Automating the Construction of Compiler Heuristics Using Machine Learning

by

Mark W. Stephenson

Bachelor of Science, Computer Engineering
University of Utah, 1998

Master of Science, Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2000

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2006

Author . . . .
Department of Electrical Engineering and Computer Science
May 23, 2006

Certified by . . . .
Saman Amarasinghe
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Automating the Construction of Compiler Heuristics Using Machine Learning

by

Mark W. Stephenson

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2006, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy of Science in Computer Science and Engineering

Abstract

Compiler writers are expected to create effective and inexpensive solutions to NP-hard problems such as instruction scheduling and register allocation. To make matters worse, separate optimization phases have strong interactions and competing resource constraints. Compiler writers deal with system complexity by dividing the problem into multiple phases and devising approximate heuristics for each phase. However, to achieve satisfactory performance, developers are forced to manually tweak their heuristics with trial-and-error experimentation.

In this dissertation I present *meta optimization*, a methodology for automatically constructing high quality compiler heuristics using machine learning techniques. This thesis describes machine-learned heuristics for three important compiler optimizations: hyperblock formation, register allocation, and loop unrolling. The machine-learned heuristics outperform (by as much as $3x$ in some cases) their state-of-the-art hand-crafted counterparts. By automatically collecting data and systematically analyzing them, my techniques discover subtle interactions that even experienced engineers would likely overlook. In addition to improving performance, my techniques can significantly reduce the human effort involved in compiler design. Machine learning algorithms can design critical portions of compiler heuristics, thereby freeing the human designer to focus on compiler correctness.

The progression of experiments I conduct in this thesis leads to *collaborative compilation*, an approach which enables ordinary users to transparently train compiler heuristics by running their applications as they normally would. The collaborative system automatically adapts itself to the applications in which a community of users is interested.

Thesis Supervisor: Saman Amarasinghe
Title: Associate Professor

Acknowledgments

I want to begin by thanking the people who directly contributed to this thesis. First on the list is my advisor, Saman Amarasinghe. I can't thank him enough for his time, ideas, sound advice, friendship, and yes, financial support during the seven years he advised me. It's pretty scary how quickly Saman can digest, distill, and evaluate ideas; his ability to quickly discard my harebrained ideas came in pretty handy. I also thank Saman for giving me the flexibility to work on the projects that interested me the most. I feel genuinely lucky to have had him as an advisor. Thanks to Una-May O'Reilly, Martin Martin, Ben Wagner, and Diego Puppin for contributing so much to the ideas presented in this dissertation. Thanks also to my thesis committee — Leslie Kaelbling, Una-May O'Reilly, and Martin Rinard — for giving me great feedback on this dissertation. It was a pleasure and a privilege to work with you all. Thanks to all my sources of funding: NSF Graduate Fellowship, NSF grants EIA9810173, EIA-0071841, and CCR- 0073510, and DARPA grants DBT63-96-C-0036, F29601-01-2-0166, PCA-F29601-03-2-0065, F3060200-2-0562, and HPCA/PERCS-W0133890.

Next on the list, I give a collective shout-out to the "Institute"; I really can't think of a better (academic) place to spend eight years. I thank the student body and faculty at MIT for being so damn brilliant and open to anything.

I have had a lot of mentors during my academic career. I want to thank Erik Brunvand, Al Davis, and Mike Parker at the University of Utah; it was truly an honor to work under them. Thanks to Matt Frank, Michael Taylor, Walter Lee, Ben Greenwald, David Wentzlaff, and the rest of the RAW group for showing me the ropes. Jon Babb, MIT just hasn't been the same since you graduated: thanks for your guidance during my first couple of years and for showering me with all of your "Babb" ideas. Thanks to the many fine members of the Assam group, especially Heidi Pan, Mark Hampton, and Jessica Tseng. Anant Agarwal and Krste Asanović deserve special recognition for being such approachable and wise sounding boards. Anant has an amazing blend of intellect, charisma, and suaveness that makes people clamor to be around him. Krste is incredible because, among other things, he's the smartest person in the world. Seriously.

There are two people who I want to thank for caring so much about the welfare of students: Mary McDavitt and Srini Devadas. To properly thank Mary McDavitt would take too long. Mary goes *way* above the call of duty and I hope she knows how much the students love her for that (she baked me a Guinness cake for my defense for heaven's sake). Similarly, Srini takes it upon himself to know the students around him. I love running into him at the water cooler because he always has some bit of advice or funny observation to share.

I am fortunate to have belonged to the commit group, and I thank all of its past and current members for making the lab a pleasant place to be. The group has seen its share of amazing people. Bill Thies, you are a brilliant man, and I thank you for all your insights and help over the years; you will most likely win a Nobel Prize someday. The whole commit group was also lucky to have recruited Rodric Rabbah. Rodric is the first (and usually the last) person I go to whenever I have a question. He is the man. Ben Wagner is a star, and it has been fantastic working alongside him.

I've made the best friends I could *ever* have during my time at MIT. Michael Zhang, you made *80 Allston Street* what it was. Michael "Gordo" Gordon, pub crawling wouldn't have been the same without you. Ronny "Ralph's® Rewards Card" Krashinsky, I truly think you're the man (you have to share this title with Rodric). Steve Gerding, buddy, you are quite possibly the most entertaining person I have ever met. And Sam "[expletive]" Larsen! Thanks for not only being such a great colleague, but for being such a damn fantastic friend over the years. I'm going to miss you guys.

I'd like to thank the people who have *always* been there for me: my family. Even though my parents have *no idea* what I've been working on for the last eight years, they couldn't be prouder, and I'm grateful for that. I'm proud of you too mom and dad. My sister Donna and my brothers Joe and Bob (and their families) are the best, and it saddens me that I don't get to see them more often. And Dominic Primé isn't technically a brother of mine, but it sure seems like he is. I don't know where I'd be without you all.

Finally, über thanks to my fiancée, Kristen Grauman. She's without a doubt the best thing that's ever happened to me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The level of complexity in today's systems is unprecedented. Figure 1-1 shows that playing a simple Java game on a typical user's computer spans several abstraction layers. The game runs within a virtual machine, which is likely embedded within a Web browser; the Java virtual machine runs on top of the Windows ® operating system which executes on a Pentium 4 ®. Even the instruction set architecture of the Pentium 4 is an abstraction: the processor first converts x86 instructions into substantially simpler "micro-operations" on which the rest of the processor operates [45].

The numerous layers of abstraction in the computing hierarchy facilitate the design of sophisticated systems. However, as systems become increasingly layered and modularized, the designers' scope becomes increasingly myopic. Stitching together hundreds of locally designed pieces into a single system creates several formidable challenges, one of which is finding a synergy between all of the individual components.

