PEEPHOLE SUPEROPTIMIZATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Sorav Bansal

July 2008

© Copyright by Sorav Bansal 2008
All Rights Reserved

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Alex Aiken) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Kunle Olukotun)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Christos Kozyrakis)

Approved for the University Committee on Graduate Studies.

Abstract

The classical meaning of *superoptimization* [23] is to find the optimal code sequence for a single, loop-free assembly sequence of instructions. Superoptimization has previously been studied for compiling small, performance critical code fragments, such as compute-intensive inner loops. This thesis investigates the use of superoptimization techniques in optimization and code generation for whole programs.

The first part of the thesis describes peephole superoptimizers and their construction. A peephole superoptimizer first generates a peephole optimizer using superoptimization techniques and then applies the generated peephole optimizer to improve executables. Using this approach, we are able to generate many useful peephole optimizations automatically and find improvements over code optimized by mature compilers.

The second part of the thesis applies peephole superoptimizers to perform efficient binary translation between two divergent architectures. We use a superoptimizer to generate equivalence relations between code snippets of two different architectures. The equivalence relation is represented as a peephole transformation. We discuss our PowerPC-x86 binary translator implemented using peephole superoptimization techniques, and compare it with existing binary translation tools.

The third part of the thesis discusses a novel approach to significantly lower the computational complexity of brute-force superoptimization. Our approach, which we call meet-in-the-middle superoptimization, uses reverse execution of instructions to significantly prune the superoptimizer's search space.

Acknowledgements

First and foremost, I wish to thank my adviser Alex Aiken for his guidance and support throughout my Ph.D. career. Alex has dedicated countless hours with me in discussions, writing papers, attending practice talks, and in general, ensuring an excellent research environment. He has taught me by example, how to be a good researcher and presenter. Alex, I feel privileged to be your student.

I wish to thank Mendel Rosenblum for his guidance and support during my first steps into the world of systems research. The OS taint analysis project that I pursued under his guidance has benefitted me greatly. I also thank Mary Baker for supporting me during my tenure as a Masters student.

I thank Dharmendra Modha for his mentorship during my internship at IBM Almaden Research Center. Dharmendra played a crucial role in helping me secure the one year IBM-Stanford studentship.

I thank my office mates Peter and Suhabe and other colleagues at Stanford including Adam, Brian, Isil, Mayur and Tom for their camaraderie. I also thank my friends Abhishek, Amruta, Ankur, Pavan, Rahul, Siddharth, Somik and Tarun for making my time at Stanford very memorable. I especially thank Pramod and Reema whose house had been my home away from home for the last few years.

Finally, I wish to thank my family. Their unwavering love and support is my source of inspiration and courage. I dedicate this thesis to them.

To my family

Contents

Abstract **v**

Acknowledgements **vii**

1 Introduction **1**

1.1 Peephole Superoptimizers 1

1.2 Binary Translation Using Peephole Superoptimizers 5

1.3 Goal-Directed Superoptimization Using Meet-in-the-Middle 9

2 Peephole Superoptimizers **11**

2.1 Design of the Optimizer 11

2.2 Harvesting Target Instruction Sequences 14

2.2.1 Canonicalization 14

2.2.2 Fingerprinting 15

2.3 Enumerator 18

2.3.1 Enumerable Instruction Set 18

2.3.2 Reducing the Search Space 20

2.3.3 Searching the Fingerprint Hashtable 22

2.4 Learning an Optimization 23

2.4.1 Equivalence Test 23

2.4.2 Optimization Database 25

2.5 Experimental Results 26

2.6 Discussion 29

2.7 Related Work 34

2.8 Conclusions and Summary of Contributions 37

3 Binary Translation Using Peephole Superoptimizers 38

3.1 Applications of Binary Translation 38

3.2 Binary Translation Using Peephole Superoptimizers 40

3.2.1 Peephole Superoptimizers 40

3.2.2 Binary Translation 43

3.3 Other Issues in Binary Translation 44

3.3.1 Static vs Dynamic Translation 45

3.3.2 Register Maps 46

3.3.3 Endianness 51

3.3.4 Control Flow Instructions 52

3.3.5 System Calls 52

3.4 Implementation 53

3.4.1 Endianness 53

3.4.2 Stack and Heap 53

3.4.3 Condition Codes 54

3.4.4 Indirect Jumps 55

3.4.5 Function Calls and Returns 57

3.4.6 Register Name Constraints 58

3.4.7 Self-Referential and Self-Modifying Code 58

3.4.8 Untranslated Opcodes 59

3.4.9 Compiler Optimizations 59

3.5 Experimental Results 60

3.5.1 Translation Time 67

3.6 Related Work 69

3.7 Conclusions and Summary of Contributions 72

4 Goal-Directed Superoptimization Using Meet-in-the-Middle 73

4.1 Introduction 73

4.2 Instruction Inverses and Don't-Know Bits 77

4.2.1 Inverse of an Instruction 77

4.2.2 Don't-Know Bits 81
4.2.3 Inverse Execution Constraints 81
4.3 Matching forward-enumerated and backward-enumerated states . . . 82
4.4 Experimental Results 85
4.5 Conclusions and Future Directions 87

5 Conclusions and Future Work **88**

A Runtime Characteristics of SPEC Benchmarks **90**

Bibliography **92**

List of Tables

2.1 Search space of the x86 superoptimizer 22
2.2 Profile of optimizations performed 29
2.3 Description of superoptimized kernels 30
2.4 Results on SPEC CINT2000 benchmarks 31
2.5 Examples of peephole optimizations 32

3.1 Examples of x86 peephole optimization rules 42
3.2 Examples of Register Maps 44
3.3 Examples of PowerPC->x86 translation rules 45
3.4 An example table of peephole translation rules. 48
3.5 Switching Costs 48
3.6 Function call and return instructions in PowerPC and x86 architectures 57
3.7 Examples of x86 instructions that operate only on certain fixed registers. 58
3.8 Manually added peephole translation rules 59
3.9 Results of applying binary translator to compute-intensive microbenchmarks 62
3.10 Results of applying binary translator to SPEC CPU2000 benchmarks 63

4.1 Examples of x86 instruction inverses. 81
4.2 Examples of x86 instruction inverse constraints. 82

A.1 Runtimes of SPEC benchmarks compiled using `gcc` with `-O0` flag . . 90
A.2 Runtimes of SPEC benchmarks compiled using `gcc` with `-O2` flag . . 91

A.3 Percentage of time spent by the SPEC benchmarks in floating point emulation. 91

List of Figures

1.1 Example of Meet-in-the-Middle Pruning 10

2.1 Flowchart of the superoptimizer. 12
2.2 Profile of Harvested Sequences 16
2.3 Sandboxing a Memory Address 17
2.4 Register usage pattern of harvested sequences 20
2.5 Instruction Set Examples 21
2.6 Kernel Speedups 27

3.1 Harvesting instruction sequences from training executables 40
3.2 Brute-force enumeration of all instruction sequences 41
3.3 Application of peephole optimizations 43
3.4 The enumerated register maps for an example function. 50
3.5 PowerPC Condition Codes 54
3.6 X86 Condition Codes 55
3.7 Handling Indirect Jumps in a Static Translator 56
3.8 Experimental Setup 61
3.9 Performance comparison on `-O0` executables by toggling optimization flags in the peephole translator. 65
3.10 Performance comparison on `-O2` executables by toggling optimization flags in the peephole translator. 66
3.11 Sensitivity of translation time on prune size 69
3.12 UQBT Framework 71

4.1 Brute-force exhaustive search approach 74
4.2 Meet-in-the-middle approach 76
4.3 The inverse of an instruction. 78
4.4 Construction of a Trie from Machine States 84
4.5 The distribution of don't-know bits produced by the instructions in our instruction set. 86

Chapter 1

Introduction

Using brute-force to find the optimal code for a given function (also termed *superoptimization*) has previously been studied in the context of computing small mathematical functions or optimizing performance-critical code fragments[15]. Our goal in this thesis is to understand the practicality of using superoptimization as a useful code generation and optimization tool. In this chapter, we first introduce a peephole superoptimizer (Section 1.1), discuss its application as a code generation tool for a binary translator (Section 1.2) and finally present an overview of our meet-in-the-middle superoptimization strategy (Section 1.3).

1.1 Peephole Superoptimizers

Peephole optimizers are pattern matching systems that replace one sequence of instructions by another equivalent, but cheaper, sequence of instructions. The optimizations are usually expressed as parameterized replacement rules, so that, for example,

```
mov r1, r2; mov r2, r1 => mov r1, r2
```

expresses that if the value of register `r1` is copied to register `r2`, then the following instruction `mov r2,r1` is useless and can be deleted. Today, peephole optimization

rules are hand-written, relying on the experience and insight of experts in the machine architecture to recognize and codify the important rules.

In this part of the thesis, we explore a different approach to building peephole optimizers that is both completely automatic and more systematic. The basic idea is to use *superoptimization* techniques (described further below) to automatically discover replacement rules that are optimizations; this computation is done off-line. The optimizations are then organized into a lookup table, mapping original sequences to their optimized counterparts. Optimization of a compiler's generated code can then be done as efficiently as a normal peephole optimizer, simply using the precomputed rules.

This architecture, where optimizations are computed off-line and then presented as an indexed structure for efficient lookup, is much closer to a search engine database than to a traditional optimizer. Unlike standard compilers where every user has a copy of the entire system, search engines have so much data that it is more efficient to keep the data at a central site and provide access to users over a network. We believe it is possible to build a peephole optimizer using our approach with many millions of learned optimizations, and at that scale the most efficient deployment may also be as a network-based search engine. The goal in this thesis is considerably more modest, focusing on showing that an automatically constructed peephole optimizer is possible and, even with limited resources (i.e., a single machine) and learning hundreds to thousands of useful optimizations, such an optimizer can find significant speedups that standard optimizers miss.

The classical meaning of *superoptimization* [23] is to find the optimal code sequence for a single, loop-free assembly sequence of instructions, which we call the *target sequence*. As noted in later work [19], the term superoptimization is an oxymoron: If a program has been optimized—meaning it is optimal—then what can it mean to be superoptimized? The terminology problem lies in the need to distinguish superoptimization from garden variety *optimization* as that term is normally used; compiler optimizations are really just code improvers and it is an accident if a conventional optimizer produces an optimal program. However, for brevity, we will often refer to our own system as an optimizer rather than as a superoptimizer.

There have been two approaches to superoptimization explored in the past. The first, used in Massalin's original paper [23], simply enumerates sequences of instructions of increasing length or cost, testing each for equality with the target sequence; the lowest cost equivalent sequence found is the optimal one. The second approach, pursued in Denali, constrains the search space to a set of equality-preserving transformations expressed by the system designer. For a given target sequence, a structure representing all possible equivalent sequences under the transformation rules is searched for the lowest cost equivalent sequence [19]. A common point of view in both approaches is that superoptimization is something that is expensive, potentially requiring many hours of computation to optimize a single target instruction sequence, and that the main application is as an aid to human performance experts in speeding up the occasional critical inner loop.

Our work differs from this previous work in a number of ways, beginning with the goal. Our main interest is in creating a peephole superoptimizer that is fast enough and systematic enough to be worth using in every compilation. We are also interested in investigating, to what extent the considerable human labor needed to write an optimizer can be automated. To this end, we make the following contributions:

- We superoptimize many target sequences (potentially millions) simultaneously in a first, off-line phase. The target sequences are extracted, or *harvested*, from a *training set* of programs. The idea is that the important sequences to optimize are the ones emitted by compilers; we simply take all instruction sequences up to a given length from a representative collection of existing binaries as our training set.

- Because we aim to be applicable to general binaries, our prototype implementation handles nearly all of the 300+ opcodes of the x86 architecture; previous efforts have focused on a much smaller set of register-to-register operations. In particular, we present the first techniques for correctly inferring superoptimizations involving memory accesses and branches, as well as the first approach that takes the context (e.g., the set of live variables) of an instruction sequence into account.

- A key problem in superoptimization is spending as little time as possible considering instruction sequences that cannot be optimal versions of target sequences. We introduce a new technique, *canonicalization*, based on the observation that having once considered a sequence, we need never consider a sequence that is equal up to consistent renaming of registers and symbolic constants. We show that canonicalization dramatically reduces the search space for our system.

- The output of our system is a set of replacement rules. Each rule gives a source (canonical) instruction sequence and the resulting optimized (canonical) instruction sequence. Thus, these rules can be indexed and used as efficiently as the rules in a standard peephole optimizer. The rules we discover may be less general than rules written by humans—i.e., it may require multiple rules discovered by the superoptimizer to cover the same functionality as a single rule written in a more general form. However, a peephole superoptimizer can compensate for less general rules by automatically discovering many more rules than are written for normal peephole optimizers.

- We report experimental results on a number of kernels where our system achieves speedups of between 1.7 and a factor of 10 over code already optimized by a standard compiler. The improvements show that even mature compilers do not come close to the best possible code in at least some relatively simple situations.

1.2 Binary Translation Using Peephole Superoptimizers

A common worry for machine architects is how to run existing software on new architectures. One way to deal with the problem of software portability is through *binary translation*. Binary translation enables code written for a *source architecture* (or instruction set) to run on another *destination architecture*, without access to the original source code. A good example of the application of binary translation to solve a pressing software portability problem is Apple's Rosetta, which enabled Apple to (almost) transparently move its existing software for the PS/2 to a new generation of Intel x86-based computers [2].

Building a good binary translator is not easy, and few good binary translation tools exist today. There are four main difficulties:

1. Some performance is normally lost in translation. Better translators lose less, but even good translators often lose one-third or more of source architecture performance for compute-intensive applications.

2. Because the instruction sets of modern machines tend to be large and idiosyncratic, just writing the translation rules from one architecture to another is a significant engineering challenge, especially if there are significant differences in the semantics of the two instruction sets. This problem is also exacerbated by the need to perform optimizations wherever possible to minimize problem (1).

3. Because high-performance translations must exploit architecture-specific semantics to maximize performance, it is challenging to design a binary translator that can be quickly retargeted to new architectures. One popular approach is to design a common intermediate language that covers all source and destination architectures of interest, but to support needed performance this common language generally must be large and complex.

4. If the source and destination architectures have different operating systems then source system calls must be emulated on the destination architecture. Operating

systems' wide interfaces combined with subtle and sometimes undocumented semantics and bugs make this a major engineering task in itself.

In the second part of the thesis, we present a new approach to addressing problems (1)-(3) (we do not address problem (4)). The main idea is that much of the complexity of writing an aggressively optimizing translator between two instruction sets can be eliminated altogether by developing a system that automatically and systematically learns translations. In Section 3.5 we present performance results showing that this approach is capable of producing destination machine code that is at least competitive with existing state-of-the-art binary translators, addressing problem (1). While we cannot meaningfully compare the engineering effort needed to develop our research project with what goes into commercial tools, we hope to convince the reader that on its face automatically learning translations must require far less effort than hand coding translations between architectures, addressing problem (2). Similarly, we believe our approach helps resolve the tension between performance and retargetability: adding a new architecture requires only a parser for the binary format and a description of the instruction set semantics (see Section 3.2). This is the minimum that any binary translator would require to incorporate a new architecture; in particular, our approach has no intermediate language that must be expanded or tweaked to accommodate the unique features of an additional architecture.

Our system uses peephole rules to translate code from one architecture to another. Peephole rules have traditionally been used for compiler-optimizations, as we do in Chapter 2. For our binary translator, we use peephole rules that replace a source-architecture instruction sequence by an equivalent destination architecture instruction sequence. For example,

```
ld [r2]; addi 1; st [r2] => inc [er3]  { r2 = er3 }
```

is a peephole translation rule from a certain accumulator-based RISC architecture to another CISC architecture. In this case, the rule expresses that the operation of loading a value from memory location `[r2]`, adding 1 to it and storing it back to `[r2]` on the RISC machine can be achieved by a single in-memory increment instruction

on location `[er3]` on the CISC machine, where RISC register `r2` is emulated by CISC register `er3`.

The number of peephole rules required to correctly translate a complete executable for any source-destination architecture can be huge and manually impossible to write. We automatically learn peephole translation rules using *superoptimization* techniques: essentially, we exhaustively enumerate possible rules and use formal verification techniques to decide whether a candidate rule is a correct translation or not. This process is slow; in our experiments it required about a processor-week to learn enough rules to translate full applications. However, the search for translation rules is only done once, off-line, to construct a binary translator; once discovered, peephole rules are applied to any program using simple pattern matching, as in a standard peephole optimizer. Superoptimization has been previously used in compiler optimization [5, 15], but our work is the first to develop superoptimization techniques for binary translation.

Binary translation preserves execution semantics on two different machines: whatever result is computed on one machine should be computed on the other. More precisely, if the source and destination machines begin in equivalent states and execute the original and translated programs respectively, then they should end in equivalent states. Here, *equivalent states* implies we have a mapping telling us how the states of the two machines are related. In particular, we must decide which registers or memory locations on the destination machine emulate which registers of the source machine. Note that the example peephole translation rule given above is conditioned by the *register map* `r2 = er3`. Only when we have decided on a register map can we compute possible translations. The choice of register map turns out to be a key technical problem: better decisions about the register map (e.g., different choices of destination machine registers to emulate source machine registers) lead to better performing translations. Of course, the choice of instructions to use in the translation also affects the best choice of register map (by, for example, using more or fewer registers), so the two problems are mutually recursive. We present an effective dynamic programming technique that finds the best register map and translation for a given region of code (Section 3.3.2).

We have implemented a prototype binary translator from PowerPC to x86. Our

prototype handles nearly all of the PowerPC and x86 opcodes and using it we have successfully translated large executables and libraries. We report experimental results on a number of small compute-intensive microbenchmarks, where our translator surprisingly often outperforms the native compiler. We also report results on many of the SPEC integer benchmarks, where the translator achieves a median performance of around 66% of natively compiled code and compares favorably with both Qemu [28], an open source binary translator, and Apple's Rosetta [2]. While we believe these results show the usefulness of using superoptimization as a binary translation and optimization tool, there are two caveats to our experiments that we discuss in more detail in Section 3.5. First, we have not implemented translations of all system calls. As discussed above under problem (4) this is a separate and quite significant engineering issue. We do not believe there is any systematic bias in our results as a result of implementing only enough system calls to run many, but not all, of the SPEC integer benchmarks. Second, our system is currently a static binary translator, while the systems we compare to are dynamic binary translators, which may give our system an advantage in our experiments as time spent in translation is not counted as part of the execution time. There is nothing that prevents our techniques from being used in a dynamic translator; a static translator was just easier to develop given the tool base we began with. We give a detailed analysis of translation time for our benchmarks, which allows us to bound the additional cost that would be incurred in a dynamic translator.

In summary, our aim in this thesis is to demonstrate the ability to develop binary translators with competitive performance at much lower cost. Towards this end, we make the following contributions:

- We present a design for automatically learning binary translations using an off-line search of the space of candidate translation rules.

- We identify the problem of selecting a register map and give an algorithm for simultaneously computing the best register map and translation for a region of code.

- We give experimental results for a prototype PowerPC to x86 translator, which

produces consistently high performing translations.

1.3 Goal-Directed Superoptimization Using Meet-in-the-Middle

Superoptimization normally involves a brute-force search over an exponentially large space of instruction sequences. An important goal is to be able to discover techniques to scale a superoptimizer to longer instruction sequence lengths. In this third part of the thesis, we observe that it is possible to significantly prune this search space by using a strategy we call *meet-in-the-middle*. Unlike the naive approach where all instruction sequences are enumerated checking each of them for a match with the goal function, the meet-in-the-middle strategy works backwards from the goal function to enumerate only those instruction sequences that could possibly yield the goal state.

To explain our meet-in-the-middle algorithm, we first define *forward* and *backward* execution of instruction sequences. A forward execution of a sequence is a simple in-order execution of the instructions in the sequence. A backward execution of a sequence is the operation to undo the effects of the instruction sequence on a machine state. To understand this better, consider a machine state Y that is obtained by forward execution of a sequence on a machine state X. A backward execution of the same sequence on Y attempts to recover the state X as much as possible. To reduce the computational time of a brute force superoptimization search, we execute instruction sequences backwards from the goal machine state checking the result for a match with any of the states obtained by forward enumeration of instruction sequences on the initial machine state. An optimization exists only if one of the states obtained by forward-enumerated sequences matches a state obtained from a backward-enumerated sequence (see Figure 1.1). Since we can eliminate any intermediate state that cannot be obtained by backward execution on the goal state, we can prune our search space.

The meet-in-the-middle superoptimization strategy significantly reduces the search space of a brute-force superoptimizer. In this third part of the thesis, we

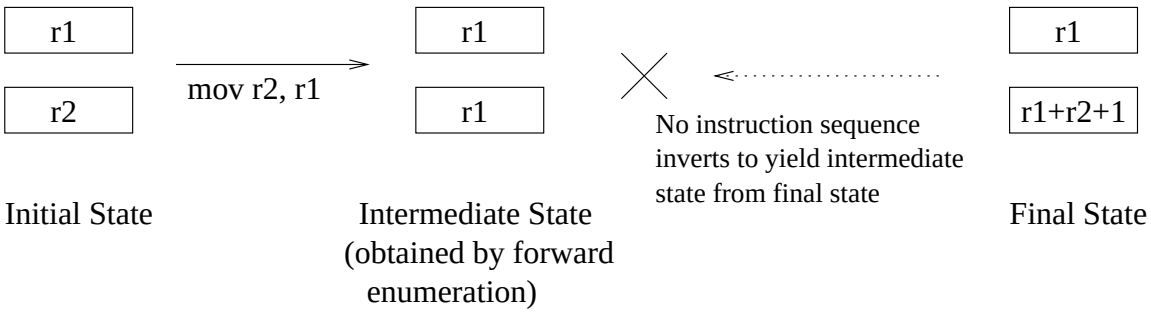


Figure 1.1: An Example of Meet-in-the-Middle Pruning: The intermediate state can be pruned away because there is no instruction sequence beginning in the intermediate state that results in the final state — or, equivalently, there is no sequence of instruction inverses leading from the final state to the intermediate state.

make the following contributions:

- We describe and analyze our meet-in-the-middle strategy to perform efficient goal-directed superoptimization.
- We define the notion of executing an instruction backwards on a machine state through instruction inverses and don't-know bits. These concepts are explained both formally using mathematical constructs and intuitively using examples from the x86 architecture.
- We implement the meet-in-the-middle strategy in both our superoptimizer and the publicly available GNU Superoptimizer[15] and report experimental results.

