



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.

COMPILER TECHNIQUES TO IMPROVE INDIRECT
BRANCH PREDICTION

JASON MARSHALL MCCANDLESS

Department of Computer Science
Trinity College, Dublin

A thesis submitted for the degree of
Doctor of Philosophy (Ph.D.)

January 2012



Thesis 9830

Jason Marshall McCandless: *Compiler Techniques to Improve Indirect Branch Prediction*, © January 2012

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/3.0/> or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

SUPERVISOR:

Dr. David Gregg

Chop wood, carry water.

— Zen Proverb

Dedicated to the important people in my life.

DECLARATION

I declare that this thesis has not been submitted as an exercise for a degree at this or any other university and it is entirely my own work.

I agree to deposit this thesis in the University's open access institutional repository or allow the library to do so on my behalf, subject to Irish Copyright Legislation and Trinity College Library conditions of use and acknowledgement.

Dublin, January 2012

SUMMARY

Computers employ a class of branches called indirect branches to realize various programming language features. Multiway branching (switch statements), virtual function dispatch and function calls are all realized through the use of indirect branch instructions. One of the most important uses of multiway branching is the implementation of virtual machine interpreters, which are commonly used to execute programs written in high-level languages with dynamic features.

While hardware methods have been proposed by researchers to improve indirect branch prediction, dedicated support has only recently been realized in commercial processor designs. In this dissertation we propose a set of techniques that are able to improve program performance by making better use of these indirect branch predictor units.

Interpreters are a significant use case of indirect branch instructions. A long-standing question in the design of virtual machine interpreters is whether it is worthwhile to reorder the cases in the main interpreter loop to improve code locality. We investigate this phenomenon using an iterative, metaheuristic approach. We show that the ordering of the cases in the interpreter loop has a significant impact on performance on recent processors due to indirect branch prediction, not instruction cache locality. We propose feedback-directed strategies to achieve better orderings, and evaluate these strategies in the Python and Lua virtual machine interpreters, showing speedups of up to 40%.

Based on knowledge of the branch predictor structure, we present a more general code generation technique to maximize the branch history information available to the predictor. We devise two methods to achieve this, presenting several heuristics. We implement our optimization as an assembly language transformation, and evaluate it for SPEC benchmarks and interpreters using simulated and real hardware, showing indirect branch misprediction decreases.

Since interpreters use indirect branch instructions extensively, implementers sometimes use token-threaded interpreters, which have better indirect branch predictability than interpreters implemented using a switch statement, though they are not portable. We present a code transformation that will detect interpreter-like structures (switch statements inside a loop) and automatically create token-threaded interpreters from them. We implement the optimization

as an assembly language transformation, therefore making it independent of the compiler. We evaluate the impact of our technique on execution time and branch prediction and show speedups of up to 2x for the Lua, Python and Jam (Java) interpreters.

PUBLICATIONS

Some ideas and figures in this dissertation have appeared previously in the following publications (reverse chronological order):

IN PREPARATION

- Jason McCandless and David Gregg. Automatically Detecting and Optimizing Interpreters. To be submitted to *22nd International Conference on Compiler Construction*.

PUBLISHED

- Jason McCandless and David Gregg. Compiler techniques to improve dynamic branch prediction for indirect jump and call instructions. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):24:1–24:20, January 2012. ISSN 1544-3566. URL <http://doi.acm.org/10.1145/2086696.2086703>.
- Jason McCandless and David Gregg. Optimizing Interpreters by Tuning Opcode Orderings on Virtual Machines for Modern Architectures. In *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java, PPPJ '11*, pages 161–170, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0935-6. URL <http://doi.acm.org/10.1145/2093157.2093183>.

Other publications resulting from this Ph.D.:

IN PREPARATION

- Jason McCandless, Kevin Williams and David Gregg. Fast Dynamic Interpretation. To be submitted to *ACM Transactions on Architecture and Code Optimization*.

PUBLISHED

- Kevin Williams, Jason McCandless, and David Gregg. Dynamic Interpretation for Dynamic Scripting Languages. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization, CGO '10*, pages 278–287, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-635-9. URL <http://doi.acm.org/10.1145/1772954.1772993>.
- Kevin Williams, Jason McCandless, and David Gregg. Portable Just-in-Time Specialization of Dynamically Typed Scripting Languages. In Guang Gao, Lori Pollock, John Cavazos, and Xiaoming Li, editors, *Languages and Compilers for Parallel Computing*, volume 5898 of *Lecture Notes in Computer Science*, pages 391–398. Springer Berlin / Heidelberg, 2010. ISBN 978-3-642-13373-2. URL http://dx.doi.org/10.1007/978-3-642-13374-9_27

ACKNOWLEDGMENTS

I am deeply indebted to my supervisor, David Gregg, who showed great patience and incredible support throughout my time as a Ph.D. student. His door was always open whenever I wanted to talk about anything. His strategic approach to all things research and administrivia has been crucial in my completion.

Thanks to the original 'lads' in the lab: Bobb, Kevin, Nick, Paul & Scott for the interesting conversations inside and outside of the Lloyd. My more recent lab companions: Andrew, Avinash, Martin, Mircea & Servesh have been entertaining and helpful likewise.

I would also like to thank Martin O'Riordan for sharing his knowledge of assemblers and linkers. Nicholas Nash was particularly helpful by providing time to discuss the complexity of the NOP insertion problem presented in Chapter 5. Mark Rodgers and M. Anton Ertl deserve credit for reading draft versions of papers that were published from this research.

My girlfriend Anna has been incredibly supportive and patient while I worked on this Ph.D.

My friends have been supportive of me, especially during the final year when my social contact was more limited. In particular I'd like to thank Gareth, Nigel, Tom, Steven, Mark & Ronan.

Chuck Rashleigh's support and unconditional positive regard has been invaluable to me in my year of completion. He knew exactly when I needed to be pushed.

I am grateful to David Abrahamson and James power who examined this thesis and provided many useful corrections.

I express my gratitude towards The Irish Council for Science Engineering and Technology (IRCSET), who provided the funding that made it possible for me

to pursue a Ph.D. in Ireland.

All of this of course would not have been possible but for the help of my dear parents. They have always been supportive of my endeavours and understanding of the direction that I put my attention. Thank you.

CONTENTS

1	INTRODUCTION	1
1.1	Indirect Branches	1
1.2	Interpreters & High-Level Languages	1
1.3	Indirect Branch Prediction	2
1.4	Our Thesis	2
1.5	Contributions	3
1.6	Outline	3
2	BACKGROUND	5
2.1	Introduction	5
2.2	Interpreters	5
2.2.1	Bytecode Interpreters	6
2.2.2	Lua	7
2.2.3	Python	7
2.2.4	Java	8
2.3	Pipelined Processors	9
2.4	Superscalar	10
2.5	Branch Prediction	10
2.5.1	Static Prediction	11
2.5.2	Dynamic Prediction	12
2.5.3	Branch Target Prediction	14
2.5.4	Indirect Branch Prediction	15
2.5.5	Reverse Engineering Branch Predictors	17
2.5.6	Limits of Branch Prediction	18
2.6	Instruction Cache	18
2.7	Dominance Frontiers	19
2.8	Conclusions	21
3	LITERATURE REVIEW	23
3.1	Introduction	23
3.2	Compiler Techniques to Improve Branch Prediction	23
3.2.1	Semi-Static Techniques	23
3.2.2	Reducing Branch Interference	25
3.2.3	Code Placement for Branches	25
3.2.4	Branch Predication	26
3.2.5	Target Predictability for High-level Languages	27

3.2.6	Measurement Bias	27
3.3	Improving Interpreter Performance	28
3.3.1	Introduction	28
3.3.2	Switch	29
3.3.3	Token-Threading	30
3.3.4	Direct-Threading	30
3.3.5	Indirect-Threading	31
3.3.6	Call-Threading	31
3.3.7	Subroutine/Context-Threading	32
3.3.8	Inline-Threading	32
3.3.9	Comparing Threading Techniques	33
3.3.10	Just-In-Time Compilation	33
3.4	Interpreters and Indirect Branch Prediction	34
3.5	Efficient Polymorphic Calls	35
3.6	Conclusions	37
4	TUNING VIRTUAL MACHINE OPCODE ORDERINGS	39
4.1	Introduction	39
4.2	Background	41
4.3	Most Frequent First	42
4.4	Graph Selection	45
4.5	Monte Carlo Generation	48
4.6	Hardware Analysis	52
4.6.1	Cache	54
4.6.2	Branch Prediction	54
4.6.3	Branch prediction as a Cost Function	55
4.7	Feedback-Directed Search	57
4.7.1	Simulated Annealing	57
4.7.2	Hill Climbing	59
4.8	Experimental Methodology	63
4.8.1	Benchmarks	63
4.8.2	Hardware Configuration	64
4.8.3	Bytecode Traces	64
4.8.4	Compiler Label Alignment	64
4.9	Related Work	65
4.10	Conclusions	66
5	IMPROVING BRANCH PREDICTION FOR INDIRECT JUMPS AND CALLS	69
5.1	Introduction	69

5.2	Background	70
5.3	NOP Insertion	73
5.3.1	Idea	73
5.3.2	Problem Formulation	73
5.3.3	Heuristics	74
5.3.4	Simulation Results	76
5.4	Re-ordering	77
5.4.1	Idea	77
5.4.2	Problem Formulation	77
5.4.3	Heuristics	78
5.5	Hybrid	79
5.6	Performance Evaluation	79
5.6.1	Implementation	79
5.6.2	Pin Simulation	81
5.6.3	Indirect-Branch-Heavy Programs	83
5.6.4	C++ Programs	88
5.6.5	Hardware Configuration	91
5.7	Related Work	91
5.7.1	Reducing Branch Interference	91
5.7.2	Code Placement for Branches	92
5.7.3	Target Predictability for High-Level Languages	92
5.7.4	Compiler Assisted Branch Prediction	93
5.7.5	'Weird' Behavior	93
5.8	Conclusions	94
6	DETECTING AND AUTOMATICALLY CONVERTING INTERPRETERS	95
6.1	Introduction	95
6.2	Background	96
6.3	Converting Switch to Token	99
6.3.1	'Join Block' Method	101
6.3.2	Frontier Method	103
6.4	Bounds Check Removal	104
6.5	Experimental Results	105
6.5.1	Implementation	105
6.5.2	Experimental Setup	106
6.5.3	Lua	106
6.5.4	Python	107
6.5.5	Java	109
6.5.6	Instruction Cache	111

6.6	Related Work	111
6.6.1	Generating Token-Threaded Interpreters	111
6.6.2	Code Grafting	113
6.6.3	LLVM indirectbr Instruction Lowering	114
6.6.4	Interpreter Layout	114
6.7	Conclusions	114
7	FINAL THOUGHTS	117
7.1	Introduction	117
7.2	Practical Applications	117
7.3	NOP Insertion vs. Opcode Ordering	118
7.4	Efficacy of Feedback-directed Search Techniques	118
7.5	NP-Completeness of NOP Insertion and Re-ordering	119
7.6	Measurement Practices	119
7.7	Are Efficient Indirect Branches Necessary?	120
7.8	Hardware Support for Interpreters	121
7.9	Indirect Branch Prediction Hardware Development	122
7.10	Interpreter Optimization	122
7.11	Optimizing Assembly Code	123
7.12	Architecture-aware Compiler Optimizations	124
7.13	Future Work	124
7.14	Summary & Conclusions	125
	BIBLIOGRAPHY	127

LIST OF FIGURES

Figure 2.1	Instruction Pipeline.	10
Figure 2.2	One-bit and two-bit branch predictors.	13
Figure 2.3	A two level adaptive predictor.	14
Figure 2.4	A branch target buffer. Some bits from the instruction pointer are used to index the cache-like structure. If there is an entry and it is predicted as taken then the processor should begin fetching at the predicted target.	15
Figure 2.5	Memory hierarchy.	18
Figure 2.6	A simple control flow graph and its corresponding dominator tree.	20
Figure 4.1	Typical interpreter structure.	41
Figure 4.2	Opcode profile graph for the Lua program: 'for i = 1,3 do print(i) end'.	46
Figure 4.3	Histogram showing the frequencies of different ordering execution times found by Monte Carlo.	49
Figure 4.4	Monte Carlo progress on Lua fannkuch benchmark.	49
Figure 4.5	Execution frequency of opcodes and their implementation positions in memory.	53
Figure 4.6	Hardware performance data for the Lua interpreter executing the fasta benchmark.	56
Figure 4.7	Progress over time of simulated annealing on Lua fannkuch benchmark on Nehalem.	57
Figure 4.8	Simulated annealing results for the Lua interpreter on Core 2.	59
Figure 4.9	Progress over time of the first-choice stochastic hill climbing algorithm.	60
Figure 4.10	Progress over time of the first-choice stochastic hill climbing with error consideration and periodic restarts algorithm.	62
Figure 4.11	First-choice stochastic hill climbing with periodic restarts (and error consideration) algorithm results.	63
Figure 5.1	Bits of the indirect branch target address used to update the path information register in Intel processors.	72

- Figure 5.2 Our Pin model for the indirect branch target buffer. Dashed lines represent the path information register update. 81
- Figure 5.3 Simulated iBTB misprediction rate for baseline vs. NOP insertion vs. re-ordering. vs. hybrid. 82
- Figure 5.4 Simulated iBTB misprediction rate reduction for three techniques compared to baseline. (Smaller is better.) 82
- Figure 5.5 Lua misprediction rate for baseline vs. NOP insertion for a selection of benchmarks. 85
- Figure 5.6 Lua speedups (due to fewer mispredictions) for the NOP insertion technique. Showing an average speedup of 7%. The *mandelbrot* benchmark achieves an almost 80% speedup. 86
- Figure 5.7 Lua misprediction rate for baseline vs. re-ordered indirect branch targets. 86
- Figure 5.8 Lua misprediction rate for baseline vs. hybrid indirect branch targets. 87
- Figure 5.9 Lua indirect branch misprediction rate for the baseline vs. NOP insertion vs. re-ordering vs. hybrid for a selection of benchmarks. (Smaller is better). 87
- Figure 5.10 Lua level 1 instruction cache miss rate compared to baseline for the three techniques for a selection of benchmarks. (Smaller is better.) 88
- Figure 5.11 Java misprediction rate for baseline vs. NOP insertion vs. re-ordering for a selection of benchmarks. 89
- Figure 5.12 Java speedups for the three techniques compared to the baseline (Larger is better). 89
- Figure 5.13 Java level 1 instruction cache miss rate for the baseline vs. NOP insertion vs. re-ordering vs. hybrid for a selection of benchmarks. (Smaller is better). 90
- Figure 6.1 Control flow graphs showing switch and token dispatch mechanisms. 98
- Figure 6.2 Lua results comparing the baseline switch interpreter, the automatically generated interpreters using the single join method and the frontier method, and our hand-coded token-threaded interpreter. (Bounds check removal is also performed.) 108

Figure 6.3	Lua bounds check removal results on the baseline (switch) interpreter. 109
Figure 6.4	Python results comparing the baseline switch interpreter, the automatically generated interpreters using the single join method and the frontier method, and the Python token-threaded interpreter. (Bounds check removal is also performed.) 110
Figure 6.5	Python bounds check removal results on the baseline (switch) interpreter. 111
Figure 6.6	Java results comparing the baseline switch interpreter, the automatically generated interpreters using the single join method and the frontier method, and the Jam token-threaded interpreter. (Bounds check removal is also performed.) 112
Figure 6.7	Java bounds check removal results on the baseline (switch) interpreter. 113

LIST OF TABLES

Table 4.1	Opcode frequencies for the Lua spectral norm benchmark. 43
Table 4.2	Most frequent first speedups on Core 2. Each benchmark is executed by an interpreter tuned using its profile. 44
Table 4.3	Most frequent first speedups on Nehalem. Each benchmark is executed by an interpreter tuned using its profile. 45
Table 4.4	Graph selection speedups on Core 2. Each benchmark is executed by an interpreter tuned using its profile. 47
Table 4.5	Graph selection speedups on Nehalem. Each benchmark is executed by an interpreter tuned using its profile. 48
Table 4.6	Monte Carlo generation speedups on Core 2 (individually tuned orderings). 50

Table 4.7	Monte Carlo generation speedups on Core 2 (using one Pareto optimally tuned ordering).	51
Table 4.8	Monte Carlo generation speedups on Nehalem (individually tuned orderings).	51
Table 4.9	Monte Carlo generation speedups on Nehalem (using one Pareto optimally tuned ordering).	51
Table 4.10	PAPI performance counter results for Lua showing stalled cycles due to branch misspredictions and cache misses as a percentage of total cycles. The 'total' row is the ratio of the total number of stalled cycles due to cache misses (or branches) to the total number of cycles from running all benchmarks.	52
Table 5.1	Simulated average NOP insertions for various heuristics for 60 target addresses and modulo 64.	76
Table 5.2	Simple interpreter results.	84
Table 5.3	Simple interpreter (C++) results.	91

LISTINGS

Listing 3.1	Typical ANSI C source code for a switch interpreter.	29
Listing 3.2	C source code for a token-threaded interpreter using the (non-ANSI C) <i>labels-as-values</i> extension.	30
Listing 3.3	Code to convert a bytecode into direct threaded code dynamically.	31
Listing 3.4	Worst case scenario for inline caching.	37
Listing 4.1	Problem of joined opcode implementations in Python.	44
Listing 5.1	Typical jump-table assembly compiled by GCC for our experimental interpreter.	73
Listing 5.2	Our simple interpreter.	84
Listing 6.1	Typical ANSI C source code for a (portable) switch interpreter.	97
Listing 6.2	C source code for a (non-portable) token-threaded interpreter using the (non-ANSI C) <i>labels-as-values</i> extension.	97

- Listing 6.3 Using multiple switch statements to create unique dispatch locations for each opcode. 99

LIST OF ALGORITHMS

2.1	Computing dominance frontiers.	21
4.1	Brunthaler's graph selection algorithm.	46
4.2	Simulated annealing algorithm used to traverse the solution space.	58
4.3	First-choice stochastic hill climbing with periodic restarts (and error consideration) algorithm.	61
5.1	Greedily insert NOPS	75
5.2	Insert NOPS, minimizing the conflicts at each point.	75
5.3	Insert NOPS, maximizing the distance until a conflict occurs with the offsetting at each point.	76
5.4	Re-order 'blocks' using a 'pair-swap'.	78
5.5	Re-order 'blocks' using a greedy fit.	79
5.6	Hybrid: re-order 'blocks' using a greedy fit, insert NOPS if necessary.	80
6.1	Computing the <i>canReachDispatch</i> set.	100
6.2	Computing the data structure for the number of targets that stop dominating.	102
6.3	Computing the frontier of edges where targets stop dominating.	103

INTRODUCTION

Multiway branching is an important programming technique which is all too often replaced by an inefficient sequence of if tests.

— Knuth [1974]

1.1 INDIRECT BRANCHES

Indirect branch (and call) instructions are an important class of branching in computer systems. They are used in the implementation of:

- Switch statements (using a jump table).
- Function calls where function pointers are used by the programmer and their destinations cannot be resolved statically.
- Virtual function calls in object-oriented languages such as C++.

1.2 INTERPRETERS & HIGH-LEVEL LANGUAGES

In 2012, the most popular programming languages are those which provide a large amount of expressive power to the programmer, enabling rapid program development, portability and safety.¹ These kinds of languages are typically multi-paradigm, reflective² and dynamically typed.

These high-level languages usually require run-time support, and are difficult to compile, so they are often targeted at virtual machines. These virtual machines in turn need to be run on real computers. The usual way of doing this is to use an interpreter to implement the virtual machine.

Another option is to use a just-in-time (JIT) compiler to execute the program. JIT compilers produce very fast code for the most commonly executed parts

¹ The GitHub code repository host—one of the most popular collections of open source software—lists JavaScript, Ruby, Python, Shell and Java as the top five languages. Six out of the top ten are dynamically typed languages. <https://github.com/languages>

² A reflective language enables the creation of programs that can introspect and modify their structure and behavior at run-time.

of a program, but interpreters are nonetheless common for both software engineering and performance reasons. Interpreters are simpler than JIT compilers, and consume less memory both for the interpreter itself and the generated code. Interpreters make tools such as debuggers and profilers easy to build. This profiling information can be useful to identify hot spots and common paths. As a result, many modern VMs use a profiling interpreter that executes code that is unprofitable to JIT compile. For example, the TraceMonkey [Gal et al., 2009] JavaScript engine uses a profiling interpreter.

Another feature that programmers have come to expect in high-level languages is object polymorphism [Cardelli and Wegner, 1985]. The dispatching of function calls on objects with multiple inheritance is usually accomplished by indirect branches [Dixon et al., 1989]. Research has shown that modern object-oriented languages like C++ use significantly more indirect branch instructions than classic imperative languages such as C [Calder et al., 1994]. In Java, JIT compilation, native interface routines (which are implemented with indirect function calls) and virtual methods are significant sources of indirect branches in the compiled code [Li and John, 2001].

1.3 INDIRECT BRANCH PREDICTION

Modern processors utilize deep pipelines to enable them to be run at high frequencies and execute instructions speculatively to maximize throughput. Branch prediction is a technique that is used to avoid some pipeline stalls. One class of branches, indirect branches, are notoriously hard to predict because they have multiple possible targets which can change from one execution of a particular branch instruction to the next. These branches can limit processor performance and waste energy.

Historically, indirect branches received attention much later than prediction for other types of branches, but effective strategies have been proposed and are now appearing in production microprocessors.

1.4 OUR THESIS

Indirect branch prediction for ordinary programs can be improved by adapting the compiler to perform NOP insertion and re-ordering; and for interpreter-like programs by using feedback-directed techniques for ordering and through tail duplication based on block dominance analysis.

1.5 CONTRIBUTIONS

We identify several significant contributions that this work makes:

- We show that the layout of interpreter opcodes noticeably affect performance due to indirect branch mispredictions, rather than instruction cache misses as previous literature assumed.
- We provide useful automatic methods to improve the opcode layout in interpreters using feedback-directed techniques.
- We present an opportunity to improve indirect branch prediction by removing or reducing aliasing in the branch predictor.
- We establish the open problems of the complexity of optimal NOP insertion and optimal reordering and we present heuristics for these two problems that produce good solutions in practice.
- We describe an optimization to automatically detect interpreters or interpreter-like structures created with switch statements and convert them to a more efficient token-threaded version.
- We evaluate all of our optimizations on modern processors through extensive experiments.

1.6 OUTLINE

It is prudent at this point to describe how we have structured the rest of this dissertation. Firstly, we describe some background material that is necessary to understand the context of the research: this appears in Chapter 2. Following that, in Chapter 3, we survey the landscape of previous work that is relevant to the core ideas in this dissertation, discussing compiler techniques to improve branch prediction, interpreter optimization techniques, the previous work linking interpreters and indirect branches, and software approaches to efficient polymorphic calls. In Chapter 4 we investigate the effect opcode orderings have on interpreter performance, and introduce feedback methods to obtain good orderings. Then in Chapter 5 we show a more general method to improve indirect branch performance of programs on modern processors by using NOP insertion and re-ordering and provide algorithms and experimental results. In Chapter 6 we describe an optimization to automatically detect

and convert a switch-based interpreter (or interpreter-like structure) to a faster, token-threaded interpreter. Lastly, we share our final thoughts on the research with a critical eye, and discuss possible future avenues of inquiry in Chapter 7.

BACKGROUND

2.1 INTRODUCTION

Our thesis is about improving indirect branch prediction using compiler methods. To provide a context for the work, in this chapter we seek to familiarize the reader with the concepts required to follow the research material presented later. The chapter also serves to motivate the relevance of the research.

We begin with a description of virtual machines and interpreters, and catalog the interpreted languages on which we perform our experiments to evaluate our research.

The computer architecture subsystems that typically influence program performance are instruction cache, data cache, translation lookaside buffer (TLB), and branch prediction. Since in this dissertation we are concerned with branch prediction (indirect branch prediction in particular) and to a lesser extent instruction cache, we provide material that explains their operation.

We conclude this chapter with an explanation of dominance frontiers. Dominance frontiers (and dominators) are an old idea in compiler research that have been used for various optimizations. We briefly describe them, providing the classic equations to calculate dominators and show a previously devised efficient algorithm to compute the dominance frontier.

2.2 INTERPRETERS

Computer languages may be compiled or interpreted. A compiled language is converted to native code that is directly executed by the target CPU. Interpreted languages, on the other hand, are executed by another program that may be implemented as a native program. The earliest interpreted programming language is widely considered to be Lisp, when Steve Russell wrote the Lisp *eval* function in IBM 704 machine code [Graham, 2004, p. 185].

Some languages require run-time support to realize language features such as *eval* or to resolve the correct implementation of polymorphic code. These high-level languages have numerous advantages over other languages: higher levels of abstraction, incremental debugging facilities, run-time code modifica-

tion and usually platform independence. Interpreters are a natural fit for these languages as they support these dynamic features naturally and can be made to be portable relatively easily.

Interpreters are much slower than the native code produced by just-in-time compilers (even the fastest interpreters are around 5-10 times slower), but they have several advantages that can make them attractive. If an interpreter is written in a high-level language such as C, it can be made portable, and can simply be recompiled for a new target architecture. In contrast, retargeting a just-in-time (JIT) compiler can require a lot of effort. Interpreters also require little memory: the interpreter itself is typically much smaller than a JIT compiler [Radhakrishnan et al., 2000], and the interpreted bytecode is usually a fraction of the size of the corresponding executable native code. For this reason, interpreters are commonly found in embedded systems, or systems such as games consoles, where the memory budget is carefully divided among the components of the system. Furthermore, if the application consists of a lot of code, interpreting rarely-executed code can actually be *faster* than JIT compilation. This is the basis of many hybrid interpreter/JIT systems that interpret the bulk of the code, and JIT compile the hotspots. A further advantage is that the interpreter can collect information about the program as it runs, which can be useful for JIT compilation. Interpreters are also dramatically simpler than compilers; they are easy to construct, and easy to debug. Finally, it is easy to provide tools such as debuggers and profilers when using an interpreter because it is straightforward to insert additional code into an interpreter loop. Providing such tools for native code is much more complex. These advantages make interpreters common among language implementations.

2.2.1 Bytecode Interpreters

Typically, modern high-level languages with dynamic features are first compiled into an intermediate, bytecoded representation for a *virtual machine*¹ (VM).

There are several motivations for bytecode. It is amenable to optimizations, can be compressed and, significantly, interpreters for bytecoded languages offer performance benefits compared to an interpreter that operates on an abstract syntax tree representation. Another advantage of bytecodes is that they are usually portable across machines. The bytecodes that constitute the VM may then be executed by an interpreter or just-in-time compiled to native code. Pascal

¹ In this dissertation we use this term solely to refer to what Smith and Nair [2005] designate as high-level language VMs in their taxonomy.

p-code [Berry, 1978], a stack-oriented instruction set, was the first successful bytecode format.

The amount of work done in a bytecode varies by language and VM design. Higher-level dynamically typed languages must perform a type-check in every operation that works on variables. Languages that perform more work in opcodes (SELF for instance) tend to have a fewer total number of opcodes. Brunthaler [2009] classifies several interpreted languages according to their abstraction level.

We now describe the interpreted languages that are relevant to this dissertation as they are used in our experimental evaluations.

2.2.2 *Lua*

Lua is a dynamically typed high-level language. It is used as a scripting language in many practical applications such as games engines due to its embeddable nature. It has been developed at PUC-Rio [Ierusalimschy et al., 2007, 2005] since 1993.

Lua 5.1 uses a register virtual machine with a switch-based interpreter. The virtual machine contains thirty-eight opcodes. One of the main features (and initial motivations) of Lua is its portability, and to further that goal the authors programmed it in ‘strict’ ANSI C² (absent of any extensions to the C language, such as the GNU C extensions).

Despite Lua being an interpreted language, it performs well in kernel benchmarks compared to other high-level interpreted languages (Python, Ruby, etc.).³ If greater performance is desired, then there exists a popular JIT compiler called LuaJIT that is able to improve the execution time of many programs [Pall, 2011].

2.2.3 *Python*

Python is a multi-paradigm dynamically typed language [van Rossum, 1997]. Similarly to Lua, it is used in applications as a scripting language, but perhaps more frequently as a standalone solution in various domains. For instance, mature libraries like NumPy and SciPy have led to Python becoming a popular choice for scientific computing applications.

² Technically ISO/IEC 9899:1990, but researchers still colloquially refer to it as ANSI C.

³ Previously performance data was available at the Computer Language Benchmarks Game, however, it has now been removed [CLBG, 2011].

The reference implementation of Python is written in C and is thus usually termed CPython. The implementation of Python 2.6 uses a stack virtual machine with a switch-based interpreter and has 112 opcodes. Python 3 uses a stack virtual machine which contains 100 opcodes. Python also has a number of alternative implementations: PyPy, Jython, IronPython and Unladen Swallow are the main ones. The Unladen Swallow project is now stagnant, however.⁴

2.2.4 *Java*

Java is a high-level statically typed object-oriented programming language. It was originally developed by James Gosling for Sun Microsystems [Gosling et al., 1996]. Java programs are usually compiled to Java bytecode before execution on a Java Virtual Machine (JVM). The Java language was developed with five design goals:

1. Simple, Object Oriented, and Familiar.
2. Robust and Secure.
3. Architecture Neutral and Portable.
4. High Performance.
5. Interpreted, Threaded, and Dynamic.

While Java is a statically-typed language, class loading, however, is dynamic, which complicates the implementation.

There are a number of implementations of the Java Virtual Machine. Some of the most popular in the research community are:

- The Java HotSpot virtual machine, maintained by Oracle (originally by Sun) is considered the reference implementation of Java [Paleczny et al., 2001]. There are two versions of HotSpot depending on usage scenario: ‘Client’ and ‘Server’, the Client uses some interpretation to reduce loading time, whereas the Server optimizes more aggressively.
- The Jikes RVM (originally called Jalapeño) developed by IBM is a meta-circular virtual machine (it itself is implemented in Java) [Alpern et al., 1999]. It is an experimental virtual machine that does not include an interpreter: all code is compiled by a fast compiler and frequently executed

⁴ Unfortunately the project was unable to achieve its goals

code is compiled by a second, optimizing, compiler. Due to it being written in Java, research on optimizing Java is frequently performed in this virtual machine.

- JamVM is a Java virtual machine that was designed to be extremely small compared to the other Java VMs (the striped executable is only 200K on x86 architectures) [Lougher, 2011]. It can be configured in different ways, from a switch interpreter to an inline-threaded interpreter.⁵
- The CACAO virtual machine, developed at the Vienna University of Technology, is another research and pedagogy oriented virtual machine implementation [Krall and Grafl, 1997] .
- The SableVM developed by researchers at McGill university is a ‘robust, clean, easy to maintain and extend, extremely portable, efficient, and specification-compliant Java virtual machine’ [Gagnon and Hendren, 2001]. It is also configurable and supports switched, threaded and inline interpretation.

There are also more specialized virtual machines for Java on mobile devices, such as the Dalvik VM which is used by the Android operating system. Dalvik is quite different from regular Java as it uses its own bytecode format (the Dalvik Executable) that is more suited to constrained systems, and it also is a register-based architecture.

2.3 PIPELINED PROCESSORS

Modern processors contain deep pipelines in order to achieve a high instruction throughput. By separating processing into multiple tasks, the processor can be run at a higher frequency than otherwise possible [Hennessy and Patterson, 2003]. Typically, the process is split into the following steps:

1. Fetch an instruction from memory.
2. Decode that instruction.
3. Fetch the required operands from memory.
4. Execute the instruction.
5. Store any results back to memory.

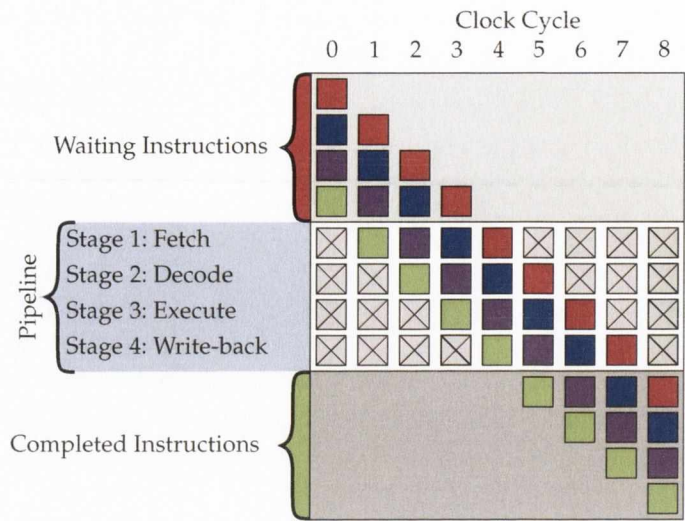


Figure 2.1: Instruction pipeline, showing four basic pipeline stages: fetch, decode, execute and write-back. The pipeline is initially empty, but when it is full it completes one instruction per clock cycle.
This diagram is a derivative work of the original by Colin M.L. Burnett. This derived image is licensed under Creative Commons CC-BY-SA.

However, modern processors contain many more, finer, stages than this. For example, the Intel Core processor has a fifteen stage pipeline.⁶

2.4 SUPERSCALAR

In addition to being pipelined for high throughput, modern processors are also typically *superscalar* to achieve instruction level parallelism. A superscalar processor may complete more than one instruction per clock cycle. These kinds of processors have multiple function units that can compute results in parallel (or out-of-order). In order to do this they require dependency checking logic to ensure consistency in program semantics. Modern multiple-issue processors are able to dispatch eight or more instructions per cycle.

2.5 BRANCH PREDICTION

Branch instructions are used to transfer execution from one point of a program to another. They can be classed into two types: conditional branches and uncon-

⁵ We describe these dispatch mechanisms (and others) in detail in Section 3.3.
⁶ Some of the pipeline stages are due to the latency of signals travelling across the chip, and not just to separate the stages neatly as the diagram illustrates.

ditional branches. Unconditional branches can be further classified into direct branches and indirect branches.

As we explained earlier, modern general purpose processors contain pipelines to improve instruction throughput. An issue however, is that the pipeline may stall due to data hazards, structural hazards or control hazards (branch hazards)⁷. Deep pipelines and higher instruction issue only exacerbate the problem, since a full pipeline flush could result in up to a 20-cycle delay, as in the case of the Intel Pentium 4.

When the processor encounters a branch instruction at the fetch stage (see Figure 2.1), it will not yet know the outcome of the branch, so it cannot know what the next instruction to fetch should be. Typically the destination address of the branch will not be resolved until stage 3 in our diagram. To account for this, the processor could wait until the branch instruction is executed, creating a bubble in the pipeline. This may significantly reduce the instruction throughput of the processor. An approach used by recent processors is to speculatively fetch the next instruction and begin processing from there. Obviously this speculation will not always be correct, so the *speculative execution* may have to be undone when the correct destination is resolved. The choice of the next instruction to speculatively fetch is determined by branch prediction, which predicts the most likely next instruction following the current branch instruction. Branch prediction can be classified by whether it is performed statically (with possible compiler hints) or whether it occurs at execution time on the hardware: static prediction and dynamic prediction respectively.⁸

2.5.1 Static Prediction

Static prediction is the simplest but also the least effective form of branch prediction [Smith, 1981]. The branch instruction is always predicted to be taken or not-taken by the processor. The performance of static prediction is evaluated by McFarling and Hennessy [1986].

One scheme is to predict a branch as taken. This will have an average misprediction rate equal to the untaken branch frequency, which is 34% for the SPEC

⁷ A data hazard occurs when two instructions have a data dependence. For example, an instruction may attempt to read data that refers to the output of the previous instruction, but that instruction has not yet reached the write stage of the pipeline. A structural hazard occurs when two instructions being executed require the same part of the processor. A control hazard occurs in the presence of a branch. The processor does not know the outcome of a branch and consequentially does not know what the next instruction to insert into the pipeline should be.

⁸ A extensive survey of branch prediction techniques is provided by Uht et al. [1997].

programs [Hennessy and Patterson, 2003]. The Sun SuperSparc uses this approach; whereas the opposite scheme is used by the Intel i486, which predicts that all branches will not be taken.

A more effective strategy is to make predictions based on the direction of the branch. For example, predict all backward branches as taken since they are much more likely to be taken due to loops. The HP PA-7x00 is one processor that implements this scheme.