The compiler is a critical link in the chain of complexity. The efficiency of the code that the compiler generates has an enormous effect on a compiled application's system-level performance. Optimizing compilers are themselves the product of hundreds of thousands of lines of code, and a typical compiler applies over a dozen optimizations to an input program. Furthermore, many important compiler optimization problems are NP-complete, and therefore they necessitate suboptimal heuristics to reasonably constrain compilation time. Because compilers are so complex, most of the optimizations that a compiler performs are

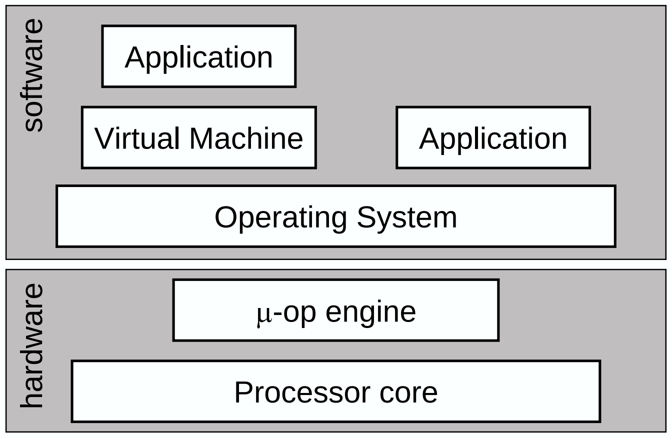


Figure 1-1: Layers of abstraction in a typical system.

not well-coordinated with "downstream" optimizations, let alone the target architectures, which today are comprised of over 100 *million* transistors and feature such innovations as deep pipelines, speculation, hyperthreading, and trace caching.

It is difficult for human engineers to design effective compiler optimizations because of the myriad of complexities in the compiler and target system. Despite these difficulties, optimizing compilers can extract high levels of performance from programs written in high level languages. One study empirically shows that modern compiler technology accounts for large speedups ($2.5x$ to $17.3x$) on a common set of benchmarks [86]. It is because of this potential boost in performance that compiler writers are willing to cope with the complexities wrought by compiler optimizations; in fact, they often spend a large portion of their time tweaking heuristics in order to achieve suitable performance improvements. For instance, the Open Research Compiler [77], on which I validate some of my techniques, features a loop unrolling heuristic that researchers manually updated between every major release (significantly in some cases).

I will show how to *automatically* construct effective compiler heuristics that are specialized for a given computing platform. Specifically, in this dissertation I describe *meta optimization*, an approach that can automatically induce compiler heuristics using well-known machine learning techniques. In most cases, the machine-learned heuristics perform substantially better than their state-of-the-art hand-crafted counterparts. Furthermore, meta optimization reduces the human effort involved with designing an optimizing compiler:

machine learning techniques can automatically design significant portions of a compiler heuristic, thereby freeing the human designer to focus on compiler correctness.

The rest of this chapter discusses the challenges facing compiler writers, examines the inherent complexity of compilers, describes the increasing complexity of architectures, gives an overview of meta optimization, and highlights the contributions of this research.

1.1 Challenges Facing Compiler Writers

This section discusses the many reasons why it is difficult to design optimizing compilers. A compiler, for the most part, creates the applications that run on top of all subsequent layers of abstraction, and therefore, the effectiveness of the compiler is critical to achieving top-notch application performance.

An optimizing compiler is a development tool that has the nontrivial task of converting a high-level description of a program into instructions that can natively execute on a machine. During the conversion, the semantics of the input program are maintained,[1] but the program itself often undergoes dramatic transformations. Figure 1-2(a) shows the flow of a typical optimizing compiler, while part (b) focuses on the *code generation* component, which is responsible for producing quality code from input programs. The blocks in part (b) of the figure are referred to as *passes* (or equivalently *phases*).

Each of the passes in the code generator performs an *optimization*, or in some cases a set of optimizations. An optimization restructures the code, while preserving the code's semantics, in order to attain a higher performance on the target architecture. Optimizing a program for a given computer architecture is inherently a high-dimensional problem. In other words, there are many facets of performance that the compiler must try to balance; there are optimizations to improve register, processor, memory, and communication resource utilization.

Each optimization, which the compiler sequentially applies, focuses on a subset of these dimensions, and it is not uncommon to optimize along a single dimension of performance. For

[1]Sometimes the user of the program can instruct a compiler to sacrifice numerical precision for increased performance, but the vast majority of transformations preserve the semantics of the original program.

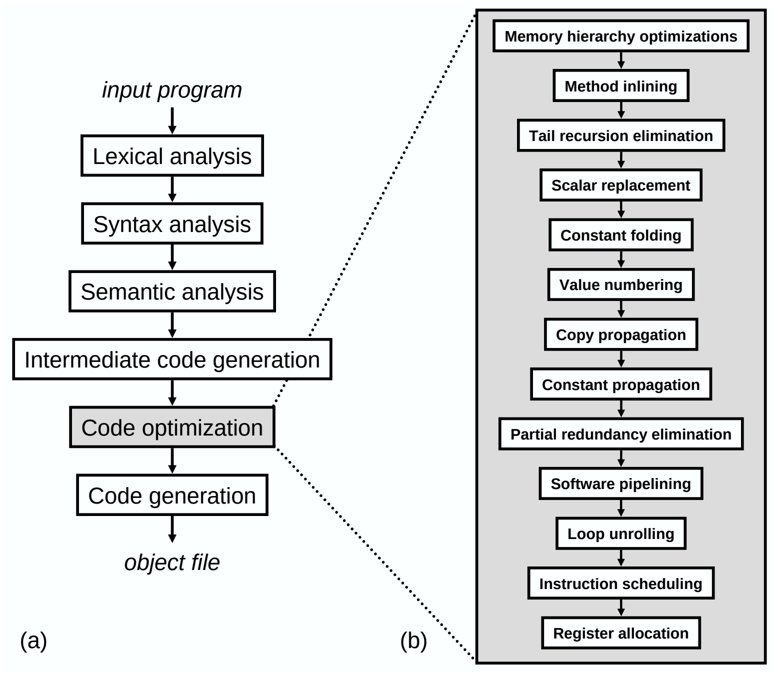


Figure 1-2: The anatomy of an optimizing compiler. Part (a) shows a coarse view of the compilation process, from a high-level program description to an object file. Part (b) zooms in on the flow of the compiler's code optimizer; a typical optimizer applies a dozen optimizations or more to an input program. Many of these optimizations are not well-coordinated with the other optimizations that are performed.

simplicity's sake, compiler writers often assume that compiler optimizations are orthogonal and independent, but this is definitely not the case. For example, in multiprocessor environments, maximizing processor utilization (across all N processors in the multiprocessor) may be completely at odds with minimizing traffic to the architecture's communication system. Likewise, optimizations aimed at reducing inter-processor communication are likely to affect processor utilizations. Another more subtle example is the interaction between the register allocator and the instruction scheduler. Instruction schedulers often tend to increase the lifetime of program variables, which can tax the register allocation optimization; on the other hand, the register allocation optimization constrains the instruction scheduler.

As shown in Figure 1-2, a compiler typically performs a whole host of high-level optimizations first (*e.g.*, method inlining, constant folding, constant propagation) [75]. Many of these optimizations try to simplify the input program. The compiler performs loop optimizations next: optimizations such as loop unrolling and software pipelining aim to expose massive amounts of instruction parallelism. Afterwards the compiler schedules the order

Infrastructure	Source Language	Input Language	Lines of Code
JikesRVM [6]	Java	Java Bytecodes	300,000
ORC [77]	C++	C	1,200,000
Trimaran [94]	C/C++	C	750,000

Table 1.1: Compiler infrastructures considered in this thesis. The source language column describes the language in which the compiler is written. The language that the compiler translates is shown in the third column. The last column lists the number of lines of source code that comprises the compiler infrastructure.

in which instructions should execute. A register allocation phase, which assigns program variables to fast hardware registers, is often one of the last optimizations performed before the object file is generated. A seemingly worthwhile optimization performed early on in the compilation process may hamper the effectiveness of the subsequent passes. Chapter 2 discusses such situations.

