorithm. 87
6.7 The Sieve of Eratosthenes dynamic representation built and executed by our interpreter. 89
6.8 Relative measure of instructions issued 91
6.9 Relative measure of conditional branch instructions executed 91
6.10 Indirect branches executed and mis-predicted 92
6.11 Relative measure of level 1 data and instruction cache accesses 94
6.12 Relative measure of cycles stalled on any resource 95
6.13 Relative measure of Level 2 data cache misses. 96
6.14 Relative measure of Level 2 data cache writes. 96
6.15 Relative measure of Level 2 instruction cache misses. 98
6.16 A comparison of micro benchmark run-time performance 99
6.17 A comparison of application benchmark run-time performance 101
6.18 Branch behavior for various dispatch techniques 102

7.1 A new scripting language pipeline 111

Chapter 1

Introduction

The requirements of modern computing applications demand a diverse set of processing abilities. Increasingly, processors and languages are being designed with particular classes of application in mind.

The unprecedented growth in web technology and web applications has resulted in a demand for programming languages that are expressive, portable and accessible to non-experts. The dynamic features of scripting languages like Python, PHP, Perl, JavaScript, Ruby and Lua [Lut09; Hud09; LWO08; Com99; FM08; Ier06] make them ideal targets for less skilled programmers. These languages remove many of the run-time concerns programmers have when developing in traditional system languages such as C, C++ and Fortran [Sch90; Str03; ABM+97]. The dynamic type systems of these languages encourage fast prototyping that allow programmers to write powerful disposable scripts. These languages are also extremely portable. Scripted applications are typically distributed as source files which are executed by a client virtual machine. This portability is crucial for the new breed of web based applications like Facebook, Twitter, GMail and Google Maps, to name but a few.

Scripting languages can be used to execute small tasks, create applications or extend a larger system written in a traditional system language. The games industry is a leader in the extensible approach to application development. Commonly, game developers will write the core processing of their game in C or C++. They then use a scripting language to *'glue'* these core components together. This approach speeds up the production and development of large games. The most popular scripting language in the games industry is the Lua scripting language.

The down side of Lua and all other scripting languages is their poor performance compared to static natively compiled alternatives. In this dissertation, we attempt to tackle the issue of scripting language performance by profiling, at run-time, the dynamic

features of a scripted program. We use these profiles to make improvements through just-in-time compilation (in Chapter 5) and specialized interpretation (in Chapter 6).

The efficiencies of portable languages is not just dependent on the features of the language but also on the hardware they execute on. Consumer demand for ever improving multi-media processing have placed large demands on computing hardware. The effects of Moore's Law have made traditional single core computing less and less feasible [DeB04]. The solution to this hardware crisis is the development of computing processors with multiple processing *'cores'*. These multi-core machines place more power in a single chip but also introduce the complex problem of multi-process development into everyday computing problems.

An example multi-core processor is the Cell Broadband Engine (Cell BE)[KDH+05]. The Cell BE is the processor in the Playstation 3 games console. It provides one general purpose core which runs the operating system and management processes. This main core is extended by eight sub-cores which are specialized for the execution of vector rich computing problems, as found in games and multi-media processing. The heterogeneous design of the Cell BE architecture provides a significant challenge to programmers. Virtual machines, like the JavaVM [GJSB05; LY99], allow code to be distributed in an architecture-neutral format, and executed in a safe, managed run-time. Our research aims to take a similar approach to heterogeneous multi-core computing. Our solution provides a developer with a single abstract virtual machine as a target platform. An advantage of such a virtual machine is that the multi-processing decisions of application execution will be taken at run-time by the virtual machine.

Although the Java virtual machine (JVM) can be implemented on the general purpose main core of the Cell BE using known techniques, it is desirable that Java threads could also run on the special purpose SPEs. Efficiently implementing the JVM on Cell BE SPEs is a significant challenge since the architecture is aimed primarily at regular vector code, rather than branch heavy code with irregular data access patterns. Chapter 4 presents a feasibility study for the interpretation of Java virtual machine bytecode on the Cell BE. We evaluate a number of existing interpreter optimizations on the Cell BE SPE, and report the resulting effects on running time. We also propose a number of novel optimizations aimed specifically at the Cell BE SPE, and measure their performance.

1.1 Thesis

The demands of modern computing systems has resulted in a dramatic and rapid change in computer architecture design. Many core processors with lower clock speeds have replaced high clocked single core processors. Specialized applications like gaming and multimedia have adopted heterogeneous processors. At the same time, the growth of software applications such as web 2.0 scripting, mobile apps and cloud computing has resulted in an increased demand for portable, dynamic and managed programming systems (virtual machines).

These rapid changes in both computer architecture and programming language design has presented a new set of challenges to virtual machine architects. Virtual machines must now efficiently accommodate both hardware with heterogeneous features and software with dynamic features. Virtual machine interpreter performance is highly sensitive to changes in computer architecture. The thesis of this dissertation is that the optimization of portable languages can be achieved through: (1) interpreter data flow (pipelining), (2) just-in-time type specialization and (3) interpreter based type specialization. In this dissertation we take an experimental approach to validate our hypothesis.

1.2 Contributions

The work in this dissertation makes a number of contributions including:

- **Operand Pipelining** — Our Java interpreter in Chapter 4 extends previous work on software pipelining in interpreters. We show that creating a new pipeline stage that fetches an operation's operand provides further gains over previously published work which only pipelines the program counter fetch for branch dispatch improvements. This result is somewhat surprising as not all Java operations have operands yet a pipelined interpreter must still perform dummy fetches of those operands. The gains provided in critical operations are enough to justify a complete operand fetch stage in a pipelined interpreter.

- **Virtual Instruction Branch Prediction** — Using software implemented hint instructions, we present the first branch predictor inside a virtual machine interpreter. The predictor predicts the direction of the application rather than the direction of execution in the interpreter implementing the application. The predictor improves the performance of conditional virtual instructions inside a

pipelined interpreter, in the same way regular branch prediction improves pipeline performance at a machine level.

- **Multi-region type profiling** — We adopt the paradigm of finer compilation unit granularity for improved code generation and optimization. We extend previous work on dynamic partial method compilation in statically typed programming languages and use similar techniques to compile dynamically typed languages with the intent of removing run-time type checks.

- **Type-safe dispatch of JIT code** — Dynamically typed scripting languages have variables which can change type at any assignment during program execution. A specialized region of code therefore could become invalid from one call to the next. We show a practical solution for the storage and management of specialized code regions. These techniques ensure type safe dispatch of specialized code.

- **Interpreter Specialization** — Interpreters for scripting languages perform type checks before each operation. In Chapter 6, we present an intermediate representation to represent a scripting program that an interpreter can execute without performing type checks of variables.

1.3 Relevant Academic Publications

In chronological order, from earliest:

Optimization Strategies for a Java Virtual Machine Interpreter on the Cell Broadband Engine — Kevin Williams, Albert Noll, Andreas Gal, and David Gregg — 2008 Proceedings of the 5th ACM International Conference on Computing Frontiers — Ischia, Italy — ISBN 978-1-60558-077-7 — Pages 189–198

Portable Just-in-time Specialization of Dynamically Typed Scripting Languages — Kevin Williams, Jason McCandless, and David Gregg — 2009 Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing — Newark, Delaware, USA — Springer Lecture Notes in Computer Science

Dynamic Interpretation for Dynamic Scripting Languages — Kevin Williams, Jason McCandless, and David Gregg — 2010 Proceedings of the 8th Annual IEEE/ACM

International Symposium on Code Generation and Optimization — Toronto, Ontario, Canada — ISBN 978-1-60558-635-9 — Pages 278–287

The publications above were researched as collaborations with a number of co-authors. The work presented at the ACM International Conference on Computing Frontiers (and Chapter 4 of this dissertation) was completed in collaboration with Albert Noll of ETH Zurich and Dr. Andreas Gal of UC Irvine. Their contribution to this work was the original virtual machine, CellVM, in which we implemented the interpreter. The design, implementation and evaluation of the interpreter was the sole work of the author under the guidance of his supervisor, Dr. David Gregg.

Jason McCandless provided much needed support in evaluating and documenting the scripting languages research papers published in Languages and Compilers for Parallel Computing (Chapter 5 of this dissertation) and in submission to the ACM International Symposium on Code Generation and Optimization (Chapter 6 of this dissertation). Again, the design, implementation and evaluation of the research ideas in these chapters was the work of the author under the guidance of his supervisor.

1.4 Dissertation Outline

The title of this dissertation is *Data and Type Optimizations in Virtual Machine Interpreters.* The work presented in this document covers a varied selection of virtual machine, language, and micro-processor. The dissertation begins with a background chapter discussing the hardware, software and languages used as research tools in this work. This discussion is followed by a survey of related published materials to familiarize the reader with the relevant scientific research. Chapter 4 is the first of the research chapters in which we present the *'data optimizations'* of our work. This chapter presents the design of an efficient Java interpreter running on a VLIW vector processor, the Cell Broadband Engine. In Chapters 5 and 6, we present our *'type optimizations'* for virtual machines. In particular, this research is concerned with scripting languages where types of variables may change at any point. The aim of the research is to optimize program execution by removing type checks and specializing execution to known variable types. The final chapter is the author's critique of this dissertation as well as a discussion of what direction future virtual machine research should take.

Chapter 2

Background

2.1 Architecture Features

We begin our background discussion with an introduction to some of the architecture features which are relevant to this dissertation. For further reading on these topics the author recommends Hennessy and Patterson editions 3 and 4 [HP03; HP07].

2.1.1 Pipeline

Machine-level instructions can be executed in stages. The number and type of stages in an execution unit is processor dependent. Pipelining allows instructions to be executed in parallel. Each instruction occupies a single stage of the pipeline at a time. On every clock cycle a new instruction can be sent to the first stage of the pipeline as the earlier instruction completes and moves to the next stage. From the widespread adoption of pipelining in the mid 1980s until the end of the single-core era (~2004) the depth of pipelines increased continuously. For example, the first processor in the Pentium family, the P5, had a five stage pipeline. The last, the Pentium 4 Prescott, had a 31 stage pipeline. Figure 2.1 sketches a five stage pipeline with an instruction in execution at each stage of the pipeline. These classic pipeline stages are:

1. IF — Instruction Fetch loads the instruction from its location in a register/memory.
2. ID — Instruction Decode breaks the instruction into its constituent parts (i.e instruction and operands).
3. OF — Operand Fetch loads the instructions operands from registers/memory.

	Cycle								
Instruction	0	1	2	3	4	5	6	7	8
1	IF	ID	OF	EX	WB				
2		IF	ID	OF	EX	WB			
3			IF	ID	OF	EX	WB		
4				IF	ID	OF	EX	WB	
5					IF	ID	OF	EX	WB

Figure 2.1: Example processor pipeline

4. EX — Executes the defined operation.

5. WB — Write Back returns the operation's result to register/memory.

Longer pipelines allow processors to be clocked at higher speeds as a processor can only run as fast as its slowest pipeline stage. Breaking up complex stages into simplified parts shortens the most complex stages of a pipeline and hence an increase in processor performance is expected. However, instructions in execution are not always completely independent. There are a number of scenarios where instructions have dependencies which force a pipeline to *stall.*

Structural Stalls

Structural stalls occur in more complex pipelines when two instructions in the pipeline require access to the same processor resource.

Data Stalls

Data stalls occur when the operand of an instruction is the result of an earlier instruction. In this case, the instruction must *stall* until the previous instruction has completed the WB stage.

Control Stalls

Not all machine code has sequential control flow. Loops and conditions are implemented with branches. These branches have more than one possible target instruction. The next instruction is rarely known until late in the pipeline causing the instruction fetch of the next instruction to be delayed.

Modern processors have various solutions to these problems. Of particular interest to this work are the *branch prediction* techniques used to reduce control stalls. These will be discussed later in this Section.

2.1.2 Superscalar and VLIW

The pipeline presented above issues up to one instruction per clock cycle. In this scenario not every resource of the processor will be utilized at all points. It is desirable to increase processor utilization by issuing multiple fully independent (structural, data and control) instructions in a clock cycle. These types of pipelines are called multi-issue pipelines. There are two types of multi-issue processor, superscalar and VLIW. A superscalar machine issues a variable number of instructions per clock cycle. Instructions can be scheduled statically by a compiler, resulting in in-order execution, or dynamically by hardware resulting in out-of-order execution. In contrast, VLIW processors issue a fixed number of instructions per clock cycle. Multiple instructions are formatted as one large instruction and are statically scheduled by a compiler.

2.1.3 Branch Prediction

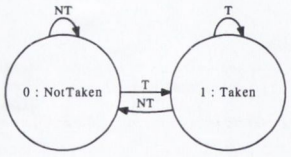
The concept of control stalls in pipelined processors was introduced in Section 2.1.1. By correctly predicting the behaviour of branches in a program it is possible to reduce the number of control stalls in a pipeline [USS97]. If a correct prediction is made then a pipeline will fetch the correct next instruction and will continue execution without any control stalls. There are a couple of different branch prediction techniques. They include:

Static Prediction

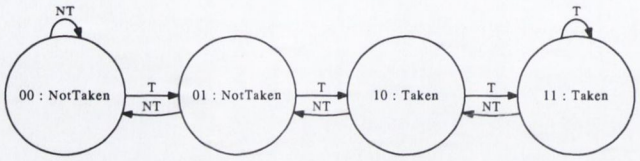
A static predictor always makes the same prediction for a given branch, i.e. taken or not taken. An opportunistic static technique improves prediction by predicting loop control branches to be taken and all others to be not taken. The technique is referred to as *Backward Taken; Forward Not Taken* (BTFN) as loop control branches jump backward to the head of the loop and conditional branches jump forward.

Dynamic Prediction

Dynamic predictors observe behaviour of recent branches and make predictions of the direction of future branches based on the these observations. The most basic dynamic



(a) One bit Predictor



(b) Two bit Predictor

Figure 2.2: Dynamic Branch Predictors: states are label with their value and prediction, transitions are label T for taken and NT for not-taken.

predictor is the *one-bit predictor*. Figure 2.2(a) shows a state machine for such a predictor. The one-bit predictor always predicts the branch to behave the same as the previous execution.

Given that programs typically execute many loops, this one-bit prediction has some fundamental problems. Any given loop will always be mispredicted twice. The first loop iteration will be predicted as not-taken as the last time the loop exited the state was set to *'0 : not-taken'*. The final iteration of the loop will also be mispredicted as the prediction state will be in *'1 : taken'* from all previous branches. A remedy to this problem is the two-bit predictor, as seen in Figure 2.2(b). In this scheme a branch must be not-taken twice before it transitions from an anticipated loop state of *'11 : Taken'* to *'01 : NotTaken'*. With this two-bit predictor, the first iteration of a given loop will be predicted correctly as the exit branch will leave the predictor in the state *'10 : Taken'*.

One and two bit predictors only consider the branch itself when making a prediction. Two-level adaptive predictors [USS97] are mores sophisticated and consider the direction of branches taken previously to the current branch. They use a *branch history register* to record the history of branch behavior. They use this register to index a branch prediction table. Two-level adaptive predictors can achieve prediction rates of greater than 90% [USS97].

The predictors described so far only make predictions as taken or not taken. Branch instructions like direct jumps and indirect branches will not benefit from these predictions. Instead a *branch target buffer*, BTB, is added to the processor. BTB's are indexed by a hash of the bits of an instruction's address. They store addresses which

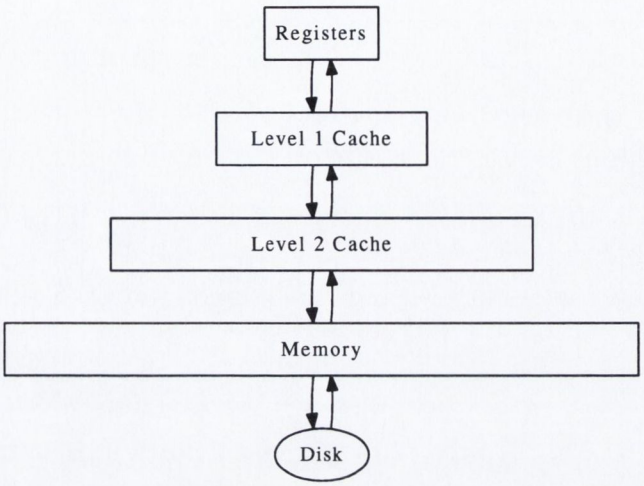


Figure 2.3: Classic Memory Hierarchy

are predictions for the target address of the branch. When a branch has been resolved the BTB is updated with the correct target address for the branch. The next time this branch executes the correct address will be fetched, assuming no collisions occur in the BTB. In regular programs indirect branches are uncommon, however in interpreters they can be up to 13% of instructions executed and with poor prediction and long pipelines can represent up to 61% of total cycles [EG03].

Software Prediction

Another approach to prediction is to allow programmers and compilers to create the predictions. This software level prediction is available in IBM's Cell BE SPE [EOO+05], a VLIW processor that provides a machine level hint instruction. The hint instruction sets a target register which the next executing branch will be predicted to jump to. Limitations of the hint instruction are that only one hint can be live at a time and the hint must be thrown eleven cycles before the branch.

2.1.4 Cache

Long data fetch latencies are a common bottleneck in computing. Modern processors use memory hierarchies to reduce the load time of frequently accessed data. Each level of the hierarchy is physically closer to the processor and has faster access times. Registers are the top level and store values which are accessed by machine instructions. The level 1 and 2 caches are next. These caches store recently accessed memory locations. The final two layers are physical memory and disk. A hierarchical system of

data access like this requires an addressing mechanism to map locations in memory to locations in the cache. Two common addressing mechanisms in caches are direct map caching and set associative caching.

- Direct mapping has a single location in the cache where each memory address can be mapped too. Direct mapping can lead to collisions and require data to be swapped in and out of caches.
- Set Associative caches address this problem by storing N sets of locations to which items can be mapped too. Each set has a fixed number of entries. Collisions in addressing are thus resolved by storing data in one of the M entries in the set.

2.2 Virtual Machine Languages

Virtual machines [Cra05] are used to implement many types of language. The motivation for each of these languages is quite different. However, the fundamental reasons for deploying each language on a virtual machine remains the same. Virtual machines provide a high-level abstract environment for a programmer to develop on. They remove the low-level machine and operating system details and allow the programmer to concentrate on the task of creating an application.

Virtual machines have been in use since the 1960s implementing languages like Pascal, Prolog and many others. Today, the best known virtual machine is the *JVM* which implements the Java programming language. Java was first released in the mid-1990s with the marketing slogan, *'Write Once, Run Anywhere'*. It became popularly known as a web language when it was incorporated into the web-browser Netscape Navigator. The language allowed programmers to create *applets* that could be deployed around the web and run on any machine. In the same Netscape Navigator release, a seldom-discussed (at the time) scripting language named JavaScript was released. While the two languages share naming similarities and some high-level syntactical characteristics their low level implementations differ greatly. Java is a *compile-and-run* statically typed language while JavaScript is a dynamically typed language with many dynamic characteristics such as *eval* statements. Javascript also classifies functions as first class values. Since its first release, JavaScript has grown to be a language of great importance to the web. Many of the most popular web applications (GMail, Google Maps, YouTube, Facebook, etc.) have client sides written in JavaScript. Virtual machines are not just popular in web programming circles. Popular modern languages like Python, Perl, PHP, Ruby and Lua are all implemented with virtual machines.

Python is arguably the most popular portable general-purpose programming language. Its design motivations were to create a clean readable language. It has a comprehensive supporting standard library which makes it a powerful tool for programmers who want to quickly develop powerful applications.
The languages of interest to this project are Java and Lua. The following sections discuss in further detail these languages and the virtual machines they run on.

2.3 Java

The Java technology we know today originated from a somewhat unsuccessful consumer electronics team in Sun Microsystems. The team, code named *'the Green Team'*, was chartered to plan for future trends in computing. A vision of a *'convergence of digitally controlled consumer devices and computers'* emerged as a conclusion of the Green Team. With this vision in hand, they began their work on a hand-held home-entertainment device controller code named *StarSeven*. Part of the development of this device controller was the design of a new language for device applications to run on. The language was named *'Oak'*. When *StarSeven* ultimately failed to gain industry support the Green Team disbanded. However, members of the team managed to rescue Oak. They realized that the newly booming internet was a suitable platform for use of their language. They went about introducing support into the popular web-browser Netscape Navigator. The new technology was named 'Java' and was first released in May 1995. The following sections introduce some of the basic concepts of the Java virtual machine architecture sketched in Figure 2.4.

2.3.1 JavaVM

Both the Java language [GJSB05] and the Java virtual machine (JVM) have comprehensive specifications. This section gives a broad overview of the JVM and introduces its constituent parts.

Java Source

Syntactically Java source resembles a C++ style language in that it is an object-oriented language supporting a class model. On top of this its basic block boundaries are implemented using { } notation and it shares many of the same keywords as C++, e.g. *for*, *while*, *class*, *int*, *double*, *float*, etc.

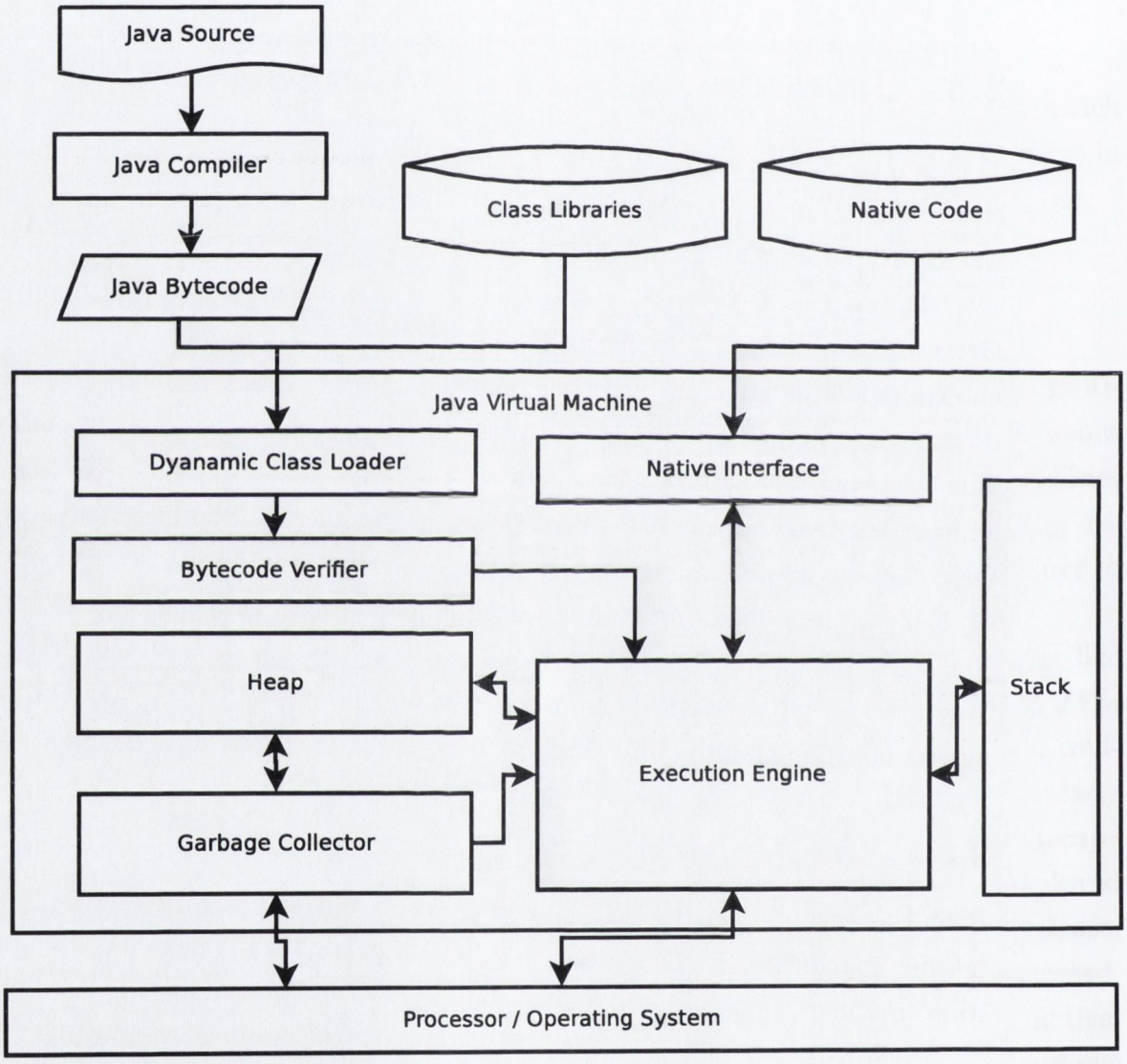


Figure 2.4: JVM Architecture

Java Bytecode

This is where any similarities to C or C++ end. Java source is not compiled and linked into native machine-level executables. Rather it is compiled into *class files* of Java bytecodes. The bytecode format is designed to be compact. Instructions are represented as a single byte with n operands. Classically, JVMs are implemented as stack machines. Many bytecodes have no instruction-level operands, instead they fetch their operands from an operand stack.

Dynamic Class Loader & Bytecode Verifier

At some point in execution it becomes necessary to load a set of class files. It is the job of the *Dynamic Class Loader* to load the desired classes into their run-time representations.

The JVM is designed to execute bytecode received from foreign sources. This source may be untrusted. For this reason all loaded classes are first verified as being *correct*. *Bytecode verifiers* confirm correctness according to three criteria[1]:

- There are no operand stack overflows or underflows.
- The types of the parameters of all bytecode instructions are known to always be correct.
- Object field accesses are known to be legal–private, public, or protected.

After these checks are complete the execution of Java bytecode can begin.

Execution

The initial processing unit of the classic JVM is the interpreter. The interpreter contains a segment of code for each bytecode in the language. An interpreter has access to some global state variables such as a *program counter*, a *stack pointer*, a *heap* and some *constant pool*. The program counter points to the bytecode currently being executed. The implementation of each virtual instruction updates the program counter to point to the instruction that will be executed next. This is generally the next instruction in the program except in the case of conditional branches and loop instructions where some offset value is used to control dispatch.

Interpreters are generally sluggish and lack the performance a developer or user might expect from a natively compiled application. This becomes a major issue for

[1] source http://java.sun.com/docs/white/langenv/Security.doc3.html

Java (and virtual machines in general) when applications become large and run for a long time. In order to improve performance, virtual machine designers augment interpreter execution with a just-in-time (JIT) compiler. While interpreting, the virtual machine performs some profiling to count the number of times certain regions of code are executed. When this profile reaches a certain threshold a JIT compiler will compile the bytecode to native machine code and the virtual machine will execute this native code from that point on.

Stack & Heap

Every Java thread executing in a virtual machine has their own call stack. The call stack is a stack of function frames. Frames are used to pass parameters and return results, at method invocation and return respectively. A method's frame also stores all local variables associated with the method. These frames are not necessarily contiguous in memory and can be heap allocated. Frames are created and destroyed as methods are called and returned. The top of the frame is used as an operand stack where local variables are pushed and popped as instruction operands. If a stack grows larger than permitted by the configuration of the JVM then a *StackOverflowError* will be thrown. Should more frames be allocated to the heap than is permitted then an *OutOfMemoryError* will be thrown.

The JVM heap is the run-time data store shared among all threads. It stores all objects and classes declared in the program. The heap is initialized on JVM startup and deallocated on shutdown. During run-time, dead objects are freed by an automatic memory manager called a *garbage collector* [Jon96]. The garbage collector frees the programmer of the task of explicitly allocating and deallocating his own objects.

Threading

Java allows programmers to declare different threads of execution. Multi-threading allows these threads to run concurrently inside the JVM. Threading reduces idle time and increases responsiveness for users. Java threads have priority values. The higher the priority the higher the precedence for scheduling. Threads are initialized with the same priority as their parent.

2.3.2 CellVM

CellVM [NGF06] is a Java virtual machine running on the Cell Broadband Engine. The motivation for CellVM is to allow threads of the JavaVM to run on the SPE processors

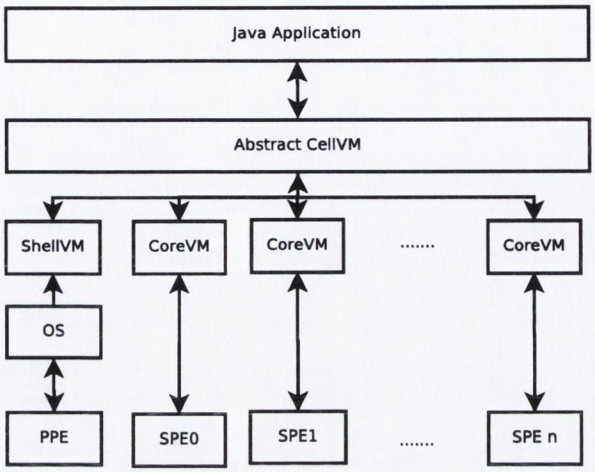


Figure 2.5: CellVM Architecture

on the Cell. The advantage of such a system to Cell programmers is the single abstract machine they target, rather than the complex array of main processor and eight sub processors of the Cell hardware. Figure 2.5 sketches the architecture of the CellVM. A single main virtual machine, the ShellVM, runs on the main PPE processor. An array of CoreVMs run on each of the sub-processors (SPEs). The CoreVMs execute kernels containing the most common bytecode instructions. Uncommon instructions and those which allocate to the heap are executed by the ShellVM. A software based caching mechanism stores local copies of frequently accessed data structures in the SPE's local store. Utilizing CoreVMs, the CellVM processor achieves an average speedup of 2.5 for an interpreted execution, and a speedup of 3.5 for JIT execution. Performance speedups are heavily dependent on good data locality. Applications with large data sets are likely to swamp the communication of the ShellVM and CoreVM and kill any likely performance increase.

2.4 Lua

Lua was created and is maintained by the LuaLab research group at PUC-Rio, Brazil. It is a dynamically typed embeddable scripting language. It combines simple procedural syntax with data constructs based on associative arrays and extensible semantics. Its authors are disciplined guardians of the language. Throughout its life Lua's design has had four key principles [Ier06]:

1. Extensibility: Lua is highly extensible. It has an extensive API that makes it an excellent choice of language for extending large applications, in particular those

written in C/C++. In these types of applications, the core performance-critical parts are written in C/C++ and are glued together using Lua.

2. Simplicity: Lua is a small language. It is simple to learn as there are very few features for a beginner to become familiar with. Simplicity in the Lua project extends to the virtual machine implementing the language. The source of the Lua virtual machine is extremely clear. Features are only added to the language when they are necessary.

3. Efficiency: Lua is fast. Independent benchmarking *'shootouts'* have shown it to be the fastest interpreted scripting language. The authors of Lua attribute much of this performance to the Lua standard libraries.

4. Portability: The source code for the Lua virtual machine strictly adheres to the ANSI/ISO C standard. This makes it extremely portable. It never uses conditional compilation to adapt source code to different targets. Portability does not simply refer to Windows or Linux but to all combinations of platform and operating system including , but not limited to: Playstation, XBox, PowerPC, x86, ARM, MacOS, Windows, Symbian OS and Palm OS.

These principles have helped develop a clean, well documented language [Ier06; LFC06; FCI08]. Conservative development has not resulted in a sparse featureless language. Far from it, Lua has had large growth since its initial implementation and the authors have changed many aspects of both language implementation and language features including:

1. Stack to Register: In release 5.0, the Lua instruction set was redesigned as a register based instructions set. Before this Lua was always a stack based machine.

2. Incremental Garbage Collection: There are two types of garbage collector. The first, *stop-the-world* collectors, halt program execution to perform a complete collection cycle. Until release 5.1, Lua had a stop-the-world collector. The second, incremental garbage collectors, collect dead objects in phases (or incrementally). This removes long pauses from program execution, something which is important for real time applications like games.

3. Meta-mechanisms: The authors of Lua claim: 'to not provide policy but mechanisms to implement policy'. For example, objects and classes are not a feature of the language but they can be implemented using Lua's table structures. Tables

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31			
OPCODE	A	B	C
OPCODE	A	BX	
OPCODE	A	SBX	

Table 2.1: The Lua 5 instruction format

in Lua have *meta-tables*. By default these meta-tables are *nil*. When used, they provide programmers with a powerful design mechanism.

4. Coroutines: A coroutine is a type of thread, in that it has its own line of execution and its own stack. They differ from standard threads in that they are not run concurrently. Instead, coroutines explicitly start, halt and resume. At any given time a program is only ever running one of its coroutines. One of the motivations for the authors adding this feature was demand from game programmers to *'somehow suspend a character and continue it later'*.

2.4.1 Lua Virtual Machine

In this section we will discuss some of the implementation details of the Lua virtual machine that are of interest to this project.

The Interpreter

Lua is designed to be portable. The interpreter therefore is implemented using an ANSI C switch statement. Each opcode in the instruction set has a *case* for its implementation. Lua instructions are 32-bit values. Figure 2.1 sketches the three format types for Lua instructions. The switch statements takes the OPCODE field of the instruction as the value to branch on. The most common of the three formats is 'OPCODE A B C'. A typical use of this instruction format is an operation on registers B and C that stores the result in register A. A full listing of instructions and their implementations follows in Section 2.4.2 and Figure 2.2.

Representation of Values

Lua is a dynamically typed language with nine primitive types. They include:

1. NIL: is the void or null type. It has no value.
2. BOOLEAN: are represented by the *'b'* field in Figure 2.6(b). They have two values, true or false.

```
1 typedef struct TValue {
2  Value value;
3  int tt;
4 } TValue;
```

(a) TValue

```
1 typedef union {
2  GCObject *gc;
3  void *p;
4  lua_Number n;
5  int b;
6 } Value;
```

(b) Value

Figure 2.6: Structures representing Lua's 'tagged values'

3. LIGHTUSERDATA: are blocks of memory allocated and freed by the user.

4. NUMBER: by default are represented as *doubles*. Lua can be easily configured to use *floats* or *longs* instead.

5. STRING: are arrays of bytes of an explicit size. String values are fetched via the garbage collector object, the *'gc'* field in Figure 2.6(b).

6. TABLE: are associative arrays that can be indexed by any value except NIL. Further discussion in Section 2.4.1.

7. FUNCTION: can be either Lua functions or C functions.

8. USERDATA: are blocks of memory allocated by Lua and are subject to garbage collection.

9. THREAD: represent coroutines (see Section 2.4)

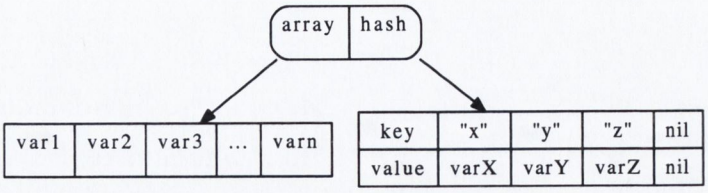
Values of all primitive types are treated as *first class values*. They can be stored in local and global variables, used as table keys, stored in tables, passed as parameters, and returned from functions. Local variables are stored on a call stack. At every function call the function allocates space on the call stack for all its local variables.

Structure of Tables

Tables are the only data structuring mechanism in Lua. The *meta-table* feature of tables allows a programmer to define his own structures. Tables are associative arrays — they can store any value and are indexed by any value. The Lua 5 implementation has implemented tables in two parts (see Figure 2.7(b)). The first part is the array part, this stores values indexed by a value of type integer that is within the bounds of the allocated array. These arrays grow as more integer keys are added. The second part is the hash part. This part stores values indexed by all non-integer keys. A hash

```
1 typedef struct Table {
2   TValue *array;
3   int sizearray;
4   Node *hash;
5   lu_byte lsizenode;
6   Node *lastfree;
7   struct Table *metatable;
8   GCObject *gclist;
9 } Table;
```

(a) Table



(b) Illustrates the separation of integer keys and others.

Figure 2.7: Structures representing Lua's Tables

function first hashes the value of the key. This hash value indexes a list of possible values. The correct entry is found by searching the list for an index which is an exact match for the key. A good hash function minimizes collisions. Therefore, the list search should rarely perform more than one key comparison.

Functions and Closures

Lua compiles functions to a structure called a *prototype*. Prototypes store the array of instructions implementing the function as well as all constants, literals and function details such as the number of variables etc. They also store additional debug information for the function. Prototypes are stored inside *closures*. A closure is created when a 'FUNCTION ... END' statement is called. Closures store the function type (i.e. Lua or C) as well as the prototype and upvalues for a Lua function. Lua is lexically scoped [IdFC05] meaning variables are accessed at their highest point of definition (i.e. non-local variables are accessed as upvalues from the previous scope of execution / the calling function).

2.4.2 The Lua Instruction Set

The Lua instruction set presented in Figure 2.2 has 35 instructions. The first five instructions load a destination register R(A) with a given value. The next seven instructions are table accesses instructions. In general these instructions have a table operand, a key operand and either a source or destination operand depending on whether the operation is a load or store. The eight instructions after these are arithmetic instructions. They perform standard arithmetic operations (there is also a string concatenation instruction). These are followed by five branch instructions, both conditional and direct. Next are some call and return instructions for calling both Lua functions and library

functions. Instructions that implement two kinds of for loop are next. The two for loops are: (1) the common for loop incrementing an index from a start to finish point and (2) a *tail for loop* which increments an iterator over some data-structure. The last four instructions are a miscellaneous group of instructions for; storing a list to a table, closing a set of variables, creating a closure, and setting a variable number of arguments.