Intel processors also employ static prediction to an extent: on the Pentium III, a branch that has not been seen before (or is 'forgotten' by the prediction hardware) is predicted as taken if it is a backward branch and vice versa if it is a forward branch. This can be seen as a case where dynamic branch prediction is used, but then 'falls back' to static prediction.

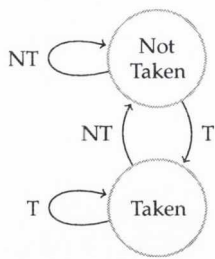
2.5.2 *Dynamic Prediction*

More accurate than static prediction, dynamic branch prediction adapts at run-time to the currently observed program behavior.⁹ Since it requires run-time support to do this, it must be implemented in hardware. There are a few different aspects to dynamic prediction; we begin by explaining dynamic outcome predictors, the most common of which are the one-bit predictor and the two-bit predictor.

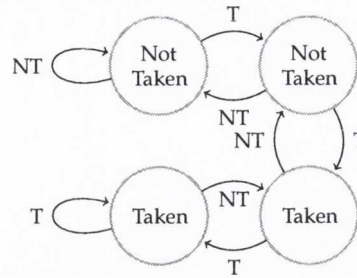
ONE-BIT PREDICTOR The simplest kind of branch predictor for guessing the outcome of a branch is the one-bit predictor. It has only two states, 'taken' and 'not taken', so it predicts that a branch will behave the same way as it did on last execution. Each branch in a program will typically have its own state automaton. This kind of predictor has an accuracy of 77 to 79%. A one-bit predictor is shown in Figure 2.2(a).

TWO-BIT PREDICTOR To improve on the predictability of the one-bit predictor, a two-bit saturating counter predictor, first introduced by Lee and Smith [1984], uses four states; in this way a branch must be mispredicted twice before the predictor predicts differently. Two-bit predictors have accuracies between 78 and 89%. Conditional jumps that have a recurrence such as 'taken every third jump' are not modelled well by the two-bit predictor. A two-bit predictor is shown in Figure 2.2(b).

⁹ Dynamic prediction can be viewed as static prediction that 're-trains' itself



(a) One-bit predictor. The state of this simple automaton will be 1 if a branch is predicted as taken, 0 otherwise.



(b) Two-bit predictor. Two bits are used to encode four states in this automaton. A branch in one of the strongly predicted states must be mispredicted twice before the prediction changes.

Figure 2.2: Saturating counter (state machine) branch outcome predictors. (Edges between states are labelled 'T' for taken and 'NT' for not taken.)

TWO-LEVEL ADAPTIVE PREDICTOR A *two-level adaptive predictor* improves upon the one-bit and two-bit counter predictors by storing the execution history of a branch. It consists of a history register that records the direction of the last n branches and a branch pattern table that is indexed by the history register. Each entry in the branch pattern table is a two-bit saturating counter, of which there will be 2^n . The branch history register is updated by shifting the outcome into it after every resolution. The branch history register may be shared among all branches in a program, which can help prediction by utilizing correlations between branches. The index can also be formed by concatenating or hashing part of the program counter. This kind of predictor can model branches that have recurring patterns that cannot be modelled by the simple counters. The two-level adaptive predictor was first described by Yeh and Patt [1991]. Two-level adaptive predictors are significantly more accurate than the simple automata based predictors, with typical accuracy of around 93% [Uht et al., 1997]. Figure 2.3 shows a simple two-level predictor.

GLOBAL AND LOCAL HISTORY PREDICTION There are two alternate schemes for predicting branches: A global scheme that shares history between all branches, which can take advantage of correlation between different branch instructions, but these can also negatively affect each other. In contrast, a local scheme uses a separate history buffer for every branch instruction. Global branch prediction is used in AMD microprocessors and in Intel Pentium M, Core and Core 2.

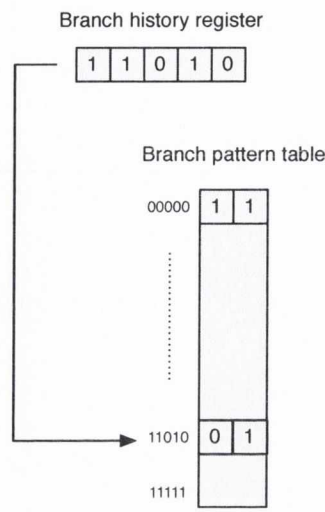


Figure 2.3: A two level adaptive predictor. The branch history register is updated with the recent branch outcomes and is used to index a table of two-bit saturating counters.

McFarling [1993] proposes the idea of combining branch predictors. In other words, keeping track of which predictor is most accurate for each branch so the most accurate predictor will be used. He combines the predictions of a global and a local predictor. The method is implemented in the Alpha 21264. This is also the first paper to propose hashing the branch history with the branch instruction address: a scheme called *gshare*, that uses more bits from the instruction pointer than the earlier *gselect* technique. The benefit of more bits is that it reduces aliasing in the branch counter table. A ‘*gshare*’ predictor then *xors* the global history and branch instruction address, whereas a ‘*gselect*’ predictor concatenates them.

2.5.3 Branch Target Prediction

In addition to the outcome of a conditional branch, we would also like to know the target address of a branch (whether direct or indirect). Consider again the pipeline shown in Figure 2.1. The processor must know from what address to fetch the next instruction by the end of the fetch stage. If the undecoded instruction is a branch then its target needs to be known. The purpose of a *branch target buffer* (BTB) (or *branch target cache*) is to predict the next address to fetch from [Sussenguth, 1971]. The branch target buffer acts as a cache, and can be organized in the usual direct-mapped or set-associative ways. If an entry is found in the BTB for the current instruction pointer (which may not necessarily

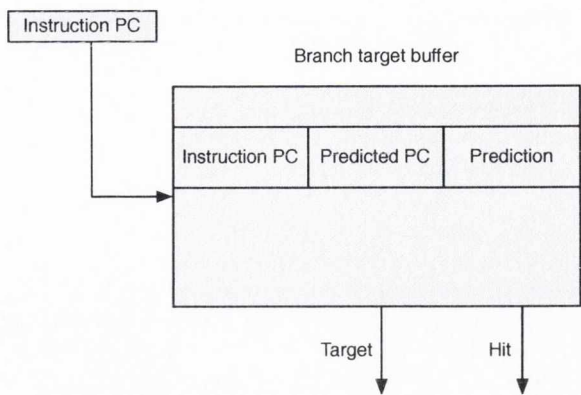


Figure 2.4: A branch target buffer. Some bits from the instruction pointer are used to index the cache-like structure. If there is an entry and it is predicted as taken then the processor should begin fetching at the predicted target.

be a branch) and it is correct, then there will be no branch penalty incurred. Intel x86 processors since the Pentium II have featured a branch target buffer.

Branch target prediction is concomitant to branch outcome prediction — both techniques complement each other. An entry in the BTB may have a two-bit saturating counter (to save space, instead of storing a strongly not-taken prediction, there will simply not be an entry in the BTB for this branch) to make predictions. Alternatively, the target buffer and prediction buffer may be separate; as is the case for PowerPC processors.

2.5.4 Indirect Branch Prediction

Indirect branch (and call) instructions are used for a few different reasons: 1) As one potential method of implementing a switch statement (using a jump table), 2) to implement function calls where function pointers are used by the programmer and their destinations cannot be resolved statically, and 3) to implement virtual function calls in object-oriented languages such as C++. Unfortunately, indirect branches are much more trickier to predict than regular conditional branches because the indirect branch can have several potential targets which are liable to change from one execution to the next.

As a consequence of Moore’s law, there is now an abundance of transistors on CPUs. Unfortunately, due to heat constraints, the clock frequency of processors has effectively plateaued. Recently, CPU manufacturers have started to use the extra transistors to improve performance in other ways. One of the ways has been to improve the branch predictor unit (thereby potentially increasing the

instructions-per-cycle of the processor). One of the more recent improvements to branch predictor hardware is support for the prediction of indirect branches.

The simplest type of indirect branch predictor simply uses the BTB which is used to predict the (unchanging) target of conditional and unconditional direct branches. When used with indirect branches, the BTB simply predicts that the indirect branch will jump to the same target as last time it was executed. This works well for branches that usually jump to the same target, which are often known as *monomorphic indirect branches*. However, in the case of indirect branches which jump to many different targets (*polymorphic indirect branches*) the prediction accuracy is usually poor. This scheme is employed by the Alpha 21264, Itanium 2, Pentium and Athlon processors.

Driesen and Hölzle [1997] pointed towards the potential for better indirect branch prediction. Addressing the problems with the BTB approach, researchers have adapted the idea of two-level branch predictors to indirect branches [Driesen and Hölzle, 1998a, Chang et al., 1997]. Rather than having only a single branch prediction entry per branch, a two-level predictor uses both the address of the branch itself and a history of recent branch outcomes to index a table and find a prediction. In the case of indirect branches, the history of recent branches usually consists of some bits selected from the addresses of recent indirect branch targets, and combined together into a single history register. By capturing information about the outcome of recent branches, a two-level branch predictor is able to exploit correlations between different indirect branches, or between multiple executions of the same branch.

Driesen and Hölzle [1998b] describe a hybrid predictor architecture. They show that cascaded prediction obtains similar prediction rates to a two-level predictor for path-based indirect branch predictors, yet incurring only about one quarter of the cost.

One of the largest sources of indirect branches in programs are procedure returns. To account for these, processors, such as Intel processors, use a *return stack buffer*. This buffer stores the return addresses of subroutine calls. By using this buffer, the processor can pipeline instructions beyond a return from a subroutine.

Only in recent years have two-level indirect branch predictors started to appear in real processors. The earliest example we are aware of is the Intel Pentium M [Gochman et al., 2003] which was introduced in 2003. More recent examples include the AMD Barcelona processor [Advanced Micro Devices, 2008], and various Intel Core 2 processors (based on the Core microarchitecture).

2.5.5 *Reverse Engineering Branch Predictors*

The processor manufacturing business is extremely competitive and CPU manufacturers aim for a market advantage by keeping many of the design details of the processor secret and providing only high-level information on processor advances. The complexity of modern processor branch predictors has led some researchers to attempt to reverse engineer the predictor structures to understand their design. They do this by forming hypotheses, then crafting programs that exercise the predictor structures in particular ways that will confirm or refute those hypotheses. In effect, applying the scientific method to understand behavior. This work is aided by the addition of performance monitoring counters to recent processors, which usually include counters for various events related to branch prediction.

The first effort that we are aware of to reverse engineer predictors was by Milenkovic et al. [2002, 2004]. In this work they first propose experiments for discovering branch predictor details. This research was performed on the Intel Pentium III and Pentium 4 predictors (the P6 and NetBurst microarchitectures).

Uzelac and Milenkovic [2009] present experiments to reverse engineer the branch predictor structures on modern processors. Their tests are able to determine size, organization, and operation of target predictors and the size, organization, and operation of hybrid outcome predictors. They focused on reverse-engineering the structures in the Intel Pentium M processor. They validate their findings by building a Pin model and confirming its results correlate with the real hardware performance counters.

This work is elaborated on extensively in Vladimir Uzelac's Master's thesis [Uzelac, 2008]. He provides microbenchmarks to reverse engineer the branch target buffer, the loop predictor, the indirect predictor and the global predictor.

We make use of their experimental results concerning the indirect predictor in this dissertation: in Chapter 5 we use the information they provide on the Intel Pentium M indirect branch predictor to improve indirect branch prediction on the Intel Core microprocessor.

Rahman et al. [2009] describe a method to reverse-engineer microarchitectural features structures using object code re-ordering. They evaluate a model of the branch predictor from the Intel Core 2. Unfortunately they do not reverse-engineer the indirect branch predictor in this processor, but it is assumed that the Core microarchitecture includes the same indirect branch predictor as the Pentium M [Kim et al., 2009].

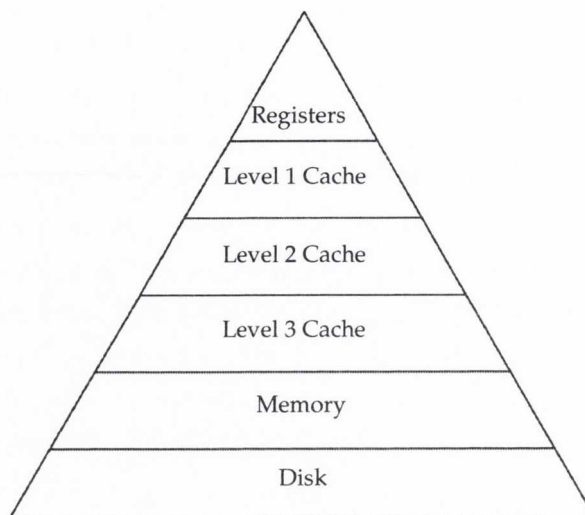


Figure 2.5: Memory hierarchy.

2.5.6 Limits of Branch Prediction

Branch prediction is ultimately limited by the amount of regularity in a program: many non-trivial programs will have stages that are random, that is, they will contain lots of information in the Shannon information-theoretic sense.

Information entropy quantitatively expresses the regularity of an information stream such as a program. Yokota et al. [2007] apply information theory to branch prediction. They introduce *Branch History Entropy*, which represents the amount regularity in program behavior and is independent of predictors. They also introduce *Table Entry Entropy* which provides the upper bound on the performance of branch prediction.

Federovsky et al. [1998] attempt to reduce branch prediction to the more studied problem of data compression. They describe prediction algorithms based on two universal data compression algorithms: prediction by partial matching (PPM) and context tree weighting. In a similar vain, but for indirect branches, Kalamatianos and Kaeli [1999] show that correlation with previous branches using PPM can improve accuracy.

2.6 INSTRUCTION CACHE

Memory in a computer system is typically slow compared to processor speed, especially with the technologies used to realize larger storage volumes. Fortunately, much data (and code) tends to have a temporal locality: if it was used

recently then it is likely to be used again soon, and also a spatial locality: if a memory location was used recently then it is likely nearby locations will be referenced soon. Based on these observations, computer systems are usually designed with a memory hierarchy. Figure 2.5 shows the typical memory hierarchy of a modern computer system. Modern processors, such as the Intel Core 2 further separate the L1 cache into data and instruction cache.

Caches are a small memory that may or may not contain data at a memory address. They are referenced with the memory address, which will typically be hashed. There are two main organizations of cache memory: with a direct-mapped cache, a memory address can only map to one entry in the cache; with a fully associative cache, then a memory address can map to any entry in the cache.

The complexity of a fully associative cache means it is too expensive to realize in hardware. A set-associative cache is a compromise between fully-associative and the simpler direct-mapped scheme. This type of cache organizes items into K sets. A 2-way set associative cache for instance means that a particular piece of data may be stored at two possible locations. Different replacement policies can be used to decide which old entry to evict from the cache. For instance, least recently used (LRU) will replace the entry that has not been used in the longest amount of time. This is just a heuristic and may not be the optimum choice of entry to evict.

2.7 DOMINANCE FRONTIERS

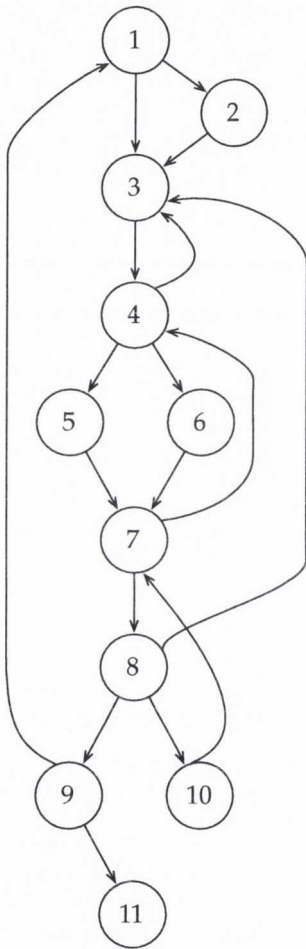
Dominance relations are one of the earliest techniques developed for structured compiler optimizations [Aho et al., 2006]. A node i is said to dominate a node j if every path from the start node to j must go through i . Representing dominators visually, Figure 2.6(a) shows a simple example control flow graph, and Figure 2.6(b) shows the corresponding dominator tree for that flow graph.

The earliest algorithm for computing dominators was provided by Lowry and Medlock [1969]. To calculate the dominators, the following dataflow equations can be used:

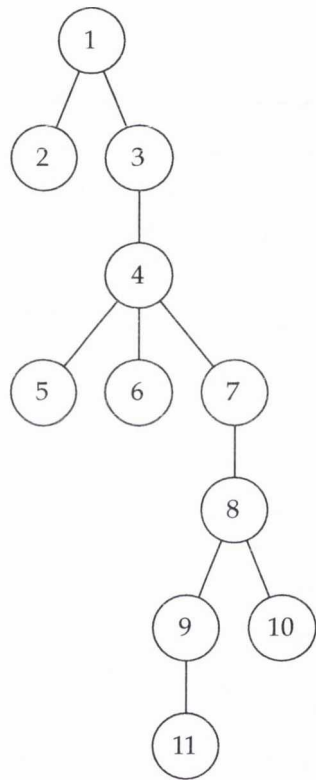
The start node dominates itself: $\text{Dom}(n_o) = \{n_o\}$

$$\text{Dom}(n) = \left(\bigcap_{p \in \text{preds}(n)} \text{Dom}(p) \right) \cup \{n\}$$

The dominance frontier is an important idea based on dominators that is used by many compiler optimizations; for instance, static single assignment (SSA) form is computed using dominance frontiers. The dominance frontier for a node b is defined formally by Cytron et al. [1991] as:



(a) Control flow graph.



(b) Dominator tree.

Figure 2.6: A simple control flow graph and its corresponding dominator tree.

the set of all nodes y such that b dominates a predecessor of y , but does not strictly dominate y .¹⁰

Cooper and Torczon [2003] provide an efficient algorithm for computing the dominance frontier for every node in a control flow graph, which we repeat in Algorithm 2.1.¹¹ The main reason for dominance frontiers is the efficient placement of ϕ functions in SSA form; but later in Chapter 6 we develop our own use for them.

Algorithm 2.1 Computing dominance frontiers.

```

1: Input: Control flow graph, IDOM maps.
2: Output: DF sets for each node in the control flow graph.
3: for all nodes  $n$  do
4:    $DF(n) \leftarrow \emptyset$ 
5: end for
6: return DF
7: for all nodes  $n$  do
8:   if  $n$  has multiple predecessors then
9:     for all predecessors  $p$  of  $n$  do
10:       $runner \leftarrow p$ 
11:      while  $runner \neq IDOM[n]$  do
12:         $DF(runner) \leftarrow DF(runner) \cup n$ 
13:         $runner \leftarrow IDOM(runner)$ 
14:      end while
15:    end for
16:  end if
17: end for

```

2.8 CONCLUSIONS

Interpreted languages are more relevant than ever, so finding good techniques to aid their efficient implementation is clearly a worthwhile goal. In this chapter, we described some of the background topics necessary to apprehend and to motivate the research presented in this dissertation. We described interpreters, pipelined processors, branch prediction (outcome predictors, target buffers, indirect prediction and reverse engineering), the instruction cache and dominance.

¹⁰ A node i strictly dominates a node j if i dominates j and i does not equal j .

¹¹ $IDOM[n]$ refers to the node that is the immediate dominator of n . The immediate dominator of a node is its parent in the dominator tree.

In the next chapter we provide a literature review of the existing research that is relevant to our thesis.

LITERATURE REVIEW

3.1 INTRODUCTION

Our thesis is that indirect branch prediction for programs can be improved through the use of compiler techniques. To support that claim, this dissertation proposes and describes a series of compiler techniques to improve indirect branch prediction for ordinary programs and for interpreters. This chapter serves as a literature survey in order to acquaint the reader with similar previous work on this topic.

This chapter is divided into several sections. We first survey previous compiler techniques to improve branch prediction, discussing semi-static prediction techniques, code placement strategies to improve prediction accuracy, and branch predication. Then we describe interpreter dispatch optimization strategies which attempt to reduce the number of indirect branches that an interpreter must execute (or improve their predictability). Following that, we report related work on the relationship between interpreter performance and indirect branch prediction hardware. We close the chapter with an account of various ideas from object-oriented language implementation that attempt to reduce the burden of virtual function dispatch.

3.2 COMPILER TECHNIQUES TO IMPROVE BRANCH PREDICTION

3.2.1 *Semi-Static Techniques*

In Section 2.5.1 we introduced static branch prediction. On its own, static prediction is a very ineffective technique. Fortunately, there are a number of compiler techniques that can be used to help the static prediction of branches. Semi-static branch prediction techniques rely on the compiler to perform analysis of the program and use instructions that will inform the static hardware predictor of what outcome to predict.¹ Some PowerPC processors for instance use semi-static prediction: the compiler can insert a ‘hint-bit’ into the generated code

¹ While more powerful than static techniques, semi-static techniques are only effective for ‘stubborn’ branches, which regularly branch in the direction they were predicted to offline.

to specify the statically predicted branch direction. This compiler support for static prediction can be improved through the use of profiling; though profiling makes the software development process more cumbersome.

Profile-based techniques for branch prediction can be cumbersome and time-consuming. Trying to alleviate this problem, Ball and Larus [1993] describe a series of heuristics that a compiler can use when performing semi-static branch prediction. They term their prediction strategies a *program-based* branch predictor. Their results are a factor of two worse than profile based techniques, but they are useful.

To improve the accuracy of semi-static branch prediction, Krall [1994] describes a code replication technique that combines different correlation strategies. An advantage of increasing the accuracy is that other compiler optimizations such as code motion can benefit.

Many static branch prediction techniques utilize either simple heuristics or profile information. *Value range propagation* [Patterson, 1995] is an approach to static prediction that tracks weighted value range of variables through a program, similar to constant propagation. The statically determined value range of a variable is then used to make branch prediction decisions at compile time. Value range propagation subsumes constant and copy propagation. The authors provide experimental results showing that their method performs better than heuristic approaches; however, when value propagation cannot make an inference for variables then it has to fall back to using heuristics.

A compiler-based technique for dynamic branch prediction is presented by Mahlke and Natarajan [1996]. It uses profile feedback to insert a prediction function which captures information about the current context that is then used to make predictions. Hardware support is required in this case: the prediction functions must be allowed to write their predictions to a *branch predict register* that the processor then uses to make predictions. They evaluate this technique on their experimental *PlayDoh* architecture and achieve prediction rates that are better than BTBs with 2-bit counters and comparable to two-level hardware branch predictors.

Since users are often averse to profiling, static heuristics remain important. Deitrich et al. [1998] explore the reasons why these heuristics are effective. They also utilize source-level information in their heuristics; making the work suitable for implementation in a compiler (whereas previous work was based on program traces). They introduce new heuristics and build upon existing ones.

3.2.2 Reducing Branch Interference

In real hardware there are a limited number of 2-bit outcome predictors, and these must be shared among conditional branches. Applying the pigeonhole principle² in this scenario leads to the problem of branch *aliasing*³. Attempting to reduce these collisions, Chen and King [1999] adjust branch addresses at compile (or link) time. Their technique adds NOP instructions to perform the address adjustment; therefore enabling the control of the mapping between conditional branches and the outcome predictors. They present a constrained and a relaxed method, where the constrained method will only insert NOPs following an unconditional branch and relaxed will insert NOPs where they may get executed by the processor. They also describe *branch classification*, which maps branches with the same history pattern to the same counter (therefore utilizing constructive interference). They find that branch classification has the highest improvement of all their techniques.

Jiménez [2005] build upon that work. They describe a feedback-directed code placement technique that also inserts NOP instructions to reduce destructive interference between conditional branches called *pattern history table partitioning*. Their results show that mispredictions are reduced by up to 22% for SPEC CPU benchmarks on an Intel Pentium 4 processor.

Most branch prediction studies have relied on SPEC benchmarks for evaluation. However, these benchmarks typically exercise a limited amount of code and may not be accurate representations of larger applications. Sechrest et al. [1996] look at larger programs and simulate branch prediction schemes with them. Their results show that for resource limited predictors, destructive interference significantly affects prediction accuracy; and that even local history based predictors can suffer problems with aliasing in the branch history table. The main point of this work is that aliasing effects can override any possible branch correlation benefits.

3.2.3 Code Placement for Branches

There has been much research on code rearrangement to improve cache behavior. Related to branches, Calder and Grunwald [1994a] present a profile-

² The pigeonhole principle states that if m pigeons are put into m pigeonholes, there is an empty hole if and only if there is a hole with more than one pigeon. Stated differently, if there are n pigeons and m pigeonholes and $n > m$, then at least one pigeonhole must contain more than one item [Graham et al., 1994].

³ Branch aliasing occurs when more than one branch uses a single entry in a branch history table.

based technique that 'aligns' branches. This alignment minimizes the number of taken branches, therefore ensuring frequently executed code is closer together, thereby improving cache performance.

Uh and Whalley [1999] describe a transformation that coalesces conditional branches into indirect jumps. Their results show that this transformation to indirect jumps can lead to fewer instructions executed and improved prediction. In order to avoid pipeline flushes, they perform a compiler analysis to determine the most likely target of the branch, and place it after the branch instruction. In this way, even if the BTB holds no record for the branch, the processor may still fetch and dispatch the correct target without penalty.

Conditional branches are expensive and should be minimized where possible. Yang et al. [2002] describe a code transformation that detects sequences of conditional branches that can be re-ordered; then, based on profiling information (and estimates of the cost of the branches), they select the ordering that results in the execution of the fewest conditional branches in the average case. Their transformation can even insert new conditional branches into the program. They report that their technique can result in 8% fewer instructions and 13% of branches executed on the SPARC architecture.

Young et al. [1995] cause branches that would have been mispredicted to be correctly predicted. They use feedback-directed techniques to induce constructive interference between branches. Their work re-orders branches so that the common case will be the fallthrough path of the branch; this is important on architectures where a taken branch causes a delay.

3.2.4 *Branch Predication*

One technique to reduce branch overhead is to attempt to eliminate the use of branch instructions entirely. This is the case with predicated instructions [Allen et al., 1983]. A predicated instruction is an instruction that will only be executed by the processor if certain condition codes are set (or unset). In the case where the predicate is false, the processor may have performed some computation already, but the results will not be committed. Some processor architectures such as the Intel Itanium (IA-64) feature branch predication instructions and the ARM architecture enables most instructions to be predicated.

Usually, branch predication requires the instruction set architecture (ISA) to be modified. Dynamic predication [Klauser et al., 1998] is a form of multi-path execution that is designed for architectures without support for predication in their ISA. Here, the decode stage detects a sequence and passes predicated

instructions to a dynamically scheduled execution unit. Originally it only supported simple branch hammers (a short forward branch that spans a few instructions in the form of an if-then or if-then-else construct), concurrently executing both paths of the branch. Researchers then extended the idea to a larger class of control-flow graphs [Kim et al., 2007b].

Traditionally, predication (both ISA supported and dynamic) was limited to conditional branches, as predication assumes two possible destinations of a branch. Joao et al. [2008b,a] describe *dynamic indirect jump predication (DIP)*. With DIP the processor executes the most likely targets until they merge. This way, if one of the targets was correct, a pipeline flush is avoided. A question is whether the compiler or the hardware should determine which target paths to predicate; in this case, in contrast with conditional branches, not all targets may be known at compile time. They therefore evaluate both methodologies, using profiling in the case where the compiler decides. They evaluate their idea on the Java DaCapo benchmark suite, showing an improvement of 37.8% over BTB based prediction of indirect branches. They also evaluate their idea on some of the SPEC 2000 and SPEC 2006 benchmarks, and see a performance increase of 26% on average, even though those benchmarks are not as indirect branch intensive as Java.

3.2.5 Target Predictability for High-level Languages

Li et al. [2005] improve the target predictability of Java, particularly for virtual methods. They propose a *rehashable branch target buffer* that dynamically adapts branch target storage for polymorphic branches. Their method identifies polymorphic branches that cause repeated mispredictions and ‘rehashing’ their multiple targets into unified target storage. An advantage of their proposal is that it does not share resources between direct and indirect branch prediction hardware. Their results show that for Java programs in interpreter mode, this rehashable BTB approach can eliminate 61% of the mispredictions suffered by a traditional BTB, and 46% for Java programs in JIT mode.

3.2.6 Measurement Bias

In this dissertation we describe cases where strategic code placement can improve performance. A different way of viewing this is to see it as affecting measurement bias. The effect of linking order (among others) as a source of

measurement bias has been described by Mytkowicz et al. [2009]. They attribute measurement bias due to link order as an alignment issue, and using the m5 simulator they see instruction cache miss variances when the link order is changed. They suggest branch prediction may also be a cause of measurement bias in general.

Based on the understanding that the hardware can be too difficult to model to achieve maximum performance, Knights et al. [2009] propose *blind optimization*. This approach generates many variants of a program, attempting to find the best performing version. For one benchmark they observe a 12% speedup, and 1.58% on average.

3.3 IMPROVING INTERPRETER PERFORMANCE

3.3.1 Introduction

In Section 2.2 we provided a brief introduction to interpreters. Ever since Pascal's p-code, there has been work to improve performance of virtual machines. In this section we first describe the established interpreter dispatch methods. Following that, we describe the previous research on the interaction between interpreters and indirect branches.

Interpreters are most likely the class of program that uses the highest ratio of indirect branches to other instructions. This is because run-time dispatching is an operation that needs to be implemented using indirect branches. The actual usage of indirect branches, however, varies from interpreter to interpreter. Typically an interpreter that performs little work per opcode will spend a larger percentage of execution time in dispatch, so these kinds of interpreters will use more indirect branches since they need to dispatch more.

Bytecoded interpreters can be implemented using different techniques, some of these techniques differ only by the dispatch mechanism (how the next opcode implementation is reached from the current one) [Kogge, 1982].

Performance of interpreters was largely ignored in research literature until the work of Romer et al. [1996]. In their paper they study interpreter performance from the software and hardware perspectives, looking at four different interpreters: MIPS (a MIPS instruction-level emulator), Java, Perl, and Tcl. They show that performance depends on native run-time libraries. They claim that interpreters are similar to other types of program in their use of processor resources (a claim which is contested by later research [Ertl and Gregg, 2001]). Their conclusion is that interpreters do not warrant special hardware support,

and that efforts should instead focus on compilation and run-time code generation for the interpreted languages. This has essentially come to pass, except for the introduction of indirect branch prediction support in hardware, which very likely was motivated by interpreters.⁴

The main steps in interpreting a VM instruction are to fetch the opcode, fetch the operands, and *dispatch* to the real machine code routine that implements the opcode. This last step is typically the one that dominates the performance of interpreters, at least for those with relatively simple VM instructions. In the rest of this section we describe the various techniques used to dispatch opcodes in interpreters.

3.3.2 *Switch*

The simplest and most common kind of interpreter is the ‘switch’ interpreter. A switch statement is used to select which opcode handler to execute for the current opcode. This technique has the major advantage of being a portable ANSI C method.

Listing 3.1 shows how a simple interpreter can be implemented with a switch statement. The cost of switch dispatch is six instructions: a load instruction to load the opcode number from the table, two instructions for table range check, the computed indirect jump through the table, the instruction to update the program counter and a direct jump back to the beginning of the interpreter loop. (Potentially one more if the computed indirect jump cannot be done in one instruction)

```
while(1) {
    switch(ip->op) {
        case OP_LDI:
            *sp++=ip->arg;
            ++ip;
            break;
            /* ... */
    }
}
```

Listing 3.1: Typical ANSI C source code for a switch interpreter.

⁴ Interpreters are one of the most intensive users of indirect branches, and the introduction of indirect branch prediction in hardware coincides with the increase in popularity of interpreted languages.

3.3.3 Token-Threading

Token-threading is a technique to improve the performance of an interpreter. It involves a table of opcodes that correspond to code addresses. Each opcode implementation has its own dispatch point, so a direct jump is removed compared to switch dispatch, and as we explain later, performance is improved due to better branch prediction. Another advantage is that the token-threaded interpreter does not have a bounds-check that the compiler would generate for a switch statement (this bounds-check is not always needed, as the compiler may be able to determine that no value larger than the size of the opcode table will ever be used as an index into it).

Unfortunately, in the C language it must be implemented by taking the address of a label, which is not part of the ANSI C standard. Some compilers, such as GCC (GNU C), provide this extension, and call it *labels as values*. Using this construct produces code that is not portable.

Listing 3.2 shows how a token-threaded interpreter can be implemented by using the labels-as-values extension. The cost of token-threaded dispatch is three instructions on the x86 architecture: an instruction to update the program counter, a load instruction to load the opcode number from the table and a computed indirect jump through the table. (Potentially one more instruction if the computed indirect jump cannot be performed in one instruction.)

```
static void *optable[]={
    [OP_LDI]=&LBL_OP_LDI, ...};

goto *optable[ip->op];

LBL_OP_LDI:
    *sp+=ip->arg;
    ++ip;
    goto *optable[ip->op];
/* ... */
```

Listing 3.2: C source code for a token-threaded interpreter using the (non-ANSI C) *labels-as-values* extension.

3.3.4 Direct-Threading

An early technique, direct threading consists of a bytecode where the codes are actually pointers to the opcode handling routines. The dispatch in such an

interpreter is a matter of loading the next pointer from the instruction list and jumping to it [Bell, 1973].

One way of implementing direct threading is to build the table by bootstrapping from the token-threaded implementation. Code to perform this dynamic translation is sketched in Listing 3.3.

```
//table from token-threaded implementation
void *bytecode_table[]={
    [BYTECODE_ADD]=&&op_add, [BYTECODE_MUL]=&&op_mul,
    /* ... */
}

instr *ip;
void **opcode_table = ...;
for (ip=instructions_start;;ip++) {
    *opcode_table++ = bytecode_table[ip->op];
    if (ip->op == OP_EOF)
        break;
}
```

Listing 3.3: Code to convert a bytecode into direct threaded code dynamically.

The cost of direct-threaded-dispatch is three instructions: a load instruction to load the opcode address from the program array, an instruction to update the program counter and an instruction to jump to the next opcode implementation.

3.3.5 Indirect-Threading

Somewhat similar to the idea of token-threading is indirect-threading, which is first described by Dewar [1975]. Here, however, an opcode is a pointer to a struct (the C structured data type). This struct in turn contains a pointer to the opcode implementation and also the immediate values or operands for that opcode. Like token-threading, indirect-threading involves an extra level of indirection compared to direct threading. An indirect-threaded program array consists of a linear list of addresses of words that contain the address of the opcode handler.

3.3.6 Call-Threading

Call threading consists of using function pointers (in a language like C) to call the appropriate opcode handler function. This involves the use of an indirect

call instruction, which is similar to using an indirect jump but with the added cost of a return instruction. The dispatch routine is shared, and the opcode implementations return to it. With call threading, shared data, like the instruction pointer and stack pointer must be declared as global variables.

Call-threading has a dispatch overhead of four instructions: a load instruction to load the opcode address (function pointer) from the program array, an indirect call instruction to call the function, an instruction to update the program counter and a return instruction.

One optimization to call-threading is to use tail-calls. This depends on compiler support for tail-call optimization, which is available in GCC for instance. Each function that implements an opcode will end with a tail-call to the next opcode handler, like this: `(*program[pc++]())`;

3.3.7 Subroutine/Context-Threading

Subroutine threading involves the generation of machine code to call opcode handler routines. Machine code calls to the opcode handlers are copied into a native code buffer where execution then proceeds to. For example, the following block of native assembly instructions:

```
call &GETARG
call &MUL
call &RETURN
```

If the program counter update is removed from the opcode handlers, then subroutine-threading is highly efficient, with a dispatch penalty of just two instructions per opcode: the call and return.

More recently, this technique has been used by Berndt et al. [2005] for a type of threading they term *context threading*. Each opcode is implemented as a function, and therefore the virtual program counter and the hardware program counter share the same contextual information, improving the predictability of branch instructions.

3.3.8 Inline-Threading

Inline-threading is a more advanced technique where the code to implement a series of opcodes is copied into a buffer, and the buffer (which is considered a macro opcode) is then executed [Piumarta and Riccardi, 1998]. This removes all

dispatching overhead, and improves performance significantly. If the program counter update is removed from the opcode handlers, then inline-threading effectively has no dispatch cost, as no instructions will be executed that are not part of an opcode implementation. Inline threading can be considered as a very basic form of JIT compilation.