The compiler flow shown in Figure 1-2 mirrors the compiler infrastructures that this thesis considers, all of which are listed in Table 1.1. While there are many differences between these three infrastructures, they all have one vital characteristic in common: they are all big systems. The last column in the table lists the number of lines of source code that comprise each infrastructure. The size of the source base is not the best indication of complexity, but these figures do begin to convey the significant amount of work a compiler performs.

Perhaps a more convincing argument about compiler complexity is that several key optimizations are NP-complete. In fact, even the simplest form of instruction scheduling is NP-complete for realistic architecture models [13]. Likewise, optimal register allocation is NP-complete [35] since the problem can be reduced to a graph coloring problem. For such optimizations, it is simply not practical to find the optimal solution. Given enough time, optimal solutions to NP-hard problems can be found, but compile time must be reasonably bounded. Fortunately there are many *heuristics* that approximately solve NP-hard problems.[2]

Because the set of all programs is diverse (and infinite), compiler writers are currently responsible for crafting, and then fine-tuning heuristics that work well across a wide range of input programs; this is a formidable task. A heuristic that is tuned to perform well on one class of applications may not perform well on other types of applications. For instance,

[2]In general this thesis refers to any potentially suboptimal compiler algorithm as a heuristic.

Processor	**Year**	**Transistors**	**Frequency**	**New Features**
Pentium	1993	**3,100,000**	233 MHz	2-way superscalar, 5-stage pipeline, integer SIMD instruction set (MMX).
Pentium II	1997	**7,500,000**	450 MHz	3-way superscalar, 12-stage pipeline, out-of-order execution, larger L1 cache.
Pentium III	1999	**9,500,000**	1 GHz	Floating-point SIMD instructions (SSE), new registers.
Pentium 4	2000	**169,000,000**	3.73 GHz	Hyper-threading, trace cache, double clocked ALU, 20-deep pipeline, improved branch prediction.

Table 1.2: Hardware complexity. This data shows the chronological progression of Intel's Pentium line of microprocessors [44]. Massive increases in the number of transistors from generation to generation have allowed Intel's computer architects to increase the sophistication and complexity of their processors.

scientific applications are more amenable to loop optimizations than control-intensive "integer" programs; aggressive loop optimizations may help scientific applications, but harm the performance of branchy codes.

Increasingly complex computer architectures, the target of optimizing compilers, also make a compiler's task more difficult. In order to generate high quality code, the compiler must have an accurate model of the target architecture. Whereas processors were once easy to model, abstract models can no longer adequately describe today's sophisticated processor designs. Exponentially improving process technology continues to give computer architects additional resources with which to incorporate sophisticated processing mechanisms.

Table 1.2 contains data on Intel's Pentium line of microprocessors [44]. The last column of the table shows some of the key features added to each successive generation of Intel's microprocessors. This indicates that architects have not squandered the sizable silicon real estate increases. For example, in a relatively short amount of time, Intel has added a significant number of architectural features to its Pentium processors. The original Pentium processor, introduced in 1993, was a simple, in-order superscalar processor that was designed on a budget of three million transistors. The most recent Pentium 4 processor at the time of this writing consists of 169 million transistors. The designers used this extra silicon real estate to add several performance-enhancing features (*e.g.*, larger caches, speculative execution, branch prediction, multi-media units, and improved floating point computation units). An aggressive architecture like Intel's Pentium 4 actually performs many (compiler) optimizations at *runtime* (*e.g.*, register renaming and trace construction). The Pentium 4 processor is capable of simultaneously executing up to 126 instructions. Because such processors are not easily modeled, it is difficult to predict what impact a given compiler optimization will have on a processor like the Pentium 4.

Not only is the coupling of compiler technology and the target architectures difficult to model, their relationship changes over time. Consider the re-tuning effort that a compiler writer must undertake for new releases of a processor. At the time of this writing there are no fewer than 16 revisions of the Pentium 4, some of which provide substantially different features than their predecessors [44]. How can compiler writers manually specialize heuristics for the array of architectures even within a single processor family? Because the effort involved in is too drastic, developers typically do not re-tune the compiler for every release.

Nevertheless, it is well known that achieving the best performance requires that the compiler match the target processor. I argue in this thesis that there is a considerable amount of slack in a typical compiler heuristic. In other words, a well-tuned heuristic can produce far more efficient code than an un-tuned heuristic. Chapters 5, 6, and 7 experimentally support my claim.

1.2 Overview of Meta Optimization

This chapter has discussed the complexity inherent to compilers and modern-day architectures. A major implication of systems complexity is that manually tuning a compiler heuristic to work well within a modern day system is not an easy task: for a given system, a designer has to ensure that the heuristic works well across a wide range of programs.

I propose to address this by using machine learning techniques to create heuristics that are catered to the target system. Machine learning techniques, which have the ability to "make sense" of complex, high-dimensional spaces, have the potential to change the way compiler writers develop heuristics. By offloading much of the tedium of heuristic tuning to an automated process, my techniques stand to both improve performance as well as reduce the human effort involved in compiler construction.[3]

[3] I make no claims about reducing compiler complexity. My focus is on *dealing* with compiler complexity.

I call my approach meta optimization because it essentially optimizes compiler optimizations. Meta optimization takes advantage of the fact that many compiler optimizations are naturally separated into two distinct phases: *correctness checking*, and *policy*. The correctness checking phase finds a bounded (but large) set of *legal* optimization possibilities. For many optimizations, finding the set of legal options is straightforward and precise; the more difficult task is the policy phase, which chooses the best option from this set. While the correctness checking component of an optimization ensures that the compiler will produce code that is correct, the policy completely dictates what the attainable performance of the optimization will be.

Meta optimization makes it possible to *automatically* search for effective policies. Consider the global ramifications of a compiler optimization such as loop unrolling, which replicates the body of a loop some number of times to increase parallelism. Loop unrolling interacts with all other compiler passes; in some cases it can enable a host of downstream optimizations, but it can also adversely affect instruction cache performance and the effectiveness of the register allocator. Constructing a heuristic to predict when loop unrolling is appropriate is nontrivial; yet if the heuristic is not well-designed, the performance of the whole system may suffer. Currently human engineers use intuition and manual trial-and-error experimentation to painstakingly construct loop unrolling heuristics.

One instantiation of meta optimization, which I describe in Chapter 6, uses empirical measurements and *supervised learning* to automatically create effective policies. For the loop unrolling problem, the approach first automatically gathers a database of empirical observations by inspecting over 2,500 loops drawn from multiple benchmarks: for each loop, the meta optimizer extracts a vector of characteristics $\vec{x}$ that abstractly describes the loop's structure; it also empirically, and exhaustively finds the best unroll factor y for each loop. With this database in place we can use well-known supervised learning techniques to automatically construct a function $y \leftarrow f(\vec{x})$ that can accurately predict the unroll factor y for a loop described by the vector $\vec{x}$. In this way, the supervised learning algorithm learns to predict the loops for which a given unroll factor best applies.