OPCODE	OPERANDS	PSEUDO-IMPLEMENTATION
MOVE	A B	R(A) := R(B)
LOADK	A Bx	R(A) := K(Bx)
LOADBOOL	A B C	R(A) := (Bool)B; if (C) PC++
LOADNIL	A B	R(A) := ... := R(B) := nil
GETUPVAL	A B	R(A) := U[B]
GETGLOBAL	A Bx	R(A) := G[K(Bx)]
GETTABLE	A B C	R(A) := R(B)[RK(C)]
SETGLOBAL	A Bx	G[K(Bx)] := R(A)
SETUPVAL	A B	U[B] := R(A)
SETTABLE	A B C	R(A)[RK(B)] := RK(C)
NEWTABLE	A B C	R(A) := (size = B,C)
SELF	A B C	R(A+1) := R(B); R(A) := R(B)[RK(C)]
ADD	A B C	R(A) := RK(B) + RK(C)
SUB	A B C	R(A) := RK(B) - RK(C)
MUL	A B C	R(A) := RK(B) * RK(C)
DIV	A B C	R(A) := RK(B) / RK(C)
POW	A B C	R(A) := RK(B) ^ RK(C)
UNM	A B	R(A) := -R(B)
NOT	A B	R(A) := not R(B)
LEN	A B	R(A) := len(R(B))
CONCAT	A B C	R(A) := R(B) .. .. R(C)
JMP	sBx	PC += sBx
EQ	A B C	if ((RK(B) == RK(C)) = A) then PC++
LT	A B C	if ((RK(B) < RK(C)) = A) then PC++
LE	A B C	if ((RK(B) <= RK(C)) = A) then PC++
TEST	A C	if (R(A) <=> C) then PC++
TESTSET	A B C	if (R(B) <=> C) then R(A) := R(B) else PC++
CALL	A B C	R(A), ... ,R(A+C-2) := R(A)(R(A+1), ... ,R(A+B-1))
TAILCALL	A B C	return R(A)(R(A+1), ... ,R(A+B-1))
RETURN	A B	return R(A), ... ,R(A+B-2)
FORPREP	A Bx	check type(R(A)), type(R(A+1)), type(R(A+2)) == number
FORLOOP	A sBx	R(A)+=R(A+2); if R(A) ¡?= R(A+1) then PC+= sBx
TFORLOOP	A C	R(A+2), ... ,R(A+2+C) := R(A)(R(A+1), R(A+2));
SETLIST	A Bx	R(A)[Bx-Bx%FPF+i] := R(A+i), 1 <= i <= Bx%FPF+1
CLOSE	A	close stack variables up to R(A)
CLOSURE	A Bx	R(A) := closure(KPROTO[Bx], R(A), ... ,R(A+n))
VARARG	A B	R(A), ... ,R(A+B) := ...

Table 2.2: The Lua 5 instruction set [IdFC05]

Chapter 3

Literature Survey

The work in this dissertation covers both statically and dynamically typed languages. This chapter provides a survey of the relevant publications from scientific literature. The survey is divided into two sections. The first section surveys the optimizations of virtual machine interpreters. We have divided this section into groups of related optimizations discussing each set of optimizations independently. The second surveys publications on just-in-time compilation and run-time specialization of dynamic scripting languages. There are many publications on just-in-time compilation for static languages; as we are interested in run-time specialization of scripting languages we have only discussed the most relevant publications for static languages. The survey ends with analysis of the techniques used in the optimization of dynamic languages (e.g. Self), logic languages (e.g. Prolog) and scripting languages (e.g. JavaScript).

3.1 Interpreters

Interpreters can be generalized as having three main stages: the first stage fetches the operands of the instruction in execution, the second performs the execution and stores the result, the third dispatches execution to the next instruction. The majority of interpreter research aims to optimize one of these these stages. For this reason the research ideas behind interpreter optimization are quite disjoint. We adopt this generalized characterization of interpreter stages to group the related research ideas into the following groups:

- **Investigative:** There are a number of interesting investigative surveys on interpreter performance. We discuss the findings of these surveys in Section 3.1.1.
- **Dispatch:** Dispatching is the term used to describe the technique for jumping

from one virtual instruction to the next. There are several options an interpreter designer has for dispatching. They have a range of trade-offs in terms of portability, space and performance. We present each of the dispatching options in Section 3.1.2.

- **Pipelining:** The three generalized stages of interpreter execution result in long dependence chains. The penalty for these dependencies differs from architecture to architecture. Interpreter pipelining is a technique to break the stages of interpretation apart to reduce dependency stalls. Section 3.1.5 introduces the concepts behind interpreter pipelining.

- **Superinstructions:** Despite the performance improvements gained from optimized instruction dispatch, it is often still a bottleneck for interpreters. An alternative approach is to reduce the number of dispatches required by grouping sets of instructions into superinstructions. Section 3.1.6 ends the interpreter part of our survey with a discussion of superinstructions.

- **Operand Access:** Virtual instruction sets can be designed to access operands from a stack or from a set of local registers. The pros and cons of each of these architectures has been the subject of much debate within the research community. We present the ideas behind each architecture in Section 3.1.3.

- **Stack Caching:** Virtual machines with stack architectures can suffer long load latencies in the operand fetch stage of execution. Some interpreters implement caching schemes to store the top values of a stack in registers. Section 3.1.4 discusses these techniques.

3.1.1 Investigating Interpreter Performance

The structure of interpreters and the feature of the pipelined processors on which they run have resulted in several interesting surveys of their efficiencies/inefficiencies. Romer *et al.* [RLV+96] examined interpreters for MIPSI, Java, Perl and Tcl running on a DEC Alpha platform. They concluded that interpreters did not motivate special hardware support at that time. They found that, in general, interpreter performance was not affected by the application benchmark, but rather dominated by the structure of the interpreter. Their claim that hardware support was not required was supported by observations that interpreters shared characteristics of the SPECint benchmarks (a crucial figure in their simulated results was a penalty of four cycles for a branch misprediction — a number far smaller than that expected in the Pentium 3 family

of processors popular at the time). Ertl and Gregg rebutted Romer's claim in their study of the efficiency of interpreters [EG03], which extended their earlier work [EG01; GEK01]. They measured indirect branches to be 3.2%–13% of the total number of instructions executed and that on processors with longer pipelines misprediction stalls represented up to 61% of all cycles. Their work found threaded code to be twice as fast as switch based dispatch (see Section 3.1.2) as fewer instructions are executed and prediction accuracy is greater on BTBs. They suggested two ways to improve interpreter performance: (1) replicate virtual machine instructions to increase BTB accuracy and (2) create superinstructions to reduce the number of indirect branches and BTB mispredictions (see Section 3.1.6).

Individual virtual machine instructions often contain similarities with other virtual instructions in the instruction set. Ertl *et al.* [EGKP02] created *vmgen*, a generator of efficient virtual machine interpreters. Their generator took descriptions of virtual instructions as input and output a C implementation of the languages interpreter. The interpreters *vmgen* generated for Java and Forth were faster than existing interpreters for those languages. Casey [CGE05] extended work on vmgen by creating *tiger*, an automatic interpreter generator. Tiger created optimized interpreters using many of the optimizations discussed in the following sections especially stack caching and superinstructions. Casey *et al.* [CEG07] used tiger to produce a study of indirect branch prediction accuracy in virtual machine interpreters. They studied the two ways suggested by Ertl to improve branch accuracy, virtual instruction replication and superinstructions. They found that using dynamic superinstruction and replication it was possible to eliminate almost all branch mispredictions. Using their techniques they produced an interpreter for the Forth system which acheived a max speedup of 4.55x over threaded equivalent.

3.1.2 Dispatch Techniques

In virtual machine interpretation, dispatching is the term used to describe the mechanism used to jump from one virtual instructions implementation to the next. The basic and most portable implementation of dispatching is using an ANSI C [Sch90] *switch* statement. The switch value is an integer representation of the virtual instruction, and there exists a *case* for each virtual instruction in the language. A bounding loop repeats the dispatch process until an exit instruction is executed. Bell [Bel73] presented the concept of *threaded code* as a more efficient dispatching scheme. In his threaded code, a linear array of memory addresses represented a program's instruction stream. Dispatching of an instruction was a simple indirect branch to the address in

the instruction stream. Each instruction updates a program counter, indexing the instruction stream, to point to the next instruction to execute. Bell's method provides a far more efficient dispatch as fewer instructions are executed and, in modern processors, correct branch prediction rates are much higher. Dewar [Dew75] extended Bell's work and presented a threading technique requiring less space. Dewar proposed storing a table of addresses, each entry in the table is the address of a virtual instruction's implementation. The program stream is represented by integers that index the table. Virtual instructions are dispatched using the memory location fetched from the table lookup. This technique has the same branch prediction advantages of threaded code but has traded an extra load for the more compact program representation.

The body of virtual instructions can alternatively be stored in callable routines. Both direct call threading and subroutine threading [BVZB05] store virtual instructions in native functions. The program stream for a direct call interpreter is an array of function pointers. The interpreter passes over this array calling each function as it goes in a similar manner to direct threading. Subroutine threading adds another level of indirection to direct call threading. The program stream in subroutine threading is an array of references to call points. The interpreter passes over the instruction stream loading and calling each virtual instruction. This process achieves fewer branch mispredictions than threaded dispatch.

3.1.3 Stack versus Register

Virtual instruction sets can be designed to work on either stack or register operands. A long standing question in literature is whether a stack or register architecture is more favorable for interpretation. Myers [Mye77] was the first to make a claim for register machine's superiority to stack. In his paper, he explored the size of code necessary to express common expressions found in Fortran programs. He used metrics of (1) number of instructions, (2) size of code in bytes, and (3) number of elements to find that his *storage-to-storage* (what this document regards as register) representation to be superior to both stack and a mixed register approach. Davis *et al.* [DBC+03] presented a comparison of stack and register representations for the SPECjvm98 benchmarks [SPE98]. They claim to be the first publication to make a comparison of stack and register instruction sets across a full benchmark set. They found that a register representation of their benchmarks on average reduced instructions executed by 34.88%. They further measured an increase in loads of 44.81%. This equates to an additional 2.32 loads per executed instruction. They postulate that register machines offer a preferred instruction set due to the high cost of instruction dispatch in pipelined

processors. In extending work by Davis, Shi *et al.* [SCEG08; Shi07] presented a more sophisticated stack-to-register translation and optimization scheme. The translation scheme presented used a combination of: (1) forward copy propagation, (2) backward copy propagation, (3) combining constant iadd/isub into an iinc instruction, and finally, (4) peeling constant loads outside of loops and removing redundant constant loads. Their work improved the instruction execution reduction achieved by Davis *et al.*, reducing instructions executed by 47%. They found that their register representation was on average 25% larger but 32.3% faster than an equivalent stack representation on a Pentium 4 processor. Two scripting language implementations cite Shi's work as major motivation for switching from a stack implementation to a register implementation. They are the JavaScript virtual machine, SquirrelFish Extreme, and the Lua 5.0 implementation [IdFC05].

3.1.4 Stack Caching

Virtual instructions in stack based virtual machines perform operations on values stored on the top of the operand stack. For every operand accessed a load is required and for every result a corresponding store. Ertl [Ert95; Ert96] presented a 'caching' solution to reduce these loads and stores. Stack caching requires the interpreter to access machine registers rather than stack locations while performing operations. The number of registers used in this caching solution can be static, *one-state* cache, or dynamic, *multi-state* cache. Ertl presented the following heuristics in [Ert96] to measure the advantages of caching n registers in a machine with constant load and store costs[1], where x is the number of loads and y is the number of stores in an operation:

- caching n is better than caching n–1 items, if $x \geq n \wedge y \geq n$, due to fewer loads from and stores to the stack.
- caching n is usually slower than caching n–1 items, if $x \neq y \wedge x < n \wedge y < n$, due to additional moves between registers
- caching n is as fast as caching n–1 items in all other cases.

Profiling of a language implementation and using the above heuristics measures the effectiveness of caching for a given implementation. Ertl's instrumentation of the Forth language found that caching more than one value was not worth while.

Having a constant number of cached registers makes for a simpler implementation. However, it can increase the number of loads and stores required. For example, a

[1] in Chapter 4 the Cell BE has a constant load cost of 6 cycles

Instruction i	Fetch	Execute	Dispatch					
Instruction i+1				Fetch	Execute	Dispatch		

(a) Sequential interpreter execution.

Instruction i	Fetch	Execute	Dispatch					
Instruction i+1		Fetch	Execute	Dispatch				
Instruction i+2			Fetch	Execute	Dispatch			
Instruction i+3				Fetch	Execute	Dispatch		

(b) Pipelined interpreter execution.

Figure 3.1: A comparison of sequential and pipelined interpreter execution

conditional instruction loads a value from the top of stack into the cache, should this instruction be followed by a push of a constant value the load will now become redundant and a further store and load are required. To overcome this problem Ertl, proposed a multi-state cache solution. In this solution, a version of the whole interpreter was stored for every possible state of the cache. The virtual machine transitions through states during execution keeping the load and storing of values to the stack to a minimum. Storing multiple versions of an interpreter's state results in a large amount of code repetition and increase in code size. Peng *et al.* [PWL04] used code sharing among states to reduce code size. With careful code layout an instruction state that shared terminating code with another would fall-through to the next state. Peng's interpreter was only 13.6% larger than an equivalent threaded interpreter.

3.1.5 Interpreter Pipelining

Recall the three stages of interpreter execution: (1) argument fetching, (2) execution, and (3) instruction dispatch. In a standard interpreter these stages are executed sequentially. Each stage, however, is dependent on results from previous stages, i.e. dispatch is dependent on the program counter defined in execution and execution is dependent on the arguments fetched in the first stage of execution. With these dependence chains program stalls relative to the size of the processor's load latency will be measured in any interpreter with a sequential layout (see Figure 3.1(a)).

Hoogerbruge *et al.* [HA00] designed an interpreter for the TriMedia VLIW Media processor (models TM1000 and TM1100). This processor is a five-issue VLIW processor where each instruction holds up to five operations that can be issued to a subset of 28 functional units. The load/store latency is three cycles, and the branch latency is four cycles. A branch latency of four cycles requires three delayed instructions to be executed before a branch becomes effective. Given that five operations can be issued per instruction, fifteen operations can be scheduled in the three delayed slots

before a branch. A stack interpreter add operation has a dependence chain of: (1) loading two values from a stack, (2) adding them, (3) storing the results back to the stack, (4) loading the virtual instruction and (5) branching to the next instruction. In the TriMedia processor this dependence chain creates at least 9 stalls[2], 3 for the add operation waiting for the stack loads, 3 for the branch waiting for the instruction load, and a further 3 for the delayed branch.

Hoogerbruge proposed a pipelined interpreter (see Figure 3.1(b)) as a more efficient layout. He performed the argument fetch phase of dispatch in the previous instruction. This broke the dependence chain and allowed the compiler to schedule instructions more efficiently. With pipelining, he measured simple instructions, like additions, could be executed in four clock cycles. His interpreter achieved on average a 26.4% cycle decrease across the SPECjvm98 benchmarks.

Hoogerbruge [HATVDW99] used his pipelined interpreter in a mixed-mode execution system. This system compiled programmer defined critical code parts into native code and used an optimized and compressed interpreter for the non-critical parts. Compression reduced the size of non-critical code parts by a factor of five on the VLIW TriMedia processor.

Ogata *et al.* [OKN02] researched a similar problem for superscalar machines. They noted that Java virtual instructions are frequently small (one byte in many cases). This means that many of the virtual instruction loads in the interpreter are redundant. They wished to remove these redundant loads. Their solution to this problem was to create different *bytecode handlers*. Each handler is designed to support the current virtual instruction in execution. The handler accesses a set of *bytecode prefetch registers*. They use their current position – and the type of bytecode – in the program stream to determine the next position. Through the use of a state machine, they were able to specialize handlers and remove bit and shift operations executed by the handler.

3.1.6 Superinstructions

Romer *et al.* [RLV+96] observed two different types of interpreter. They classified them as high-level interpreters and low-level interpreters. High-level interpreters execute thousands of machine level instructions for each execution of a virtual instruction. Low-level interpreters, on the other hand, typically execute fewer than 100 machine instructions per virtual instruction and in some Java instructions fewer than 10. For this reason, the bottleneck of virtual instruction dispatch through mispredicted di-

[2]this is an estimation based on Hoogerbruge's figures, the details of the TriMedia instruction-level parallelism and compiler scheduling may alter the figure

rect branches is greater in low-level interpreters than it is in high-level interpreters. Superinstructions are formed by combining a number of low-level virtual instructions together to form a new large instruction. The benefit of these superinstructions is that the overhead of dispatching each individual instruction is removed. While interpreters become larger, executables typically become smaller as many bytecodes are combined into one. Hughes [Hug82] first introduced the idea of superinstructions, he called super-combinators, for functional programming languages. Proebsting 95 [Pro95] later published the first work on superinstructions in bytecode interpreters. He claimed superinstructions can achieve a double or triple performance increase over standard interpretation. He further suggested the use of static or dynamic feedback to specify suitable instructions.

3.1.7 Instruction (De)Specialization

Stack based virtual machines have fewer instruction-level operands than register based machines. Instead they load their operands from the stack. Instruction specialization is a technique used to remove operand loads by merging instructions with common constant operands forming a new specialized instruction. For example, the ILOAD instruction in Java has a single operand. This instruction loads the local variable indexed by the operand to the top of the stack. The most frequently loaded variables are the lower indexed variables, hence most Java virtual machines support specialized instructions like ILOAD_0 and ILOAD_1. Not all specialized instructions can be justified [DHPW01], depending on the executing application, it may be beneficial to drop rare specialized instructions like FSTORE_0.

3.2 Run-Time Optimization

The publications discussed in previous sections have concentrated on ahead-of-time optimizations of a virtual machine's interpreter. These optimizations increase bytecode execution performance. However, even supported by these optimizations, interpreter performance still lags behind the performance of natively compiled code. To make further improvements virtual machine designers augment interpreter execution with a just-in-time (JIT) compiler [Ayc03]. These JIT compilers transform the portable bytecode of the program into an equivalent native representation. The native representation is optimized towards the target architecture and typically runs many times faster than the equivalent interpreted bytecode. The most popular JVMs from IBM

and Sun both provide a host of literature for their JIT compilers [AFG+00; PVC01; SOT+00; SYN03; SYN06]. Some of these techniques are related to our work in Chapter 5 and are discussed further in Section 3.2.1. For the most part, the work in this dissertation concentrates on the run-time optimization of dynamically typed scripting languages. The optimization of scripting languages is a relatively new research topic. As a result, we present the few publications on scripting language specialization in Section 3.2.4. Before this, we discuss some earlier work on dynamic languages. The Self [US87] programming language is an object-oriented dynamic language. The Self research project ran from the late 1980's until the mid-1990's. The project resulted in many execution techniques for dynamic languages, a selection of the ideas related to this work are presented in Section 3.2.2. First, we begin this section of our survey with a discussion of JIT compilation and compilation shapes.

3.2.1 Functions, Loops and Traces

As we already know, mixed-mode virtual machines typically begin execution in an interpreter. They perform some form of profiling of executed code, commonly a method execution count. When this count becomes sufficiently *hot*, by reaching some defined threshold, interpretation halts and the method is compiled into native code. Examples of such virtual machines are Sun's Java Hotspot virtual machine [PVC01], IBM's just-in-time compiler [SOT+00] and Self's just-in-time compiler [Cha92].

Compilation of whole-methods frequently results in the compilation and optimization of *cold-code*. Compilation of this rare code causes two problems, (1) JIT compilations takes more time as more code has to be compiled and (2) *cold-code* hinders the optimization of the compiled region. Whaley [Wha01] argued that the trade-off of fast JIT compilation versus fast code performance could be improved by compiling at a sub-method granularity. He presented a simple technique for compiling and optimizing at a *partial-method* level. He observed that even hot functions contained cold fragments of code. He claimed that by removing *cold-code* from compilation that JIT compilation time could be reduced and native code performance improved. Whaley experimented on his technique by performing compilation as an off-line step, he measured an average performance increase of 10% compared to whole-method compilation. Suganuma *et al.* [SYN03; SYN06] used similar region based techniques in the IBM JVM [SOT+00]. They measured an average performance increase of 4% in their virtual machine, however more importantly they achieved a 10% – 30% reduction in compile time versus a regular method boundary approach.

A third level of granularity is the use of inter-procedural *traces*. Gal's HotpathVM [GPF

Gal06] is a trace based JIT for the Java language. Hotpath begins execution in a threaded interpreter and uses a profiling step at each dispatch to record a trace of executed instructions. The JIT compiler for HotpathVM performs compilation of traces in linear time. Gal broke the compilation steps into three phases: (1) stack deconstruction and transformation into SSA form, (2) analysis and (3) code generation. A major problem for trace compilers is the fact that control flow is rarely regular. Gal used techniques first presented in the trace profiler Dynamo [BDB00] – inner traces of code are profiled and compiled first – subsequent outer regions are recorded as child traces which inner trace exits jump to directly. Zaleski's YETI [ZBS07; Zal08] is another trace compiler for the Java language. YETI takes an alternative approach to building traces. It uses a modified subroutine threaded technique to gradually build a trace from program execution. YETI virtual instructions point to *dispatcher structures* rather than the virtual instructions implementation. These dispatcher structures contain fields for a pre-worker, post-worker, body and payload. The pre-worker and post-worker perform the profiling and construction of traces. The body executes the block or trace that it points to. YETI achieves an average of 1.51x speedup over interpretation. This is a disappointing figure when compared to other Java JIT compilers. However, to date, they have only implemented a sub-set of the Java instruction set. Further development is required to evaluate the full benefits of the technique.

3.2.2 Self

Self is an object-oriented programming language initially designed by David Ungar and Randall B. Smith at Xerox Parc. Craig Chambers and Urs Holzle, of Stanford University, researched the efficient design and implementation of the language and many of the features discussed in this section. Self is a pioneering dynamic language, in that its implementation addressed many of the key issues of efficiency in implementing dynamic languages. The Self implementation is a prototype-based implementation [DMC92] of an object-oriented language. Prototypes are an alternative to the classical class based implementation of object-oriented languages. Objects are defined by *cloning* the prototype describing the desired object. Features can be dynamically added and removed from a prototype.

Self implements this prototype based language using *maps* [CUL89]. Maps are a space efficient implementation for defining classes, when compared to a naive implementation which would store all variables, methods, and inheritance information for each instance of a class. Maps take advantage of the fact that few instances of objects are completely unique. The majority of objects are in practice created from a clone

and keep the same prototype for the duration of their life.

Special primitives within the language are used to make changes to an object's format or inheritance. Self, therefore classifies objects cloned from a prototype as a *clone family.* In the object storage system, Self objects are represented by the values of their *slots* (i.e. the class attributes) and a pointer to the object's map. The map contains all of the information for the *slots* in the object, e.g. the slot offset index within the object, the name of the slot, and whether it is a parent slot. For methods, the map stores a pointer to the bytecode implementing the function. Maps are immutable, meaning their state cannot be changed after they are created. When an object changes format, a new map describing this format is created and the new object points to this map.

Self is a dynamically-typed language. A lack of concrete type knowledge complicates the optimization of functions. The Self implementation introduced a concept of *customization*, where functions are optimized for each object type. Chambers and Ungar [CU89] claimed a doubling in performance over their initial implementation after method customization. The penalty for this performance increase is an increase in code size as duplications of functions are stored. Customization performed static type prediction to optimize code to a set of types. Predictions were not guaranteed to be correct and guards were used to protect operations. This initial implementation of customization had a number of shortcomings including: (1) argument types of functions were still unknown, (2) decisions of inlining were primitive and (3) compile times were prohibitive as large number of customizations were compiled.

Method calling was also still a bottleneck due to the expense of looking up the dynamic function. At the time other dynamic languages like Smalltalk [DS84] had implemented caching techniques to improve the performance of dynamic calls. A system method lookup uses an *(object, method name)* pair to find the callee. Inline caches remove this lookup by caching the fetch function at the call site. On the next call, a simple object check is all that is required to dispatch the function. Inline caching is reported to have up to a 95% success rate for Smalltalk [DS84]. The Self implementation extend this idea of caching. Holzle *et al.* [HCU91] presented polymorphic inline caching (PIC) in the Self system. The difference between a regular inline cache and the Self polymorphic cache is that the Self version is extensible and caches all calls from that site. When a call is made the call site checks all object types in the cache and dispatches the correct method. Should the number of entries in a PIC become large an efficient lookup would be necessary, perhaps an ordering based on program behaviour (for example move the most recently used method to the first line of the cache). The

success of PICs is entirely dependent on the behaviour of the program. For example, in the Self benchmarks, some benchmarks with large numbers of polymorphic method dispatches have lower inline-cache miss rates than those with fewer PIC dispatches.

Self's third and final implementation introduced the concept of run-time type feedback and incremental compilation [HU94b; HU94a]. The core idea behind the new system was that lazy compilation, that is delaying optimization until needed, saved compilation time and improved performance. The designers of Self wanted an interactive system. The motivation behind these advances was the existence of user perceived program *'pauses'* in the Self system. Lazy compilation combined with optimization based on run-time type feedback removed pauses and improved program optimization.

3.2.3 Prolog

Prolog is a declarative logic programming language. It was conceived by Alain Colmerauer [CR96] in Marseille, France in the early 1970's. The first system implementing Prolog was completed in 1975 by a team led by Alain Colmerauer and Phillipe Roussel. This initial implementation ran on a 16-bit mini computer, the T1600. The implementation, while correct, suffered from severe efficiency problems [CR96]. Prolog's use of dynamic typing, backtracking and unification placed high computational demands on the general purpose processor [HSRPD96]. In the years after the first implementation a number of efforts were made to optimize Prolog. Some of the optimizations created were later adopted by the scripting language community to solve similar efficiency problems. We give a brief overview of the evolution of Prolog optimizations and present details of the relevant optimizations adopted by the scripting language community.

In 1983, David Warren – a member of this first Prolog team, continued his efforts and defined an abstract instruction set (WAM) for the implementation of Prolog [War83]. The decision to define an abstract instruction set was motivated by the desire to implement the language in either hardware, firmware or software. Dobry *et al.* [DDP85] later designed the first co-processor implementing the WAM instruction set. This processor, the PLM, introduced a number of innovations to Prolog execution including the implementation of four stack areas. The usual environment stack and heap were augmented with separate stacks for unification and trail operations. Further, they introduced the concept of tagged pointers. A memory system where the most significant 4 or 6 -bits (depending on type) of a pointer stored the type information for that value. This concept of pointer tagging has been adopted in modern scripting language implementations like *l2j*, a Lua just-in-time compiler. Later work on optimization of the WAM instruction set focused on the optimization of backtracking

in Prolog. Van Roy *et al.* [VRDW87] presented optimization techniques which focused on avoiding backtracking by performing clause selection based on deterministic testing. This work was followed later by Benker *et al.*'s [BBD+89] Knowledge Crunching Machine (KCM). KCM was a 64-bit machine that used tagged pointers and backtrack avoidance for optimization. In the early 1990's, the Aquarius team at the University of California, Berkley created the VLSI-BAM processor [Hol90; Van92; Bru96]. They developed this processor with the hypothesis that the dual development of compiler and processor would produce a faster Prolog environment. They preformed experimentation on pointer tagging using both the most-significant and least-significants bytes of the pointer. They found that most-significant bit tags to be more efficient giving a 64% performance increase for a 13% increase in chip area. While all of these Prolog implementations were performing optimization of dynamic variables, none presented techniques for the runtime evaluation of variable types as later scripting language work attempts to do, including this work.

3.2.4 Scripting Language Specialization

The deployment of modern scripting languages differs from that of classical dynamic languages like Self and Prolog. They are typically used to extend a main application, often written in C/C++. They provide all the features of a standard dynamic language but have extra properties like *eval* statements which can evaluate (execute) a string of valid source code at run-time. Some research aims to analyze and/or compile scripting languages ahead of time [BdG09; FAFH09; JMT09]. Scripting language specialization refers to just-in-time program optimization based on concrete program properties discovered in some run-time profiling mechanism.

The first publication on the problem of run-time specialization was Rigo's Psyco [Rig04], a representation-based just-in-time specialization for the Python language. Psyco performs just-in-time specialization, or *'specialization by need'*, by promoting, or *'unlifting'*, a value from run-time into compile time. Specialization by need refers to the desire to avoid program structs like indirect function calls and large switches. Psyco performs well on computationally intensive arithmetic instructions but does not handle large, unalogrithmic applications well. A combination of some form of Self's PIC and psyco's tracing maybe a useful next step.

Apart from Psyco there have been two other papers on scripting language specialization. Both of these are motivated by the need for increased performance in the web space. Recent booms in internet *'web applications'* like Facebook, Twitter, Flickr, Gmail, and Google Maps have resulted in an increasing bottleneck in web performance.

This bottleneck is the execution of these application's client-sides written in JavaScript.

Chang *et al.* [CSR+09] wrote a trace based just-in-time compiler for Adobe's virtual machine called *Tamarin-Tracing*. Tamarin executes ActionScript, a JavaScript like language that is packaged inside Adobe's well known flash web player. Tamarin-Tracing is targeted towards both desktop and mobile environments requiring any tracing JIT to fit in a small amount of memory. Chang *et al.* choose Forth as an intermediate representation for their JIT. Its ease in refactoring and resulting small code size. The representation pipeline for Tamarin-Tracing starts with ActionScript bytecode that is transformed into Forth bytecode, Forth bytecode is then optimized and translated into their low-level intermediate representation for code generation. Tamarin-Tracing uses a standard profiling technique recording the execution of backward-branches. When a backward branch becomes hot the virtual machine enters a trace recording mode. Traces are built and specialized based on the run-time execution discovered during trace recording. Chang *et al.* presented three methods of trace compilation:

1. Trace fragments — each trace is independent of all other traces: Traces do not take advantage of the existence of any other trace. Thus, when specialization fails *bailout-code* brings execution back to the interpreter. Chang reports that bailout-code can represent up to 80% of generated code for trace fragments.

2. Trace linking — the case where two traces merge: Take the case where trace A jumps to trace B. If traces A and B act as trace fragments then trace A will save-state, execute, restore-state and jump to trace B which will in turn save-state, execute and restore-state. Saving and restoring state allows the virtual machine to branch to any form of execution, i.e. interpreter or JIT. Trace-linking removes the requirement for having trace A restore-state and trace B save-state. Instead, trace A jumps directly to some point in execution in trace B where their states are the same. Execution therefore never leaves optimized machine code.

3. Trace trees — a collection of traces that share the same entry point: Unlike linked traces, trace trees are compiled as one. Therefore, from our example above, opportunities for optimization can be made available across the trace boundary of A and B.

Tamarin-Tracing reports performance that is much slower than that of equivalent interpreting virtual machines *SquirrelFish* and *SpiderMonkey* for many of their benchmarks. They claim further performance will be found by spending more time in traced code and less time in the interpreter.

Gal *et al.* [GES+09] continue the work of just-in-time type specialization for scripting languages in their work on *SpiderMonkey*, the Firefox JavaScript engine. They also take a trace based approach and name their new virtual machine *TraceMonkey*. TraceMonkey's trace building is similar to Tamarin-Tracing. Profiling is again performed by recording backward branches. When a hot branch is found a trace is recorded and compiled. The next time the branch executes the trace is dispatched. TraceMonkey does not use Forth as an intermediate representation. Instead, it uses a *'trace-flavored SSA'*. A number of specializations can be performed on this representation:

- Local variable operations are specialized to the primitive types profiled during the record stage. Trace recording cannot guarantee that these types will reappear in all cases. Guards are required to check that these type specializations still hold during subsequent trace executions.
- JavaScript represents objects with maps. An object `o` can have a field `x`. To access `o.x` an interpreter first hashes `x` to find a property index. This index is then used to access the field in the property vector. This whole access can be specialized according to the type of the object `o`. Just like primitive types, trace recording cannot guarantee that the type of objects will reappear in all cases. Hence, similar guards are used to ensure correctness of these specialized field accesses.
- Variable representation specialization is used to specialize JavaScript number variables from doubles to integers. JavaScript has no integer type. All numbers are represented as doubles. However, operations such as array accesses and bit operators execute on integers. In interpretation these operations convert the double to an integer, perform the operation and then convert the result back to a double representation. Tracing specializes these run-time values to integers avoiding these conversions.
- Traces by definition are inter-procedural as they cross function boundaries, this results in the function inlining of called functions. Move instructions are added to pass parameters and return results but are later optimized away. Tracing must record all calls and returns in case specialization fails and the interpreter state has to be restored. The trace frame saves this information and restores it on exit from the trace.

A major attraction of tracing is the simplification of optimization. Web applications must be interactive, therefore JIT optimization must be fast. TraceMonkey performs

optimizations in two passes, one forward and one backward. On the forward pass, it performs constant sub-expression elimination, expression simplification and semantic-specific simplification (converting temporary doubles to integers). On the backward pass, it performs dead data-stack store elimination, dead call-stack store elimination and dead code elimination.

A major problem with tracing is that as program execution continues new traces are inevitably discovered. These traces may not be suitable for trace compilation or may connect or overlap with previous traces. Gal *et al.* [GES+09] present solutions for each of these scenarios.

Blacklisting is a mechanism used in TraceMonkey profiling to ignore untraceable regions. Traces usually fail to compile because of unsupported features in the implementation (in TraceMonkey's case exception handlers are not supported in trace compilation). If a trace is recorded but fails to compile, this fact is recorded by the virtual machine. The trace will be added to a blacklist and not recompiled for a number of iterations. This number is defined by a *'backoff'* counter set by TraceMonkey. When the counter reaches the backoff value the JIT compiler will attempt compilation again.

Trace Nesting occurs when the trace compiler compiles nested control flow structures. The problem of nested structures is that inner-traces are discovered and compiled first and later the outer trace is compiled resulting in recompilation and duplication of the inner-trace. The basic method for handling the problem is to continue trace recording out of the inner loop and up to the outer loop header. The outer loop thus becomes a child trace of the inner loop. This method has poor performance when there is more than one inner-trace path in a region. Gal *et al.* show that if loops are nested to k depth, and each loop has n paths, then there will be $O(n^k)$ traces. They present an alternative algorithm where loops are recorded as individual traces. When a trace exits a loop region, recording stops. If during trace recording they now reach the header of another loop, that loop must be an inner loop. Outer-loop tracing is suspended and the inner-loop is traced, if it has not already being traced. The outer-loop continues tracing and executes the inner-loop as a callable trace. The call site for the inner-trace requires a guard as execution of the trace cannot be guaranteed to have followed the same control flow as when it was recored and compiled.

Trace Stitching is TraceMonkey's mechanism for branching from a main trace into

a sub-trace discovered later in execution. The exiting trace is only required to store live variables back to its saved-state. The same saved-state is then reused by the sub-trace. While trace stitching is faster than the standard exiting and entering of traces, Gal *et al.* still report stitching to have a noticeable cost. The alternative solution is to merge the two traces in a new recompiled trace. However, Gal *et al.* also report that the number of traces can be in the worst case quadratic making the recompilation option prohibitive. They suggest, but have not implemented, a heuristic based recompilation scheme to recompile only when a recognized gain can be achieved by recompiling.

These three approaches (Psyco, Tamarin-Tracing and TraceMonkey) to just-in-time scripting compilation all contribute clever means to implement specialization optimizations. However, they have flaws, in particular, the profiling and recording approach of Tamarin-Tracing and TraceMonkey. Their approach performs specialization by speculation. We suggest that by investing more time in profiling and gathering information during interpretation a more constructive approach to compilation can be taken (e.g. compile regions that are profiled to be profitable). This investment in profiling will improve specialization and as a result performance. Chapter 6 of this dissertation presents a specialized interpreter we believe is a good starting point for a profiling system. Further discussion of our thoughts on the future of scripting specialization are presented in the final chapter.

Chapter 4

A Cell BE Java Interpreter

A significant challenge for the portability of virtual machine code is the growing popularity of multi-core architectures with specialized processing cores aimed at computation-intensive applications such as media processing. Such cores differ greatly in architectural design compared to traditional desktop processors. One such processing core is the Cell Broadband Engine's (Cell BE) Synergistic Processing Element (SPE). An SPE is a light weight VLIW processor core with a SIMD vector instruction set. In this chapter we investigate some popular interpreter optimizations and introduce new optimizations exploiting the special hardware properties offered by the Cell BE's SPE.

4.1 Introduction

The requirements of modern computing applications demand a diverse set of processing abilities, and increasingly processors are being designed with particular classes of application in mind. An example of this phenomenon is the Cell Broadband Engine (Cell BE)[KDH+05]. The Cell BE provides one general purpose superscalar core (the Power Processing Element, or PPE), and eight identical special-purpose, VLIW vector cores known as synergistic processing elements (SPE). These SPEs are optimized for multimedia, streaming and numerical computing.