Piumarta and Riccardi [1998] limit inline-threading to basic blocks. This is for a few reasons; one reason is that function calls from within an opcode are not allowed on architectures that use relative branch addressing, as the correct jump offset will be unknown. The technique works better for virtual machines that perform little work per opcode, since in these systems the relative cost of dispatch to useful program work is highest and may even exceed it.

3.3.9 *Comparing Threading Techniques*

The choice of optimal dispatch technique depends on the architectural characteristics of a particular system. Ertl [2011] compares various dispatch techniques for the Forth stack language on different hardware architectures. Call threading is the slowest dispatch technique on every architecture examined.

3.3.10 *Just-In-Time Compilation*

Just-in-time (JIT) compilation⁵ is an established technique to implement virtual machines when high performance is an important concern. Typically running code will be profiled and ‘hot’ regions discovered; optimized native code that implements these hot regions will then be emitted into a code buffer and execution will jump to it, when the native code exits it will return to the interpreter to resume execution and profiling. Aycock [2003] provides a substantial history of just-in-time research up to 2003.

Trace compilation is a technique to improve the performance of software by optimizing for the common case. A trace is a ‘hot’ code path which includes function calls. Trace compilation profiles and records program traces and applies more aggressive optimizations to them than could be applied to methods. One of the earliest uses of trace compilation was as a replacement for profile-guided optimization in the Dynamo project [Bala et al., 2000] which optimizes native machine code at run-time. Recently there has been interest in trace compilation for dynamic languages [Gal et al., 2009, Chang et al., 2009]. This in-

⁵ Also known as dynamic compilation.

terest has grown out of the desire to optimize languages used primarily on the World-Wide Web. In a dynamically typed language, a trace will record variables as having a particular type so type-checks can be omitted from the optimized native code.

The main downside to just-in-time compilation is the lack of portability. Some techniques have been proposed to deal with this problem, such as generating C code and using the platform's C compiler to generate the required machine code [Williams et al., 2010].

3.4 INTERPRETERS AND INDIRECT BRANCH PREDICTION

In Chapter 2 we highlighted the importance of branch prediction, and we also described interpreted languages. Interpreters use a significant amount of branch instructions: using indirect branches for opcode dispatch and conditional branches for operation type checks (in dynamically typed languages).

Ertl and Gregg [2001] were the first to investigate the interaction between indirect branch prediction and interpreter performance, showing that contrary to the then established notion that interpreters behaved like general purpose integer programs (the work of Romer et al. [1996] as we mentioned earlier), efficient interpreters have different characteristics. They show that for current branch predictors, threaded code interpreters cause fewer mispredictions, and are almost twice as fast as switch based interpreters on the generation of processors of the time. Their journal paper builds upon that work and provides experimental results for different interpreters and branch prediction schemes [Ertl and Gregg, 2003b].

The authors go on to investigate strategies to improve indirect branch prediction in interpreters. They propose techniques to improve the indirect branch prediction rate of interpreters. They describe *superinstructions* (new VM instructions consisting of a sequence of existing instructions) and VM instruction replication. These techniques are measured on the existing processors of the time, which use simple branch target buffers to predict indirect branches [Ertl and Gregg, 2003a]. Casey et al. [2007] elaborate on this work, providing an implementation and experimental results for a Java VM. In this work they show the techniques to be generally applicable. They also solve JVM specific problems, such as issues regarding quick instructions⁶.

⁶ A quick instruction acts as an inline cache that stores things that can only be resolved at run-time, such as the location within objects of the text name for a field. The original 'quickable' instruction is replaced by the quick instruction.

Writing a high performance interpreter can be a challenging task as there are a range of optimizations that an implementer can apply: stack caching, superinstructions, and threaded-code are but a few. *Vmgen* is a tool that generates virtual machine interpreters in C from a description of the virtual machine opcodes [Ertl et al., 2002]. Among other time-saving tasks, it automatically generates code to dispatch the next VM instruction and to profile frequently executing instruction sequences. *Vmgen* can generate interpreters for Forth and Java that are faster than existing interpreters. Another tool for generating interpreters called *Tiger* is described by Casey et al. [2005] and detailed in Kevin Casey's Ph.D. thesis [Casey, 2005]. *Tiger* builds upon the ideas from *Vmgen*, adding support for interpreter optimizations like instruction specialization and replication.

3.5 EFFICIENT POLYMORPHIC CALLS

In this thesis we describe different compiler optimizations to improve the prediction rates of indirect branches. Virtual function calls in object-oriented languages are one important programming construct that are usually realized in hardware using indirect branches.

An analysis of the different kinds of dispatch available to language designers is provided by Rose [1988]. The analysis is performed in the context of the Common Lisp Object System. The author suggests different table-based algorithms to perform dispatch.

Calder and Grunwald [1994b] analyzed the overhead of indirect function calls in C++ programs. They report that accurate indirect branch prediction is usually more important than call prediction. They describe and evaluate compiler optimizations to improve the performance of indirect function calls. For instance, profiling can be used to eliminate indirect function calls, replacing it with a type check that will directly call the common case. For instance the following code:

```
if (typeof(shape) == Circle)
    shape->Circle::draw();
else
    shape->draw();
```

Depending on the architecture, this approach can reduce the number of instructions necessary, and on architectures without indirect branch prediction then it can substantially improve predictability. Grove et al. [1995] study the dynamic profiles from C++ and *Cecil*, showing that profile guided receiver prediction

is effective in their Cecil programs. They provide a model to describe the prediction of profile data and show that call chains lead to more stable profiles. Similar work to optimize dynamically dispatched calls was described by Hölzle and Ungar [1994] in the context of the SELF language, a dynamic object-oriented language. Ishizaki et al. [2000] describe these techniques as *devirtualization* and further class them into guarded devirtualization, as in the example we have shown, and direct devirtualization, which converts an indirect call into a direct call if the compiler can determine that there is only ever one possible target (which is not possible for languages which allow dynamic code loading). They introduce a new method called *direct devirtualization with the code patching* that uses code patching to overcome the problem in languages with code loading. They evaluate their work in the context of Java.

In Section 2.5.4 we described how processors predict indirect branches. A downside is that the indirect prediction can require a significant amount of extra hardware which is sometimes not possible on resource constrained devices. To tackle this, researchers have taken the idea of converting indirect branches to conditional branches and implemented it in hardware, therefore effectively enabling the conditional predictor to predict indirect branches [Kim et al., 2007a]. The authors call this technique *virtual program counter (VPC) prediction* as it treats an indirect branch as multiple ‘virtual’ conditional branches. Some of the advantages of this scheme over the software approaches we described are that it applies to any indirect branches and it is able to adapt dynamically to observed program behavior. VPC prediction is compared with a BTB and found to increase average performance by 26.7%.

Another idea, described by [Lea, 1990], to avoid the dispatch cost in C++ is to use *customization* (and idea adapted from SELF [Chambers and Ungar, 1989]); here the compiler will generate different versions of a function which are specialized to its input types, and can then direct call or inline the virtual function calls.

Driesen and Hölzle [1996] measure the overhead of virtual function calls in C++, showing that the C++ programs they measure spend a median of 5.2% of their time in dispatch code.

In his Ph.D. thesis, Driesen [1999] describes both hardware and software techniques to optimize polymorphic calls. The hardware techniques are methods to improve indirect branch predictors and the software techniques are ideas such as the polymorphic inline cache.

Part of the reason dynamically typed languages run slower than statically typed languages is that some object-oriented dynamically typed languages use

```

values = [1, "a", 2, "b", 3, "c", 4, "d"];
for (v in values) {
    print(v.toString());
}

```

Listing 3.4: Worst case scenario for inline caching.

message passing on objects extensively. This dynamic message call is more expensive than a static function call since the correct target must be determined at run-time according to the type of the object the message is called on. One technique to alleviate the cost is to employ a *lookup cache*. This cache will map object receiver and message name pairs to their target methods, making the call faster. A further improvement called *inline caching* is to exploit locality of usage—types of receivers at a call site rarely vary—and store the method address directly at the call site, along with a check to ensure that the type of the object has not changed.⁷

With inline caching, which is used in all Smalltalk implementations, the target method at a call site is rebound if the receiving object changes between calls. This works well when a call site is *monomorphic*, however, in some kinds of programs, Listing 3.4 for example, the object does change; these call sites are termed *polymorphic*. *Polymorphic inline caching* [Hölzle et al., 1991] is an extension of inline caching for polymorphic call sites. When a miss is encountered, a stub function is created and the call site is updated to call this stub. The stub checks the receiving object type against those most recently encountered and calls the correct target method. A polymorphic inline cache (PIC) improves the performance of the Richards benchmark in the SELF language by 52%.

3.6 CONCLUSIONS

In this chapter we surveyed the research literature that is relevant to this dissertation. We described previous compiler techniques to improve branch prediction, including semi-static techniques, and compiler assisted branch predication. We described the previous work on improving interpreter performance, outlining the different types of opcode dispatch. We described the previous research which investigated the relation between interpreters and indirect branching. We also described some of the existing work on optimizing virtual function dispatch.

⁷ This optimization is only relevant when native code is generated.

The remaining chapters of this dissertation focus on presenting novel material, establishing the thesis we set out in Chapter 1. In the next chapter we investigate ordering the opcode implementations in an interpreter in order to improve performance.

TUNING VIRTUAL MACHINE OPCODE ORDERINGS

The year was 1945. Two earthshaking events took place: the successful test at Alamogordo and the building of the first electronic computer. Their combined impact was to modify qualitatively the nature of global interactions between Russia and the West. No less perturbative were the changes wrought in all of academic research and in applied science. On a less grand scale these events brought about a [renaissance] of a mathematical technique known to the old guard as statistical sampling; in its new surroundings and owing to its nature, there was no denying its new name of the Monte Carlo method.

— N. Metropolis [Cooper et al., 1989]

4.1 INTRODUCTION

In Chapter 2 we introduced virtual machine interpreters, which are widely used to implement programming languages such as Java, Lua and Python.

As discussed in Section 2.2.1, language implementations based on virtual machines typically compile high-level source code into an array of bytecodes (opcodes). They then use an interpreter to execute the array of opcodes. Often this interpreter is written in C and uses a large switch statement to select the appropriate code to execute for the current opcode. The code implementations (switch cases) of these opcodes are usually laid out in an ad-hoc manner.¹ The most common ordering in the source code of VM interpreters we have examined is the (arbitrary) numeric order of the corresponding opcodes.

A reasonable question to ask then is whether the order of the cases in the interpreter loop can have a significant impact on performance. For example, if the commonly executed cases are kept together, then it may improve code locality. Sun Microsystem's K Virtual Machine for embedded Java [Sun, 2000] does exactly this, which suggests that there may be real benefits from code locality. There may also be less obvious benefits. For example, arbitrary changes in the order of the interpreter code may change the relative positions of conditional branch instructions to each other, which may result in fewer collisions in the

¹ Since compiler optimizations often move code around, this might be considered a case of 'the compiler knows better'.

branch predictors. Finally, arbitrary re-orderings of code can have almost arbitrary effects on performance. For example, even code that is never executed can have an impact on performance if it is fetched into the processor pipeline. The less obvious benefits are much more difficult to model than locality effects, and it is thus difficult to design code optimization strategies that take advantage of them. However, it is valuable to discover the extent to which these arbitrary outcomes can improve performance.

We are not the first to investigate the question of ordering the cases within the interpreter loop to improve locality. Brunthaler [2011], who refers to this problem as interpreter instruction scheduling,² provides an algorithm to re-order cases to improve locality, and reports speedups for a modified Python interpreter.

In this chapter we address the case ordering question for VM interpreters, but take a different approach. In addition to improving locality we also want to investigate other effects of case re-ordering. We therefore take a feedback-directed iterative approach to finding good case orderings. This allows us to capture other speedups beyond those resulting in improved locality, and investigate their causes. This chapter makes a number of contributions.

- We evaluate Brunthaler's [Brunthaler, 2011] graph selection approach in our experimental framework and present results.
- We propose an iterative solution based on the Monte Carlo method, showing it to be a reasonable search method for this problem.
- We evaluate two alternatives to this method, based on hill climbing and simulated annealing.
- We apply our methods to the Python 2.6, Python 3.2, and Lua 5.1 interpreters. We measure the difference the effect has between switch and token-threaded interpreters, and present results for the Intel Core and Nehalem architectures.
- Using hardware performance counters, we demonstrate that in our experiments performance improvements from code re-ordering are not primarily the result of improved locality.
- We show that on recent processors the performance impact of case re-ordering is primarily the result of its effect on indirect branch prediction.

² We refer to the concept as ordering, as scheduling implies a temporality property, which is not the case here.

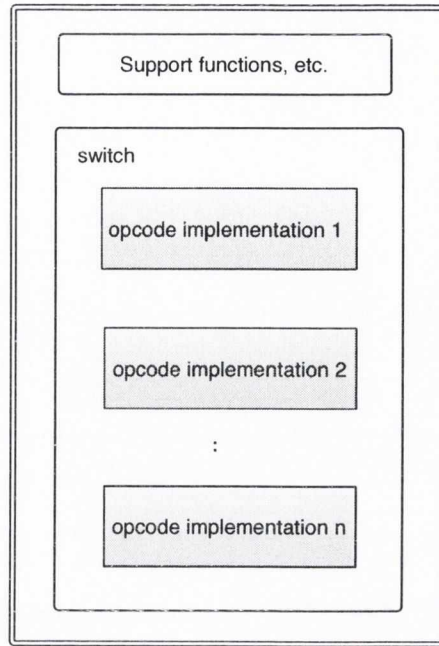


Figure 4.1: Typical interpreter structure.

4.2 BACKGROUND

In Chapter 2 we described some of the computer architecture subsystems that typically influence program performance. We discussed the instruction cache, data cache, and branch prediction. Interpreters are influenced by these subsystems also. An interpreter does not usually stall waiting for data, so we focus our attention on instruction cache and branch prediction misses.

Instruction caches, which we described in Section 2.6, rely on the property that program flow is highly predictable, and therefore instruction memory access is predictable (it has good spatial and temporal locality). If programmers want to improve the cache behavior, they should generate the smallest code possible. In addition, they can improve prefetching through code layout, and/or by use of explicit prefetching.

As we stated, modern processors contain highly sophisticated branch predictors that attempt to avoid branch misprediction and a flush of the long pipelines that they contain. Effective branch predictors are so important for performance that they are a competitive edge for CPU manufacturers, so their specific behavior is not publicly documented. In Section 2.5.5 we described the attempts to reverse engineer the exact details of branch predictors [Milenkovic et al., 2002].

An interpreter is frequently written in C, with code fragments that implement the instruction set of the virtual machine. Opcode ordering is the arrangement of the implementations of opcodes in memory. Figure 4.1 shows the typical layout of an interpreter core. Here we presume it is a switch interpreter; however, direct and indirect threaded interpreters will be structured in a similar fashion.

In Section 2.2 we described the Lua and Python languages. The interpreters we analyze in this chapter are Lua 5.1, Python 2.6 and Python 3.2. Lua 5.1 is a register virtual machine with a switch-based interpreter and has 38 opcodes. Python 2.6 is a stack virtual machine with a switch-based interpreter and has 112 opcodes. Python 3.2 is token-threaded when compiled with a compiler that supports labels-as-values (computed goto), otherwise switch-based. It has 100 opcodes.

4.3 MOST FREQUENT FIRST

To find a good ordering, perhaps the most ‘obvious’ choice would be to arrange the opcode implementations such that the most frequent are clustered together at the top of the interpreter dispatch loop. Indeed it appears that this is a common practice amongst interpreter designers: we see `LOAD_FAST` appear at the top of the Python dispatch loop, and `OP_MOVE` appear at the top of the Lua dispatch loop.

As a concrete example, Table 4.1 shows the frequencies of various opcodes for the spectral norm benchmark. The most frequent opcode, `ADD` would be the first opcode implementation in the memory address space, and similarly for the following opcodes.

We implement ordering based on dynamic execution frequency for the Lua and Python interpreters by aggregating opcodes from profiles we collect by running the various benchmarks in our set. We place the most frequently executed ones at the top of the switch.

An issue with the Python implementation is the use of ‘fall through’ semantics for the overlapping implementations of some opcodes. For instance the implementation of the `PRINT_ITEM_TO` opcode (which prints the item second from the top of stack) shown in Listing 4.1 has only one extra statement compared to the `PRINT_ITEM` opcode, so the implementations are overlapped to save code memory. This is a complication, since our opcode traces will have two different opcodes. We do not want to consider duplicate implementations.³ We solve this

³ A potential approach might be to split the opcodes, however, we have not explored this. The effects of interpreter opcode replication have been explored by Casey et al. [2005]

Opcode	Frequency
ADD	18,000,600
MUL	10,800,600
MOVE	7,200,220
FORLOOP	3,624,653
GETTABLE	3,600,604
CALL	3,600,064
RETURN	3,600,061
GETUPVAL	3,600,040
DIV	3,600,001
LOADK	36,090
SETTABLE	12,300
FORPREP	12,043
SUB	12,042
GETGLOBAL	6
CLOSURE	4
NEWTABLE	3
TEST	2

Table 4.1: Opcode execution frequencies for the Lua spectral norm benchmark, showing a possible ordering from most frequent to least frequent.

```
case PRINT_ITEM_T0:
    w = stream = POP();
    /* fall through to PRINT_ITEM */

case PRINT_ITEM:
    v = POP();
    ...
    break;
```

Listing 4.1: Problem of joined opcode implementations in Python. The two opcodes implement almost identical functionality, so a ‘fall though’ between cases is used to avoid code duplication.

Benchmark	Lua	Python2.6	Python3	Python3-token
fannkuch	-3%	-1%	5%	-4%
mandelbrot	10%	2%	6.5%	2.5%
fasta	4.5%	-1.5%	-1%	2.5%
n-body	-0.5%	1.5%	6.5%	0%
binary-trees	-7.5%	10.5%	6.5%	0%
spectral-norm	5.0%	13.5%	5.5%	-4%
geo-mean	1%	4%	5%	-0.5%

Table 4.2: Most frequent first speedups on Core 2. Each benchmark is executed by an interpreter tuned using its profile.

by pairing the opcodes that fall through with the following opcode — we are only considering opcode implementations, not opcodes.

Our traces include the start-up functions of the Python interpreter. That is, 70,394 opcodes in Python 2.6, and 823,060 opcodes in Python 3.2. A slight caveat is that we convert the parallel benchmarks (mandelbrot & binary-trees) to sequential equivalents in order to get a coherent trace.⁴

Tables 4.2 and 4.3 show the speedups from most frequent first ordering on the Core 2 and Nehalem machines⁵ respectively (rounded to the nearest 0.5%) using benchmarks from the Computer Language Benchmarks Game [CLBG, 2011]. Our experimental methodology is described in Section 4.8. We observe some speedups, but also some slowdowns. These results suggest that most frequent first is not a good heuristic for ordering opcode implementations.

⁴ An issue with our mandelbrot benchmark for Python 3 is that it uses a concurrent worker-writer model, so we could not remove the concurrency beyond a minimum of two processes.
⁵ We describe our experimental machines in Section 4.8.2.

Benchmark	Lua	Python2.6	Python3	Python3-token
fannkuch	8%	13%	-2.5%	-0.5%
mandelbrot	32.5%	17%	6.5%	-0.5%
fasta	10.5%	8.5%	0%	1%
n-body	0%	10%	4.5%	2.5%
binary-trees	-1%	3.5%	1.5%	0%
spectral-norm	5%	17%	0%	0.5%
<i>geo-mean</i>	8.5%	11.5%	1.5%	0.5%

Table 4.3: Most frequent first speedups on Nehalem. Each benchmark is executed by an interpreter tuned using its profile.

4.4 GRAPH SELECTION

Brunthaler [2011] presents the problem of finding a good ordering as a graph problem, where the vertices represent opcodes and the edges represent the frequency of transitions between opcodes (collected from a profile). This graph is a weighted directed graph that is weakly connected. We wish to find a tour that visits each vertex once, and maximises the cost function. This problem is similar to the asymmetric travelling salesman problem, where the triangle inequality⁶ does not hold. However, the graph is incomplete, so it also resembles the Hamiltonian path problem. This is not exactly right either though, since a tour is permitted to include edges that do not exist in the graph. These edges may be added because it is legal to place opcode implementations adjacent to each other even if they do not form a pair in the profile used to construct the graph. The only guaranteed tour in the graph (without adding extra edges) is the tour formed by the sequence of opcodes from the profile.

In Algorithm 4.1, we outline the graph selection algorithm described by Brunthaler. `Schedule` and `open` are ordered sets. Sorting the graph means to sort the nodes in descending order of their indegree. The algorithm is similar to the classical nearest neighbour (greedy) algorithm [Bang-Jensen et al., 2004], with some differences:

- Revisiting a vertex is permitted.

⁶ An instance of the travelling salesman problem satisfies the triangle inequality if i, j, k are cities and $d_{ij} + d_{jk} \geq d_{ik}$ where d is the distance between two cities. In other words, a direct connection is never longer than a route through an intermediate city.

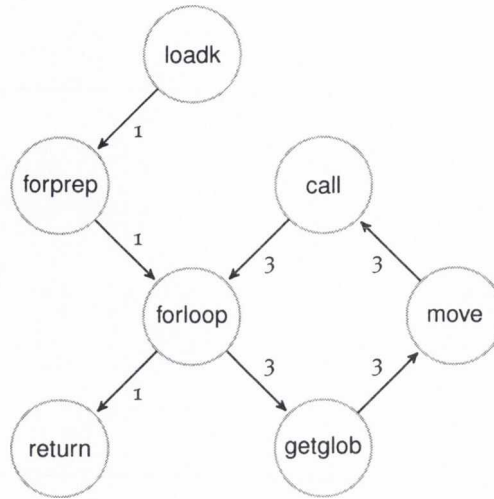


Figure 4.2: Opcode profile graph for the Lua program:
 'for i = 1,3 do print(i) end'.

Algorithm 4.1 Brunthaler's graph selection algorithm.

```

1: Input: Graph.
2: Output: Ordered set of nodes (schedule).
3: schedule  $\leftarrow \emptyset$ 
4: open  $\leftarrow \text{sorted}(\text{graph})$ 
5: while open  $\neq \emptyset$  do
6:   n  $\leftarrow \text{largest}(\text{open})$ 
7:   c  $\leftarrow \text{true}$ 
8:   while c do
9:     schedule  $\leftarrow \text{schedule} \cup \{n\}$ 
10:    reachable  $\leftarrow \text{sorted}(n.\text{destinations}) \setminus \text{schedule}$ 
11:    c  $\leftarrow \text{false}$ 
12:    if reachable  $\neq \emptyset$  then
13:      n  $\leftarrow \text{largest}(\text{reachable})$ 
14:      open  $\leftarrow \text{sorted}(\text{reachable} \cup \text{open})$ 
15:      c  $\leftarrow \text{true}$ 
16:    end if
17:  end while
18: end while
19: return schedule

```

Benchmark	Lua	Python2.6	Python3	Python3-token
fannkuch	-4.5%	-1.5%	2.5%	-4%
mandelbrot	10%	5.5%	5.5%	-1.5%
fasta	1.5%	0.5%	-1.5%	3%
n-body	-3%	6%	5%	-1.5%
binary-trees	-0.5%	8.5%	8%	0%
spectral-norm	4%	21.5%	9%	1%
<i>geo-mean</i>	1%	6.5%	4.5%	-0.5%

Table 4.4: Graph selection speedups on Core 2. Each benchmark is executed by an interpreter tuned using its profile.

- If we reach a ‘dead-end’ (recall that the graph is not complete) then the next node is fetched from the open set.
- The final difference is the open set updating:

$$\text{open} \leftarrow \text{sorted}(\text{reachable} \cup \text{open})$$

Items reachable from the present node are added to the open list. These items will have higher precedence, because items initially put in the open list are sorted, but are placed in with weight zero (the ordering of the original nodes is not affected, since a stable sort is used).

As an example, applying the algorithm to the profile graph shown in Figure 4.2 yields the ordering: {FORLOOP, GETGLOBAL, MOVE, CALL, RETURN, FORPREP, LOADK}.

We diverge slightly from the application of the algorithm as presented by Brunthaler. Instead of determining program kernels and applying the algorithm to the traces of those kernels, we apply the algorithm to traces of whole programs. This should not have detrimental effects, as the algorithm selects for frequency, so infrequent opcodes should get relegated in any case.

Brunthaler provides Python source and we base our implementation on this [Brunthaler, 2011]. We evaluate his graph technique for the Lua, Python 2.6 and Python 3.2 interpreters on the Core 2 and Nehalem machines. The results we found are shown in Tables 4.4 and 4.5.

These results show the speedups (and slowdowns) when an ordering found from the trace of a benchmark is evaluated on that benchmark. To our surprise,

Benchmark	Lua	Python2.6	Python3	Python3-token
fannkuch	7.5%	8.5%	-7%	-0.5%
mandelbrot	8.5%	17%	-0.5%	-4%
fasta	9%	7.5%	0.5%	0.5%
n-body	-3.5%	9%	1.5%	1.5%
binary-trees	2.5%	6%	-1.5%	-1%
spectral-norm	2%	9%	4.5%	-6%
<i>geo-mean</i>	4%	9.5%	-0.5%	-1.5%

Table 4.5: Graph selection speedups on Nehalem. Each benchmark is executed by an interpreter tuned using its profile.

we found that better speedups can be achieved when we do not restrict what benchmark an ordering was tuned on, and instead try all the possible orderings generated from traces of the different benchmarks. We do not present those numbers here, as the algorithm is meant to find good orderings for the trace it is presented with.

To improve upon these results, a better algorithm than the greedy algorithm could be used to minimize the global distance of a tour (an optimal solution might be possible in $O(2^n)$). However, we do not explore this: since an ordering tuned from a particular benchmark is not always the best ordering for that benchmark, it must follow that the graph technique is modelling the wrong thing. We instead now look at orderings generated stochastically.

4.5 MONTE CARLO GENERATION

In the previous two sections we looked at opcode frequency based methods to layout opcodes in memory. The results we found were unsatisfactory, so we now investigate a more general, combinatorial technique. We want to find the opcode ordering that has the lowest execution time (our cost function). The combinatorial search space of all possible opcode orderings is every permutation, which is $O(n!)$ complexity for a brute force approach. Since most interpreters have more than thirty opcodes, this search space will be upwards of 10^{32} (For Lua, which has a relatively low number of opcodes (38), there are over 10^{44} permutations). Without any indication of an optimal algorithm, we investigate stochastic search methods.

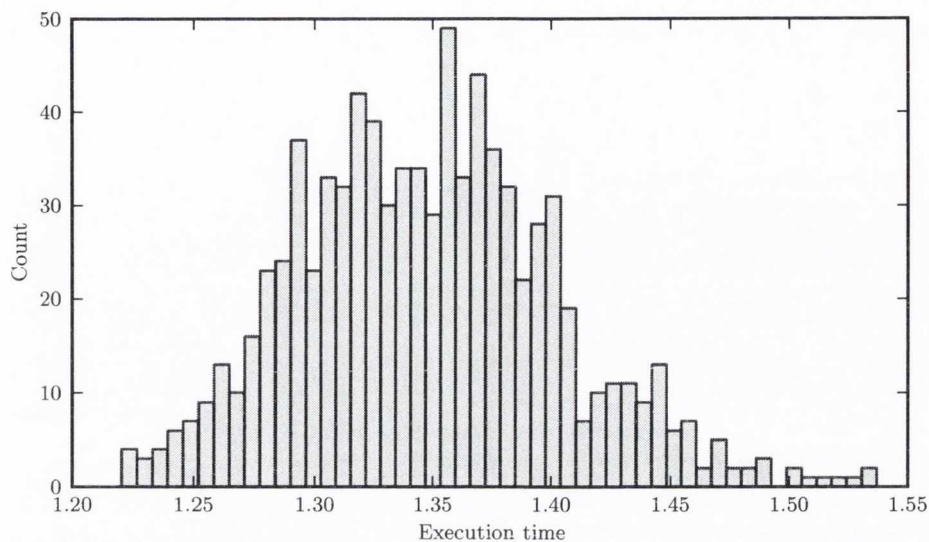


Figure 4.3: Histogram showing the frequencies (number of orderings that ran in that time) of different ordering execution times (in seconds) found by Monte Carlo sampling for the mandelbrot benchmark on Python 3 (switch).

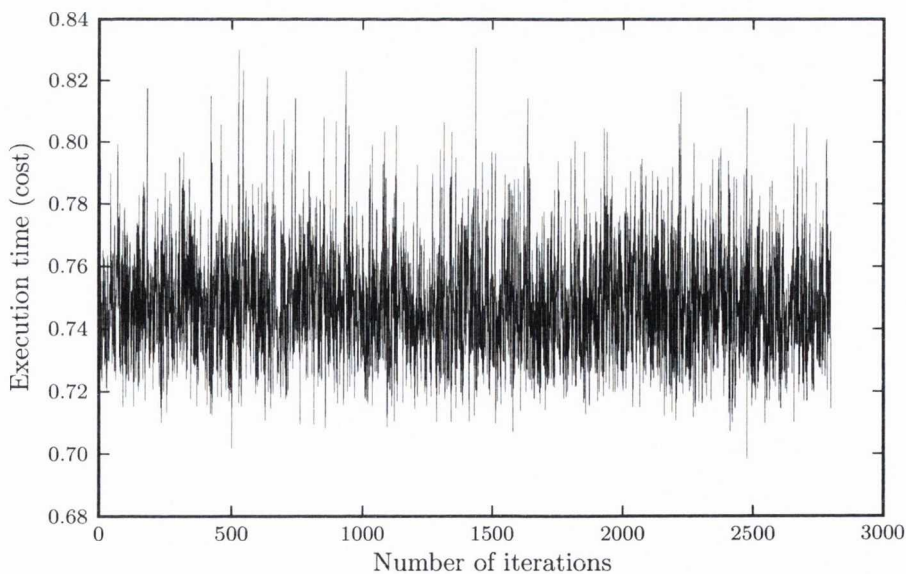


Figure 4.4: Monte Carlo progress on Lua fannkuch benchmark on Core 2. The best ordering in this case was found after 2500 iterations.

Benchmark	Lua	Python2.6	Python3	Python3-token
fannkuch	5%	18%	17%	5%
mandelbrot	14%	13%	20%	4.5%
fasta	22%	6%	3.5%	4.5%
n-body	5.5%	9%	15%	5%
binary-trees	4%	17%	12%	4%
spectral-norm	7%	39%	22%	1%
<i>geo-mean</i>	9.5%	16.5%	15%	4%

Table 4.6: Monte Carlo generation speedups on Core 2 (individually tuned orderings).

Monte carlo methods are used to model phenomena with significant uncertainty in inputs. With Monte Carlo generation of orderings, we randomly (from an underlying uniform distribution) permute the opcodes, compile the interpreter and evaluate it on a set of benchmarks, recording the best time observed so far.

The combinatorial complexity of all possible opcode orderings is similar to that of finding the shortest tour in the travelling salesman problem. Hence we can borrow the theorem that to stochastically achieve a result that is within $1 + \frac{1}{c}$ of the optimal solution for TSP, for some constant c , with at least 50% certainty, requires the following complexity: $O(n2^{c^{O(1)}} + n \log n)$ [Kao, 2008, p. 282]. With this exponential increase in time as c grows linearly, we rapidly stop seeing improvements. We see this in our experimental observations similarly.

An example of Monte Carlo generation is shown in Figure 4.3. The figure shows a histogram of the execution times for the various opcode orderings generated for the mandelbrot benchmark. This example is for the Python 3 switch interpreter on our Core 2 machine. Figure 4.4 shows the progress of the Monte Carlo method over time for the fannkuch benchmark. This is for the Lua interpreter on our Core 2 machine.

Table 4.6 summarises the various speedups achieved on the different interpreters we measured on the Core 2. These speedups are achieved by running each benchmark with the best possible ordering found for it. In contrast, Table 4.7 shows speedups (and an overall average speedup) when just one ordering is used per set of benchmarks. Tables 4.8 and 4.9 show the same results, but for our Nehalem machine.

Benchmark	Lua	Python2.6	Python3	Python3-token
fannkuch	0%	14%	13%	5%
mandelbrot	13%	11.5%	19.5%	4.5%
fasta	18%	3%	2%	4.5%
n-body	3.5%	4.5%	13.5%	5%
binary-trees	-1%	14%	9.5%	4%
spectral-norm	4%	38%	18%	1%
<i>geo-mean</i>	6%	14%	12.5%	2.5%

Table 4.7: Monte Carlo generation speedups on Core 2 (using one Pareto optimally tuned ordering).

Benchmark	Lua	Python2.6	Python3	Python3-token
fannkuch	10%	18%	6%	4%
mandelbrot	40%	20%	13.5%	2.5%
fasta	14%	9.5%	2.5%	1.5%
n-body	3%	18%	9%	2%
binary-trees	4.5%	12.5%	6.5%	3%
spectral-norm	23%	28%	11.5%	2%
<i>geo-mean</i>	15%	17.5%	8%	2.5%

Table 4.8: Monte Carlo generation speedups on Nehalem (individually tuned orderings).

Benchmark	Lua	Python2.6	Python3	Python3-token
fannkuch	7.5%	16%	1.5%	2.5%
mandelbrot	35%	17%	10%	1%
fasta	12.5%	8.5%	-1%	0.5%
n-body	-2%	12.5%	6%	0%
binary-trees	1.3%	10.5%	4%	2%
spectral-norm	15%	19.5	10.5%	1%
<i>geo-mean</i>	11.5%	14%	5%	1%

Table 4.9: Monte Carlo generation speedups on Nehalem (using one Pareto optimally tuned ordering).

Benchmark	L1-I miss %	Branch miss %
fannkuch	0.01	14.46
mandelbrot	0.05	13.74
fasta	0.00	8.18
n-body	0.01	16.42
binary-trees	0.01	3.45
spectral-norm	0.03	13.17
<i>har-mean</i>	0.01	8.69
<i>total</i>	0.03	12.10

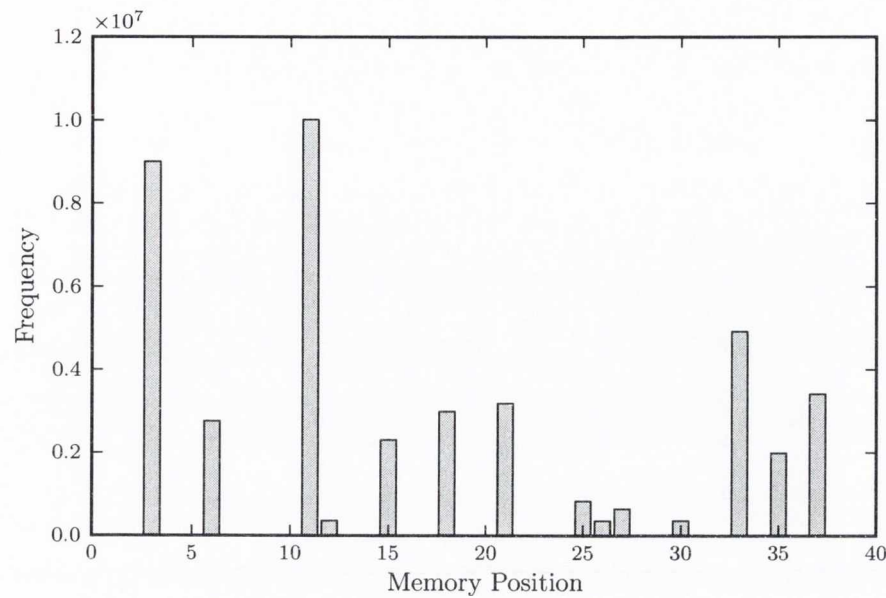
Table 4.10: PAPI performance counter results for Lua showing stalled cycles due to branch misspredictions and cache misses as a percentage of total cycles. The ‘total’ row is the ratio of the total number of stalled cycles due to cache misses (or branches) to the total number of cycles from running all benchmarks.

The random permutations for Monte Carlo are generated by the Mersenne twister pseudorandom number generator, which has a period of $2^{19937} - 1$ [Matsumoto and Nishimura, 1998]. This will work for sequences up to size 2080 (which will have $\approx 2^{19932}$ permutations).