Meta optimization frees compiler writers of the burden of trial-and-error tuning associated with human-constructed heuristics: once the database of loop examples is in place, it only takes a few seconds to generate an effective compiler heuristic. Furthermore, the meta optimized heuristics generate faster compiled code than their human-designed counterparts. For loop unrolling, meta optimization achieves speedups (over a widely used human-designed heuristic) of up to 26%, and attains an average speedup of 9% on a common set of scientific applications.

On today's complex systems, the only way to know how well a compiler heuristic works is to test it empirically; as such, all of the experiments that I present in this thesis use empirical performance observations to drive the machine learning process.

1.3 Contributions

This thesis investigates the hypothesis that machine learning algorithms can automatically create effective compiler heuristics. I present several novel ideas and findings throughout this document. This following list summarizes the contributions that this thesis makes:

- I have developed a generalized methodology for automatically creating compiler heuristics. The technique is aided by the fact that many compiler heuristics are naturally separated into two components. The first component is immutable and ensures the correctness of the heuristic. The second component however, which I call the policy, is flexible and it dictates the effectiveness of the heuristic. By "learning" the policy — and leaving the correctness component intact — my technique drastically reduces the search space size, making machine learning a feasible, and indeed a palatable approach to heuristic design. In addition, any policy derived by a machine learning technique will generate *correct* code. Chapter 3 discusses this approach.

- I introduce a technique that focuses on small, but high-impact functions called priority functions [91]. By iteratively searching for quality priority functions via policy search, a compiler's effectiveness can, in many cases, be improved by large margins. Chapter 5 discusses this work.

- I show that it is possible to effectively learn compiler policies that make *multi-class* decisions [90]. Recently, other researchers have applied supervised machine learning to compilation problems where possible actions are binary. However, there are many compiler optimizations for which the set of possible actions is larger; these optimizations require a multi-class approach. Furthermore, I evaluate the multi-class experimentation in a real computing environment using real applications, proving the scalability of the technique. Chapter 6 describes this work in detail.

- I show that *feature selection* can automatically identify the most salient features for creating effective compiler policies [90].
- I introduce *collaborative compilation*, a paradigm that enables ordinary users of already-deployed systems to transparently contribute to and benefit from the machine learning data collection process. Collaborative users share their automatically extracted empirical runtime data with other members of a user community. In exchange for sharing, collaborative users are granted access to the community knowledge base, which houses the most current and effective machine-learned heuristics. These users are essentially training the compiler to perform well on the applications and systems in which the community as a whole is interested.

1.4 Roadmap

The next chapter describes the anatomy of an optimizing compiler. I outline my general approach to tuning compiler heuristics with machine learning in Chapter 3. Chapter 4 discusses the machine learning concepts that are necessary for an adequate understanding of the experiments performed in the remainder of the thesis. Chapter 5 shows how policy search can effectively fine-tune small, but high-impact portions of a compiler heuristic. Chapter 6 describes an experiment with multi-class supervised learning. I discuss collaborative compilation in Chapter 7. Finally, I present related work in Chapter 8, and I conclude in Chapter 9.

Chapter 2

Compiler Construction

This chapter presents an overview of modern-day compiler construction. I focus on the details that directly pertain to this thesis, and I purposely omit many aspects of compiler construction that are not particularly relevant to my work. I relegate some important, but not necessarily essentially details to footnotes. Readers who are familiar with compiler construction can safely skip to the next chapter where I outline my approach.

2.1 Compiler Flow

The last chapter diagrammed the high-level flow of an optimizing compiler. This section fills in many of the details that the last chapter omitted. Figure 2-1(a) graphically depicts the operation of a modern compiler. A compiler begins with a high-level description of a program. The compiler first ensures that the input program is well-formed by applying *lexical*, *syntactic*, and *semantic* analyses [75, 4].

From this point, the compiler generates an intermediate representation (IR) on which the remainder of the compiler will operate. The IR merely provides a semantic-preserving representation of the input program that facilitates optimization and analysis. There are typically a couple of different types of IR: high-level IR, which retains many of the high-level constructs associated with a modern day programming language, and low-level IR, which resembles assembly language for an abstract machine. A compiler typically uses high-level

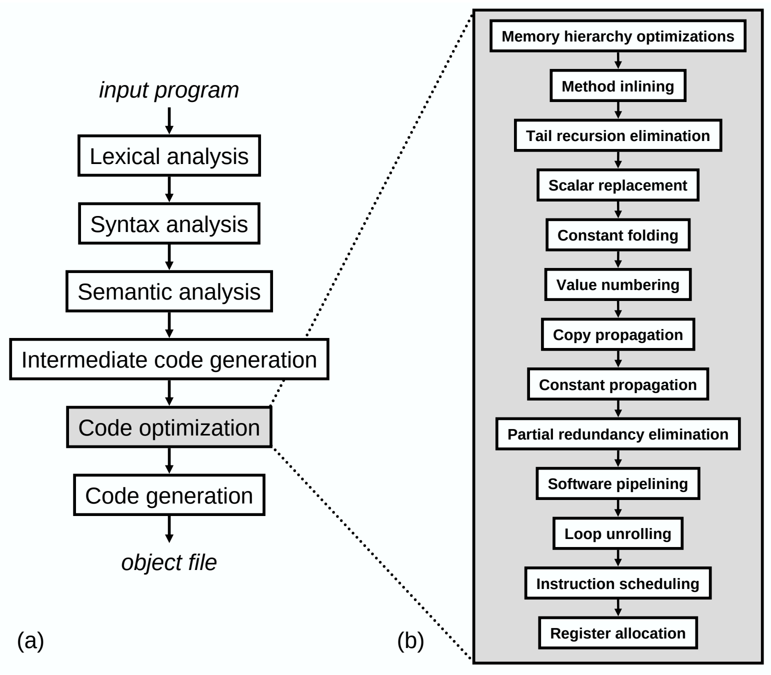


Figure 2-1: Anatomy of an optimizing compiler (repeated from Chapter 1).

IR early in the compilation of a program [75]. For instance, optimizations that aim to improve the performance of the memory hierarchy (*i.e.*, the performance of caches and main memory) benefit from high-level knowledge of memory access patterns, which may not be available in the low-level IR. However, all of the optimizations that I study in this thesis operate on low-level IR.