Chapter 2

Peephole Superoptimizers

In this chapter, we describe the automatic generation of peephole superoptimizers. Section 2.1 discusses the flowchart of a peephole superoptimizer, Sections 2.2-2.4 describe the steps in the flowchart in detail, Sections 2.5-2.6 present experimental results, Section 2.7 discusses related work and finally Section 2.8 concludes. This chapter of the thesis is based on work presented in [5].

2.1 Design of the Optimizer

We begin by defining a few terms that we use throughout the thesis. An *instruction* is an opcode together with some valid operands. For example, on a machine with eight registers `r0` through `r7`, the increment opcode (`inc`) generates eight unique instructions:

```
inc r0 ; inc r1 ; inc r2 ; inc r3 ;
inc r4 ; inc r5 ; inc r6 ; inc r7
```

A potential problem arises with opcodes that take immediate operands, as they generate an unbounded number of instructions. For example, an add-immediate instruction (`addi`) can have 2^{32} different variants (on a 32-bit machine) based on the immediate operand alone.

. . .

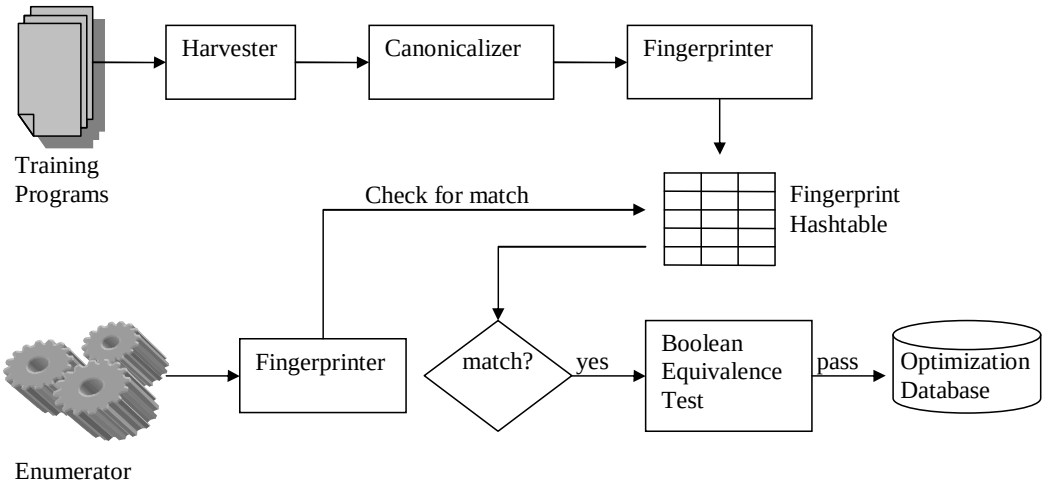


Figure 2.1: Flowchart of the superoptimizer.

```
addi $0x12345678, r0
addi $0x12345679, r0
. . .
```

We restrict immediate operands to a small set of constants and symbolic constants. For example, we simply use a symbolic constant $\$c0$ to capture all the $2^3 2$ possibilities using one instruction

```
addi $c0, r0
```

We enumerate certain constants (e.g. 0 and 1) separately to capture special optimizations. In this way, we ensure opcodes with immediate operands generate only a small number of distinct instructions.

A *cost function* captures the approximate cost of an instruction sequence on a particular processor. We use different cost functions for different purposes; e.g., running time to optimize speed, instruction byte count to optimize the size of a binary. An instruction sequence is *optimal* if no equivalent sequence of lower cost exists. Equivalence of two instruction sequences is defined under a *context*, which is a subset of the machine state that is live beyond the instruction sequences themselves. Since

we ignore I/O instructions, the machine state for our purposes consists of registers, stack and memory. The context of a target instruction sequence can potentially include registers, memory locations and stack locations live at the program point where the instruction sequence ends. However, for implementation simplicity, we currently conservatively assume memory and stack locations are always live. The context of an instruction sequence is thus reduced to the set of registers live on exit from the sequence.

An *equivalence test* $\cong_L$ tests two instruction sequences for equivalence under the context (set of live registers) L. For a target sequence T and a cost function c, we are interested in finding a minimum cost instruction sequence O such that

$$(O \cong_L T)$$

Unlike previous efforts, our superoptimizer computes the optimal instruction sequences for several different target sequences simultaneously. Moreover, once an optimization is found, it is saved in an indexed *optimization database* so that the expensive work done to compute the optimizations need never be repeated again. Thus, the database represents all the optimizations acquired by running the superoptimizer. Once computed, these optimizations can be used to optimize any number of programs.

Our optimizer is structured in three parts:

- The *harvester* extracts target instruction sequences from the training applications. The target instruction sequences are the ones we seek to optimize.
- The *enumerator* exhaustively enumerates all possible candidate instruction sequences up to a certain length, checking if each candidate sequence is an optimal replacement for any of the target instruction sequences.
- The optimizer applies the *optimization database*, an index of all discovered optimizations, to applications.

There are two key challenges for our approach. First, we must reduce the search space of the enumerator as much as possible (Section 2.3.2). Second, we need a very efficient test for equivalence of two instruction sequences (Section 2.4.1).

A flowchart of the superoptimizer is shown in Figure 2.1. We discuss the components shown in the flowchart in the following sections.

2.2 Harvesting Target Instruction Sequences

The first step in creating a superoptimizer using our approach is to obtain target instruction sequences from a representative set of applications. These *harvested* instruction sequences form the corpus used to train the optimizer. Not all instruction sequences are harvestable in our current implementation. A harvestable instruction sequence I must have a single entry point—no instruction in I (except the first instruction) should be a jump target of any instruction outside of I. To enforce this constraint, we identify all jump targets of direct-jump instructions in the binary executable. Also, we identify all instructions starting at addresses pointed to by object symbols since these instructions are possible targets of indirect jump instructions. Any such instructions should not be a part of a harvested instruction sequence I (except possibly being the first instruction in I). Notice that a harvested instruction sequence can have multiple exits since we allow jump instructions in the sequence.

When the harvester extracts instruction sequences from a binary, it also records the set of registers live at the end of the sequence; this context is used in determining equivalence as discussed in Section 2.1.

2.2.1 Canonicalization

All well-formed instruction sequences are valid candidates for optimization, but many sequences are just transformations of each other under renamings of registers and immediate operands. For example, on a machine with eight registers, an instruction `mov r1, r0` has $8*7 = 56$ equivalent versions with different register names. To reduce wasted effort, one would like to eliminate all unnecessary instruction sequences that are mere renamings of others—a process we call *canonicalization*.

An instruction sequence is *canonical* if its registers and constants are named in the order of their appearance in the instruction sequence i.e., the first register used

is always `r0`, the second distinct register used is always `r1`, and so on. Similarly, the first constant used in a canonical instruction sequence is (the symbolic constant) `c0`, the second distinct constant `c1`, and so on.

An instruction sequence is *canonicalized* by renaming registers and constants. An optimization that applies to a sequence is also valid for its canonicalization (with registers and constants suitably renamed). Hence, we store only canonical forms of instruction sequences in our optimization database. Optimizing an instruction sequence I requires first canonicalizing I to $\theta(I)$, where θ is the canonical renaming of registers and symbolic constants of I. We then search the database for a sequence R equivalent to $\theta(I)$, and then "uncanonicalize" R to $\theta^{-1}(R)$ so that the registers and constants have their original names as in I. The sequence $\theta^{-1}(R)$ then replaces I in the application.

Dealing with only canonical instruction sequences dramatically reduces the size of the corpus of target instruction sequences. Figure 2.2 plots the number of unique harvested instruction sequences before and after canonicalization. At short instruction sequence lengths, there are many fewer unique canonical instruction sequences than the number of unique harvested sequences. At longer lengths, the number of harvested instruction sequences decreases because fewer sequences meet the harvester's constraints.

2.2.2 Fingerprinting

The most common operation in our off-line computation of optimizations is determining whether an instruction sequence I is equivalent to any target instruction sequence. We execute I on test machine states and then compute a hash of the result, which we call I's *fingerprint*. The fingerprint is the index into a hashtable; each bucket holds the target instruction sequences, if any, with that fingerprint. The most important properties of the fingerprint are that it is very fast to compute and results in at most a small set of target sequences that might be equivalent to I.

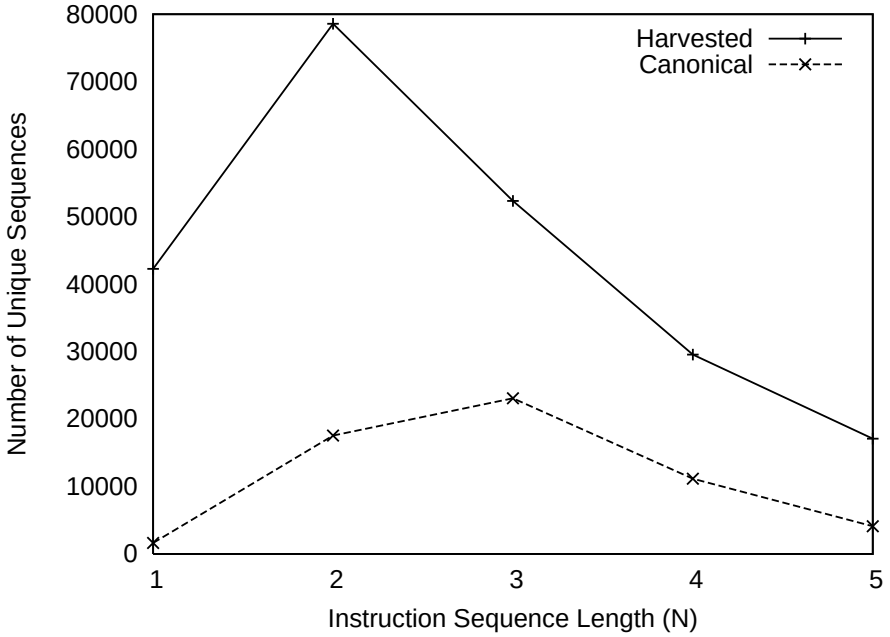


Figure 2.2: The number of unique harvested instruction sequences in SPEC CINT2000 benchmarks, before and after canonicalization.

We have found it sufficient to use two pseudo-random machine states called *testvectors* to compute fingerprints.[1] The instruction sequence is first converted into an executable binary form. The machine is loaded with a testvector and control is transferred to the instruction sequence. The machine state (the contents of registers, status bits, and memory—see below) is recorded after the instruction sequence finishes execution. This process is repeated for both testvectors and a hash is then computed on the machine states that were obtained.

Executing the instruction sequence on the bare machine has three advantages. First, it is extremely fast. Second, it eliminates sources of error due to incorrect simulation of instructions. And third, machine counters can be used to estimate the time spent in executing the instruction sequence on hardware, providing hints for shaping the time-based cost function.

While executing the instruction sequence directly on hardware is good, it presents

[1]Each bit in the two testvectors is set randomly, but the same testvectors are used for fingerprinting every instruction sequence.

Original Instruction Sequence

```
inc r1
xchg (r2), r1
```

With Memory Sandboxing Instructions

```
inc r1
  mov r2, r7
  and $0xff, r7
  add $membase, r7
xchg (r7), r1
```

Figure 2.3: The memory array M starts at address `membase` and is $2^8 = 256$ bytes long. Every memory access is prepended with three instructions ensuring the memory access is contained within M. In this example, a temporary register `r7` was used to perform this function.

its own set of challenges. In particular, we must isolate the state of our system from any side-effects of the instruction sequence. We save all registers before executing the instruction sequence and restore them after the execution is finished. We *sandbox* all memory and stack references by adding extra instructions to the executed code. Both memory and stack accesses are constrained to small regions of memory in the address space of the superoptimizer. The memory is approximated by a small array M of size 2^s starting at a memory address `membase`. Each instruction performing a memory access is then prepended with instructions ensuring that the memory access does not fall outside M. A similar approach is taken for stack references. Figure 2.3 shows the sandboxing instructions used for the x86 instruction set. Note that this strategy for handling memory references preserves the property that if two instructions sequences are equivalent they result in the same machine state on any testvector and therefore have the same fingerprint. We have found that a sandboxed memory of size $M = 256$ bytes is sufficient for minimizing fingerprint collisions between inequivalent instruction sequences.

The function used to hash the machine states obtained after the execution of the instruction sequences on the testvectors must have some special properties to ensure minimal collisions. First, it should be asymmetric with respect to different

memory locations and registers, which is necessary to distinguish between instruction sequences performing identical operations at two different locations. Second, it should not be based on a single operator (like `xor`); otherwise, there are likely to be many collisions on instruction sequences using that particular operator. We employ a combination of `xor` and weighted-add operations to compute the hash of the machine state. To handle context correctly, when fingerprinting a target sequence the hash function includes only the live registers; the values of the dead registers are discarded.

Finally, the full structure of the fingerprint hashtable is more elaborate than we have described so far. For each target instruction sequence I, the hashtable records I and the fingerprint not only for the canonicalization of I, but also for all of I's different register and symbolic constant renamings. This, as we describe in Section 2.3.2, helps us in reducing the search space of the enumerator. Hence, an instruction sequence using r distinct registers and c distinct constants can generate at most $r! * c!$ fingerprints. Typically $r \leq 5$ and $c \leq 2$, so the blow-up is upper-bounded by 240. In practice, we find that the blow-up is around 18. The fingerprint hashtable is indexed by an instruction sequence's fingerprint and set of live registers.

In summary, the fingerprint hashtable maps a fingerprint and set of live registers to a set of instruction sequences with the same fingerprint under that context. This table forms the corpus of instruction sequences that we wish to superoptimize.

2.3 Enumerator

The enumerator simply enumerates all possible, unique instruction sequences. We discuss the enumerable instruction set, techniques to reduce the search space, and the search for useful optimizations in the following subsections.

2.3.1 Enumerable Instruction Set

Instruction sequences are enumerated from a subset of all instructions. At most one branch instruction is allowed in an instruction sequence. For the branch instruction, a canonical target is defined which represents an exit point outside of I. Hence,

an enumerated instruction sequence is allowed to have at most two different exits: the straight-line exit point in the code, and the exit defined by the branch instruction. Notice that while an enumerated instruction sequence can have at most one branch instruction, a target instruction sequence could have more branches; many optimizations eliminate or reduce the number of branches in the target sequence.

To bound the search space, we restrict the maximum number of distinct registers and constants that can appear in an enumerable instruction sequence. For instructions using a restricted subset of registers, only that subset is considered during enumeration. For constants we allow the numbers `0` and `1`, the symbolic constants `c0` and `c1`, and addition or subtraction where the first argument is a symbolic constant and the second argument is a symbolic constant or 1. Allowing addition and subtraction of constants enables discovery of local constant-folding optimizations. Constant-folding optimizations involving more than two constants are captured by repeated application of optimizations to a code sequence.

We also restrict the number of distinct registers used in an enumerated instruction sequence. The number of registers used by instruction sequences varies greatly. We profiled some CPU-intensive applications to gauge this distribution (see Figure 2.4) and observed that more than 50% of harvested instruction sequences of length 8 use fewer than 4 machine registers. Thus, we decided to allow at most 4 distinct registers in an enumerated instruction sequence. Again, notice that a target instruction sequence can use more registers than the corresponding optimal instruction sequence. In fact, many optimizations produced by the superoptimizer eliminate redundant registers.

The number of indirect memory accesses in an instruction sequence is constrained by the number of registers allowed, since an indirect memory access dereferences a register. For direct memory accesses, we allow at most two distinct direct memory addresses (`c0` and `c1`). Because we use the symbolic constants `c0` and `c1` as both values of immediate operands and memory addresses, we capture optimizations involving the transformation of indirect memory accesses to direct memory accesses.

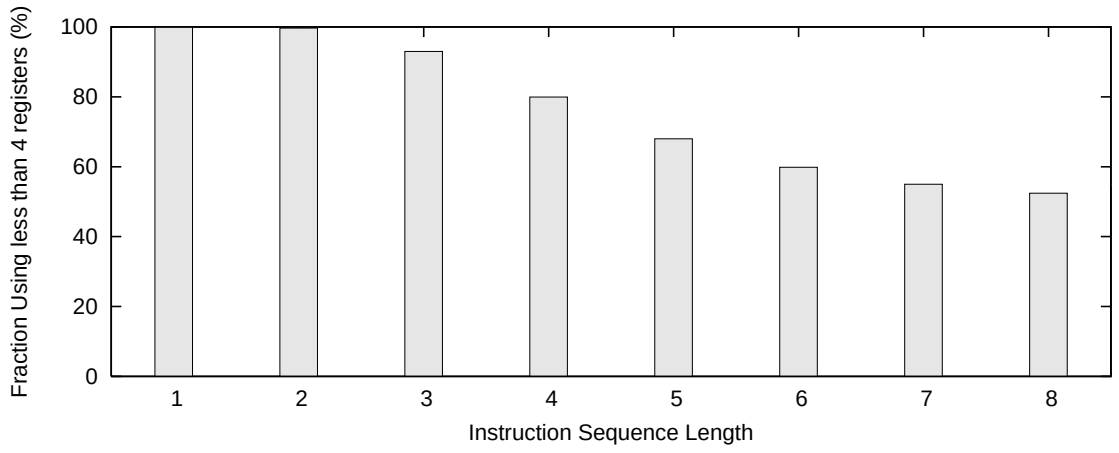


Figure 2.4: Pattern of register usage of harvested instruction sequences in SPEC CINT2000 benchmarks.

Figure 2.5 shows examples of opcodes of different types and the instructions generated by them.

2.3.2 Reducing the Search Space

Once the enumerable instruction set is fixed, the enumerator's search space is exponential in the length of the instruction sequence. We use two techniques to reduce the size of the search space.

- We enumerate only canonical instruction sequences. While this decision reduces the size of the enumerated set of sequences, it does cause a blow-up in the size of the fingerprint hashtable (recall Section 2.2.2).

- We prune the search space by identifying and eliminating instructions that are functionally equivalent to other cheaper instructions.

  For simple cost functions, it is possible to further prune the search space by observing that all subsequences of a length n instruction sequence must be optimal—if any subsequence is not optimal, then it can be replaced by a cheaper sequence and hence the sequence is not optimal. This is always true when we are optimizing for codesize, since the cost function is simply the sum of individual instruction lengths. For runtime optimizations, this is not true in general

```
not <register>
        not r0
        not r1
        not r2
        not r3
dec <memory location>
        dec (r0)
        dec (r1)
        dec (r2)
        dec (r3)
        dec (c0)
        dec (c1)
add <mem-indirect>, <immediate>
        add (r0), 0
        add (r0), 1
        add (r0), c0
        add (r0), c1
        add (r0), c0+1
        add (r0), c0-1
        add (r0), c0+c1
        add (r0), c0-c1
        and repetition of the above for r1, r2, ...
```

Figure 2.5: Examples of instructions generated by opcodes taking different operand-types in the x86 instruction set.

> because running time is also dependent on the combination and order of instructions in the sequence. In our experiments, we employed this aggressive pruning strategy only when optimizing for codesize. Pruning the search space at smaller instruction sequence lengths provides a significant benefit for longer instruction sequences. This idea was first proposed by Massalin [23]. We currently check that all subsequences of length 2 are optimal using a table that lists all length 2 optimal sequences, when optimizing for codesize.

Table 2.1 lists the size of the set of enumerated instruction sequences with and without canonicalization and pruning. While canonicalization provides the biggest reduction, the effect is cumulative and using both techniques we achieve over 50x improvement

in the size of the search space for instruction sequences of length 3 on the x86 architecture. In Table 2.1, the reduction due to pruning is only due to elimination of single instructions that are equivalent to other single instructions. For the codesize cost function, where we can employ the more aggressive pruning strategy, we get a total improvement of 60x (20% more) in the size of the search space at length 3.

Length	Original Search Space	After Canonicalization	After Pruning	Reduction Factor
1	5,453	997	644	8.5
2	29 m	2.49 m	1.2 m	24.7
3	162.1 b	8.6 b	3.11 b	52.1

Table 2.1: The size of the search space for x86 instruction sequences of length 1 to 3. The last column shows the reduction in search space achieved through pruning and canonicalization.

Many of the enumerated sequences are redundant and it is tempting to avoid enumerating them by placing checks in the enumerator. For example, it is possible to check for instruction sequences of the form {`mov r0, r1; mov r0, r1`} and avoid fingerprinting them. However, such checks in the inner loop of the enumerator result in an overall slowdown. In the interest of speed, we let the system weed out such special cases automatically through fingerprinting and equivalence checks.

The enumerator stores enumerable instructions in a table with information about the registers and constants used to help the enumerator generate only canonical instruction sequences. The table is sorted in an order to make enumeration fast. Using the fast fingerprint technique, about $500,000$ instruction sequences per second can be enumerated and fingerprinted on a single processor.

2.3.3 Searching the Fingerprint Hashtable

Each enumerated instruction sequence is fingerprinted as described in Section 2.2.2. The fingerprint is computed for all possible sets of live registers. The fingerprint value and the corresponding set of live registers is then used to look up any matching

target instruction sequence in the fingerprint hashtable. If there is a match, we have found a candidate optimization and proceed with the equivalence test described in Section 2.4.1. If there is no match in the fingerprint hashtable, the enumerated instruction sequence is simply discarded.