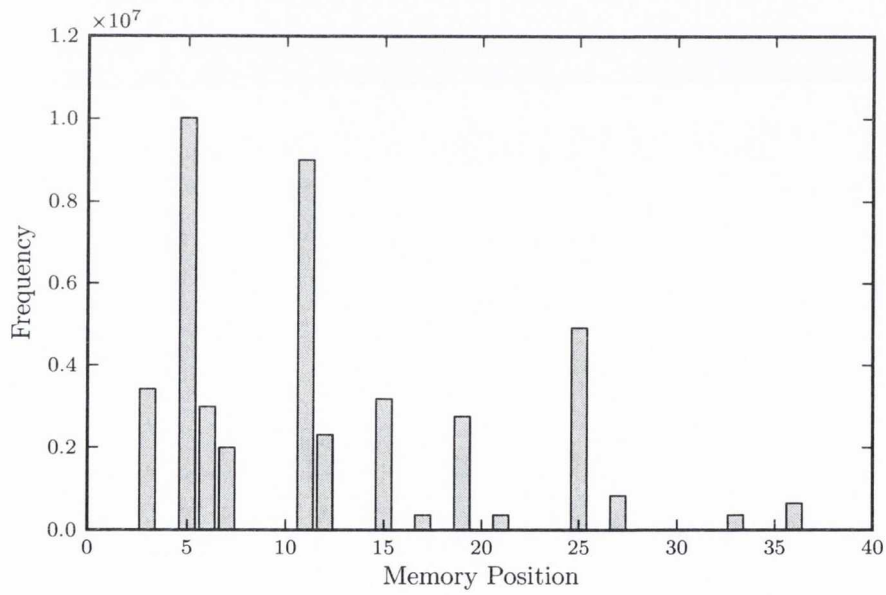
We see respectable speedups from the Monte Carlo approach, but we want a deeper understanding of the cause. Figure 4.5 shows two memory layouts of opcode implementations with opcode execution frequencies for the fannkuch benchmark on the Lua interpreter. On the left we see the best ordering found for that benchmark, on the right we see the worst. After inspecting a number of these graphs, we did not see any common visual pattern to account for the performance variations.

4.6 HARDWARE ANALYSIS

We desire a deeper understanding of what is really going on in hardware with the performance variation from different opcode orderings, so we look at cache and branch prediction:



(a) Opcode memory layout for best ordering found via Monte Carlo.



(b) Opcode memory layout for worst ordering found via Monte Carlo.

Figure 4.5: Execution frequency of opcodes and their implementation positions in memory. The x-axis shows the relative position of opcodes for an ordering (there are 38 opcodes) and its units are index positions in memory. Both figures show frequencies for the fannkuch benchmark on Lua (38 opcodes). The figures do not show any clear visual pattern as to what a better ordering may be.

4.6.1 Cache

Firstly, we investigate the effect of level 1 instruction cache on the performance of a switch-case interpreter, since the instruction cache is considered to be the bottleneck in Brunthaler's work [Brunthaler, 2011]. For our Lua benchmark sample, we do not observe a significant number of instruction cache misses. We evaluate this using Pin [Luk et al., 2005] and Valgrind [Nethercote and Seward, 2007] with a 32KB 8-way set associative cache (the size of the cache in our Core 2 machine).⁷ We also record hardware performance counters via PAPI [Browne et al., 2000]. Finally, we measure the code distance between the beginning and end of the switch block by taking the address of labels (this excludes executed code that is in functions outside of the interpreter loop; but in Lua, code such as this is often the special case anyway). The total address distance in the Lua interpreter loop is 5953 bytes. This is considerably less than the 32KB level 1 instruction cache on the Intel Core. Additionally, the full code size of the interpreter is not all going to be needed in the instruction cache — the frequency of opcodes is not distributed in such a uniform manner, as the opcode frequency counts of the spectral norm benchmark shown in Table 4.1 illustrate.

Table 4.10 shows estimated figures of the percentage of cycles stalled due to L1-I misses.⁸ We calculate these from the cycle count of the benchmarks and the L1-I misses as reported by the hardware performance counters. In all cases, the proportion of time lost to L1 instruction cache misses is significantly less than 1%.

4.6.2 Branch Prediction

Since instruction cache misses do not appear to be that much of a concern for the interpreter, we turn to investigating the effect of branch prediction.

Table 4.10 shows estimated figures of the percentages of cycles stalled due to branch mispredictions. We calculate these from the cycle count of the bench-

⁷ Pin is dynamic binary instrumentation tool that is useful for modelling caches and branch predictors. Valgrind is another instrumentation framework that includes a tool called *cachegrind* for simulating caches. We use these simulators as they give precise figures for the models they use, whereas hardware counters could suffer from a transitory error.

⁸ We assume a fourteen cycle branch penalty (pipeline size) and a fifteen cycle L1-I miss penalty. Obviously these penalties are highly dynamic in a modern out-of-order superscalar CPU, but our estimates are representative, and show a large difference between cycles stalled due to branch mispredictions and due to cache misses.

marks and the branch misses as reported by the hardware performance counters. We see that branch misprediction is commonplace and significant.

Branch mispredictions can be caused by conditional branch misses. We looked at those for the orderings generated by our Monte Carlo technique, but did not observe a correlation between conditional branch mispredictions and execution time. The alternative cause of branch mispredictions is indirect branch misses.

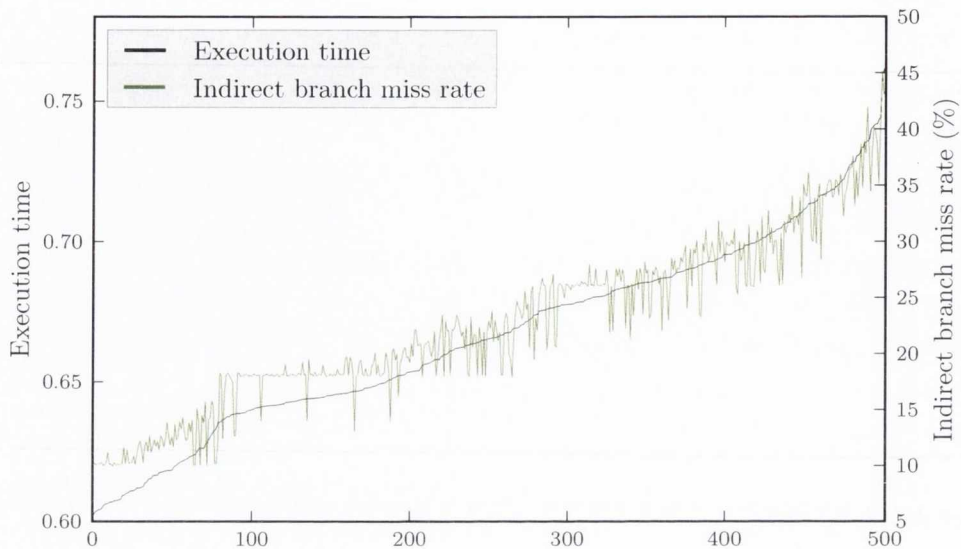
Figure 4.6(a) shows execution times and indirect branch misprediction rates for various opcode orderings found through the Monte Carlo method (x-axis is sorted by execution time). It is clear from this figure that ordering affects execution speed mainly through indirect branch misses. Figure 4.6(b) shows execution times, total indirect branch mispredictions and level 1 instruction cache misses for various opcodes orderings found through Monte Carlo. This figure also shows that the instruction cache is not a significant determinant of execution speed (the cache axis is not shown, but it is 25 times smaller than the indirect branch axis).

The low token-threaded speedups we witnessed from Monte Carlo generation also support the claim that indirect branch prediction is the cause of varying execution speed.

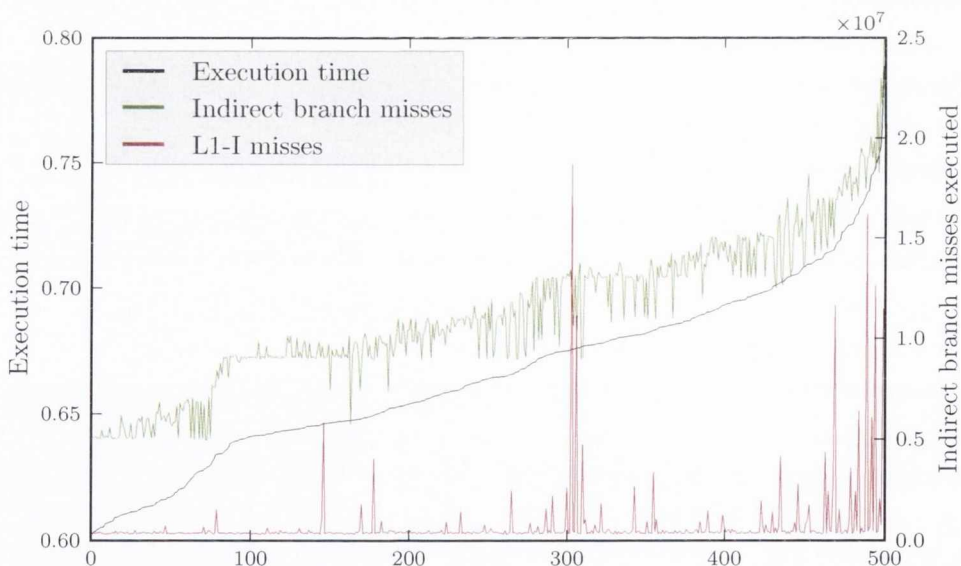
An explanation of the effect of opcode ordering on indirect branch misprediction is that the Core microarchitecture includes an indirect branch predictor like the Intel Pentium M [Gochman et al., 2003]. In Section 2.5.5 we describe the work of Uzelac and Milenkovic [2009], who reverse engineer the indirect branch predictor in the Pentium M. Their work shows that the iBTB is indexed by a path information register, which is updated with a shift and xor on the six least significant bits of the indirect branch target address. Moving opcode implementations in an interpreter will change the bits of the target, and this will lead to changes in accuracy in the path information register, and consequently changes in the accuracy of indirect branch misprediction.

4.6.3 Branch prediction as a Cost Function

Since we know that the ordering primarily affects execution time through branch mispredictions, we could also have used the branch misprediction hardware counter while searching for a good ordering. We tried this, but found that it was not as good as execution speed, and it was unnecessary to use a proxy in any case. This means that indirect branch prediction is not the only issue, even if it is the dominant one.



(a) Execution times vs. indirect branch misprediction rates for different opcode orderings generated by Monte Carlo (sorted by execution time).



(b) Execution times vs. indirect branch mispredictions vs. level 1 instruction cache misses (axis not shown) for different opcode orderings generated by Monte Carlo (sorted by execution time).

Figure 4.6: Lua interpreter executing the fasta benchmark on Core 2. The two figures show data from different orderings.

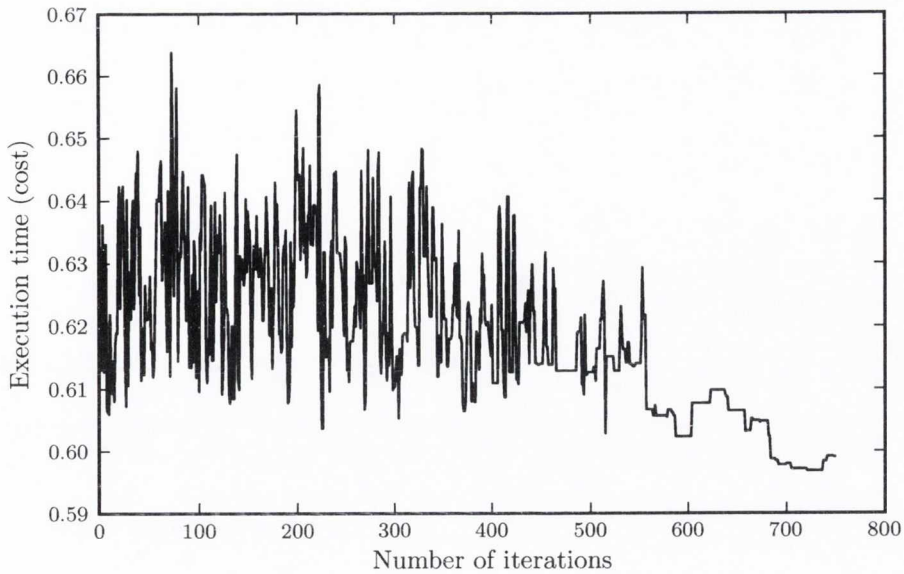


Figure 4.7: Progress over time of simulated annealing on Lua fannkuch benchmark on Nehalem.

4.7 FEEDBACK-DIRECTED SEARCH

In the previous sections we looked at frequency-based algorithms to find orderings and an initial metaheuristic based on Monte Carlo generation of orderings. We now explore a more guided approach for finding satisficing⁹, orderings.

4.7.1 Simulated Annealing

Once more inspired by the travelling salesman problem, we use simulated annealing as a ‘smarter’ method to generate a good ordering. The solution search space is coherent, so we can traverse it by transitioning to the next candidate solution by modifying the current one. Simulated annealing is an appropriate heuristic search method as it attempts to avoid getting stuck in local maxima. The diameter of the search space is small, which is good for simulated annealing — we can get from one state to any other in $n(n-1)/2$ steps.

Algorithm 4.2 shows our simulated annealing algorithm. Our initial candidate is simply the default ordering in the interpreter. Figure 4.7 shows the progress of the simulated annealing algorithm over time on the fannkuch benchmark on Lua. We evaluated the algorithm for Lua on the Core architecture and

⁹ Adequate, as opposed to optimal.

Algorithm 4.2 Simulated annealing algorithm used to traverse the solution space.

```

1: Input: Evaluation function.
2: Output: A satisficing ordering.
3: cooling_fraction  $\leftarrow$  0.8
4: cooling_steps  $\leftarrow$  30
5: steps_per_temp  $\leftarrow$  100
6: k  $\leftarrow$  1
7: temperature  $\leftarrow$  1
8: current_perm  $\leftarrow$  ...
9: current_value  $\leftarrow$  evaluate(current_perm)
10: for i = 0 to cooling_steps do
11:     temperature  $\leftarrow$  temperature  $\times$  cooling_fraction
12:     for j = 0 to steps_per_temp do
13:         possible_solution  $\leftarrow$  random_swap(current_perm)
14:         cost  $\leftarrow$  evaluate(current_perm)
15:         merit  $\leftarrow e^{(\text{current\_value} - \text{cost}) / (k \times \text{temperature})}$ 
16:         if cost < current_value then
17:             current_value  $\leftarrow$  cost
18:             current_perm  $\leftarrow$  possible_solution
19:         else if merit > random[0, 1] then
20:             current_value  $\leftarrow$  cost
21:             current_perm  $\leftarrow$  possible_solution
22:         end if
23:     end for
24: end for
25: return current_perm

```

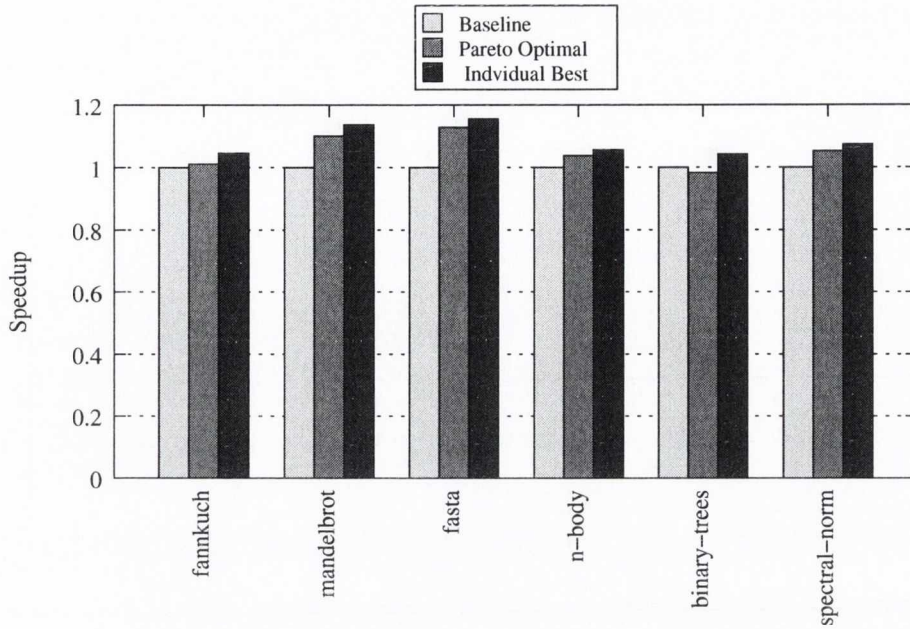


Figure 4.8: Simulated annealing results for the Lua interpreter on Core 2. Pareto optimal refers to one ordering that works best with all benchmarks.

present our results in Figure 4.8. We provide results for individually tuned orderings for each benchmark as well as a global Pareto optimal ordering.¹⁰ We do not see improvements over the best Monte Carlo generated orderings.

4.7.2 Hill Climbing

With simulated annealing, we did not see any results that were better than Monte Carlo. This is likely because we are limited to a small number of iterations (750), due to the high cost of evaluating each ordering, whereas typical simulated annealing systems might run for millions of iterations. We also may not have found the ideal parameters for simulated annealing — the values for k , cooling fraction and steps per temperature. We did observe progress after the algorithm had cooled and was not accepting bad solutions anymore.

We evaluate hill climbing as a search strategy as it is an approach that converges faster than simulated annealing. The simplest form of hill climbing, steepest ascent, would evaluate every neighbor, and transition to the best. However, that is not practical for us, as our neighbors are all the possible switches of pairs in the ordering and there are $\binom{n}{2}$ pairs, which means that even in Lua,

¹⁰ We consider an ordering Pareto optimal if it maximizes the geometric mean of the speedups over the baseline for each benchmark.

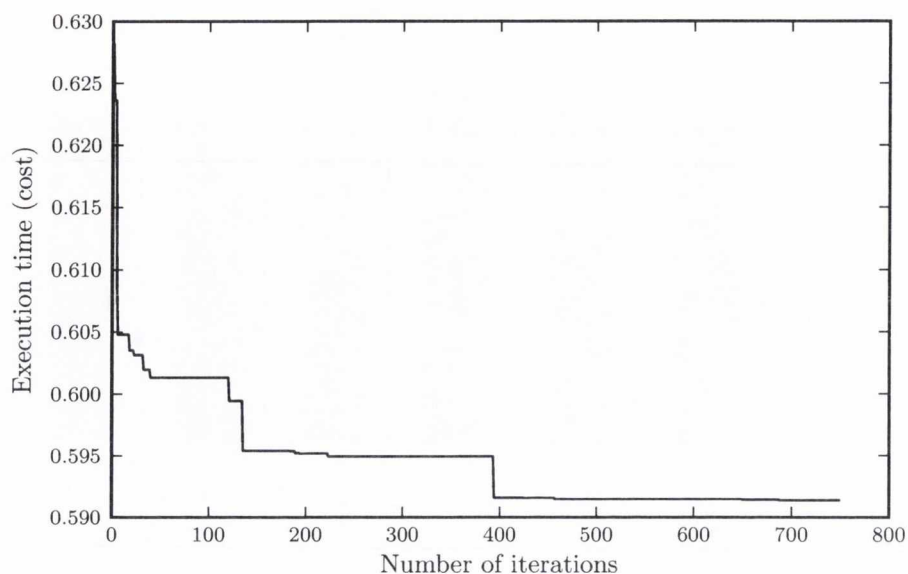


Figure 4.9: Progress over time of the first-choice stochastic hill climbing algorithm on Lua fannkuch benchmark on Nehalem.

with 38 opcodes, we will have 703 neighbors at each step. Because of this, and our high evaluation cost, we evaluate stochastic hill climbing. In particular we do first-choice stochastic hill climbing (always picking the first neighbor if it is uphill).

Figure 4.9 shows the progress of the basic algorithm over time for the Lua interpreter running the fannkuch benchmark. We see the algorithm go for long periods while making little or no progress. We look at two techniques to combat this:

- Since we are measuring execution time, sampling error is a concern. The hill climbing algorithm is greedy, and will only accept better solutions, but it should also accept equal solutions; in this way it can traverse a plateau and possibly ascend again. However, it is possible to get stuck on a plateau and not make progress if — due to a measurement error while evaluating an ordering — we measure a particularly low execution time. This way, there will be no transitions to neighbors that may have true equal cost, and the algorithm will not make progress. To avoid this scenario, we modify the algorithm to allow transitions to inferior solutions, provided they are within the error range (a figure we find empirically). This change alone would allow gradual descent, so we also keep track of

Algorithm 4.3 First-choice stochastic hill climbing with periodic restarts (and error consideration) algorithm used to traverse the solution space.

```

1: Input: Evaluation function, acceptance predicate.
2: Output: A satisficing ordering.
3: not_improved_count  $\leftarrow$  100
4: repeat
5:   if not_improved_count  $\geq$  100 then
6:     not_improved_count  $\leftarrow$  0
7:     current_perm  $\leftarrow$  random_perm()
8:     current_value  $\leftarrow$  evaluate(current_perm)
9:     best_value  $\leftarrow$  current_value
10:  end if
11:  possible_solution  $\leftarrow$  random_swap(current_perm)
12:  possible_solution_cost  $\leftarrow$  evaluate(possible_solution)
13:  delta  $\leftarrow$  current_value - possible_solution_cost
14:  delta_best  $\leftarrow$  best_value - possible_solution_cost
15:  not_improved_count  $\leftarrow$  not_improved_count + 1
16:  if delta > -error and delta_best > -error then
17:    current_perm  $\leftarrow$  possible_solution
18:    current_value  $\leftarrow$  possible_solution_cost
19:    if current_value < best_value then
20:      best_perm  $\leftarrow$  current_perm
21:      best_value  $\leftarrow$  current_value
22:      not_improved_count  $\leftarrow$  0
23:    end if
24:  end if
25: until acceptable solution found
26: return best_perm

```

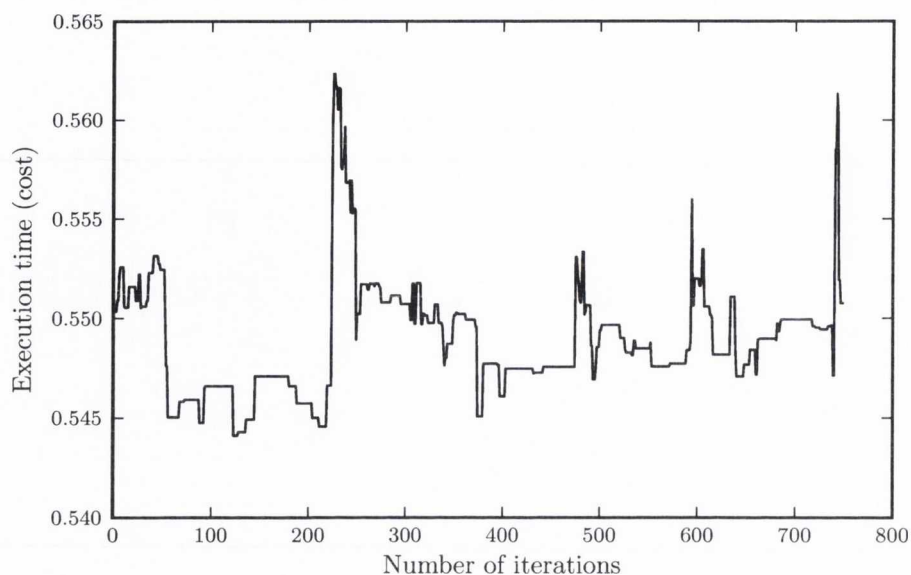


Figure 4.10: Progress over time of the first-choice stochastic hill climbing with error consideration and periodic restarts algorithm on Lua fasta benchmark on Nehalem.

the best value found so far, and refuse to accept transitions to potential solutions that are not within the error range of the best.

- We also perform periodic restarts of the algorithm if no progress is made (an improvement on the best solution is not found) after a fixed number of iterations.

Algorithm 4.3 shows our modified hill climbing algorithm and Figure 4.10 shows the progress of this modified algorithm over time for the Lua interpreter running the fasta benchmark (plateaus are not as obvious due to the error consideration correction). The algorithm restarts if no progress is made in 100 iterations. In this example, it restarts four times. In our experiments we observe that it is able to make more progress than the initial naive hill climbing algorithm. (Figure 4.10 does not show the usual case of more progress following a restart.)

Figure 4.11 shows the results of the algorithm for our benchmark set on our Nehalem machine. The algorithm does not find better individually tuned orderings than the Monte Carlo method, but it does find a better Pareto optimal ordering for the benchmark set, even though we evaluate it using a larger benchmark set than we did for Monte Carlo.

Part of the reason we did not see even larger improvements over Monte Carlo is that algorithms such as hill-climbing and simulated annealing suit a problem

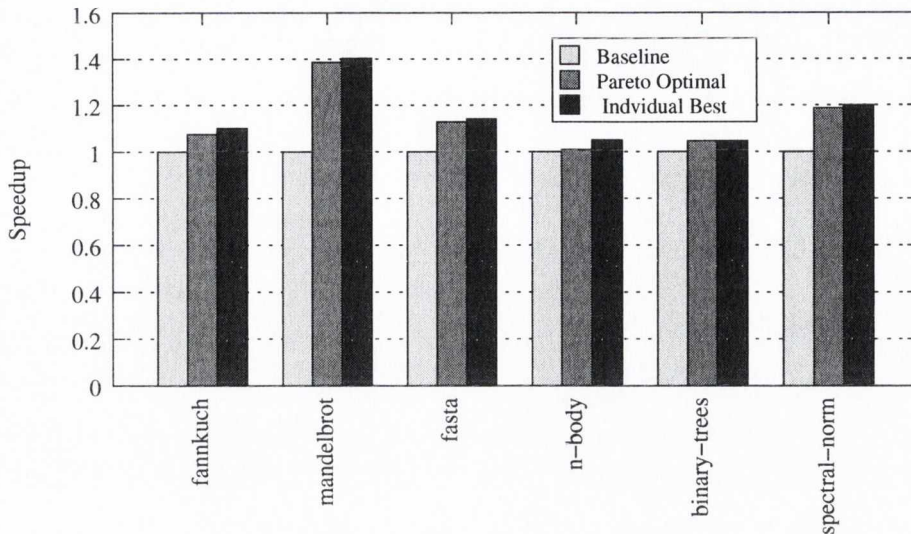


Figure 4.11: First-choice stochastic hill climbing with periodic restarts (and error consideration) algorithm results for the Lua interpreter on Nehalem. Pareto optimal refers to one ordering that works best with all benchmarks.

best when the cost of incremental evaluation is lower than cost of global evaluation. This is not the case here — we must perform a global evaluation on each transition. The cost of evaluating a given ordering is constant, but it is a high constant (approximately 100 seconds per ordering).

4.8 EXPERIMENTAL METHODOLOGY

4.8.1 Benchmarks

Our benchmark set is taken from the Computer Language Benchmarks Game [CLBG, 2011]. A lack of established benchmark sets for scripting languages makes these suites a common source of benchmarking for scripting language implementations [Brunthaler, 2011, Pall, 2011].

We use the GCC compiler with the option *-fno-reorder-blocks* to stop our reorderings from being negated by basic block re-ordering performed by the compiler’s software trace cache. We compile both interpreters with maximum optimization (*-O3*).

When evaluating opcode permutations, we run each benchmark fifteen times, and take the median. While a higher number of runs is obviously more desirable, we found this to be a good trade-off between accuracy and time for our purposes; a larger number would not enable us to evaluate as many candidate

solutions. More complex techniques have been proposed by Leather et al. [2009] to dynamically determine the number of runs required to eliminate noise in an iterative tuning scenario. Regardless, our results for different opcode orderings are fully reproducible.

4.8.2 *Hardware Configuration*

In this chapter, we evaluate our techniques on two systems:

- Intel i5 650 (Nehalem Westmere) 3.2 GHz CPU with 4 GB memory, running Ubuntu GNU/Linux 9.10 with kernel version 2.6.31-22. GCC version 4.4.1.
- Intel Core 2 Q6600 (Core Kentsfield) 2.4 GHz CPU with 6 GB memory, running Ubuntu GNU/Linux 10.04 with kernel version 2.6.32-29. GCC version 4.4.3.

4.8.3 *Bytecode Traces*

Python provides an internal data structure, *dxpairs*, that encodes the opcode pair frequencies (the number of times an opcode follows another one). However, we opted against using it since it is only practical to access it from a Python program itself, and the bytecodes related to this access code will also get included in the trace. Therefore we simply dumped the opcode numbers from the interpreter as they were executed, and translated these into the appropriate opcode names.

4.8.4 *Compiler Label Alignment*

Compilers attempt to make intelligent alignment decisions for code. This is to maximize the amount of code that fits into a cache line, achieving the best hit ratio from the instruction cache (lowest amount of time stalled waiting to fetch code from memory). The GCC compiler can align functions, jumps, labels and loops.

GCC has an option *-falign-labels=[n]* that allows the programmer to specify an alignment for labels. We experimented with this option, however, we found from assembly inspection that we could only increase the alignment bit count, and not reduce it. We manually removed the alignment pseudo-

assembler (*.p2align*) directives and found a speedup of 1.2% on the *fannkuch* benchmark with the Lua interpreter. For comparison, disabling re-ordering for this benchmark led to a 2% speedup. Disabling these two ‘optimizations’ led to a total speedup of 3.3%.

For Python 2 and 3, we did not observe alignment directives between every opcode implementation. This may be due to Python having significantly more code in opcode implementations.

4.9 RELATED WORK

As we described previously in Section 4.4, Brunthaler [2011] presents a formalization of the problem of interpreter opcode ordering (or instruction scheduling) for an interpreter with an extended opcode set, and proposes that the bottleneck is instruction cache misses. We investigated that algorithm for the ‘stock’ Lua and Python interpreters but did not find it to be a good approach. We also found that the main influence of varying performance in these interpreters was indirect branch mispredictions. With an extended opcode set virtual machine, it is conceivable that cache misses are more important to execution time than branches. We note that our feedback-based approach for finding good orderings would also work if cache misses were the bottleneck.

McFarling [1989] presents a profile-based algorithm to reposition programs in memory to reduce instruction cache misses. The effect of his algorithm was comparable to a tripling of cache size in some instances. However, he saw declining results as cache size increased — modern workstation caches are even larger than the maximum size he measured on a MipsX simulator.

Code positioning is also explored by Pettis and Hansen [1990]. Their work presents profile-guided code positioning at the procedure and basic block levels. Their profiles guide the movement of code that will probably not get executed out of the local memory space of the ‘hot’ code. They primarily get performance increases from better code locality in the instruction cache, but they also achieve some speedup from a better ordering of conditional branches. Our work investigates the interplay between code positioning and indirect branches, for which modern hardware has much more sophisticated predictors.

Zhao and Amaral [2005] use feedback-guided techniques to generate optimized code for switch-case statements. They present two feedback-guided techniques to generate code. Their work aims to improve performance by (1) enabling functions with large switch statements to be inlined, (2) improving instruction cache behavior by moving ‘cold’ code, and (3) improving case se-

lection performance by decreasing the number of cases to chose from. Their experiments show speedups of up to 4.9%. Unfortunately, they do not provide hardware performance counter numbers to show the cause of their speedups. Our results are for the specific case of switch statements in interpreters, and we have shown that indirect branch prediction is much more important for performance.

Interpreter opcode ordering (partitioning) is investigated by Lin and Chen [2008]. They present Java code arrangement for embedded NAND flash memory. They use profiling to relocate basic blocks within the interpreter to improve locality. Their technique was able to reduce cache misses by 96% in the best case. The instruction cache size of embedded devices is much smaller than modern workstation processors, so their work is not directly comparable to ours.

In Section 3.4 we described previous work showing that indirect branch prediction is a large factor in the performance of VM interpreters. Superinstructions (new VM instructions which consist of a sequence of existing instructions) and VM instruction replication are proposed methods to reduce indirect branch mispredictions on previous generation processors, which used simple branch target buffers to predict indirect branches. Our results show that where a modern processor has a two-level indirect branch predictor, the order and placement of the case statements for VM instructions can also have a significant impact on indirect branch prediction.

In Section 3.2.6 we described the work establishing the effect of linking order as a source of measurement bias. Measurement bias was attributed to link order as an alignment issue, which affected cache performance. Branch prediction was suggested as a possible cause. Our work in this chapter investigates code positioning for interpreters, and shows indirect branch mispredictions to be the most important factor in performance.

4.10 CONCLUSIONS

We have shown that interpreter opcode ordering can have a significant impact on the performance of VM interpreters. Existing work in this area has aimed at improving locality to reduce instruction cache misses. We have implemented two existing algorithms for ordering, and shown that in our experiments instruction cache misses do not have a major impact on performance.

To better understand the effect of opcode ordering on execution time we implemented a metaheuristic (Monte Carlo) to generate better orderings than can be found with frequency-based approaches. By analysing the resulting order-

ings, we found that—contrary to previous assumptions—the main impact of opcode re-ordering is from its effect on indirect branch prediction on modern processors.

We also implemented two feedback-directed search techniques and provided results. We have demonstrated our conclusions through extensive experiments on the interpreters of Lua, Python 2.6 and Python 3 on the Intel Core and Nehalem architectures.

As a practical implication of the work in this chapter, we suggest that opcode implementations should be statically ordered according to the Pareto optimal found on a representative training sample.

In the next chapter we explore an architecture-aware optimization that uses the knowledge of the indirect branch predictor internals to improve indirect branch prediction for general programs.

IMPROVING BRANCH PREDICTION FOR INDIRECT JUMPS AND CALLS

5.1 INTRODUCTION

In Chapter 1 we introduced indirect jumping, which is used to implement common programming constructs such as switch statements, virtual function calls and calls through function pointers. The performance of programs that make a lot of use of these features can depend heavily on the underlying hardware's ability to correctly predict the target address of the indirect branch. As we described in Section 2.5.4, predicting the target of an indirect branch is often more difficult than conditional branch prediction, because indirect branches can have many different targets.

In Section 2.5.4, we explained that the simplest type of indirect branch predictor just uses the branch target buffer (BTB), which when used with indirect branches will simply predict the indirect branch will jump to the same target as last time. We also described that for polymorphic indirect branches, prediction accuracy is poor with the BTB approach, so researchers have adapted the idea of *two-level branch predictors* (Figure 2.3) to indirect branches. So instead of having only a single branch prediction entry per branch, a two-level predictor uses both the address of the branch itself and a history of recent branch outcomes to index a table and find a prediction. With indirect branches, the history of recent branches usually consists of some bits selected from the addresses of recent indirect branch targets, combined together into a single history register.

A limitation of two-level indirect branch predictors, however, is that it is only possible to select a subset of the bits from each indirect branch target. For example, the indirect branch predictor in the Intel Pentium M processor selects the lower six bits of the indirect branch target, and hashes them into the branch history register [Uzelac and Milenkovic, 2009]. A problem is that if more than one target has the same pattern in those bits, then the predictor will not be able to distinguish between them. In this case, the predictor will have less contextual information to make the prediction and the likely outcome is poorer prediction accuracy.

In the previous chapter, we posited that the effect of opcode ordering on indirect branch prediction was as a result of the *path information register* being updated with the six least significant bits of the indirect branch target address. In this chapter we present two compiler-based software techniques to improve the predictability of indirect branches that are used to implement switch statements. Both techniques have the goal of ensuring that different targets of the switch branch have bit patterns that can be used to distinguish the branches in the indirect branch predictor. This chapter makes the following contributions.

- We show that there is an opportunity to reduce collisions in the indirect branch predictor using compiler techniques.
- We present a solution based on NOP insertion and describe the problem formally, and present heuristic solutions. We use simulations to determine the effectiveness of the heuristics.
- We present a second solution based on re-ordering the cases inside the switch statement and present heuristic solutions.
- We present a hybrid of both these techniques.
- We implement both strategies in an assembly optimizer tool, and evaluate them using both simulation and real hardware.
- We provide preliminary experiments that show these techniques can also be applied to improving branch prediction accuracy for virtual function calls.

5.2 BACKGROUND

Programming language switch (or case) statements can be implemented using different techniques, depending on the number of cases and the distribution of the switch constant values. For small numbers of cases, a sequence (or tree) of conditional branches suffice. Even for larger numbers of cases, a binary tree of branches can work well. However, in general the most efficient switch code can be generated for large number of cases when the *density* of the switch constant values is high. The density is high if the constant values fall within a fairly narrow range, and there are few gaps between the constants. In this situation, the compiler can create a jump table, where the table contains the addresses of the code that implements each case, and the table is indexed by the switch

value. Jump tables are highly efficient, and require only a table lookup and an indirect jump to implement a switch statement. They are so efficient that GCC generates a jump table for switch statements with as low as 10% density when optimizing for speed [Sayle, 2008].