Figure 2-2(a) shows a simple method written in C. A modern-day processor cannot directly execute the high-level statements in this example because they are too expressive; it is the job of a compiler to disassemble the statements into simple instructions that a processor can execute, and conversion to low-level IR is a big step toward that end. Figure 2-2(b) shows the method's corresponding low IR representation.[1] Notice in particular how the compiler has transformed the high-level control flow (*i.e.*, the for loop, and the if-statement) to low-level *branches*, *labels*, and *conditional tests*. If we were to omit control flow, the low-level IR instructions would execute sequentially, and while that would certainly simplify compiler analyses and code generation, it would severely limit the types of programs that we could write. Labels designate the locations of code. In Figure 2-2(b) for example, the label

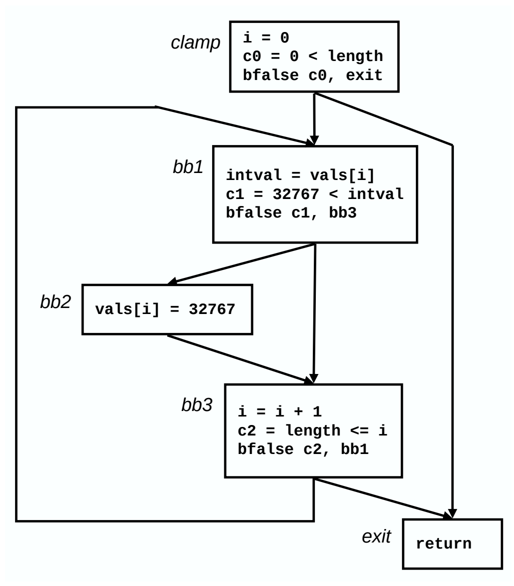
[1]Intermediate representations differ from compiler to compiler, and thus I just mean for this example to give the reader a feel for what an IR looks like.

```
void clamp (int vals[], int length)
{
  int i;
  for (i = 0; i < length; i++)
  {
    int intval = vals[i];
    if ( intval > 32767 )
      vals[i] = 32767;
  }
}
```

(a) Original program.

```
clamp:  i = 0
        c0 = 0 < length
        bfalse c0, exit
bb1:    intval = vals[i]
        c1 = 32767 < intval
        bfalse c1, bb3
bb2:    vals[i] = 32767
bb3:    i = i + 1
        c2 = length <= i
        bfalse c2, bb1
exit:   return
```

(b) Low-level IR.



(c) Control flow graph.

Figure 2-2: An example control flow graph.

`bb3` identifies the location of the instruction that increments the variable `i`. The IR also includes instructions to set a variable's value based on the outcome of a conditional test. For instance, the instruction

```
c0 = 0 < length
```

assigns `c0` a value of true if `length` is greater than zero. The branch instructions in the figure (*i.e.*, the `bfalse` instructions) transfer execution to the specified label if the conditional test is false. If, on the other hand, the conditional test is true, execution simply *falls through* to the next instruction in the sequence.

Branches and jumps introduce ambiguity about which portions of code will execute. For instance, in Figure 2-2(a), the if-statement on line (7) controls whether the statement on line (8) is executed. This uncertainty complicates program analysis, but does not prohibit it. One commonly used structure that facilitates many compiler analyses in the presence of branches is a *Control Flow Graph* (CFG). Figure 2-2(c) shows the CFG for the example in part (a).

The nodes of a CFG are referred to as *basic blocks*, each of which is an ordered sequence of instructions. A basic block can have at most one branch or jump instruction, which if present must occur at the end of the block. All jumps or branches must transfer execution to the beginning of a basic block. As a ramification of these traits, if one instruction in a basic block executes they will all execute. The edges of the CFG denote control dependencies between two basic blocks. For instance, in the entry block labeled "clamp" in Figure 2-2(c) execution can either fall through to "bb1", or it can branch to the "exit" block.

With an IR and a CFG in place, the compiler is armed to apply many classical compiler optimizations. This thesis focuses on the next step of compilation: optimizing the IR to produce a high-quality program. Figure 2-1(b) shows the phases, or equivalently, the *passes* of a typical code optimizer [75]. Code optimizers are essentially a pipeline of optimization passes, and remember, there is little or no coordination between these passes.

Sometimes one pass will facilitate a downstream optimization pass, and other times it

```
double fhelper(double x,
               double omega,
               int select)
{
  switch(select)
  {
  case 0:  return(pow(x+1.0,x));
  case 1:  return(pow(x+1.0,x) * cos(omega*x));
  case 2:  return(pow(x+1.0,x) * sin(omega*x));
  }

  return 0.0;
}

double f()
{
  return fhelper(1.0, 3.14, 0);
}
```

(a) Original program.

```
double f()
{
  double t0 = 1.0;
  double t1 = 3.14;
  int    t2 = 0;

  switch(t2)
  {
  case 0:  return(pow(t0+1.0,t0));
  case 1:  return(pow(t0+1.0,t0) * cos(t1*t0));
  case 2:  return(pow(t0+1.0,t0) * sin(t1*t0));
  }

  return 0.0;
}
```

(b) After inlining.

```
double f()
{
  switch(0)
  {
  case 0:  return(pow(1.0+1.0,1.0));
  case 1:  return(pow(1.0+1.0,1.0) * cos(3.14*1.0));
  case 2:  return(pow(1.0+1.0,1.0) * sin(3.14*1.0));
  }

  return 0.0;
}
```

(c) After constant propagation.

```
double f()
{
  switch(0)
  {
  case 0:  return(pow(2.0,1.0));
  case 1:  return(pow(2.0,1.0) * cos(3.14));
  case 2:  return(pow(2.0,1.0) * sin(3.14));
  }

  return 0.0;
}
```

(d) After constant folding.

```
double f()
{
  return(pow(2.0,1.0));
}
```

(e) After dead code elimination.

Figure 2-3: Optimization examples.

may inhibit the ability of a downstream pass to perform effectively. The *method inlining* optimization nicely illustrates this concept. Method inlining, also referred to as procedure integration, is an optimization that replaces method calls (*i.e.*, calls to other methods) with the actual body of the method [75]. Figure 2-3(a) shows the C description of a method that calls another method. The method `f` calls `fhelper`, and the `switch` statement in `fhelper` steers the computation based on the value of the variable `select`. Figure 2-3(b) shows the result of inlining the `fhelper` method directly into `f`.[2,3]

[2]Note that for clarity I omit the `fhelper` method in Figure 2-3(b) even though a C compiler would have generated code for it (since an external method may call it).

[3]The compiler cannot inline the calls to `cos`, `pow`, and `sin` because they are defined in an external library and the source code may not be available.

In this example, inlining `fhelper` opens several optimization opportunities to the downstream compiler passes. For instance, the constant propagation pass identifies program variables that represent constants, and replaces each of them with the actual constant they represent. In Figure 2-3(b), the constant propagation pass would replace all occurrences of `t0` with the value 1.0 since this is the only value to which `t0` ever refers. Note that constant propagation cannot optimize the code in part (a); only through method inlining does constant propagation become effective for this example. Constant propagation of the code in Figure 2-3(b) results in the code of part (c). Figure 2-3(d) reveals how constant propagation can facilitate the *constant folding* optimization. When all of an arithmetic operation's operands are constants, constant folding can replace the operation with the value that would have been computed at runtime. For instance, the argument to `cos` in Figure 2-3(c) is known at compile time to be `3.14*1.0`, or just `3.14`.

For this example, method inlining combined with constant propagation also enables the *dead code elimination* optimization. Dead code elimination removes code that program execution cannot possibly reach. For instance, it is only possible for execution to ever reach case `0` in Figure 2-3(d), and thus the compiler can eliminate the `switch` statement. Figure 2-3(e) shows the streamlined version that results after the compiler applies dead code elimination.

It is important to note that compiler optimizations are not universally applicable [4, 75, 18]. For instance, method inlining is not *always* a beneficial optimization to perform. Furthermore, predicting the circumstances under which inlining is beneficial is difficult. Method inlining expands the memory footprint of an application — since the optimization essentially creates redundant versions of a method — which may lead to deleterious memory system behavior. For example, the register allocation phase attempts to map program variables to a limited number of fast hardware registers. In the event that the register allocator cannot assign a hardware register to all variables in a method, the allocator has to *spill* a subset of the variables to memory, which is much slower to access. Inlining can burden the register allocator by increasing the density of variables in a method. Chapter 7 studies a method inlining heuristic; in some cases, aggressive inlining leads to enormous slowdowns.