The heterogeneous design of the Cell BE architecture provides a significant challenge in supporting languages such as Java and C# which are implemented with a virtual machine. The JavaVM allows code to be distributed in an architecture-neutral format, and execute in a safe, managed run-time. Although the Java virtual machine (JVM) can be implemented on the general purpose PPE core using known techniques, it is desirable that Java threads could also run on the special purpose SPEs. Such a solution provides a developer with a single abstract virtual machine as a target plat-

form. An advantage of such a virtual machine is that the execution details of whether to run on SPE or PPE can be decided at run-time by the virtual machine. Efficiently implementing the JVM on Cell BE SPEs is a significant challenge since the architecture is aimed primarily at regular vector code, rather than branch heavy code with irregular data access patterns.

The JavaVM is typically implemented with some combination of interpretation and just-in-time (JIT) compilation. A popular approach is to interpret code initially, and compile the code only when it has been identified as an execution hot spot. Interpreters are also commonly used where memory is limited, because an interpreter is normally much smaller and simpler than a JIT compiler.

This chapter presents a feasibility study for the interpretation of the most popular portable code, Java virtual machine bytecode, on the most pervasive high-performance system, the Cell BE. We evaluate a number of existing interpreter optimizations on the Cell BE SPE, and report the resulting effects on running time. We also propose a number of novel optimizations aimed specifically at the Cell BE SPE, and measure their performance.

The rest of this chapter is organized as follows. Section 4.2 provides some background information. Section 4.3 discusses the architectural features of the Cell BE SPE. Section 4.4 addresses the design issues for the construction of our JVM interpreter. Section 4.5 describes the experimental setup employed for analysing and testing our optimizations. Section 4.6 discusses the optimizations implemented in our virtual machine. Finally, Section 4.7 concludes the chapter.

4.2 Background

4.2.1 CellVM

CellVM (see Section 2.3.2) contains two VM cores. This chapter is concerned with the design and implemenation of the CoreVM interpreter. In this interpreter all virtual instructions are alligned to a 16 byte boundary. This allignement ensures efficient loading of instructions from the instruction stream. The first 32 bit slot of the instruction encoding acts as a padding, the second 32 bit slot is the address of the target instruction. The third and fourth slots are used as operand storage in those instructions where operand values are required, otherwise they act as padding for alignment purposes. Aligning to 16 byte boundaries is important for performance, as all loads and stores perform memory access on 16 byte boundaries and additional instructions would be

```
1 Cycle  Pipeline Schedule      Pseudo Assembler     C Code
2 01     12                     addi    pc,pc,1      pc++;
3 01     123456                 load    op1,sp[-1]   op1 = sp[-1];
4 02      234567                load    op2,sp[-2]   op2 = sp[-2];
5 03       34                   addi    sp,sp,-1     sp--;
6 08         ----89             add     op2,op2,op1  op2 = op2 + op1;
7 10             --012345       store   op2,sp[-1]   sp[-1] = op2;
8 11                123456      load    next,pc[0]   next = pc[0];
9 17                 -----7890  branch  next         goto next;
```

Figure 4.1: Pipeline schedule for virtual instruction IADD

required to move the value in the preferred slot. Section 4.6.2 provides more detail on operand storage and byte alignment and how the CoreVM interpreter accesses these values.

4.3 Architectural Features

The architectural features of the Cell BE SPE are targeted towards large compute-intensive tasks. Hence, the execution of instructions is strictly in-order and branch-prediction logics are not present. The instruction set of the SPE provides a rich set of vector operations meeting the high numerical requirements of scientific applications. Each SPE is an individual execution unit with its own program counter, registers and local store. This section briefly discusses some of these features, the impact they have on interpreter design and the difference between the Cell BE implementation to that of a standard general purpose processor.

4.3.1 Local Store

Modern architectures provide a memory hierarchy which includes a number of hardware controlled caches located close to the processing core. The Cell BE takes a different approach. Each SPE has 256KB of software controlled *Local Store* in which it stores all instructions and data used by the SPEs. A load instruction from local store has a 6 cycle latency. This is significantly higher than the level 1 cache hit latency of many processors, which is often 2 – 3 cycles [BE06].

Long *dependence chains* [1] over a small number of machine instructions are common among stack-based JVM instructions. Each phase of execution depends on values

[1] A series of instructions where each instruction is dependant on results from previous instructions.

resulting from the previous phase. The first phase of the code to implement a JVM instruction typically performs a load of some value(s), the second performs an operation on those values, the final phase then stores the result. With a six cycle per load latency, phase two will spend much time waiting for operands to load from the first execution phase. To demonstrate the significant costs of these loads and stores in an interpreter let us examine the schedule of a typical IADD implementation.

Figure 4.1 shows the profile of an IADD instruction schedule. The first column shows the cycle at which an instruction is issued. The second column displays the number of cycles an instruction requires to complete (with dashes indicating a stall in the issue of an instruction). The third column is a pseudo assembler listing for the corresponding SPE native assembler code, and column four is an equivalent C implementation.

From the profile in Figure 4.1 we can see 8 instructions are issued in 17 cycles resulting in 11 stalls (11 dashes). The reason for 11 stalls is due to the SPE's dual issue pipeline (see Section 4.3.4). The first two instructions are issued on the first cycle of execution, each down a separate pipeline. The store of the result back to memory produces two stalls, as it would have been possible to perform a dual issue here only for the data dependency between the addition and the store. The further 9 stalls are caused by one or more of the three loads contained in the code sequence. The first four stalls in the sequence preceded the addition (this is phase two of the instruction), the stalls occur because the values loaded in phase one have not yet been fully loaded. The additional five stalls occur in the loading of the program counter (PC) target address. These memory accesses account for over half of the processing of the IADD instruction.

4.3.2 Branch Predictors

Many high performance vector algorithms have few if any conditional branches. For this reason the SPE processor assumes sequential execution of software and so does not have a hardware branch predictor. Instead all branches are assumed not taken unless hinted otherwise by specialized *hint instructions* in the SPE instruction set architecture [EOO+05] . The hint instruction loads a *branch target buffer* (BTB) with two addresses. One address points to the location of the branch, the second to the target location which the branch is to be hinted towards. A correct hint allows the program to continue execution without penalty. An incorrect hint behaves just like a mispredicted branch in regular hardware, the pipeline must be flushed and the correct instruction stream fetched and executed. This flush has a penalty of 18 - 19 cycles [EOO+05]. Given that JVM instructions like IADD and ILOAD can take anything from 10 - 30 cycles to execute on a Cell BE SPE, a further stall of 18 cycles will have an

impact of approximately 50% increase in the total cycle count of the JVM instruction.

With hinting it is possible to achieve perfect prediction across all the interpreters dispatches, provided the target of the branch is known early enough. To avoid stalling the pipeline, the hint must be scheduled four instruction pairs and 11 cycles (i.e. approximately 15 cycles) before the corresponding branch [IBM07]. If the hint is scheduled later then the pipeline will stall even if the hint is correct.

4.3.3 Register Files

A Cell BE SPE has a 128-entry register file. Each register is 128 bits wide. This is a substantial register file and has been included on the SPE to mitigate the cost of replacing traditional caching for a single local storage unit. The significant size of the register file allows a Java interpreter to cache many more variables in registers and hence reduce the number of loads and stores in the interpreter. Many of our optimizations discussed in Section 4.6 utilize the register file to store novel pieces of information about the interpreter. For example, the pipeline technique, in Section 4.6.1, uses registers for program counter and operand values, stack caching, in Section 4.6.4, stores values from the top of stack in registers and JVM branch prediction, in Section 4.6.3, uses register values to make predictions about the direction a dispatch branch will take.

4.3.4 Dual Issue Pipeline

Each SPE has two pipelines (pipeline 0 and pipeline 1) into which it can issue up to two instructions per cycle. Each pipeline can receive a fixed set of instructions from the SPEs instruction set. The set of instructions issued to pipeline 0 include arithmetic operations and single and double precision floating point operations. Pipeline 1 receives instructions including loads, stores and branches. This technique allows, for example, an addition and a load to be performed in the same cycle. Dual issued instructions can be observed throughout the code listings of this chapter.

4.4 Design Considerations

The Cell BE SPE is a memory constrained VLIW in-order architecture. A successful interpreter optimization for this particular type of architecture is required to be memory efficient and sensitive to the in order nature of the execution pipeline. This requires the implementation for each opcode in the interpreter to be conscious of the data dependencies through the JVM Instruction Execution Cycle presented in Section 3.1.

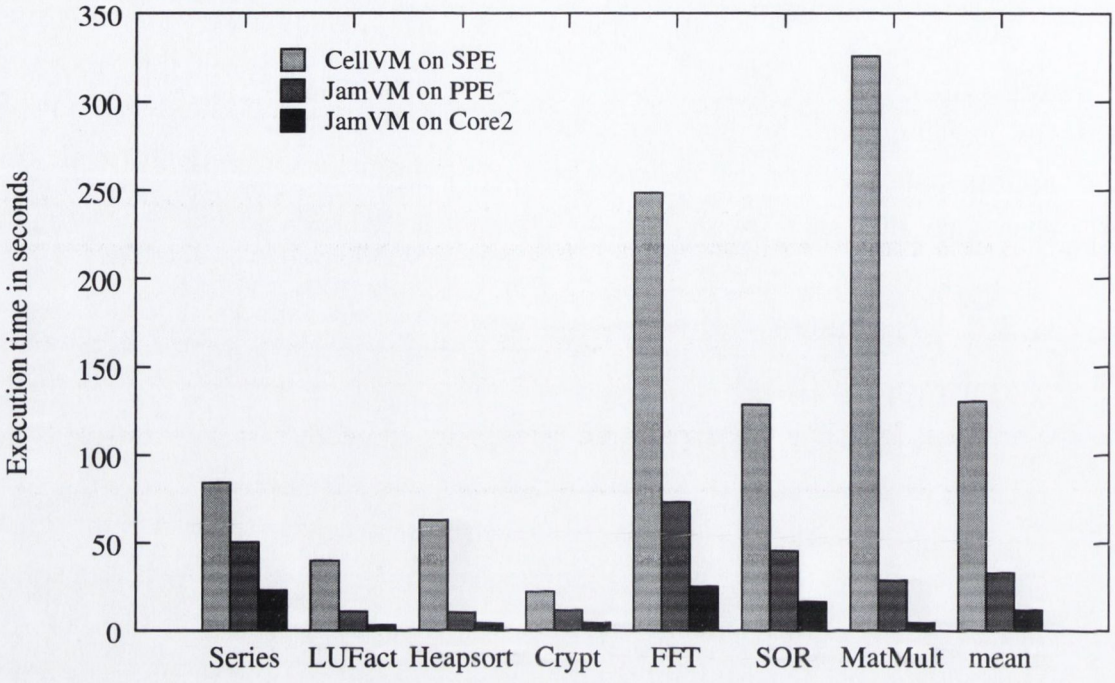


Figure 4.2: PPE interpreter versus SPE interpreter running times (smaller is better)

The Philips TriMedia VLIW Media Processor has some analogous features to the Cell BE SPE. Hoogerbrugge *et al.* [HA00] present a pipeline interpreter design in their paper for the Trimedia Processor. We have implemented their techniques and improved on the concept by adding operand fetching to the pipeline (see Section 4.6.1).

There are a number of interpreter optimizations presented in Chapter 3 that are not feasible for implementation on the Cell BE SPE. These include Ogata *et al.'s* [OKN02] bytecode fetch model and Ertl's [Ert95] multi-state stack caching techniques. These optimizations store a different interpreter for every state a virtual machine can enter. We have implemented a stack caching technique for a number of different caching levels but only store a single interpreter per implementation. Superinstructions lead to increased memory usage and so we have implemented a small set of possible instructions as an evaluation of the technique on the Cell BE SPE.

4.5 Experimental Setup

The experiments performed for this study were carried out on an IBM BladeCenter QS20 server running RedHat Linux (kernel 2.6.20-CBE). The PPE and SPE compiler used were *ppu-gcc* and *spu-gcc* (GCC) version 4.1.1 along with IBM Cell SDK 2.1. The benchmark suite used is the Java Grande Forums Version 2 Section 2 Size A. Figure 4.2

```
1  Cycle  Pipeline Schedule  Pseudo  Assembler        C Code
2  01     12                 move    pc_next,pc_pref  pc_next = pc_pref;
3  01     123456             load    op1,sp[-1]       op1 = sp[-1];
4  02      234567            load    op2,sp[-2]       op2 = sp[-2];
5  03       34               addi    sp,sp,-1         sp--;
6  03       345678           hint    LABEL,pc_next
7  04        456789          load    pc_pref,pc[2]    pc_pref = pc[2];
8  05         56             addi    pc,pc,1          pc++;
9  08          --89          add     op2,op2,op1      op2 = op2 + op1;
10 10             -012345     store   op2,sp[-1]       sp[-1] = op2;
11                            LABEL:                   LABEL:
12 11                1234     branch  pc_next          goto pc_next;
```

Figure 4.3: Pipeline schedule for virtual instruction IADD with PC fetch pipelining

plots timing results for JamVM running on the Cell BE PPE, Intel Core2 (2.4GHz) and CellVM running on a single SPE. Both JamVM running on the Cell BE PPE and Intel Core2 significantly outperform CellVM, its port to the Cell BE SPE. This graph outlines how difficult the interpretation of Java bytecode is on the Cell BE SPE.

The interpreter was written in C to allow testing of different variations, hence the performance of the interpreter is dependent on the compiler used. Observations about bottlenecks and latencies were made from assembler listings and performance results provided by the IBM *spu_timing* tool in IBM Cell SDK 2.1. Code listings from this tool are provided throughout Section 4.6 of this chapter in order to illustrate design decisions and performance results. For readability purposes assembler code is listed in a pseudo assembler format and commented with the equivalent C code listings.

4.6 Optimizations

This section is split into five different sections each detailing a different set of optimization strategies. Section 4.6.1 investigates a technique first proposed by Hoogerbrugge *et al.* [HA00] called interpreter pipelining. Section 4.6.3 presents some unique techniques for the dispatch of JVM instructions. Section 4.6.4 presents stack caching results. Section 4.6.5 presents some code merging techniques and finally a set of superinstructions are investigated in Section 4.6.6. Performance results are presented for each optimization throughout the rest of this section.

4.6.1 Interpreter Pipelining

The instruction dispatch phase in the JVM instruction's life cycle has a dependence chain that includes the program counter increment (addition — 2 cycle latency), the program counter fetch (load from memory — 6 cycle latency) and the instruction dispatch (indirect branch — 4 cycle latency and 18 - 19 cycle mispredict penalty). The location of the program counter fetch can be moved around the interpreter to allow software pipelining of the JVM instructions execution cycle. By performing this rescheduling, the dependence chains that produce many of the latencies identified in the JVM Instruction IADD are broken (see Figure 4.1). Thus, the compiler is given room to produce improved scheduled code and the dispatch phase of the instruction cycle can be issued without a data dependant delay.

Hoogerbrugge's techniques decreased VLIW instruction count by up to 19.4% and cycle count by up to 14.4%. The TriMedia VLIW Media Processor has an issue width of five and a jump latency of four cycles allowing fifteen operations to be scheduled in three delayed slots after the jump and another four in parallel with the jump. Cell BE SPE has a different approach to branches. Its hinting scheme requires four instruction pairs and eleven cycles to be executed between a hint and a branch. This hint-to-branch latency requires the interpreter to access the dispatch value very early in the execution of a JVM instruction, pipelining the instruction dispatch phase enables this access.

In the code sample of Figure 4.1 the program counter for the next instruction is loaded immediately before the indirect branch dispatching the next instruction. This causes a large penalty due to memory latency and a further penalty for a mispredicted branch. The first instruction in Figure 4.3 defines the program counter for the dispatch phase of the current JVM instruction. The value is copied from a prefetch variable defined in the previous JVM instruction. This value is subsequently used to throw a correct hint to the dispatch instruction, *branch*. Figure 4.4 shows the speed-ups for this technique over all our benchmarks.

4.6.2 Operand Pipelining

To allow for efficient loading of bytecodes, the program's instruction stream is aligned to a 16 byte boundary on the Cell BE SPE local store. The instruction's bytecode is stored in the *'preferred slot'* (the second value in a 32 bit vector on which scalar operations are performed) of the 16 bytes while the operand is stored in the third slot of the vector. To access the operand the 16 bytes must be loaded into memory (6

```
1  Cycle Pipeline Schedule     Pseudo Assembler        C Code
2  01    12                    move   pc_next,pc_pref pc_next=pc_pref;
3  02     234567               load   pc_cur,pc[0]    pc_cur=pc[0];
4  03      345678              load   pc_pref = pc[2] pc_pref=pc[2];
5  04       456789             hint   LABEL,pc_next
6  05        56                addi   pc,pc,1          pc++;
7  08          --8901          rotate op,pc_cur,8     op=pc_cur->opr;
8  12             ---2345      shift  op,op,4          op=op << 4;
9  16                ---678901 load   local,op,lvars  local=lvars[op];
10 22    234567          ----- store  local,sp[0]     sp[0]=local;
11 23     34                   addi   sp,sp,1         sp++;
12                             LABEL:
13 23     3456                 branch pc_next         goto pc_next;
```

(a) ILOAD with Program Counter Pipelining

```
1  Cycle Pipeline Schedule  Pseudo Assembler        C Code
2  01    12                 move   pc_next,pc_pref  pc_next=pc_pref;
3  02     234567            load   pc_tmp = pc[2]   pc_tmp=pc[2];
4  03      3456             shift  op,op_pref,4     op=op_pref << 4;
5  04       4567            rotate op_pref,pc_tmp,8 op_pref=pc_tmp.op;
6  05        56             addi   pc,pc,1          pc++;
7  06         678901        hint   LABEL,pc_next
8  07          789012       load   local,op,lvars   local=lvars[op];
9  08           89          move   pc_pref,pc_tmp   pc_pref=pc_tmp;
10 13           -----345678 store  local,sp[0]      sp[0]=local;
11 14                 45    addi   sp,sp,1         sp++;
12                          LABEL:
13 14                 4567  branch pc_next         goto pc_next;
```

(b) ILOAD with Program Counter and Operand Pipelining

```
1  Cycle Pipeline Schedule Pseudo Assembler          C Code
2  01    12                move   pc_next,pc_pref    pc_next=pc_pref;
3  02     234567           load   pc_tmp = pc[2]     pc_tmp=pc[2];
4  03      34              addi   pc,pc,1           pc++;
5  04       456789         load   local,op_pref,lvar local=lvar[op_pref];
6  05        567890        hint   LABEL,pc_next
7  06         6789         rotate op_pref,pc_tmp,8  op_pref=pc_tmp.op;
8  08          -89         move   pc_pref,pc_tmp    pc_pref=pc_tmp;
9  10            -012345   store  local,sp[0]       sp[0]=local;
10 11               12     addi   sp,sp,1           sp++;
11                         LABEL:
12 12                2345  branch pc_next           goto pc_next;
```

(c) ILOAD with Program Counter and Operand Pipelining removing the shift

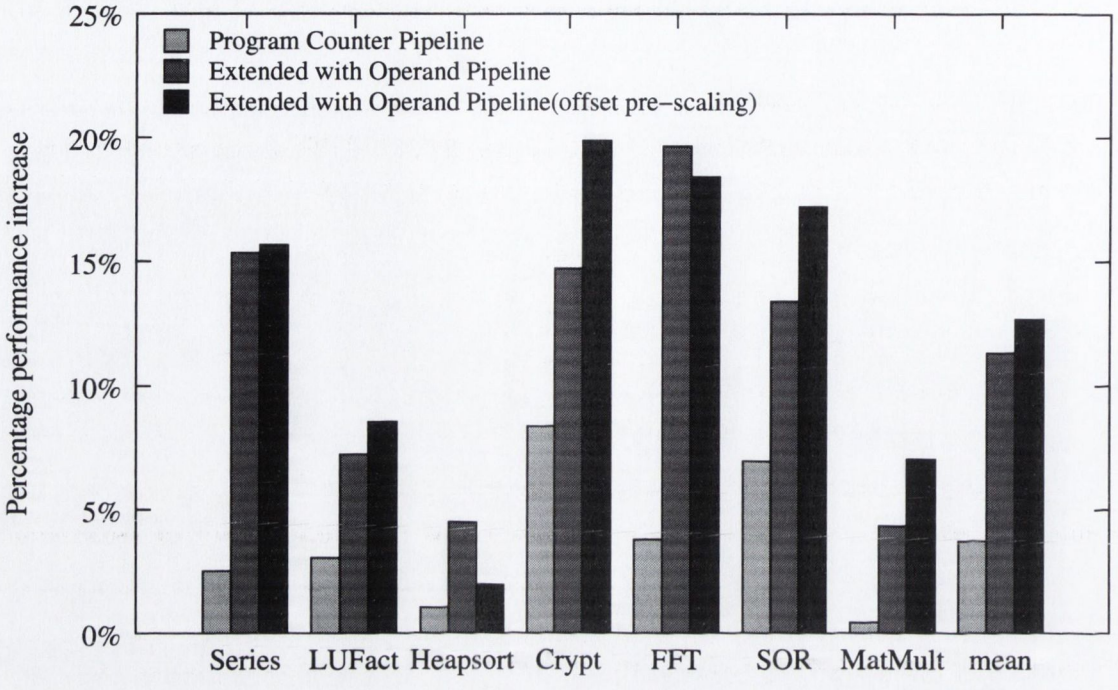


Figure 4.4: Pipelining performance increases (bigger is better)

cycles), a rotate instruction must align the third slot into the preferred slot (4 cycles) and for the purposes of the JVM instruction ILOAD, a shift instruction must scale the operand for offsetting the local variable array (4 cycles). This dependence chain of 14 cycles is illustrated in Figure 4.6.1 (a) where there are 2, 3 and 3 stalls preceding each dependence in this chain.

Operand pipelining removes this dependence in its entirety. Extending program counter pipelining the 16 bytes of the instruction stream are pre-loaded in registers removing the need to perform a load for the operand value. The rotate stage to align the operand is performed in the previous instruction. The benefits of this schedule are illustrated in Figure 4.6.1 (b) where the stall cycles have been removed and the total cycle count of ILOAD is reduced from 23 to 14.

The final optimization of the ILOAD instruction is to pre-scale the operand value in the code preparation stage of the JVM. As illustrated in Figure 4.6.1 (c) this allows the prefetched operand value to be immediately used to offset the local variable array and perform the load from memory of the necessary value.

Figure 4.4 graphs the results for operand fetch pipelining. As ILOAD is the most commonly executed JVM instruction [KCSL00] a substantial increase in speed is observed when operand pipelining optimizes both it and other instructions with an operand value.

4.6.3 JVM Branch Prediction

A number of JVM instructions compute their branch target from operands (e.g. JSR, GOTO) or from conditional statements comparing values from the operand stack (e.g. IFEQ, IF_ICMPLT). In these cases the pipeline chain fetching the program counter in the previous instruction is broken and an accurate hint can be issued only at a very late stage of execution. In this schedule the hint latency illustrated in Section 4.3.2 will reduce the effectiveness of a branch hint by introducing stalls to the issue of the indirect branch at the dispatch stage of execution.

If a prediction can be made earlier on in the execution cycle a saving of the full 18 - 19 cycles can be made. Regular processors use the techniques presented in Section 2.1.3 to solve this issue. This project has implemented a one-bit predictor inside the Java virtual machine interpreter.

One-bit predictors issue predictions based on a heuristic giving a single piece of information (e.g. direction of the last branch or the type of branch - loop, conditional statement). A number of different heuristics were analysed with the results graphed in Figure 4.5 (a), they include:

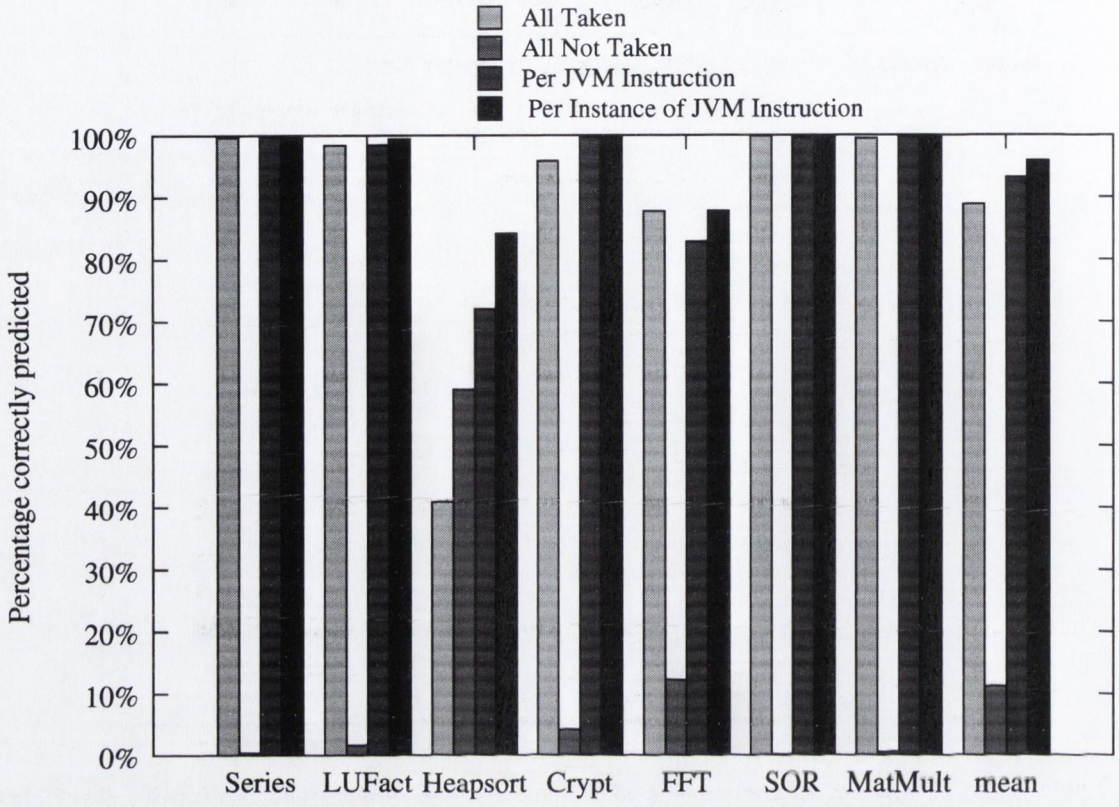
1. Taken - All branches are hinted as taken. The hint value is the operand value of the JVM instruction.

2. Not Taken - All branches are hinted as not taken. The hint value is the value of the next instruction in the instruction stream.

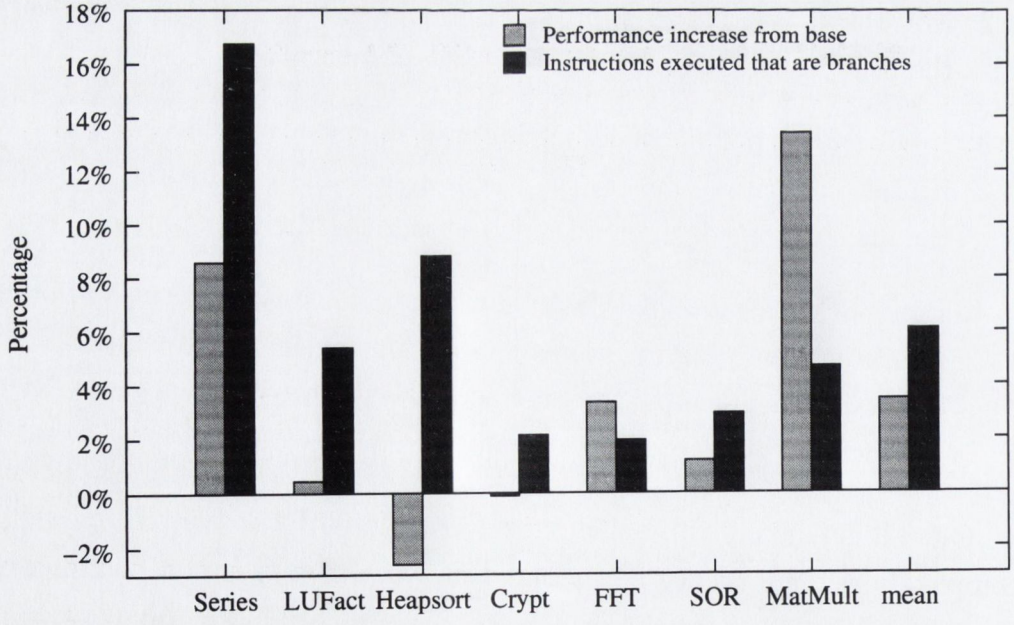
3. Predict Per JVM Instruction - Each JVM instruction has a register value which it uses to statically hint at the beginning of each instruction. On each execution the register value is updated with the target address computed for that iteration.

4. Predict Per Instance of JVM Instruction - Each instance of a JVM branch instruction in a given program stores its own hint value in an operand. Each iteration of the instruction updates the operand with the target address for the current instruction.

One-bit predictors will achieve on average a 50% success rate in a program with completely random branch behaviour. However, most programs contain branches that control loop iterations and other flow control that are highly predictable. In our final two implementations, the branch at the end of a standard non-nested loop will be miss-predicted twice, in the first iteration and the final iteration. All other phases of the loop will be correctly predicted.



(a) Rates of correct predictions for each prediction technique.



(b) 'Predict Per JVM Instruction': the left bar graphs the percentage increase in run timing for each benchmark. The right bar charts the percentage of instructions in a given benchmark that are branch instructions.

Figure 4.5: JVM Branch Prediction Evaluation

Both implementations for (3) and (4) achieve extremely similar high prediction rates on the Java Grande Section 2 benchmarks (see Figure 4.5 (a)). Because of the penalty the final implementation incurs accessing its second operand value (prefetching of two operand values is not implemented in interpreter pipelining) the performance results for it are lower than that of the first implementation. As a result, performance figures for the third heuristic are presented in Figure 4.5 (b) which charts the percentage of instructions that are branches alongside the percentage speed up achieved by the predictor. By examining the Heapsort and Matrix Multiplication benchmarks we can see a relationship between the number of branches, the prediction rate and the performance of a program. A program with a large number of branches has a speed increase directly related to the predictability of the branches. JVM branch prediction has a negative impact on unpredictable programs like Heapsort. Heapsort has many branches and as seen in Figure 4.5 (a) these branches are less predictable than other benchmarks. The overhead taken to produce a prediction is larger than the improvement given by predicting branches correctly. We therefore have a negative speed up for heapsort. In Matrix Multiplication on the other hand we have few branches but as they are highly predictable we see a large performance increase.

4.6.4 Top Of Stack Caching

The discussion of Cell BE SPE's local store in Section 4.3.1 illustrates the expense of loading values from the stack into registers. In Section 3.1.4, we presented the stack caching interpreter optimization. Since the local store of the SPE is limited (256kB) and is used for both code and data we choose the static stack caching design for our interpreter. Three different levels of caching were measured – the top one, two and four variables of the stack.

Figure 4.7 illustrates the effect of stack caching one variable on the cycle count and pipeline stalls for the JVM instruction IADD. In Figure 4.1, preceding the addition of the two values are two load instructions. These loads move operand values from the stack into registers. A further six cycles are then required before issuing the addition. In total ten cycles pass before the addition instruction is executed. In Figure 4.7 the top value of the stack is cached in a register – *top1*. The second value is loaded from the top of the stack in local store. The result of this addition is stored back into the top of stack register, *top1*, removing the need for a further store instruction to push the value back onto the top of stack.

In total the stack cached IADD operation takes 10 cycles compared to 21 in the original IADD. This reduction in cycles causes scheduling of the hint to become more

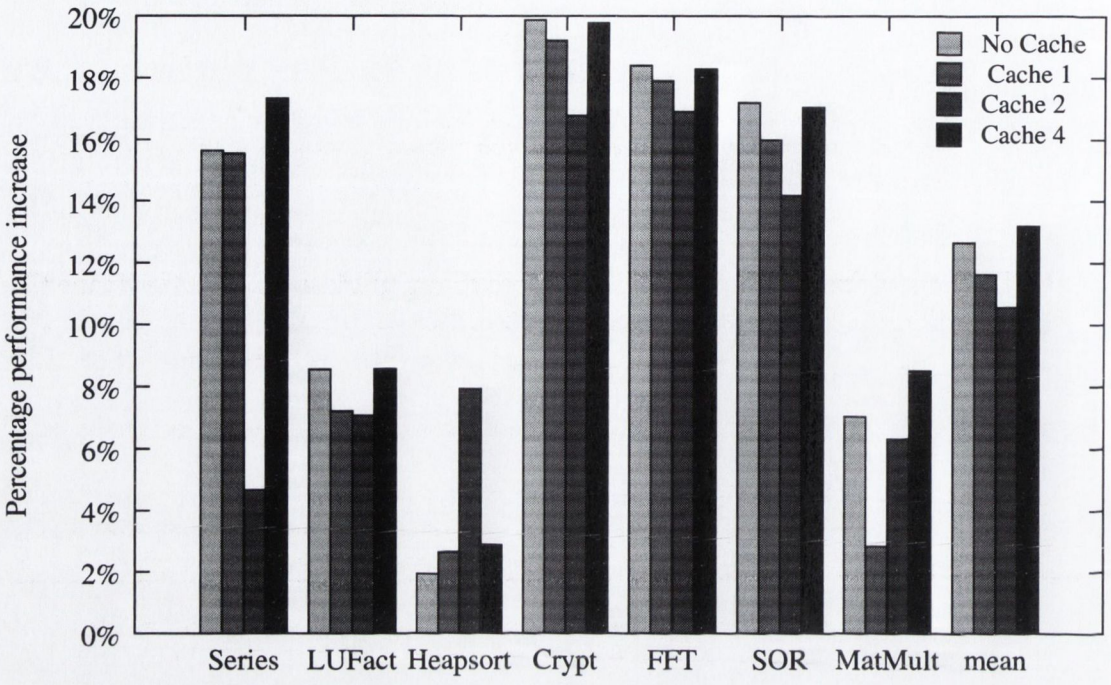


Figure 4.6: Stack caching performance increases

difficult for the compiler. There are fewer cycles to schedule a hint in and the required approximation of 15 cycles for a hint to branch are not present. The effect of this scheduling challenge on run-time is graphed in Figure 4.6. For the majority of benchmarks stack caching produces a small variance in performance. The variance in time depends on the level of caching performed. For example, caching one value reduces the instruction count and the number of cycles available for scheduling hints. This produces a less efficient hint schedule and hence a slight decrease in running time over many benchmarks. In comparison caching four variables increases the instruction count as the interpreter has an increased number of variables to manage. This is the most effective caching level across all our benchmarks as the increased instruction count allows for more efficient scheduling of the branch to hint.

4.6.5 Merged Operators

The bottleneck of execution for the majority of JVM instructions in the pipelined interpreter presented in Section 4.6.1 is the hint to branch latency. This latency requires 4 instruction pairs and 11 cycles of processing between a hint and a corresponding branch. This bottleneck allows for more work to be performed in this *stall window* than otherwise possible. A traditional way to increase the work load of instructions

```
1  Cycle  Pipeline Schedule  Pseudo  Assembler          C Code
2  01     12                 move    pc_next,pc_pref    pc_next = pc_pref;
3  01     1234               load    op2,sp[-1]         op2 = sp[-1];
4  02      23                move    op1,top1           op1 = top1;
5  02      234567            load    pc_temp,pc[2]      pc_temp = pc[2];
6  03       34               addi    sp,sp,-1           sp--;
7  03       345678           hint    LABEL,pc_next
8  04        45              addi    pc,pc,1            pc++;
9  04        4567            rotate  op,pc_pref,8       op = pc_pref->opr;
10 07         --78           add     op1,op1,op2        op1 = op1 + op2;
11 08            89          move    op,opfetch         op = opfetch;
12 09              90        move    top1,op1           top1 = op1;
13 09              9012      move    pc_pref,pc_temp    pc_pref = pc_temp;
14                           LABEL:
15 10               0123     branch  pc_next            goto pc_next;
```

Figure 4.7: Pipeline schedule for virtual instruction IADD with stack caching of one variable

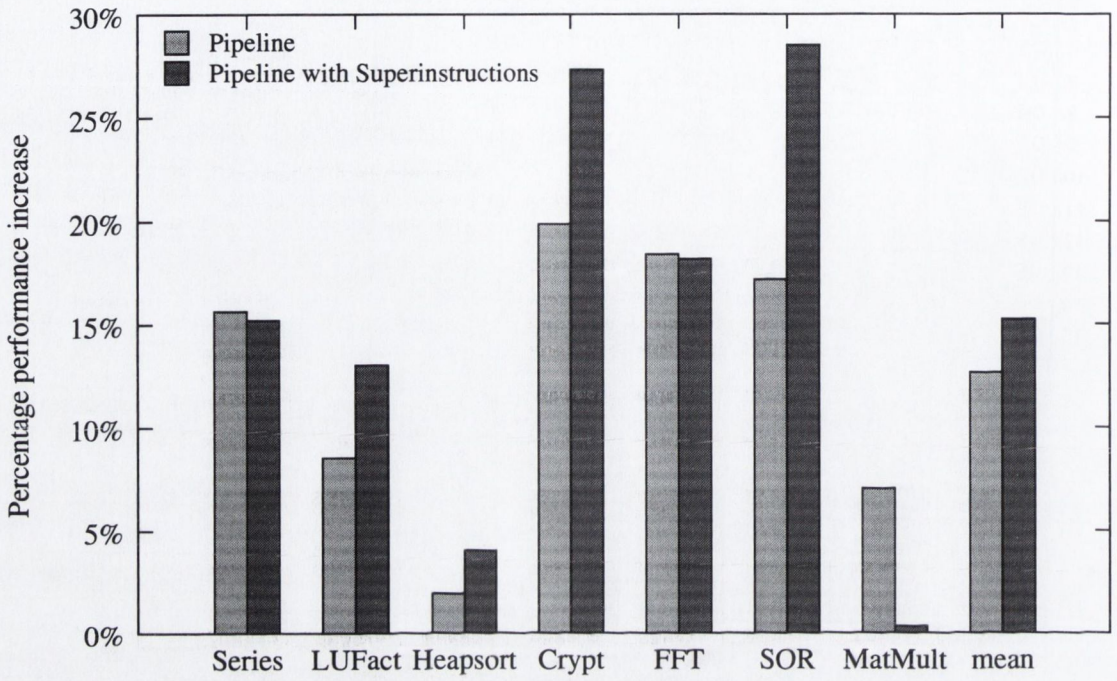
per dispatch, as proposed here, is to introduce superinstructions. As discussed in Section 3.1.4, this process increases the code size as extra instructions are added.