In order to correctly predict the indirect branches used to implement switch statements and other constructs such as virtual functions and function pointers, researchers and computer architects have developed increasingly sophisticated indirect branch predictors. As we mentioned in Section 2.5.4, Driesen and Hölzle [1998a] adapted two-level conditional branch correlation techniques to indirect branches. Kalamatianos and Kaeli [1999] showed that correlation with previous branches using data compression techniques could further improve accuracy. As we described in Section 3.5, Kim et al. [2009] propose a low-cost mechanism to adapt an existing two-level conditional branch predictor to indirect branches.

In recent years, two-level indirect branch predictors have started to appear in real processors. The earliest example we are aware of is the Intel Pentium M [Gochman et al., 2003] which was introduced in 2003. More recent examples include the AMD Barcelona processor [Advanced Micro Devices, 2008], and various Intel Core 2 processors.

In Section 2.5.5, we stated that effective branch predictors are so important for performance that they are a competitive edge for CPU manufacturers, so their specific behavior is seldom publicly documented, yet fortunately there have been attempts to reverse-engineer the exact details of branch predictors [Milenkovic et al., 2002]. Uzelac and Milenkovic [2009] provide experiment flows and microbenchmarks for reverse-engineering cache-like branch predictor structures. In particular, they provide details of the branch predictor structures of the Intel Pentium M, including the indirect branch target buffer (iBTB). Of interest to us, they find that the Pentium M takes the six lower bits of the address of the branch target (Figure 5.1), and hashes them into a branch history register (called the *path information register* (PIR) by Intel). This PIR is used in the prediction of future branches by the Pentium M.

The PIR in the Pentium M is a fifteen bit register, which is updated by both taken conditional branches and indirect branches as follows:

$$\begin{aligned} \text{PIR}[14 : 0] &= (\text{PIR}[12 : 0] \ll 2) \text{ xor} \\ &\begin{cases} \text{IP}[18 : 4] & \text{if cbt} \\ \text{IP}[18 : 10] \text{ concat TA}[5 : 0] & \text{if ibt,} \end{cases} \end{aligned}$$

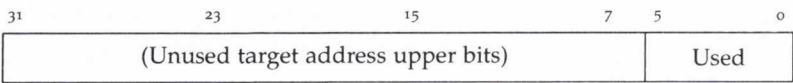


Figure 5.1: Bits of the indirect branch target address used to update the path information register in Intel processors.

where *cbt* is a taken conditional branch, *ibt* is a taken indirect branch, *IP* is the memory address of the branch instruction, and *TA* is the target address of an indirect branch. In other words, when a new conditional or indirect branch is executed, the existing value of the PIR is shifted left by two places, and a new value is *xor*'d into the PIR. In the case of a conditional branch, this new value is simply fifteen bits of the address of the branch in memory. In the case of an indirect branch, nine bits are taken from the address of the branch, and six least significant bits from the address of the branch target. It has been reported that the Core microarchitecture includes the same indirect branch predictor [Kim et al., 2009].

Given that indirect branch predictors use only a selection of bits from the branch target when updating their history register, the effectiveness of two-level indirect branch prediction schemes depends on the bit pattern being different for the common targets. The actual addresses at which the branch targets appear in memory depends on choices made by the compiler and linker. In general, compilers and linkers make no attempt to optimize for the indirect branch predictor. In some systems the locations of the branch targets in memory are simply arbitrary, and whether the chosen bits are distinct for different branch targets is a matter of luck.

Sometimes compilers actually make the situation worse, by aligning branch targets for supposed cache gains — if the target is at the lowest address, then more code may end up fitting on a cache line. However, this leads to less information being available to the branch predictor hardware, and consequentially, worse predictions. Listing 5.1 shows an example extract of assembly code compiled by GCC for a switch statement. The branch targets of cases in the switch statement are labels `.L4` and `.L5`. The compiler has inserted directives (the `.p2aligns`, whose first argument is the number of lower bits) to align these branch targets on a 16-byte boundary, which will ensure that the lower four bits always have the value zero. If the indirect branch predictor uses these bits, then they will be useless for distinguishing between different branch targets. In rare cases, aliasing can be helpful to the indirect branch predictor, but this is unlikely.

```

        .p2align 4,,10
        .p2align 3
#NOP bytecode implementation
.L4:
        addq    $8, %rsi        #, ip
        jmp     .L41
        .p2align 4,,10
        .p2align 3
#LOAD IMMEDIATE bytecode implementation
.L5:
        movl    4(%rsi), %eax
        addq    $8, %rsi        #, ip
        movl    %eax, (%r8)
        addq    $4, %r8         #, sp
        jmp     .L41

```

Listing 5.1: Typical jump-table assembly compiled by GCC for our experimental interpreter. The jump targets will be 16-byte aligned by the linker, which means that the lower four bits of the address will be zero.

5.3 NOP INSERTION

5.3.1 Idea

Our first technique to modify indirect branch target addresses is to insert NOP instructions directly before a target instruction in order to move its memory address to a higher memory location that will not conflict with other targets.

5.3.2 Problem Formulation

It is clear that we can only have a bijective mapping if the number of targets is less than or equal to the number of ‘buckets’.

Definition 5.3.1 (Unique Target Address Keys). Given a positive integer k , a positive integer t , and a sequence of distinct nonnegative integers $X_1, \dots, X_n$ is there a sequence of nonnegative integers $D_1, \dots, D_n$ such that there does not exist $1 \leq i < j \leq n$ such that $X_i + D_i + \dots + D_1 \equiv X_j + D_j + D_i + \dots + D_1 \pmod{k}$ where the sum of $D_1, \dots, D_n$ is less than t .

CONJECTURE 5.3.2. *Deciding if there is a solution to the NOP insertion problem less than t is NP-complete (where t is the total number of inserted NOPs).*

Unfortunately a proof for this conjecture is not known to exist. A ‘brute-force’ algorithm to evaluate all possible NOP insertion permutations will have

$O(\binom{t+n-1}{n-1})$ complexity, where t is the total ‘budget’ and n is the number of target positions. Enumerating all the possible solutions in this manner is the problem of integer compositions. An algorithm for generating integer compositions is described by Knuth [2005].

5.3.3 *Heuristics*

On the assumption that the problem is NP-hard, we now present three heuristics that perform well, and we show simulation results.

5.3.3.1 *Greedy*

The simplest heuristic, shown in Algorithm 5.1, is a greedy algorithm. It simply iterates through the addresses from lowest to highest (therefore making updated address calculation easy), and inserts NOPS until the current address no longer conflicts with previously processed addresses. The average case for this algorithm can be analyzed in the same manner as a series of unsuccessful searches in linear probing, and an aggregation of the following formula from Knuth [1998] will provide an average for the number of NOPS inserted:

$$C_N \approx \frac{1}{2} \left(1 + \left(\frac{1}{1 - \frac{N}{M}} \right)^2 \right),$$

where N is the number of targets already considered and M is the total number of targets. The worst case of number of NOPS inserted is $M(M+1)/2$.

5.3.3.2 *Minimize conflicts*

The greedy algorithm can be improved upon: algorithm 5.2 also iterates through target addresses from lowest to highest and it inserts enough NOPS to minimize the number of all conflicting targets at each stage while being cautious about how many NOPS it inserts by minimizing (conflicts $\times$ number of NOPS).

5.3.3.3 *Maximize ‘clean distance’*

A third heuristic is shown in Algorithm 5.3. This algorithm will also iterate through targets from lowest to highest, and it will choose the number of NOPS to insert that maximizes the distance from the current target to the next target that

Algorithm 5.1 Greedily insert NOPs

```

1: Input: Target memory addresses ( $T$ ).
2: Output: New target memory addresses ( $T'$ ).
3:  $\text{uniqueKeys} \leftarrow \emptyset$ 
4:  $\text{offset} \leftarrow 0$ 
5: for all target addresses  $T_i$  in  $T$  do
6:    $T'_i \leftarrow T_i + \text{offset}$ 
7:   while  $T'_i \pmod{2^6}$  in  $\text{uniqueKeys}$  do
8:      $\text{insertNop}()$ 
9:      $\text{offset} \leftarrow \text{offset} + 1$ 
10:  end while
11:   $T'_i \leftarrow T_i + \text{offset}$ 
12:   $\text{uniqueKeys} \leftarrow \text{uniqueKeys} \cup \{T'_i \pmod{2^6}\}$ 
13: end for
14: return  $T'$ 

```

Algorithm 5.2 Insert NOPs, minimizing the conflicts at each point.

```

1: Input: Target memory addresses ( $T$ ).
2: Output: New target memory addresses ( $T'$ ).
3:  $\text{uniqueKeys} \leftarrow \emptyset$ 
4:  $\text{offset} \leftarrow 0$ 
5: for all target addresses  $T_i$  in  $T$  do
6:    $\text{smallest} \leftarrow 0$ 
7:    $\text{bestNN} \leftarrow 0$ 
8:   for  $nn = 0 \rightarrow (2^6 - 1)$  do
9:      $\text{nextAddrs} \leftarrow \text{getNextAddresses}(\text{offset}, nn, T)$ 
10:     $\text{conflicts} = \text{countConflicts}(\text{nextAddrs}, \text{uniqueKeys})$ 
11:    if  $T_i + \text{offset} + nn \pmod{2^6}$  in  $\text{uniqueKeys}$  and  $(\text{conflicts} \times nn) <$ 
         $\text{smallest}$  then
12:       $\text{bestNN} \leftarrow nn$ 
13:       $\text{smallest} \leftarrow \text{conflicts} \times nn$ 
14:    end if
15:  end for
16:   $\text{offset} += \text{bestNN}$ 
17:   $T'_i \leftarrow T_i + \text{offset}$ 
18:   $\text{uniqueKeys} \leftarrow \text{uniqueKeys} \cup \{T'_i \pmod{2^6}\}$ 
19: end for
20: return  $T'$ 

```

	Heuristic	Average number of NOPs
	Greedy	137
	Minimize conflicts & greedy (pick best)	128
	Maximize 'clean distance'	127
	Maximize 'clean distance' & greedy (pick best)	116

Table 5.1: Simulated average NOP insertions for various heuristics for 60 target addresses and modulo 64.

has a conflict, while also avoiding aggressively inserting NOPs by maximizing distance/NOPs.

Algorithm 5.3 Insert NOPs, maximizing the distance until a conflict occurs with the offsetting at each point.

```

1: Input: Target memory addresses ( $T$ ).
2: Output: New target memory addresses ( $T'$ ).
3:  $\text{uniqueKeys} \leftarrow \emptyset$ 
4:  $\text{offset} \leftarrow 0$ 
5: for all target addresses  $T_i$  in  $T$  do
6:    $\text{greatest} \leftarrow 0$ 
7:    $\text{bestNN} \leftarrow 0$ 
8:   for  $nn = 0 \rightarrow (2^6 - 1)$  do
9:      $\text{nextAddr} \leftarrow \text{getNextAddresses}(\text{offset}, nn, T)$ 
10:     $\text{dist} = \text{getCleanDist}(\text{nextAddr}, \text{uniqueKeys})$ 
11:    if  $T_i + \text{offset} + nn \pmod{2^6}$  not in  $\text{uniqueKeys}$  and  $\frac{\text{dist}}{nn+1} > \text{greatest}$ 
12:      then
13:         $\text{bestNN} \leftarrow nn$ 
14:         $\text{greatest} \leftarrow \frac{\text{dist}}{nn+1}$ 
15:      end if
16:    end for
17:     $\text{offset} += \text{bestNN}$ 
18:     $T'_i \leftarrow T_i + \text{offset}$ 
19:     $\text{uniqueKeys} \leftarrow \text{uniqueKeys} \cup \{T'_i \pmod{2^6}\}$ 
20: end for
21: return  $T'$ 

```

5.3.4 Simulation Results

To compare three heuristics (and variants thereof), we simulated input target addresses using random numbers. First, 60 random variables α with a uniform

distribution within the interval $(5, 29)^1$ are generated. These are used as the sizes to generate a sequence of memory addresses. This is repeated 1000 times for each heuristic that is being evaluated.

Table 5.1 shows simulated results for four of the best heuristics. The second and fourth rows show heuristics that also run the greedy algorithm for each set of addresses and choose the answer with the smallest number of NOPs. The best heuristic, with an average of 116 NOPs inserted, is the combination of maximize ‘clean distance’ heuristic and greedy.

5.4 RE-ORDERING

5.4.1 Idea

It may be wasteful to insert NOP instructions when memory size is severely constrained, so another technique to change the memory addresses of the indirect branch targets (when the targets are placed sequentially in memory) is to re-order the code blocks.

5.4.2 Problem Formulation

Definition 5.4.1 (Unique Target Address Keys Ordering where $n = m$). Given a sequence of positive integers $S_1, \dots, S_n$, is there a permutation of S , such that for $\forall x \in X$, where $x_i = S_0 + S_1 + \dots + S_{i-1}$, every x_i is unique mod k .

LEMMA 5.4.2. *For $n < m$, there is not always a permutation that results in no collisions in the hash values.*

PROOF. If the address of two cases p_i and p_{i+1} differ by a value v , such that $v \bmod k \equiv 0$, then both will hash to the same number. This happens when $\text{length}(p_i) \bmod k \equiv 0$. If we have more than one such case, then we will always have at least one collision in the hashed values. $\square$

LEMMA 5.4.3. *The decision problem for unique targets when $n = m$ is NP-hard.*

PROOF. To ensure a particular x_i is unique mod k , we must choose a subset S_j of $S_1, \dots, S_n$ such that its sum mod k equals a particular integer. This subproblem is the NP-complete problem of subset-sum on a finite field [Nathanson, 1996]. $\square$

¹ These sizes are representative of test programs we looked at.

5.4.3 Heuristics

Since we also assume the problem of re-ordering in the general case is NP-hard, we present two heuristics to order the blocks in the program. These heuristics do not attempt to remove all conflicts, but to reduce them.

The simplest heuristic to reduce the number of conflicts is a 'pair-swap'. We perform a single pass over all the triplets of target addresses (pairs of 'blocks'²) and swap them if it will mean fewer conflicts with target addresses already processed. Algorithm 5.4 shows this heuristic.

Algorithm 5.4 Re-order 'blocks' using a 'pair-swap'.

```

1: Input: Target memory addresses ( $T$ ).
2: Output: New target memory addresses ( $T'$ ).
3:  $\text{blocks} \leftarrow \emptyset$ 
4: for all pairs of target addresses  $T_i, T_{i+1}$  in  $T$  do
5:    $\text{blocks} \leftarrow \text{blocks} \cup \{T_i, T_{i+1}\}$ 
6: end for
7:  $\text{offset} \leftarrow 0$ 
8:  $T'_0 \leftarrow \text{offset}$ 
9: for all pairs of blocks  $b_i, b_{i+1}$  in  $\text{blocks}$  do
10:   if  $\text{size}(b_i) \neq \text{size}(b_{i+1})$  then
11:     if  $\text{offset} + \text{size}(b_i) \pmod{2^6}$  in  $\text{uniqueKeys}$  and  $\text{offset} + \text{size}(b_{i+1}) \pmod{2^6}$  not in  $\text{uniqueKeys}$  then
12:        $\text{swapBlocks}(b_i, b_{i+1})$  ▷ swap the two blocks in memory.
13:     end if
14:   end if
15:    $\text{offset} \leftarrow \text{offset} + \text{size}(b_i)$ 
16:    $T'_{i+1} \leftarrow \text{offset}$ 
17: end for
18: return  $T'$ 

```

A second, more effective heuristic to reduce conflicts is to greedily schedule them: Select the next block that causes no conflicts and place it after the previously scheduled one. If there is no block that does not cause a conflict, then just pick the first block in the list. Algorithm 5.5 shows this heuristic.

For these heuristics, the worst case would be a new ordering where all targets conflict with each other. This could happen if the input consists of blocks that are all the size of the 'buckets'.

² These are not basic blocks in the CFG sense, since there may be arbitrary control flow between two branch targets.

Algorithm 5.5 Re-order ‘blocks’ using a greedy fit.

```

1: Input: Target memory addresses ( $T$ ).
2: Output: New target memory addresses ( $T'$ ).
3:  $\text{blocks} \leftarrow \emptyset$ 
4: for all pairs of target addresses  $T_i, T_{i+1}$  in  $T$  do
5:    $\text{blocks} \leftarrow \text{blocks} \cup \{T_i, T_{i+1}\}$ 
6: end for
7:  $\text{offset} \leftarrow 0$ 
8:  $T'_0 \leftarrow \text{offset}$ 
9: while  $\text{blocks} \neq \emptyset$  do
10:   $\text{choice} \leftarrow \text{blocks.first}$ 
11:  for all  $b$  in  $\text{blocks}$  do
12:    if  $\text{offset} + \text{size}(b) \pmod{2^6}$  not in  $\text{uniqueKeys}$  then
13:       $\text{choice} \leftarrow b$ 
14:    end if
15:  end for
16:   $\text{offset} \leftarrow \text{offset} + \text{size}(\text{choice})$ 
17:   $i \leftarrow i + 1$ 
18:   $T'_i \leftarrow \text{offset}$ 
19: end while
20: return  $T'$ 

```

5.5 HYBRID

Re-ordering attempts to reduce conflicts by only moving blocks. A more sophisticated approach is to combine that with NOP insertion to get the best of both. Blocks are re-ordered to reduce collisions, and NOPs inserted if no block is the required size. Algorithm 5.6 shows this hybrid algorithm.

5.6 PERFORMANCE EVALUATION

5.6.1 *Implementation*

We implemented our two techniques using the assembly language optimization framework developed by Hundt et al. [2011]. For NOP insertion, we implemented the greedy algorithm (Algorithm 5.1) for simplicity. For re-ordering, we implemented Algorithm 5.5. In some cases, the compiler (GCC) inserts `.p2align` assembler directives before indirect branch targets (as shown in Figure 5.1, aligning them to 16-byte boundaries – this is especially detrimental to the performance of the branch predictor, as only two unique bits of the tar-

Algorithm 5.6 Hybrid: re-order ‘blocks’ using a greedy fit, insert nops if necessary.

```

1: Input: Target memory addresses ( $T$ ).
2: Output: New target memory addresses ( $T'$ ).
3:  $\text{blocks} \leftarrow \emptyset$ 
4: for all pairs of target addresses  $T_i, T_{i+1}$  in  $T$  do
5:    $\text{blocks} \leftarrow \text{blocks} \cup \{T_i, T_{i+1}\}$ 
6: end for
7:  $\text{offset} \leftarrow 0$ 
8:  $T'_0 \leftarrow \text{offset}$ 
9: while  $\text{blocks} \neq \emptyset$  do
10:    $\text{choice} \leftarrow \text{nil}$ 
11:    $\text{nopsNeeded} \leftarrow 0$ 
12:   for all  $b$  in  $\text{blocks}$  do
13:     if  $\text{offset} + b_{\text{size}} \pmod{2^6}$  not in  $\text{uniqueKeys}$  then
14:        $\text{choice} \leftarrow b$ 
15:     end if
16:   end for
17:   if  $\text{choice} = \text{nil}$  then
18:      $\text{nopsNeeded} \leftarrow \infty$ 
19:     for all  $b$  in  $\text{blocks}$  do
20:       if  $\text{countNopsNeeded}(b) < \text{nopsNeeded}$  then
21:          $\text{choice} \leftarrow b$ 
22:          $\text{nopsNeeded} \leftarrow \text{countNopsNeeded}(\text{choice})$ 
23:       end if
24:     end for
25:   end if
26:   for  $i=1 \rightarrow \text{nopsNeeded}$  do
27:      $\text{insertNop}()$ 
28:   end for
29:    $\text{offset} \leftarrow \text{offset} + \text{choice.size} + \text{nopsNeeded}$ 
30:    $i \leftarrow i + 1$ 
31:    $T'_i \leftarrow \text{offset}$ 
32: end while
33: return  $T'$ 

```

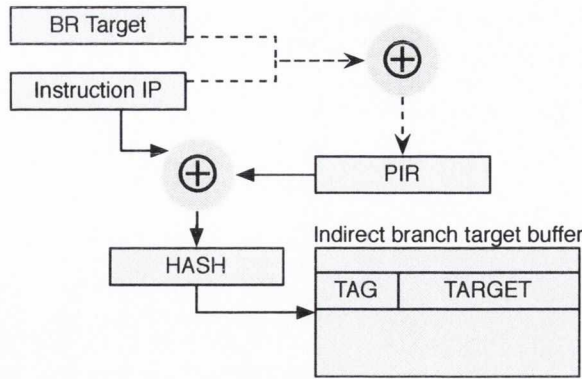


Figure 5.2: Our Pin model for the indirect branch target buffer. Dashed lines represent the path information register update.

get address would be used in the predictor history. In our implementation we remove these `.p2align` before the indirect branch targets.

For re-ordering, the code between target labels may include arbitrary control-flow, so we make a simplification, and treat the code between any two target labels as a ‘block’ to be moved (unless there is a fall-through path from one block to another, in which case we coalesce the two).

A small adaption was required when inserting NOPs for the problem cases where $N > M$, that is, where the number of branch targets is greater than the number of ‘buckets’ (64). If $N > M$, for the first 64 targets we insert NOPs to make each unique, for the next 64 we insert nops until each target only conflicts with one other target, etc.

5.6.2 Pin Simulation

We created a simulation of the indirect branch target buffer and the path information register using the Pin dynamic binary instrumentation framework [Luk et al., 2005]. Figure 5.2 shows our model. We then evaluated our techniques on the SPEC2006 C benchmarks [Standard Performance Evaluation Corporation, 2011]. Figure 5.4 shows the misprediction rate reductions of the programs with NOPs inserted to increase target uniqueness, the programs using re-ordering, and the programs using the hybrid technique compared to the baseline (100%). Figure 5.3 shows the actual misprediction rate for those same programs. The majority of the benchmarks show reductions in the misprediction rate, some of them large (>90% reduction for the *mcf* benchmark). The worst performing case was re-ordering targets for the *milc* benchmark, where the misprediction

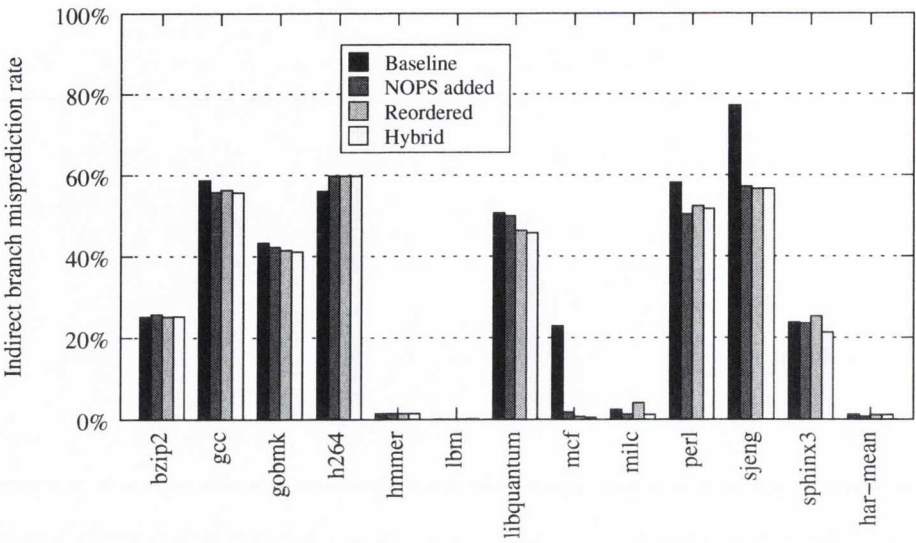


Figure 5.3: Simulated iBTB misprediction rate for baseline vs. NOP insertion vs. re-ordering. vs. hybrid.

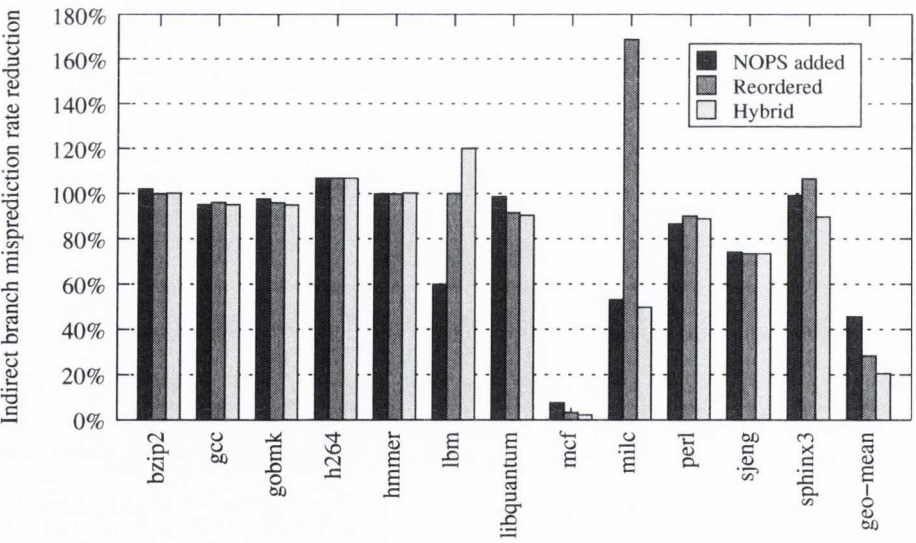


Figure 5.4: Simulated iBTB misprediction rate reduction for three techniques compared to baseline. (Smaller is better.)

rate was increased by over 60%, however, in this case the original misprediction rate was very low to begin with, as Figure 5.3 shows.

We did not observe a significant change in branch mispredictions from running our techniques on the SPEC2006 benchmarks on real hardware. We believe this is because on our test machine, conditional branches share the path information register with the iBTB. This is then compounded by the low ratio of indirect branches to conditional branches in those programs. For example, the *gcc* benchmark, which has a significant number of indirect branches, has 28 times more conditional branches than indirect branches. Given that taken conditional branches update the branch history, and assuming a significant number of the conditional branches are taken, and the history depth is ≈ 8 , updates to the history by indirect branches will be ‘flushed’ by the time another indirect branch is reached. In the next subsection, we will look at programs that make intensive use of indirect branches.

5.6.3 *Indirect-Branch-Heavy Programs*

As we stated in Chapter 1, an important use of multiway branches is in interpreter implementation. We therefore evaluate our techniques on three interpreters: a simple interpreter written in C; Lua, a switch-based dynamic language interpreter; and JamVM, a Java language interpreter (switch implementation).

Our first interpreter is a simple interpreter we designed to experiment with optimizations (Listing 5.2). It has a full set of opcodes, but only a few are exercised in our test bytecode program, which is just a nested loop with arithmetic in the inner loop. The results of applying our code transformation technique on our simple interpreter are enumerated in Table 5.2. These results are for the Intel Core 2. We see a significant decrease in the number of branch mispredictions, and a pronounced reduction in execution time for all three techniques. The re-ordering and hybrid show a 6% increase in level 1 instruction cache reads, but have fewer misses than either the baseline or the version with NOPS inserted.

The hybrid algorithm shows more indirect branch mispredictions than the other techniques; this is likely due to the other techniques producing addresses that are more fortuitous for that benchmark, but the overall effect is not as significant as the change from the baseline. The instruction cache miss rates are tiny ($< 0.001\%$), so we do not infer anything from the small changes in misses.

```
enum {OP_LDI, /* ... */};
typedef struct inst {
    int32_t op, arg;
} inst;
void engine() {
    inst program[]={ { OP_LDI,    0 } /* ... */ };
    int *sp=s;
    inst *ip = program;
    while(1) {
        switch(ip->op) {
            case OP_LDI:
                *sp+=ip->arg;
                ++ip;
                break;
                /* ... */
        }
    }
}
```

Listing 5.2: Our simple interpreter.

	Event	Baseline	NOPs inserted	Re-ordered	Hybrid
	Branches executed	4,400,625,607	4,500,244,910	4,499,465,407	4,501,921,449
	Indirect branches	1,396,710,949	1,394,288,997	1,399,635,984	1,400,056,605
	Indirect branch mispredictions	399,755,351	22,221	82,269	88,948
	L1 Instruction cache reads	6,196,712,454	6,192,419,852	6,605,654,635	6,606,098,075
	L1 Instruction cache misses	43,864	32,038	12,755	15,209
	Execution time (seconds)	6.42	2.89	2.76	2.76

Table 5.2: Simple interpreter results.

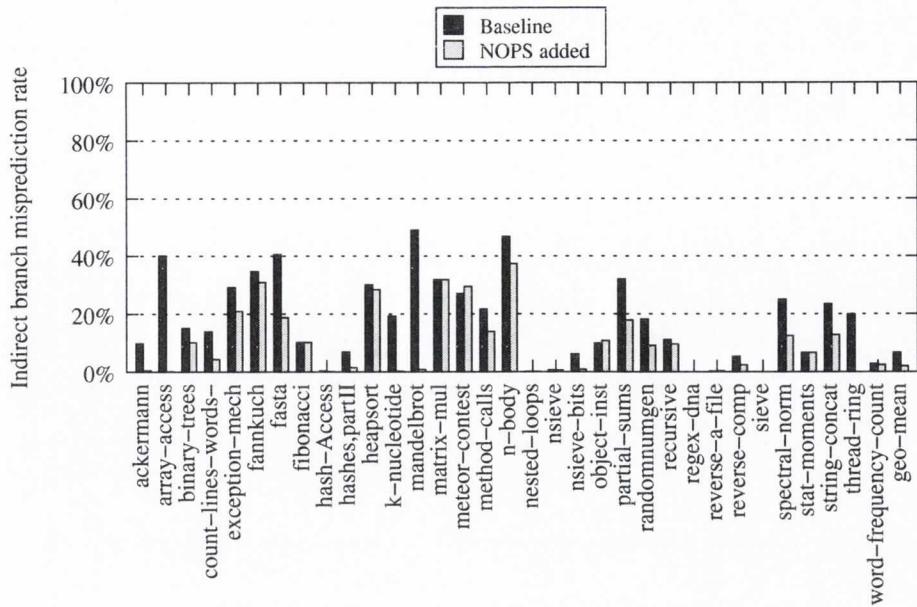


Figure 5.5: Lua misprediction rate for baseline vs. NOP insertion for a selection of benchmarks.

The second interpreter we evaluate is Lua, a dynamically typed language interpreter, which we described in Section 2.2.2. We evaluate both our techniques for Lua using a selection of benchmarks from the Computer Language Benchmarks Game [CLBG, 2011] and The Great Win32 Computer Language Shootout [GWCLS, 2008] on the Intel Core 2. Figure 5.5 shows the misprediction rate of the interpreter before and after performing the NOP insertion technique. Figure 5.6 shows speedups resulting from the reduction in indirect branch misses. Figure 5.7 shows the misprediction rate of the interpreter before and after performing re-ordering and Figure 5.8 shows the misprediction rate of the interpreter before and after the hybrid technique. The reduction in mispredictions compared to the baseline for the three techniques is shown in Figure 5.9. All three techniques lead to a substantial decrease in mispredictions for the majority of Lua benchmarks. Figure 5.10 shows the level 1 instruction cache results for Lua. The size of the instruction cache (32 KB) dwarfs the code size of the interpreter.

The third interpreter we investigate is the JamVM Java virtual machine interpreter we described in Section 2.2.4 [Lougher, 2011]. We compile the ‘switch’³ version of the interpreter by disabling the threaded code mode. We evaluated our techniques on this interpreter using benchmarks from the Java Grande Forum [Mathew et al., 1999]. Figure 5.11 shows the misprediction rates of the

³ We described switch based interpreters in Section 3.3.2.

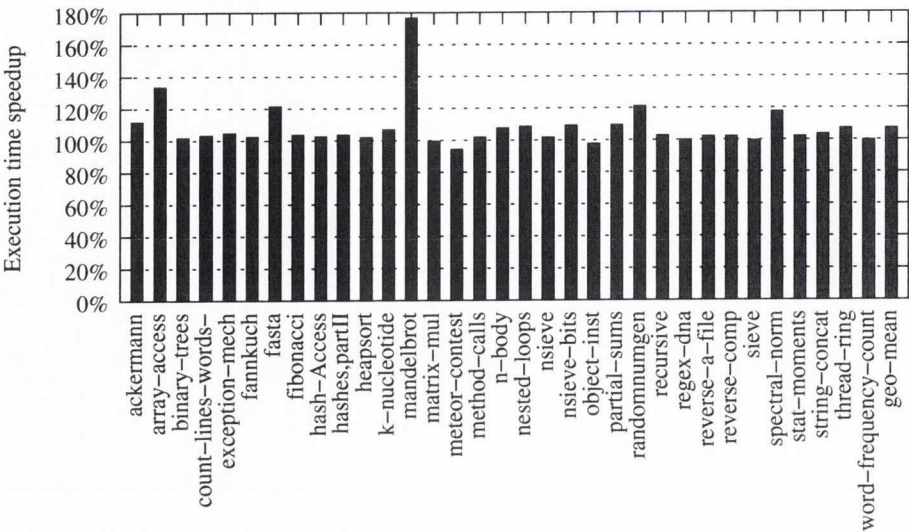


Figure 5.6: Lua speedups (due to fewer mispredictions) for the NOP insertion technique. Showing an average speedup of 7%. The *mandelbrot* benchmark achieves an almost 80% speedup.

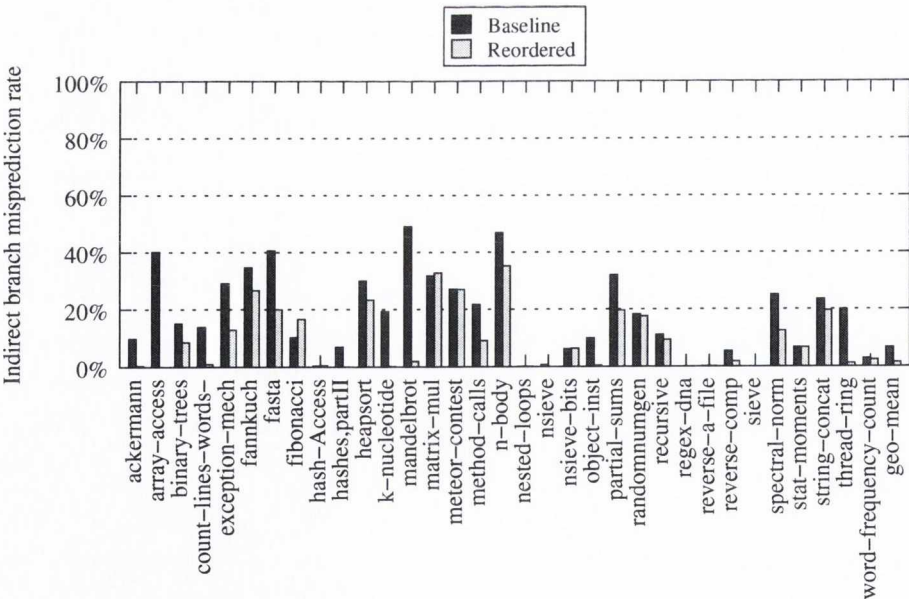


Figure 5.7: Lua misprediction rate for baseline vs. re-ordered indirect branch targets.