Recall that while we enumerate only canonical instruction sequences, the fingerprint hashtable contains instruction sequences in both canonical and non-canonical forms. This is important, because it is possible to optimize a canonical instruction sequence with a non-canonical instruction sequence and vice-versa. For example, a canonical length 2 instruction sequence T {`mov r0, r1; mov r1, r2`} can be optimized using a non-canonical length 1 instruction sequence O {`mov r0, r2`} (assuming `r1` is not live). To catch this optimization, we keep all renamings of T in the fingerprint hashtable and enumerate only the canonical version of O. In this example, the non-canonical renaming of T {`mov r0, r2; mov r2, r1`} in the fingerprint hashtable is optimized by the canonical enumerated sequence {`mov r0, r1`}.

2.4 Learning an Optimization

Once a match is found in the fingerprint hashtable for an enumerated instruction sequence, an equivalence test is performed. If the target instruction sequence and the candidate instruction sequence are found to be equivalent, and the cost of the candidate instruction sequence is lower than the target (or a previously discovered optimization for that target), the optimization is stored in the optimization database. Each of these steps is described in the following subsections.

2.4.1 Equivalence Test

The equivalence test proceeds in two steps—a fast but incomplete execution test and a slower but exact boolean test.

Execution Test

Our fast execution test is similar to fingerprinting. We run the two sequences over a set of testvectors and observe if they yield the same output on each test. In our experiments, we use a total of 18 testvectors: one is all zeros, one is all ones and in the remaining 16, each bit is set randomly.

Contrary to Massalin's experience [23], we found a number of pairs of instruction sequences that passed the execution test and failed the boolean test.[2] This situation arises due to a variety of reasons, almost all involving loss of bits during the computation. For example, an equality comparison of two computed registers on the testvectors is likely to always return false. Similarly, memory addresses are almost never aliased by execution tests, while a boolean deterministic test catches all inconsistencies due to the possibility of memory aliasing.

Boolean Test

The boolean verification test represents an instruction sequence by a boolean formula and expresses the equivalence relation as a satisfiability constraint. The satisfiability constraint is tested using a SAT solver.

A machine state is represented by a finite set of registers and a model of the full memory and stack. Registers are represented as bitvectors. Memory is modeled by a map from address expressions to data bits. The first use of a memory location is encoded by fresh boolean variables representing the data bits at that address. Boolean clauses are used to encode the relationship between the data bits and address bits. e.g., for a sequence performing two memory reads at addresses $addr_1$ and $addr_2$, and returning data bytes $data_1$ and $data_2$ respectively, the following clause captures their aliasing relationship:

$$(addr_1 = addr_2) \Rightarrow (data_1 = data_2)$$

All memory writes are stored in a table in order of their occurrence. For a memory

[2] Massalin did not implement a complete test, relying on humans to confirm that candidate optimizations that passed an execution test were correct in all circumstances.

read occurring after memory writes, the read-address needs to be compared with the address expressions of the writes. Each read-access R is checked for address-equivalence with each of the preceding write accesses W_i in decreasing order of i, where W_i is the i'th write access by the instruction sequence. The following clause encodes this relationship between the data of the read access $data_R$ and the data of one of the preceding write accesses $data_{W_i}$.

$$\bigvee_{j \geq i} (addr_R \neq addr_{W_j}) \wedge addr_R = addr_{W_i} \Rightarrow data_R = data_{W_i}$$

For each pair of memory accesses, a boolean clause is generated to capture the possibility of their address expressions aliasing with each other. Where information is not available, we conservatively assume that two memory addresses may alias. The equivalence of two memory states is checked by reading the bits at each address location for both states and checking them for boolean equivalence. The model of the stack is identical to that of memory, with additional bits representing the stack and frame pointers.

Instructions are encoded as boolean circuits transforming an input machine state to an output machine state. Branch instructions are handled by predicating the execution of instructions on the true and false paths with the branch condition or its negation. The program counter is modeled to indicate if a branch to a target outside the instruction sequence was taken. The input state is shared between the two instruction sequences being checked for equivalence. Two instruction sequences are equivalent iff the registers, memory and stack expressions obtained in the final state are equivalent. The equivalence relation of the output machine states is expressed as a satisfiability constraint before giving it to the SAT solver.

2.4.2 Optimization Database

The optimization database records all optimizations discovered by the superoptimizer. The database is indexed by the original instruction sequence (in its canonical form) and the set of live registers, and returns the corresponding optimal sequence if one

exists. Because instruction sequences stored in the fingerprint hashtable need not be canonical, they must be canonicalized (and their optimal versions renamed) before storing them in the optimization database.

The operation of optimizing a binary executable is fast: it involves only harvesting a target sequence, canonicalizing it, and searching the indexed optimization database. Multiple optimization passes are performed on the executable until no further optimizations are found.

2.5 Experimental Results

Our implementation of the optimizer is written in C++ and O'Caml [22]. We use the Diablo link-time rewriting framework [1, 27] to compute liveness information for an x86 executable binary. We use zChaff [25, 38] as our backend SAT solver because of its performance and incremental SAT solving capabilities. It took around two weeks to write formulas modeling the opcodes of the Intel Pentium instruction set for the boolean test. We compared our optimizer on executables compiled using `gcc` version 3.2.3. The default optimization level used was `-O2`.

Our experiments were done using a Linux machine with a single Intel Pentium 3.0GHz processor and 100 Gigabytes local storage. We limited the peephole size to instruction sequences of length 3, which were not too time consuming to enumerate. Given more resources, we can easily scale the system to length 4 instruction sequences, which we believe, would produce even better results. Going beyond length 4 instruction sequences requires additional techniques to further reduce the search space of the enumerator. Although, we enumerate only up to length 3 instruction sequences, we optimized windows of up to length 6 instruction sequences in our experiments.

We use two different cost functions, one capturing runtime and the other codesize. The codesize cost function simply considers the size of the executable binary code of a sequence as its cost. The runtime cost function is more involved. It first takes into account the number of memory accesses and branch instructions in the sequence. Then, the approximate cycle costs of the instruction are considered, as obtained from the technical manuals on Intel architectures. In case of a tie, the number of registers

used and the code length are used as tie-breakers. [3]

In our first set of experiments, we took some kernels operating on arrays of integer elements. All the kernels were written in C. A description of each of the kernels is given in Table 2.3. These kernels were compiled using architecture specific (`-march=pentium4`, `-mmmx` and `-msse`) optimization options in `gcc`, with the loops unrolled 8 times.

Figure 2.6 plots the runtime improvements our superoptimizer obtained in the different kernels over `gcc`. We achieved improvements of between 1.7 and 10 times over already-optimized code. Some (but not all) of the large improvements in running time are because the superoptimizer finds clever ways to use the SIMD (single instruction multiple data) instructions available in the Intel architecture. The problem of emitting efficient SIMD code has confounded compiler-authors for many years; `gcc` at least does not appear to attempt to use SIMD instructions. Most code involving the use of complex instructions is currently hand-coded by expert assembly programmers. Our results show that an automatically generated optimizer is at least a partial solution to this problem.

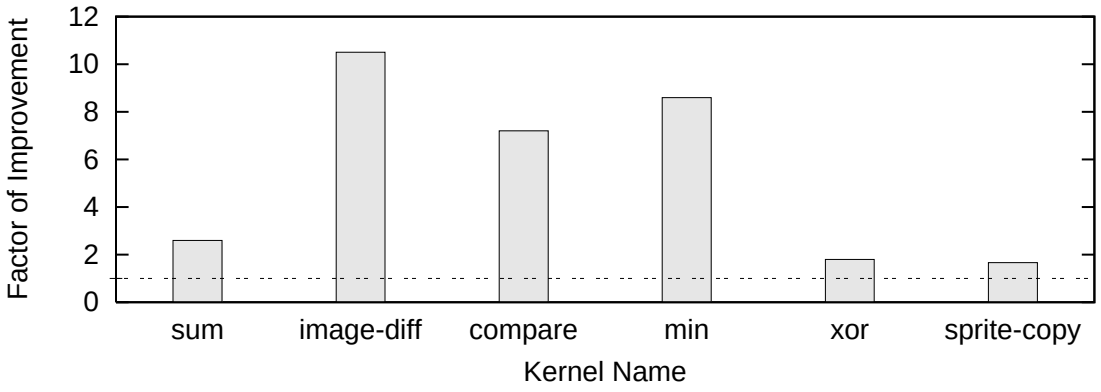


Figure 2.6: Speedups for the kernels in Table 2.3.

Next, we applied the superoptimizer to applications from the SPEC CINT2000

[3] We tried using Pentium performance counters to estimate the runtime of an instruction sequence. In our experience, that was not useful for short sequences due to the large variance in the numbers obtained across different runs.

benchmarks [17]. The number of optimizations performed and the corresponding improvements over `gcc` are shown in Table 2.4. As one would expect, the improvements are much less dramatic for full applications than for compute-intensive kernels. We found speedups of 0-5% with these improvements, though we found that speedup varied across different runs and machine configurations. We saw improvements in code size of 1-6% over executables already optimized for size using `-Os`.

We also ran our optimizer on SPEC executables compiled using the architecture specific Intel C++ compiler `icc` [18]. For the SPEC benchmarks, the speedups obtained on `icc` optimized executables were less than 1%, but we found that the codesize of these executables reduced by 2.5-4% with no performance penalty. On the kernels, our optimizer achieved speedups over `icc` comparable to the results with `gcc`.

A sample of some interesting optimizations performed on binaries that had been already optimized using `gcc` are given in Table 2.5. The system found a range of optimizations, from ones that are well-known (constant folding, redundant load elimination, strength-reduction) to very architecture specific optimizations (the use of the `xchg` instruction to swap registers, and various uses of the SIMD instructions). We discuss two discovered optimizations in detail. In Example 1 of Table 2.5, the superoptimizer finds a three-instruction sequence to compute the sum of eight unsigned byte integers using the 64 bit registers available on the x86 platform. It first zeros out one of the 64 bit registers (`mm0`) by subtracting it from itself. It then uses the `psadbw` instruction, which computes the sum of absolute differences of two 64-bit values. Since one of the registers in this sequence is zero, this amounts to the computation of the sum of the eight bytes in the other operand. The third instruction then stores the computed sum to the memory location `sum`. In Example 5, the destination (register `esi`) is intended to be zeroed out only if the comparison flag in the machine is set; here `gcc` produces clever code to avoid a branch instruction. The target sequence emitted by `gcc` reads the flag to a register `eax`, decrements it (causing it to be either 0 or -1) and then computes the bitwise-and of `eax` and `esi`. Since -1 is represented by all 1s in two's complement, this effectively sets `esi` to zero only if the comparison flag was set. The superoptimizer proposes the use of a simple conditional-move `cmov` instruction to achieve the same result.

A total of around 3000 codesize optimizations and 2100 runtime optimizations were learnt after training the optimizer on a diverse set of integer programs. One metric of importance is the frequency of use of these optimizations. We found a tremendous amount of re-use. Table 2.2 presents a profile of the optimizations that were applied to the SPEC integer benchmarks. Five optimizations were used more than 1,000 times each; in total over 600 distinct optimizations were used at least once each on these benchmarks. To further study the re-use of optimizations, we trained the optimizer on one set of executables and optimized another set of executables. We found that most optimizations are captured even though the executable being optimized was not a part of the training set. For example, 97% of the optimizations were captured when we ran the optimizer on the popular internet browser `firefox` after training it only on the SPEC benchmarks.

Frequency Of Use	Number of Optimizations	Number of Applications
> 1000	8	18679
$201 - 1000$	7	4098
$51 - 200$	33	2823
$11 - 50$	82	1737
$1 - 10$	474	1256

Table 2.2: Profile of the number of optimizations and the number of times they were applied on SPEC CINT2000 benchmarks.

The process of optimizing a full binary using the optimization database is very fast, completing in less than two seconds on these benchmarks. A prototype of our system is available online at [33].

2.6 Discussion

In this section, we show in detail how our system optimizes a simple loop; the purpose is to illustrate what our techniques can, and cannot, do using a small but fairly realistic example. Consider the following C program to traverse a linked list of integers,

Kernel Name	Description	Pseudo-code
sum	Calculate the sum of unsigned byte-integers in an array	sum += a[i]
image-diff	Calculate the sum of absolute differences of image pixels	sum += ABS (a[i] - b[i])
comparison	Compare each element of two arrays	c[i] = (a[i] < b[i]) ? c0 : c1
min	Find the minimum of each element of two arrays	c[i] = (a[i] < b[i]) ? a[i] : b[i]
xor	Computes exclusive-OR over two arrays	c[i] = b[i] ⊕ a[i]
sprite-copy	Rendering sprite graphics (Game Programming)	c[i] = (a[i] == 0) ? b[i] : a[i]

Table 2.3: Superoptimized kernels, operating on arrays of 4 million elements.

multiplying each element by 2:

```
struct node
{
  int val;
  struct node *next;
};

void traverse (struct node *head)
{
  while (head)
  {
    head->val *= 2;
    head = head->next;
  }
}
```

The following assembly code is generated by gcc without optimizations for the loop body of traverse() (eax, edx are machine registers, ebp is the register holding the

		Runtime		Codesize	
Program	Description	Number of Optimizations	Instructions Eliminated	Number of Optimizations	Codesize Improvement
gzip	Data Compression Utility	621	4.16%	402	3.95%
mcf	Minimum Cost Network Flow Solver	381	3.73%	335	5.86%
crafty	Chess Program	1074	2.19%	758	1.71%
bzip2	Data Compression Utility	396	4.11%	301	4.58%
gcc	C compiler	10326	2.44%	2996	1.12%
parser	Natural Language Processing	1123	3.84%	582	3.06%
twolf	Place and Route Simulator	1125	2.17%	619	1.47%

Table 2.4: Results of running the optimizer on SPEC CINT2000 benchmark applications. The runtime improvements are shown over 'gcc -O2' optimization. The codesize improvements are shown over 'gcc -Os'.

frame pointer).

```
1 : movl 8(%ebp), %edx    #edx  := head
2 : movl 8(%ebp), %eax    #eax  := head
3 : movl (%eax), %eax     #eax  := head->val
4 : sall %eax             #left-shift eax by 1
5 : movl %eax, (%edx)     #head->val := eax
6 : movl 8(%ebp), %eax    #eax  := head
7 : movl 4(%eax), %eax    #eax  := head->next
8 : movl %eax, 8(%ebp)    #head := eax
9 : cmpl $0, 8(%ebp)      #head == null?
```

The superoptimizer first replaces instruction 2 with

$2'$:`movl %edx, %eax`

	Description	Target Sequence	Optimal Sequence	Live Registers
1.	Sum of byte-integers in an array	sum += a[i] sum += a[i+1] ... sum += a[i+7]	psubb %mm0, %mm0 psadbw &a[i], %mm0 movd %mm0, sum	sum

2.	eax ← ecx - eax - 1	sub %eax, %ecx
		mov %ecx, %eax
		dec %eax
			notl %eax
			add %ecx, %eax
				eax

3.	Elimination of Branch Instructions	sub %eax, %ecx
		test %ecx, %ecx
		je .END
		mov %edx, %ebx
		.END:
			sub %eax, %ecx
			cmovne %edx, %ebx
				eax, ecx, ebx, ebx

4.	Swap two registers	mov %eax, %ecx
		mov %edx, %eax
		mov %ecx, %edx
			xchg %eax, %edx	eax, edx

5.	Use of Conditional Move Instruction	setg %al movzbl %al, %eax dec %eax and %eax, %esi	mov $0, %eax cmovg %eax, %esi	esi

6.	Constant Folding	mov $8, %eax
		sub %ecx, %eax
		dec %eax
			mov $7, %eax
			sub %ecx, %eax
				eax, ecx

7.	Elimination of Redundant Loads	mov %eax, -20(%ebp) mov -20(%ebp), %ecx	mov %eax, -20(%ebp) mov %eax, %ecx	ecx

Table 2.5: Examples of runtime optimizations performed by the superoptimizer on gcc-optimized executables.

and instruction 9 with

$9'$:`cmpl $0, %eax`

eliminating two redundant loads. Then, the instruction sequence $2', 3, 4, 5$ is replaced with a single instruction

$3'$:`sall (%edx)`

taking advantage of the fact that `eax` is not live at the end of instruction 5. It is inferred that locations `8(%ebp)` and `(%edx)` in instructions 1 and $3'$ cannot alias with each other by comparing the types of instruction operands. Hence, in the third step, the instruction sequence $1, 3', 6$ is replaced by the sequence $1, 3', 6'$ with

$6'$:`movl %edx, %eax`

eliminating another redundant load. Instructions $6'$ and 7 are replaced by

$7'$:`movl 4(%edx), %eax`

eliminating a register copy and finally the use of register `eax` is eliminated in instructions $7'$, 8 and $9'$ by replacing it with `edx` in all three instructions. After these optimizations, the assembly code is:

```
1 : movl 8(%ebp), %edx   #edx  := head
3': sall (%edx)          #left-shift head->val by 1
7': movl 4(%edx), %edx   #edx  := head->next
8': movl %edx, 8(%ebp)   #head := edx
9': cmpl $0x0, %edx      #edx == null?
```

A standard optimizing compiler produces the following code (`eax` holds the value of `head` before entering the loop body):

```
1 : sall (%eax)          #left-shift head->val by 1
2 : movl 4(%eax), %eax   #eax  := head->next
3 : testl %eax, %eax     #eax == null?
```

In this example, our automatically generated optimizer performs all but one of the optimizations performed by a standard optimizing compiler. The optimization that is missed involves the iteration variable (instructions 1 and 8). Because dataflow analysis gives the standard compiler a global view of the loop's behavior across all iterations, the standard compiler can cache the iteration variable (`head`) in a register avoiding loads and stores at loop boundaries. Our rule-based system cannot currently find this optimization because it does not understand loop-carried dependencies. Unrolling the loop a few times would mitigate this limitation since the intermediate loads can still be eliminated by pattern-matching on short sequences of instructions.

2.7 Related Work

Superoptimization of code sequences was first proposed nearly 20 years ago, but we are aware of just three efforts that have developed the idea. Massalin first described an exhaustive-search based approach using a fast probabilistic test to discover short optimal programs[23]. By constraining the set of instructions to a few register-register operations, Massalin was able to scale the length of enumerated programs to 12 instructions. For the probabilistic test, Massalin's superoptimizer chose a set of carefully chosen inputs for the program being optimized. In the first stage, the probabilistic test used 3 hand-chosen input vectors; if the two programs produced identical output on all 3 inputs, the second stage compared the two programs on many more randomly selected inputs. Massalin also used a pruning strategy to eliminate sub-optimal subsequences at every intermediate step. He filtered out instruction sequences that are known not to occur in any optimal program, using the property that any subsequence of an optimal program must also be optimal. To enable this pruning, he used manually coded equivalences between shorter programs. Although Massalin proposes and describes a boolean verifier to determine program equivalences in his paper, his prototype implementation used only a probabilistic test. He used manual inspection to further ascertain equivalence of program pairs. Massalin's superoptimizer was able to test 50,000 programs per second.

We have adopted the same basic approach to searching (enumerating) instruction

sequences, with the addition of simultaneously optimizing many target sequences and reducing the search space using canonicalized instruction sequences. While Massalin was interested in computing optimal programs for mathematical functions (e.g. `signum`), our interest is in computing optimal versions of small instruction sequences found in commonly executed code. Our probabilistic test and pruning strategy are very similar to those proposed by Massalin. We have enhanced the equivalence checker by supporting a large set of instructions (including those accessing memory) and an efficient SAT-based implementation of a boolean equivalence checker. Because we aim to superoptimize several sequences simultaneously, a boolean checker becomes essential to remove false positives produced by the probabilistic test. Massalin's work reported on the optimization of relatively long sequences (12 instructions), at least compared to ours. To achieve such lengths it was necessary to restrict the enumerable instructions to a very small set of 10-15 hand-chosen opcodes. We deal with roughly 300 opcodes, and so the number of instruction sequences for us grows much more rapidly with length. Even though our optimizer can test many more instruction sequences (500,000 per second), our optimizer scales to only length-3 sequences.

The GNU Superoptimizer (GSO) [15] learns optimizations involving elimination of branch instructions for the RS/6000 processor, for later use with the GNU C Compiler (GCC). They use exhaustive search to find the fastest straight-line code computing a goal function. In particular, they find optimal versions of the computation of comparison operators (`A` *rel-op* `B`). This work is perhaps the closest to ours in its goals; we are both interested in learning peephole optimizations. GSO has a large manual component, as a user is required to specify the goal function and if an optimization is found, add it to GCC. Our approach is completely automatic. While GSO has been used to learn a few tens of optimizations, our system has learned thousands and there is no reason the algorithms should not scale to millions of optimizations.

One of GSO's primary goals is to ensure portability across architectures and they achieve it using instruction simulation. We instead choose to directly execute instructions on hardware for speed. Therefore, our optimizer can run only on the target architecture. GSO generates only register-register operations where the output and inputs of the goal functions are assumed to be in specific registers. They prune the

search by trying only operands that are either inputs or have been generated by previous instructions. For three-operand architectures, the destination of each operation is assumed to be the next available register; for two-operand machines, one of the operands is used as the destination. For commutative operations, only one ordering of operands is tried. Using such optimizations, GSO restricts their branching factor to between 100 and 1000. The longest sequence reported in their examples is four instructions long. Unlike GSO, we are interested in optimizing arbitrary sequences including those that modify multiple registers and memory locations. For this reason, our branching factor is bigger (around 2000-3000). Similarly, while GSO's equivalence checker needs to compare only one pair of values (the last generated value and the target function's value), our equivalence checker needs to compare the full representation of the machine state.