We observe that an alternative process can be performed where by different JVM instructions with similar characteristics in operand fetching and instruction dispatch are co-located in a single interpreter case. This process has two distinct advantages:

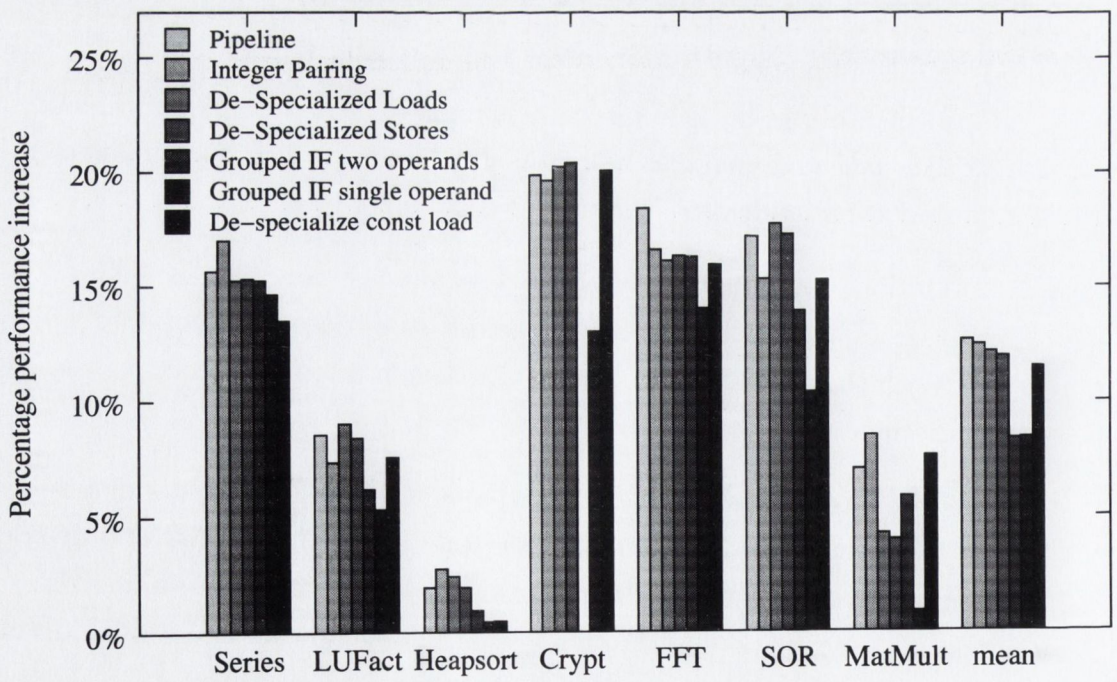
1. JVM Instruction Reduction: The number of instructions requiring individual implementations in the interpreter is reduced. For example, arithmetic instructions like *iadd* and *isub* can be grouped together, reducing the number of arithmetic operation implementations required from 14 to 8.

2. Options for Superinstructions: The number of schedules that can take advantage of superinstructions is increased as options become more widely available. With the merging of instruction implementations sequences of these instructions become more common.

As the latency costs of loading and operating on JVM operands has been removed from the interpreter with our pipeline techniques, the merged and de-specialized instructions tested in our implementation have shown minor changes in performance across all benchmarks (see Figure 4.8(a)). The following subsections discuss the individual groupings tested in our implementation.



(a) Superinstruction performance increases



(b) Instruction merging performance increases

Figure 4.8: Superinstruction and Instruction merging performance increases

```
1  Cycle Pipeline Schedule  Pseudo Assembler          C Code
2  01    12                 move   pc_next,pc_pref   pc_next = pc_pref;
3  01    123456             load   op1,sp[-1]        op1 = sp[-1];
4  02     23                ceqi   cond,opftch,96    cond=(opftch==96);
5  02     234567            load   op2,sp[-2]        op2 = sp[-2];
6  03      34               addi   sp,sp,-1          sp--;
7  03      345678           hint   LABEL,pc_next
8  04       4567            rotate opftch,pc_pref,8 opftch = pc_pref.op;
9  05        567890         load   pc_pref,pc[2]     pc_pref = pc[2];
10 06         67            addi   pc,pc,1           pc++;
11 08          -89          add    r1,op1,op2        r1 = op1 + op2;
12 09            90         sub    r2,op1,op2        r2 = op1 - op2;
13 11             -12       select r1,r1,r2,cond     r1 = cond ? r1 : r2;
14 13                -345678 store  r1,sp[-1]         r1 = sp[-1];
15                           LABEL:
16 14                  4567  branch pc_next          goto pc_next;
```

Figure 4.9: Pipeline schedule for merged-instruction IADD_ISUB with program counter and operand pipelining

Integer Operation Pairing

There are 14 integer operations in the Java virtual machine. Figure 4.9 is an example of one of these pairings. Here the execution phase of the instruction calculates both the integer addition and subtraction result for the two operands on the top of stack. A single specialized instruction from the SPE Instruction Set Architecture allows a conditional assignment to store the desired result back onto the top of stack. The condition for this assignment is the operand value of the virtual machine instruction. This is set to either the opcode value for IADD or ISUB depending on what operation is required.

Conditional Instruction Grouping

A similar scheme to integer operation pairing can be used to group conditional statements together. Each conditional instruction firstly reads either one or two operands from the operand stack. In the case of one operand, it is compared to zero using an operator specific to the operation. Similarly for two operands, the operands are compared to each other using a specific operation for that virtual machine instruction. Next, the condition is examined to be true or false and assignment to a program counter is performed, either from the operand of the instruction (the target) when the condition is true, or the next program counter in the instruction stream (the fall through path) when the condition is false.

In total there are twelve conditional virtual instructions. This grouping technique

forms these JVM instructions into two groups, one for single stack value operand instructions, the other for double stack value operand instructions. For each of the six possible conditions to be compared a result for that comparison is defined. To choose the correct condition a series of conditional assignments using the second operand to the virtual instruction as a condition. This second operand is an integer representing the instruction's byte code value. This series of conditional assignments will result in a correct condition value. The program counter can now be defined as in a standard implementation.

De-specializing LOAD / STORES

A common technique for reducing the total number of JVM instructions is to reduce the number of 'specialized' JVM instructions. Specialized instructions are instructions that have had their operand value incorporated into the virtual machine instruction, for example ILOAD_0 has had its operand '0' defined as part of the instruction. De-specializing is the process of removing these types of instructions and replacing them with an equivalent virtual instruction and its correct operand. In total there are eight integer load and stores with a number of long and double loads and stores too. For every different group we can reduce the number of JVM instructions required for it down to one.

Constant Load Grouping

In total there are fourteen instructions in the CellVM interpreter that simply load a constant value to the top of stack. Ten of these fourteen instructions are specialized instructions, for example the virtual instruction ICONST_0 loads the constant value 0 to the top of the stack. The others load the constant value defined in their operand to the top of stack. Constant load grouping removes all these instructions and replaces them with a single instruction that loads a constant defined by the JVM instructions operand to the top of stack. The specialized instructions in the group are de-specialized and their load values are defined in their operand.

4.6.6 Superinstructions

In our interpreter, we have taken a modified approach to superinstructions by merging unrelated instructions and hence creating an even more common pool of instructions to choose from. To demonstrate the performance of superinstructions on our interpreter a number of instructions were grouped in pairs as superinstructions (see Figure 4.8(a)).

Matrix multiplication achieved a negative speed up from our pipelined version, this negative speed is a result of the combined impact of the negative speed up from the merged instructions in Figure 4.8(a). Merging of conditional instructions and de-specialized loads and stores have had a greater impact on matrix multiplication than the savings found through the superinstructions implemented that were applicable to matrix multiplication. Other benchmarks such as *Crypt* and *LU Factorization* achieve in the order of 5-7% increase over the pipelined interpreter, giving a total performance increase of approximately 23% for both benchmarks.

4.7 Discussion

This chapter has presented analysis of existing interpreter optimization techniques on the Cell BE Processor and introduced novel optimizations made possible by the architectural features of the Cell BE SPE. Firstly, interpreter pipeline techniques first published by Hoogerbrugge *et al.* and tested on the Trimedia VLIW processor were tested and shown to provide increases in performance on the Cell BE. Further, these techniques were extended by introducing an optimization referred to as operand pipelining in which the existing pipeline structure was extended by performing a decode stage in which operands were fetched in advance. This technique proved extremely successful providing a speed increase of just under 20% on our best performing benchmark. Secondly, a novel optimization for the dispatch of conditional JVM instructions was described. This technique used the hint feature of the Cell BE to improve the dispatch performance of JVM instructions that compute their control flow target. Finally, the limitations of the Cell BE's abilities to interpret JVM bytecodes efficiently were identified. They include the large hint to branch stall (approximately 15 cycles) and the penalty incurred by expensive load and stores (6 cycles). The opportunities of the first limitation were demonstrated through a useful technique that eliminated the number of JVM instruction implementations required by performing a merging and pairing optimization co-locating similar instructions in the same implementation. These merged instructions were subsequently used to provide a rich environment for producing superinstructions combinations that produced an increase in speed over many benchmarks. Through the use of software pipelining, an efficient pipelined interpreter was designed that removed the load and store limitation of program counter fetch and decode as well as prefetching operands.

Chapter 5

A Portable Mixed-Mode Lua virtual machine

Dynamic scripting languages are most commonly implemented using interpreters. These interpreters are highly portable but lack the performance required for use in demanding systems. Just-in-time (JIT) compilation has been used to improve the performance of some of these dynamic scripting languages. JIT compilers typically target a single platform and thus remove the portability of the original implementation. In this chapter, we present a portable approach to JIT compilation for scripting languages. At run-time we generate ANSI C code and use the system's native C compiler to compile this code. We present a profiling interpreter that ensures the type safe linking and dispatch of just-in-time compiled functions.

5.1 Introduction

Mixed-mode interpreter/JIT systems are a popular method of increasing the run-time performance of portable interpreted scripting languages. Commonly, systems use a simple method execution count and perform whole method compilation when this count reaches a certain threshold. This can result in the compilation of large amounts of rarely executed code and produce a *'big bang'* pattern of program performance [ZBS07]. Section 3.2.1 presented alternatives to this whole method compilation. This chapter proposes the adoption of some of the partial method techniques for use in the compilation of dynamically typed scripting languages. In order to generate efficient code in these dynamic systems, type knowledge must be gathered from the run-time virtual machine and propagated at JIT time. We recognize a potential for greater type knowledge when profiling both whole and partial-methods. Type knowledge is not just

constrained to parameters but rather, at a partial method level, to all initialized local variables. Our signature scheme presented in Section 5.3.1 provides a safe and accurate mechanism to perform type profiling of function and partial function regions of code.

In order to guarantee type safety within a run-time environment, specialized compiled regions must be managed and dispatched only when it is safe to do so. The first motivation for our work is to create an efficient profiling mechanism to profile running programs at multiple levels (both whole method and partial method) of granularity. Decisions to safely compile, link and dispatch are based on frequency counts and flags of recognized regions executing under a known set of run-time types. Our second motivation is to allow the compilation of JIT code by an ahead-of-time compiler with full optimizations by implementing an efficient multi-threaded approach to compilation.

5.1.1 Advantages of JITing with C

Just-in-time compilers are large, complex pieces of software. They can even become many times more complex than the original virtual machine of a small language. Industrial JIT compilers written by IBM [SOT+00] and Sun [PVC01] were created by teams with tens of programmers. They require additional modification for every target platform they support.

Scripting languages are typically maintained by a smaller group of programmers. To these small groups, portability is hugely important due to the man-hour cost of maintaining versions of code targeted at different platforms. For these reasons, generating a portable object code (ANSI C [Sch90]) at JIT time and compiling with existing software is an attractive option for improving the performance of a scripting language across many platforms.

The most popular native C compilers are constantly under development by the compiler community. A scripting language augmented with a portable JIT will therefore be gaining from the incremental performance increases developed by the individual compiler communities.

5.1.2 Contributions

This chapter makes the following contributions: (1) we adopt the paradigm of finer compilation unit granularity for improved code generation and optimization. We extend previous work on dynamic partial method compilation in statically typed programming languages and use similar techniques to compile dynamically typed languages with the intent of removing run-time type checks. (2) Dynamically typed scripting languages

have variables which can change type at any assignment during program execution. A specialized region of code therefore could become invalid from one call to the next. We show a practical solution for the storage and management of specialized code regions. These techniques ensure type safe dispatch of specialized code.

5.2 Background

5.2.1 Mixed-mode Interpretation & Dynamic Compilation

As discussed in Chapter 2, dynamically typed scripting languages are typically implemented using interpreters. They are highly portable and allow implementations to target many different platforms easily; they are also typically slow and significantly under-perform compared to a native implementation of a program executing on hardware. To negate this performance loss, virtual machine vendors have designed mixed-mode run-time systems. The majority of these systems are targeted towards the Java language with some recent developments within the JavaScript community. These systems begin by interpreting a program and when a *'hot'* method is identified, compile it just-in-time and execute the native code. Dynamic compilation systems generally lack portability as they are targeted toward a particular native instruction set. Our mixed-mode implementation uses a modified execution loop (see Figure 5.2) to profile for common whole-methods and partial-methods — when a unit has been identified, a compilation is requested and further calls are implemented using the resulting native function.

5.2.2 Dynamic Typing

All high level programming languages have type systems. A type system defines a set of valid values and a set of operations that can be applied to those values. The type system allows high level languages to detect and prevent invalid operations such as an arithmetic operation between two strings. The types of variables can be defined statically at compile time, or dynamically at run-time. Languages such as C and Java have static type systems whereas Perl, Python and Lua are dynamically typed. In a statically typed language, the type of a value that a variable can contain is determined at compile time. In dynamically typed languages however, variables can contain values of any type and each time an operation is applied to a variable during program execution the type of the variable must be checked. This typically adds a significant overhead to executing programs in dynamically typed languages. A goal of much re-

search is to reduce or eliminate dynamic type checks in compilers or virtual machines for dynamically typed languages.

5.2.3 Compilation Units

An important question for any JIT compiler is the decision of what size and shape (function, loop, etc) a unit of compiled code will be. Our JIT compiler profiles for and compiles at both method and partial-method levels.

We define a partial method as an intra-method region of code. These regions are segments of code within a method which could be considered *hot*, as they are commonly executed. For example, a `for` loop.

We define a program trace as an inter-method region of code that exceeds method boundaries and represents a commonly executed path through a whole program. For example, in a section of code with an exception handler the exception code would not be considered part of commonly executed program trace.

Specialization of a program is the transformation of a high-level representation of that program into a more specialized lower-level representation with the goal of improving the performance of the program by expressing the program with. When referring to the specialization of a dynamically typed program we express an intention to implement a specialized representation of a program for a given set of variable types.

5.3 Profiling Architecture

In our mixed-mode Lua virtual machine, execution of a program begins with the profiling interpreter. This interpreter is a modified version of the original interpreter. Profiling *side-exits* have been added to each function call and backward branch. Each Lua function in the system is provided with a signature list data structure which is used to store profile information. A side-exit creates a signature entry in the list of signatures for the executing function — if one does not exist — and increments the profile count of the signature. A profile signature includes information about the instruction being profiled, the set of local types in the call stack as well as a profile count and an executing function. The executing function is initialized as the profiling interpreter. As the profile count of the signature increases and the function or region is JITed, the resulting function is dynamically linked and stored in the signature. All subsequent calls from the signature now dispatch JIT code rather than the original interpreter. Similarly, should a JITed function be required to call a non-JITed Lua function, the

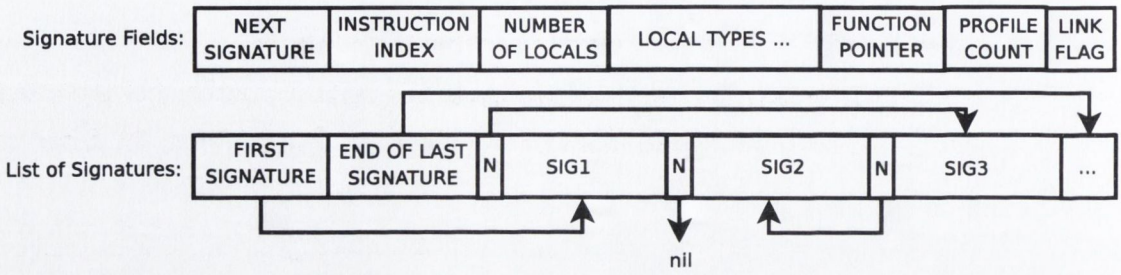


Figure 5.1: The upper figure shows the fields of an individual signature, they are (1) pointer to the next signature, (2) the index of the call or backward branch instruction, (3) the number of local variables, (4) the list of local variable types, (5) a pointer to the function that executes the Lua code from this point, (6) the profile counter, (7) a link flag byte which is written to by the compile thread when compilation completes — this flags the virtual machine to link the function on its next call. The lower figure sketches an example signature list structure. There are three signatures. Currently the list has the order sig1,sig3,sig2. The lookup function has a move to front routine so the order of the signatures may evolve during execution.

signature look-up and dispatch results in execution flowing from JITed code back to interpretation. Section 5.3.1 sketches the fields of our signature data-structure and Section 5.3.2 provides a flow chart of our profiling architecture.

5.3.1 Profile Signatures

In Figure 5.1 a sketch of an example list of signatures is provided. Every instance of a Lua function in our modified mixed-mode virtual machine contains a signature list structure. It is the state of this signature that triggers the compilation, linking and dispatch of JIT functions. Every profiled function call and backward branch of the function has an entry in the list of signatures. To reduce signature checks, a move to front scheme is used to move the latest signature to the front of the list. Upon profiling a loop where type state does not change from iteration to iteration, the profiler performs only a single signature comparison per profile increment.

5.3.2 VM Execution Flow

Figure 5.2 is a flowchart of the profiling mechanism implemented inside the Lua virtual machine. Execution begins in the profiling interpreter, for every function call or back-

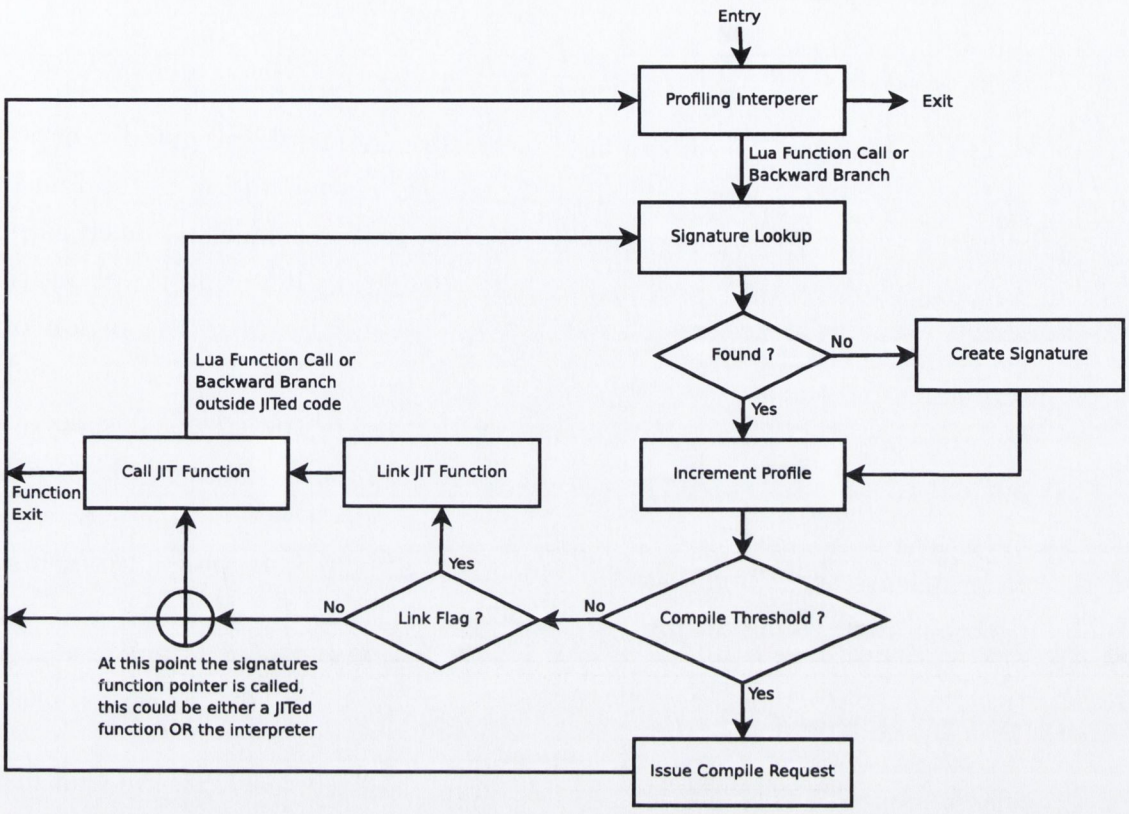


Figure 5.2: A flowchart showing the flow of execution between JIT and profile-interpreting execution. See Section 5.3.2 for further discussion.

ward branch taken within the interpreter, a *side exit* increments a profile count within the signature for that region of code. When this count reaches a defined threshold, the profiled region is added to a request queue which the compile thread receives jobs from. The compile thread then creates an ANSI C representation of the Lua function and calls the native C compiler to compile the generated code. In the meantime, the execution of the Lua program continues in the profiling interpreter. When the compile thread has completed execution it sets the link flag in the region's signature. The first time the virtual machine reads the link flag as set it dynamically links the compiled function from its shared object file. The flag is then reset and execution continues in JITed code. Execution of the Lua program may leave JITed code at any time and return to profile interpretation if: (1) a branch takes control flow outside of the JITed region, (2) there is a call to a function which has not yet been compiled or (3) the program exits from the initial called JIT function.

5.4 Data Flow Analysis

Once the profiling interpreter has identified a frequently executed function or region of code, the code generator creates an ANSI C representation of the Lua function. In order to generate efficient code for the compiler to optimize as much as possible, the code generator must be provided with some additional information about the function. On entry to code generation the JIT compiler is aware of the region of code it is compiling and the set of local types on entry to that region. Data flow analysis is used to propagate this type information over the rest of the code. This section discusses our data flow analysis implementation. Section 5.4.1 describes the construction and structure of the flow graph. Section 5.4.2 discusses the type inference algorithm used to propagate types over the whole function. Finally, Section 5.4.3 presents our library function analysis algorithm.

5.4.1 Flow Graph

The region's flow graph is a directed graph constructed from the Lua instructions of the function. Each node in the graph represents a virtual instruction. Nodes contain a list of predecessor nodes as well as either one or two successor nodes. Successor nodes are either the next instruction in the program or an instruction that the program may jump to, depending on the result of the instruction. A node can have many predecessor nodes. Nodes contain an array of types, one entry for each local variable in the function. This array represents the known types at that point of execution. Nodes also contain an array of library function states. These states represent the states of any possible library functions stored in local variables. This information is used to inline library calls when they are called during execution.

5.4.2 Type Inference

Type inference refers to the stage in compilation where a type for each variable at every execution point is deduced from a set of known inputs. The output from our type inference algorithm is a flow graph where each node has a known set of types to generate code from. Types can be of any Lua primitive type or may be UNKNOWN if control flow or other structures have impeded the data flow analysis from inferring types.

The entry node of the graph contains the type information retrieved from the run-time state of the virtual machine. All other nodes contain the unique type ANY. An

iterative algorithm infers types from predecessor nodes to successor nodes. A join function creates a special union of type sets from multiple predecessor nodes. Any conflicts in the sets results in a type becoming UNKNOWN. Each Lua instruction (hence each graph node) has its own fixed-point equation for propagating types. For example, the addition of two NUMBERS infers a result type NUMBER. Similarly, the loading of a constant STRING, infers a result type STRING. In the Lua language most variables are stored in table structures. Values fetched from these tables are always unknown to data-flow analyzer. This structure of the language results in many UNKNOWN types. An evaluation of our type inference is provided in Section 5.7.3. The type knowledge created in the type inference stage is used in code optimization to both remove type checks and store known variable types out of the Lua stack and in C variables.

5.4.3 Library Function Analysis

Lua has a standard set of libraries providing the programmer with common functions used in IO, MATH and STRING operations. During program execution these library functions are called after being fetched from the global table. Fetches use constant string values equal to the library and function name. These constant values are known to the data flow analyzer and hence the output of these global fetches can be propagated through the flow graph with the same fixed point algorithm as type inference. The analysis is used to inline library calls in the generated code and hence remove stack operations required to pass parameters and return results.

While library function analysis can inform the code generator of where a function has been fetched from, this can not be taken as a guarantee that a given function will be executed. As Lua is a dynamic language programmers are free to overwrite library functions with their own implementations. In order to safely inline library calls the Lua virtual machine must either be: (1) *sandboxed*, to guarantee the programmer does not overload library functions or (2) have run-time guards to ensure correctness of calls to inlined functions.

5.5 Optimizations

Having completed data flow analysis, the control flow graph is passed to the code generator. The code generation stage of our compiler performs a linear pass over the region's code. For each virtual instruction in the region, a specialized implementation for that instruction is output. Specialization is performed using operand values decoded from

the opcode as well as the type information inferred in the previous stage of compilation. This section discusses a number of ways in which code can be optimized with this information. Section 5.5.1 describes how the interpreter overhead of decoding virtual instructions, handling dispatch and control flow is removed. Section 5.5.2 presents scenarios where type check removals occur. Section 5.5.3 presents our techniques for storing Lua Variables in C variables in order to reduce access times. Function call optimizations are discussed in Sections 5.5.4 and 5.5.5.

5.5.1 Control Flow and Interpreter Overhead

Lua instructions [Man03] are encoded in a fixed size of 32 bits and are represented by a 32 bit unsigned integer by default. The least significant six bits of an instruction are the opcode value of the instruction, further bits represent either indices of registers or a constant value. During interpretation, these operands are accessed by performing bit operations on the executing instruction. Virtual registers and other values can only be referenced after the decode completes. Our code generator decodes these operand values and generates a specialized implementation of the virtual instruction; the specialized code now accesses the virtual registers and operand values directly.

In interpretation, dispatching of instructions and control flow is handled by the interpreter's switch statement. A program counter is updated during each instruction and on the following interpreter loop increment the instruction is dispatched. Our code generator handles control flow using direct branching to the location of the desired branch. As we are generating C code we use the `goto` feature of this language to perform the dispatch and in this way the control flow of the program is efficiently handled with minimal overhead.

5.5.2 Type Check Removal

There are three main types of type check in Lua execution. They are (1) arithmetic and string operand checks, (2) conditional branch checks and (3) table key type checks. This section describes the nature of these checks and how our code generator removes them.

I Arithmetic/String operations typically take the form A = B OP C. In interpretation mode for an arithmetic operation, operands B and C are both checked to be type NUMBER before the operation takes place. If our type inference has inferred this information then the code generator can

simply remove the type check of either or both the operands and generate a simple standalone arithmetic operation.

II Conditional branches typically compare two Lua variables and branch based on the outcome of this comparison. All of these comparisons first check if the types of both operands are equal. If they are, then the specialized check for that value type is dispatched. For example, the comparison for type NUMBER is a standard C comparison operator but the comparisons for type STRING and TABLE have a specialized function to perform their comparison. If one operand type is known an optimized condition checking the type of the second operand and dispatching the conditional check can be generated. If both types are known then we can simply generate the specialized conditional check to compare the two values. Some conditional branches can be completely optimized away if type inference guarantees that the type state of both variables will always result in the same result. In these cases a `goto` can be generated to jump to the branch's target.

III Table accesses require a table and a key. Before an access occurs, the interpreter must first check if the table variable is in fact a table. If we can infer this information we can then remove this check. Next, a specialized lookup function is dispatched based on the type of the key. In Lua tables there is both an array and a hash part to the table. Integer accesses are fetched from the array part and STRING are fetched from the hash part. This lookup is implemented with two function calls and a switch statement. With knowledge of the key type the code generator can directly call the specialized lookup on the table and remove a function call and a switch.

5.5.3 Lua Variables to Primitive C Variables

Lua opcodes operate on virtual registers. These virtual registers are located on the virtual call stack. Each virtual register contains a *type* field and a *value* field. The value field is a union structure implemented in ANSI C. Depending on the value type, a different field in the union is accessed. During interpretation an access to a virtual register requires a load of the virtual register from the stack, a field access targeting the union, and finally a field access in the union to access the variable's value. This level of abstraction causes many load instructions for access to a single value.

With knowledge of a variable's type, our code generator can simply generate code which accesses a C variable, for example a double, int or pointer. This removes many

of the loads required throughout program execution.

However, as Lua is a dynamic language with full memory management support, the virtual stack must be coherent at all times. Lua's garbage collector performs passes through the stack and requires access to all values on the stack. In order to keep correctness in memory management, our code generator creates a write-through scheme which writes-back to the virtual stack the value of cached C variables. Execution continues accessing values from C primitives but on each store the value is written-through to the stack.

5.5.4 Library Inlining

A significant proportion of Lua execution time is spent in the standard libraries of the language, especially in applications with large amounts of string and file accesses. Library calls are costly, as parameters and results are passed and returned via the virtual stack. For example the cost of performing a square root operation requires the allocation of the stack, moving the parameter to the correct position on stack and returning the result to its correct position. In JITed code where Lua variables have been cast to C variables, operations like math.sqrt can be optimized to perform the operation on the C variable and return the result to a C variable. When the compile thread generates specialized code for these operations it will first check the identity of the last called function as profiled by the interpreter. Inlining of the last called library function is then performed with a guard checking that the function value of the executing function has not changed since compilation time.

5.5.5 Lua Inlining

In a naive implementation, a call from a JITed Lua function to another JITed Lua function requires a lookup across the callee's signature (see Section 5.3.2). This lookup will find the executing function specialized for the callee's parameters. Recall that type inference has inferred many of the variable types inside the calling function, possibly including those of the parameters. To remove the cost of lookup, our current generation of a Lua function call generates direct calls to compiled Lua functions. A guarded direct function call is generated for every set of parameters in the callee's signature list. Parameters whose types have not been inferred require a type check before the call of the function. In cases where all parameter types are inferred, then a direct call is generated without any type guard. Where no function can be identified as a correct callee, a default signature lookup is generated.

5.6 Experimental Setup

Experimental data was gathered on our development machine, an Intel Xeon Dual Core 2.13Ghz processor with 4MB caches and 12GB of memory. The operating system is Ubuntu 9 with x86_64 GNU/Linux kernel 2.6.29.2. Lua source code is version 5.1.4. We have used the GNU gcc compiler version 4.3.3. The dynamic linker in the virtual machine links the compiled native code at run-time using the GNU linker and loader, *ld*, version 2.19.1. All virtual machines were compiled with the optimization options -O3 -fomit-frame-pointer.

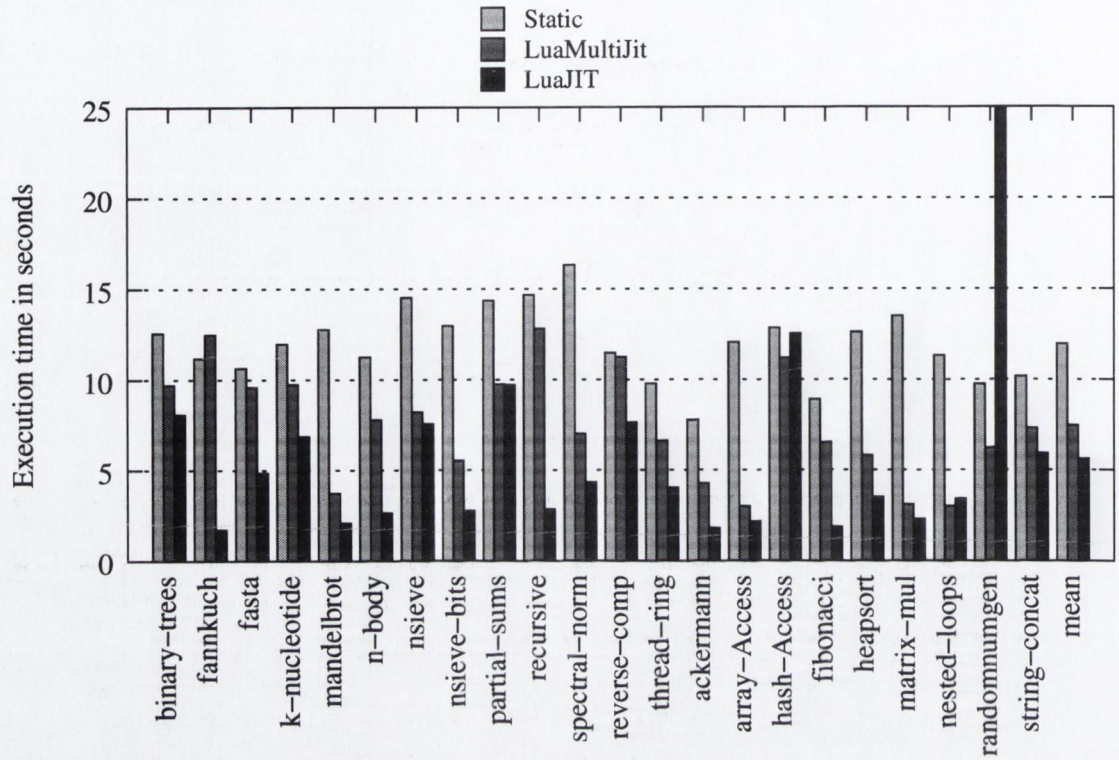
Our micro-benchmark set is taken from the Computer Language Benchmarks Game [11] and the Great Win32 Computer Language Shootout [12]. Due to a lack of formal benchmarking in scripting languages both are popular benchmark suites for use in the publication of scripting language implementations [GES+09; BdG09; CSR+09; Pal08]. Running times for these benchmark suites are presented to demonstrate the speedups achieved over varying programming structures (e.g. loops, recursion, table kernels, number kernels, string kernels). We provide further analysis of our type inference performance in our application benchmark suite we gathered from multiple sources: (1) applications hosted on the luaforge webpage, (2) ports from the V8 java script benchmark suite and (3) some benchmarks written ourselves. We also present performance speedups for this suite.