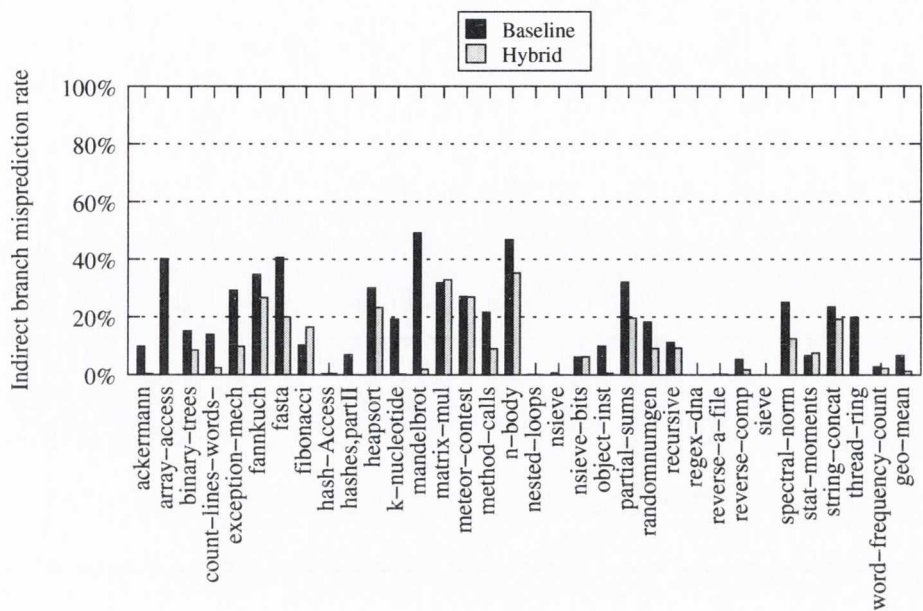


Figure 5.8: Lua misprediction rate for baseline vs. hybrid indirect branch targets.

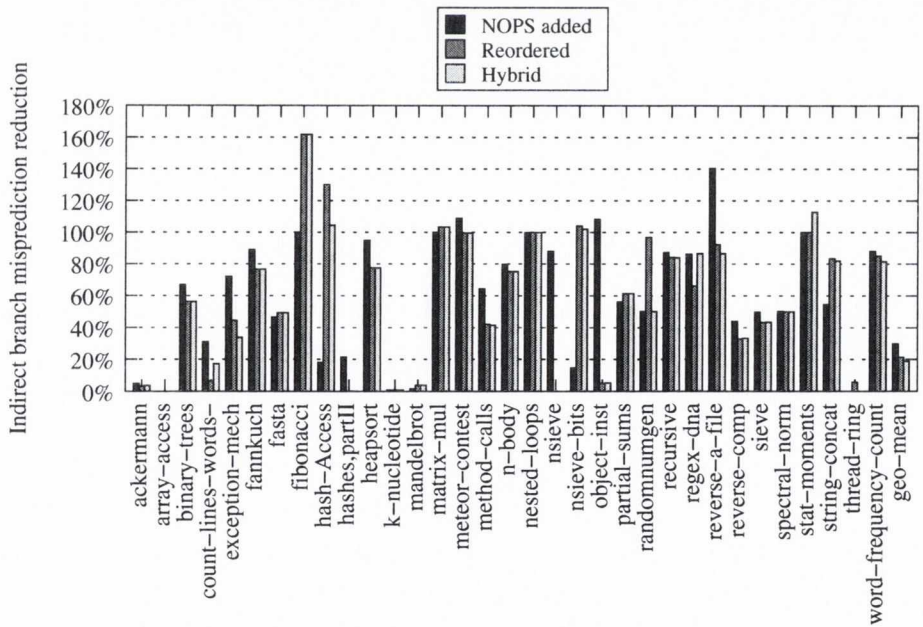


Figure 5.9: Lua indirect branch misprediction rate for the baseline vs. NOP insertion vs. re-ordering vs. hybrid for a selection of benchmarks. (Smaller is better).

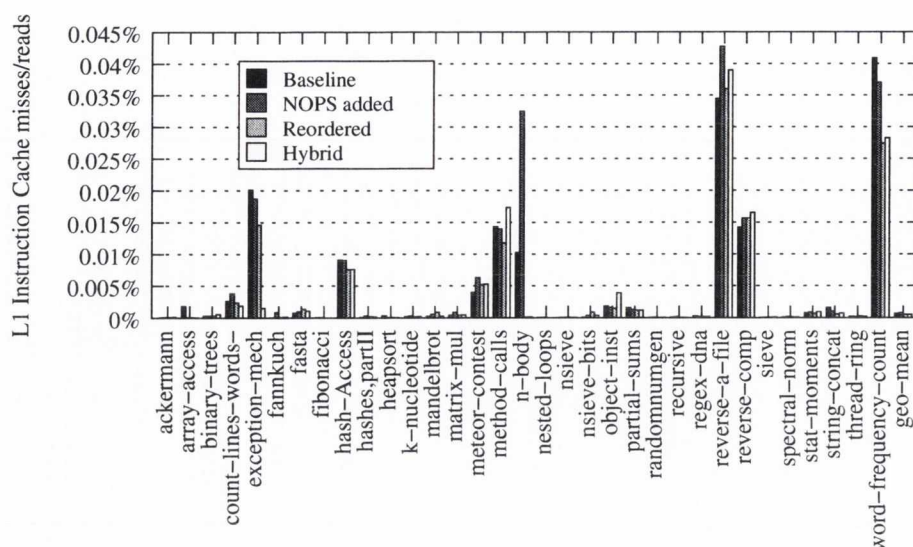


Figure 5.10: Lua level 1 instruction cache miss rate compared to baseline for the three techniques for a selection of benchmarks. (Smaller is better.)

baseline interpreter, the interpreter using NOP insertion, the interpreter using re-ordering and the interpreter using the hybrid technique on the Intel Core 2. The effect of reducing branch target address collisions is not as pronounced for this interpreter, perhaps because there are many more opcodes (>100), and consequentially many more indirect branch targets. Figure 5.12 shows speedups attained from the reduction in indirect branch misses. Figure 5.13 shows the level 1 instruction cache results for Java. These have a similar scale as the Lua instruction cache results.

5.6.4 C++ Programs

In addition to multiway branches, indirect branches are also used to implement indirect function calls. Function calls are used in programs that use function pointers where the target cannot be resolved statically at compile time. Indirect function calls are also used to implement run-time dispatching which is required for polymorphism in object-oriented languages.

For C++, the GCC compiler implements virtual method calls using dynamic dispatch with *virtual function tables*, which involve an indirect call instruction. Indeed, this may be one of the greatest motives for the addition of indirect branch prediction hardware to recent Intel and AMD CPUs.

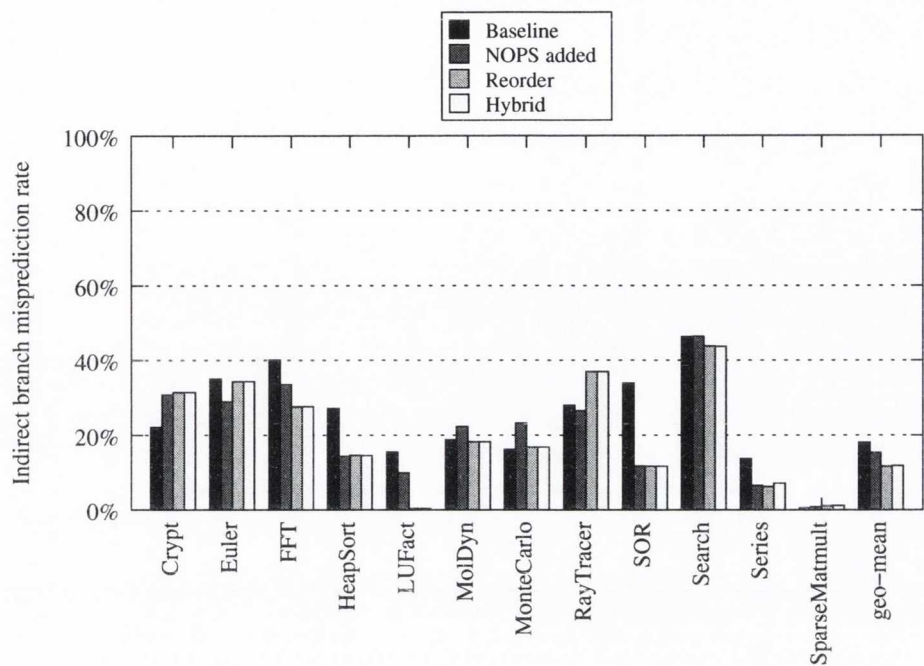


Figure 5.11: Java misprediction rate for baseline vs. NOP insertion vs. re-ordering for a selection of benchmarks.

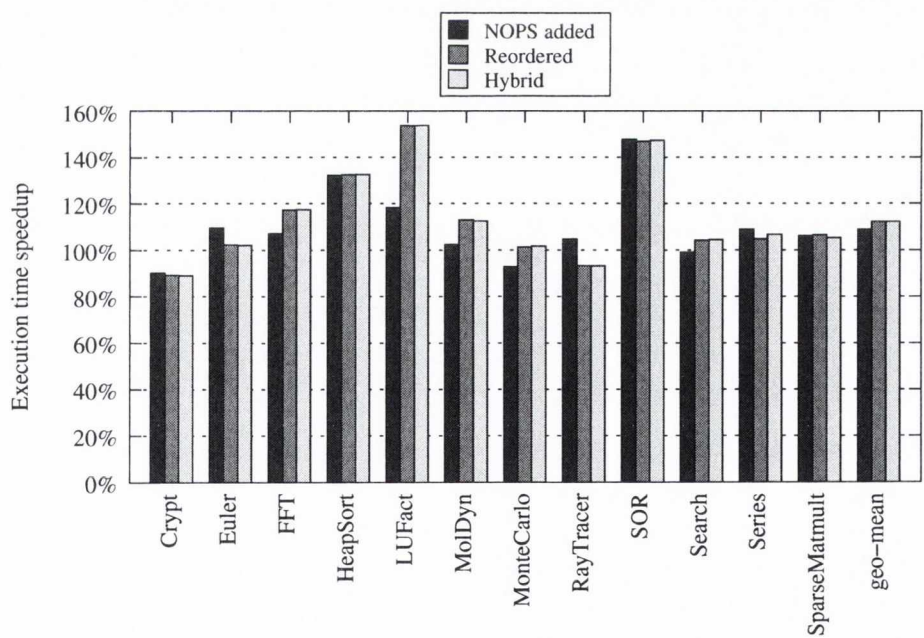


Figure 5.12: Java speedups for the three techniques compared to the baseline (Larger is better).

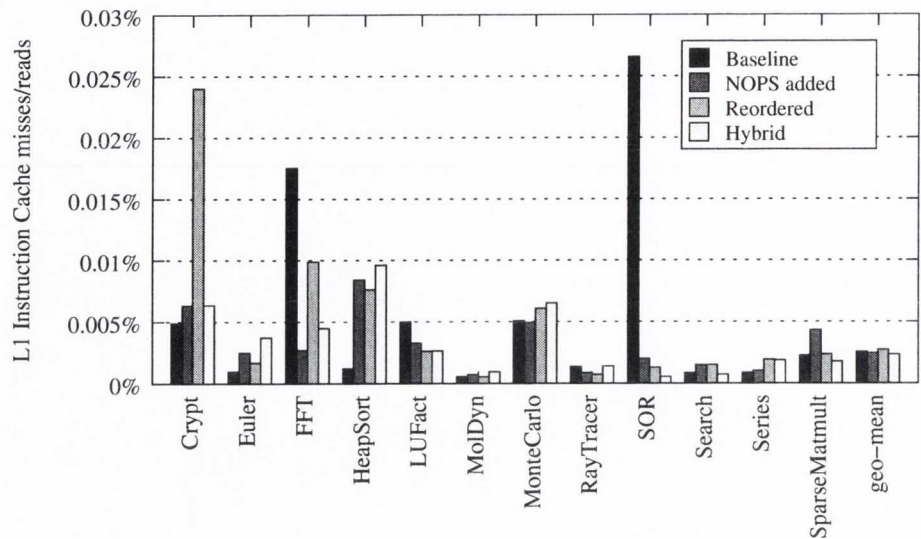


Figure 5.13: Java level 1 instruction cache miss rate for the baseline vs. NOP insertion vs. re-ordering vs. hybrid for a selection of benchmarks. (Smaller is better).

To apply our techniques to this use-case of indirect calls, we adapted our assembly language optimization passes to work with function branch targets. We parse the virtual tables in our optimization pass and determine the start labels of all virtual functions. We also include *thunks*⁴ as these are possible targets of indirect calls.

GCC puts functions into separate sections in the assembly file, and these are usually 16-byte aligned by the linker. In our first implementation, we took this constraint into account, and inserted enough NOP instructions to ensure each branch target had unique lower bits. However, we did not observe a significant decrease in indirect branch misses, so we changed the implementation to move all the virtual functions into one section before making targets unique.

To evaluate our optimization pass, we created a C++ version of our simple interpreter, using an abstract base `Instruction` class which has a virtual `execop` method that child classes override to implement the various opcodes. Thus the simple interpreter loop: `while (1){ (*ip)->execop(); }`

The results for our C++ simple interpreter are enumerated in Table 5.3. The baseline has a branch misprediction rate of 13%, and the version with NOPs inserted has a misprediction rate of 0.004%. Our re-ordering technique gives a misprediction rate of 0.006%. These are both much better than the baseline,

⁴ Thunks are an optimization for virtual function calls. A thunk is a ‘wrapper’ function that the compiler generates to optimize the typical use case of method lookup. In the usual case it will eliminate a memory fetch and add of an offset.

Event	Baseline	NOPs inserted	Re-ordered
Branches executed	4,293,992,704	4,301,493,720	4,300,948,941
Indirect branches	1,400,250,180	1,399,109,088	1,399,971,862
Indirect branch mispredictions	186,606,245	58,434	84,555
L1 Instruction cache reads	20,911,541,414	14,275,657,052	14,524,382,937
L1 Instruction cache misses	30,678	103,103	115,485
Execution time (seconds)	8.73	6.05	6.05

Table 5.3: Simple interpreter (C++) results.

which has a misprediction rate of 13%. Both techniques show a reduction in the number of L1 instruction cache reads — with fewer mispredictions, fewer instructions are executed, and consequently the instruction cache is accessed less.

5.6.5 Hardware Configuration

In this chapter we evaluate our optimization on the following system: Intel Core 2 Q6600 (Core Kentsfield) 2.4 GHz CPU with 6 GB memory, running Ubuntu GNU/Linux 11.04 with kernel version 2.6.38-8. GCC version 4.5.2.

Performance counters were collected using the *perf* tool (an interface to the Linux kernel perf events) and PAPI [Browne et al., 2000]. For indirect branch misses, we recorded the BR_IND_MISSP counter.

5.7 RELATED WORK

5.7.1 Reducing Branch Interference

In Section 3.2 we described the work of Chen and King [1999], who adjust addresses to reduce collisions in 2-bit counters for conditional branches. Similarly to us, they also add NOP instructions to perform the address adjustment. They present a constrained and a relaxed method. The constrained method will only insert nops following an unconditional branch (where they will never be executed) whereas the relaxed method will insert nops in locations where they may get executed by the processor. They also describe *branch classification*, which maps branches with the same history pattern to the same counter. Our work similarly inserts NOP instructions to move branch addresses, but we in-

stead consider indirect branches, and we also present a technique using code re-ordering.

Jiménez [2005] build upon that work, describing *pattern history table partitioning*, a feedback directed technique that moves conditional branch addresses to avoid destructive interference. We have not considered any feedback techniques for moving branch addresses, but in Chapter 4 we did devise a feedback technique for re-ordering interpreter opcode implementations. Another difference is that in their work they attempt to avoid violating alignment heuristics in order to not affect cache performance. We don't attempt to obey alignment constraints; indeed, we show that they can be a significant barrier to performance in certain circumstances.

5.7.2 Code Placement for Branches

Krall [1994] uses profiling to collect information about conditional branches and then uses code replication to reduce the branch misprediction rate. The work targets static branch prediction, whereas we are concerned with dynamic indirect branches, and do not investigate profiling methods.

Uh and Whalley [1999] describe a transformation that coalesces conditional branches into indirect jumps. They show that this transformation to indirect jumps can improve prediction. We are not concerned with how indirect jumps are generated, we always try to maximize their predictability.

Yang et al. [2002] re-order branches using profile data to reduce the number of conditional branches executed by a program. Our work is concerned with indirect branches whose execution can generally never be avoided by code rearrangement.

There has been much research on code rearrangement to improve cache behavior. Related to branches, Calder and Grunwald [1994a] present a profile-based technique that 'aligns' branches: it will minimize the number of taken branches, therefore ensuring frequently executed code is closer together, thereby improving cache performance.

5.7.3 Target Predictability for High-Level Languages

In Chapter 4, we demonstrated that re-ordering the cases of a switch statement has a significant impact on indirect branch prediction for interpreters. We hypothesized this to be due in part to aliasing in the indirect branch predictor.

The work in this chapter tests that hypothesis by directly reducing the amount of aliasing in the predictor, and we see that the prediction rates do increase.

Li et al. [2005] improve the target predictability of Java, particularly for virtual methods by proposing a rehashable branch target buffer that dynamically adapts branch target storage for polymorphic branches. They show improvements over traditional techniques, including target caches. In contrast, our work is focused on improving the performance of such code on already existing hardware.

5.7.4 *Compiler Assisted Branch Prediction*

Deitrich et al. [1998] concern themselves with static branch prediction from the compiler's perspective. They present heuristics, evaluate them for a selection of benchmarks and provide insights about their efficacy.

Mahlke and Natarajan [1996] present a compiler technique for branch prediction by using profile feedback to insert a prediction function that captures information about the current context that is then used to make predictions. They show performance comparable to that of two-level hardware branch predictors.

5.7.5 *'Weird' Behavior*

In Section 3.2.6 we mentioned the work of Mytkowicz et al. [2009], who describe the effect of linking order as a source of measurement bias. They attribute measurement bias due to link order as an alignment issue, and using the m5 simulator they see instruction cache miss variances when the link order is changed. They suggest branch prediction may also be a cause in general. Our work investigates optimizing code by maximizing information available in branch predictor history. We do not propose that measurement bias may be eradicated, but that bias produced by branch predictor aliasing may be decreased. In a later paper [Knights et al., 2009], the authors propose 'blind optimization'. We believe this will be a useful technique, but there remain traditional hardware-aware optimizations to be discovered and exploited.

Hundt et al. [2011] describe changes in execution time from *nopinizer*, a random NOP insertion pass they implemented using their assembly language optimization framework. They also observed performance variations from a *nop killer* pass. We do not insert NOPs at random, but instead attempt to utilize them

to provide as much information to the predictor as possible. However, for any program, there will always be an unknown optimal address layout that maximizes entropy for a particular history. Indeed, constructive interference [Young et al., 1995] causes branches that would have been mispredicted to be correctly predicted.

5.8 CONCLUSIONS

In this chapter we presented an optimization opportunity where the indirect branch predictor does not perform as well as possible, due to indirect branch target aliasing that can be removed or reduced.

We described our optimization to improve the branch prediction of indirect jumps and calls. We presented two techniques to increase the uniqueness of the bits in target addresses that are used by the predictor, the first based on NOP insertion, and the second based on re-ordering; and we describe them formally. We provided heuristics to approximate these techniques and implemented two as assembly language optimizations. Experimental results, both simulation and hardware, were provided for C programs using SPEC2006 and our own intensive indirect branch programs, showing that providing more information to the predictor by making targets unique can reduce the number of indirect branch mispredictions. We also provided preliminary results for virtual function calls.

In the next chapter we describe an optimization that detects switch-based interpreters (or interpreter-like code) in programs and converts them to token-threaded interpreters.

DETECTING AND AUTOMATICALLY CONVERTING INTERPRETERS

6.1 INTRODUCTION

In Chapter 2 we described virtual machines. As we mentioned, they are commonly used to implement languages such as Java, Python and Lua and typically an interpreter or just-in-time (JIT) compiler is used to execute the program. JIT compilers produce very fast code for the most commonly executed parts of a program, but interpreters are nonetheless common for both software engineering and performance reasons. Interpreters are simpler than JIT compilers, and consume less memory, both for the interpreter itself and the generated code. Interpreters make tools such as debuggers and profilers easy to build. This profiling information can be useful to identify hot spots and common paths. As a result, many modern VMs use a profiling interpreter that executes code that is not profitable to JIT compile. For example, the TraceMonkey [Gal et al., 2009] JavaScript engine uses a profiling interpreter.

In Section 3.3 we described the main steps in interpreting a VM instruction: are to fetch the opcode, fetch the operands, and jump to the real machine code routine that implements the opcode. In the most common form of interpreter, this jump to the routine is implemented using a switch (or case) statement; which is the simplest method to dispatch to the various bytecode implementations, and has the advantage of being portable ANSI C. Typically a compiler will translate these switch statements into a jump table. For instance, the GCC compiler generates a jump-table for switch statements with as low as 10% density when optimizing for speed [Sayle, 2008].

Some implementers are willing to sacrifice the portability and ‘readable’ code of the switch implementation for performance, converting the interpreter to use token-threaded dispatch, using the non-standard *labels-as-values*¹ C extension available in some compilers. (We previously described alternative interpreter dispatch methods, including token-threaded dispatch in Section 3.3.)

In this chapter we propose detecting interpreters (or interpreter-like structures) in code and automatically generating the faster, token-threaded inter-

¹ Also known as computed goto.

preter implementation from that portable, switch implementation. Our optimization works by performing an analysis of the control flow graph (CFG) and then *grafting*² (copying) code.

In this chapter we make the following contributions:

- We describe the design of our code optimization to detect interpreters or interpreter-like structures in code and generate token-threaded interpreters from them (Section 6.3).
- We describe how it can be implemented with a shared grafting location, or with multiple, unique locations for each target.
- We describe our bounds-check removal optimization (Section 6.4).
- We implement the work as an assembly language transformation, making it compiler independent.
- We apply the two transformation variants to the Lua, Python, and Jam (Java) interpreters and provide results, including hardware performance counters (Section 6.5).

6.2 BACKGROUND

As we described earlier in Chapter 3, token-threaded code is similar to code implemented with a switch statement, where a jump-table consisting of program addresses is used to direct control to opcode implementations. Token-threaded code, however, has a separate dispatch point for each opcode, and does not have a bounds check. A downside, however, is that it is not possible to create a token-threaded interpreter in ANSI C. To implement it, the *labels-as-values* extension of the GCC compiler must be used;³ this allows storing the address of a label and branching to it.⁴ As far as we are aware, token-threaded interpreters are the only use of this C extension.

Listing 6.1 shows typical ANSI C source code of an interpreter implemented using a switch statement; Figure 6.1(a) shows the corresponding control flow graph of such code. There is a single dispatch block. In this simple case, the control flow of all targets join at the same block.

² The grafting idea seems to trace back to [Hsu and Davidson, 1986].

³ Full support for this extension was added to LLVM in version 2.7 <http://blog.llvm.org/2010/01/address-of-label-and-indirect-branches.html>. In previous versions, LLVM converted labels-as-values code to a LLVM switch instruction.

⁴ Token-threading can also be implemented with inline assembly using the `asm` environment.

```

while(1) {
    switch(ip->op) {
        case OP_LDI:
            *sp++=ip->arg;
            ++ip;
            break;
            /* ... */
    }
}

```

Listing 6.1: Typical ANSI C source code for a (portable) switch interpreter.

```

static void *optable[]={
    [OP_LDI]=&&LBL_OP_LDI, ...};

goto *optable[ip->op];

LBL_OP_LDI:
    *sp++=ip->arg;
    ++ip;
    goto *optable[ip->op];
    /* ... */

```

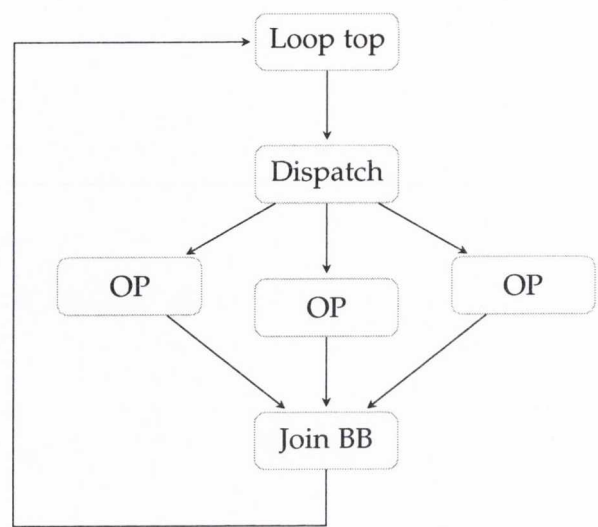
Listing 6.2: C source code for a (non-portable) token-threaded interpreter using the (non-ANSI C) *labels-as-values* extension.

In contrast, Listing 6.2 shows the code of a token-threaded interpreter implemented using the *labels-as-values* extension of the GCC compiler. Figure 6.1(b) shows the control flow of this token-threaded interpreter.⁵ Here we can see that every opcode has its own unique dispatch point. Recall also that the token-threaded version does not include a bounds check, nor a direct jump back to the top of the interpreter loop.

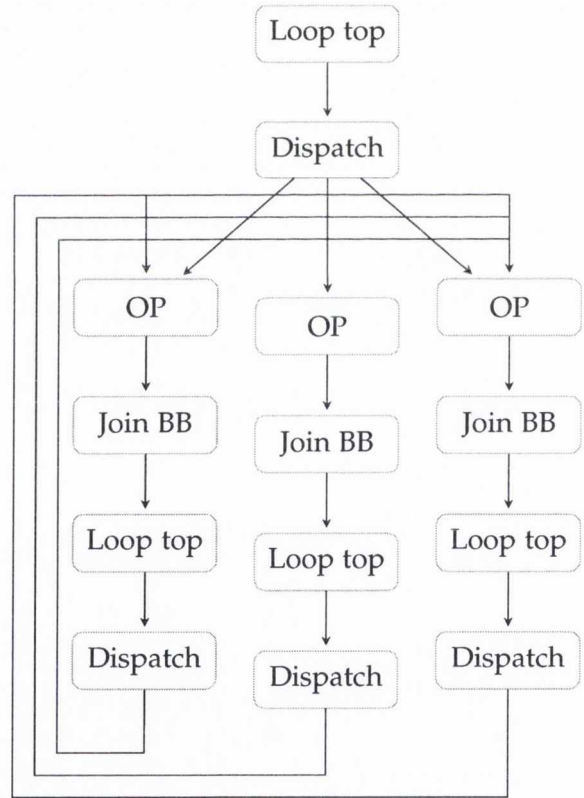
In Section 3.4 we discussed the research that has established branch prediction as a source of slowdown in interpreter implementations. We explained that the benefit of token-threading for branch prediction comes from the dispatch location: with the token-threaded method there is a separate dispatch at the end of each opcode and this improves predictability. Even though modern architectures include an indirect branch target buffer which improves predictability, token-threaded interpreters are still more predictable than switch interpreters, have one less (direct) branch, and do not require a bounds check.

Interpreter-like code structures consist of a jump-table dispatch contained within a loop. We want our optimization to be free from requiring user inter-

⁵ For clarity, not all edges are included. In reality there are edges from all the dispatch blocks to each of the OP targets.



(a) Control flow graph of an interpreter using a switch statement.



(b) Control flow graph of a token-threaded interpreter. The nodes labeled 'Join BB' and 'Loop top' are meant to illustrate they contain instructions that perform the same function as in the nodes with those labels in the switch interpreter.

Figure 6.1: Control flow graphs showing switch and token dispatch mechanisms.

```

while(1) {
    switch(ip->op) {
        case OP_LDI: goto label1;
        /* ... */
    }
    label1: //implementation for OP_LDI
        *sp++=ip->arg; ++ip;
        switch(ip->op) {
            case OP_LDI: goto label1;
            /* ... */
        }
    }
}

```

Listing 6.3: Using multiple switch statements to create unique dispatch locations for each opcode. This will result in jump table duplication, less readable code, and gotos may interact poorly with optimizations.

vention since interpreter-like structures can arise even though programmers may not realize that they have created an interpreter. We also want our transformation to be safe, so that it will preserve the semantics of the original code.

Our idea then is to: 1) automatically detect these interpreter-like code structures and 2) automatically generate the token-threaded interpreter control flow shown in Figure 6.1(b) from the switch statement control flow shown in Figure 6.1(a) that we have detected.

Outlining a possible way to do the transformation from a source code perspective, Listing 6.3 shows a method where multiple switch statements are used to provide multiple dispatch locations.⁶ The problems with this approach compared to token-threading using labels-as-values and our method is that the jump table will be duplicated for every dispatch target, and bounds checking will still be performed on the value of the opcode. This technique also leads to less readable code.

6.3 CONVERTING SWITCH TO TOKEN

Our goal is to detect interpreter-like structures in the code. The first step is to identify interpreter dispatch. The general pattern that we look for is an indirect jump through a jump-table.

Once we have found a candidate interpreter dispatch, we must then check that the dispatch is contained within a loop. A Havlak loop finding algorithm [Havlak, 1997] can be employed to determine the loops in a program. Initially,

⁶ Assuming that compiler optimizations do not detect that one switch statement would suffice and revert it to the code shown in Listing 6.1.

we planned to use the loop structure graph generated by the Havlak algorithm to find the innermost loop that the dispatch site is contained within, and use this as the interpreter loop. However, this does not work in some interpreters, as the dispatch block may be nested in a loop that is below the 'true' interpreter loop in the graph. There is also a problem with irreducible loops, which can arise for various reasons, such as when goto statements are used.

Algorithm 6.1 Computing the *canReachDispatch* set.

```

1: Input: Set of targets of the interpreter dispatch (targets).
2: Output: Set of nodes than can reach dispatch block (canReachSet).
3: canReachSet  $\leftarrow \emptyset$ 
4: dispatchBlock  $\leftarrow \dots$ 
5: for all target in targets do
6:   visited  $\leftarrow \emptyset$ 
7:   UPDATECANREACHSET(target)
8: end for
9: return canReachSet
10: procedure UPDATECANREACHSET(thisBlock)
11:   canReach  $\leftarrow$  false
12:   if not thisBlock  $\in$  visited then
13:     visited  $\leftarrow$  visited  $\cup$  thisBlock
14:     if thisBlock = dispatchBlock then
15:       return true
16:     else
17:       for all succ in successors(thisBlock) do
18:         UPDATECANREACHSET(succ)
19:         if succ  $\in$  canReachSet then
20:           canReachSet  $\leftarrow$  canReachSet  $\cup$  thisBlock
21:         end if
22:       end for
23:     end if
24:   end if
25: end procedure

```

$$\text{CanReachSet} = \bigcup_{t \in \text{targets}} \text{CanReach}(t) \quad (6.1)$$

$$\text{CanReach}(x) = \{x\}, \forall x \in \text{preds}(\text{dispatch}) \quad (6.2)$$

$$\text{CanReach}(n) = \bigcup_{m \in \text{succ}_n} \text{CanReach}(m) \cup \{n\} \text{ if } \exists m \in \text{succ}_n \text{ CanReach}(m) \quad (6.3)$$

To overcome the limitations with the loop structure graph, we use our own alternative method. We first calculate the set of blocks that can reach the dispatch from each of the dispatch targets. This set, which we call *canReachDispatch*, is used by subsequent algorithms that previously would have checked if a block was in the loop. We present the calculation of this set as a backward data-flow problem in Equation 6.1, 6.2 and 6.3. In Algorithm 6.1 we outline an efficient algorithm to compute the set using a depth-first search with every target as the starting location.⁷ Using this set instead of the loop structure graph means more blocks may be included, for instance a block that returned to the loop head of an outer loop, making code growth more of an issue. To deal with the problem of growth, heuristics can be used to only graft code that is below a distance threshold (blocks that are not more than some distance from the loop head).

This approach to detection means we identify all interpreter-like structures in code and avoid any false positives, ensuring our code optimization is a semantics-preserving transformation.

6.3.1 'Join Block' Method

Once an interpreter is detected, the second task is to graft code to duplicate the paths back to the dispatch location. The first approach to grafting we investigate is to emulate the structure of a token-threaded interpreter by searching for a single location where code should be grafted from.

A single block is picked which we call the *join block*. This block is chosen as the location where the largest number of targets stop 'dominating'.⁸

Instead of computing the full dominator tree from the targets, we use the algorithm shown in Algorithm 6.2 to compute a map from blocks to the number of targets that stop dominating at that block. This algorithm relies on the ancestors map, which is precomputed using a depth-first traversal from each of the targets.

A depth-first search from the join block is then used to record the blocks required to return control back to the dispatch block, and they are added to a list. This list of blocks to copy is then scanned for the following branch pattern:

```
jmp L4
```

⁷ Since a depth-first search is used, some blocks inside loops will be missed on the first pass, so we re-run the search to include all blocks that can reach the dispatch.

⁸ This is not dominance in the usual sense, as we described in Section 2.7 [Lowry and Medlock, 1969], since we do not consider paths from the start node to the node that is dominated, and we do not consider any blocks outside of the *canReachDispatch* set.

Algorithm 6.2 Computing the data structure for the number of targets that stop dominating.

```

1: Input: Set of targets of the interpreter dispatch (targets).
2: Output: Map of the number of nodes that stop dominating at each block
   (numStopDominating).
3: for all target in targets do
4:   visited  $\leftarrow \emptyset$ 
5:   UPDATENUMSTOPDOMINATING(target)
6: end for
7: return numStopDominating
8: procedure UPDATENUMSTOPDOMINATING(thisBlock)
9:   visited  $\leftarrow$  visited  $\cup$  thisBlock
10:  if ancestors[thisBlock] > 1 then
11:    numStopDominating[thisBlock] += 1
12:  else if thisBlock  $\neq$  dispatchBlock then
13:    for all succ in successors do
14:      if succ  $\in$  canReachSet and not succ  $\in$  visited then
15:        UPDATENUMSTOPDOMINATING(succ)
16:      end if
17:    end for
18:  end if
19: end procedure

```

L4: ...

These branches are obviously now redundant, and we remove them.

For each edge into the join block, a position to ‘link’ the new code must be found. This is simple for the fall-through blocks, and blocks that end with an unconditional jump. For blocks that end with a conditional jump, we look for the nearest ‘safe’ spot in both directions, i.e., a place where control flow cannot reach. For example, after an unconditional jump.

Next, the blocks to copy are scanned for labels which are remapped to new names. Then the copying begins, iterating through the list of blocks to copy. Any jumps to the old labels are replaced with the new ones.

Finally, the edge may have contained a jump to the join block. This is also remapped to the new label. Alternatively, if it was a direct jump and the first block grafted onto this edge is the join block⁹, then it is simply removed.

The result of this transformation is that each of the edges that enter the join block are now guaranteed to direct control to a separate dispatch location, yet have the exact same semantics as the original code.

⁹ Due to looping, the first block copied may not be the join block.

6.3.2 Frontier Method

Instead of having a shared graft location, an improvement is to graft code onto the point where each target stops dominating. For each dispatch target, we compute the dominance frontier of edges (the last edge(s) that the dispatch target dominates their source).¹⁰ We graft code onto these edges. This ensures that every target has a path to a unique dispatch location.

Again, computing the entire dominance frontier for the control flow graph is unnecessary, since we only need to know about the dominance of the targets, and only then for blocks that are inside the interpreter loop. Additionally, we need to compute the edges, rather than the blocks, and most dominance frontier algorithms only compute blocks. Algorithm 6.3 shows how we compute the dominance frontier of edges for targets in a jump-table.

Algorithm 6.3 Computing the frontier of edges where targets stop dominating.

```

1: Input: Set of targets of the interpreter dispatch (targets).
2: Output: Set of edges where targets stop dominating (frontier).
3: frontier  $\leftarrow \emptyset$ 
4: for all target in targets do
5:   visited  $\leftarrow \emptyset$ 
6:   UPDATEFRONTIER(target)
7: end for
8: return frontier
9: procedure UPDATEFRONTIER(thisBlock)
10:   visited  $\leftarrow$  visited  $\cup$  thisBlock
11:   if thisBlock  $\neq$  dispatchBlock then
12:     for all edge in outEdges do
13:       if edge.dest  $\in$  canReachSet then
14:         if ancestors[edge.dest]  $> 0$  then
15:           frontier  $\leftarrow$  frontier  $\cup$  edge
16:         else if not edge.dest  $\in$  visited then
17:           UPDATEFRONTIER(edge.dest)
18:         end if
19:       end if
20:     end for
21:   end if
22: end procedure

```

Of course there may be multiple locations where a block stops dominating, so typically the number of edges in the frontier is larger than the number of tar-

¹⁰ In the literature, dominance frontier usually refers to the first *blocks* reachable from some block *n*, that *n* does not dominate.

gets, and there is correspondingly more code copied compared to the previous approach.

We provide results for both this and the join block method in Section 6.5.