Another interesting approach to superoptimization is proposed in a system called Denali [19]. Denali is targeted primarily at optimizing performance-critical inner loops. They divide the problem of finding the optimal sequence into two steps: in the first step, a search procedure finds the space of programs equivalent to the program being searched; and in the second step, they decide the optimal program in the space of equivalent programs. To determine equivalence, Denali requires a set of axioms expressed in first order logic, capturing mathematical operators and the instruction set of the architecture. For example, an axiom could express the fact that integer addition is associative, or that the `leftshift` instruction multiplies its operand by 2. The system then proceeds by matching the program constructs with the corresponding axioms to find all possible ways to compute a goal function and formulates a satisfiability constraint, the solution to which expresses the fastest among all possible equivalent instruction sequences. Because Denali uses goal-directed search, it can find much longer sequences than we can currently generate using exhaustive search. However, Denali has two drawbacks that led us to prefer exhaustive search. First, Denali is dependent on having enough rules (axioms) to cover all interesting cases; we didn't want to rule out optimizations simply because we hadn't thought of them. Second, it is unclear how this approach can be used to optimize several instruction sequences simultaneously; we gain significant efficiency by amortizing the cost of a

single exhaustive enumeration of instruction sequences over the optimization of many target sequences.

Peephole optimizers, apart from their typical use in the final optimization pass, have also been used to perform code selection at link time to generate highly portable compilers [6, 9, 10, 11, 21]. In these systems, peephole optimization through pattern-matching is a primary method to perform code optimization. For example, the "very portable optimizer" (VPO) in [6] uses peephole optimization to reduce the volume of intermediate code by a factor of two to three. These systems share our goal of automatically and systematically discovering peephole optimizations. The primary differences with our work are that our equivalence test based on SAT is more general (able to detect more equivalent sequences) and works for longer sequences than previous systems. Discovering each optimization is also more expensive in our approach; however, by partitioning the work into an off-line learning phase that computes a database of optimizations and an actual optimization phase that simply looks up transformations in the database, our optimization phase can be as fast or faster than traditional peephole optimizers.

2.8 Conclusions and Summary of Contributions

We have described the construction of a system to automatically generate a peephole superoptimizer for a target architecture. The system is capable of automatically learning thousands of peephole optimization rules, each replacing the target sequence with the corresponding optimal sequence. Our superoptimization-based approach is capable of generating efficient code involving SIMD instructions. It is also useful approach to automatically discover many different classes of optimizations in already-compiled code.

Chapter 3

Binary Translation Using Peephole Superoptimizers

In this chapter, we discuss the use of peephole superoptimizers to perform efficient binary translation. We begin with a discussion on the recent applications of binary translation (Section 3.1). We then provide a necessarily brief overview of peephole superoptimizers followed by a discussion on how we employ them for binary translation (Section 3.2). We discuss other relevant issues involved in binary translation (Section 3.3) and go on to discuss our prototype implementation (Section 3.4). We then present our experimental results (Section 3.5), discuss related work (Section 3.6), and finally conclude (Section 3.7).

3.1 Applications of Binary Translation

Before describing our binary translation system, we give a brief overview of a range of applications for binary translation. Traditionally, binary translation has been used to emulate legacy architectures on recent machines. With improved performance, it is now also seen as an acceptable portability solution.

Binary translation is also useful to hardware designers for ensuring software availability for their new architectures. While the design and production of new architecture chips complete within a few years, it can take a long time for software to be

available on the new machines. To deal with this situation and ensure early adoption of their new designs, computer architects often turn to software solutions like virtual machines and binary translation[8].

Another interesting application of binary translation for hardware vendors is backward and forward compatibility of their architecture generations. To run software written for older generations, newer generations are forced to support backward compatibility. On the flip side, it is often not possible to run newer generation software on older machines. Both of these problems create compatibility headaches for computer architects and huge management overheads for software developers. It is not hard to imagine the use of a good binary-translation based solution to solve both problems in the future.

Binary translation is also being used for machine and application virtualization. Leading virtualization companies are now considering support for allowing the execution of virtual machines from multiple architectures on a single host architecture[31]. Hardware vendors are also developing virtualization platforms that allow people to run popular applications written for other architectures on their machines[26]. Server farms and data centers can use binary translation to consolidate their servers, thus cutting their power and management costs.

People have also used binary translation to improve performance and reduce power consumption in hardware. Transmeta Crusoe [20] employs on-the-fly hardware binary translation to execute x86 instructions on a VLIW architecture thereby cutting power costs[16]. Similarly, in software, many Java virtual machines perform on-the-fly binary translation from Java bytecode to the host machine instructions[37] to improve execution performance.

3.2 Binary Translation Using Peephole Superoptimizers

In this section we give a necessarily brief overview of the design and functionality of peephole superoptimizers, focusing on the aspects that are important in the adaptation to binary translation, which is discussed in Section 3.2.

3.2.1 Peephole Superoptimizers

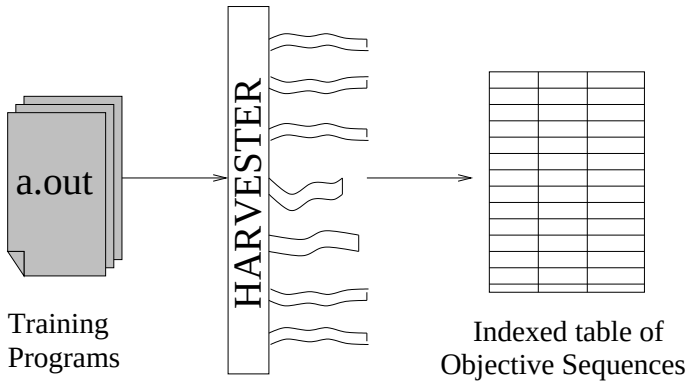


Figure 3.1: In the first phase, the harvester extracts instruction sequences from a set of training executable binaries and constructs an indexed data structure of target sequences

Peephole superoptimizers are an unusual type of compiler optimizer [5, 15], and for brevity we usually refer to a peephole superoptimizer as simply an optimizer. Constructing a peephole superoptimizers has three phases:

1. A module called the *harvester* extracts *target instruction sequences* from a set of training programs (see Figure 3.1). These instruction sequences are the ones we seek to optimize.

2. A module called the *enumerator* enumerates all possible instruction sequences up to a certain length. Each enumerated instruction sequence s is checked to see if it is equivalent to any target instruction sequence t. If s is equivalent to some target sequence t and s is cheaper according to a cost function (e.g., estimated

execution time or code size) than any other sequence known to be equivalent to t (including t itself), then s is recorded as the best known replacement for t (see Figure 3.2). A few sample peephole optimization rules are shown in Table 3.1.

3. The learned (target sequence, optimal sequence) pairs are organized into a lookup table indexed by target instruction sequence.

Once constructed the optimizer is applied to an executable by simply looking up target sequences in the executable for a known better replacement (see Figure 3.3). The purpose of using harvested instruction sequences is to focus the search for optimizations on the code sequences (usually generated by other compilers) that appear in actual programs. Typically all instruction sequences up to length 5 or 6 are harvested, and the enumerator tries all instruction sequences up to length 3 or 4. Even at these lengths, there are billions of enumerated instruction sequences to consider, and techniques for pruning the search space are very important [5]. Thus, the construction of the peephole optimizer is time-consuming, requiring a few processor-days. In contrast, actually applying the peephole optimizations to a program typically completes within a few seconds.

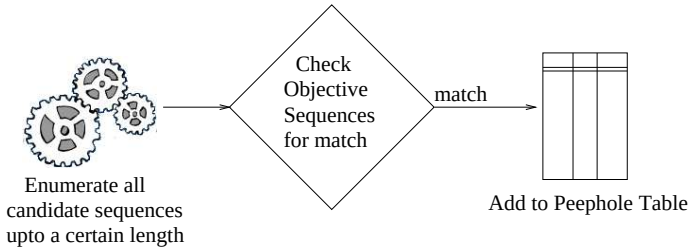


Figure 3.2: In the second phase, the enumerator enumerates all instruction sequences up to a certain length, checking each of them with any of the target sequences for a match. If a suitable match is found, the corresponding replacement rule is added to the peephole table.

The enumerator's equivalence test is performed in two stages: a fast execution test and a slower boolean test. The execution test is implemented by executing the target sequence and the enumerated sequence on hardware and comparing their outputs on random inputs. If the execution test does not prove that the two sequences

are different (i.e., because they produce different outputs on some tested input), the boolean test is used. The equivalence of the two instruction sequences is expressed as boolean formula: each bit of machine state touched by either sequence is encoded as a boolean variable, and the semantics of instructions is encoded using standard logical connectives. A SAT solver is then used to test the formula for satisfiability, which decides whether the two sequences are equal.

Target Sequence	Live Registers	Equivalent Enumerated Sequence
`movl (%eax), %ecx` `movl %ecx, (%eax)`	`eax,ecx`	`movl (%eax), %ecx`

`sub %eax, %ecx`
`mov %ecx, %eax`
`dec %eax`
	`eax`	`not %eax`
		`add %ecx, %eax`

`sub %eax, %ecx`
`test %ecx, %ecx`
`je .END`
`mov %edx, %ebx`
`.END:`
	`eax, ecx,`
	`edx, ebx`
		`sub %eax, %ecx`
		`cmovne %edx, %ebx`

Table 3.1: Examples of peephole rules generated by a superoptimizer for x86 executables

.

Using these techniques, *all* length-3 x86 instruction sequences have previously been enumerated on a single processor in less than two days[5]. This particular superoptimizer is capable of handling opcodes involving flag operations, memory accesses and branches, which on most architectures covers almost all opcodes. Equivalence of instruction sequences involving memory accesses is correctly computed by accounting for the possibility of aliasing. The optimizer also takes into account live register information, allowing it to find many more optimizations because correctness only requires that optimizations preserve live registers (note the live register information qualifying the peephole rules in Table 3.1).

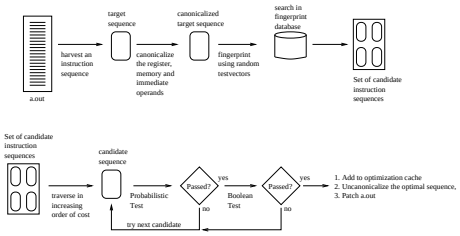


Figure 3.3: In the final phase, the optimizations in the peephole table are applied to the target executable.

3.2.2 Binary Translation

Here, we discuss how we use a peephole superoptimizer to perform efficient binary translation. The approach is similar to that discussed in Section 3.2.1, except that now our target sequences belong to the source architecture while the enumerated sequences belong to the destination architecture.

The binary translator's harvester first extracts target sequences from a training set of source-architecture applications. The enumerator then enumerates instruction sequences on the destination architecture checking them for equivalence with any of the target sequences. A key issue is that the definition of equivalence must change in this new setting with different machine architectures. Now, equivalence is meaningful only with respect to a *register map* showing which memory locations on the destination machine, and in particular registers, emulate which memory locations on the source machine. Some valid register maps are shown in Table 3.2. A register in the source architecture could be mapped to a register or a memory location in the destination architecture. It is also possible for a memory location in the source architecture to be mapped to a register in the destination architecture. The choice of the register determines the renaming of registers in performing a translation from a source sequence to an equivalent target sequence.

During enumeration, all possible register maps are enumerated and a corresponding target sequence searched. We reduce the search space by observing that having once considered a register map, we need never consider a register map that is equal up to a consistent register renaming. In case a match is found, the corresponding peephole rule is added to the translation table. The peephole rule now has an extra

Register Map	Description
`r1`→`eax`	Maps PowerPC register to x86 register
`r1`→$\mathtt{M}_1$	Maps PowerPC register to a memory location
$\mathtt{M}_s$→`eax`	Maps a memory location in source code to a register in the translated code
`r1`→`eax` `r2`→`eax`	Invalid. Cannot map two PowerPC registers to the same x86 register
$\mathtt{M}_s$→$\mathtt{M}_t$	Maps one memory location to another (e.g. address space translation)

Table 3.2: Some valid (and invalid) register maps from PowerPC-x86 translation (M_i refers to a memory location).

field specifying the register map under which it is valid. Some examples of peephole translation rules are shown in Table 3.3.

Once the binary translator is constructed, using it is relatively simple. The translation rules are applied to the source-architecture code to obtain destination-architecture code. The application of translation rules is more involved than the application of optimization rules. Now, we also need to select the register map for each code point before generating the corresponding translated code. The right choice of register maps can make a visible difference to the performance of generated code. We discuss the selection of optimal register maps at translation time in Section 3.3.2.

3.3 Other Issues in Binary Translation

In this section, we discuss the main issues relevant to our approach to binary translation.

PowerPC Sequence	Live Registers	State Map	x86 Instruction Sequence
`mr r1,r2`	`r1,r2`	`r1`→`eax` `r2`→`ecx`	`movl %ecx,%eax`
`mr r1,r2`	`r1,r2`	`r1`→`eax` `r2`→M_1	`movl` M_1`,%eax`
`lwz r1,(r2)`	`r1,r2`	`r1`→`eax` `r2`→`ecx`	`movl (%ecx),%eax` `bswap %eax`

`lwz r1,(r2)`
`stw r1,(r3)`
	`r1,r2,`
	`r3`
		`r1`→`eax`
		`r2`→`ecx`
		`r3`→`edx`
			`movl (%ecx),%eax`
			`movl %eax,(%edx)`

`mflr r1`	`r1,lr`	`r1`→`eax` `lr`→`ecx`	`movl %ecx,%eax`

Table 3.3: Examples of peephole translation rules from PowerPC to x86.

3.3.1 Static vs Dynamic Translation

Binary translation can either be performed statically (compile-time) or dynamically (runtime). Most existing tools perform binary translation dynamically for its primary advantage of having a complete view of the current machine state. Moreover, dynamic binary translation provides additional opportunities for runtime optimizations. The drawback of dynamic translation is the overhead of performing translation and book-keeping at runtime, which is especially visible while running small user-interactive applications that are invoked multiple times, such as many desktop applications. A static translator translates programs offline and can apply more extensive (and potentially whole program) optimizations. However, performing faithful static translation is a slightly harder problem since no assumptions can be made about the runtime state of the process.

Our binary translator is static, though we have avoided including anything in our implementation that would make it impractical to develop a dynamic translator (e.g., whole-program analysis or optimizations) using the same algorithms. Most of the techniques we discuss are equally applicable in both settings and when they are not, we discuss the two separately.

3.3.2 Register Maps

While translating code from one architecture to another, we need to choose which registers (or memory locations) on the destination machine will emulate which registers on the source machine. Choosing a good *register map* is crucial to the quality of translation, and moreover the best code may require changing the register map from one code point to the next. Thus, the best register map is the one that minimizes the cost of the peephole translation rule (generates the fastest code) plus any cost of switching register maps from the previous program point—because switching register maps requires adding register move instructions to the generated code to realize the switch at run-time, switching register maps is not free.

We formulate a dynamic programming problem to choose a minimum cost register map at each program point in a given code region. At each code point all feasible register maps are enumerated. For each enumerated register map M, the peephole translation table is queried for a matching translation rule T and the corresponding translation cost is recorded. Assume for simplicity that the code point under consideration has only one predecessor, and the possible register maps at the predecessor are $P_1, \ldots, P_n$. The best cost register map is the one P_i that minimizes the cost of switching from P_i to M, the cost of T, and, recursively, the cost of P_i:

$$cost(M) = cost(T) + min_i(cost(P_i) + switch(P_i, M))$$

We solve the recurrence in a standard fashion. Beginning at start of a code region (e.g., a function body), the cost of the preceding register map is initially 0. Working forwards through the code region, the cost of each enumerated register map is computed and stored before moving to the next program point and repeating the computation. When the end of the code region is reached, the register map with the lowest cost is chosen and its decisions are backtracked to decide the register maps at all preceding program points. For program points having multiple predecessors, we use a weighted sum of the switching costs from each predecessor. The weights as a proxy for profiling or other hints that would tell us how frequently each code path is taken. To handle loops, we perform two iterations of this computation.

An Example

We use an example to further explain our algorithm. Consider a function `foo` with three PowerPC instructions:

```
foo:
   mr r2, r1
   mr r1, r3
   mr r3, r2
   blr
```

`foo` swaps the registers `r1` and `r3` using register `r2` as a temporary store. For simplicity, we assume that all three registers (namely `r1`, `r2` and `r3`) are live at the end of the function. In Table 3.4, we show the peephole translation rules relevant to this example. A row in the table represents that a PowerPC instruction sequence in Column 1 can be translated to the x86 instruction sequence in Column 3 if the registers at that program point are mapped according to Column 2. For example, the first rule states that the instruction `mr r1, r2` can be translated to `mov M, R` if PowerPC registers `r1` and `r2` are mapped to the x86 register `R` and memory location `M` respectively. The cost of using a peephole translation rule (column 4) is the cost of the corresponding x86 instruction sequence; our cost function captures the approximate runtime of the x86 sequence. The other significant component of the cost is the cost of switching register assignments. Table 3.5 gives the switching costs for a single PowerPC register. The table represents that the cost of switching from either register to memory or vice versa has the cost of a memory access (which is 10 in our cost model), while the cost to remain in the same state is 0.

We now describe the solution of our dynamic programming formulation for a straight line sequence of PowerPC code. At each step in our algorithm, w move forward by one PowerPC instruction. At the end of each step, we would have computed the best possible translation and it's associated cost for each register map.

In our example, we start with cost 0 at function entry `foo`. At this point all registers are assumed to be in memory. At Step 1, we consider all possible ways to translate the first instruction. There are three valid possibilities (depending on the

PowerPC Sequence	Map	x86 Sequence	Cost
`mr r1, r2`	r1 → R r2 → M	`mov M, R`	10
`mr r1, r2`	r1 → M r2 → R	`mov R, M`	10
`mr r1, r2`	r1 → R_1 r2 → R_2	`mov` R_1, R_2	1

`mr r1, r2`
`mr r2, r3`
`mr r3, r1`
	r1 → M
	r2 → R_2
	r3 → R_3
		`mov` R_2, M
		`xchg` R_2, R_3
			11

Table 3.4: An example table of peephole translation rules.

Transition	Cost
R⟶M	10
M⟶R	10
R⟶R	0
M⟶M	0

Table 3.5: Switching Costs

register map used):

- Use peephole rule 1 with `r1`←`R` and `r2`←`M`. The peephole rule cost in this case is 10 while the switching cost from the previous program point (where both PowerPC registers were in memory) is 10, totalling to 20.
- Use peephole rule 2 with `r1`←`M` and `r2`←`R`. This is identical to the first case, with a total cost of 20.
- Use peephole rule 3 with `r1`←R_1 and `r2`←R_2. In this case, the peephole rule cost is 1, while the switching cost is 20 since it involves bringing two PowerPC registers from memory to x86 registers.

At this point, the minimum cost translation for the first instruction is 20, obtained by using either one of peephole rules 1 and 2. We store all these three possibilities to compute the best translations at the next step.

At Step 2, we search for the minimum cost translation for the first two instructions. In this case, there are 6 possibilities for register maps at instruction 2, three of which we discuss below. (The other three possibilities are very similar to the ones discussed).

- Use peephole rule 1 with `r1`←`R`, `r2`←$\mathtt{M}_1$ and `r3`←$\mathtt{M}_2$. The lowest cost translation is achieved by using the register map `r1`←`R` at the previous instruction. The total cost in this case is $30 = 20 + 10$ where 20 is the cost of the previous translation and 10 is the cost the peephole rule. The switching cost in this case is 0.

- Use peephole rule 3 with `r1`←$\mathtt{R}_1$, `r2`←`M` and `r3`←$\mathtt{R}_2$. In this case, the lowest cost translation is achieved by using the register map (`r1`←`R`, `r2`←`M`) at the previous instruction. The total cost in this case is $31 = 20 + 10 + 1$ where 20 is the cost of the previous translation, 10 is the switching cost of bringing `r3` into an x86 register and 1 is the cost of the peephole rule.

- Use peephole rule 3 with all three registers `r1`, `r2` and `r3` mapped to x86 registers $\mathtt{R}_1$, $\mathtt{R}_2$ and $\mathtt{R}_3$ respectively. At this point, the best register map at the previous instruction is (`r1`←$\mathtt{R}_1$, `r2`←$\mathtt{R}_2$). The total cost in this case is $32 = 21 + 10 + 1$, where 21 is the cost of the previous translation, 10 is the switching cost from the previous register map to this one, and 1 is the cost of the peephole rule.

Next, we attempt to match the instruction sequence formed by the first two instructions to one of the rules in the peephole table. In this case, no matches exist and so, we move on to the next instruction. We would like to point out that since the optimal register map at a program point does not depend on the register maps at program points before the predecessor program point, it suffices to store only the current register maps and their optimal costs.