5.7 Experimental Evaluation

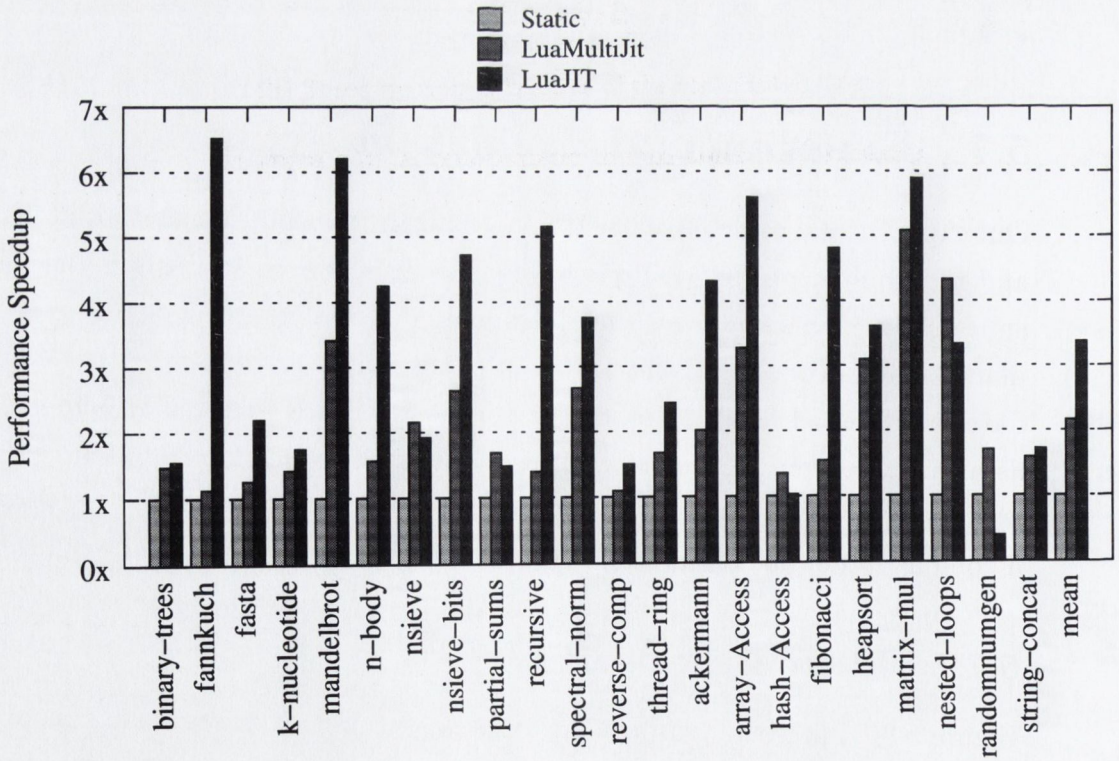
Our experimental evaluation begins by comparing the speedups for both LuaJIT [Pal08] and our multi-threaded JIT (LuaMultiJit) over the computer language benchmark suite [CLB08]. Section 5.7.2 then introduces our application suite and presents performance figures collected from execution of the suite on the Lua virtual machine. This data is presented as an introduction to the overheads involved in Lua execution and as an introduction to the suite itself. Next we present an evaluation of our profiling technique in the removal of type checks across the benchmarks. This section ends with a comparison of our performance versus that of LuaJIT across the application suite.

5.7.1 Micro Benchmarks

Figure 5.3(b) presents performance increases for both our approach (LuaMultiJit) and that of LuaJIT. The baseline for these comparisons is execution using the Lua 5.1.4 interpreter. To date, our multi-threaded approach has not reached the performance



(a) Execution time in seconds



(b) Speed up from a baseline of the static interpreter

Figure 5.3: A comparison of run-time performance.

Arithmetic /String	Register Load/Store	Branch	Call/Return	Other
ADD	MOVE	JMP	CALL	NEWTABLE
SUB	LOADK	EQ	TAILCALL	SELF
MUL	LOADBOOL	LT	RETURN	SETLIST
DIV	LOADNIL	LE		CLOSE
MOD	GETUPVAL	TEST	**Forloops**	CLOSURE
POW	SETUPVAL	TESTSET	FORPREP	VARARG
UNM			FORLOOP	
NOT	**Table Fetch**	**Table Store**	TFORLOOP	
LEN	GETGLOBAL	SETGLOBAL		
CONCAT	GETTABLE	SETTABLE		

Table 5.1: Table of Lua opcodes grouped by type.

of LuaJIT. Benchmarks such as *mandelbrot*, *heapsort*, *matrix-mul* and *nested-loops* show that our profiling and code generator can produce efficient code. Peformance of benchmarks with many calls to smalls functions, like *recursive*, is disappointing. Some of these benchmarks, like *binary-trees*, almost match the performance of LuaJIT. For the most part, our system performs well on benchmarks with computationally intensive algorithms, with large loops and many table accesses.

5.7.2 Benchmark Introduction

There are eight benchmarks in our application suite. Two of these — Deltablue and Richards — are ports of the JavaScript benchmarks in the V8 suite [Inc08]. Four were written for benchmarking during the development of our multi-threaded JIT — LZ77, LZ78 are compression algorithms, Pacman is a simulation of the pacman game using Lua *coroutines* and maze is maze generator/solver application. Yueliang is a Lua compiler written in Lua and is hosted on the luaforge website. Finally, markov is the markov algorithm presented in the Programming in Lua book [Ier06].

Figure 5.4 presents the run-time frequencies of instructions executed grouped by table. Figure 5.5 sketches the cost of these instructions execution in relation to the total execution time. Figure 5.6 shows the workload for each of the major components of the Lua Virtual machine — also in relation to total execution time.

Looking at the mean measurement for each of these experiments we can see a number of interesting results. Firstly, on average the Lua virtual machine spends approximately 25% of execution time in the interpreter, 25% in table fetches, 25% in library calls, and 5% in both function calls and memory management. As our JIT is targeting the optimization of Lua execution the percentage of time spent in the

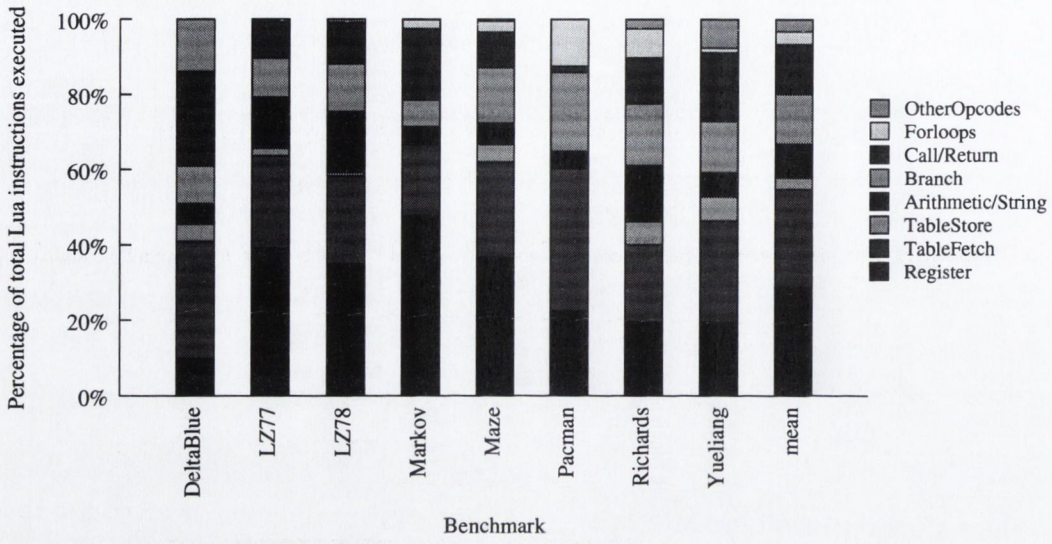


Figure 5.4: Plots the frequency of Lua opcodes use per benchmark. Opcodes are grouped as in Figure 5.1. Frequency is a run-time count of the number of executions of a particular instruction. Measurements were collected using the gcov coverage program (see Section 5.6).

interpreter limits our overall speedup. Some additional performance can be achieved from specializing table fetches to key types and in future work inlining library calls.

We can also see that while register operations have a very high frequency their overhead costs are extremely low. Function calls represent approximately 10% of all executed instructions which indicates that inlining Lua functions in future work will result in further performance increases. Lua function calls require costly stack operations for passing parameters and returning results. An efficiently inlined routine utilizing cached C variables will remove large portions of this work.

5.7.3 Type Inference Performance

Figures 5.7, 5.8 and 5.9 present an evaluation of our type inference performance achieved from our signature based profiling and dispatch mechanism.

Figure 5.7 shows the number of known operand types during execution of JITed code. From the mean bar we can see that on average 50% of operand types are successfully inferred from our profiling mechanism. Figure 5.9 shows the source of all unknown types. Here we can see that over 60% of unknown types are unknown as they are results of a table fetch. In Lua, table entries can be of any type and so during

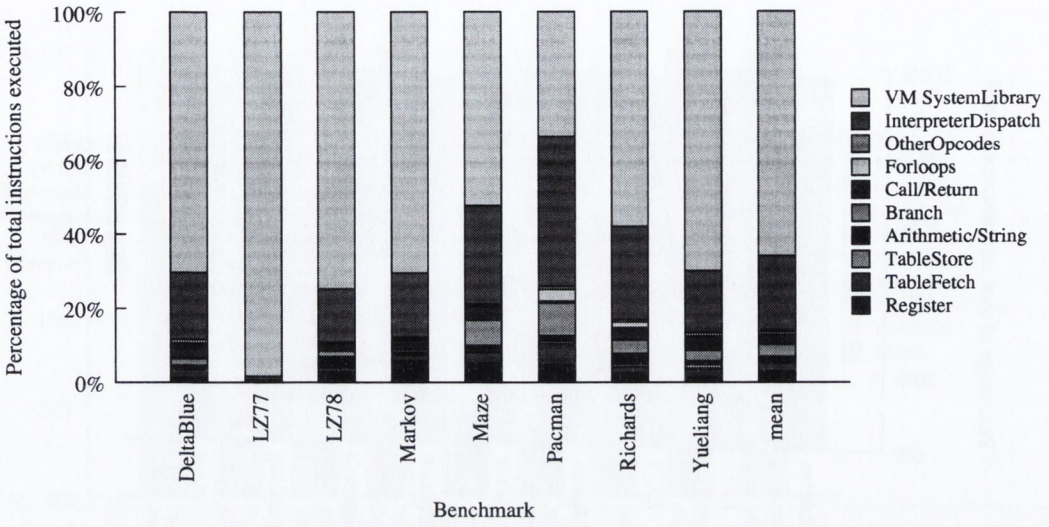


Figure 5.5: Plots the overhead of opcode use by group. Also included are the cost of Lua instruction dispatch and system calls. System calls represent all other computation, most notably table access, memory management and library calls (see Figure 5.6 for a break down of these work loads). Overhead is measured in machine instructions executed and was collected using the gprof execution profiler (see Section 5.6).

JIT compilation our data analyzer is unable to infer any types for the result of these fetches. The remaining unknowns are a result of control flow conflicts or unknown returns from function calls.

Figure 5.8 presents a breakdown of the run-time type checks and type accesses performed during program execution. On average our technique achieves more than a 40% reduction in type checks with half the benchmarks losing the majority of type checks. This figure shows the success of our profiling technique in removing run-time type checks.

5.7.4 Performance

Performance results for our application suite (Figure 5.10) are very similar to those of the computer language shootout micro-benchmarks. We see again that for the majority of applications our current implementation does not achieve the same speed increases as LuaJIT. Pacman is the exception to this trend. This is expected as pacman, a simulator of the pacman game, has many table accesses and numerical operations. Intra-procedural optimizations alone are not sufficient to achieve performance. For most of the applications, efficient function inlining is a necessity for increased perfor-

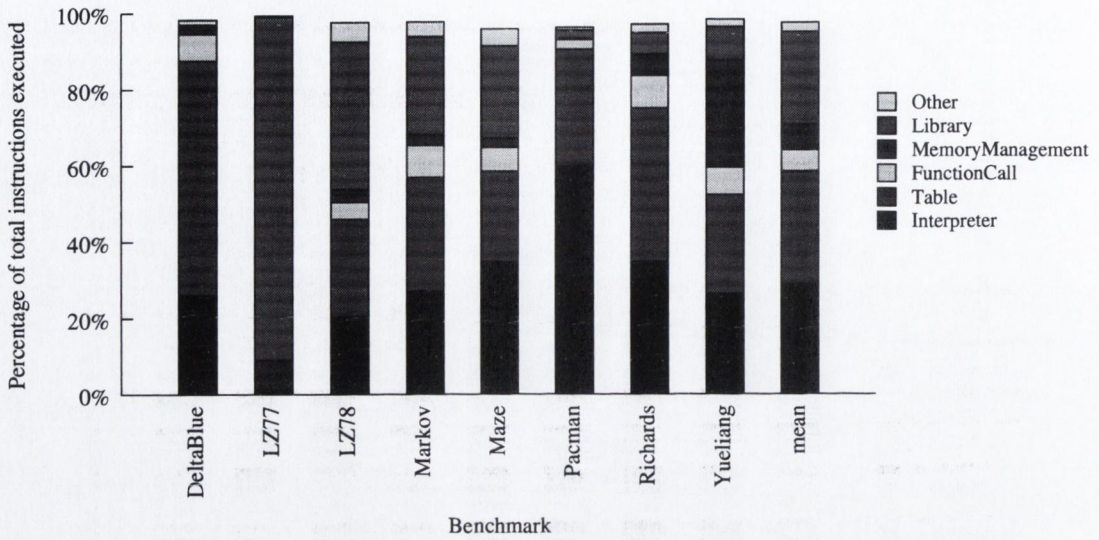


Figure 5.6: Plots the total computation overhead for the classified work units which include: (1) executor (interpreter etc.), (2) table access, (3) function calls, (4) memory management, and (5) library calls. The sixth unit, other, includes all other computation not classified in other primary work units. This includes the parsing of Lua source code, compilation to bytecode and the starting up and shutting down of the virtual machine. Overhead is measured in machine instructions executed and was collected using the gprof execution profiler (see Section 5.6).

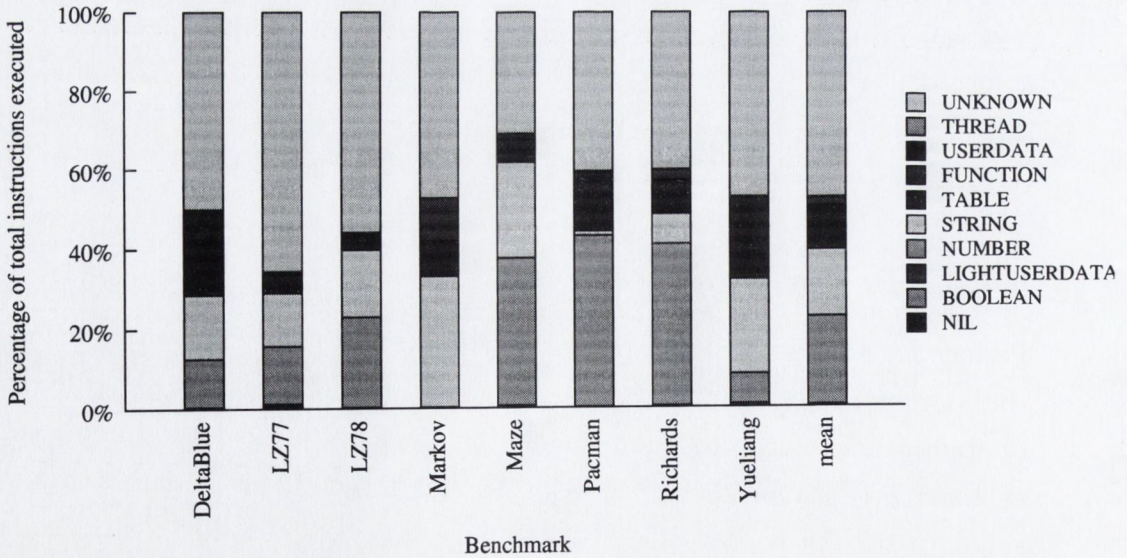


Figure 5.7: Presents the run-time operand type knowledge for Lua operands as a result of our type inference

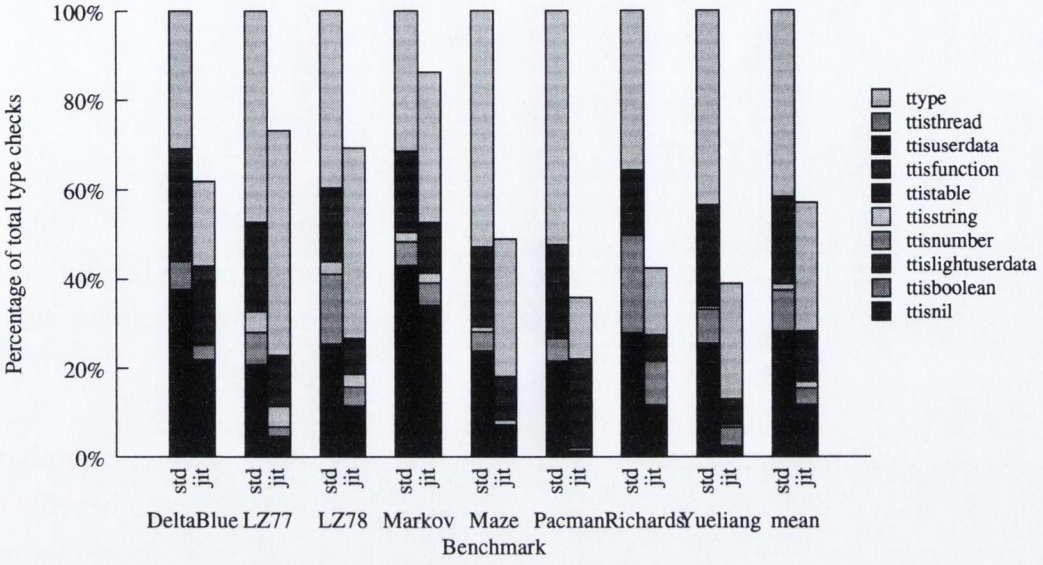


Figure 5.8: Presents the total number of type check removals achieved as a result of our code specialization

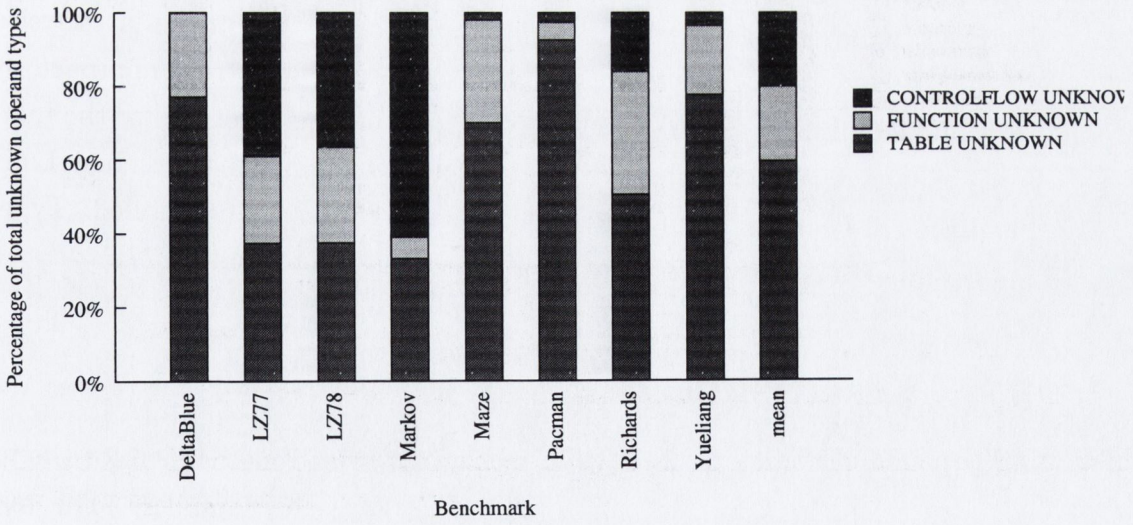


Figure 5.9: Details the source of unknown operand types from our type inference algorithm

mance. Inlining would remove many of the expensive signature look-ups executed when calling JITed functions from JITed functions.

5.8 Related Work

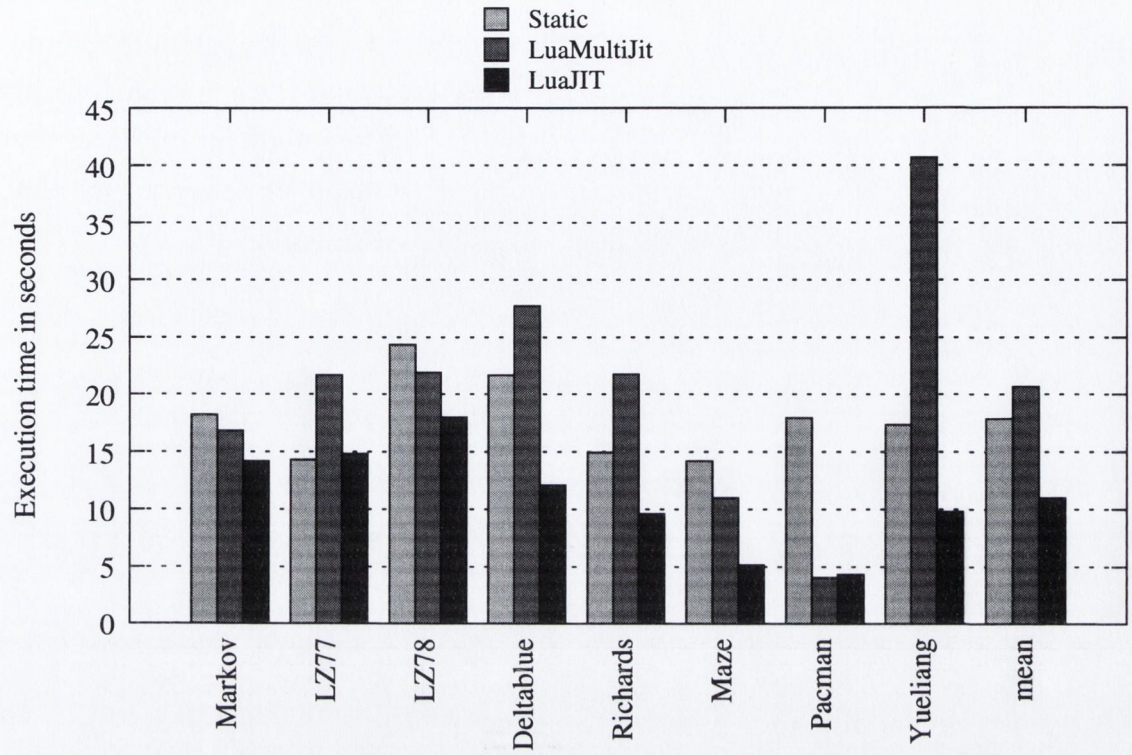
The most relevant work to this chapter is that to which we are extending. In Section 3.2.1, we presented Whaley's [Wha01] work on partial method compilation techniques in JIT compilation systems for the statically typed Java programming language. Further in this Section, we also discussed Gal's [GPF06] and Zaleski's [ZBS07] use of trace based compilation in the HotpathVM and Yeti VM respectively.

Bruening and Duesterwald [BD00] present a study of several compilation unit shapes in which they use interpreter based simulations to survey the benefits of compilation unit shape and size for their benchmarks. They find that whole method shapes combined with partial-methods or traces provides superior performance to the compile all approach. They also determine that partial methods provide greater opportunity for optimization, but the linear nature of traces mean simpler compilation.

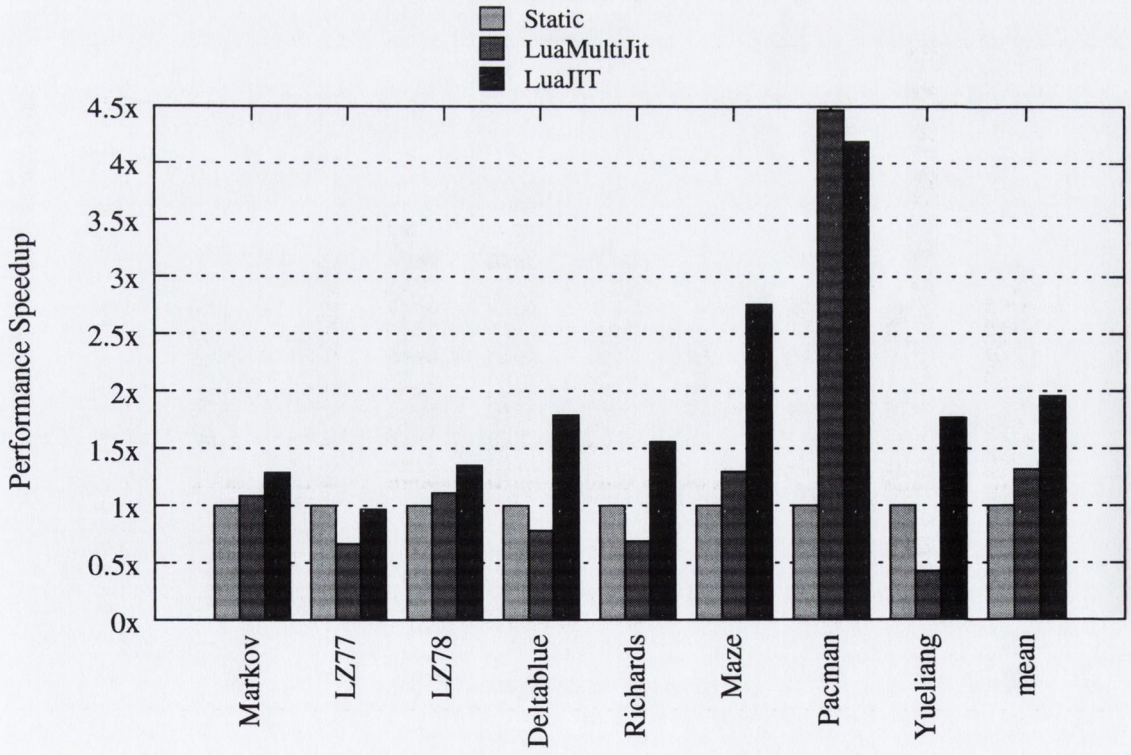
In our scripting language survey of Chapter 3, we described Rigo's [Rig04] run-time specialization technique for the Python programming language. It performs run-time specialization *by need* using a mixed execution/specialization phase of execution to specialize code fragments. Firefox's JavaScript engine, *spider monkey* [Eic08; CSR+09], has been deployed with a trace based compiler [GES+09] in the latest release of the popular web browser. They have adopted the incremental trace approach by Gal and Franz [GF08]. LuaJit [Pal08] is a successful just-in-time compiler for the Lua programming language targeting the x86 architecture. It compiles individual Lua functions when they are called at run-time, functions not run are not compiled. Type inference and optimizations are performed via abstract interpretation [CC77].

5.9 Discussion

This chapter has presented a successful multi-threaded mechanism for the just-in-time compilation of dynamically typed code by a native C compiler. The result of our scheme is a portable, safe, type specializing JIT architecture. Our novel use of a native ahead-of-time compiler allows us to take advantage of the optimizations found in these compilers. It also ensures the portability and maintainability of our system. Future work will include the inlining of library and Lua function calls (as discussed in Section 5.5). Our planned inlining optimizations will again take full advantage



(a) Execution time in seconds



(b) Speed up from a baseline of the static interpreter

Figure 5.10: A comparison of run-time performance.

of our native compilation approach as we will generate inlined code for the compiler to aggressively optimize away expensive function calls. Our approach to just-in-time compilation is highly novel and only made feasible by the recent advances in multi-core architectures. The contributions of this chapter included strategies for the indexing, management and type safe dispatch of specialized code regions.

Chapter 6

A Type-Specialized Lua Interpreter

Dynamic scripting languages offer programmers increased flexibility by allowing properties of programs to be defined at run-time. Typically, program execution begins with an interpreter where type checks are implemented using conditional statements. Recent JIT compilers have begun removing run-time checks by specializing native code to program properties discovered at JIT time.

This chapter presents a novel intermediate representation for scripting languages that explicitly encodes types of variables. The dynamic representation is a flow graph, where each node is a specialized virtual instruction and each edge directs program flow based on control and type changes in the program. The interpreter thus performs specialized execution of whole programs. We present techniques for the efficient interpretation of our representation showing speedups of greater than 2x over *static interpretation*, with an average speedup of approximately 1.3x.

6.1 Introduction

Scripting language virtual machines typically compile high-level source code into an array of low-level opcodes (bytecodes). They then use a standard interpretation loop to execute the array of opcodes. Run-time type checks are implemented with either conditional statements or switch statements. Performing these type checks forms a significant portion of program execution. These costs seem unnecessary especially when we realize the same variable types are inevitably rechecked as loop iterations repeat the execution of the same block of code.

Recent research has shown the advantages of type specialization in JIT compilation [Rig04; GES+09]. However, little research exists on the specialization of interpreters for scripting languages. We propose a scripting language representation

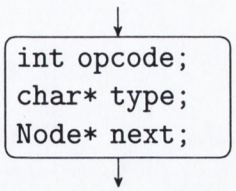
and interpretation technique that performs specialized execution of dynamic code. We present a dynamic intermediate representation (DIR) which explicitly encodes the types of variables at each point of execution. The DIR is a flow graph, where each node is a specialized virtual instruction and each edge directs program flow based on control and type changes in the program. The proposed DIR therefore has a specialized path in the graph for every sequence of control and type changes found during execution.

In this chapter we present the initial development of our prototype implementation in the Lua virtual machine [IdFC05; IdFC07]. We present the design of our dynamic representation as well as the techniques used for its efficient interpretation. We illustrate our representation with a motivating example and present our plans for future interpreter optimizations. We compare the interpretation of our representation to the standard Lua implementation and show early speedups reaching a max of over 2.5X and an average of ~1.45X. We present reductions in the total number of instructions executed including reductions in conditional branches (used in interpreter type checking) as well as decreases in data and instruction cache accesses.

6.2 Dynamic Intermediate Representation

In this section we present the DIR we propose for the specialized interpretation of Lua programs. Our representation is a flow graph where nodes represent specialized instructions and edges represent either control-flow or type-flow. Type-flow is the dispatching of a virtual instruction based on the type of the result from the current operation. Therefore, type-flow provides a separate path in the DIR for each variable type defined by the node's operation. These paths provide the opportunity for type specialization. As every type change results in a new path, all variable types are known at **every** point in execution. The interpreter therefore achieves complete specialization of whole programs.

Paths in the graph are built on demand, meaning only those control flow and type flow paths which occur during execution are ever created. Nodes on paths are built one-by-one. As complete knowledge of local types is available, creation of a new node and selection of its specialized instruction is directed by the types of the instruction's operands. The program's structure and behavior is defined by the SIR generated by the script compiler (the Lua compiler in our case). Hence construction of the dynamic flow graph is guided by the SIR. The rest of this section describes the structure of nodes forming our DIR followed by a discussion of some of the specializations performed on the Lua instruction set.

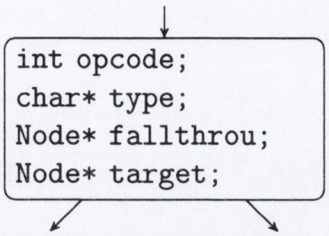


(a) Standard Node

```
1 node = node->next;
2 dispatch(node->opcode);
```

(b) Standard Dispatch

Figure 6.1: Node structures and interpreter dispatch techniques for standard nodes



(a) Conditional Node

```
1 if(<cond>)
2  node = node->target;
3 else
4  node = node->fallthrou;
5 dispatch(node->opcode);
```

(b) Conditional Dispatch

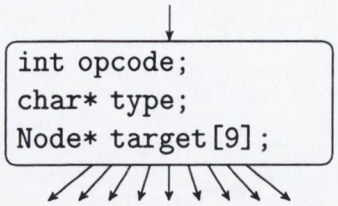
Figure 6.2: Node structures and interpreter dispatch techniques for conditional nodes

6.2.1 Standard Node

The most basic node in our representation contains three fields: (1) the opcode of the specialized instruction, (2) a pointer to the array of live types and (3) a pointer to the next node in the path. Instructions which use this basic node structure always result in the same control and type flow. Examples of such instructions include loading of constant values, arithmetic instructions and direct branches. Figure 6.1(a) sketches the structure of this node and Listing 6.1(b) shows a pseudo-implementation of its dispatch.

6.2.2 Conditional Node

Conditional nodes have all the fields of a standard node and an additional pointer to a target node. They are used to implement instructions that can result in two paths of control flow but never result in any type changes. Examples of such instructions include conditional instructions such as *equal to*, *greater than* and *less than*, as well as loop instructions which control the flow of loops. Figure 6.2(a) sketches the structure of this node and Listing 6.2(b) shows a pseudo-implementation of its dispatch. Here we can see a condition statement defining the direction of execution.



(a) Type-Directed node

```
1 dest_reg = ...
2 node=node->target[dest_reg->type];
3 dispatch(node->opcode);
```

(b) Type-Directed Dispatch

Figure 6.3: Node structures and interpreter dispatch techniques for type-directed nodes

6.2.3 Type-Directed Node

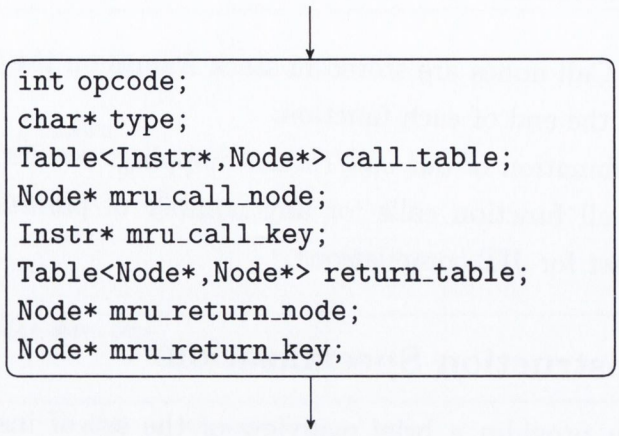
Type-directed nodes dispatch the next node based on the type of the operation's result variable. This node has three fields, the first two fields are the same as the standard node fields. The third field is an array of target nodes. Each entry in the array is a pointer to the first node of a new path. During execution, the type of the result of the operation is used to index the target array. The node selected is dispatched and execution continues down that path.

These nodes are used at any point where there is a single change of variable type and no change in control flow (i.e. all paths exiting the node execute the same instructions, the difference between paths is the types they are specialized to). The Lua language has nine types and so the dispatch array for our implementation has nine entries. Figure 6.3(a) sketches the structure of this node and Listing 6.3(b) shows a pseudo-implementation of its dispatch. Here we can see that the instruction defines some variable in *dest_reg*. This register has two fields, a value and a type. The type is used to index the dispatch array.

6.2.4 Call Node

Recall that each node in the DIR is an instruction specialized to a set of local variable types. Therefore, call nodes represent a call site and a known set of parameter types. A key/value mapping structure maps the call node to a function and the set of parameter types. The key for this mapping is the address of the called function. The value returned is the entry node of the function in the dynamic representation. The effect of this mapping is a flow of types across function boundaries.

In dynamic languages, the function called at a given point may change from execution to execution as functions are treated as first-class values. In practice, our experiments show that the majority of call points have a single called function, with a



```
int opcode;
char* type;
Table<Instr*,Node*> call_table;
Node* mru_call_node;
Instr* mru_call_key;
Table<Node*,Node*> return_table;
Node* mru_return_node;
Node* mru_return_key;
```

(a) Call Node

```
1 pc = lua_function_init();
2 if(node->mru_call_key == pc)
3  node = node->mru_call_value;
4 else
5  node = lookup(node,pc);
6 dispatch(node->opcode);
```

(b) Call Dispatch

Figure 6.4: Node structures and interpreter dispatch techniques for call nodes

few having more than one. As a result of this profile information, call node mappings are implemented efficiently using two arrays. One array to store keys of functions and the second to store the value of nodes. A linear search of the key array will find the index of the correct node in the value array. Once found, this node can be dispatched in the usual way. Our implementation further improves the efficiency of this search by storing the most recently used key and value in 'cached' fields of the node. In practice these cached fields are usually a correct match, so linear searches only occur in a small number of cases.

6.2.5 Return Node

Having completed execution of a function, control flow must return to the calling function. As is the case for call nodes, return nodes represent a return instruction and a set of return types. Hence, an equivalent table mechanism using *(key,value)* pairs to find the next node is also used in return nodes. The return table is located in the calling node, as functions may regularly be called from different call sites but individual call sites rarely call multiple functions. Storing the return table in the call site therefore reduces the number of *(key,value)* pairs in a table and improves the search time for the

correct key. Call nodes are stored in stack frames on the Lua stack and accessed from the stack at the end of each function.

The combination of our call/return mappings results in the inter-procedural type profiling of all function calls for any number of parameter and return variables (a powerful asset for JIT compilation).

6.2.6 Instruction Specialization

This section provides a brief overview of the set of instructions in the Lua virtual machine and the range of specialization our interpreter performs.

1. Register Loads: Several different instructions in the Lua instruction set implement different types of loads. A specialization applicable to all of these instructions is the assignment of type. A register in Lua stores a value and a type. Commonly a load instruction loads a variable of the same type from a source register to a destination register. In these cases we have removed the redundant assignment of type to the destination register.

2. Arithmetic Operations: The usual set of arithmetic instructions are available in the Lua instruction set. Static implementations of these operations first check that both operands are of type NUMBER before proceeding with the operation. Having full type knowledge of variables, our specialized implementations of arithmetic instructions do not require type checking and simply perform the desired operation. The same is true for string operations like concatenation.

3. Table Access: Tables are the sole data structuring mechanism in the Lua language. They are used to implement a wide range of data structures. They are associative structures and can be indexed by any value and can store values of any type. They contain two separate parts (1) a hash part, for storing values indexed by hashed values and (2) an array part for storing values indexed by integer keys. Our implementation provides specialized table access based on the type of the key. The wide use of tables in Lua makes this specialization an important optimization for overall performance.