6.4 BOUNDS CHECK REMOVAL

The other advantage of writing a token-threaded interpreter, as opposed to a switch interpreter, is that the implementer knows that the values of the opcode variable will always be within certain bounds, so including a bounds check before using the jump-table is unnecessary. Contrast this with the switch, where the compiler typically will insert a bounds check when implementing a switch statement through a jump-table.¹¹ However, in a number of common cases it is possible to automatically determine, using a small analysis, that the range check can be safely eliminated.

For example, the Lua interpreter loop includes the following bitwise function to retrieve the opcode from the instruction:

```
typedef enum { .../*38 opcodes*/ } OpCode;
unsigned int i; //instruction is a 32-bit unsigned integer
switch ( (OpCode) ((i) & (~ ( ~(uint32)0) <<6)) ) ) ...
```

Which compiles to the following x86 assembly:¹²

```
movl    %r14d, %edx
...
andl    $63, %edx
...
cmpl    $37, %edx
...
jbe     .L420 //if less than or equal to 37, jumps to block which performs table jump
```

In this case, the bitwise operation is compiled to a single and instruction, masking the register `edx` with the value 63. The bounds check compare is then performed on this register; however, we know the upper value is at most 63, so we could instead just extend the size of the jump table to accommodate the values that do not map to any opcodes. We use this masking observation to perform our bounds check removal pass.

The second observation that enables bounds check removal is that, in interpreters, the opcode is typically a byte (an unsigned char variable used for

¹¹ This seems to always be true at least for the GCC compiler.

¹² In this discussion, we describe our optimization with regard to our implementation at the assembly level (see Section 6.5.1), but it is adaptable to compiler IRs.

the opcode in C), hence the name ‘bytecode’, allowing the jump-table to be extended to 256 entries.

Our bounds check removal pass works as follows:

1. First, we find the appropriate compare instruction by searching backwards in each of the predecessor blocks of the dispatch block for a `cmp` that uses the same register used for dispatch (moves are also taken into account). If none is found, then bounds check does not exist, so we stop.
2. We count the number of entries in the jump-table¹³ and ensure that the immediate operand to the `cmp` corresponds to this, otherwise we stop.
3. We check if the size of the register used in the compare is a byte, if so we extend the jump-table to size 256 and skip to step 7.
4. Otherwise we note the register used in the compare and proceed backwards to the last instruction that modifies it.
5. If the last instruction that modifies the register is an `and`, we note the mask, otherwise we stop.
6. If the value of the mask is below a certain threshold, we add new targets to the jump-table that go to the previous bounds check fail path, otherwise we stop.
7. We can now safely remove the bounds check. We remove the `cmp` and depending on the original bounds check pass path, we either remove the conditional jump entirely, or we graft in the code from the dispatch block.

We provide separate results for the bounds check removal optimization on our three test interpreters. These are provided in Section 6.5 (Figures 6.3, 6.5 and 6.7).

6.5 EXPERIMENTAL RESULTS

6.5.1 *Implementation*

We implement our transformation in the same assembly language optimization framework as our techniques in Chapter 5 [Hundt et al., 2011]. A major advantage of performing the transformation at this level is that it is a

¹³ There may be more entries in the table than unique targets (multiple entries map to the same target code). This is the case for the Jam interpreter.

compiler-independent optimization. Determining interpreter (or interpreter-like) dispatch is performed by looking for indirect jump instructions that index through a label. For example, the following instruction is an interpreter dispatch candidate:

```
jmp    *.L231(,%rdx,8)
```

For the bounds check removal pass, there are a number of finer implementation details which we omit for brevity: accounting for register moves, interoperability of 32-bit and 64-bit register names, negative numbers used as immediate operands in an unsigned comparison, and ignoring redundant moves.

6.5.2 Experimental Setup

6.5.2.1 Hardware Configuration

Similarly to the previous chapter, in this chapter we evaluate our optimization on the following system: Intel Core 2 Q6600 (Core Kentsfield) 2.4 GHz CPU with 6 GB memory, running Ubuntu GNU/Linux 11.04 with kernel version 2.6.38-10. We compile using the GCC compiler version 4.5.2. Performance counters were collected using the *perf* tool, which is an interface to the Linux kernel perf events. For indirect branch misses, we recorded the BR_IND_MISSP counter.

6.5.2.2 Benchmarks

As in the previous chapters, our benchmark sets for Lua and Python are taken from the Computer Language Benchmarks Game [CLBG, 2011]. As we stated, a lack of established benchmark sets for scripting languages makes these suites a common source of benchmarking for scripting language implementations [Pall, 2011]. Similarly to the previous chapter, our benchmark set for Java is the Java Grande Forum benchmarks [Mathew et al., 1999]. We run each benchmark fifteen times and take the median.

6.5.3 Lua

We evaluate our detection and transformation on the Lua interpreter. As we described in Section 2.2.2, Lua is a dynamically typed scripting language [Ierusalimsky et al., 2005]. One of its main features is its portability, and to maximize that it is written in strict ANSI C (therefore a token-threaded implementation is not available). The virtual machine has 38 opcodes. Since the Lua in-

interpreter only ships as a switch-based interpreter, we ‘hand-coded’ our own token-threaded version in order to provide a meaningful comparison with our automatically generated token-threaded versions.

Figure 6.2 shows the results of our transformation to token-threaded (with bounds check removed) for the Lua interpreter. Figure 6.2(a) shows the speedups we achieve. On average, our techniques generate code that is marginally faster than the ‘hand-coded’ token-threaded interpreter, with the ‘join block’ method performing best. Figure 6.2(b) shows the reductions in indirect branches mispredicted. Figure 6.3 shows the results of performing the bounds check removal alone on the Lua interpreter.

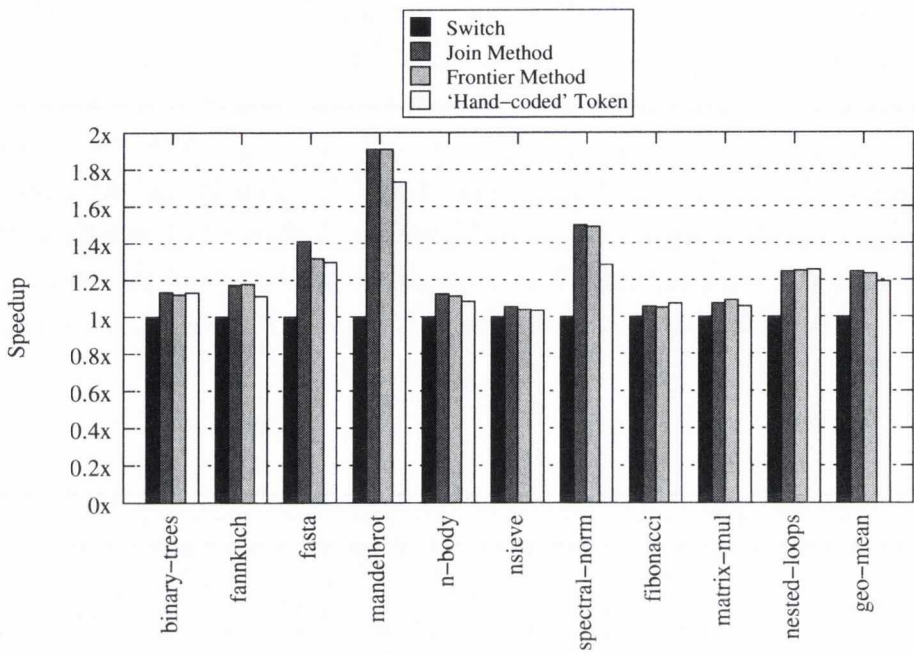
6.5.4 Python

The second interpreter we evaluate is Python 3. In Section 2.2.3 we described Python, which is also a dynamically typed language, but the interpreter is much more complex than Lua. The virtual machine contains 100 bytecodes. To enable our bounds check removal algorithm with Python, one small change is required in the source code; we must change the size of the opcode variable:

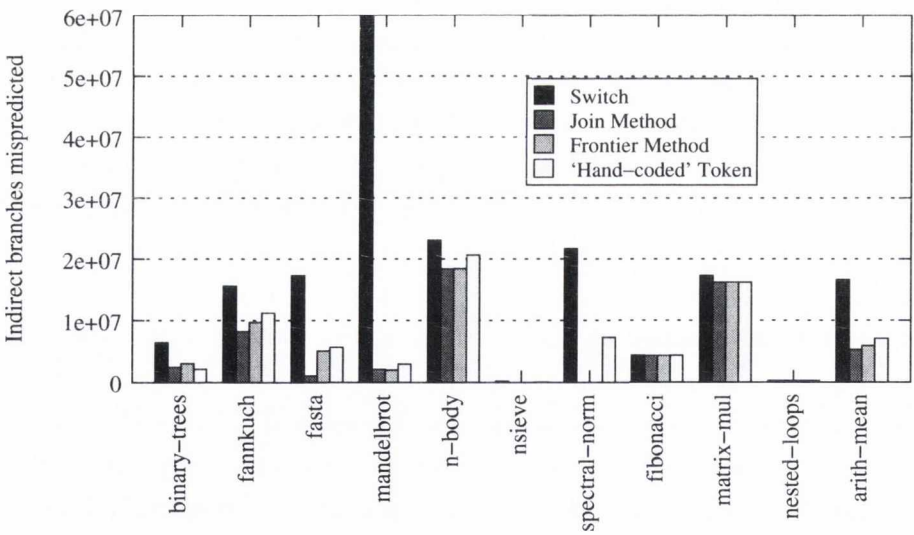
```
unsigned char *next_instr;
int opcode = *next_instr++;
```

The opcode is an integer that is assigned to from the array of bytes. This means we can simply change the size of *opcode* to *unsigned char*.

Figure 6.4 shows the results for the Python interpreter. Figure 6.4(a) shows the speedups. On average, our techniques generate code that is marginally faster than the baseline token-threaded interpreter, and only slightly slower than the ‘hand-coded’ token-threaded interpreter. There are two reasons for the less impressive speedup with Python: 1) the more complex control flow means that for our ‘join block’ technique, the block that gets picked only has nineteen targets that stop dominating at that point and 2) the *call* bytecode calls the interpreter itself, and programs written in Python typically make extensive use of function calls. The join block method performs marginally better than the frontier method, which is surprising given the small number of *inedges* (19) to the join block chosen. Figure 6.4(b) shows that our techniques reduce the number of mispredicted indirect branches in almost all cases. Figure 6.5 shows the results of performing the bounds check removal alone on the Python interpreter.



(a) Execution time speedups.



(b) Indirect branch mispredictions.

Figure 6.2: Lua results comparing the baseline switch interpreter, the automatically generated interpreters using the single join method and the frontier method, and our hand-coded token-threaded interpreter. (Bounds check removal is also performed.)

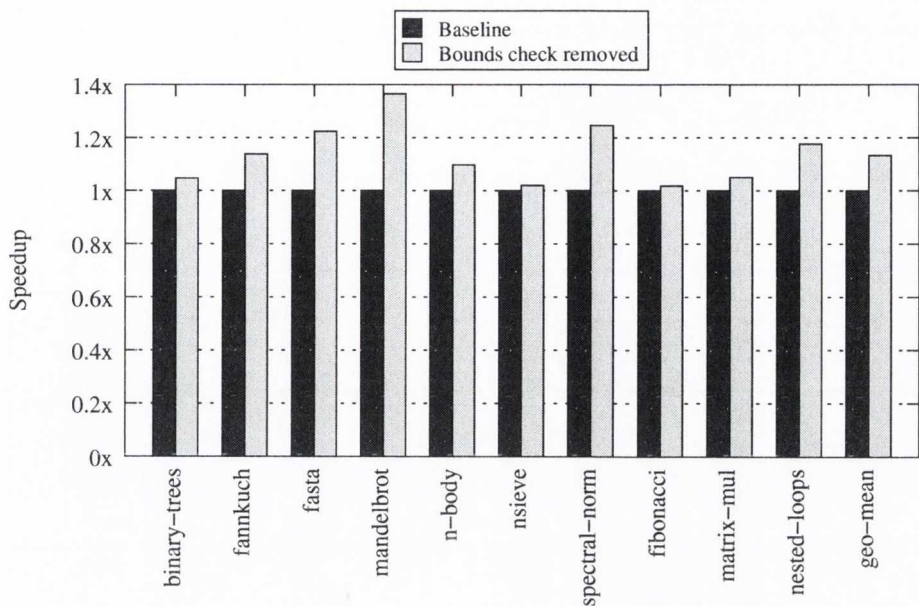


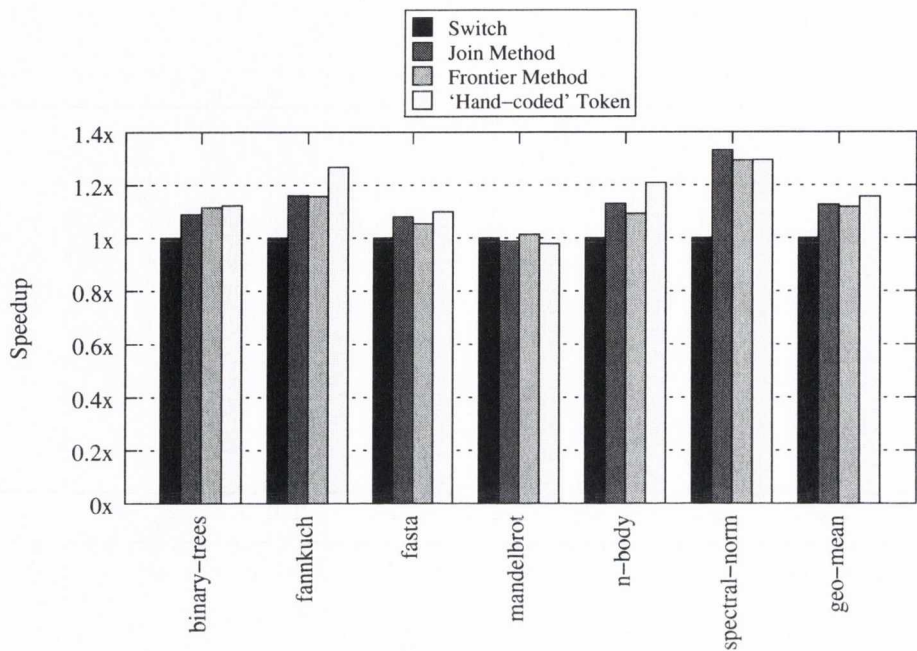
Figure 6.3: Lua bounds check removal results on the baseline (switch) interpreter.

6.5.5 *Java*

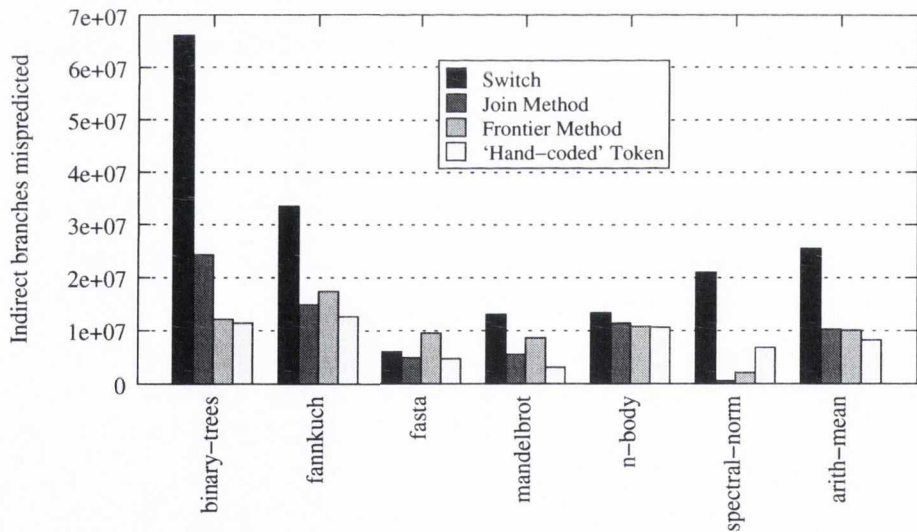
The third interpreter we evaluate is Jam, an interpreter for the Java virtual machine that we described in Section 2.2.4 . The Jam interpreter is configurable, and can be compiled as switch, token-threaded, or direct threaded [Lougher, 2011]. There are 235 opcodes in the Jam virtual machine, significantly more than either Lua or Python.

Figure 6.6 shows the results for the Jam (Java) interpreter. In Figure 6.6(a) we observe the most impressive speedups of all our interpreters. This can be attributed to the fewer number of opcodes in Python and Lua and also the higher relative cost of indirect branches for a statically typed language such as Java: dynamically typed languages such as Lua and Python have conditional branches to check types in most opcodes, and mispredictions of these type checks are a further source of slowdown in those interpreters; additionally, the conditional branches share their history with the indirect branches, further reducing prediction quality.¹⁴ Figure 6.6(b) shows the number of indirect branch mispredictions. We see significant improvements over the baseline for benchmarks such as Eu-

¹⁴ Another aspect to consider here is that each Java opcode performs less work, so the relative cost of dispatch is higher. Therefore gains in prediction rates will have a larger effect on execution time.



(a) Execution time speedups.



(b) Indirect branch mispredictions.

Figure 6.4: Python results comparing the baseline switch interpreter, the automatically generated interpreters using the single join method and the frontier method, and the Python token-threaded interpreter. (Bounds check removal is also performed.)

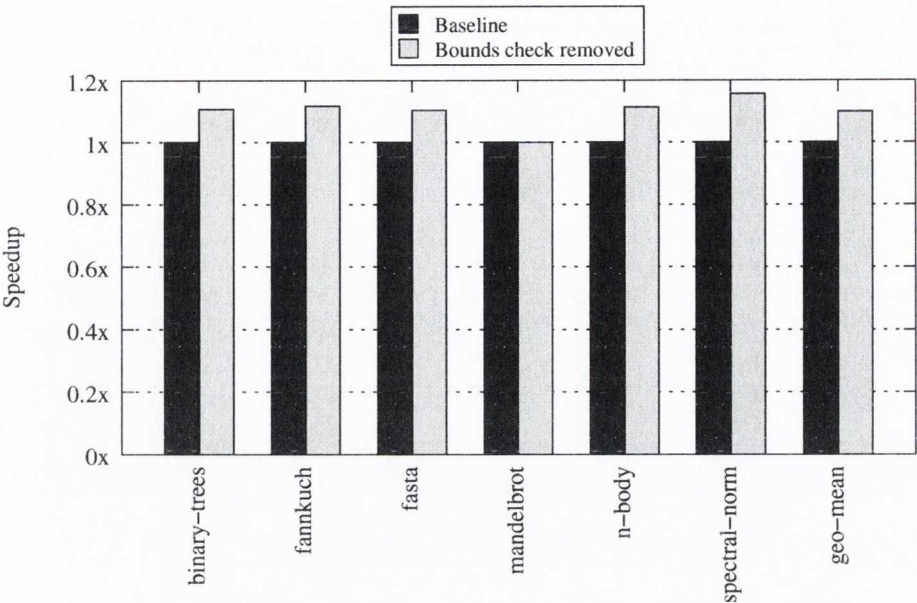


Figure 6.5: Python bounds check removal results on the baseline (switch) interpreter.

ler. On average, our techniques generate code that is marginally faster than the ‘hand-coded’ token-threaded interpreter.

Figure 6.7 shows the results of performing the bounds check removal alone on the Jam interpreter. Noticeably, there are some slowdowns. We believe this may be a case of ‘blind de-optimization’ [Knights et al., 2009] (see also Section 6.6.4).

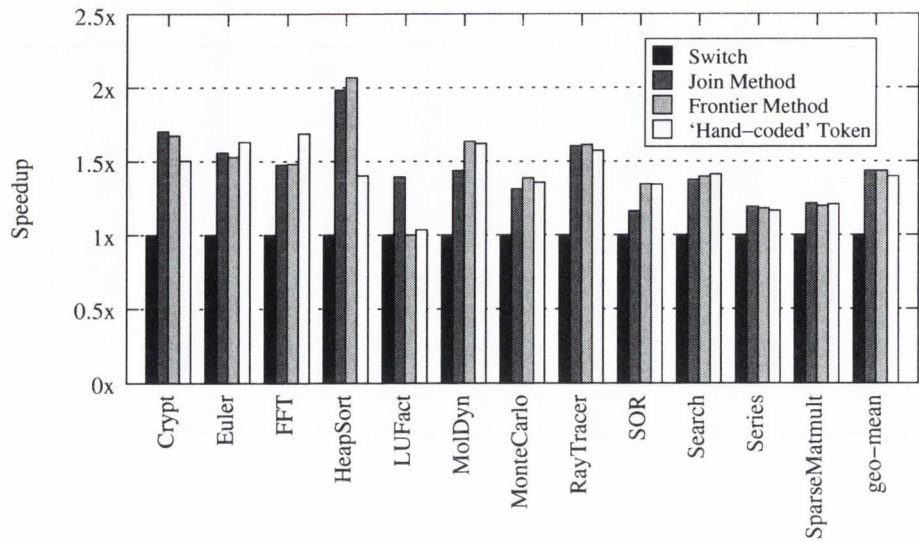
6.5.6 Instruction Cache

We did not observe any increase in instruction cache misses due to code growth on any of the interpreters. This is likely due to the relatively small size of interpreters compared to modern L1 instruction caches.

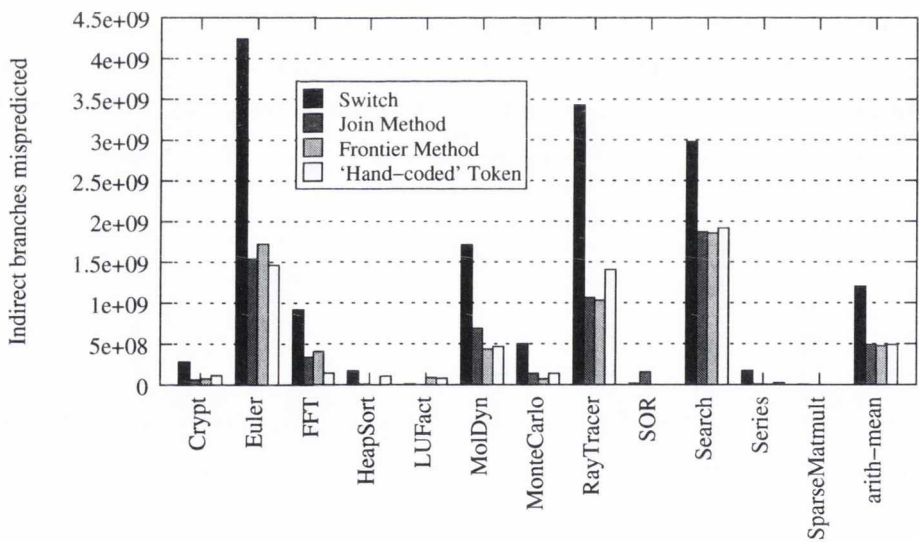
6.6 RELATED WORK

6.6.1 Generating Token-Threaded Interpreters

Hoogerbrugge et. al [Hoogerbrugge et al., 1999, Hoogerbrugge and Augusteijn, 2000] describe a technique to create a token-threaded interpreter by grafting; making *pragmas* available to the programmer to duplicate code from the next



(a) Execution time speedups.



(b) Indirect branch mispredictions.

Figure 6.6: Java results comparing the baseline switch interpreter, the automatically generated interpreters using the single join method and the frontier method, and the Jam token-threaded interpreter. (Bounds check removal is also performed.)

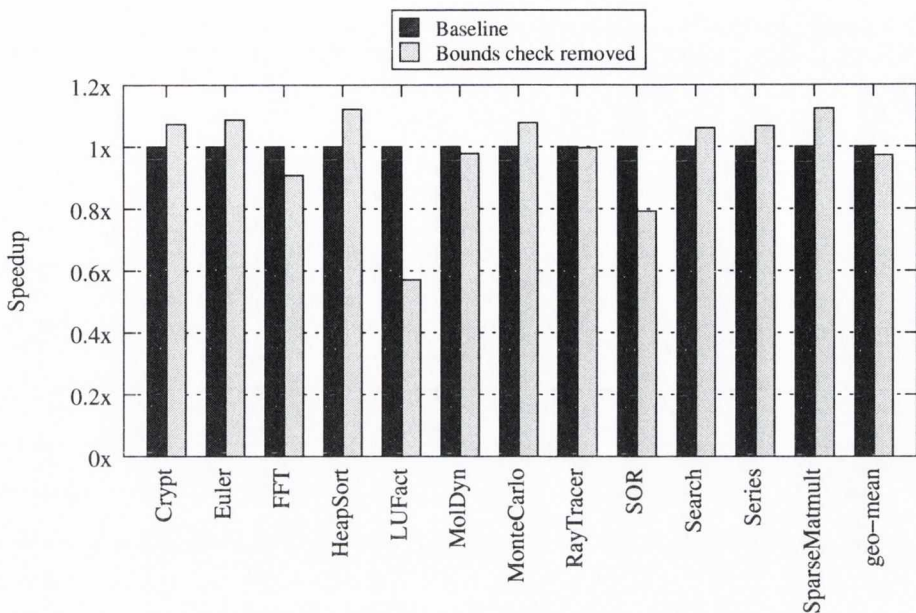


Figure 6.7: Java bounds check removal results on the baseline (switch) interpreter.

basic block into the current one. The technique they describe only grafts code onto the jump corresponding to the back edge of the while loop (thus only duplicating the loop header and dispatch blocks).

Our approach differs significantly as it does not require user intervention through pragmas, and is instead fully automatic. Our approach is also more versatile as it can perform the grafting even if targets merge before the loop back edge (which is the case in all the interpreters we studied). With our approach, each target is guaranteed to have a path to a unique dispatch point. In contrast, their work was carried out before the importance of indirect branch prediction for interpreters was thoroughly studied.

Similarly to us, their compiler also recognizes range checks as unnecessary when the switch value is an unsigned byte (but not when it is a larger type that is masked) and they extend the jump-table accordingly.

6.6.2 Code Grafting

Trace scheduling [Fisher, 1981] schedules regions (traces) of a program using profile information. It is similar to extended basic block scheduling, however the scheduler can move an operation upward across a join point in the CFG; it therefore needs to copy those operations into the non-trace path entering that join point. This copied code is called compensation code.

Superblock scheduling [Hwu et al., 1993] attempts to optimize the most frequent path by creating a trace of the blocks, and removing joins (like trace scheduling, but without side entrances). This allows optimizations to be applied that would not be valid previously. Instruction scheduling may move above control flow joins, requiring code duplication to preserve semantics. This code duplication is called tail duplication [Mahlke, 1997]. Our transformation can be considered to be an application of tail duplication.

6.6.3 *LLVM indirectbr Instruction Lowering*

The LLVM compiler lowers the labels-as-values C extension by generating *indirectbr* IR instructions. To avoid a N^2 explosion in the number of CFG edges, at most one *indirectbr* is used per function, and other indirect gotos are lowered as a branch to a communal *indirectbr*. The code generator then will perform tail duplication to re-introduce all the separate indirect branches [Lattner, 2011].

The *indirectbr* instruction was added to LLVM to support implementing labels-as-values, whose main (if not only) use case is in interpreter implementation. The interpreter writer is still required to write the token-threaded interpreter manually. Our work alleviates this extra effort, and allows the writer to simply write a (portable) switch statement. The tail duplication resemblance between our work and LLVM is superficial as LLVM is simply modifying a direct jump to an indirect jump in its tail duplication, whereas we graft several basic blocks onto different sites.

6.6.4 *Interpreter Layout*

In Chapter 4, we demonstrated that reordering the cases of a switch statement has a significant impact on indirect branch prediction for interpreters. We showed that this is mainly due to aliasing in the indirect branch predictor. Some of our speedups in Section 6.5 may be partly due to these effects, which happen whenever addresses change (not just when opcode implementations are re-ordered).

6.7 CONCLUSIONS

In this chapter we presented a code transformation technique that detects interpreters (or interpreter-like structures) in code and transforms them into token-

threaded interpreters. We described two techniques based on dominance to find positions to perform the code grafting, and gave algorithms to quickly calculate the necessary data sets. We also described a bounds check removal pass.

Our results indicate that our two approaches lead to speedups over the baseline in almost all benchmarks and can perform as well or better than a ‘hand-coded’ token-threaded interpreter. We demonstrated this through extensive experiments on the interpreters of Lua, Python 3, and Java on the Intel Core architecture; comparing our two techniques against the baseline interpreters and the ‘hand-coded’ token-threaded interpreters.

Our transformation works without any user intervention, and will detect all interpreter-like structures whether or not the programmer realizes that is what they have constructed, while avoiding false-positives. We therefore propose that compilers adopt the optimization, as it is a safe transformation. To our knowledge, no compilers perform the code transformation we recommend.

In the next chapter we conclude by relaying our final thoughts on our research and recapitulate the main points presented hitherto in this dissertation.

FINAL THOUGHTS

The purpose of computing is insight, not numbers.

— Hamming [1986]

7.1 INTRODUCTION

In this final chapter we review the research ideas and issues presented earlier in the dissertation, providing some analysis and critique. We identify the practical applications of this work, compare the performance of NOP insertion to the ‘blind’ opcode re-ordering, discuss the efficacy of feedback-directed techniques and reconsider the NP-complete problems. We reflect on measurement, question the necessity of efficient indirect branches, and speculate on hardware support for interpreters. We also comment on the indirect prediction in commodity processors, restate the importance of interpreter optimizations and comment on assembly-level optimizations and architecture-aware compiler optimizations. We conclude this chapter with our ideas for future work and a synopsis of all the material presented in the dissertation.

7.2 PRACTICAL APPLICATIONS

Much of the work we presented in this dissertation has practical applications. One practical implication of our work is that we suggest that in interpreters, opcode implementations should be statically ordered according to the Pareto optimal found on a representative training sample (Section 4.7).

Based on our work in Chapter 5, we suggest that compilers should not align the branch targets of indirect branches, as this will lead to a worst case scenario in the amount of information available that will be used to make branch predictions. Ideally, the compiler would make the lower order bits of the targets unique, as we suggest. Our work also shows that the supposed instruction cache benefits of that alignment is negligible.

Another practical application is that we suggest that compilers adopt the interpreter detection and conversion optimization we described in Chapter 6.

It is a safe transformation (will not affect semantics) and should be relatively straightforward to implement in a compiler.

7.3 NOP INSERTION VS. OPCODE ORDERING

In Chapter 4 we ‘blindly’ re-ordered interpreter opcode implementations at the source-code level, but in Chapter 5 we used architectural knowledge to perform NOP insertion and re-ordering. Since we applied both techniques to the Lua interpreter, we can compare them both. Comparing Figure 4.8 with Figure 5.6, which share some benchmarks (and have been evaluated on the same system), we observe that inserting NOP instructions performs better than re-ordering opcode implementations. For the mandelbrot benchmark the difference is significant: NOP insertion results in an 80% performance improvement whereas Pareto optimal re-ordering from simulated annealing only shows a 10% improvement for that benchmark. This result points strongly at the efficacy of approaches that have knowledge about the hardware parameters.

7.4 EFFICACY OF FEEDBACK-DIRECTED SEARCH TECHNIQUES

In Chapter 4 we implemented and evaluated two feedback-directed techniques to find suitable orderings. It is often assumed that simulated annealing is a better technique than hill-climbing because it is able to avoid getting stuck in local maxima. In our problem though, the cost of solution evaluation was so high (and not incremental) that the suggested number of runs of simulated annealing was not really feasible; leading us to also evaluate hill-climbing and discovering that for us it was a much better method.

Usually the point of using these techniques (in compilers) is to relieve compiler engineers from tedious, time-consuming tasks by automating them, and often yields better results because they do not simply depend on high-level heuristics and intuition. We were faced with a combinatorial search problem, so had no other choice but to resort to using these techniques. We were happy with their outcome and would recommend them to others.

7.5 NP-COMPLETENESS OF NOP INSERTION AND RE-ORDERING

In Chapter 5 we desired proofs that both optimal NOP insertion and re-ordering are NP-complete problems. However, proofs for both still remain elusive.¹

Our most developed insight is that the problem bears similarity to using optimal Golomb rulers for minimizing collisions in closed hashing. Researchers have shown that minimizing collisions in closed hashing is the problem of finding optimal Golomb rulers [Lundberg et al., 2005], and finding optimal Golomb rulers is a long established NP-complete problem [Soliday, 1990]. We plan to continue searching for a reduction of our problem to something else.

7.6 MEASUREMENT PRACTICES

As part of the research during this Ph.D., a significant amount of time was spent investigating correctly measuring execution time using statistically rigorous techniques.

We were concerned about the best practices for measuring execution times and the speedup from particular optimizations: from our reading we knew that much programming language research literature lacks statistically rigorous techniques. However, the epigraph that opened this chapter is relevant here—we should consider the importance of the thing we are measuring and its ability to show a general pattern.

A concrete example of this: While trying to accurately measure performance improvements in the Lua interpreter from optimizations, we observed widely varying execution times (increases and reductions), that could not be explained from the optimization alone. From this, we formed the hypothesis that the execution times were affected by the arbitrary changes in code memory layout. This led to the idea in Chapter 4 of re-ordering interpreter opcodes systematically to improve performance. In this example, increasing the precision of measurement (or the statistical rigor) was superfluous, as the measurement was not providing us with the attribute we wanted to quantify; namely, what effect did the interpreter optimization have on program performance.

Our advice to readers then is simply to remember Hamming's point. It is vitally important to understand the general context of the thing one is measuring; this will lead to knowledge that is perennial.

¹ It is possible of course these problems could belong to the class of problems where proofs can never be found.

7.7 ARE EFFICIENT INDIRECT BRANCHES NECESSARY?

This thesis has focused on improving the predictability of indirect branches. An important question to ask, however, is whether this is even the proper direction. Let us again consider the important classes of programs that utilize indirect branches, and how the use of indirect branches can be mitigated in those programs.

- Interpreters: here the indirect branch problem can be solved by inline-threading or JIT compilation,
- Run-time method binding (in dynamically typed languages): can be solved by polymorphic inline cache [Hölzle et al., 1991], as described in Section 3.5.
- Virtual function dispatch, as for C++: can be solved through type-feedback (based on profiling) to perform if-conversion / devirtualization as described in Section 3.5.
- Function pointers: again, type-feedback could be employed to handle the case of a polymorphic dispatch site.

So the use of a PIC, type-feedback, or JIT compilation, which in this context can be viewed as similar, can replace slow indirect branches. In each case, the software observes its own behavior and adapts its instructions dynamically to optimize for the expected case, inserting appropriate guards to ensure values are as predicted. A problem with each of these techniques though is that they are too difficult for most programmers to be expected to implement.²

It might be worth considering Amdahl's law in this case too: a sufficiently large system may have most indirect branches optimized away by these kinds of software techniques, but then the bottleneck will be any remaining code that still contains them.

Another thing to examine is the cost of translation. This is difficult to evaluate as it depends on how frequently something will be executed and also depends on the cost of dynamic translation on a particular architecture; for a multicore processor it may be almost free by performing the translation in parallel.³

² In each case (except function pointers) we are talking about language implementation, so perhaps language implementers should be expected to have the necessary skills. Regardless, it still places burden upon programmers, which much compiler research is motivated to reduce.

³ To perform the translation in parallel means that a separate thread on a multiprocessor computer compiles code before it is needed by the thread that is executing the program. There has been several publications on achieving this for JIT compilation [Krintz et al., 2001].

The choice of utilizing either conditional branches or indirect branches ultimately depends on hardware cycle counts and misprediction penalties for those instructions. In some cases it may be better to use an indirect branch: this is the motivation for the work by Uh and Whalley [1999] we described in Section 3.2.3.

7.8 HARDWARE SUPPORT FOR INTERPRETERS

Hardware acceleration can drastically speed up certain types of application (and reduce power usage). Recently, CPU manufacturers have been adding more acceleration to processors through the use of *coprocessors*. For instance, applications such as string processing (regular expressions), signal processing and encryption⁴.

In Chapter 3 we mentioned the work of Romer et al. [1996], who suggest that interpreters as a class of program do not require specialized hardware support. Their work dates from 1996, but in 2012, with more interest in mobile systems and reducing power usage, and with transistors being relatively cheap, we are now seeing trends towards more hardware acceleration. So the question of adding hardware support for interpreters is likely to open again.

The indirect branch target buffer we described in Chapter 2 can almost be considered hardware support for interpreters. In our experiments we found that the programs that make the most intensive use of indirect branches are interpreters (and we also saw in the SPEC benchmarks that iBTBs which share history register updates with conditional branches are really only useful for interpreters).