Problem
Boolean simplification [5]
Cluster assignment [29]
Cyclic instruction scheduling [29]
Instruction scheduling with limited resources [35]
Loop fusion with loop interchange [5]
Loop selection [5]
Minimizing waits in presence of control flow [5]
Multi-dimensional scheduling [29]
Profitability determination [5]
Register allocation [75]

Table 2.1: Compiler optimizations that have been proven to be NP-hard.

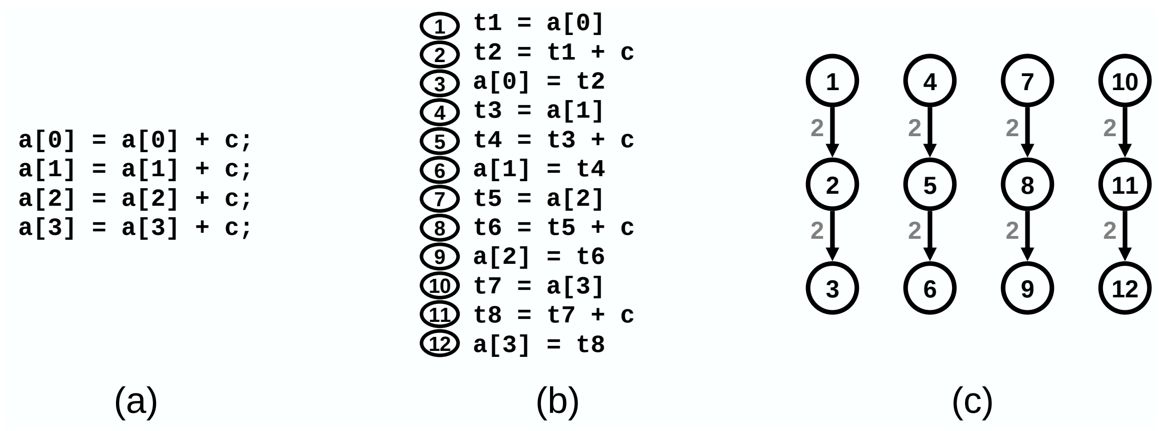
2.2 Important NP-Hard Compiler Problems

Many of the most performance-critical compiler problems are NP-hard [75, 29]. Table 2.1 lists some example compiler problems that have been proven to be NP-hard. Even the most basic scheduling problem is NP-complete when targeting a machine with limited resources. Recent research has shown that integer linear programming (ILP) can effectively solve *some* instances of important NP-hard compiler problems *somewhat* quickly [57]. However, ILP techniques still take orders of magnitude longer to compute than the heuristic techniques compiler writers have crafted to cope with NP-hard problems.

Heuristic techniques strike a balance between compile time and code generation quality that suits most compiler users. The remainder of this section takes an in-depth look at instruction scheduling, a critical NP-complete compiler problem; I also describe a popular heuristic solution that (sub-optimally) solves it. The next chapter will use this problem to illustrate meta optimization.

2.2.1 Case Study: Instruction Scheduling

The instruction scheduling pass is responsible for arranging the order in which instructions execute on a processor, and scheduling order can have a large impact on performance [75, 4, 29]. A simple example illustrates the instruction scheduling problem. Figure 2-4(a) shows four C statements that add a scalar value to a four-element vector. The compiler lowers the high-level description in (a) to low-level IR, which I show in part (b).



```
a[0] = a[0] + c;
a[1] = a[1] + c;
a[2] = a[2] + c;
a[3] = a[3] + c;
```

```
1  t1 = a[0]
2  t2 = t1 + c
3  a[0] = t2
4  t3 = a[1]
5  t4 = t3 + c
6  a[1] = t4
7  t5 = a[2]
8  t6 = t5 + c
9  a[2] = t6
10 t7 = a[3]
11 t8 = t7 + c
12 a[3] = t8
```

Figure 2-4: Instruction scheduling example. The low-level instructions in (b) show how a typical compiler would convert the high-level instructions in (a). Only a small fraction of the 12! possible orderings of the instructions in (b) are legal. The dependence graph in (c) shows the partial ordering for this example that the compiler must satisfy to generate correct code. The labels on the edges in the dependence graph correspond to instruction latencies.

For each high-level statement, the compiler creates three low-level IR instructions. Take the first statement for instance:

```
a[0] = a[0] + c;
```

The compiler first generates an instruction (1) to load the current value of `a[0]` from memory into the temporary variable `t1`. The next compiler-generated instruction (2) adds the variable `c` to `t1` and stores this result in the temporary variable `t2`. To complete the high-level statement, the compiler generates an instruction (3) that stores `t2` to the memory location `a[0]`. The low IR description in part (b) contains a list of twelve instructions. There are $12! = 479,001,600$ ways to schedule the instructions in (b), but only a small fraction of those schedules are actually legal.

It is important for the instruction scheduler to adhere to *dependencies* in a program, otherwise the compiler will likely generate incorrect code. *Data dependencies* impose a partial order on instruction schedules which insures that data correctly flows between instructions in a program. For instance, the read of `t1` in instruction (2) requires that `t1`'s value first be set in instruction (1). Figure 2-4(c) depicts a *dependence graph* — which is straightforward to generate given a list of instructions [75] — that places a partial order on the instructions in part (b). The nodes of the dependence graph correspond to the instructions in part (b), and an edge between two nodes implies a constrained order between two instructions. For instance, the example dependence graph says that instruction (3) is dependent on (2) and (1); in other words, (3) cannot be scheduled before (1) or (2). The constraints that the dependence graph specifies are intuitive: in the running example, adding `c` to `t1` before `t1` contains the value of `a[0]` will almost certainly produce an incorrect result.

We can see from the dependence graph that there are four independent "computations" in the example (*e.g.*, [(1),(2),(3)] is a single computation), and the relative ordering of the instructions within a computation is completely constrained. We can think of this problem in the following way: we have twelve slots, each of which will hold a single instruction from the list in part (b). We can first choose any three of these locations to place one of the completely constrained computations. There will then be nine remaining slots in which to schedule an additional computation. Finally, after scheduling the next computation, there will be six remaining slots in which to schedule an additional computation. Therefore, there are $\binom{12}{3}\binom{9}{3}\binom{6}{3}\binom{3}{3} = 369,600$ legal schedules, which is still a rather large number considering the size of this small example.

While finding the set of legal schedules is straightforward, finding the best-performing subset of schedules can be difficult. For a given target processor (*i.e.*, the processor for which the compiler generates code), some of the legal schedules are much better than others. The numbers next to each edge in the dependence graph in part (c) correspond to the *latency* of the associated dependence. The latency is the amount of time, measured in the number of processor clock cycles, until an instruction's result is ready to pass along to dependent instructions.