4. Conditional Branches: The Lua instruction set has several different conditional instructions. These instructions compare two values and direct control flow based on the result of these comparisons. Valid comparisons can only be made between two values of the same type. Each conditional instruction in the instruction set is

```
1 local N = 100
2 local flags = {}
3 for i = 2, N do
4   if not flags[i] then
5     for k = i+i, N, i do
6       flags[k] = true
7     end
8   end
9 end
```

(a) Lua source code, a high-level representation.

```
1  loadk     r0 k0    ; reg0 = constant0
2  newtable  r1 0 0   ; reg1 = new table(0,0)
3  loadk     r2 k1    ; reg2 = constant1
4  move      r3 r0    ; reg3 = reg0
5  loadk     r4 k2    ; reg4 = constant2
6  forprep   r2 L15   ; perform forloop prep, goto[15]
7  gettable  r6 r1 r5; reg6 = reg1[reg5]
8  test      r6 L15   ; if reg6, goto[9] else goto[15]
9  add       r6 r5 r5; reg6 = reg5 + reg5
10 move      r7 r0    ; reg7 = reg0
11 move      r8 r5    ; reg8 = reg5
12 forprep   r6 r1    ; perform forloop prep, goto[14]
13 settable  r1 r9 k3; reg9[reg1] = constant3
14 forloop   r6 L13   ; if loop, goto[13] else goto[15]
15 forloop   r2 L7    ; if loop, goto[7]  else [end]
```

(b) Lua opcode, a low-level representation.

Figure 6.5: A Lua implementation of the Sieve of Eratosthenes algorithm.

therefore specialized to the type of its operands. The value of this specialization is large because type checking in these instructions is performed by an expensive ANSI C switch statement.

6.3 Motivating Example

6.3.1 SIR

Figure 6.5 shows an example of a Lua implementation of the well known Sieve of Eratosthenes algorithm. The program calculates all prime numbers in the first N natural numbers. The implementation presented in Figure 6.5(a) marks all numbers which are not prime in the *flags* table with the boolean value *true*. At the end of this program all prime numbers will remain unmarked in the *flags* table.

Figure 6.5(b) is the SIR generated by the Lua compiler. Execution of this opcode

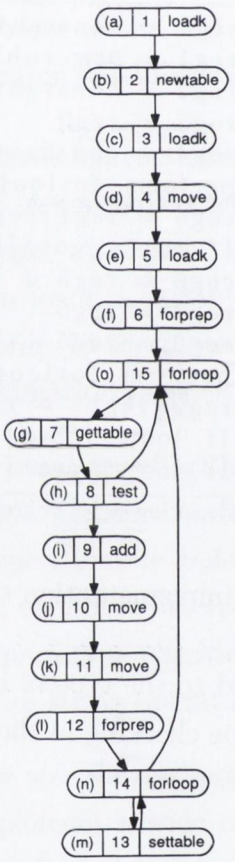


Figure 6.6: An *'imaginary'* control flow graph for the static interpretation of the Sieve of Eratosthenes algorithm.

results in the control flow pattern depicted in Figure 6.6. This graph is presented to illustrate the structure of a static flow graph to the reader and is never constructed at any point in static interpretation.

6.3.2 DIR

Figure 6.7 is an illustration of the graph that is constructed and interpreted by our dynamic interpreter for the Sieve of Eratosthenes program. The graph has twenty-six nodes in total. Each node has three labels, the first is a letter (a–z) which the authors will use for referencing nodes during discussion. The second is a number (1–15) which identifies the equivalent static instruction that the node specializes (found in Figure 6.5(b)). The last label is the opcode name. Opcodes in our DIR are specialized to the types of variables they are operating on. For example, node v is an add operation operating on two numbers, therefore the opcode selected is *add_number_number*.

Edges in the graph represent program flow. Edges with no labels represent fall-through flow, i.e. instructions which always dispatch to the same next instruction in the program. Control flow edges are labeled *loop branch* and *cond branch*. These edges represent control flow dispatches for branch instructions. The remaining edges are type flow edges. In this graph, they are labeled either *nil* or *boolean*. An example of type flow can be found in nodes g and r. These nodes are table fetch instructions and return a value from a table. They therefore dispatch based on the type of the return value. In node g there is only a single path, for type *nil*, as only one type value is ever returned at this point. In contrast, node r has two paths where values of type *nil* and *boolean* are returned. The two paths exit to the same program instruction, however the next and all subsequent instructions are specialized to the returned type.

The Sieve of Eratosthenes program is implemented with fifteen static Lua instructions. When these instructions are specialized to the local variable set found during execution, twenty-six specialized instructions are generated. In Figure 6.7 we have clustered these nodes into three sets. The first cluster represents the instructions that initialize the variables of the program. The second cluster is the first iteration of the for-loop spanning the static instructions 5–15. The final cluster is all subsequent iterations of the same loop. Two separate paths are generated for iterations of the outer loop as there are uninitialized variables in the first iteration. Only after the first iteration completes does the graph become stable.

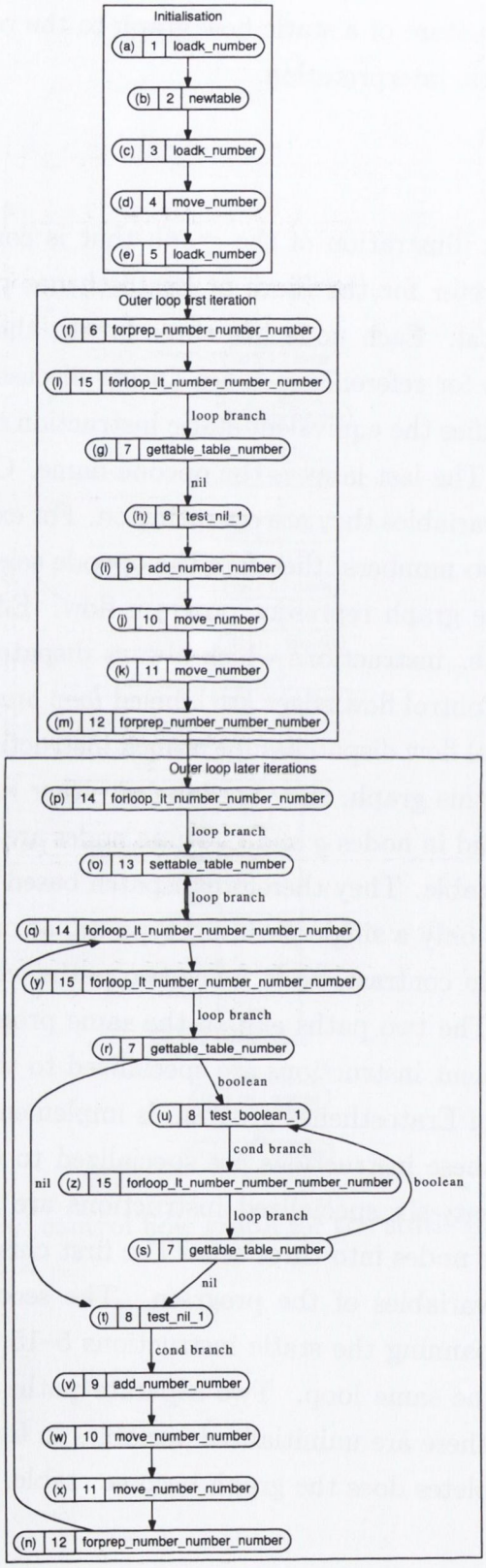


Figure 6.7: The Sieve of Eratosthenes dynamic representation built and executed by our interpreter.

6.4 Experimental Evaluation

6.4.1 Experimental Setup

The following sections provide a comparison of hardware performance and running times between static and dynamic interpretation. The test machine used to run these experiments has two Intel Xeon Dual Core 2.13Ghz processors each with 4MB caches and 12GB of memory. The operating system is Ubuntu 9 with x86_64 GNU/Linux kernel 2.6.29.2. Lua source code is version 5.1.4. We have used the GNU gcc compiler version 4.3.3. Both the static and dynamic versions of code were compiled with the optimization options '-O3 -fomit-frame-pointer'. The hardware performance counters presented were collected using the PAPI profiling tool.

Our micro-benchmark set is taken from the Computer Language Benchmarks Game [CLB08] and the Great Win32 Computer Language Shootout [GWC08]. A lack of formal benchmark sets for scripting languages make these suites a common source of benchmarking for scripting language implementations [GES+09; BdG09; CSR+09; Pal08].

Figures 6.8, 6.9, 6.11(a), 6.11(b) and 6.12 present hardware performance counters. Results are presented relative to the absolute result for the static interpreter. Hence, in all these figures, static bars are at 100% and the dynamic versions show percentage increases or decreases relative to the absolute static numbers.

6.4.2 Machine Instructions Issued

We first look at the total number of machine instructions issued to execute both static and dynamic interpreters (see Figure 6.8). All benchmarks have returned a decrease in the number of machine instructions issued. There is a best case below 50% for *array-access* and an average across all benchmarks of ~25% decrease. These figures show that the trade-off of dynamic interpretation's longer dispatch and fewer type checks gives an overall reduction in machine instructions over static interpretation's shorter dispatch and run-time type checks. However, as we are dealing with a superscalar machine we must examine further results to determine more about the behavior of both interpreters.

6.4.3 Branch Instructions

Conditional Branch Instructions: The effect of interpreter specialization can be best seen in the dramatic decrease of conditional branch machine instructions. By far the best performing benchmark in this experiment is the *mandelbrot* benchmark.

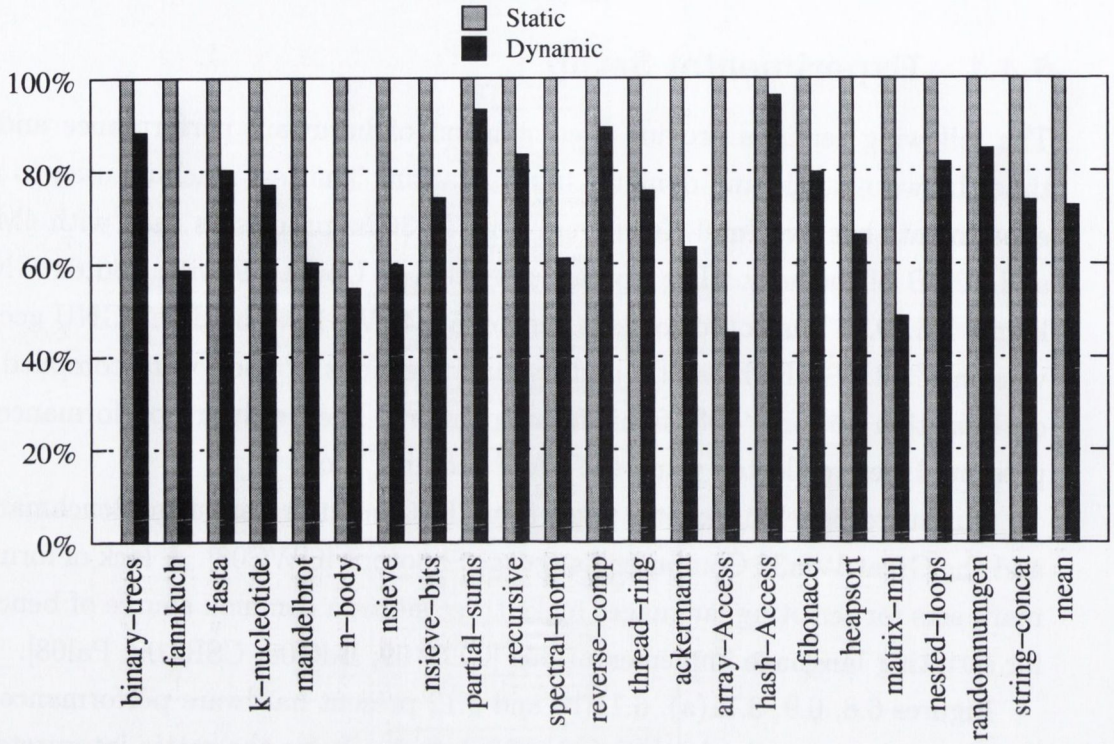


Figure 6.8: Relative measure of **instructions issued**

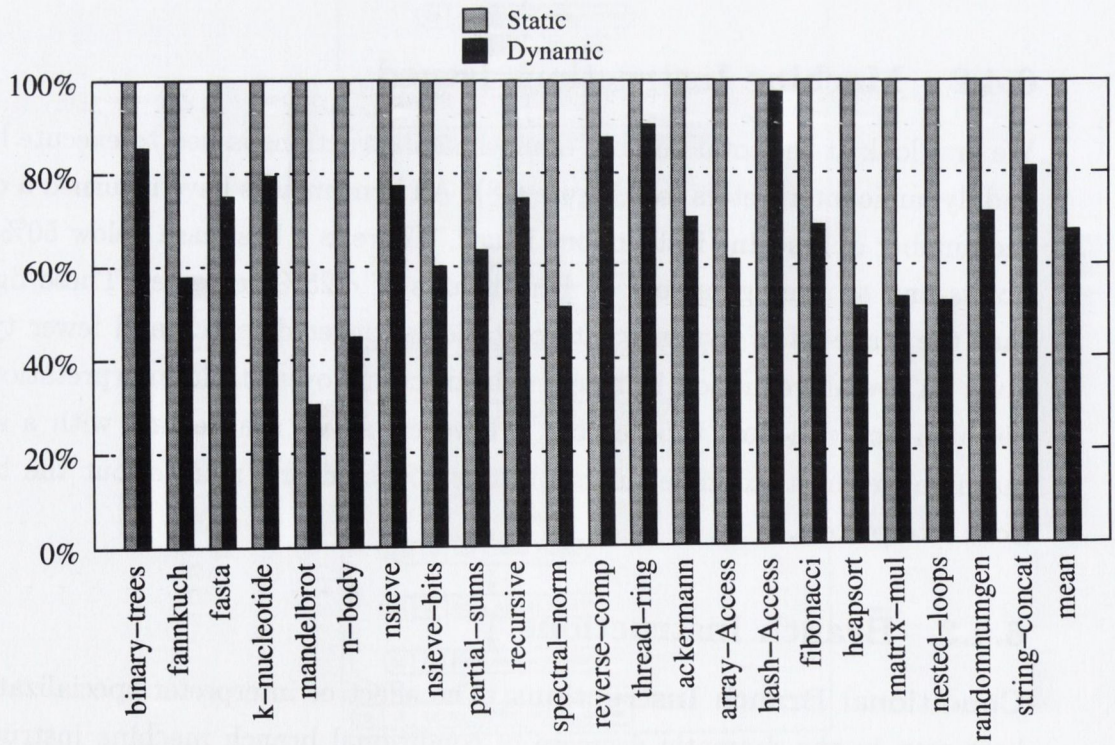
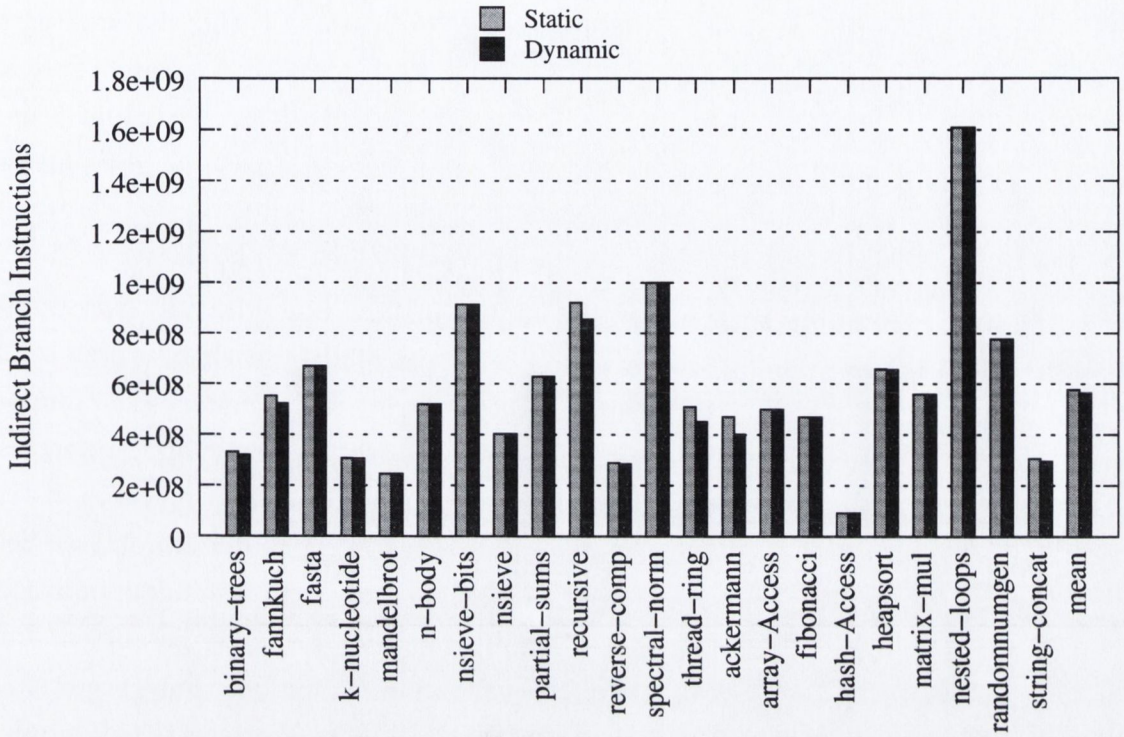
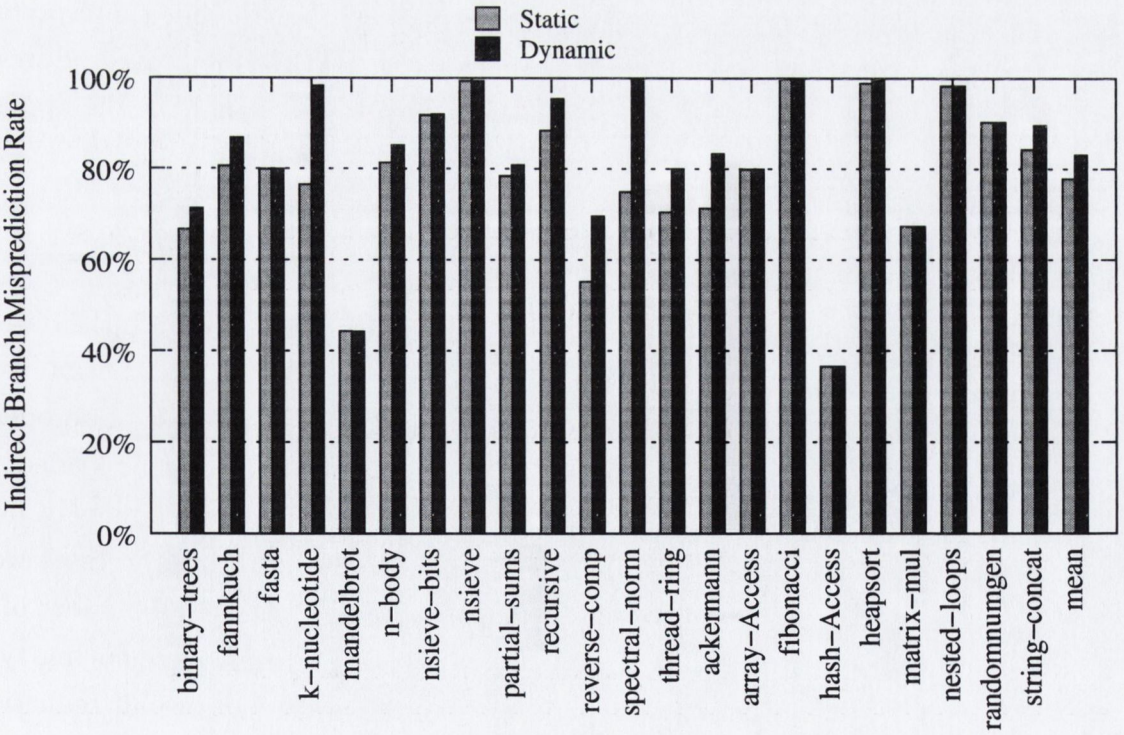


Figure 6.9: Relative measure of **conditional branch instructions** executed



(a) Indirect branch instructions (valgrind simulation)



(b) Indirect branches mis-predicted (valgrind simulation)

Figure 6.10: Indirect Branches executed and mis-predicted

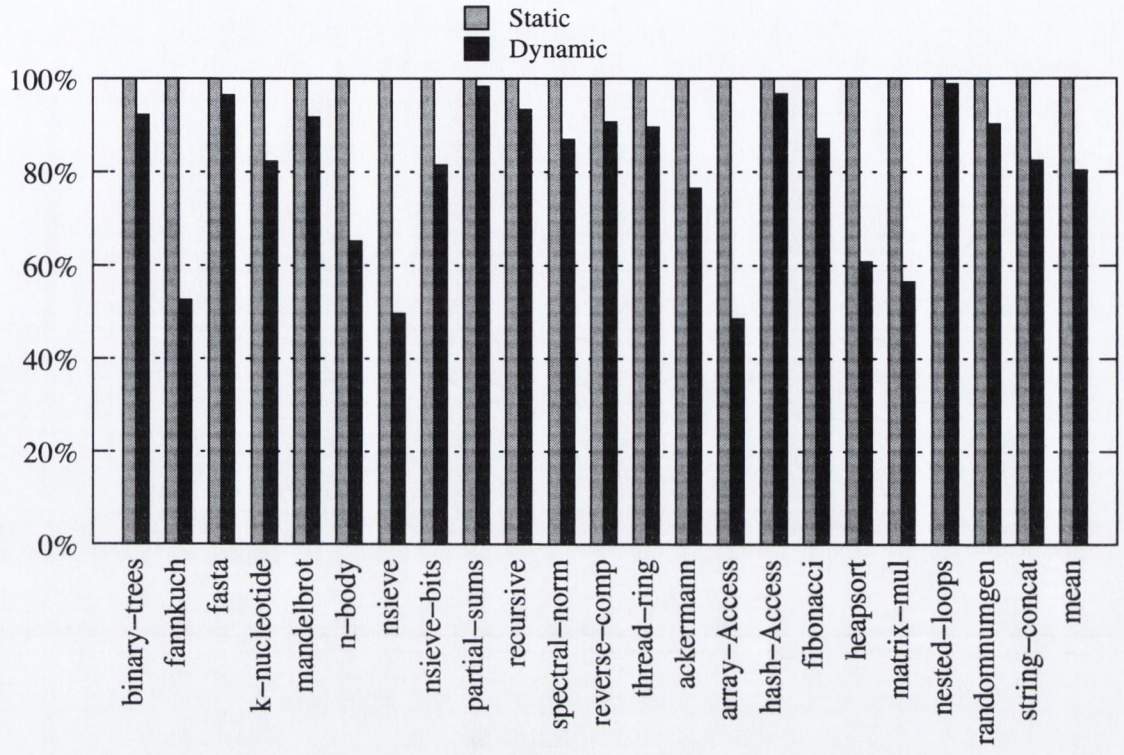
This benchmark executes many arithmetic instructions. And the greater than 75% reduction in conditional branches is achieved largely from specializing those arithmetic instructions. Other notable results are for *array-access*, *matrix-mul* and *heapsort* all of which have many number-key table accesses. All of these table accesses are specialized to the key type.

Indirect Branch Instructions[1]: Figure 6.10(a) presents valgrind simulations for the number of indirect branch instructions executed by each interpreter. As well as branch prediction rates for each. For the majority of benchmarks the number of indirect branch instructions are equal for both the static and dynamic interpreters. This is to be expected as each execute the same number of virtual machine instructions — specialization has not yet lead to any virtual instruction elimination (see Section 6.5). *Thread-ring*, a coroutine benchmark, has an increase in the number of indirect branches. This is a result of an inefficient resumption of coroutines in the current implementation. Figure 6.10(b) plots the mis-prediction rates for both interpreters. A number of benchmarks have an equal prediction rate but on average the dynamic predictor mis-predicts more branches. This interpreter behavior is explained by Casey *et al.* in [CEG07]. Adding more specialized instructions to a threaded interpreter achieves an increase in branch prediction accuracy as the BTB has improved addressing of instructions. The structure of switch based interpreters means that the opposite is true and prediction becomes worse when more instructions are added.

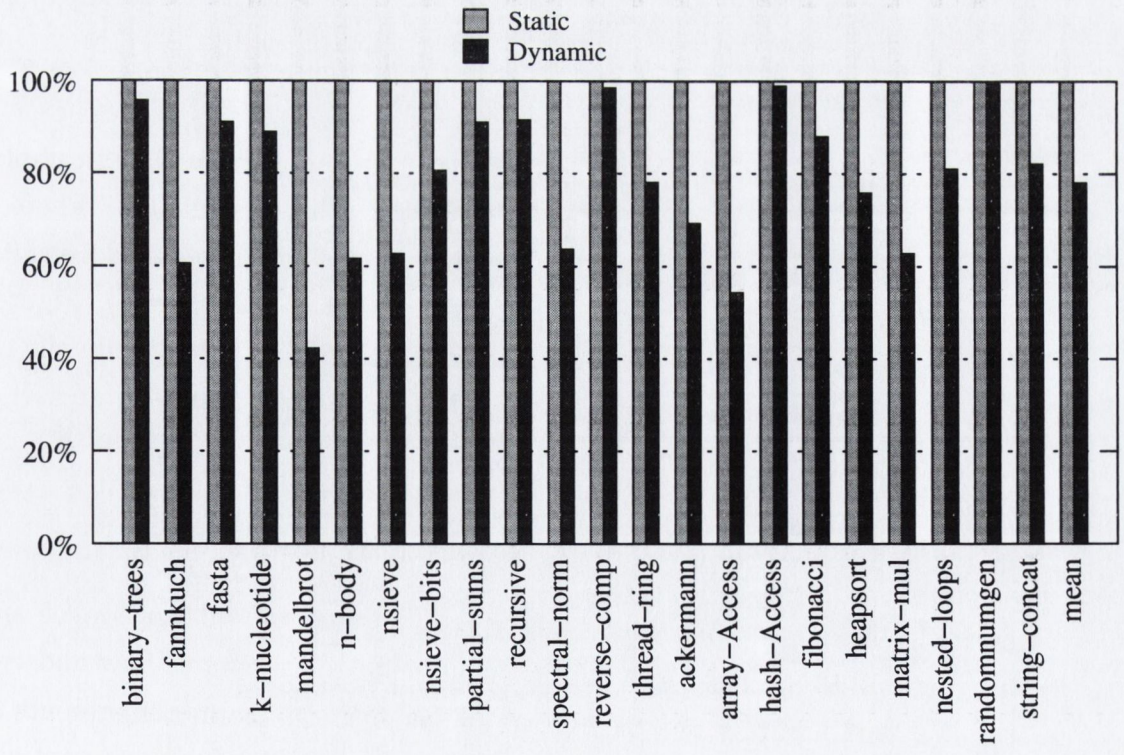
6.4.4 Cache Access

Figures 6.11(a) and 6.11(b) present Level 1 cache accesses for both the data cache and instruction cache. The majority of data cache accesses in a static interpreter are a result of virtual instruction loads and operand loads. In the case of an operation with a type check both the type and value field of the operand must be loaded. The graph structure of our DIR requires a further load per virtual instruction in our dynamic interpreter. Because of this, an overall increase in data cache accesses would be expected. Surprisingly, dynamic interpretation reduces the overall number of data cache accesses. The reduction in data cache accesses is a result of removing type accesses for run-time type checking. The instruction cache has an overall reduction in cache accesses as would be expected after a reduction in instructions issued.

[1]This section is a comparison of two portable switch based interpreters, Section 6.5 provides details of threaded implementations for the dynamic interpreter.



(a) Relative measure of **Level 1 data cache** accesses.



(b) Relative measure of **Level 1 instruction cache** accesses.

Figure 6.11: Level 1 cache access

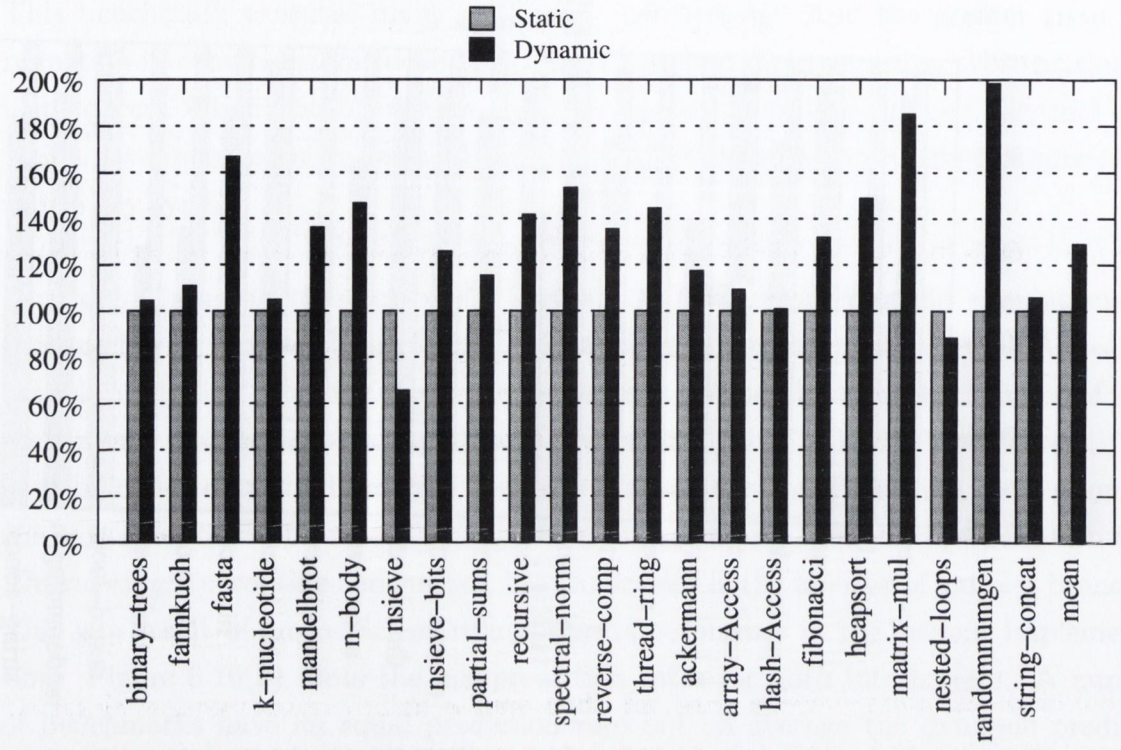


Figure 6.12: Relative measure of **cycles stalled** on any resource

6.4.5 Processor Stalls

Despite the reduction in cache accesses, Figure 6.12 shows that the DIR implementation leads to many more processor stalls across almost all benchmarks. There are several factors contributing to these stalls. They include: (1) an increase in Level 2 data cache misses (Figure 6.13), (2) an increase in Level 2 data cache writes (Figure 6.14), (3) an increase in Level 2 instruction cache misses (Figure 6.15) (despite the reduction in instructions issued the increased size of the interpreter results in greater instruction cache misses).

6.4.6 Memory

Dynamic interpretation builds a new node for each specialization of an individual instruction. Programs which contain many changes in control-flow and type-flow will result in large graphs and an increase in memory use. Table 6.1 presents the memory usage for each of our benchmarks. The first and second columns present absolute sizes of the DIR and SIR. This is followed by the relative increase in size from SIR to DIR. The next two columns are the static run-time memory use and the relative percentage

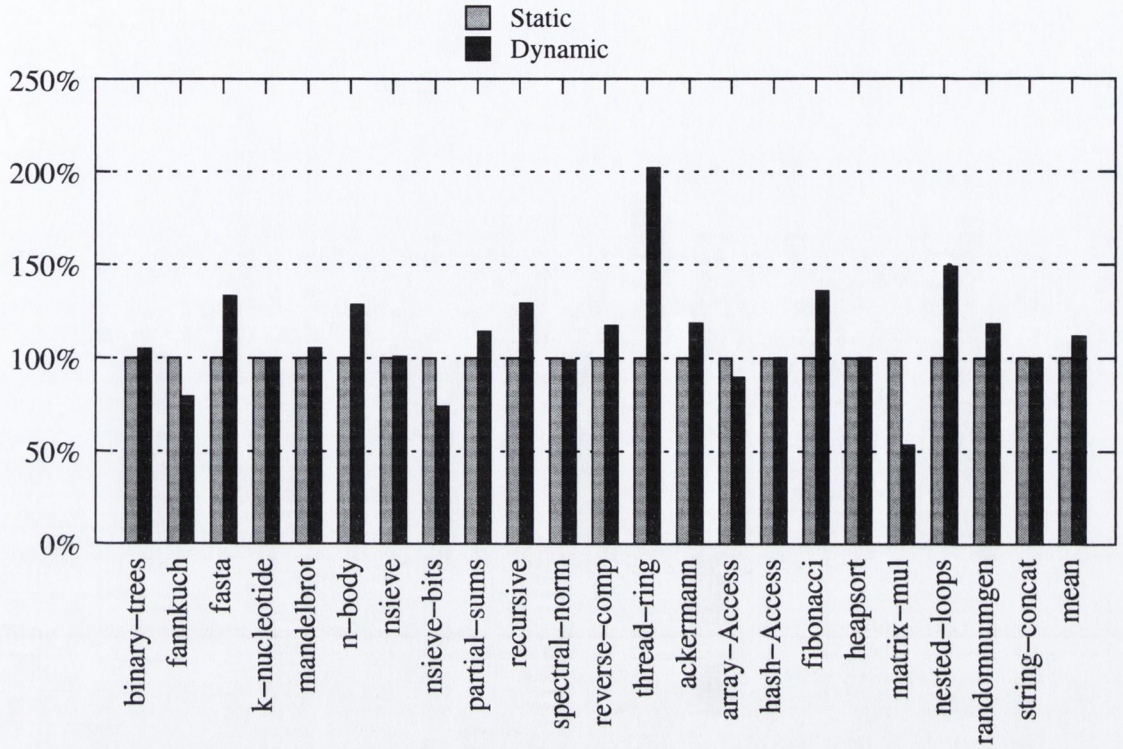


Figure 6.13: Relative measure of **Level 2 data cache** misses.

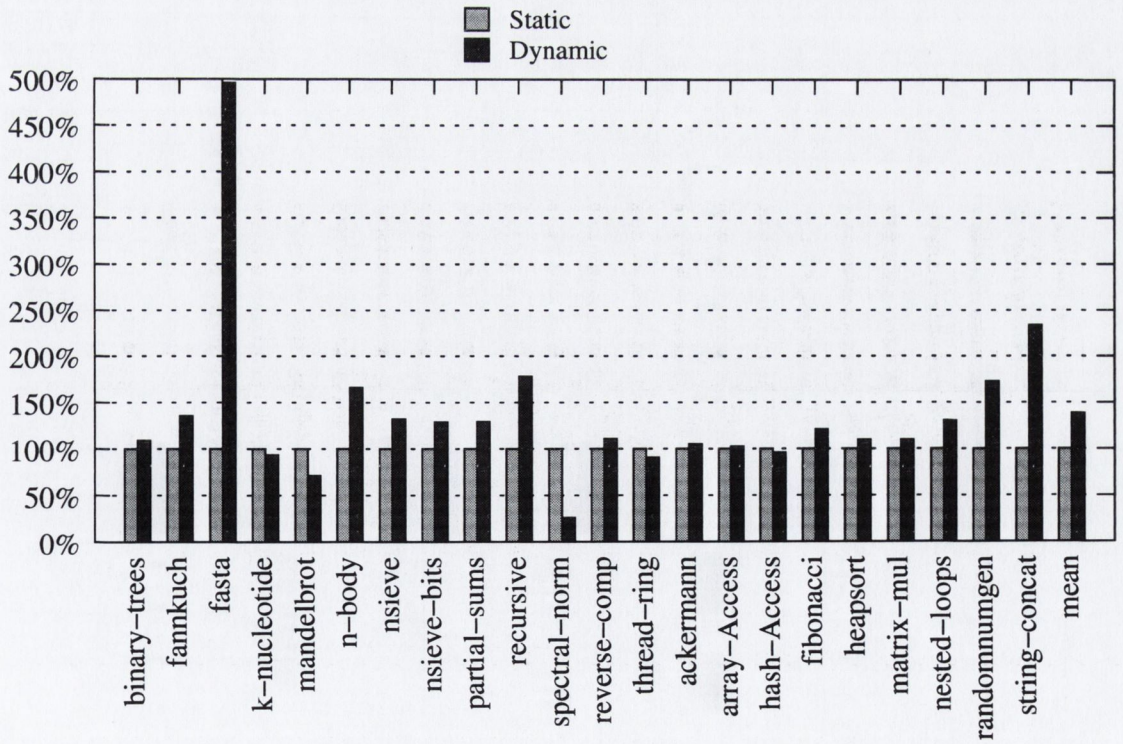


Figure 6.14: Relative measure of **Level 2 data cache** writes.