JIT compilers could also benefit from hardware acceleration. As we want to aggressively optimize more code to boost execution speed, the cost of dynamic compilation becomes much higher. To keep power usage low on mobile devices, it may become more feasible to add hardware support for the kinds of compute intensive activities that a typical compiler performs. We speculate that specialized hardware support for dynamic compilation is a research question that will be addressed in the next few years.

⁴ Intel and AMD have extended the x86 instruction set architecture with six Advanced Encryption Standard (AES) instructions.

7.9 INDIRECT BRANCH PREDICTION HARDWARE DEVELOPMENT

In Chapter 5 we presented an optimization opportunity where the branch predictor history was being updated with less information than it could be. It seems unfortunate that the computer architects at Intel chose to use the six lowermost bits of the indirect branch target address as the component to be hashed for indirect branches. It would suggest that the choice was not strongly evidence based, since if they had looked at traces of code that contained branch targets aligned by compilers they would have seen zeroes in the lower bits. Going forward, we expect to either see the choice of bits in the indirect predictor to change (to something like bits 4–10), or for compilers to stop lower bit alignment. In the case of different bit selection, our ideas are still relevant; however, it may involve increasing the number of NOPS to be inserted, making the re-ordering approach more attractive. An issue though with using higher bits is that the targets of a multiway branch are usually laid out nearby in memory, so their higher bit patterns will also be identical.

History based indirect branch predictors have improved the performance of switch based interpreters to be more comparable to token-threaded interpreters, but as our experiments in Chapter 6 show, there is still a significant performance gap between a single dispatch site and multiple sites (some of the difference in performance is due to the direct jump and bounds check, the bounds check being necessary when the compiler cannot determine that the value indexing the jump table is not larger than it). This shows that there is room to improve the hardware, so we expect to see further enhancements to indirect branch prediction in the future.

7.10 INTERPRETER OPTIMIZATION

In this dissertation we described different methods to improve the performance of interpreters. A reasonable question to ask is *are interpreters still relevant?*

We believe strongly that they are still relevant. There are still several use-cases where interpreters are likely to remain. Here are just a few:

- Early implementations of new languages.
- Resource-constrained devices (JIT compilers consume more memory).

- In mixed-mode systems, which consist of a JIT compiler along with a profiling interpreter. These seem to be the most popular kinds of JIT configurations.

As evidence of the continuing relevance of interpreters, we point to the fact that they still influence compiler development. For instance, the LLVM compiler initially supported the ‘labels-as-values’ extension inefficiently, essentially converting it to a switch statement. In LLVM 2.7, however, the IR was altered so that the extension could be implemented efficiently.⁵ This was done to improve interpreter performance.

7.11 OPTIMIZING ASSEMBLY CODE

The work in Chapters 5 and 6 was carried out using an assembly optimization framework developed by engineers at Google [Hundt et al., 2011]. There are several reasons why researchers would want to optimize at the lower level: it ‘frees’ one from many of the complexities of large modern compilers which are composed of many thousands of lines of code, it provides the researcher with access to lower-level details when they are necessary (such as the ability to manipulate NOP instructions in our case), and it enables the idea to be evaluated easily on code that may be generated by different compilers.

The tool we used is not the first tool available for performing such optimizations. There are a number of low-level optimizers. PLTO for instance is a link time optimizer for x86, described by Schwarz [2001]; and an assembly-to-assembly optimizer for the Cray-1 was described by Sites [1978]. However, as the authors of the tool we used state: “MAO is unique in that it is an accessible low-level optimizer designed to help software developers (both compiler experts and non-experts) navigate the complex landscape of performance cliffs.”

In the case of making indirect branch target sites unique, the tool we used already provided code to parse jump tables, and it was then a matter of just inserting NOP instructions. It was also a good solution for interpreter detection and conversion; however, we had to implement more basic tasks (like copying code) as we were operating outside the bounds of what it was designed to accomplish.

We found implementing extensive assembly language transformation to be relatively straightforward and painless, and would recommend assembly transformation to other researchers.

⁵ <http://blog.llvm.org/2010/01/address-of-label-and-indirect-branches.html>

7.12 ARCHITECTURE-AWARE COMPILER OPTIMIZATIONS

In Chapter 5 we presented an optimization that uses architectural knowledge to improve branch prediction rates. This kind of optimization is beneficial, but poses problems with portability. If the processor on which the optimized code was executing did not have the indirect branch predictor structure we expect then there will be no benefit; additionally, cache performance may suffer as a result of the extra (NOP) instructions. One would imagine therefore that some of these optimizations only apply to binaries that are distributed for particular microarchitectures. In a programming language system with dynamic recompilation support this may be easier as we should be able to detect the processor model.⁶ We see an opportunity for more architecture-aware optimizations in the future, concurrently with *blind optimization*.

7.13 FUTURE WORK

During the course of a Ph.D. a number of ideas transpire that cannot be explored properly due to time constraints. We have a number of things we would like to explore in the future.

- We would like to further experiment with indirect branch target addresses and implement a profile-based optimization that will automatically determine good target addresses for a given profile to maximize entropy in the global history. An algorithm would be presented with a trace and using some machine learning techniques, would discover what target addresses would perform the best with the pattern history register update mechanism in hardware. Unlike the work of Jiménez [2005], we would not be using profiles, but actually training our method on the model of the indirect predictor.
- For automatic interpreter conversion, we would like to further explore the trade-off associated with code grafting, and limit the grafting locations to those that are frequently executed (discovered through profiling).

⁶ It should be possible to detect hardware parameters such as the processor model without resorting to code which needs to run with kernel mode privileges.

7.14 SUMMARY & CONCLUSIONS

Accurate indirect branch prediction is increasing in importance through the continued uptake of things like polymorphic dispatch, and the increasing popularity of interpreted languages (despite the existence of JIT compilers, many languages such as Lua and Python remain mainly interpreted). In this dissertation we presented a series of compiler techniques that improve indirect branch prediction performance.

In Chapter 4 we presented a technique to tune interpreters by re-ordering the opcodes, improving the predictability of indirect branches, and affecting performance appreciably. We evaluated this technique on different interpreters and provided results. In Chapter 5 we presented a technique that can improve indirect branch prediction for programs generally. We described two methods to achieve this, and presented formal statements of the problems, which we believe to be NP-complete. As far as we are aware, these problems have not been discovered elsewhere. We provided extensive experimental results for this work. In Chapter 6 we presented a code transformation technique that detects interpreters (or interpreter-like structures) in code and transforms them into token-threaded interpreters. We described two techniques based on dominance to find positions to perform the code grafting, and gave algorithms to quickly calculate the necessary data sets. We also described a bounds check removal pass. We evaluated the techniques on interpreters for Lua, Python and Java; showing speedups of up to 2x, and an average speedup of almost 1.5x for the Java interpreter.

Through this research, we have validated the hypothesis of this dissertation—*Indirect branch prediction for ordinary programs can be improved by adapting the compiler to perform NOP insertion and re-ordering; and for interpreter-like programs by using feedback-directed techniques for ordering and through tail duplication based on block dominance analysis.*

BIBLIOGRAPHY

- Advanced Micro Devices. *Software Optimization Guide for AMD Family 10h Processors*, April 2008.
- Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321486811.
- John Randal Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, POPL '83*, pages 177–189, New York, NY, USA, 1983. ACM. ISBN 0-89791-090-7. URL <http://doi.acm.org/10.1145/567067.567085>.
- Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing Jalapeño in Java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '99*, pages 314–324, New York, NY, USA, 1999. ACM. ISBN 1-58113-238-7. URL <http://doi.acm.org/10.1145/320384.320418>.
- John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35:97–113, June 2003. ISSN 0360-0300. URL <http://doi.acm.org/10.1145/857076.857077>.
- Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming Language Design and Implementation, PLDI '00*, pages 1–12, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. URL <http://doi.acm.org/10.1145/349299.349303>.
- Thomas Ball and James R. Larus. Branch prediction for free. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation, PLDI '93*, pages 300–313, New York, NY, USA, 1993. ACM. ISBN 0-89791-598-4. URL <http://doi.acm.org/10.1145/155090.155119>.
- Jørgen Bang-Jensen, Gregory Gutin, and Anders Yeo. When the greedy algorithm fails. *Discrete Optimization*, 1(2):121–127, 2004. ISSN 1572-5286. URL <http://dx.doi.org/10.1016/j.disopt.2004.03.007>.

- James R. Bell. Threaded Code. *Commun. ACM*, 16(6):370–372, 1973. ISSN 0001-0782. URL <http://doi.acm.org/10.1145/362248.362270>.
- Marc Berndt, Benjamin Vitale, Mathew Zaleski, and Angela Demke Brown. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *Proceedings of the international symposium on Code generation and optimization*, CGO '05, pages 15–26, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2298-X. URL <http://dx.doi.org/10.1109/CGO.2005.14>.
- R. E. Berry. Experience with the pascal p-compiler. *Software: Practice and Experience*, 8(5):617–627, 1978. ISSN 1097-024X. URL <http://dx.doi.org/10.1002/spe.4380080510>.
- Robert Bringhurst. *The Elements of Typographic Style*. Version 2.5. Hartley & Marks, Publishers, Point Roberts, WA, USA, 2002.
- Shirley Browne, Jack Dongarra, N. Garner, Kevin S. London, and Philip Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7803-9802-5. URL <http://portal.acm.org/citation.cfm?id=370049.370424>.
- Stefan Brunthaler. Virtual-machine abstraction and optimization techniques. *Electron. Notes Theor. Comput. Sci.*, 253:3–14, December 2009. ISSN 1571-0661. doi: 10.1016/j.entcs.2009.11.011. URL <http://dl.acm.org/citation.cfm?id=1660183.1660797>.
- Stefan Brunthaler. Interpreter instruction scheduling. In Jens Knoop, editor, *Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*, pages 164–178. Springer Berlin / Heidelberg, 2011. URL http://dx.doi.org/10.1007/978-3-642-19861-8_10.
- Brad Calder and Dirk Grunwald. Reducing branch costs via branch alignment. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VI, pages 242–251, New York, NY, USA, 1994a. ACM. ISBN 0-89791-660-3. URL <http://doi.acm.org/10.1145/195473.195553>.
- Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium*

- on *Principles of programming languages*, POPL '94, pages 397–408, New York, NY, USA, 1994b. ACM. ISBN 0-89791-636-0. URL <http://doi.acm.org/10.1145/174675.177973>.
- Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2: 313–351, 1994.
- Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17:471–523, December 1985. ISSN 0360-0300. URL <http://doi.acm.org/10.1145/6041.6042>.
- Kevin Casey. *Automatic Generation of Optimised Virtual Machine Interpreters*. PhD thesis, Trinity College, Dublin, Ireland, September 2005.
- Kevin Casey, David Gregg, and M. Anton Ertl. Tiger – an interpreter generation tool. In Rastislav Bodik, editor, *Compiler Construction*, volume 3443 of *Lecture Notes in Computer Science*, pages 139–139. Springer Berlin / Heidelberg, 2005. URL http://dx.doi.org/10.1007/978-3-540-31985-6_18.
- Kevin Casey, M. Anton Ertl, and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. *ACM Trans. Program. Lang. Syst.*, 29, October 2007. ISSN 0164-0925. URL <http://doi.acm.org/10.1145/1286821.1286828>.
- C. Chambers and D. Ungar. Customization: optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, PLDI '89, pages 146–160, New York, NY, USA, 1989. ACM. ISBN 0-89791-306-X. URL <http://doi.acm.org/10.1145/73141.74831>.
- 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. ISBN 978-1-60558-375-4. URL <http://doi.acm.org/10.1145/1508293.1508304>.
- Po-Yung Chang, Eric Hao, and Yale N. Patt. Target prediction for indirect jumps. In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, pages 274–283, New York, NY, USA, 1997. ACM. ISBN 0-89791-901-7. URL <http://doi.acm.org/10.1145/264107.264209>.

- Chien-Ming Chen and Chung-Ta King. Walk-time address adjustment for improving the accuracy of dynamic branch prediction. *Computers, IEEE Transactions on*, 48(5):457–469, May 1999. ISSN 0018-9340. URL <http://dx.doi.org/10.1109/12.769430>.
- CLBG. The Computer Language Benchmarks Game. Available at <http://shootout.alioth.debian.org/>, 2011. URL <http://shootout.alioth.debian.org/>.
- Keith D. Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2003.
- Necia Grant Cooper, Roger Eckhardt, and Nancy Shera. *From cardinals to chaos: reflections on the life and legacy of Stanislaw Ulam*. Cambridge University Press, 1989. ISBN 9780521367349.
- Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13:451–490, October 1991. ISSN 0164-0925. URL <http://doi.acm.org/10.1145/115372.115320>.
- Brian Deitrich, Ben-Chung Cheng, and Wen-mei Hwu. Improving static branch prediction in a compiler. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques, PACT '98*, pages 214–221, Washington, DC, USA, October 1998. IEEE Computer Society. ISBN 0-8186-8591-3. URL <http://dx.doi.org/10.1109/PACT.1998.727253>.
- Robert B. K. Dewar. Indirect threaded code. *Commun. ACM*, 18:330–331, June 1975. ISSN 0001-0782. URL <http://doi.acm.org/10.1145/360825.360849>.
- R. Dixon, T. McKee, M. Vaughan, and P. Schweizer. A fast method dispatcher for compiled languages with multiple inheritance. In *Conference proceedings on Object-oriented programming systems, languages and applications, OOPSLA '89*, pages 211–214, New York, NY, USA, 1989. ACM. ISBN 0-89791-333-7. URL <http://doi.acm.org/10.1145/74877.74900>.
- Karel Driesen and Urs Hölzle. The direct cost of virtual function calls in C++. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '96*, pages 306–323, New York, NY, USA, 1996. ACM. ISBN 0-89791-788-X. URL <http://doi.acm.org/10.1145/236337.236369>.

- Karel Driesen and Urs Hölzle. Limits of indirect branch prediction. Technical report, University of California at Santa Barbara, Santa Barbara, CA, USA, 1997.
- Karel Driesen and Urs Hölzle. Accurate indirect branch prediction. In *Proceedings of the 25th annual international symposium on Computer architecture*, ISCA '98, pages 167–178, Washington, DC, USA, 1998a. IEEE Computer Society. ISBN 0-8186-8491-7. URL <http://dx.doi.org/10.1145/279358.279380>.
- Karel Driesen and Urs Hölzle. The cascaded predictor: economical and adaptive branch target prediction. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, MICRO 31, pages 249–258, Los Alamitos, CA, USA, 1998b. IEEE Computer Society Press. ISBN 1-58113-016-3. URL <http://dl.acm.org/citation.cfm?id=290940.290993>.
- Karel Johannes Lodewijk Driesen. *Software and hardware techniques for efficient polymorphic calls*. PhD thesis, University of California, Santa Barbara, 1999. Order number AAI9956150.
- M. Anton Ertl. Speed of various Forth interpreter dispatch techniques V2. Available at <http://www.complang.tuwien.ac.at/forth/threading>, 2011. URL <http://www.complang.tuwien.ac.at/forth/threading>.
- M. Anton Ertl and David Gregg. The behavior of efficient virtual machine interpreters on modern architectures. In *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, Euro-Par '01, pages 403–412, London, UK, 2001. Springer-Verlag. ISBN 3-540-42495-4. URL <http://dl.acm.org/citation.cfm?id=646666.699857>.
- M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 278–288, New York, NY, USA, 2003a. ACM. ISBN 1-58113-662-5. URL <http://doi.acm.org/10.1145/781131.781162>.
- M. Anton Ertl and David Gregg. The Structure and Performance of Efficient Interpreters. *Journal of Instruction-Level Parallelism*, 5, 2003b. URL <http://www.jilp.org/vol5/v5paper12.pdf>.
- 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. ISSN 0038-0644. URL <http://dx.doi.org/10.1002/spe.434>.

- Eitan Federovsky, Meir Feder, and Sholomo Weiss. Branch prediction based on universal data compression algorithms. In *Proceedings of the 25th annual international symposium on Computer architecture, ISCA '98*, pages 62–72, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8491-7. URL <http://dx.doi.org/10.1145/279358.279370>.
- Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.*, 30:478–490, July 1981. ISSN 0018-9340. URL <http://dx.doi.org/10.1109/TC.1981.1675827>.
- Etienne M. Gagnon and Laurie J. Hendren. Sablevm: a research framework for the efficient execution of Java bytecode. In *Proceedings of the 2001 Symposium on Java™ Virtual Machine Research and Technology Symposium - Volume 1, JVM'01*, pages 3–3, Berkeley, CA, USA, 2001. USENIX Association. URL <http://dl.acm.org/citation.cfm?id=1267847.1267850>.
- 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. ISBN 978-1-60558-392-1. URL <http://doi.acm.org/10.1145/1542476.1542528>.
- Simcha Gochman, Ronny Ronen, Ariel Berkovits Ittai Anati, Tsvika Kurts, Alon Naveh, Ali Saeed, Zeev Sperber, and Robert C. Valentine. The Intel Pentium M Processor: Microarchitecture and Performance. *Intel Technology J.*, 7(2), May 2003.
- James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996. ISBN 0201634511.
- Paul Graham. *Hackers and Painters: Big Ideas from the Computer Age*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004. ISBN 0596006624.
- Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1994. ISBN 0201558025.

- David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-guided receiver class prediction. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications, OOPSLA '95*, pages 108–123, New York, NY, USA, 1995. ACM. ISBN 0-89791-703-0. URL <http://doi.acm.org/10.1145/217838.217848>.
- GWCLS. The Great Win32 Computer Language Shootout. *Available at* <http://dada.perl.it/shootout/>, 2008. URL <http://dada.perl.it/shootout/>.
- Richard W. Hamming. *Numerical methods for scientists and engineers*. Dover Publications, Inc., New York, NY, USA, 1986. ISBN 0-486-65241-6.
- Paul Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19:557–567, July 1997. ISSN 0164-0925. URL <http://doi.acm.org/10.1145/262004.262005>.
- John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2003. ISBN 1558607242.
- Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, PLDI '94*, pages 326–336, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. URL <http://doi.acm.org/10.1145/178243.178478>.
- Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming, ECOOP '91*, pages 21–38, London, UK, 1991. Springer-Verlag. ISBN 3-540-54262-0. URL <http://dl.acm.org/citation.cfm?id=646149.679193>.
- Jan Hoogerbrugge and Lex Augusteijn. Pipelined Java virtual machine interpreters. In *Proceedings of the 9th International Conference on Compiler Construction, CC '00*, pages 35–49, London, UK, UK, 2000. Springer-Verlag. ISBN 3-540-67263-X.
- Jan Hoogerbrugge, Lex Augusteijn, Jeroen Trum, and Rik Van De Wiel. A Code Compression System Based on Pipelined Interpreters. *Software - Practice and Experience*, 29:1005–1023, 1999. URL [http://dx.doi.org/10.1002/\(SICI\)1097-024X\(199909\)29:11<1005::AID-SPE270>3.0.CO;2-F](http://dx.doi.org/10.1002/(SICI)1097-024X(199909)29:11<1005::AID-SPE270>3.0.CO;2-F).

- P. Y T Hsu and E. S. Davidson. Highly concurrent scalar processing. In *Proceedings of the 13th annual international symposium on Computer architecture*, ISCA '86, pages 386–395, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press. ISBN 0-8186-0719-X. doi: <http://dx.doi.org/10.1145/17407.17401>. URL <http://dx.doi.org/10.1145/17407.17401>.
- Robert Hundt, Easwaran Raman, Martin Thuresson, and Neil Vachharajani. Mao – an extensible micro-architectural optimizer. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pages 1 –10, April 2011. URL <http://dx.doi.org/10.1109/CGO.2011.5764669>.
- Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: an effective technique for vliw and superscalar compilation. *J. Supercomput.*, 7:229–248, May 1993. ISSN 0920-8542. URL <http://dx.doi.org/10.1007/BF01205185>.
- 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.
- 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. ISBN 978-1-59593-766-X. URL <http://doi.acm.org/10.1145/1238844.1238846>.
- Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a Java just-in-time compiler. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pages 294–310, New York, NY, USA, 2000. ACM. ISBN 1-58113-200-X. URL <http://doi.acm.org/10.1145/353171.353191>.
- Daniel A. Jiménez. Code placement for improving dynamic branch prediction accuracy. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 107–116, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. URL <http://doi.acm.org/10.1145/1065010.1065025>.

- José A. Joao, Onur Mutlu, Hyesoon Kim, Rishi Agarwal, and Yale N. Patt. Improving the performance of object-oriented languages with dynamic predication of indirect jumps. *SIGPLAN Not.*, 43:80–90, March 2008a. ISSN 0362-1340. URL <http://doi.acm.org/10.1145/1353536.1346293>.
- José A. Joao, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Dynamic predication of indirect jumps. *IEEE Comput. Archit. Lett.*, 7:1–4, January 2008b. ISSN 1556-6056. URL <http://dx.doi.org/10.1109/L-CA.2008.2>.
- John Kalamatianos and David Kaeli. Indirect branch prediction using data compression techniques. *Journal of Instruction Level Parallelism*, 1(6), 1999.
- Ming-Yang Kao. *Encyclopedia of algorithms*. Springer Reference. Springer, 2008. ISBN 9780387301624.
- Hyesoon Kim, José A. Joao, Onur Mutlu, Chang Joo Lee, Yale N. Patt, and Robert Cohn. VPC prediction: reducing the cost of indirect branches via hardware-based dynamic devirtualization. In *Proceedings of the 34th annual international symposium on Computer architecture, ISCA '07*, pages 424–435, New York, NY, USA, 2007a. ACM. ISBN 978-1-59593-706-3. URL <http://doi.acm.org/10.1145/1250662.1250715>.
- Hyesoon Kim, José A. Joao, Onur Mutlu, and Yale N. Patt. Diverge-merge processor: Generalized and energy-efficient dynamic predication. *IEEE Micro*, 27:94–104, January 2007b. ISSN 0272-1732. URL <http://dx.doi.org/10.1109/MM.2007.9>.
- Hyesoon Kim, José A. Joao, Onur Mutlu, Chang Joo Lee, Yale N. Patt, and Robert Cohn. Virtual program counter (VPC) prediction: Very low cost indirect branch prediction using conditional branch prediction hardware. *IEEE Trans. Comput.*, 58:1153–1170, September 2009. ISSN 0018-9340. URL <http://portal.acm.org/citation.cfm?id=1591892.1592223>.
- Artur Klauser, Todd Austin, Dirk Grunwald, and Brad Calder. Dynamic hammock predication for non-predicated instruction set architectures. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques, PACT '98*, pages 278–285, Washington, DC, USA, October 1998. IEEE Computer Society. ISBN 0-8186-8591-3. URL <http://dx.doi.org/10.1109/PACT.1998.727261>.
- Dan Knights, Todd Mytkowicz, Peter F. Sweeney, Michael C. Mozer, and Amer Diwan. Blind optimization for exploiting hardware features. In *Proceedings*

- of the 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, CC '09, pages 251–265, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-00721-7. URL http://dx.doi.org/10.1007/978-3-642-00722-4_18.
- Donald E. Knuth. Structured programming with go to statements. *ACM Comput. Surv.*, 6:261–301, December 1974. ISSN 0360-0300. URL <http://doi.acm.org/10.1145/356635.356640>.
- Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998. ISBN 0-201-89685-0.
- Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions*. Addison-Wesley Professional, 2005. ISBN 0201853949.
- Peter M. Kogge. An architectural trail to threaded-code systems. *Computer*, 15(3):22 – 32, March 1982. ISSN 0018-9162. URL <http://dx.doi.org/10.1109/MC.1982.1653970>.
- Andreas Krall. Improving semi-static branch prediction by code replication. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, PLDI '94*, pages 97–106, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. URL <http://doi.acm.org/10.1145/178243.178252>.
- Andreas Krall and Reinhard Grafl. Cacao — a 64-bit Javavm just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, 1997. ISSN 1096-9128. URL [http://dx.doi.org/10.1002/\(SICI\)1096-9128\(199711\)9:11<1017::AID-CPE347>3.0.CO;2-0](http://dx.doi.org/10.1002/(SICI)1096-9128(199711)9:11<1017::AID-CPE347>3.0.CO;2-0).
- Chandra J. Krintz, David Grove, Vivek Sarkar, and Brad Calder. Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31(8):717–738, 2001. ISSN 1097-024X. doi: 10.1002/spe.384. URL <http://dx.doi.org/10.1002/spe.384>.
- Chris Lattner. Address of Label and Indirect Branches in LLVM IR. *Webpage*, 2011. URL <http://blog.llvm.org/2010/01/address-of-label-and-indirect-branches.html>.
- Douglas Lea. Customization in C++. In *Proceedings of the USENIX C++ conference*, 1990.

- Hugh Leather, Michael O'Boyle, and Bruce Worton. Raced profiles: efficient selection of competing compiler optimizations. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '09, pages 50–59, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-356-3. URL <http://doi.acm.org/10.1145/1542452.1542460>.
- Johnny K.F. Lee and Alan Jay Smith. Branch prediction strategies and branch target buffer design. *Computer*, 17(1):6–22, January 1984. ISSN 0018-9162. URL <http://dx.doi.org/10.1109/MC.1984.1658927>.
- Tao Li and L.K. John. Understanding control flow transfer and its predictability in Java processing. In *Performance Analysis of Systems and Software*, 2001. ISPASS. 2001 IEEE International Symposium on, pages 65–76, 2001. URL <http://dx.doi.org/10.1109/ISPASS.2001.990678>.
- Tao Li, Ravi Bhargava, and Lizy Kurian John. Adapting branch-target buffer to improve the target predictability of Java code. *ACM Trans. Archit. Code Optim.*, 2:109–130, June 2005. ISSN 1544-3566. URL <http://doi.acm.org/10.1145/1071604.1071605>.
- Chun-Chieh Lin and Chuen-Liang Chen. Code arrangement of embedded Java virtual machine for nand flash memory. In *Proceedings of the 3rd international conference on High performance embedded architectures and compilers*, HiPEAC'08, pages 369–383, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-77559-5, 978-3-540-77559-1. URL <http://portal.acm.org/citation.cfm?id=1786054.1786088>.
- Robert Lougher. JamVM virtual machine. Available at <http://jamvm.sourceforge.net>, 2011. URL <http://jamvm.sourceforge.net>.
- Edward S. Lowry and C. W. Medlock. Object code optimization. *Commun. ACM*, 12:13–22, January 1969. ISSN 0001-0782. URL <http://doi.acm.org/10.1145/362835.362838>.
- Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. URL <http://doi.acm.org/10.1145/1065010.1065034>.

- Lars Lundberg, Håkan Lennerstad, Kamilla Klonowska, and Göran Gustafsson. Using optimal Golomb rulers for minimizing collisions in closed hashing. In Michael Maher, editor, *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, volume 3321 of *Lecture Notes in Computer Science*, pages 3291–3292. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-24087-7. URL http://dx.doi.org/10.1007/978-3-540-30502-6_11.
- Scott Mahlke and Balas Natarajan. Compiler synthesized dynamic branch prediction. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture, MICRO 29*, pages 153–164, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7641-8. URL <http://dl.acm.org/citation.cfm?id=243846.243881>.
- Scott Alan Mahlke. *Exploiting instruction level parallelism in the presence of conditional branches*. PhD thesis, University of Illinois, Champaign, IL, USA, 1997. UMI Order No. GAX97-17305.
- J. A. Mathew, Paul D. Coddington, and Kenneth A. Hawick. Analysis and development of Java grande benchmarks. In *Proceedings of the ACM 1999 conference on Java Grande, Java '99*, pages 72–80, New York, NY, USA, 1999. ACM. ISBN 1-58113-161-5. URL <http://doi.acm.org/10.1145/304065.304101>.
- Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8:3–30, January 1998. ISSN 1049-3301. URL <http://doi.acm.org/10.1145/272991.272995>.
- Scott McFarling. Program optimization for instruction caches. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems, ASPLOS-III*, pages 183–191, New York, NY, USA, 1989. ACM. ISBN 0-89791-300-0. URL <http://doi.acm.org/10.1145/70082.68200>.
- Scott McFarling. Combining branch predictors. Technical report, Technical Report TN-36, Digital Western Research Laboratory, 1993.
- Scott McFarling and John Hennessey. Reducing the cost of branches. In *Proceedings of the 13th annual international symposium on Computer architecture, ISCA '86*, pages 396–403, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press. ISBN 0-8186-0719-X. URL <http://dx.doi.org/10.1145/17407.17402>.

- Milena Milenkovic, Aleksandar Milenkovic, and Jeffrey Kulick. Demystifying Intel branch predictors. In *In Workshop on Duplicating, Deconstructing and Debunking*, pages 52–61. John Wiley & Sons, 2002.
- Milena Milenkovic, Aleksandar Milenkovic, and Jeffrey Kulick. Microbenchmarks for determining branch predictor organization. *Softw. Pract. Exper.*, 34: 465–487, April 2004. ISSN 0038-0644. URL <http://dx.doi.org/10.1002/spe.v34:5>.
- Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 265–276, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-406-5. URL <http://doi.acm.org/10.1145/1508244.1508275>.
- Melvyn B. Nathanson. *Additive Number Theory: Inverse Problems and the Geometry of Sum-Sets*. Graduate texts in mathematics. Springer, 1996. ISBN 9780387946559.
- Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. URL <http://doi.acm.org/10.1145/1250734.1250746>.
- Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot™ server compiler. In *Proceedings of the 2001 Symposium on Java™ Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, Berkeley, CA, USA, 2001. USENIX Association. URL <http://dl.acm.org/citation.cfm?id=1267847.1267848>.
- Mike Pall. The LuaJIT Project. Available at <http://luajit.org>, 2011. URL <http://luajit.org>.
- Jason R. C. Patterson. Accurate static branch prediction by value range propagation. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 67–78, New York, NY, USA, 1995. ACM. ISBN 0-89791-697-2. URL <http://doi.acm.org/10.1145/207110.207117>.

- Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, pages 16–27, New York, NY, USA, 1990. ACM. ISBN 0-89791-364-7. URL <http://doi.acm.org/10.1145/93542.93550>.
- Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 291–300, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4. URL <http://doi.acm.org/10.1145/277650.277743>.
- Ramesh Radhakrishnan, N. Vijaykrishnan, Lizy Kurian John, and Anand Sivasubramaniam. Architectural issues in Java runtime systems. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, 2000. HPCA-6., pages 387–398, 2000. URL <http://dx.doi.org/10.1109/HPCA.2000.824367>.
- Shah Mohammad Faizur Rahman, Zhe Wang, and Daniel A. Jiménez. Studying microarchitectural structures with object code reordering. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 7–16, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-793-6. URL <http://doi.acm.org/10.1145/1791194.1791196>.
- 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 *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ASPLOS-VII, pages 150–159, New York, NY, USA, 1996. ACM. ISBN 0-89791-767-7. URL <http://doi.acm.org/10.1145/237090.237175>.
- John R. Rose. Fast dispatch mechanisms for stock hardware. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '88, pages 27–35, New York, NY, USA, 1988. ACM. ISBN 0-89791-284-5. URL <http://doi.acm.org/10.1145/62083.62087>.
- Roger Anthony Sayle. A superoptimizer analysis of multiway branch code generation. In *GCC Developers' Summit*, page 103, 2008.
- Benjamin William Schwarz. Post link-time optimization on the Intel IA-32 architecture, 2001. Undergraduate Thesis.

- Stuart Sechrest, Chih-Chieh Lee, and Trevor Mudge. Correlation and aliasing in dynamic branch predictors. In *Proceedings of the 23rd annual international symposium on Computer architecture*, ISCA '96, pages 22–32, New York, NY, USA, 1996. ACM. ISBN 0-89791-786-3. URL <http://doi.acm.org/10.1145/232973.232978>.
- R.L. Sites. Instruction ordering for the Cray-1 computer. Technical Report 78-CS-023, Univ. of California, San Diego, July 1978.
- James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, ISCA '81, pages 135–148, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press. URL <http://dl.acm.org/citation.cfm?id=800052.801871>.
- Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005. ISBN 1558609105.
- Stephen W. Soliday. The search for Golomb rulers. In *Annual zone meeting, Southeastern section of the American Physical Society*, 1990.
- Standard Performance Evaluation Corporation. SPEC CPU2006. Available at <http://www.spec.org/cpu2006/>, 2011. URL <http://www.spec.org/cpu2006/>.
- Sun. The K virtual machine (KVM). Whitepaper, May 2000.
- Ed H. Sussenguth. Instruction control sequence, January 1971. U.S. Patent 3559183.
- Gang-Ryung Uh and David B. Whalley. Effectively exploiting indirect jumps. *Softw. Pract. Exper.*, 29:1061–1101, October 1999. ISSN 0038-0644. URL <http://portal.acm.org/citation.cfm?id=334453.334469>.
- Augustus K. Uht, Vijay Sindagi, and Sajee Somanathan. Branch Effect Reduction Techniques. *Computer*, 30(5):71–81, 1997. ISSN 0018-9162. URL <http://dx.doi.org/10.1109/2.589913>.
- Vladimir Uzelac. Microbenchmarks and Mechanisms for Reverse Engineering of Modern Branch Predictor Units. Master's thesis, The University of Alabama in Huntsville, Huntsville, AL, USA, 2008.

- Vladimir Uzelac and Aleksander Milenkovic. Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2009)*, pages 207–217, April 2009. URL <http://dx.doi.org/10.1109/ISPASS.2009.4919652>.
- Guido van Rossum. Scripting the web with Python. *World Wide Web J.*, 2:97–120, April 1997. ISSN 1085-2301. URL <http://dl.acm.org/citation.cfm?id=275062.275072>.
- Kevin Williams, Jason McCandless, and David Gregg. Portable just-in-time specialization of dynamically typed scripting languages. In Guang Gao, Lori Pollock, John Cavazos, and Xiaoming Li, editors, *Languages and Compilers for Parallel Computing*, volume 5898 of *Lecture Notes in Computer Science*, pages 391–398. Springer Berlin / Heidelberg, 2010. ISBN 978-3-642-13373-2. URL http://dx.doi.org/10.1007/978-3-642-13374-9_27.
- Minghui Yang, Gang-Ryung Uh, and David B. Whalley. Efficient and effective branch reordering using profile data. *ACM Trans. Program. Lang. Syst.*, 24: 667–697, November 2002. ISSN 0164-0925. URL <http://doi.acm.org/10.1145/586088.586091>.
- Tse-Yu Yeh and Yale N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture, MICRO 24*, pages 51–61, New York, NY, USA, 1991. ACM. ISBN 0-89791-460-0. URL <http://doi.acm.org/10.1145/123465.123475>.
- Takashi Yokota, Kanemitsu Ootsu, and Takanobu Baba. Introducing entropies for representing program behavior and branch predictor performance. In *Proceedings of the 2007 workshop on Experimental computer science, ExpCS '07*, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-751-3. URL <http://doi.acm.org/10.1145/1281700.1281717>.
- Cliff Young, Nicolas Gloy, and Michael D. Smith. A comparative analysis of schemes for correlated branch prediction. In *Proceedings of the 22nd annual international symposium on Computer architecture, ISCA '95*, pages 276–286, New York, NY, USA, 1995. ACM. ISBN 0-89791-698-0. URL <http://doi.acm.org/10.1145/223982.224438>.
- Peng Zhao and J.N. Amaral. Feedback-directed switch-case statement optimization. In *Proceedings of the 2005 International Conference on Parallel Processing Workshops, ICPPW '05*, pages 295–302, Washington, DC, USA, 2005. IEEE

Computer Society. ISBN 0-7695-2381-1. URL <http://dx.doi.org/10.1109/ICPPW.2005.32>.

COLOPHON

This thesis was typeset with $\text{\LaTeX} 2_{\epsilon}$ using Hermann Zapf's *Palatino* and *Euler* type faces (Type 1 PostScript fonts *URW Palladio L* and *FPL* were used). The listings are typeset in *Bera Mono*, originally developed by Bitstream, Inc. as "Bitstream Vera". (Type 1 PostScript fonts were made available by Malte Rosenau and Ulrich Dirr.)

The typographic style was inspired by Bringhurst's genius as presented in *The Elements of Typographic Style* [Bringhurst, 2002]. It is available for $\text{\LaTeX}$ via CTAN as "classicthesis".

Final Version as of November 29, 2012 at 18:29.