For simplicity I have assigned a two cycle latency to all the instructions in the graph. Modern processors are *pipelined*, which, as the name implies, means they can execute multiple instructions simultaneously. A pipelined processor is split up into a number of disparate stages, each of which performs a small part of the overall computation of an instruction [81]. A hypothetical target processor with five stages, for example, can execute up to five instructions simultaneously. However, each of the concurrently executing instructions will be in

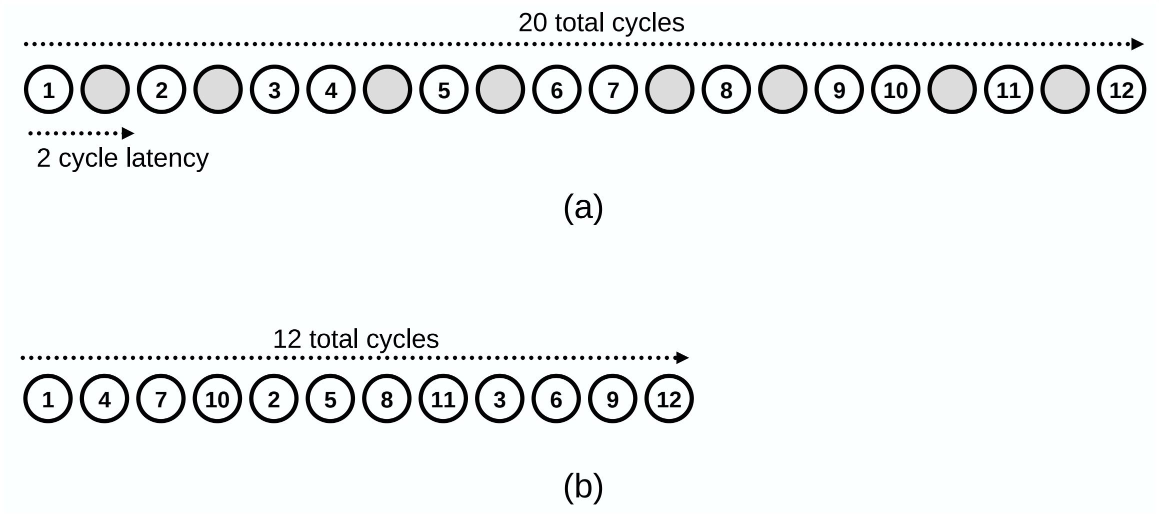


Figure 2-5: The effectiveness of two different instruction schedules.

different phases of the execution. An excellent real-world analogy is a manufacturing assembly line.

Figure 2-5 shows the effectiveness of two different instruction schedules for the code in Figure 2-4(b). The figure assumes execution occurs on an abstract, pipelined processor.[4] The processor will execute the instructions in the order in which they are scheduled,[5] and it will *stall* an instruction's execution until its operands are ready. A stall in this context represents an idle processor cycle.

The instruction schedule shown in Figure 2-5(a) often stalls the processor (stalls are represented as gray circles). For example, instruction (1) takes two cycles to execute, and therefore instruction (2), which is dependent on instruction (1), cannot begin executing until the third clock cycle. The schedule in part (a) takes 20 total cycles to compute the short vector addition. In contrast, the schedule, shown in part (b) only takes twelve total cycles, and it completely eliminates processor stalls. Table 2.2 shows the distribution of the effectiveness of all the valid schedules for the example dependence graph. Even for this simple example, over 80% of the possible, legal schedules are suboptimal.

[4]The figure also assumes that the processor does not include bypassing support.

[5]This assumption is not entirely correct for modern-day architectures which can execute several instructions in parallel and possibly out of order. Nevertheless, instruction scheduling can still drastically affect the performance of such machines.

Cycles	Number of solutions
12	75,488
13	125,096
14	100,504
15	48,608
16	16,360
17	2,720
18	800
19	0
20	24
total	**379,600**

Table 2.2: Distribution of valid schedules for the example dependence graph. Each row shows the number of valid schedules that lead to the associated schedule length. Even for this simple

Garey and Johnson proved that instruction scheduling is NP-complete for resource-constrained processors (which describes any implementable processor) [35]. While NP-completeness makes it infeasible for the compiler to optimally arrange instructions, the next section shows how heuristic solutions can create *good* schedules; this is the task of an instruction scheduler.

2.2.2 Heuristic Solution for Instruction Scheduling

Given enough time, compilers can find optimal solutions to many difficult compilation problems. While some users might be willing to exchange long compilation times for higher-quality code generation, most users prefer a balance between the effectiveness and the efficiency of their compiler. This tradeoff is especially important in just-in-time (JIT) compilation where compilation occurs every time a user runs an application (Chapter 7 describes JIT compilation). Heuristic techniques can often quickly find good approximate solutions to NP-hard problems. This section describes *list scheduling*, which is a heuristic solution for scheduling a list of instructions [4, 75]. List scheduling is a greedy, priority-based algorithm that schedules a sequential list of instructions (*e.g.*, the instructions within a basic block). List scheduling iterates N times, where N is the number of instructions to be scheduled. On each iteration the algorithm schedules one instruction from the set of instructions that can be *legally* scheduled, also called the *ready worklist*. For the example in Figure 2-4, on the first iteration, the ready worklist consists of {(1),(4),(7),(10)} (because these instructions are not dependent on any instructions that have not already been scheduled). Instructions are removed from the worklist when they are scheduled, and new instructions are added to the worklist when all of their dependencies have been satisfied.

Even randomly choosing instructions from the ready worklist will generate a legal schedule. However, generating correct code is only a part of the scheduling problem: a scheduler should also generate efficient instruction schedules. For list scheduling, the manner in which instructions are scheduled from the ready worklist is crucial. On each iteration the algorithm assigns a weight, or *priority* to each instruction in the ready worklist. Once the relative priority of each ready instruction has been computed, the algorithm chooses the instruction with the highest priority and schedules it next. For list scheduling, the function that computes instruction priorities is extremely important for performance, but again note that the priority function cannot affect the correctness of the generated code.

A popular heuristic assigns priorities using latency-weighted depths [37]:

$$P(i) = \begin{cases} latency(i) & : \text{ if } i \text{ is independent.} \\ \max\limits_{\text{j depends on i}} latency(i) + P(j) & : \text{ otherwise.} \end{cases}$$

The priority of instruction i therefore depends on its depth in the dependence graph, taking into account the latency of instructions on all paths to the leaf nodes. For the simple, in-order, pipelined machine described above, the best schedule for the example code in Figure 2-4 is obtained by scheduling instructions with the greatest latency-weighted depths first. This heuristic schedules (1),(4),(7), and (10) first because they all have a dependence of height of 4. After (1),(4),(7), and (10) have been scheduled, (2),(5),(8), and (11) are the instructions with the highest priority values, and therefore, the heuristic schedules these instructions next. Continuing in this fashion, the heuristic creates the (non-uniquely) optimal schedule shown in Figure 2-5(b).

It is important to note however, that the critical path heuristic does not always generate optimal schedules. A scheduling heuristic for modern processors must consider many factors. Production compilers use sophisticated priority functions that account for many competing factors such as instruction occupancy in the pipeline, and how a given schedule may affect downstream compiler passes. As we will see in later chapters, machine learning techniques can effectively juggle many factors that may be important for a given optimization.

We can use machine learning to automatically create priority functions of the sort described here. Priority functions can more generally be described as policies. The next chapter describes, at a high level, how we can apply meta optimization techniques to learn compiler policies.

Chapter 3

Meta Optimization

Abstractly, I automatically create powerful compiler heuristics using machine learning techniques. The purpose of this chapter is to briefly describe the approach that is common to all the experiments performed in this thesis. The reader could benefit greatly by keeping this chapter's content in mind when reading the next chapter, which describes the machine learning techniques that this thesis considers.