Benchmark	DIR	SIR	X	Static Runtime	DIR as % of Runtime
binary-trees	11200B (10KB)	520B	21.5	30816KB	<0.01%
fannkuch	14216B (13KB)	532B	26.7	76KB	0.17%
fasta	40544B (39KB)	2140B	18.9	128KB	0.3%
k-nucleotide	21672B (21KB)	948B	22.9	18461KB	<0.01%
mandelbrot	10288B (9KB)	716B	14.4	545KB	0.02%
n-body	25976B (25KB)	1164B	22.3	88KB	0.28%
nsieve	8184B (7KB)	368B	22.2	134KB	0.05%
nsieve-bits	10848B (10KB)	440B	24.7	137KB	0.07%
partial-sums	10072B (9KB)	536B	18.8	72KB	0.12%
recursive	10656B (10KB)	488B	21.8	212KB	0.04%
spectral-norm	10392B (10KB)	516B	20.1	175KB	0.06%
reverse-comp	21280B (20KB)	1060B	20.1	327KB	0.06%
thread-ring	5072B (4KB)	216B	23.5	705KB	<0.01%
ackermann	4968B (4KB)	240B	20.7	218KB	0.02%
array-Access	3800B (3KB)	168B	22.6	197KB	0.01%
hash-Access	3496B (3KB)	156B	22.4	133952KB	<0.01%
fibonacci	2984B (2KB)	128B	23.3	72KB	0.03%
heapsort	10392B (10KB)	432B	24.1	134KB	0.07%
matrix-mul	9016B (8KB)	392B	23	367KB	0.02%
nested-loops	4168B (4KB)	184B	22.7	69KB	0.06%
randomnumgen	4536B (4KB)	216B	21	66KB	0.06%
string-concat	9568B (9KB)	400B	23.9	9245KB	<0.01%
mean	11480B (11.2KB)	535B	21.9	8918KB	0.07%

Table 6.1: Intermediate representation memory usage

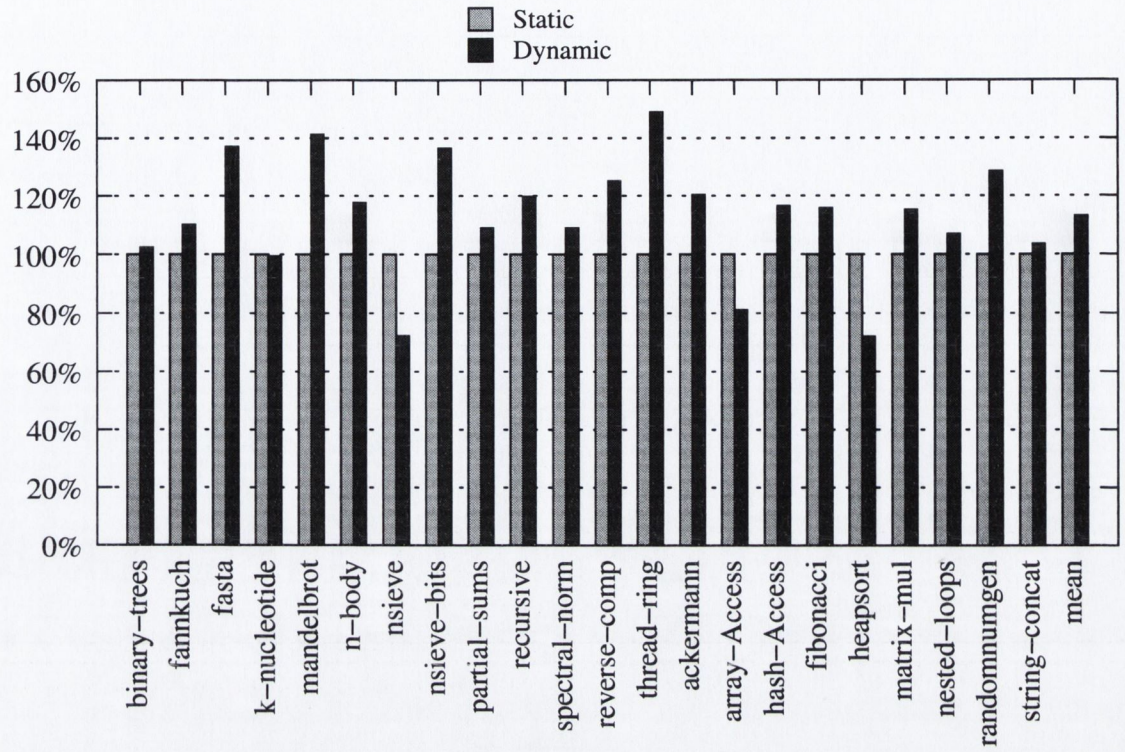
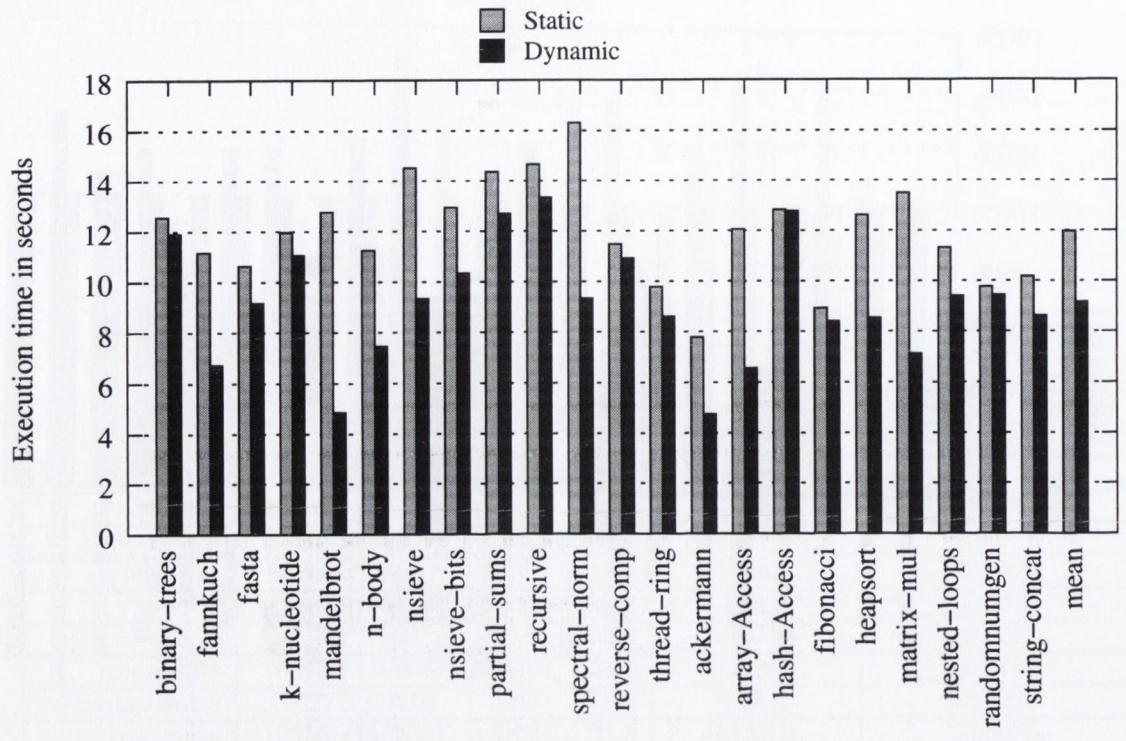


Figure 6.15: Relative measure of **Level 2 instruction cache** misses.

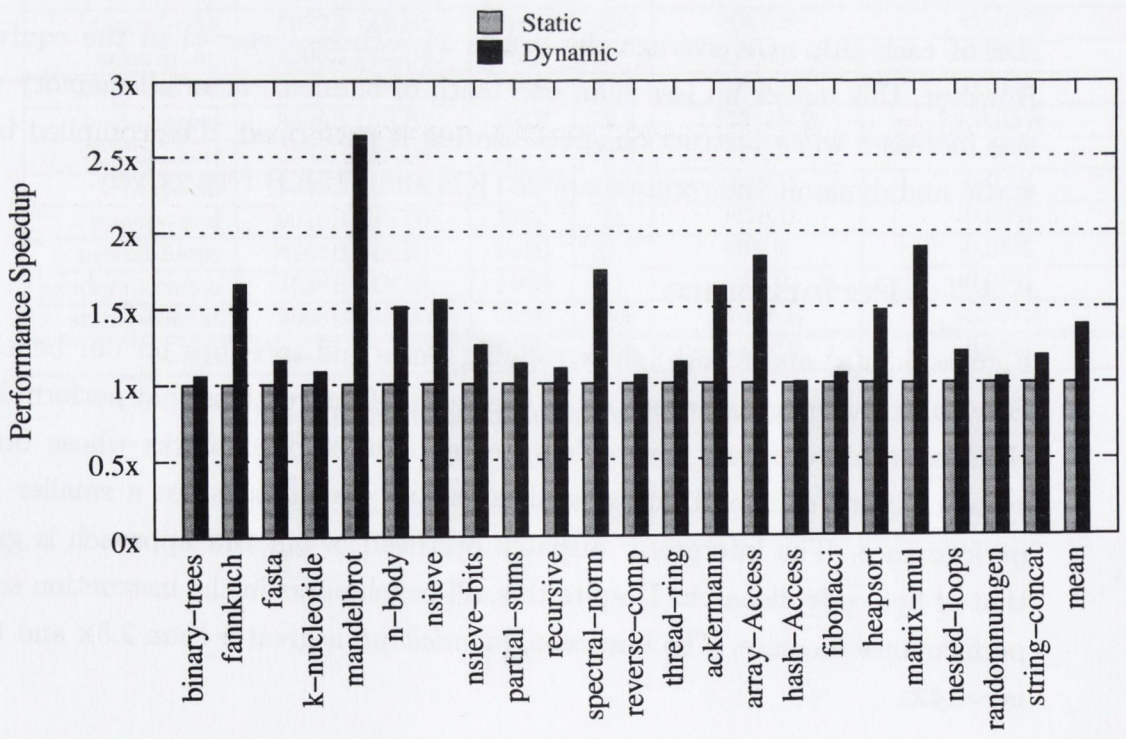
size of each DIR. On average the DIR is 21.9 times larger than the equivalent SIR. However, this makes up less than one tenth of one percent of all memory use. Code size increases when instruction specialization is performed. The compiled binaries for static and dynamic interpreters are 261KB and 315KB respectively.

6.4.7 Performance

Figures 6.16(a) and 6.16(b) show running times and speedups for our benchmark set. Benchmarks whose bottleneck is the execution of instructions that perform lots of type checking achieve a very favorable speedup. Other benchmarks whose bottleneck is system library calls and non-specializable Lua instructions see a smaller increase in performance. The interpreter dispatch overhead of our DIR approach is greater than that of bytecode dispatch. Despite this, all benchmarks in the instruction set achieve a performance increase. The best case, *mandelbrot*, is greater than 2.5X and the average is ~1.4X.



(a) Execution time in seconds



(b) Speed up from a baseline of the static interpreter

Figure 6.16: A comparison of micro benchmark run-time performance

6.5 Scope for Optimization

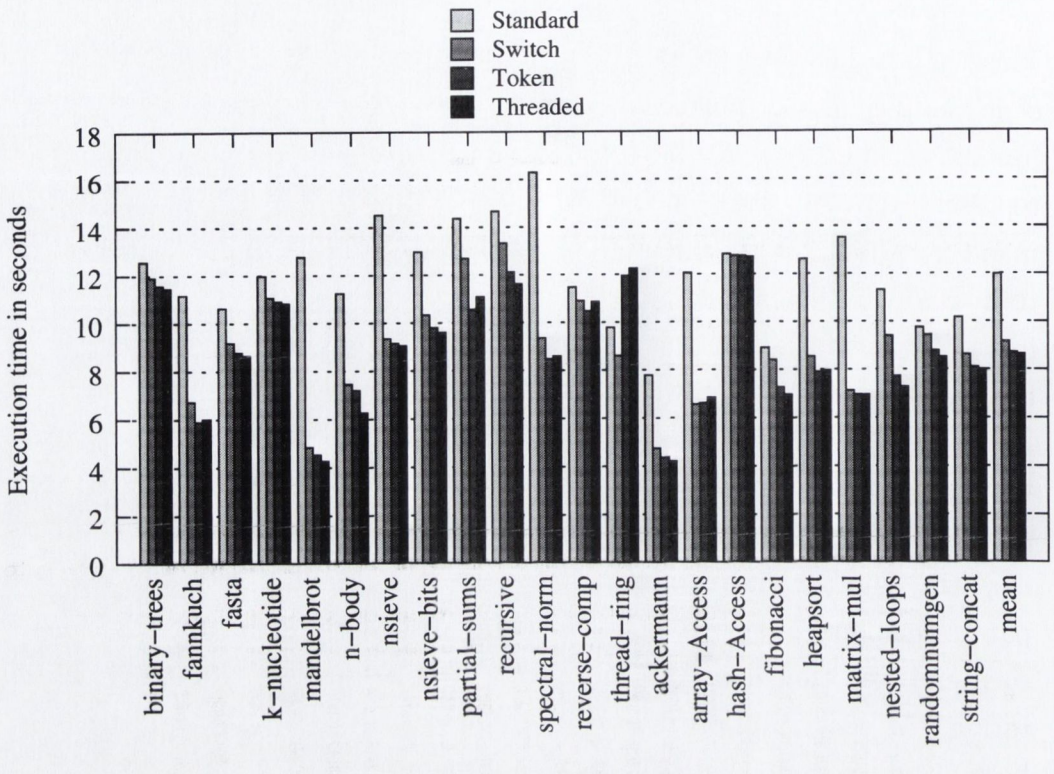
This chapter has presented a dynamic intermediate program representation that encodes both dynamic control flow and variable type flow through a whole program. It has shown interpreter dispatch techniques which enable the efficient execution of the representation. It finally presented analysis of the technique; comparing its performance to that of a static equivalent. In order to present a fair comparison of both techniques this chapter has so far neglected the possibilities of optimization of the dynamic representation. In this section we present some thoughts on our future work and the types of optimization we plan to improve performance in the future.

Interpreter Dispatch: The Lua 5.1 implementation is designed to be as portable as possible. For these reasons, the Lua authors have used a switch based interpreter dispatch. Our current representation copies this dispatch technique. More efficient dispatch techniques have been established in previous work. Threading techniques [Bel73] such as token threading and direct threading are equally applicable to our representation. As specialization reduces the bottleneck of type checks, instruction dispatch becomes more important to overall performance.

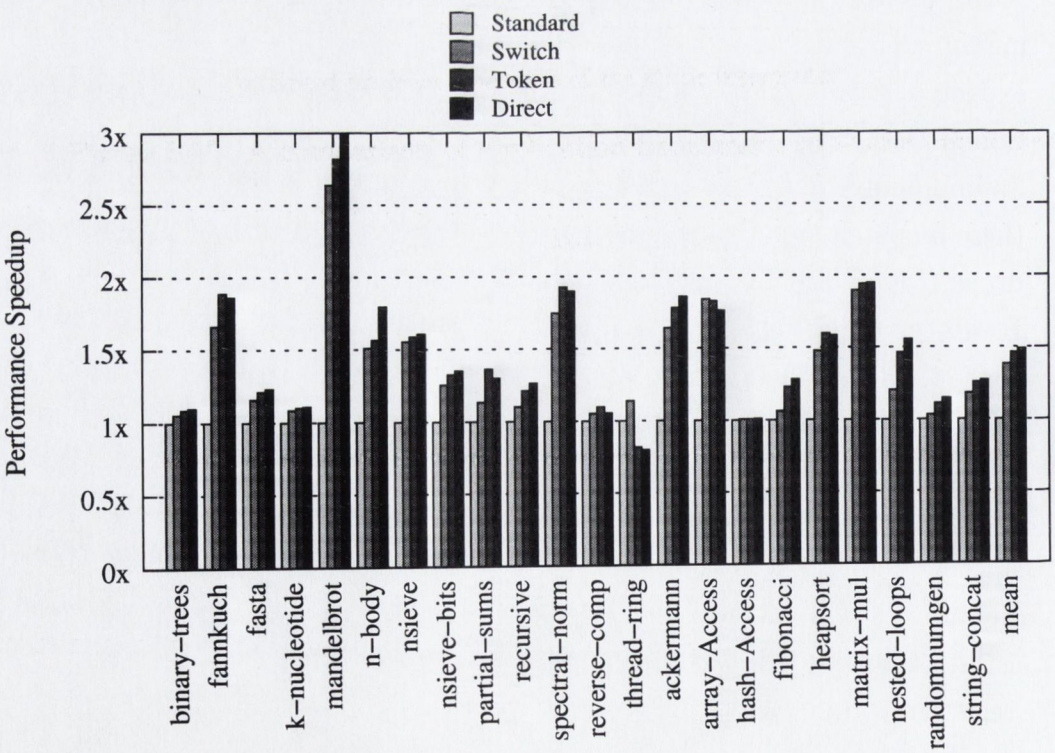
Figure 6.17 presents performance measurements for all types of dispatch. They are measured from a base of *'Standard'* which is the switch of the static interpreter. As expected, gradual performance increases are achieved with each threaded implementation. Direct threading of the dynamic interpreter brings performance to a best case of 3x and an average case of 1.5x. Figure 6.18(b) presents the prediction rates for each of these dispatch techniques. The average mis-prediction rate for dynamic interpretation drops from ~80% for switch dispatch to ~30% for threaded dispatches.

Register Caching: A popular optimization among stack based virtual machines is to cache the top value(s) of the stack in machine registers [Ert95]. An equivalent approach to caching virtual registers in machine registers is to date unestablished[2]. Using our dynamic representation we plan to specialize instructions with implicit register operands. Due to the number of virtual registers it is not advisable to cache all registers. The authors plan to create a mapping of machine registers to virtual registers. Region boundaries will be given header and footer nodes for loading and storing cached registers. Caching will be implemented by need, meaning only the most frequently used registers will be cached.

[2]It is in-fact a goal of Squirrel-fish, a JavaScript virtual machine. See http://webkit.org/blog/189/announcing-squirrelfish/

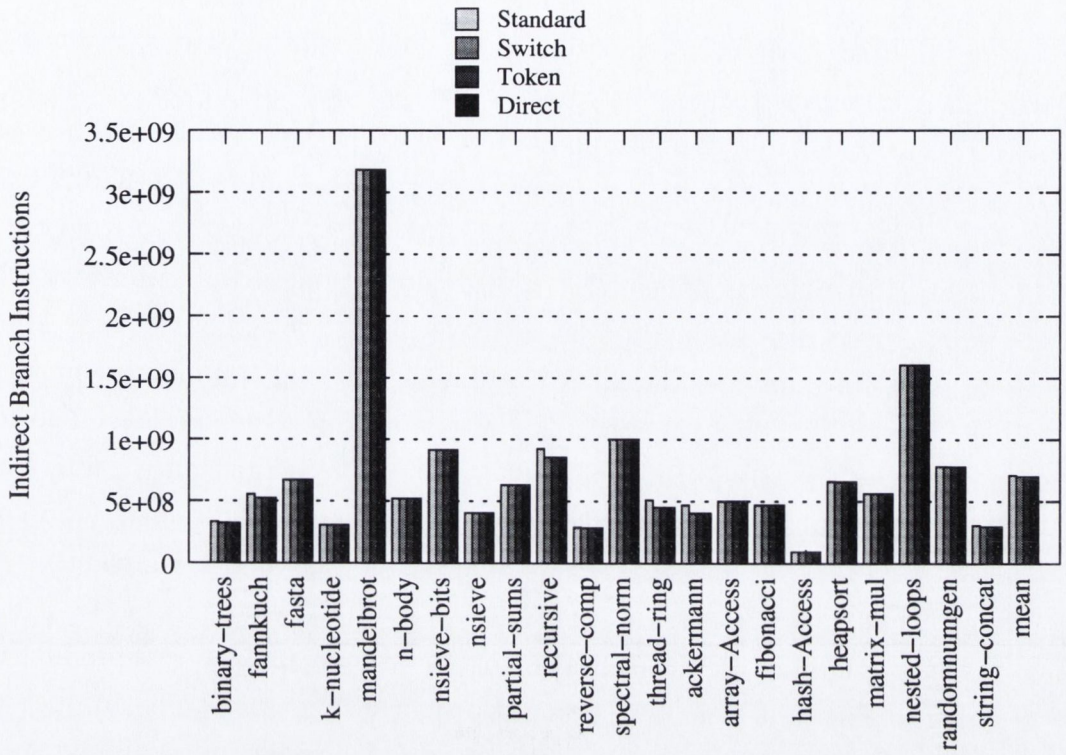


(a) Execution time in seconds

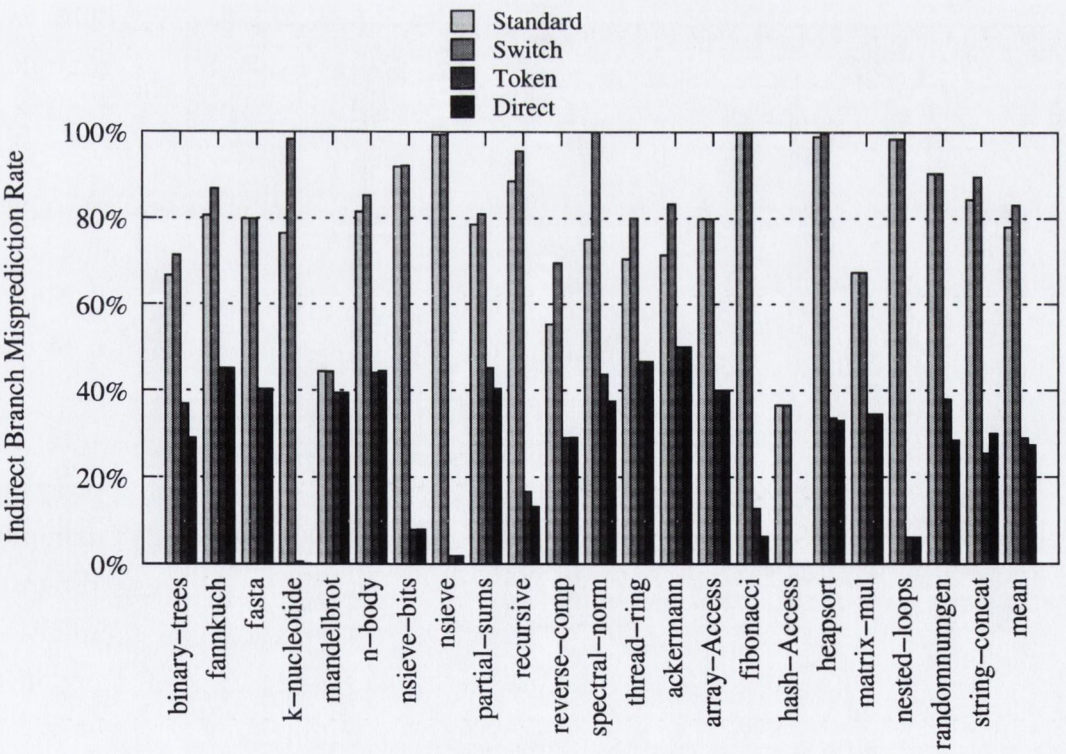


(b) Speed up from a baseline of the static interpreter

Figure 6.17: A comparison of application benchmark run-time performance



(a) Indirect branch instructions (valgrind simulation)



(b) Indirect branch mis-prediction rate (valgrind simulation)

Figure 6.18: Branch behavior for various dispatch techniques

Super Nodes: Another common interpreter optimization is to concatenate common pairs of instructions together to form a single instruction [Pro95]. This technique is equally applicable to dynamic representations. The authors plan to use established techniques in the field to implement super node optimizations [EGKP02; CGE05].

Loop Optimizations: Loops often provide opportunities for optimization in compilers — as is the case in our interpreter. We plan to leverage the type knowledge of variables inside loops to eliminate array bounds checking and move memory allocation outside the loop body. Table access nodes can be further specialized to 'cache' the array pointer; we believe this will lead to a massive performance gain. The number type in Lua is implemented using 'doubles'. We plan to optimize loop counters by specializing them to integers when we know the loop bound is constant.

Dead Node Removal: Specialization of operations can lead to redundant nodes. For example in Figure 6.7, node t specializes the TEST instruction to an operand of type NIL. The result of this operation is always false and control flow will always follow the FALSE path. This is illustrated in Figure 6.7 as there is only a single path leaving the TEST node, which is a conditional node. The TRUE path is never built as it is never executed. Nodes specialized to this level can be removed and all entry nodes can be directed to the target exit node.

Library Nodes: Scripting languages often provide standard libraries to perform complex operations like STRING, MATH and IO functions. The authors plan to inline these operations into the interpreter. This will remove the stack incrementing and decrementing required for their execution and will improve performance of the functions themselves as they are heavily inlined into the interpreter.

JIT Compilation: Mixed-mode virtual machines begin executing programs in a profiling interpreter. When a *'hot'* region of code is identified a JIT compiler compiles the opcodes to a native machine code representation of the same program. Individual JIT compilers often favor a single level of granularity as a compilation unit (e.g. functions, loops or traces). Optimization and specialization of a region of code can be based on: (1) analysis performed at JIT time [Pal08] or (2) recording a single run of execution and compiling based on behavior found during that execution [GES+09]. Future executions of the same region of code may or may not follow the same type and control flow as the compiled region and accordingly will remain in native code or

request the compilation of further paths. This approach to mixed-mode execution can cause common program flow structures such as nested loops to make multiple calls to the JIT compiler. While modern JIT compilers have established workarounds to this problem, the behavior remains undesirable.

We suggest that the use of our DIR will improve the current model of static interpretation followed by speculated specialization. When a hot region of code is discovered, a JIT compiler could compile to native code guided by the verbose profile of the DIR. Optimizations of control-flow paths and specialization of operations could be performed with greater precision, while granularity levels could be determined based on the needs and behavior of the hot region. For example, a loop containing an if-then-else statement as a body may be compiled at either a loop or trace granularity. A trace would be chosen when either of the conditional paths were executed more often than the other. However, in the case where both paths are equally likely, the JIT compiler could choose to compile the whole loop and avoid further compilation and eliminate expensive entering and exiting of native regions. In summary, we claim our proposed representation has two advantages over current models: (1) specialization can be achieved at all levels of execution from interpretation to native execution of JIT code and (2) more effective JIT shapes and levels of specialization could be achieved as compilation is directed by a full program profile created by the dynamic interpreter.

6.6 General Applicability

The DIR presented in this chapter is implemented in the Lua virtual machine. However, the technique is applicable to other scripting languages — some require minor changes to their variable structures and some would specialize instructions in different ways to Lua. For example, in Python, arithmetic instructions can overflow from one type (*int*) into another (*long*). Arithmetic behavior like this would be modeled with a type-directed node (see Figure 6.3(a)). In general, dynamic scripting language implementations have some *'tagged value'* structure to define *(type,value)* pairs. In Lua, these tagged values have a byte which encodes one of the nine possible types. Our dynamic interpreter uses this byte to index the array of targets for node dispatch. In Python, tagged values use a pointer to a type object; however objects of 'built-in' types can be augmented with extra information. The data structure for objects of these types would need to be extended with a type indicator byte. In PHP, tagged values called *ZVals* have a type byte which encodes one of eight basic types. JavaScript implementations differ in their bytecode format. In the SpiderMonkey implementation,

the tagged value, termed *jsval*, is a machine word with three bits representing the type, thus some decoding would be required.

6.7 Related Work

Scripting language specialization has been the topic of only a few recent research publications. All of these research efforts have been directed at either ahead-of-time compilation and analysis or just-in-time compilation and specialization of run-time variables.

As discussed in Section 3.2.4 and Section 5.8, the most relevant work to this chapter is that of other scripting specialization. In our literature survey we discussed Gal's tracing JIT compiler for the JavaScript virtual machine running in the Firefox web browser [GES+09]. These techniques successfully show that compiling at a trace granularity allow the specialization of JIT compiled code. They used *type guards* at trace entry and *side-exits* at selected points in the trace to guarantee type-safe execution. Their interpreter, which begins the execution of all programs and performs the profiling of control flow (but not type) uses *'static interpretation'* techniques. The other major scripting specialization work is Rigo's psyco [Rig04]. This is a run-time specialization technique for the Python programming language. It performs run-time specialization *by need* using a mixed execution/specialization phase of execution to specialize JIT code fragments.

Biggar et al. presented *phc*, an ahead-of-time compiler for the PHP language. They performed static analysis of PHP programs and developed a compiler technique that links with the language's interpreter to enable correctness through future iterations of the language [BdG09]. Jensen et al. [JMT09] present a static analysis infrastructure for the JavaScript language. Their technique uses type inference and pointer analysis to achieve precision rates greater than 90% in a large number of benchmarks. Their technique, while powerful, has some shortcomings. In their worst case they achieve 61% precision. These results show that the use of static analysis alone is not sufficient to infer large numbers of types in all programs. The dynamic representation presented in this chapter guarantees 100% precision for all run-time local variables.

Larus presented whole program paths which capture a program's complete control flow [Lar99]. An outcome of this work was effective discovery of *hot subpaths* for programmers and compilers to optimize. In Ammons' and Larus' retrospective piece [AL04] they observe the success of program path optimizations in the JIT community. The DIR in this chapter similarly builds program paths to improve interpreter efficiency.

Kistler and Franz [KF99] pioneered the concept of using tree structures in the JavaVM. The advantage of their tree representation over the original bytecode was a more compact representation that contained more high-level information that improved JIT compilation.

6.8 Discussion

This chapter has presented a novel approach to scripting language specialization. The approach is the first stage of a project which plans to bring program specialization to all levels of execution from interpretation to post-JIT native execution. Our experiments have shown our interpretation technique to be more efficient than existing techniques for scripting languages. In future work we plan to bring further efficiency to both interpretation and JIT compilation.

While our approach has shown performance to encourage adoption of our technique in the wider community, we have not addressed some of the potential scaling issues with our representation. For the most part scripting languages are in general type-stable, but the possibility still exists for a dynamic representation to grow to an unsuitably large size. A study of potential profiles is required to analyze the risk of programs which are pathological in type and/or control flow. Possible solutions to any such programs could be to (1) limit the number of paths allowed by creating unspecialized paths in problem regions, (2) remove paths which may have become obsolete or (3) require the existence of a static interpreter as a fall back processing unit.

We have claimed that our dynamic representation will bring improvements to the area of JIT specialization. Our future work includes plans to build an experimental JIT compiler which will make run-time decisions about compilation units. Leaving these compilation decisions to run-time should improve the quality of JIT compilation as whole program profiles are available to guide appropriate compilation units.

Chapter 7

Final Thoughts

Chapters 4 – 6 present some optimizations of virtual machine interpreters. This chapter presents some proposals for the future direction of interpreter and scripting language implementations. We complete this chapter with a summary of our contributions.

7.1 Thread-level Interpreter Optimization

Recall Figure 2.5, the architecture of the CellVM. The CoreVMs in this figure run individual Java threads on each of the Cell BE's sub-processors. Each thread executes a sub-set of Java instructions in a unique order. We propose that with static profiling an optimized and compact interpreter can be created for each thread at run-time. Static selection of superinstructions will create a compact bytecode representation of each thread. The interpreter will be compact as it will only support a small sub-set of Java instructions, i.e. those instructions found in the thread.

Each thread (and CoreVM) initialization will be created with a unique interpreter. The technique has similarities to Hoogerbrugge's code compression with pipeline interpreters [HATVDW99]. In this work, non-critical parts of a program were compiled to bytecode and executed with the interpreter. The motivation for this work was to compress the whole application.

In this proposal, we suggest optimizing CoreVM interpreters using similar compression techniques, while also optimizing performance using static superinstruction techniques [Cas05]. At first reading, it may seem that taking this approach an obvious step further results in a mixed-mode system with a traditional JIT compiler and therefore we should expect performance to similarly improve. However, the latest implementation of CellVM already has a JIT compiler. The resulting CoreVM produced by this JIT compiler is a factor of 12 times larger than the interpreted equivalent

	PyPy	Parrot	V8	TraceMonkey	L2J
Language	Python	Perl	JavaScript	JavaScript	Lua
Interpreter	Yes	Yes	No	Yes	Yes
Static Analysis	Yes	Yes	Yes	No	No
JIT	Yes	No	No	Yes	Yes
Trace	No	No	No	Yes	Yes
Specialize Types	Yes	Yes	Yes	Yes	No
Portability	C/Java/.NET	C	x86	x86	x86

Table 7.1: Scripting Language Implementations

[NGF06]. The bottleneck for the majority of applications running on the CellVM is throughput. Many data transfer requests cause the DMA to block and the CoreVM to suspend execution. We therefore suggest that a highly optimized interpreter executing a highly compact instruction format will out perform a larger CoreVM created using traditional JIT compilation.

7.2 Scripting Language Architecture

Scripting languages are implemented with virtual machines. The majority of scripting languages have adopted the techniques and architecture of classic virtual machines like the JVM. They compile the high-level source code into a low-level bytecode. An interpreter then executes this bytecode resolving all variable-types with run-time checks. For performance improvements some implementations provide a JIT extension to transform bytecode into native code. The analysis and profiling performed for optimization varies from implementation to implementation. In general, program profiling simply discovers hot regions of code using instruction counts. Stop-the-world analysis is then performed on the hot region. Optimization is based on types discovered from this analysis.

The Problem: Speculative specialization of scripting languages results in undesirable program behavior. JIT compilation decisions are based on a single profile count of an instruction. Compilation can fail and compiled regions can collide resulting in poor performance and duplication of JITed code.

The Solution: A stronger profiling stage building whole program paths with rich dynamic profiles will guide JIT specialization to only compile regions of code that are known to be profitable.

Table 7.1 presents some well-known scripting language implementations. In order to present a broad flavor of some various approaches to scripting language implementation. A brief discussion of each follows in this section. Section 7.2.1 presents our proposed alternative approach. This proposal is based on the experiences of the scripting language implementations presented in Chapters 5 and 6 of this dissertation.

PyPy

PyPy [RP06] is a Python translation framework. The aim of this framework is to remove virtual machine implementation details away from the language designer. Instead the designer writes a simple interpreter description that defines the language. PyPy takes this input and outputs a concrete virtual machine implementing the language. The virtual machine output by PyPy can target several platforms (i.e. C, Java and .NET). The reasoning for the author's PyPy approach is based on their categorization of scripting language implementations into three problems[1]:

- l: the design of the language to implement.
- o: the implementation of the language and the optimization of platform specific code.
- p: the support of various target platforms.

The construction of the PyPy translation framework helps solve all of the above issues to some extent. To date, it has not resulted in a re-evaluation of the run-time specialization problem we wish to solve.

Parrot

Parrot is a virtual machine which supports a number of different languages. Its implementation is targeted towards the Perl language and it is the official run-time for Perl 6. Parrot's design has one point of interest when it comes to scripting language specialization. Its intermediate representation, PIR, has an interestingly novel approach to program specialization. PIR has statically typed instructions (e.g. *iadd* to add two integers) and the ParrotVM has specialized register sets for each specialized set of instructions. However, this specialization is limited and it does not continue to program variables. These are instead implemented as *PMC*s. PMCs are equivalent to TValue's in Lua and implement all the more complex structures in Perl. So while specialization exists, it is limited to small sets of control variables.

[1] http://codespeak.net/pypy/dist/pypy/doc/architecture.html

V8

V8 is the JavaScript engine in the Google Chromium web-browser. It adopts many of the solutions developed in the Self project. V8 provides fast prototyping and efficient garbage collection. However, it has no interpreter and no run-time profiling of local variable is used for optimization.

TraceMonkey

TraceMonkey is the JavaScript engine in the Firefox web-browser. It is a trace-based JIT. Execution begins in the interpreter where function calls and backward branches are profiled. Hot regions of code are specialized based on types of variables that appear during a single iteration executed at JIT time. While this approach achieves specialization of types appearing at run-time, it still contains a certain level of speculation. Future performance is dependent on the same types appearing consistently in future executions. It is desirable to have a more precise type knowledge based on a verbose history of variable types.

L2J

L2J is the next release of LuaJIT. It is also a trace based JIT compiler. It has a very fast interpreter hand coded in assembler and has optimized method dispatch. The performance of L2J, as reported to the author in private correspondence, is very impressive. Performance can be attributed to the interpreter and optimized dispatch as well as the tracing-JIT (~2x–5x with interpretation alone, and ~2x–75x with tracing-JIT). While all of the approaches of L2J result in performance improvements, they are heavily target dependent (x86) and hence are little value to a language designer who requires portability in their language. L2J does not perform type specialization at any point in program execution. Instead, L2J's representation of TValues uses doubles. It assumes values to be of type number. When the value of the double is *NaN*, the value is not-a-number, some decoding is performed to extract the actual type and value information. L2J is an impressive implementation of the Lua language, however it does not provide a portable solution for the run-time specialization problem.