At Step 3, a similar procedure is used to compute the costs of the possible register maps at the third instruction. The final costs at the end of the third instruction are shown in Figure 3.4 for each register map. As seen in the figure, the minimum cost achieved by considering all single instruction matches is 33. Next, we attempt to match the instruction sequence formed by the second and third instructions with

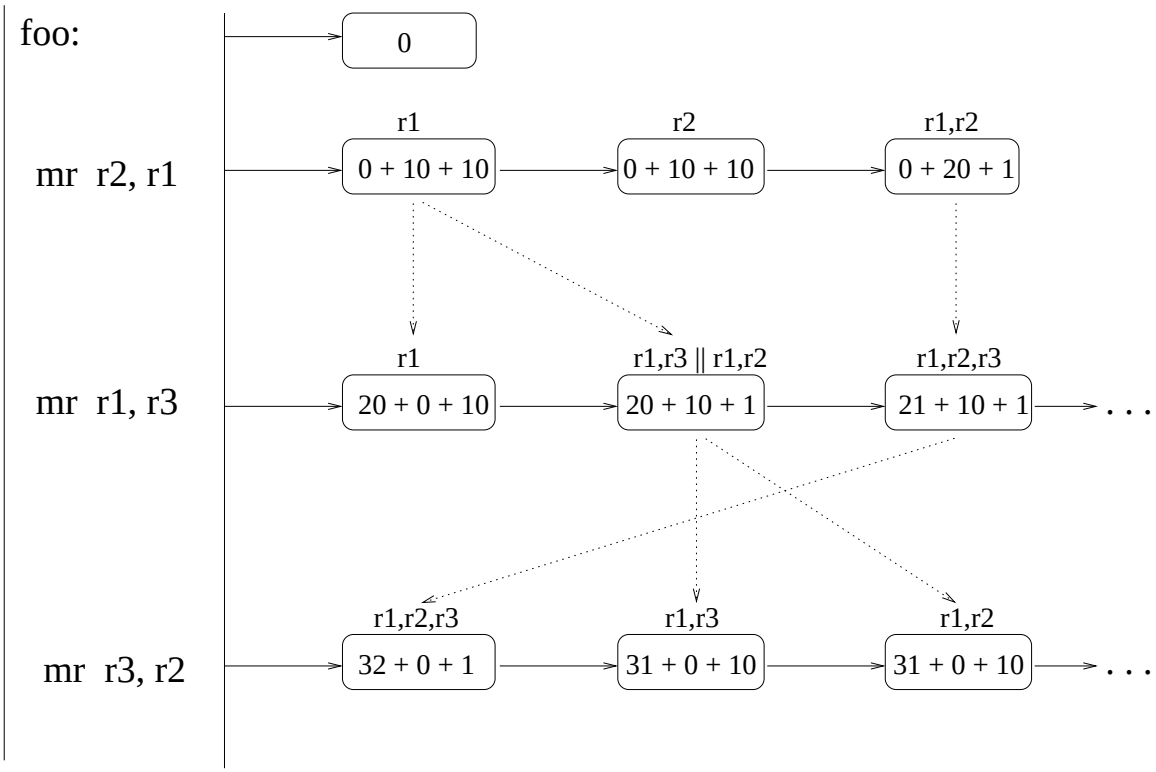


Figure 3.4: The enumerated register maps for the example. Each box represents an enumerated register map. The top label on the box indicates the PowerPC registers that are mapped to x86 registers (the other PowerPC registers are in memory). The value in the box is the minimum cost of using that register map at that program point. The cost is represented as (predecessor-cost + switching-cost + peephole-rule-cost). The dotted lines indicate the predecessor used to obtain the minimum cost.

a rule in the peephole table. We find no match in this case. Finally, we attempt to match the sequence formed by all three instructions. In this case, peephole rule 4 matches the three instructions with `r1`$\leftarrow$`M`, `r2`$\leftarrow$$\mathtt{R}_1$ and `r3`$\leftarrow$$\mathtt{R}_2$ with a total cost of $31 = 20 + 11$ (here, 20 is the cost of switching and 11 is the cost of the peephole rule).

At the end of the three instructions, the minimum cost achieved is 31 by using peephole rule 4, and that is used as the final translation of the function.

This procedure of enumerating all register maps and then solving a dynamic programming problem is computationally intensive and, if not done properly, can significantly increase translation time. While the cost of finding the best register map

for every code point is not a problem for a static translator, it would add significant overhead to a dynamic translator. To bound the computation time, we prune the set of enumerated register maps at each program point. We retain only the n lowest-cost register maps before moving to the next program point. We allow the value of n to be tunable and refer to it as the *prune size*. We also have the flexibility to trade computation time for lower quality solutions. For example, for code that is not performance critical we can consider code regions of size one (e.g., a single instruction) or even use a fixed register map. In Section 3.5 we show that the cost of computing the best register maps for frequently executed instructions is very small for our benchmarks. We also discuss the performance sensitivity of our benchmarks to the prune size.

3.3.3 Endianness

If the source and destination architectures have different endianness we convert all memory reads to destination endianness and all memory writes to source endianness. This policy ensures that memory is always in source endianness while registers have destination endianness. The extra byte-swap instructions needed to maintain this invariant are only needed on memory accesses; in particular, we avoid the additional overhead of shuffling bytes on register operations.

While dealing with source-destination architecture pairs with different endianness, special care is required in handling OS-related data structures. In particular, all executable headers, environment variables and program arguments in the program's address space need to be converted from destination endianness to source endianness before transferring control to the translated program. This step is necessary because the source program assumes source endianness for everything while the OS writes the data structures believing that the program assumes destination endianness. In a dynamic translator, these conversions are performed inside the translator at startup. In a static translator, special initialization code is emitted to perform these conversions at runtime.

3.3.4 Control Flow Instructions

Like all other opcodes, control flow instructions are also translated using peephole rules. Direct jumps in the source are translated to direct jumps in the translated code, with the jump destination in being appropriately adjusted to point to the corresponding translated code. To handle conditional jumps, the condition codes of the source architecture need to be faithfully represented in the destination architecture. Handling condition codes correctly is one of the more involved aspects of binary translation because of the divergent condition-code representations used by different architectures. We discuss our approach to handling condition codes in the context of our PowerPC-x86 binary translator; see Section 3.4.3. The handling of indirect jumps is more involved and is done differently for static and dynamic translators. We discuss this in detail in Section 3.4.4.

3.3.5 System Calls

When translating across two different operating systems, each source OS system call needs to be emulated on the destination OS. Even when translating across the same operating system on different architectures, many system calls require special handling. For example, some system calls are only implemented for specific architectures. Also, if the two architectures have different endianness, proper endianness conversions are required for all memory locations that the system call could read or write.

There are other relevant issues to binary translation. For example, different issues exist in full system emulation vs user-level emulation. A full system emulator needs to also emulate the chipset and other peripherals of the source architecture, while a user level emulation can abstract these issues at system-call interface. Other examples include precise exceptions, misaligned memory accesses, interprocess communication, signal handling, etc. These problems are orthogonal to the issues in peephole binary translation and our solutions to these issues are standard. In this work, our focus is primarily on efficient code-generation.

3.4 Implementation

We have implemented a binary translator that allows PowerPC/Linux executables to run in an x86/Linux environment. The translator is capable of handling almost all PowerPC opcodes (around 180 in all). We have tested our implementation on a variety of different executables and libraries.

The translator has been implemented in C/C++ and O'Caml [22]. Our superoptimizer is capable of automatically inferring peephole translation rules from PowerPC to x86. To test equivalence of instruction sequences, we use zChaff [25, 38] as our backend SAT solver. We have translated most, but not all, Linux PowerPC system calls. We present our results using the static translator that produces an x86 ELF 32-bit binary executable from a PowerPC ELF 32-bit binary. Because we used the static peephole superoptimizer described in [5] as our starting point, our binary translator is also static, though as discussed previously our techniques could also be applied in a dynamic translator. A consequence of our current implementation is that we can also translate all dynamically linked libraries used by the PowerPC program.

3.4.1 Endianness

PowerPC is a big-endian architecture while x86 is a little-endian architecture, which we handle using the scheme outlined in Section 3.3.3. For integer operations, there exist three operand sizes in PowerPC: 1, 2 and 4 bytes. Depending on the operand size, the appropriate conversion code is required when reading from or writing to memory. We employ the convenient `bswap` x86 instruction to generate efficient conversion code.

3.4.2 Stack and Heap

On Linux, the stack is initialized with `envp`, `argc` and `argv` and the stack pointer is saved to a canonical register at load time. On x86, the canonical register storing the stack pointer is `esp`; on PowerPC, it is `r1`. When the translated executable is loaded in an x86 environment (in the case of dynamic translation, when the translator is loaded), the `esp` register is initialized to the stack pointer by the operating system

while the emulated `r1` register is left uninitialized. To make the stack visible to the translated PowerPC code, we copy the `esp` register to the emulated `r1` register at startup. In dynamic translation, this is done by the translator; in static translation, this is done by the initialization code. The handling of the heap requires no special effort since the `brk` Linux system call used to allocate heap space is identical on both x86 and PowerPC.

3.4.3 Condition Codes

Condition codes are bits representing quantities such as carry, overflow, parity, less, greater, equal, etc. PowerPC and x86 handle condition codes very differently. Figures 3.5 and 3.6 show how condition codes are represented in PowerPC and x86 respectively.

While PowerPC condition codes are written using separate instructions, x86 condition codes are overwritten by almost all x86 instructions. Moreover, while PowerPC compare instructions explicitly state whether they are doing a signed or an unsigned comparison and store only one result in their flags, x86 compare instructions perform both signed and unsigned comparisons and store both results in their condition bits. On x86, the branch instruction then specifies which comparison it is interested in (signed or unsigned). We handle these differences by allowing the PowerPC condi-

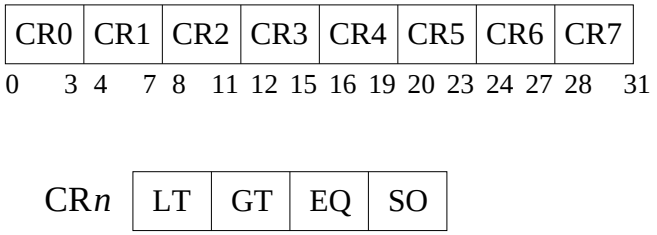


Figure 3.5: PowerPC architecture has support for eight independent sets of condition codes `CR0-CR7`. Each 4-bit `CRn` register uses one bit each to represent less than (`LT`), greater (`GT`), equal (`EQ`) and overflow-summary (`SO`). Explicit instructions are required to read/write the condition code bits.

tion registers (`cr0-cr7`) to be mapped to x86 flags in the register map. For example,

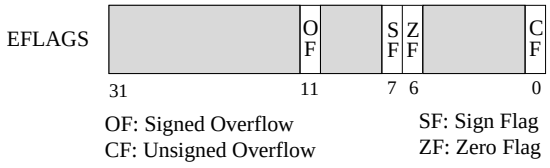


Figure 3.6: The x86 architecture supports only a single set of condition codes represented as bits in a 32-bit `EFLAGS` register. Almost all x86 instructions overwrite these condition codes.

an entry `cr0`→`SF` in the register map specifies that, at that program point, the contents of register `cr0` are encoded in the x86 signed flags (`SF`). The translation of a branch instruction then depends on whether the condition register being used ($\mathtt{cr}_i$) is mapped to signed (`SF`) or unsigned (`UF`) flags.

3.4.4 Indirect Jumps

Jumping to an address in a register (or a memory location) is an *indirect* jump. Function pointers, dynamic loading, and case statements are all handled using indirect jumps. Since an indirect jump could jump almost anywhere in the executable, it requires careful handling. Moreover, since the destination of the indirect jump could assume a different register-map than the current one, the appropriate conversion needs to be performed before jumping to the destination . Different approaches for dealing with indirect jumps are needed in static and dynamic binary translators.

Handling an indirect jump in a dynamic translator is simpler. Here, on encountering an indirect jump, we relinquish control to the translator. The translator then performs the register map conversion before transferring control to the (translated) destination address.

Handling an indirect jump in a static translator is more involved. We first identify all instructions that can be possible indirect jump targets. Since almost all well-formed executables use indirect jumps in only a few different code paradigms, it is possible to identify possible indirect jump targets by scanning the executable. We scan the read-only data sections, global offset tables and instruction immediate operands and use a set of pattern matching rules to identify possible indirect

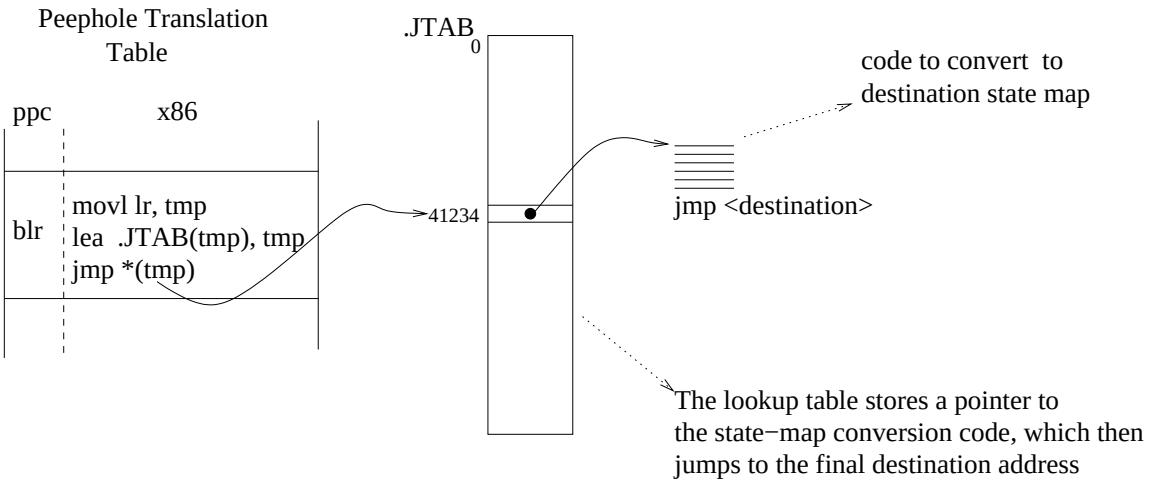


Figure 3.7: Handling of indirect jumps in a static binary translator. An indirect jump is translated to a table lookup and a jump to the corresponding address. The lookup table stores a pointer to a code fragment that first performs state-map conversion before jumping to the translated code.

jump targets. A lookup table is then constructed to map these jump targets (which are source architecture addresses) to their corresponding destination architecture addresses. However, as we need to perform register map conversion before jumping to the destination address at runtime, we replace the destination addresses in the lookup table with the address of a code fragment that performs the register-map conversion before jumping to the destination address. We illustrate this scheme in Figure 3.7.

The translation of an indirect jump involves a table lookup and some register-map conversion code. While the table lookup is fast, the register-map conversion may involve multiple memory accesses. Hence, an indirect jump is usually an expensive operation.

Although the pattern matching rules we use to identify possible indirect jump targets have worked extremely well in practice, they are heuristics and particularly are prone to adversarial attacks. It would not be difficult to construct an executable that exploits these rules to cause a valid PowerPC program to crash on x86. Hence, in an adversarial scenario, it would be wise to assume that all code addresses are possible indirect jump targets. Doing so results in a larger lookup table and more conversion code fragments, increasing the overall size of the executable, but will have

ppc	x86	Comparison
`bl`	`call`	`bl` (branch-and-link) saves the instruction pointer to register `lr` while `call` pushes it to stack
`blr`	`ret`	`blr` (branch-to-link-register) jumps to the address pointed-to by `lr`, while `ret` pops the instruction pointer from the stack and jumps to it

Table 3.6: Function call and return instructions in PowerPC and x86 architectures

no effect on running time apart from possible cache effects.

3.4.5 Function Calls and Returns

Function calls and returns are handled in very different ways in PowerPC and x86. Table 3.6 lists the instructions and registers used in function calls and returns for both architectures.

We implement function calls of the PowerPC architecture by simply emulating the link-register(`lr`) like any other PowerPC register. On a function call (`bl`), the link register is updated with the value of the next PowerPC instruction pointer. A function return (`blr`) is treated just like an indirect jump to the link register.

The biggest advantage of using this scheme is its simplicity. However, it is possible to improve the translation of the `blr` instruction by exploiting the fact that `blr` is always used to return from a function. For this reason, it is straightforward to predict the possible jump targets of `blr` at translation time (it will be the instruction following the function call `bl`). This information can be used to avoid the extra memory reads and writes required for register map conversion in an indirect jump. We have implemented this optimization; while this optimization provides significant improvements while translating small recursive benchmarks (e.g., recursive computation of the fibonacci series), it is not very effective for larger benchmarks (e.g., SPEC CINT2000).

Opcode	Registers	Description
`mul` *reg32*	`eax, edx`	Multiplies *reg32* with eax and stores the 64-bit result in `edx:eax`.
`div` *reg32*	`eax, edx`	Divides `edx:eax` by *reg32* and stores result in `eax`.
any 8-bit insn	`eax, ebx` `ecx, edx`	8-bit operations can only be performed on these four registers.

Table 3.7: Examples of x86 instructions that operate only on certain fixed registers.

3.4.6 Register Name Constraints

Another interesting challenge while translating from PowerPC to x86 is dealing with instructions that operate only on specific registers. Such instructions are present on both PowerPC and x86. Table 3.7 shows some such x86 instructions.

To be able to generate peephole translations involving these special instructions, the superoptimizer is made aware of the constraints on their operands during enumeration. If a translation is found by the superoptimizer involving these special instructions, the generated peephole rule encodes the name constraints on the operands as *register name constraints*. These constraints are then used by the translator at code generation time.

3.4.7 Self-Referential and Self-Modifying Code

We handle self-referential code by leaving a copy of the source architecture code in its original address range for the translated version. To deal with self-modifying code and dynamic loading, we invalidate the translation of a code fragment on observing any modification to that code region. We achieve this by trapping any writes to code regions and performing the corresponding invalidation and re-translation. For a static

Number of Additions	Reason
2	Overflow/underflow semantics of the divide instruction (`div`)
2	Overflow semantics of `srawi` shift instruction
1	The rotate instruction `rlwinm`
1	The `cntlzw` instruction
1	The `mfcr` instruction
9	Indirect jumps referencing the jumptable

Table 3.8: The distribution of the manual translation rules we added to the peephole translation table.

translator, this involves making the translator available as a shared library.

3.4.8 Untranslated Opcodes

For 16 PowerPC opcodes our translator failed to find a short equivalent x86 sequence of instructions automatically. In such cases, we allow manual additions to the peephole table. Table 3.8 describes the number and types hand additions: 9 are due to instructions involving indirect jumps and 7 are due to complex PowerPC instructions that cannot be emulated using a bounded length straight-line sequence of x86 instructions. For some more complex instructions mostly involving interrupts and other system-related tasks, we used the slow but simple approach of emulation using C-code.

3.4.9 Compiler Optimizations

An interesting observation while doing our experiments was that certain compiler optimizations often have an adverse effect on the performance of our binary translator. For example, an optimized PowerPC executable attempts to use all the 8 condition-registers (`cr0-cr7`). However, since x86 has only one set of flags, other condition

registers need to be emulated using x86 registers causing extra register pressure. Another example of an unfriendly compiler optimization is instruction scheduling. An optimizing PowerPC compiler separates two instructions involving a data dependency in order to minimize pipeline stalls, while our binary translator would like the data-dependent instructions to be together to allow the superoptimizer to suggest more aggressive optimizations. To alleviate this issue, we re-order the instructions in a basic block to cluster data-dependent instructions together. In our experiments, we discuss the advantage of using this optimization.

Finally, we would like to point out that while there exist these architecture-specific issues, the vast bulk of the translation and optimization complexity is still hidden by the superoptimizer.

3.5 Experimental Results

We performed our experiments using a Linux machine with a single Intel Pentium 4 3.0GHz processor, 1MB cache and 4GB of memory. We used `gcc` version 4.0.1 and `glibc` version 2.3.6 to compile the executables on both Intel and PowerPC platforms. To produce identical compilers, we built the compilers from their source tree using exactly the same configuration options for both architectures. While compiling our benchmarks, we used the `-msoft-float` flag in `gcc` to emulate floating point operations in software; our translator currently does not translate floating point instructions. For all our benchmarks except one, emulating floating point in software makes no difference in performance. All the executables were linked statically and hence, the libraries were also converted from PowerPC to x86 at translation time. To emulate some system-level PowerPC instructions, we borrowed C-code from the open source emulator Qemu[28].

In our experiments, we compare the executable produced by our translator to a natively-compiled executable. The experimental setup is shown in Figure 3.8. We compile from the C source for both PowerPC and x86 platforms using `gcc`. The

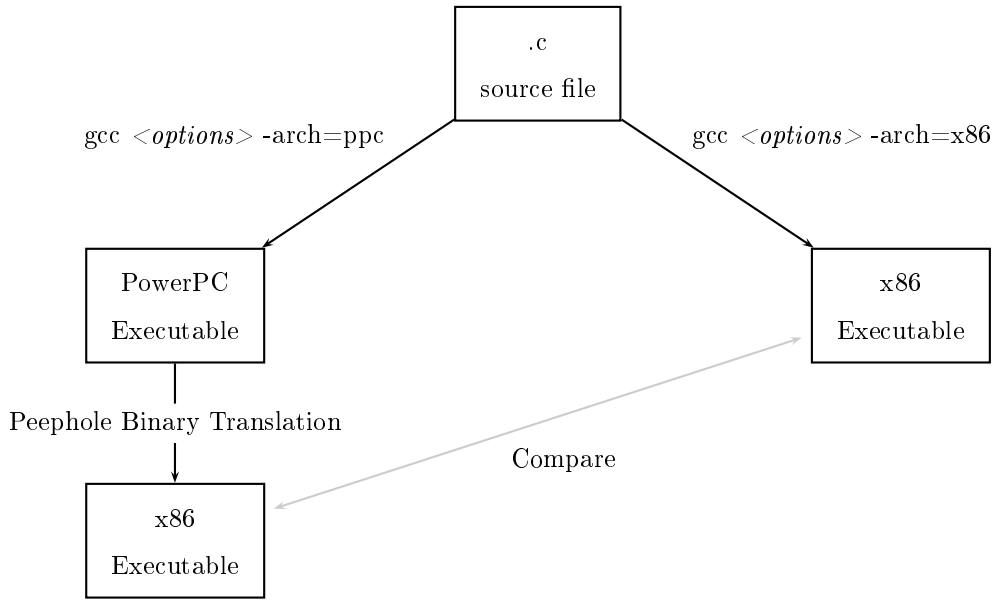


Figure 3.8: Experimental Setup. The translated binary executable is compared with the natively-compiled x86 executable. While comparing, the same compiler optimization options are used on both branches.

same compiler optimization options are used for both platforms. The PowerPC executable is then translated using our binary translator to an x86 executable. And finally, the translated x86 executable is compared with the natively-compiled one for performance.

One would expect the performance of the translated executable to be strictly lower than that of the natively-compiled executable. To get an idea of the state-of-the-art in binary translation, we discuss two existing binary translators. A general-purpose open-source emulator, Qemu[28], provides 10-20% (i.e., 5-10x slowdown) of the performance of a natively-compiled executable. A recent commercially available tool by Transitive Corporation[34] claims "typically about 70-80%" of the performance of a natively-compiled executable on their website[29]. Both Qemu and Transitive are dynamic binary translators.

Benchmark	Description	-O0	-O2	-O2+
emptyloop	A bounded for-loop doing nothing	98.56 %	128.72 %	127 %
fibo	Compute first few Fibonacci numbers	118.90 %	319.13 %	127.78 %
quicksort	Quicksort on 64-bit integers	81.36 %	92.61 %	90.23 %
mergesort	Mergesort on 64-bit integers	83.22 %	91.54 %	84.35 %
bubblesort	Bubble-sort on 64-bit integers	75.12 %	70.92 %	64.86 %
hanoi1	Towers of Hanoi Algorithm 1	84.83 %	70.03 %	61.96 %
hanoi2	Towers of Hanoi Algorithm 2	107.14 %	139.64 %	143.69 %
hanoi3	Towers of Hanoi Algorithm 3	81.04 %	90.14 %	80.15 %
traverse	Traverse a linked list	69.06 %	67.67 %	67.15 %
binsearch	Perform binary search on a sorted array	65.38 %	61.24 %	62.15 %

Table 3.9: Performance of the binary translator on some compute-intensive microbenchmarks. The columns represent the optimization options given to `gcc`. '`-O2+`' expands to '`-O2 -fomit-frame-pointer`'. '`-O2+`' omits storing the frame pointer on x86. On PowerPC, '`-O2+`' is identical to '`-O2`'. The performance is shown relative to a natively compiled application (the performance of a natively compiled application is 100%).

Table 3.9 shows the performance of our binary translator on small compute-intensive microbenchmarks. Our microbenchmarks use three well-known sorting algorithms, three different algorithms to solve the towers of hanoi problem, one benchmark that computes the Fibonacci sequence, a link-list traversal, a binary search on a sorted array, and an empty for-loop. All these programs are written in C. They are all highly compute intensive and hence designed to stress-test the performance of binary translation.

The translated executables perform roughly at 90% of the performance of a natively-compiled executable on average. Some benchmarks perform as low as 64%

	O0					O2				
	x86 (s)	peep (s)	qemu (%)	rosetta (%)	peep (%)	x86 (s)	peep (s)	qemu (%)	rosetta (%)	peep (%)
`bzip2`	311	470	18.5	65.3	66.2	195	265	25.0	54.0	73.7
`gap`	165	313	-	-	52.5	87	205	15.7	-	42.5
`gzip`	264	398	15.3	58.7	66.3	178	315	20.9	52.5	56.5
`mcf`	193	221	46.5	84.8	87.3	175	184	64.7	81.5	94.7
`parser`	305	520	16.9	54.4	58.7	228	338	22.5	49.0	67.3
`twolf`	2184	1306	55.6	-	167.2	1783	1165	59.1	-	153
`vortex`	193	463	11.3	43.1	41.7	161	-	-	38.0	-

Table 3.10: Performance of the binary translator on SPEC CINT2000 benchmark applications. The x86 column represents the performance of a natively compiled application. The percentage(%) fields represent performance relative to the x86 performance (the performance of a natively compiled application is 100%).'-' entries represent failed translations. `peep` columns represent the performance of our translator. `qemu` and `rosetta` represent Qemu and Apple Rosetta respectively.

of native performance and many benchmarks outperform the natively compiled executable. The latter result is a bit surprising. For unoptimized executables, the binary translator often outperforms the natively compiled executable, because the superoptimizer performs optimizations that are not seen in an unoptimized natively compiled executable. The bigger surprise occurs when the translated executable outperforms an already optimized executable (columns `-O2` and `-O2+`) indicating that even mature optimizing compilers today are not producing the best possible code. When compared with Apple Rosetta, our translator consistently performs better than Rosetta on all these microbenchmarks. On average, our translator is 170% faster than Apple Rosetta on these small programs.

A striking result is the performance of the `fibo` benchmark in the `-O2` column where the translated executable is almost three times faster than the natively-compiled and optimized executable. On closer inspection, we found that this is because `gcc`, on x86, uses one dedicated register to store the frame pointer by default. Since the binary translator makes no such reservation for the frame pointer, it effectively has one extra register. In the case of `fibo`, the extra register avoids

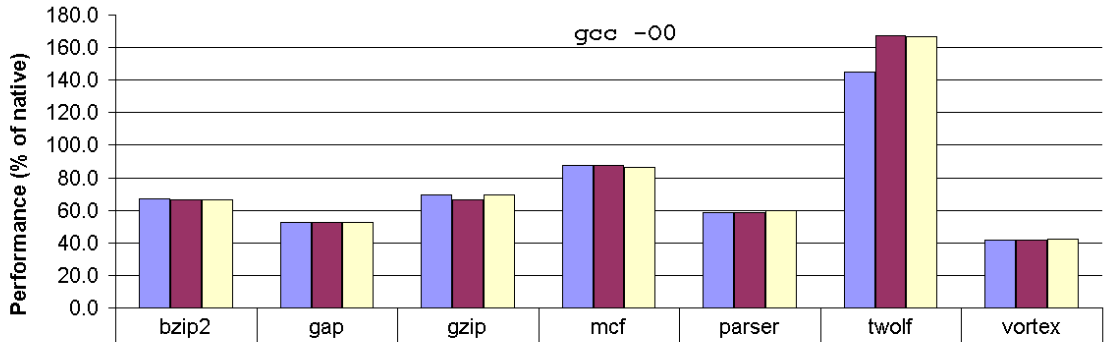
a memory spill present in the natively compiled code causing the huge performance difference. Hence, for a more equal comparison, we also compare with the '`-fomit-frame-pointer`' `gcc` option on x86 (`-O2+` column).

Table 3.10 gives the results for seven of the SPEC integer benchmarks. (The other benchmarks failed to compile correctly due to the lack of complete support for all Linux system calls in our translator). For comparison, we show the performance of two other binary translators – Apple Rosetta[2] and Qemu[28]. In our comparisons with Qemu, we used the same PowerPC and x86 executables as used for our own translator. For comparisons with Rosetta, we could not use the same executables as Rosetta supports only Mac executables while our translator supports only Linux executables. Therefore, to compare, we recompiled the benchmarks on Mac to measure Rosetta performance. To ensure a fair comparison, we used exactly the same compiler version (`gcc` 4.0.1) on the two platforms (Mac and Linux). We ran our Rosetta experiments on a Mac Mini Intel Core 2 Duo 1.83GHz processor, 32KB L1-Icache, 32KB L1-Dcache, 2MB L2-cache and 2GB of memory.

Our peephole translator fails on `vortex` when it is compiled using the `-O2` flag. Similarly, Rosetta fails on `twolf` for both optimization options. These failures are most likely due to bugs in the translators. We could not obtain performance numbers for Rosetta on `gap` because we could not successfully build `gap` on Mac OS X. Our peephole translator achieves a performance of 42-164% of the natively compiled executable. Comparing with Qemu, our translator achieves 1.3-4x improvement in performance. When compared with Apple Rosetta, our translator performs 12% better (average) on the executables compiled with `-O2` flag and 3% better on the executables compiled with `-O0` flag. Our system performs as well or better than Rosetta on almost all our benchmarks, the only exceptions being `-O0` for `vortex` where the peephole translator produces code 1.4% slower than Rosetta, and `-O2` for `vortex`, which the peephole translator fails to translate. The median performance of the translator on these compute-intensive benchmarks is 67% of native code.

A very surprising result is the performance of the `twolf` benchmark where the performance of our translator is significantly better than the performance of natively compiled code. On further investigation, we found that `twolf`, when compiled with

the `-msoft-float` flag, spends a significant fraction of time in the floating point emulation library (which is a part of `glibc`). It turns out that our translator generates faster code for the floating point emulation library than the native compiler leading to better overall performance for `twolf`. We attribute this performance difference in floating point emulation code to the availability of an extra frame pointer register, similar to what we observed in `fibo` microbenchmark. We do not see this effect in all our other benchmarks as they spend an insignificant fraction of time in floating point emulation. We present the detailed characteristics of the benchmarks (such as actual running times and percentage of times spent in floating point emulation) in Appendix A.



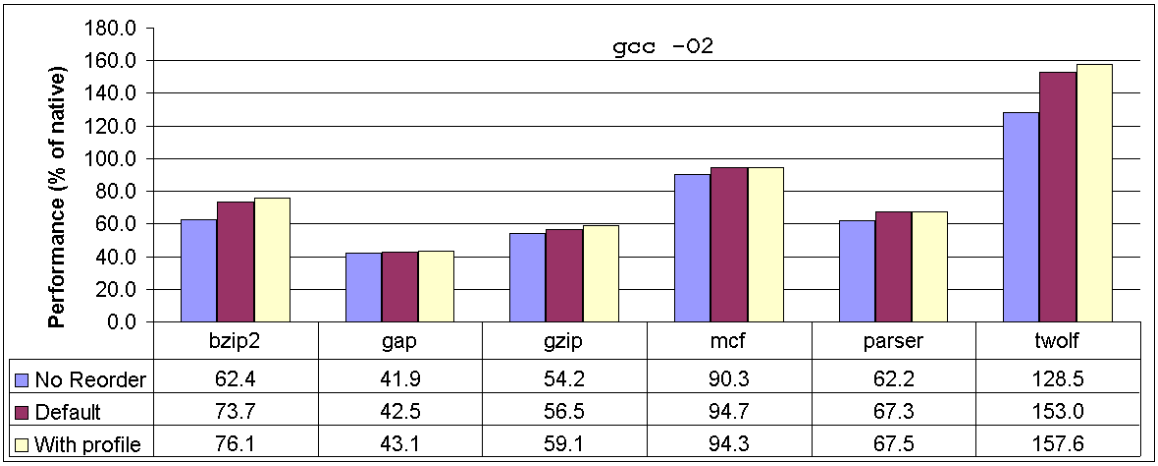
	bzip2	gap	gzip	mcf	parser	twolf	vortex
No Reorder	66.8	52.4	69.8	87.3	58.8	144.8	41.8
Default	66.2	52.5	66.3	87.3	58.7	167.2	41.7
With profile	66.4	52.5	69.3	86.4	60.0	166.8	42.5

Figure 3.9: Performance comparison on `-O0` executables by toggling optimization flags in the peephole translator.

Next, we consider the performance of our translator on SPEC benchmarks by toggling some of the optimizations. The purpose of these experiments is to obtain insight into the performance impact of these optimizations. We consider two variants of our translator:

1. `No-Reorder`: Recall that, by default, we cluster data-dependent instructions inside a basic block for better translation (refer Section 3.4.9). In this variant, we turn off the re-ordering of instructions.

2. `With-Profile`: In this variant, we profile our executables in a separate offline



	bzip2	gap	gzip	mcf	parser	twolf
No Reorder	62.4	41.9	54.2	90.3	62.2	128.5
Default	73.7	42.5	56.5	94.7	67.3	153.0
With profile	76.1	43.1	59.1	94.3	67.5	157.6

Figure 3.10: Performance comparison on `-O2` executables by toggling optimization flags in the peephole translator.

run and record the profiling data. Then, we use this data to determine appropriate weights of predecessors and successors during register map selection (see Section 3.3.2).

Figure 3.9 and Figure 3.10 show the comparisons of the two variants relative to the default configuration for executables compiled using `-O0` and `-O2` respectively. We make two key observations:

- The re-ordering of instructions inside a basic block has a significant performance impact on executables compiled with `-O2`. The PowerPC optimizing compiler separates data-dependent instructions to minimize data stalls. To produce efficient translated code, it helps to "de-optimize" the code by bringing data-dependent instructions back together. On average, the performance gain by re-ordering instructions inside a basic block is 6.9% for `-O2` executables. For `-O0` executables, the performance impact of re-ordering instructions is negligible, except `twolf` where a significant fraction of time is spent in precompiled optimized libraries.

- From our results, we think that profiling information can result in small but notable improvements in performance. In our experiments, the average improvement obtained by using profiling information is 1.4% for `-O2` executables

and 0.56% for `-O0` executables. We believe, our translator can exploit such runtime profiling information in a dynamic binary translation scenario.

Our superoptimizer uses a peephole size of at most 2 PowerPC instructions. The x86 instruction sequence in a peephole rule can be larger and is typically 1-3 instructions long. Each peephole rule is associated with a cost which captures the approximate cycle cost of the x86 instruction sequence.

We compute the peephole table offline only once for every source-destination architecture pair. The computation of the peephole table can take up to a week on a single processor. On the other hand, applying the peephole table to translate an executable is fast (see Section 3.5.1). For these experiments, the peephole table consisted of approximately 750 translation rules. Given more time and resources, it is straightforward to scale the number of peephole rules by running the superoptimizer on longer length sequences. More peephole rules are likely to give better performance results.

The size of the translated executable is roughly 5-6x larger than the source PowerPC executable. Of the total size of the translated executable, roughly 40% is occupied by the translated code, 20% by the code and data sections of the original executable, 25% by the indirect jump lookup table and the remaining 15% by other management code and data.

3.5.1 Translation Time

Translation time is a significant component of the runtime overhead for dynamic binary translators. As our prototype translator is static, we do not account for this overhead in the experiments in Section 3.5. In this section we analyze the time consumed by our translator and how it would fit in a dynamic setting.

Our static translator takes 2-6 minutes to translate an executable with around 100K instructions, which includes the time to disassemble a PowerPC executable, compute register liveness information for each function, perform the actual translation including computing the register map for each program point (see Section 3.3.2), build the indirect jump table and then write the translated executable back to disk. Of

these various phases, computing the translation and register maps accounts for the vast majority of time.

A dynamic translator, on the other hand, typically translates instructions as they (and only when they) are executed. Thus, no time is spent translating instructions that are never executed. Because most applications use only a small portion of their extensive underlying libraries, in practice dynamic translators only translate a small part of the program. Moreover, dynamic translators often trade translation time for code quality, spending more translation time and generating better code for hot code regions.

To understand the execution characteristics of a typical executable, we study our translator's performance on `bzip2` in detail. (Because all of our applications build on the same standard libraries, which form the overwhelming majority of the code, the behavior of the other applications is similar to `bzip2`.) Of the 100K instructions in `bzip2`, only around 8-10K instructions are ever executed in the benchmark runs. Of these, only around 2K instructions (hot regions) account for more than 99.99% of the execution time. Figure 3.11 shows the time spent in translating the hot regions of code using our translator.

We plot the translation time with varying prune sizes; because computing the translation and register maps is the dominant cost, the most effective way for our system to trade code quality for translation speed is by adjusting the prune size (recall Section 3.3.2). We also plot the performance of the translated executable at these prune sizes. At prune size 0, an arbitrary register map is chosen where all PowerPC registers are mapped to memory. At this point, the translation time of the hot regions is very small (less than 0.1 seconds) at the cost of the execution time of the translated executable. At prune size 1 however, the translation time increases to 8 seconds and the performance already reaches 74% of native. At higher prune sizes, the translation overhead increases significantly with only a small improvement in runtime (for `bzip2`, the runtime improvement is 2%). This indicates that even at a small prune size (and hence a low translation time), we obtain good performance.

Finally, we point out that while the translation cost reported in Figure 3.11 accounts for only the translation of hot code regions, we can use a fast and naive

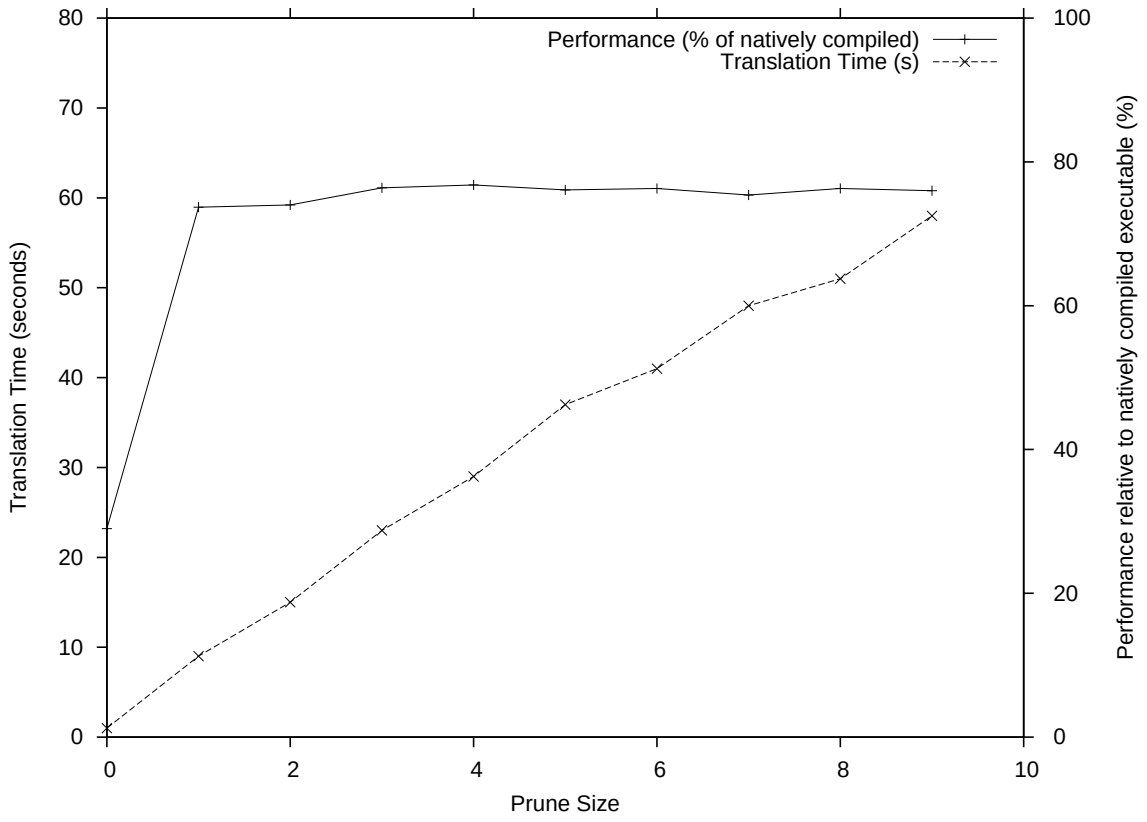


Figure 3.11: Translation time overhead with varying prune size for `bzip2`.

translation for the cold regions. In particular, we can use an arbitrary register map (prune size of 0) for the rarely executed instructions to produce fast translations of the remaining code; for `bzip2` it takes less than 1 second to translate the cold regions using this approach. Thus we estimate that a dynamic translator based on our techniques would require under 10 seconds in total to translate `bzip2`, or less than 4% of the 265 seconds of run-time reported in Table 3.10.

3.6 Related Work

Binary translation first became popular in the late 1980s as a technique to improve the performance of existing emulation tools. Some of the early commercial binary translators were those by Hewlett-Packard to migrate their customers from its HP

3000 line to the new Precision architecture (1987), by Digital Equipment Corporation to migrate users of VAX, MIPS, SPARC and x86 to Alpha (1992), and by Apple Computers to run Motorola 68000 programs on their PowerMAC machines(1994).

By the mid-1990's more binary translators had appeared: IBM's DAISY [13] used hardware support to translate popular architectures to VLIW architectures, Digital's FX!32 ran x86/WinNT applications on Alpha/WinNT [8], Ardi's Executor[14] ran old Macintosh applications on PCs, Sun's Wabi [32] executed Microsoft Windows applications in UNIX environments and Embra [36], a machine simulator, simulated the processors, caches and other memory systems of uniprocessors and cache-coherent multiprocessors using binary translation. A common feature in all these tools is that they were all designed to solve a specific problem and were tightly coupled to the source and/or destination architectures and operating systems. For this reason, no meaningful performance comparisons exist among these tools.

More recently, the moral equivalent of binary translation is used extensively in Java just-in-time (JIT) compilers to translate Java bytecode to the host machine instructions. This approach is seen as an efficient solution to deal with the problem of portability. In fact, some recent architectures especially cater to Java applications as these applications are likely to be their first adopters[3].

An early attempt to build a general purpose binary translator was the UQBT framework[35] that described the design of a machine-adaptable dynamic binary translator. The design of the UQBT framework is shown in Figure 3.12. The translator works by first decoding the machine-specific binary instructions to a higher level RTL-like language (RTL stands for register transfer lists). The RTLs are optimized using a machine-independent optimizer, and finally machine code is generated for the destination architecture from the RTLs. Using this approach, UQBT had up tp a 6x slowdown in their first implementation. A similar approach has been taken by a commercial tool being developed at Transitive Corporation[34]. Transitive first disassembles and decodes the source instructions to an intermediate language, performs optimizations on the intermediate code and finally assembles it back to the destination architecture. In their current offerings, Transitive supports SPARC-x86,

PowerPC-x86, SPARC-x86/64-bit and SPARC-x86/Itanium source-destination architecture pairs.