7.2.1 A New Approach

Based on the experiences of Chapters 5 and 6 we propose a new approach to scripting language execution and specialization. Chapter 6 has shown that a type specialized

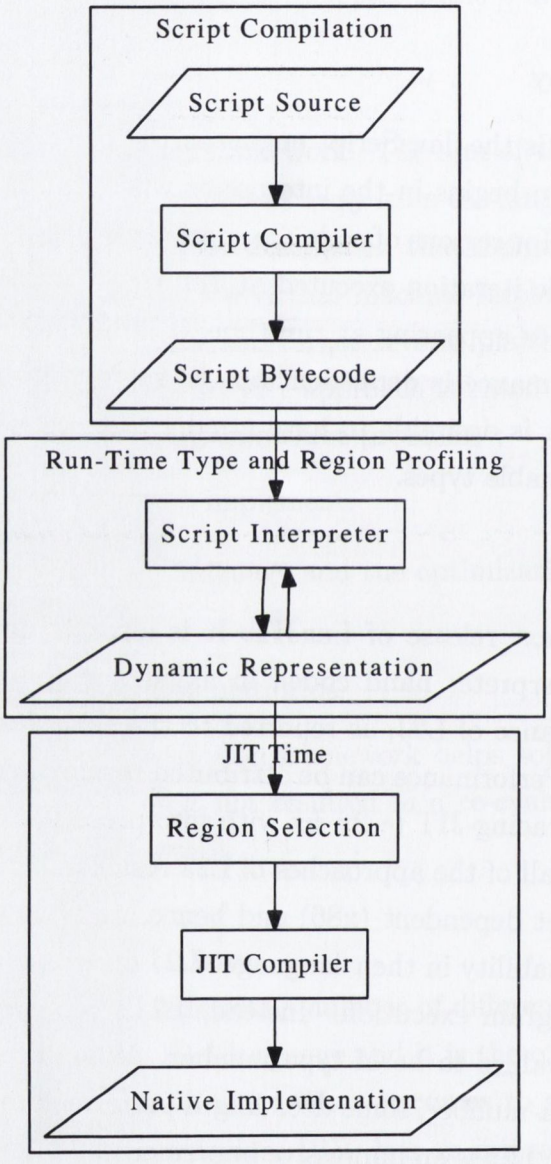


Figure 7.1: A new scripting language pipeline

interpreter is not just practical but also efficient. Further, in the realm of JIT compilation, it provides a verbose profile of complete program execution. This profile is an excellent source of direction for a JIT compiler. Little further analysis is required with such a complete whole program profile. Instead, a JIT can potentially generate code directly from the interpreted code. Traces have proven to be a very useful code-shape when it comes to JIT compilation. We propose that a run-time selection of code shape to be a better approach than only using a single shape. This idea is supported by experimentation from Bruening and Duesterwald [BD00]. Analysis of the DIR at JIT time can allow for a run-time selection of JIT shape (function, loop or trace). To this end we propose an execution pipeline similar to the one presented in Figure 7.1. Advantages of this new pipeline are:

1. Fast specialized interpretation.
2. Optimal JIT shapes.
3. Complete specialization of native code.

This new approach would be the first scripting language implementation that specializes execution from the first instruction to the last.

7.3 A Lua 6 Proposal

Chapter 6 presents a dynamic interpretation technique for the Lua programming language. The implementation for this technique is built on top of the existing internals for Lua 5.1.4. In this section, we outline some proposed changes to the Lua internals. These changes are aimed at making a more compact representation of our DIR and delivering more opportunities for specialization. More importantly, the proposed changes do not infringe any of the design principles of Lua. The language remains as extensible and portable as before. This section outlines changes to the basic type set as well as the instruction format. We complete the proposal with a specialized instruction set.

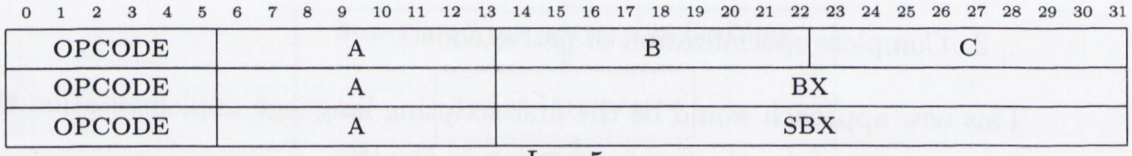
Basic Types

Table 7.2 presents a new set of basic types for Lua. The three proposed changes are:

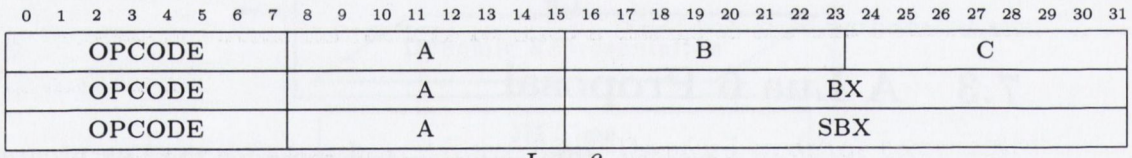
1. A new INTEGER type. This type will not be visible to the programmer. It is added to optimize loop indices and allow fast array access.

Lua 5	*'Lua 6'*
NIL	NIL
BOOLEAN	BOOLEAN
LIGHTUSERDATA	LIGHTUSERDATA
	INTEGER
NUMBER	NUMBER
STRING	STRING
TABLE	TABLE
	TABLE_META
FUNCTION	LUA_FUNCTION
	C_FUNCTION
USERDATA	USERDATA
THREAD	THREAD

Table 7.2: The Lua5 versus Lua6 basic types



Lua 5



Lua 6

Table 7.3: The Lua 5 versus *'Lua 6'* instruction format

2. Adding an additional TABLE_META type. This type will differentiate between TABLE and TABLE_META improving table access speed. Again, the TABLE_META will not be visible to the programmer.

3. Returning to the FUNCTION types of Lua 3.0. We split FUNCTION types into LUA_FUNCTION and C_FUNCTION for fast dispatch.

Instruction Format

The increase in virtual instructions requires a change to the Lua 5 instruction format. The changes are:

1. Increase opcode size from 6-bit to 8-bit. Required to define more instructions.

2. Reduce arguments B and C from 9-bit to 8-bit. The extra bit in arguments B and C are used to define local and constant registers. Instructions are now specialized to access either local or constant registers, making the extra bit redundant.

3. Reduce arguments BX and SBX from 18-bit to 16-bit.

4. Instruction remains 32 bits wide.

Instruction Set

Table 7.4 presents the new specialized instruction set for *'Lua 6'*. There is no addition of any new instructions into the set. Rather, each instruction has a set of specializations which can be applied to a given set of operands. The table columns are (1) the index, (2) the opcode, (3,4,5) the specialized register access for operands (local(R), constant(K) or upvalue (U)) and (6,7,8) the specialization of operands to either a type or a constant value.

The full instruction set presented should not be considered a requirement for all implementations of a specialized Lua interpreter. The *DIR* which implements a program is by definition, dynamic. The instruction set can therefore be tuned to the level of specialization the interpreter designer would like. The only forced limitation on the set in its current form is the requirement to differentiate between register and constant access.

This requirement can be changed in two ways, (1) limit the number of registers to indexed in a 7-bit value rather than 8 bits — this leaves the extra bit to signal local or constant, or (2) increase the instruction size to greater than 32-bits to accommodate merged local/constant addressing.

7.4 Conclusions

Chapter 4 presented a pipelined interpreter for the Cell BE. We showed that creating a new pipeline stage that fetches an operation's operand provides further gains over previously published work. The limitations of the Cell BE's abilities to interpret JVM bytecodes efficiently were also identified. They include the large hint to branch stall and the long load latency. The aim of the CellVM is to run individual Java threads on the sub-processors of the Cell BE. Therefore, each CoreVM is only required to execute a subset of instructions, namely those that appear in the thread. Our work did not attempt to optimize the interpreter based on this fact. In Section 7.1, we presented ideas for the future direction of thread-level interpreter optimization.

Chapter 4 also presents a software implemented virtual instruction branch predictor using the Cell BE's hint instructions. The predictor predicts the direction of the Java program. This is the first application of a branch predictor inside a virtual machine interpreter. However, the study is far from complete. It only investigated a single approach to branch prediction. A more complete study is required to develop the concept to completion. In order for such a study and future implementations to be maintainable compiler support is required to place indirect branch hints in source code. At the time of our research this feature was not available and hint instructions had to be hand scheduled in assembly code.

We believe that virtual machines are a useful tool in assisting programmers to target complex heterogeneous processors. From our experiences developing on the Cell BE, we also believe the engineer-hours required for development on current heterogeneous processors is inhibitive for their future deployment. A collaborative approach between hardware designer and virtual machine implementer is required to produce a programmer-friendly high-performance system which can deliver (1) performance for the user and (2) low development costs for the programmer.

Chapter 5 presents a mixed-mode Lua virtual machine. It profiles local variable types at a loop and function level. When a hot region and set of types is found C code is generated for the function and is compiled by a native compiler. The code is linked and dispatched in a type safe manner. The signature scheme provides a safe mechanism to link and dispatch JIT specialized code. Our performance on micro-benchmarks proved promising. However, to date our handling of function calls and lack of inlining of JITed functions has weakened performance on our application suite. Future development of efficient calls between JITed regions is likely to improve the performance of the application suite.

The type inference of Chapter 5 showed that tables were a major source of failure in the type analysis of Lua. We brain-stormed many ideas to rectify this issue. One possible solution is to record all variable types when they are stored in the table. A table can then be identified as either storing a mix of values or a set of values of a particular type. There are two issues with this solution. The first is the extra work added to a table store instruction. The second is the issue of the dynamic nature of tables. Accessing table keys which have no value associated with them is legal in Lua (the operation should return *nil*). The second idea is to direct execution based on the type of a value fetched from a table. The issue with this is creating an efficient representation for this behavior. In Chapter 6, we presented a novel intermediate representation for the Lua language. The representation is constructed at run-time by the interpreter.

For every change in type a new path is built in the intermediate representation. This results in complete specialization of execution. The current implementation successfully shows, for the first time, that specialized interpretation of a dynamic language can be implemented efficiently.

Future iterations of our implementation will concentrate on three issues: (1) the size of the produced DIR should be reduced, (2) more of the opportunities for specialization should be explored and (3) function calling should be improved (possibly integrating library functions into the interpreter). One of the most attractive features of the representation is its portability. Current experimentation has being limited to IA architectures. We would like to see further experimentation on more architectures to properly evaluate it's real world potential.

The results of the scripting language research presented in this dissertation has produced experimental results to direct the practical aspects of future scripting language implementations. Section 7.2 and 7.3 present the authors vision of the next generation of scripting language virtual machine and a proposal for the implementation of the next release of the Lua virtual machine.

Specialization of scripting languages, currently in its infancy, is an important problem for the future performance of scripting languages. Today, no scripting language implementation performs specialization at the interpreter level nor do any JITs make run-time decisions about the shapes of regions. This research has presented a solution to the former, the latter is yet to be researched. Lua 5.1 is the fastest interpreted scripting language. However, performance can be increased. L2J has shown that by using architecture specific *tricks*, performance can be gained. Unfortunately any non-portable solution is of little use to the wider Lua community. The proposal for *'Lua 6'* in this Chapter is a portable solution for specialization in Lua.

#	OPCODE	REG			SPECIALIZATION		
		A	B	C	A	B	C
1	MOVE	R	R	-	NIL	NIL	
2		R	R	-	BOOLEAN	BOOLEAN	
3		R	R	-	LIGHTUSERDATA	LIGHTUSERDATA	
4		R	R	-	NUMBER	NUMBER	
5		R	R	-	INTEGER	INTEGER	
6		R	R	-	STRING	STRING	
7		R	R	-	TABLE	TABLE	
8		R	R	-	TABLE_META	TABLE_META	
9		R	R	-	LFUNCTION	LFUNCTION	
10		R	R	-	CFUNCTION	CFUNCTION	
11		R	R	-	USERDATA	USERDATA	
12		R	R	-	THREAD	THREAD	
13		R	R	-		NIL	
14		R	R	-		BOOLEAN	
15		R	R	-		LIGHTUSERDATA	
16		R	R	-		NUMBER	
17		R	R	-		STRING	
18		R	R	-		TABLE	
19		R	R	-		TABLE_META	
20		R	R	-		LFUNCTION	
21		R	R	-		CFUNCTION	
22		R	R	-		USERDATA	
23		R	R	-		THREAD	
24	LOADK	R	K	-	NIL	NIL	
25		R	K	-	BOOLEAN	BOOLEAN	
26		R	K	-	LIGHTUSERDATA	LIGHTUSERDATA	
27		R	K	-	NUMBER	NUMBER	
28		R	K	-	INTEGER	INTEGER	
29		R	K	-	STRING	STRING	
30		R	K	-	TABLE	TABLE	
31		R	K	-	TABLE_META	TABLE_META	
32		R	K	-	LFUNCTION	LFUNCTION	
33		R	K	-	CFUNCTION	CFUNCTION	
34		R	K	-	USERDATA	USERDATA	
35		R	K	-	THREAD	THREAD	
36		R	K	-		NIL	
37		R	K	-		BOOLEAN	
38		R	K	-		LIGHTUSERDATA	
39		R	K	-		NUMBER	
40		R	K	-		INTEGER	
41		R	K	-		STRING	
42		R	K	-		TABLE	
43		R	K	-		TABLE_META	
44		R	K	-		LFUNCTION	
45		R	K	-		CFUNCTION	
46		R	K	-		USERDATA	
47		R	K	-		THREAD	
48	LOADBOOL	R	-	-		0	0
49		R	-	-		0	1
50		R	-	-		1	0
51		R	-	-		1	1
52		R	-	-	BOOLEAN	0	0
53		R	-	-	BOOLEAN	0	1
54		R	-	-	BOOLEAN	1	0
55		R	-	-	BOOLEAN	1	1
56	LOADNIL	R	R	-			
57	GETUPVAL	R	U	-			
58	GETGLOBAL	R	K	-		STRING	
59	GETTABLE	R	R	K			
60		R	R	R			
61		R	R	R		TABLE	NUMBER
62		R	R	K		TABLE	NUMBER
63		R	R	R		TABLE	INTEGER
64		R	R	K		TABLE	INTEGER
65		R	R	R		TABLE	STRING
66		R	R	K		TABLE	STRING

#	OPCODE	REG			SPECIALIZATION		
		A	B	C	A	B	C
67	SETGLOBAL	R	K	-		STRING	
68	SETUPVAL	R	U	-			
69	SETTABLE	R	R	R			
70	SETTABLE	R	R	K			
71	SETTABLE	R	K	R			
72	SETTABLE	R	K	K			
73		R	R	R		TABLE	NUMBER
74		R	R	K		TABLE	NUMBER
75		R	R	R		TABLE	INTEGER
76		R	R	K		TABLE	INTEGER
77		R	R	R		TABLE	STRING
78		R	R	K		TABLE	STRING
79	NEWTABLE	R	-	-			
80	SELF	R	R	R			
81		R	R	K			
82		R	R	R		TABLE	NUMBER
83		R	R	K		TABLE	NUMBER
84		R	R	R		TABLE	INTEGER
85		R	R	K		TABLE	INTEGER
86		R	R	R		TABLE	STRING
87		R	R	K		TABLE	STRING
88	ADD	R	R	R			
89		R	R	K			
90		R	K	R			
91		R	R	R		NUMBER	NUMBER
92		R	R	K		NUMBER	NUMBER
93		R	K	R		NUMBER	NUMBER
94		R	R	R		INTEGER	INTEGER
95		R	R	K		INTEGER	INTEGER
96		R	K	R		INTEGER	INTEGER
97	SUB	R	R	R			
98		R	R	K			
99		R	K	R			
100		R	R	R		NUMBER	NUMBER
101		R	R	K		NUMBER	NUMBER
102		R	K	R		NUMBER	NUMBER
103		R	R	R		INTEGER	INTEGER
104		R	R	K		INTEGER	INTEGER
105		R	K	R		INTEGER	INTEGER
106	MUL	R	R	R			
107		R	R	K			
108		R	K	R			
109		R	R	R		NUMBER	NUMBER
110		R	R	K		NUMBER	NUMBER
111		R	K	R		NUMBER	NUMBER
112		R	R	R		INTEGER	INTEGER
113		R	R	K		INTEGER	INTEGER
114		R	K	R		INTEGER	INTEGER
115	DIV	R	R	R			
116		R	R	K			
117		R	K	R			
118		R	R	R		NUMBER	NUMBER
119		R	R	K		NUMBER	NUMBER
120		R	K	R		NUMBER	NUMBER
121		R	R	R		INTEGER	INTEGER
122		R	R	K		INTEGER	INTEGER
123		R	K	R		INTEGER	INTEGER
124	POW	R	R	R			
125		R	R	K			
126		R	K	R			
127		R	R	R		NUMBER	NUMBER
128		R	R	K		NUMBER	NUMBER
129		R	K	R		NUMBER	NUMBER
130		R	R	R		INTEGER	INTEGER
131		R	R	K		INTEGER	INTEGER
132		R	K	R		INTEGER	INTEGER

#	OPCODE	REG			SPECIALIZATION		
		A	B	C	A	B	C
133	UNM	R	R	-			
134		R	R	-		NUMBER	
135		R	R	-		INTEGER	
136	LEN	R	R	-			
137		R	R	-		STRING	
138		R	R	-		TABLE	
139		R	R	-		TABLE_META	
140	NOT	R	R	-			
141		R	R	-		NIL	
142		R	R	-		BOOLEAN	
143	CONCAT	R	-	-			
144	CONCAT	R	-	-	STRINGS		
145	JMP	-	sBx				
146	EQ	-	R	R	0		
147		-	R	K	0		
148		-	K	R	0		
149		-	R	R	1		
150		-	R	K	1		
151		-	K	R	1		
152		-	R	R	0	BOOLEAN	BOOLEAN
153		-	R	K	0	BOOLEAN	BOOLEAN
154		-	K	R	0	BOOLEAN	BOOLEAN
155		-	R	R	1	BOOLEAN	BOOLEAN
156		-	R	K	1	BOOLEAN	BOOLEAN
157		-	K	R	1	BOOLEAN	BOOLEAN
158		-	R	R	0	NUMBER	NUMBER
159		-	R	K	0	NUMBER	NUMBER
160		-	K	R	0	NUMBER	NUMBER
161		-	R	R	1	NUMBER	NUMBER
162		-	R	K	1	NUMBER	NUMBER
163		-	K	R	1	NUMBER	NUMBER
164		-	R	R	0	INTEGER	INTEGER
165		-	R	K	0	INTEGER	INTEGER
166		-	K	R	0	INTEGER	INTEGER
167		-	R	R	1	INTEGER	INTEGER
168		-	R	K	1	INTEGER	INTEGER
169		-	K	R	1	INTEGER	INTEGER
170	LT	-	R|K	R|K	0|1		
171		-	R	R	0	NUMBER	NUMBER
172		-	R	K	0	NUMBER	NUMBER
173		-	K	R	0	NUMBER	NUMBER
174		-	R	R	1	NUMBER	NUMBER
175		-	R	K	1	NUMBER	NUMBER
176		-	K	R	1	NUMBER	NUMBER
177		-	R	R	0	INTEGER	INTEGER
178		-	R	K	0	INTEGER	INTEGER
179		-	K	R	0	INTEGER	INTEGER
180		-	R	R	1	INTEGER	INTEGER
181		-	R	K	1	INTEGER	INTEGER
182		-	K	R	1	INTEGER	INTEGER
183		-	R	R	0	STRING	STRING
184		-	R	K	0	STRING	STRING
185		-	K	R	0	STRING	STRING
186		-	R	R	1	STRING	STRING
187		-	R	K	1	STRING	STRING
188		-	K	R	1	STRING	STRING

#	OPCODE	REG			SPECIALIZATION		
		A	B	C	A	B	C
189	LE	-	R|K	R|K	0|1		
190		-	R	R	0	NUMBER	NUMBER
191		-	R	K	0	NUMBER	NUMBER
192		-	K	R	0	NUMBER	NUMBER
193		-	R	R	1	NUMBER	NUMBER
194		-	R	K	1	NUMBER	NUMBER
195		-	K	R	1	NUMBER	NUMBER
196		-	R	R	0	INTEGER	INTEGER
197		-	R	K	0	INTEGER	INTEGER
198		-	K	R	0	INTEGER	INTEGER
199		-	R	R	1	INTEGER	INTEGER
200		-	R	K	1	INTEGER	INTEGER
201		-	K	R	1	INTEGER	INTEGER
202		-	R	R	0	STRING	STRING
203		-	R	K	0	STRING	STRING
204		-	K	R	0	STRING	STRING
205		-	R	R	1	STRING	STRING
206		-	R	K	1	STRING	STRING
207		-	K	R	1	STRING	STRING
208	TEST	R	-	-			0 |1
209		R	-	-	BOOLEAN		0
210		R	-	-	BOOLEAN		1
211	TESTSET	R	R	-			0 |1
212		R	R	-	BOOLEAN		0
213		R	R	-	BOOLEAN		1
214	CALL	R	-	-			
215		R	-	-	LFUNCTION		0
216		R	-	-	LFUNCTION		1
217		R	-	-	LFUNCTION		
218		R	-	-	CFUNCTION		
219	TAILCALL	R	-	-			
220	RETURN	R	-	-			
221		R	-	-			0
222		R	-	-			1
223	FORPREP	R	-	-			
224		R	-	-	3NUM		
225	FORLOOP	R	-	-			
226		R	-	-	3NUM		
227		R	-	-	4NUM		
228	TFORLOOP	R	-	-			
229		R	-	-	FUNCTION		
230		R	-	-	FUNCTION		1
231	SETLIST	R	-	-			
232	CLOSE	R	-	-			
233	CLOSURE	R	Bx				
234	VARARG	R	-	-			
235	BUILDNODE	-	-	-			
236		-	-	-	TYPEDIRECTED		
237		-	-	-	CONDITIONAL		

Table 7.4: The *'Lua 6'* specialized instruction set

Bibliography

[ABM+97] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. *Fortran 95 Handbook*. The MIT Press, 1997.

[AFG+00] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive Optimization in the Jalapeno JVM. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, New York, NY, USA, 2000. ACM.

[AL04] Glenn Ammons and James R. Larus. Improving Data-Flow Analysis with Path Profiles. *ACM SIGPLAN Notices*, 39(4):568–582, 2004.

[Ayc03] John Aycock. A Brief History of Just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.

[BBD+89] H. Benker, J. M. Beacco, M. Dorochevsky, Th. Jeffré, A. Pöhlmann, J. Noyé, B. Poterie, J. C. Syre, O. Thibault, and G. Watzlawik. KCM: a Knowledge Crunching Machine. In *ISCA '89: Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 186–194, New York, NY, USA, 1989. ACM.

[BD00] Derek Bruening and Evelyn Duesterwald. Exploring Optimal Compilation Unit Shapes for an Embedded Just-In-Time Compiler. In *Proceedings of the 2000 ACM Workshop on Feedback-Directed and Dynamic Optimization FDDO-3*, pages 13–20, 2000.

[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.

[BdG09] Paul Biggar, Edsko de Vries, and David Gregg. A Practical Solution for Scripting Language Compilers. In *SAC '09: Proceedings of the 2009 ACM Symposium on Applied Computing*, pages 1916–1923, New York, NY, USA, 2009. ACM.

[BE06] Bill Brock and Michael Exerman. Cache Latencies of the PowerPC MPC7451. *Freescale Semiconductor Application Note*, AN2180, 2006.

[Bel73] James R. Bell. Threaded Code. *Communications of the ACM*, 16(6):370–372, 1973.

[Bru96] Bruce K. Holmer, Barton Sano, Michael Carlton, Peter Van Roy and Alvin M. Despain. High-Performance Implementations of Logic Programming Systems. *The Journal of Logic Programming*, 29(3):107–139, 1996.

[BVZB05] Marc Berndl, Benjamin Vitale, Mathew Zaleski, and Angela Demke Brown. Context Threading: A Flexible and Efficient Dispatch Technique for Virtual Machine Interpreters. In *CGO '05: Proceedings of the International Symposium on Code Generation and Optimization*, pages 15–26, Washington, DC, USA, 2005. IEEE Computer Society.

[Cas05] Kevin Casey. *Automatic Generation of Optimised Virtual Machine Interpreters*. PhD thesis, University of Dublin, Trinity College, 2005.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, New York, NY, USA, 1977. ACM.

[CEG07] Kevin Casey, M. Anton Ertl, and David Gregg. Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters. *ACM Transactions on Programming Languages and Systems*, 29(6):37, 2007.

[CGE05] Kevin Casey, David Gregg, and M. Anton Ertl. Tiger - An Interpreter Generation Tool. In *CC '05: Proceedings of the 14th International*

Conference on Compiler Construction, volume 3443 of *Lecture Notes in Computer Science*, pages 246–249. Springer, 2005.

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

[CLB08] CLBG. The Computer Language Benchmarks Game. *Available at http://shootout.alioth.debian.org/*, 2008.

[Com99] ECMAScript Technical Committee. *Standard ECMA-262: ECMAScript Language Specification.* ECMA International, 1999.

[CR96] Alain Colmerauer and Philippe Roussel. The Birth of Prolog. In *History of Programming Languages—II*, pages 331–367, New York, NY, USA, 1996. ACM.

[Cra05] Iain D. Craig. *Virtual Machines.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[CSR+09] Mason Chang, Edwin Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Tracing for Web 3.0: Trace Compilation for the Next Generation Web Applications. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 71–80, New York, NY, USA, 2009. ACM.

[CU89] C. Chambers and D. Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-typed Object-Oriented Programming Language. *ACM SIGPLAN Notices*, 24(7):146–160, 1989.

[CUL89] C. Chambers, D. Ungar, and E. Lee. An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes. *ACM SIGPLAN Notices*, 24(10):49–70, 1989.

[DBC+03] Brian Davis, Andrew Beatty, Kevin Casey, David Gregg, and John Waldron. The Case for Virtual Register Machines. In *IVME '03: Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, pages 41–49, New York, NY, USA, 2003. ACM.

[DDP85] T. P. Dobry, A. M. Despain, and Y. N. Patt. Performance Studies of a Prolog Machine Architecture. In *ISCA '85: Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 180–190, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[DeB04] Erik P. DeBenedictis. Will Moore's Law Be Sufficient? In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, page 45, Washington, DC, USA, 2004. IEEE Computer Society.

[Dew75] Robert B. K. Dewar. Indirect Threaded Code. *Communications of the ACM*, 18(6):330–331, 1975.

[DHPW01] Charles Daly, Jane Horgan, James Power, and John Waldron. Platform Independent Dynamic Java Virtual Machine Analysis: the Java Grande Forum Benchmark Suite. In *JGI '01: Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, pages 106–115, New York, NY, USA, 2001. ACM.

[DMC92] Christophe Dony, Jacques Malenfant, and Pierre Cointe. Prototype-based Languages: from a New Taxonomy to Constructive Proposals and their Validation. *ACM SIGPLAN Notices*, 27(10):201–217, 1992.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 297–302, New York, NY, USA, 1984. ACM.

[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 on Parallel Processing*, pages 403–412, London, UK, 2001. Springer-Verlag.

[EG03] M. Anton Ertl and David Gregg. The Structure and Performance of Efficient Interpreters. *Journal of Instruction-Level Parallelism*, 5:2003, 2003.

[EGKP02] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. Vmgen: a Generator of Efficient Virtual Machine Interpreters. *Softw. Pract. Exper.*, 32(3):265–294, 2002.

[Eic08] Brendan Eich. Mozilla Firefox - Spidermonkey. *Available at http://swik.net/Spidermonkey*, 2008.

[EOO+05] Alexandre E. Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. Optimizing Compiler for the CELL Processor. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 161–172, Washington, DC, USA, 2005. IEEE Computer Society.

[Ert95] M. Anton Ertl. Stack Caching for Interpreters. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 315–327, New York, NY, USA, 1995. ACM.

[Ert96] Martin Anton Ertl. *Implementation of Stack-Based Languages on Register Machines.* PhD thesis, Technisch-naturwissenschaftliche Fakultat, 1996.

[FAFH09] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *SAC '09: Proceedings of the 2009 ACM Symposium on Applied Computing*, pages 1859–1866, New York, NY, USA, 2009. ACM.

[FCI08] Luiz Henrique de Figueiredo, Waldemar Celes, and Roberto Ierusalimschy. *Lua Programming Gems.* Lua.Org, 2008.

[FM08] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language.* O'Reilly Media, 2008.

[Gal06] Andreas Gal. *Efficient Bytecode Verification and Compilation in a Virtual Machine.* PhD thesis, University of California, Irvine, 2006.

[GEK01] David Gregg, M. Anton Ertl, and Andreas Krall. Implementing an Efficient Java Interpreter. In *HPCN Europe 2001: Proceedings of the 9th International Conference on High-Performance Computing and Networking*, pages 613–620, London, UK, 2001. Springer-Verlag.

[GES+09] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris

Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based Just-in-time Type Specialization for Dynamic Languages. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 465–478, New York, NY, USA, 2009. ACM.

[GF08] Andreas Gal and Michael Franz. Incremental Dynamic Code Generation with Trace Trees. *Technical Report: University of California, Irvine*, 2008.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification (3rd Edition)*. Addison-Wesley Professional, 2005.

[GPF06] Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: an Effective JIT Compiler for Resource-Constrained Devices. In *VEE '06: Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 144–153, New York, NY, USA, 2006. ACM.

[GWC08] GWCLS. The Great Win32 Computer Language Shootout. *Available at http://dada.perl.it/shootout/*, 2008.

[HA00] Jan Hoogerbrugge and Lex Augusteijn. Pipelined Java Virtual Machine Interpreters. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, volume 1781/2000, pages 35–49. Springer Berlin / Heidelberg, 2000.

[HATVDW99] Jan Hoogerbrugge, Lex Augusteijn, Jeroen Trum, and Rik Van De Wiel. A Code Compression System Based on Pipelined Interpreters. *Software: Practice and Experience*, 29(11):1005–2023, 1999.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38, London, UK, 1991. Springer-Verlag.

[Hol90] Holmer, Bruce K. and Sano, Barton and Carlton, Michael and Van Roy, Peter and Haygood, Ralph and Bush, William R. and Despain, Alvin M. and Pendleton, Joan M. and Dobry, Tep. Fast Prolog with an

Extended General Purpose Architecture. *SIGARCH Comput. Archit. News*, 18(3a):282–291, 1990.

[HP03] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2003.

[HP07] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2007.

[HSRPD96] B. K. Holmer, B. Carlton Sano, M. Van Roy, and A. M. P. Despain. Design and Analysis of Hardware for High-Performance Prolog. *Journal Of Logic Programming*, 29(1):107–140, 1996.

[HU94a] Urs Hölzle and David Ungar. A Third-Generation SELF Implementation: Reconciling Responsiveness with Performance. In *OOPSLA '94: Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Language, and Applications*, pages 229–243, New York, NY, USA, 1994. ACM.

[HU94b] Urs Hölzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-time Type Feedback. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 326–336, New York, NY, USA, 1994. ACM.

[Hud09] Paul Hudson. *PHP in a Nutshell.* O'Reilly Media, 2009.

[Hug82] R. J. M. Hughes. Super-combinators a New Implementation Method for Applicative Languages. In *LFP '82: Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, pages 1–10, New York, NY, USA, 1982. ACM.

[IBM07] IBM. Cell Broadband Engine Programming Handbook, 4 2007.

[IdFC05] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The Implementation of Lua 5.0. *Journal of Universal Computer Science*, 11(7):1159–1176, July 2005.

[IdFC07] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The Evolution of Lua. In *HOPL III: Proceedings of the Third*

ACM SIGPLAN Conference on History of Programming Languages, pages 2–1–2–26, New York, NY, USA, 2007. ACM.

[Ier06] Roberto Ierusalimschy. *Programming in Lua, Second Edition.* Lua.Org, 2006.

[Inc08] Google Inc. V8 JavaScript Engine. *Available at http://code.google.com/p/v8/*, 2008.

[JMT09] Simon Holm Jensen, Anders Moller, and Peter Thiemann. Type Analysis for JavaScript. In *SAS '09: Proceedings of the 16th International Symposium on Static Analysis*, pages 238–255, Berlin, Heidelberg, 2009. Springer-Verlag.

[Jon96] Richard Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, 1996.

[KCSL00] Iffat H. Kazi, Howard H. Chen, Berdenia Stanley, and David J. Lilja. Techniques for Obtaining High Performance in Java Programs. *ACM Computing Surveys*, 32(3):213–240, 2000.

[KDH+05] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, 49(4/5), 2005.

[KF99] Thomas Kistler and Michael Franz. A Tree-Based Alternative to Java Byte-Codes. *International Journal of Parallel Programming*, 27(1):21–33, 1999.

[Lar99] James R. Larus. Whole Program Paths. *ACM SIGPLAN Notices*, 34(5):259–269, 1999.

[LFC06] Roberto Lerusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. *Lua 5.1 Reference Manual.* Lua.Org, 2006.

[Lut09] Mark Lutz. *Learning Python, Fourth Edition.* O'Reilly Media, 2009.

[LWO08] Tom Christiansen Larry Wall and Jon Orwant. *Programming Perl, Third Edition.* O'Reilly Media, 2008.

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Man03] Kein-Hong Man. A No-Frills Introduction to Lua 5.1 VM Instructions. In *http://chunkspy.luaforge.net/*. Lua Chunkspy Project, 2003.

[Mye77] Glenford J. Myers. The case against stack-oriented instruction sets. *SIGARCH Comput. Archit. News*, 6(3):7–10, 1977.

[NGF06] A. Noll, A. Gal, and M. Franz. CellVM: A Homogeneous Virtual Machine Runtime System for a Heterogeneous Single-Chip Multiprocessor, 2006.

[OKN02] Kazunori Ogata, Hideaki Komatsu, and Toshio Nakatani. Bytecode Fetch Optimization for a Java Interpreter. In *ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–67, New York, NY, USA, 2002. ACM.

[Pal08] M. Pall. The LuaJIT Project. *Available at http://luajit.org*, 2008.

[Pro95] Todd A. Proebsting. Optimizing an ANSI C Interpreter with Superoperators. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 322–332, New York, NY, USA, 1995. ACM.

[PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. The Java Hotspot Server Compiler. In *JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*, Berkeley, CA, USA, 2001. USENIX Association.

[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.

[Rig04] Armin Rigo. Representation-based Just-in-time Specialization and the Psyco Prototype for Python. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 15–26. ACM Press, 2004.

[RLV+96] Theodore H. Romer, Dennis Lee, Geoffrey M. Voelker, Alec Wolman, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henry M.

Levy. The Structure and Performance of Interpreters. In *ASPLOS-VII: Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 150–159, New York, NY, USA, 1996. ACM.

[RP06] Armin Rigo and Samuele Pedroni. PyPy's Approach to Virtual Machine Construction. In *ACM Dynamic Languages Symposium*, New York, NY, USA, 2006. ACM.

[SCEG08] Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual Machine Showdown: Stack versus Registers. *ACM Transactions on Architecture and Code Optimization*, 4(4):1–36, 2008.

[Sch90] Herbert Schildt. *The Annotated ANSI C Standard American National Standard for Programming Languages—C: ANSI/ISO 9899-1990.* Osborne/McGraw-Hill, Berkeley, CA, USA, 1990.

[Shi07] Yunhe Shi. *Virtual Machine Showdown: Stack versus Registers.* PhD thesis, University of Dublin, Trinity College, 2007.

[SOT+00] Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–, 2000.

[SPE98] SPEC. SPECjvm98 Documentation, Release 1.0. *Available at http://www.spec.org/osg/jvm98/jvm98/doc/index.html*, 1998.

[Str03] Bjarne Stroustrup. *The C++ Standard: Incorporating Technical Corrigendum 1.* Wiley, 2003.

[SYN03] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. A Region-Based Compilation Technique for a Java Just-in-time Compiler. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 312–323, New York, NY, USA, 2003. ACM.

[SYN06] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. A Region-Based Compilation Technique for Dynamic Compilers. *ACM Transactions on Programming Languages and Systems*, 28(1):134–174, 2006.

[US87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. *ACM SIGPLAN Notices*, 22(12):227–242, 1987.

[USS97] Augustus K. Uht, Vijay Sindagi, and Sajee Somanathan. Branch Effect Reduction Techniques. *Computer*, 30(5):71–81, 1997.

[Van92] Van Roy, Peter and Despain, Alvin M. High-Performance Logic Programming with the Aquarius Prolog Compiler. *Computer*, 25(1):54–68, 1992.

[VRDW87] P. Van Roy, B. Demoen, and Y. D. Willems. Improving the Execution Speed of Compiled Prolog with Modes, Clause Selection, and Determinism. In *II and Colloquium on Functional and Logic Programming and Specifications (CFLP) on TAPSOFT '87: Advanced Seminar on Foundations of Innovative Software Development*, pages 111–125, New York, NY, USA, 1987. Springer-Verlag New York, Inc.

[War83] David H.D. Warren. An Abstract Prolog Instruction Set. In *Open Publication. Release of Information.*, Menlo Park, CA 94025, USA, 1983. SRI International.

[Wha01] John Whaley. Partial method compilation using dynamic profile information. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 166–179, New York, NY, USA, 2001. ACM.

[Zal08] Mathew Zaleski. *YETI: A Gradually Extensible Trace Interpreter.* PhD thesis, University of Toronto, 2008.

[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.