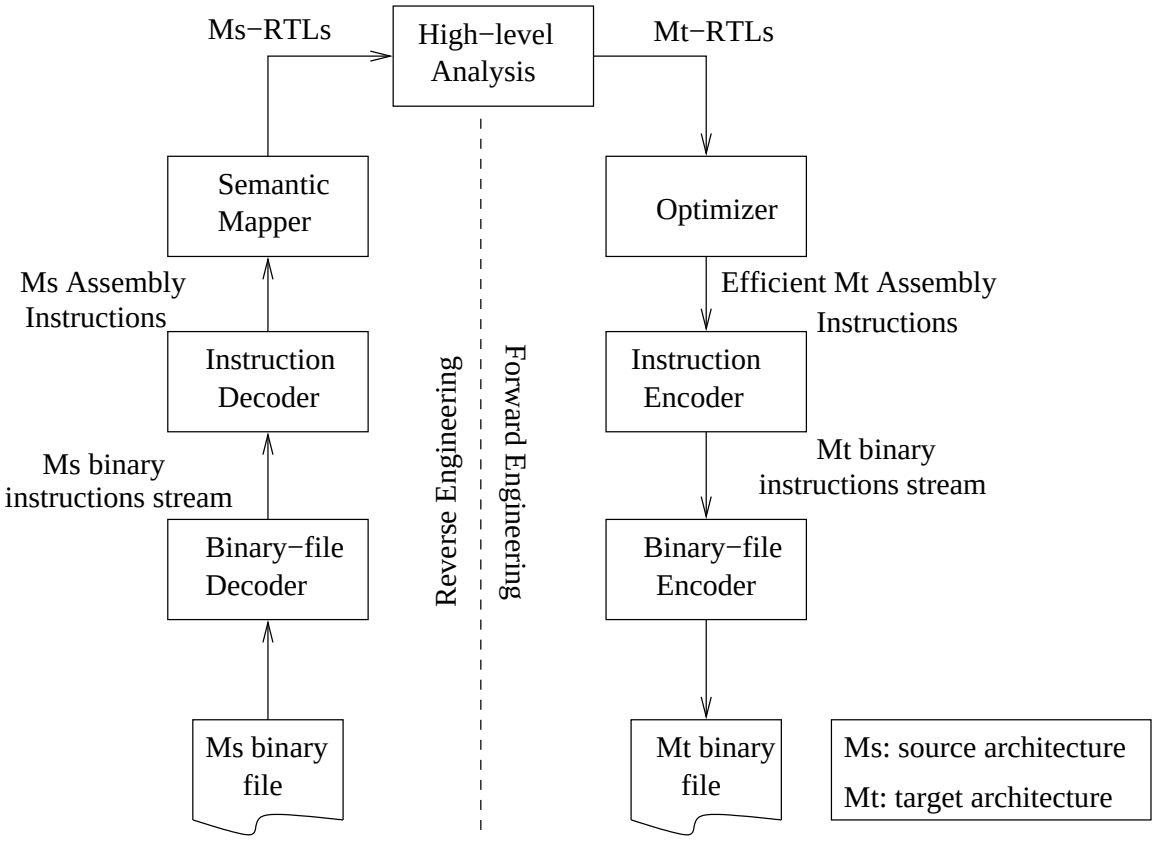


Figure 3.12: The framework used in UQBT binary translation. A similar approach is taken by Transitive Corporation.

A weakness in the approach used by UQBT and Transitive is the reliance on a well-designed intermediate RTL language. A universal RTL language would need to capture the peculiarities of all different machine architectures. Moreover, the optimizer would need to understand these different language features and be able to exploit them. It is a daunting task to first design a good and universal intermediate language and then write an optimizer for it, and we believe using a single intermediate language is hard to scale beyond a few architectures. Our comparisons with Apple Rosetta (Transitive's PowerPC-x86 binary translator) suggest that superoptimization is a viable alternative and likely to be easier to scale to many machine pairs.

In recent years, binary translation has been used in various other settings. Intel's IA-32 framework provides a software layer to allow running 32-bit x86 applications on IA-64 machines without any hardware support. Qemu[28] uses binary translation to emulate multiple source-destination architecture pairs. Qemu avoids dealing with the complexity of different instruction sets by encoding each instruction as a series of operations in C. This allows Qemu to support many source-destination pairs at the cost of performance (typically 5-10x slowdown). Transmeta Crusoe[20] uses on-chip hardware to translate x86 CISC instructions to RISC operations on-the-fly. This allows them to achieve comparable performance to Intel chips at lower power consumption. Dynamo and Dynamo-RIO [4, 7] use dynamic binary translation and optimization to provide security guarantees, perform runtime optimizations and extract program trace information. Strata[30] provides a software dynamic translation infrastructure to implement runtime monitoring and safety checking.

3.7 Conclusions and Summary of Contributions

We present an efficient and portable scheme to perform effective binary translation. We achieve this using a superoptimizer that automatically learns translations from one architecture to another. We demonstrate through experiments that our superoptimization-based approach results in competitive performance while eliminating the complexity of building a high performance translator by hand.

Chapter 4

Goal-Directed Superoptimization Using Meet-in-the-Middle

In this chapter, we discuss a technique to reduce the search space for goal-directed superoptimization. We begin by providing an overview of our approach (Section 4.1), describe the details of the technique (Section 4.2-4.3), present experimental results (Section 4.4) and finally conclude (Section 4.5).

4.1 Introduction

Our superoptimizer exhaustively enumerates and executes all instruction sequences on a fixed initial machine state. An optimization is possible only if one of the objective sequences produces the same machine state as one of the enumerated sequences. A picture showing this approach is shown in Figure 4.1. The runtime complexity of this simple technique is $O(b^n)$ where b is the size of the instruction set and n is the maximum length of the enumerated instruction sequence.

In this chapter, we observe that it is possible to reduce this runtime complexity by pruning the search space. We do this by using information about the goal states we are interested in. Many enumerated instruction sub-sequences can be eliminated as they cannot possibly lead to the goal state. For example, if we are searching for the optimal equivalent sequence for the following sequence

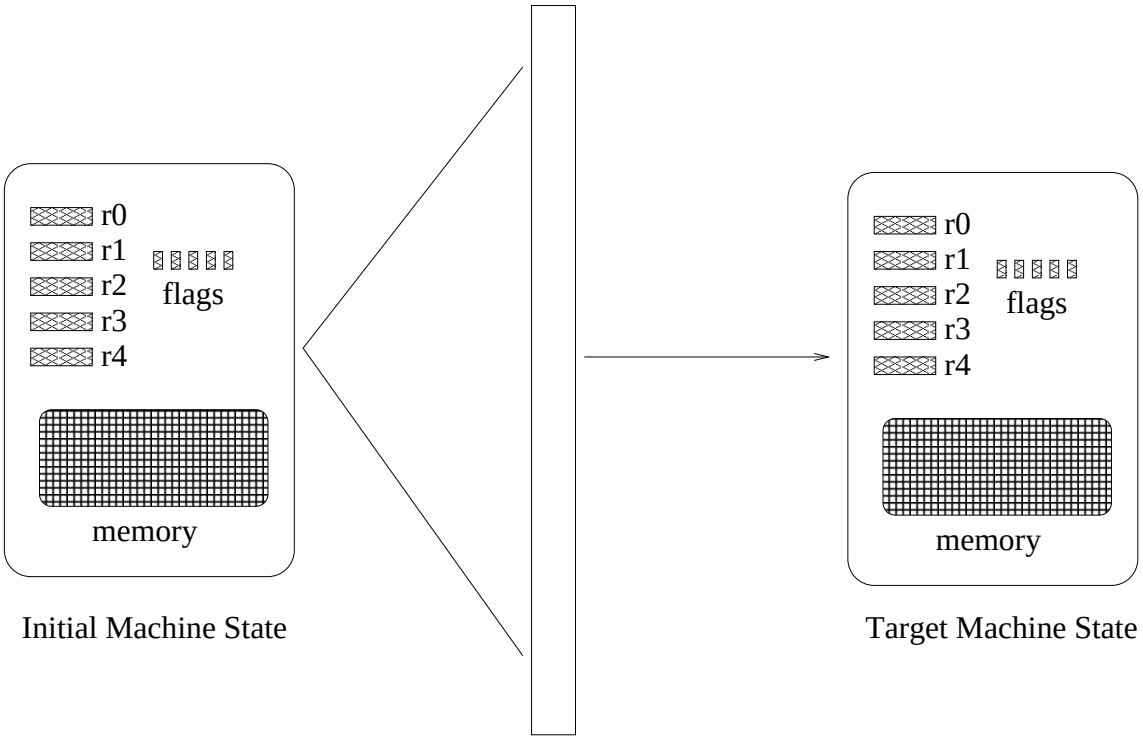


Figure 4.1: Finding a candidate instruction sequence for a target machine state using a brute-force exhaustive search

```
#(swap r1 and r2)
mov r1, r3
mov r2, r1
mov r3, r2
```

any sequence which begins with

```
mov r1, r2
```

is not useful. It would be wasteful to enumerate a length-2 sequence beginning with the `mov r1, r2` instruction. To capture this, we need to determine at an intermediate step, if the current sequence can eventually lead to the goal function. To do so, we work backwards from the goal function to enumerate only those intermediate subsequences that can eventually lead to sequences that are equivalent to the goal function. An intermediate subsequence that does not meet this criteria can be discarded. Using this approach, it is possible to work forwards from the initial state and

backwards from the goal state to prune large portions of the search space. We model our algorithm on this observation and call our approach the meet-in-the-middle strategy. The term meet-in-the middle is borrowed from a cryptographic attack[12] which uses a similar technique to decipher encryption keys. In this section, we describe our meet-in-the-middle approach and it's implementation.

Figure 4.2 illustrates the meet-in-the-middle superoptimization strategy. Our goal is to superoptimize a given instruction sequence which we call the target instruction sequence. We begin with an initial machine state which is a randomly generated vector of 0s and 1s. At the first step, we execute the target instruction sequence on the initial machine state to obtain a machine state which we call the target machine state. Our goal is to find the cheapest instruction sequence that, if executed on the initial machine state, yields the target machine state.

In the next step (Step 2), all possible length-n instruction sequences are exhaustively enumerated starting from the initial machine state and the resulting machine states are converted to bit strings and stored in a 1-bit prefix tree. A prefix tree (also called a trie) is an ordered tree data structure where all the descendants of any one node have a common prefix of the string associated with that node. A trie is generally used for storing dictionaries; in our algorithm, a trie is an efficient data structure to store, retrieve and match machine states.

In the final step (Step 3), instruction sequences of length-m are exhaustively enumerated and executed *backwards* from the target machine state. Recall from Section 4.1 that a backward execution of an instruction sequence involves undoing the operations of the sequence. The machine states obtained by backward execution of length-m sequences are the states that can reach the target state in m instructions. It is possible to reach the target machine state from the initial machine state in $n+m$ instructions only if one of the machine states enumerated in Step 2 matches one of the machine states enumerated in Step 3. We call the machine states enumerated in Step 2, the forward-enumerated machine states because they are enumerated forwards from the initial machine state; similarly we call the states enumerated in Step 3, the backward-enumerated machine states.

Each backward-enumerated machine state is compared against all the forward-enumerated machine states. The trie data structure enables efficient matching at this step. If a match is found, we have found a $n + m$ length sequence potentially equivalent to the target sequence. If no match is found, the backward-enumerated state is simply discarded.

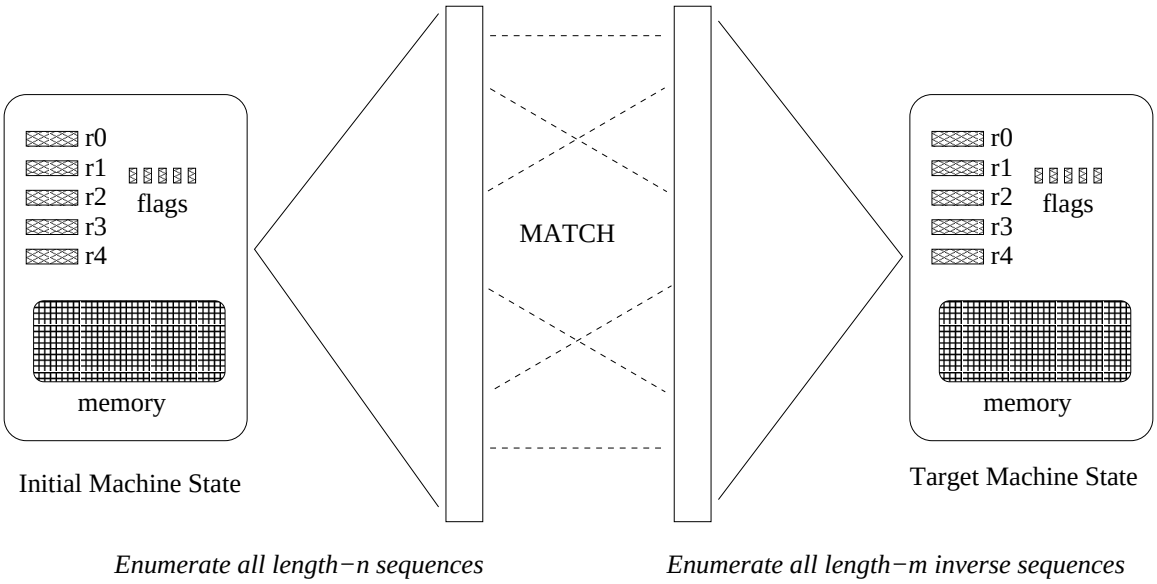


Figure 4.2: Finding a candidate instruction sequence for a target machine state using meet-in-the-middle approach. If done exhaustively, this approach searches the space of all length-$n + m$ sequences.

It is not always possible to precisely undo an instruction, and hence a backward execution may not always produce a unique machine state. We can only reproduce some of the bits by backward execution and for all others, we use the *don't-know* character. We use the character `d` to denote the don't-know character akin to the don't-care bit used in digital hardware design[24].

The inverse of an instruction is an instruction sequence that undoes the operations performed by that instruction. The inverse attempts to reverse the execution of the instruction on a machine state to reproduce the original machine state as closely as possible. The bits that cannot be reproduced are represented using don't-know bits. We discuss instruction inverses and don't-know bits in more detail in Section 4.2.

The matching of forward-enumerated machine states and backward-enumerated machine states is done using a 1-bit trie. The trie stores the prefix bits in its internal nodes and the forward-enumerated machine states at the leaf nodes. For each backward-enumerated machine state, we search the trie for a matching forward-enumerated machine state. We discuss this step of our algorithm in detail in Section 4.3.

We have implemented the meet-in-the-middle strategy in two systems: in our own superoptimizer which we described in Chapter 2 and the publicly available GNU Superoptimizer[15]. We present results obtained in both these systems in Section 4.4 and finally discuss future work in this direction in Section 4.5.

4.2 Instruction Inverses and Don't-Know Bits

The inverse of an instruction I is a sequence of instructions that undo the operations performed by I. In other words, for a given machine state s, I^{-1} is defined such that

$$I^{-1}(I(s)) = s$$

It is not always possible to recover the original machine state s completely using I^{-1}. Instead, I^{-1} recovers as much of state s as possible using don't-know bits for bits that cannot be recovered. We discuss instruction inverses and don't-know bits in more detail in the following subsections.

4.2.1 Inverse of an Instruction

Since it is not always possible to invert an instruction, the inverse aims to recover a tight approximation of the original state (Figure 4.3). In this section, we define an instruction and its inverse more precisely.

Let $\mathcal{M}$ represent the set of machine states. A machine state $s \in \mathcal{M}$ comprises of registers, status flags and memory. An instruction i can be expressed as a function

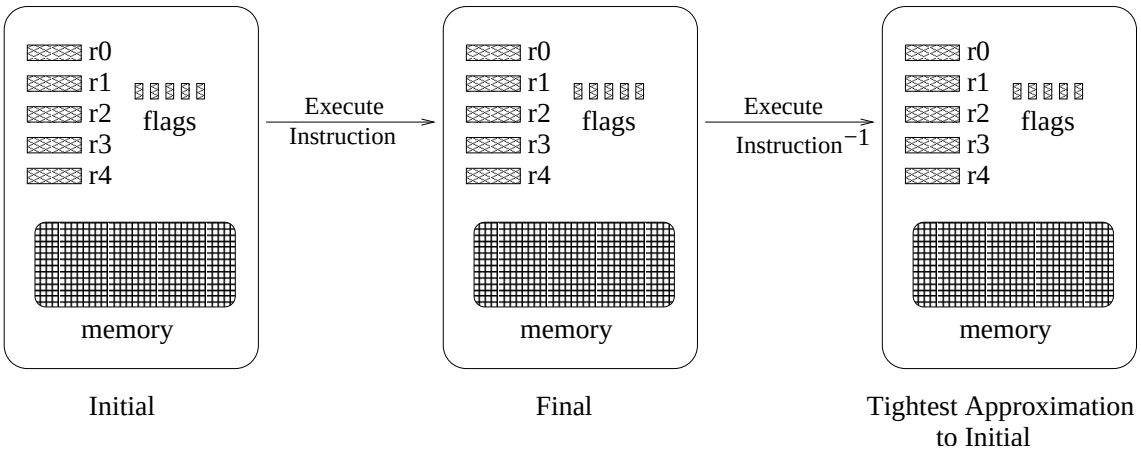


Figure 4.3: The inverse of an instruction.

$i : \mathcal{M} \rightarrow \mathcal{M}$ transforming one machine state (say s_0) to another (say s_1).

$$i(s_0) = s_1$$

An instruction sequence I can then be expressed as a composition of individual instructions. Notice that we consider only straight-line sequences with no branches.

$$I = i_0 \circ i_1 \circ i_2 \ldots i_n$$

In other words,

$$I(s : \mathcal{M}) = i_n(i_{n-1}(i_{n-2} \ldots i_0(s) \ldots))$$

The inverse of an instruction $i^{-1} : \mathcal{M} \rightarrow \mathcal{M}$ is defined as:

$$i^{-1}(s : \mathcal{M}) = \begin{cases} t & \text{if } \exists t \in \mathcal{M}, \text{ s.t. } i(t) = s \\ \texttt{undef} & \text{otherwise} \end{cases}$$

This definition of inverse provides the following useful property:

$$i(i^{-1}(s)) = s \qquad \text{if } i^{-1}(s) \text{ is defined}$$

While every instruction always has an inverse, the inverse may not be unique. Let us look at an example of an instruction and construct its inverse.

Let i_{add} represent the x86 add instruction,

$$i_{add} = \texttt{\{add r0, r1\}}.$$

The semantics of i_{add} on the x86 architecture are:

$$\texttt{r0} \leftarrow \texttt{r0 + r1.} \quad \texttt{flags} \leftarrow \texttt{f(r0,r1)} \qquad .$$

This instruction adds the contents of r1 to r0 and writes the result to register r0. The flags modified by this x86 instruction are the carry, overflow, parity, sign, zero and the auxiliary flag. The flags are a function of the original values of `r0` and `r1`. It is also possible to express the flags as a function of the new value of `r0` and `r1`, such that

$$\texttt{flags} \leftarrow \texttt{f'(r0}_{new}\texttt{,r1)}$$

Functionally, this operation can be expressed as

$$i_{add}(s) = s[\texttt{r0}_{new} \leftarrow \texttt{r0+r1; flags} \leftarrow \texttt{f'(r0}_{new}\texttt{,r1)}],$$

where $s[\ldots]$ denotes s modified by the expression inside the square brackets.

Notice that not every machine state can result from an `add` instruction. A machine state obtained after executing the `add` instruction must have its status flags set in accordance with the contents of register `r0`. In particular, the flags should obey `flags = f'(r0,r1)`.

Constructing the inverse of an `add` instruction first involves checking the register `r0` and flags to see if they agree. If not, this means that this state could not have been a result of executing i_{add}. Hence, the inverse cannot be defined. We use the symbol `undef` to denote an undefined inverse. On the other hand, if the flags agree with `r0` and `r1`, i^{-1} inverts the addition operation by using a subtract instruction.

This can be expressed as

$$i_{add}^{-1}(s) = \begin{cases} i_{sub}(s) & \text{if } \texttt{flags} = \texttt{f'(r0,r1)} \\ \texttt{undef} & \text{otherwise} \end{cases}$$

where,

$$i_{sub} = \{\texttt{sub r0, r1}\}$$

Notice that the inverse of i_{add} is not unique. Some other valid inverses are

```
{sub r0, r1; clc}
{add r0, r2; sub r0, r1; sub r0, r2}
...
```

Notice that the inverse of a single instruction could be a sequence of multiple instructions. In this case, each inverse sequence uses the second instruction to set the status flags differently.

The definition of the inverse of an instruction sequence I is identical to that of the inverse of an instruction:

$$I^{-1}(s : \mathcal{M}) = \begin{cases} t & \text{if } \exists t \in \mathcal{M}, \text{ s.t. } I(t) = s \\ \texttt{undef} & \text{otherwise} \end{cases}$$

For an instruction sequence

$$I = i_0 \circ i_1 \circ \ldots i_n,$$

its inverse is

$$I^{-1} = i_n^{-1} \circ i_{n-1}^{-1} \circ \ldots i_0^{-1}$$

If any of $i_0^{-1} \ldots i_n^{-1}$ produce `undef` on a state s, I^{-1} produces `undef` on s. The bits in the inverted machine state that cannot be recovered are represented using don't-know bits.

4.2.2 Don't-Know Bits

We use the don't-know character '`d`' to represent any bit that is not recoverable after inverse computation. To use our previous example of the add instruction i_{add}, it is not possible to recover the status flags prior the execution of i_{add}. For this reason, we denote the status flags by `d` bits.

Using `d` bits, it is possible to represent the inverse of an instruction uniquely. The inverse of i_{add} can be uniquely represented as:

$$i_{add}^{-1}(s) = \begin{cases} s[\texttt{r0} \leftarrow \texttt{r0} - \texttt{r1}, \texttt{flags} \leftarrow \texttt{d}] & \text{if } \texttt{flags} = \texttt{f(r0)} \\ \texttt{undef} & \text{otherwise} \end{cases}$$

Table 4.1 gives some examples of instruction inverses for the x86 architecture.

Instruction	Inverse	
add r0, r1	sub r0,r1	flags ← `d`
add r0, r0	shr r0; r0[31] ← d	flags ← `d`
mov r0, r1	r0 ← `d`	
and r0, $010110	r0 ← r0 & `d1d11d`	flags ← `d`
or r0, $010110	r0 ← r0 | `0d0dd0`	flags ← `d`
xor r0, r1	xor r0, r1	flags ← `d`
xchg r0, r1	xchg r0, r1	
shr r0, $3	shl r0, $3; r0[0..2] ← `d`	flags ← `d`
ror r0, $3	rol r0, $3	
inc r0	dec r0	flags ← `d`
neg r0	neg r0	flags ← `d`
not r0	not r0	

Table 4.1: Examples of x86 instruction inverses.

4.2.3 Inverse Execution Constraints

Not every machine can be inverted on any instruction. For example, to invert a machine state s using i_{add}^{-1} in the previous example, the flags of s must agree with

register $r0$ and $r1$. We call this constraint on s, the inverse execution constraint for i_{add}. Therefore, to invert a machine state on instruction i, the state must obey the inverse execution constraints of i. Table 4.2 lists some examples of inverse execution constraints for x86 instructions.

Instruction	Inverse Execution Constraint
mov r0, r1	r0 = r1
add r0, r1	flags must agree with r0, r1
and r0, $010110	bits 0,3 and 5 must be 0
or r0, $010110	bits 1,2 and 4 must be 1
shr r0, $3	three MSBs must be zero
xor r0, r1	flags must agree with r0, r1
inc r0	flags must agree with r0
dec r0	flags must agree with r0

Table 4.2: Examples of x86 instruction inverse constraints.

Inverse execution constraints are very helpful in pruning the search space during backward enumeration. If an inverse execution constraint is not satisfied by a machine state, all inverse sequences beginning with that instruction are pruned away.

4.3 Matching forward-enumerated and backward-enumerated states

The final step in the meet-in-the-middle strategy is matching the forward-enumerated machine states with the backwards-enumerated machine states. The forward-enumerated states are stored in a 1-bit trie data structure. The trie represents all machine states that can be obtained by executing a length-n instruction sequence on the initial machine state. Each length-m instruction sequence is then enumerated backwards from the goal state. The backwards-enumerated machine state is then searched in the trie for a match. In this section, we explain the construction and retrieval in a trie in more detail.

To insert a machine state in a trie, the machine state is first converted into a bit-string (Figure 4.4). At each level in the trie, the corresponding bit of the bit-string is

used to determine the appropriate branch to follow. The machine states themselves are stored at the leaf nodes.

Matching of backward-enumerated machine states is very similar to retrieving a machine state from the trie. When searching for a match for a backward-enumerated machine state, the trie is traversed using the bits in the machine state, i.e., a 0-branch (1-branch respectively) is followed if the machine state has a 0-bit (1-bit respectively) at that position. A backward-enumerated machine state may also have `d` bits. During the traversal of a trie, if we encounter a `d`-bit in the machine state, both branches of the trie are followed to search for a match. We illustrate this operation using the following pseudo-code.

```
bool search(trie_node *root, machine_state bwd_enumerated, int bitpos)
{
  if (root == NULL)
    return false;

  if (isLeafNode(root))
    return true;

  if (bwd->state[bitpos] == 0)
    return search(root->left, bwd_enumerated, bitpos+1);

  if (bwd->state[bitpos] == 1)
    return search(root->right, bwd_enumerated, bitpos+1);

  if (bwd->state[bitpos] == 'd')
    return (    search(root->left, bwd_enumerated, bitpos+1)
            ||  search(root->right, bwd_enumerated, bitpos+1));
}
```

If a match is found, we have found a candidate sequence of length $(n+m)$ (formed by

the forward-enumerated n-length sequence stored in the trie and m-length backward-enumerated sequence) that could be equivalent to the target sequence. We then proceed with the equivalence test. If no match is found, the backward-enumerated instruction sequence is discarded.

Assuming the height of the trie is h and the number of forward-enumerated states in the trie are N, the time taken to search for a match for an backwards-enumerated state can vary from $O(h)$ (when there are no `d` bits in the backwards-enumerated state) to $O(N * h)$ (when all bits in the backwards-enumerated state are `d`). Using a compressed trie, where nodes having only one child are coalesced, the average height of the trie can be reduced to $h = O(log_2 N)$.

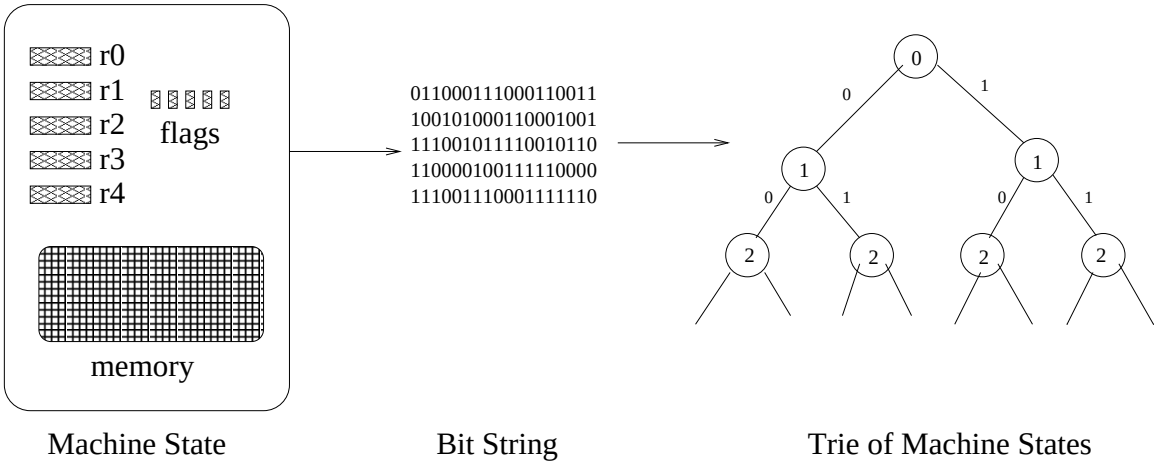


Figure 4.4: All machine states obtained by forward enumeration of instruction sequences are indexed using a trie-like data structure. The trie is then searched to match states obtained by inverse execution from the goal state.

The construction of the trie completes in $O(N * log_2 N)$ time. Since the forward-execution trie needs to be constructed only once, the cost of its construction is amortized over the optimization of several goal sequences. The time complexity of exhaustively searching a space of $n+m$-length instruction sequences using meet-in-the-middle approach is $O(I^m . \log_2 I^m + I^n . \log_2 I^m)$ in the best case, and $O(I^{n+m} log_2 I^m)$ in the worst case. The complexity depends heavily on the number of `d` bits in the backward-enumerated machine states. It also depends on the pruning achieved by inverse execution constraints during inverse execution. We discuss the improvements

and these metrics empirically in our experimental results.

4.4 Experimental Results

We have implemented the meet-in-the-middle strategy in two superoptimization-based systems : our x86 peephole optimizer and the publicly available GNU Superoptimizer[15]. For the x86 peephole optimizer, we specified instruction inverses, don't-know bits and instruction inverse constraints for each x86 instruction. We constructed a trie on the forward-enumerated sequences, and then matched the machine states obtained by backwards execution against it. If a match was found, we performed a complete equivalence test to find an optimization.

Of the 5322 distinct instructions we used in the optimizer's x86 instruction set, we found that 477 were perfectly invertible, i.e. no `d` bits were produced on inverting these instructions. We would like to point out that invertibility depends both on the opcode and the operands. We show the distribution of the don't-know bits produced by the instructions in Figure 4.5. The x-axis represents the number of don't-know bits and the y-axis represents the number of instructions that produced that number of don't-know bits. For example, 1938 instructions produce 6 don't-know bits. The majority of the instructions produced less than 16 don't-know bits. We observed that a lot of instructions produce either 6 or 32 don't-know bits. Instructions producing 6 don't-know bits are usually the ones where it is possible to recover the contents of the operands but not the flags – in which case, we represent the flags by don't-know bits. The instructions producing 32 don't-know bits are usually instructions that clobber one of their operands.

We studied the advantage of using a meet-in-the-middle approach over our previous approach of using only forward enumeration. We use the $n+m$ notation, where n represents the length of the forward enumerated instruction sequences and m represents the length of the backward enumerated instruction sequences. We first enumerated length-3 sequences using $3 = 2 + 1$ (length-2 forwards and length-1 backwards). Using this, we were able to find many optimizations involving length-3 sequences in only a few minutes (as opposed to 2 days in our previous experiments). Next, we

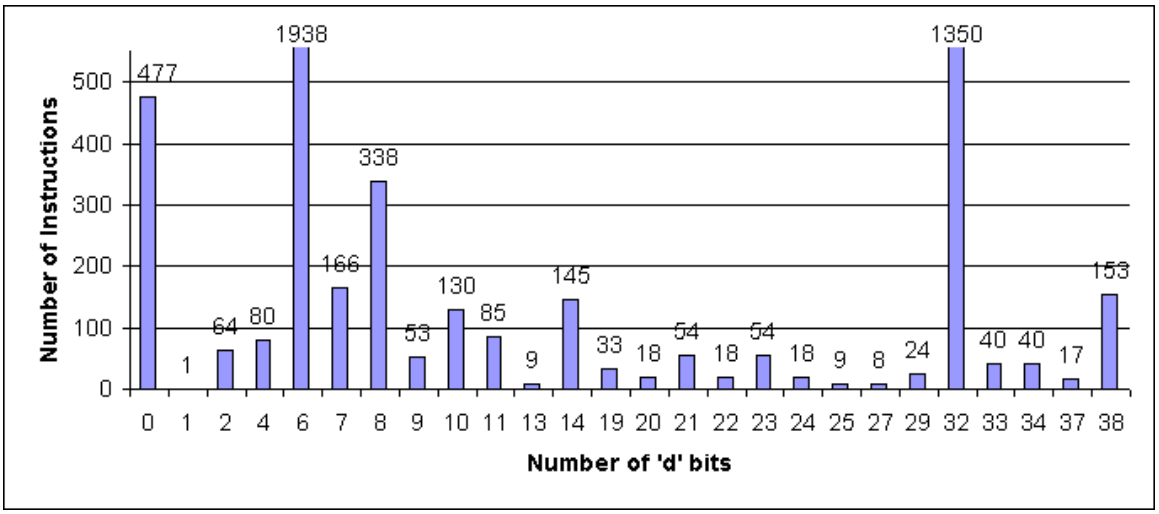


Figure 4.5: The distribution of don't-know bits produced by the instructions in our instruction set.

tried length-4 sequences using $4 = 2 + 2$. In this case, while some of our target sequences were optimized in only a few minutes, some target sequences took over a day of computation without finishing. Through these experiments, we concluded that while the meet-in-the-middle strategy can produce significant improvements in the superoptimizer's running time in some cases (2 days to few minutes), it does not always yield significant improvement. This approach fails whenever the number of `d` bits produced by backwards execution becomes too large.

Next, we discuss the results obtained by implementing the meet-in-the-middle strategy in the GNU Superoptimizer (`GSO`)[15]. We specified the instruction inverses, don't-know bits and inverse instruction checks for all x86 instructions inside `GSO`. With a maximum sequence length of 4, we computed the optimal sequences for the goal functions used in `GSO`. There are 246 unique goal functions specified in `GSO`. For all the goal functions, we obtained the same results for the optimal sequences. Once again, we found that for some goal functions, the improvement in the superoptimizer's running time was significant; while for others, there was no improvement. For 22% of the goal functions, the runtime of the superoptimizer was improved by using the meet-in-the-middle strategy. The maximum improvement was a factor of 10,000x for one of the goal functions while the minimum improvement was 1.62x. The median improvement

in the goal functions that improved was 7x. For 41% of the goal functions, the meet-in-the-middle strategy performed as well as the forwards-only strategy, i.e. we observed neither speedup nor slowdown. This was mostly because the optimal sequence in these cases was only one or two instructions long, and hence both strategies returned a result very quickly. For 37% of the goal functions, we observed slowdowns. In these cases, the overhead of trie construction and matching exceeded the benefit of pruning the search space.

4.5 Conclusions and Future Directions

To be able to scale the superoptimizer to longer lengths, we need more techniques to prune it's search space. Our meet-in-the-middle approach is an attempt in this direction, and we find that it is useful in many cases. One of the important considerations in devising new ways to prune the search space of a superoptimizer is the overhead of the pruning strategy. As we observed in our experiments, the advantage obtained by pruning must significantly outweigh it's overhead for the strategy to be practical.

Chapter 5

Conclusions and Future Work

This dissertation describes methods to perform efficient code generation and optimization by using superoptimization techniques. We describe two peephole superoptimizers – an automatically generated peephole optimizer (Chapter 2) and a binary translator (Chapter 3). In the third part of the thesis (Chapter 4), we present a scheme to lower the computational complexity of goal-directed brute-force superoptimization.

In the first part of the thesis, we describe a design to automatically generate a peephole optimizer using superoptimization techniques. The optimizer first infers thousands of optimizations automatically in an offline phase. The optimizations are organized into a lookup table, mapping original sequences to their optimized counterparts. Optimization of a compiler's generated code is then done efficiently as a normal peephole optimizer, simply using the precomputed rules. The optimizer exhaustively enumerates all length-3 instruction sequences and generates many useful optimizations that would traditionally be implemented algorithmically (Table 2.4). We also find that the automatically generated optimizer is capable of generating efficient SIMD code, at least in some simple situations (Figure 2.6).

The second part of this thesis describes a design to perform efficient binary translation using a peephole superoptimizer. Our translator automatically infers equivalence relations between sequences of the source architecture and the target architecture

(also called peephole translation rules). These equivalence relations are defined under a map of registers between the two architectures. A successful translation can be performed if each source instruction appears in at least one translation rule. Using this approach to binary translation, we find that many optimizations, that would otherwise require manual codification, are automatically inferred. Our PowerPC-x86 binary translator adaptively selects a register map at each program point to maximize efficiency of the generated code. We find that the translated code generated by our binary translator can sometimes outperform natively compiled code on the destination architecture (Table 3.9). Our comparisons with state-of-the-art binary translation tools show the benefits of using this approach (Table 3.10).

Allowing the superoptimizer to scale to longer instruction sequences is likely to produce more optimizations. In the third part of the thesis, we describe a scheme to lower the computational complexity of brute-force search for goal-directed superoptimization. We present the improvements obtained by implementing meet-in-the-middle superoptimization both in our superoptimizer and the publicly available GNU Superoptimizer[15].

In future work, there is potential to further develop peephole superoptimization as a code generation technique. Peephole optimizers have previously been used to perform code selection at link time to produce highly portable compilers[6, 9, 10, 11, 21]. More compute power and advances in SAT solving capabilities present interesting opportunities and challenges in this direction. The most promising opportunity lies in scaling this technique to longer instruction sequences and millions of peephole transformations. One can imagine an architecture quite similar to that of a search engine, where thousands of machines work in the background to infer peephole optimizations, which are presented as an efficient lookup table to be used over the network.

Another interesting application of peephole superoptimizers is binary translation. In our experience, peephole superoptimizers lend themselves as a compelling solution to the problem of efficient and portable binary translation. While we have demonstrated a binary translator from PowerPC to Intel x86, we hope this approach is adopted to perform binary translation across other architecture pairs.

Appendix A

Runtime Characteristics of SPEC Benchmarks

	O0				
Benchmark	x86 Linux	peep Linux	qemu Linux	x86 Mac	rosetta Mac
`bzip2`	310.79	469.85	1727.50	398.77	610.70
`gap`	164.54	313.40	1395.18	-	-
`gzip`	263.94	398.36	1771.11	334.55	570.34
`mcf`	193.18	221.18	402.28	188.25	221.91
`parser`	305.23	520.16	1889.61	379.48	697.15
`twolf`	2184.14	1306.41	3918.79	-	-
`vortex`	193.31	463.48	1766.75	249.71	578.97

Table A.1: Runtimes of the SPEC benchmarks (in seconds) used in Section 3.5 compiled using `-O0` on all the different platforms. The entries in x86 columns are runtimes of executables compiled natively for the x86 platform. The `peep`, `qemu` and `rosetta` columns contain runtimes of executables translated using the peephole translator, Qemu and Apple Rosetta respectively. The Linux runtimes are recorded on an Intel Pentium 4 3.0 GHz processor, 1MB cache and 4GB of memory. The Mac runtimes are recorded on a Mac Mini with Intel Core 2 Duo 1.83GHz processor, 32KB L1-Icache, 32KB L1-Dcache, 2MB L2-cache and 2GB of memory. '-' entries represent failed runs.

	O2				
Benchmark	x86 Linux	peep Linux	qemu Linux	x86 Mac	rosetta Mac
`bzip2`	195.45	265.28	792.54	221.96	410.95
`gap`	87.27	205.47	661.49	-	-
`gzip`	177.81	314.66	816.83	205.28	391.17
`mcf`	174.68	184.44	272.69	154.72	189.95
`parser`	227.72	338.47	1082.26	236.50	482.95
`twolf`	1782.62	1164.88	3011.40	-	-
`vortex`	-	-	-	187.17	492.21

Table A.2: Runtimes of the SPEC benchmarks (in seconds) used in Section 3.5 compiled using `-O2` on all the different platforms. The entries in x86 columns are runtimes of executables compiled natively for the x86 platform. The `peep`, `qemu` and `rosetta` columns contain runtimes of executables translated using the peephole translator, Qemu and Apple Rosetta respectively. The Linux runtimes are recorded on an Intel Pentium 4 3.0 GHz processor, 1MB cache and 4GB of memory. The Mac runtimes are recorded on a Mac Mini with Intel Core 2 Duo 1.83GHz processor, 32KB L1-Icache, 32KB L1-Dcache, 2MB L2-cache and 2GB of memory. '-' entries represent failed runs.

	O0	O2
`bzip2`	0.00	0.00
`gap`	0.00	0.00
`gzip`	0.00	0.00
`mcf`	0.00	0.00
`parser`	0.00	0.00
`twolf`	42.69	51.12
`vortex`	0.00	0.00

Table A.3: Percentage of time spent in floating point emulation by the SPEC benchmarks used in Section 3.5 for `-O0` and `-O2` flags on x86 platforms. Except `twolf`, all other benchmarks spend negligible time in floating point emulation.

Bibliography

[1] Bertrand Anckaert, Frederick Vandeputte, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere. Link-time optimization of IA64 binaries. In *Proceedings of the 10th International Euro-par Conference*, pages 211–220, 2004.

[2] Apple Rosetta. Webpage at `http://www.apple.com/rosetta/`.

[3] Azul Systems. Webpage at `http://www.azulsystems.com/`.

[4] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[5] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 394–403, October 21-25, 2006.

[6] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 329–338, 1988.

[7] Derek Bruening. *Efficient, Transparent and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[8] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. FX!32: A profile-directed binary translator. *IEEE Micro*, 18(2):56–64, Mar/Apr 1998.

[9] Jack W. Davidson and C.W. Fraser. Automatic inference and fast interpretation of peephole optimization rules. *Software - Practice and Experience*, 17:801–812, November 1987.

[10] Jack W. Davidson and David B. Whalley. Quick compilers using peephole optimization. *Software - Practice and Experience*, 19(1):79–97, 1989.

[11] J.W. Davidson and C.W. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(4):505–526, 1984.

[12] W. Diffie and M. E. Hellman. Special feature exhaustive cryptanalysis of the nbs data encryption standard. *Computer*, 10(6):74–84, 1977.

[13] Kemal Ebcioglu and Erik R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *ISCA*, pages 26–37, 1997.

[14] Executor by ARDI. On the web at `http://en.wikipedia.org/wiki/Executor_(software)`.

[15] Torbjörn Granlund and Richard Kenner. Eliminating branches using a superoptimizer and the gnu C compiler. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27, pages 341–352, San Francisco, CA, June 1992.

[16] Tom Halfhill. Transmeta breaks x86 low-power barrier. *Microprocessor Report*, February 2000.

[17] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millenium. *IEEE Computer*, 33(7):28–35, July 2000.

[18] Intel C++ Compiler 9.0. Software available at `http://www.intel.com/software/products/compilers/clin`.

[19] Rajeev Joshi, Greg Nelson, and Keith H. Randall. Denali: A goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, pages 304–314, Berlin, Germany, June 2002.

[20] Alex Klaiber. The technology behind Crusoe processors. Technical report, Transmeta Corp., January 2000.

[21] D.A. Lamb. Construction of a peephole optimizer. *Software - Practice and Experience*, 11:639–647, June 1981.

[22] X. Leroy, D. Doligez, J. Garrigue, and J. Vouillon. The Objective Caml system. Software and documentation available at `http://caml.inria.fr`.

[23] Henry Massalin. Superoptimizer: A look at the smallest program. In *Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 122–126, 1987.

[24] Edward J. McCluskey. *Logic design principles with emphasis on testable semi-custom circuits*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[25] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[26] PowerVM Lx86 for x86 Linux applications. `http://www.ibm.com/developerworks/linux/lx86/index.html`.

[27] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the 2005 IEEE International Symposium on Signal Processing and Information Technology*, pages 7–12, 2005.

[28] Qemu: open source processor emulator. Webpage at `http://fabrice.bellard.free.fr/qemu/`.

[29] QuickTransit for Power-to-X86. http://transitive.com/products/pow_x86.htm.

[30] Kevin Scott and Jack Davidson. Strata: A software dynamic translation infrastructure. Technical Report CS-2001-17, 2001.

[31] Server consolidation and containment with VMware Virtual Infrastructure and Transitive. http://www.transitive.com/pdf/VMwareTransitiveSolutionBrief.pdf.

[32] SunSoft Wabi. http://www.sun.com/sunsoft/Products/PC-Integration-products/.

[33] Superoptimizer prototype. Available on the web at http://cs.stanford.edu/∼sbansal/superoptimizer.html.

[34] Transitive Technologies. Webpage at http://www.transitive.com/.

[35] David Ung and Cristina Cifuentes. Machine-adaptable dynamic binary translation. In *Dynamo*, pages 41–51, 2000.

[36] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Measurement and Modeling of Computer Systems*, pages 68–79, 1996.

[37] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioglu, and Erik R. Altman. LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation. In *IEEE PACT*, pages 128–138, 1999.

[38] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.