There are four main points that I convey in this chapter. First, I focus my efforts on the portions of compiler algorithms that are most amenable to machine learning. Second, my goal is to automatically "learn" critical pieces of the compiler's code generator, which indirectly will improve the code that the compiler generates. Third, I describe how empirical observations drive the machine learning process, and finally, I mention how compiler writers can use my approach to create specialized and general-purpose compiler heuristics alike.

3.1 Insight: Correctness *v.* Policy

A key insight that enables the research that I present in this thesis is that compiler writers tend to separate heuristics into two disparate phases: one that ensures correctness, and one that focuses on performance. I call the phase of a compiler heuristic that concerns itself with performance, the *policy* of the heuristic. The work I present in this thesis focuses on learning effective policies.

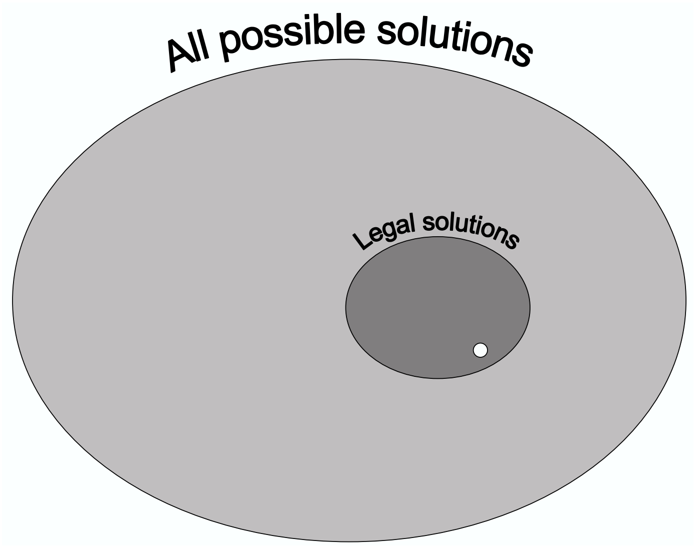


Figure 3-1: An abstract depiction of a heuristic's task. The large, light gray circle represents the space of all possible solutions to a compiler problem. The heuristic first finds a subset of solutions, represented by the dark gray circle, that will generate *legal* code. The heuristic's policy then chooses a single solution, denoted by the small white circle, from this subset.

Figure 3-1 abstractly shows the operation of a compiler heuristic. The light gray circle in the figure represents the space of *all* possible solutions for a compiler problem; this space is potentially *extremely* large. The correctness phase identifies a subset of the solutions — represented by the dark gray circle — that will generate legal code. For most compiler problems, it is relatively easy to identify legal solutions. The goal of the policy is more difficult: choose the *best* (or close to the best) solution from the legal subset of solutions, which I denote by the small white circle in the figure.

As a concrete example, consider the priority-based list scheduling heuristic we discussed at the end of the last chapter. The heuristic first identifies, using a dependence graph, the set of instructions that can legally be scheduled in the next slot. The policy for the heuristic is a priority function which has the task of choosing *a single* instruction from the set to schedule next. The policy for list scheduling completely dictates the effectiveness of the optimization; and as we saw, some policies are much better than others.

The correctness phase is an absolutely critical component of the heuristic, but in most cases it is relatively easy for humans to reason about and design. The constraints that enforce the legality of a transformation are clearly known in most cases, and when they are not, the compiler writer can always make conservative assumptions. The policy, on the other hand, is difficult to design because the factors that may affect performance are not well known, and can be too complex for humans to fully grasp.

Policies have two main qualities that make them ideal targets for machine learning techniques: first, policies have a large affect on attainable performance, and second, *any* policy that a machine learning algorithm constructs will generate *correct* code. The latter quality is extremely important from the user's perspective: users trust, above all else, that the compiler will generate code that correctly represents their input programs.

Users and compiler writers alike benefit from my approach. Using my techniques, compiler writers need only design the correctness enforcing component of a compiler algorithm. Although the policy component is a small portion of a compiler heuristic, from a performance perspective, it is extremely important that the policy is well-tuned. For instance, Chapter 5 describes an experiment where I use machine learning to modify a single priority function in a compiler that is composed of 750,000 lines of C code. On a suite of multimedia and integer applications, the machine-learned priority function attains large speedups ($\sim 1.2x$) over its human-crafted counterpart.

3.2 Meta Optimization : Learning Effective Compiler Heuristics

Broadly speaking, there are two main approaches for applying searching and machine learning techniques to compilation: *objective search* and *meta optimization.* I refer to any approach that embeds a searching algorithm *in* compilation policies as objective search (*e.g.*, [70], [12]). This approach is used to specialize an application by searching for good solutions at *compile time.* In contrast my approach, meta optimization, improves the compiler policies themselves, which will indirectly improve the performance of applications that they optimize. The next two subsections clarify the difference between the two approaches.

3.2.1 Objective Search

Every time a user invokes the compiler, objective search looks for good solutions that are specific to the code that is being compiled. A human-crafted objective function steers the search process. For the list scheduling problem that the last chapter introduced, one could imagine using an objective function to estimate the cost of a given instruction schedule.[1] A search algorithm that is embedded in the list scheduler could iteratively adjust the priorities of the instructions, looking for schedules with a lower cost than the best-so-far schedule. Chapter 8, which covers related work, discusses an approach to list scheduling that does exactly this. Field programmable gate array (FPGA) compilers also use objective search to find good placements of hardware components. These compilers estimate the cost of a given placement of hardware components, and use search techniques to find progressively better layouts.

Note that objective search does not alter compiler heuristics in any way; the heuristics, including the objective functions, are fixed. Instead, objective search iteratively perturbs and tests the code that the compiler generates. While objective search can be quite effective, its success hinges on the quality of the objective function. Remember that today's complex systems are difficult to model. Thus, it is unlikely that a human-constructed objective function can accurately describe the costs and benefits of a given solution.

3.2.2 Meta Optimization

This thesis presents an orthogonal approach, called meta optimization, that searches for effective compiler policies offline. For instance, we can use meta optimization to find effective priority functions. One could even use meta optimization to improve the quality of the objective function used in objective search.

The machine-learned policy, which for now can be thought of as a black box, has the same input and output as a human-generated policy. For instance, consider learning a policy for assigning the priorities of instructions for list scheduling. The policy in this case is a function

[1]One could even create an objective function that simulates the execution of a given instruction schedule and returns the number of cycles needed to run to completion.

that takes as input a vector of characteristics that abstractly describes an instruction, and returns a real number that represents the instruction's priority:

$$p_i \leftarrow f(\vec{x_i})$$

Here $\vec{x_i}$ is a vector of characteristics that describes instruction i. Ideally, these characteristics should be useful for determining relative instruction priorities. For example, the dependence height of instruction i and the number of instructions that depend on it are two potentially discriminating characteristics. The value p_i in this case, is the priority value that the list scheduling algorithm would use to schedule instructions. The goal of meta optimization then reduces to finding a function, f, that generates effective code sequences.

One important difference between objective search and meta optimization pertains to *when* the searching is performed. Objective search is performed every time a user invokes the compiler, and thus it can incur significant compile-time overheads. Meta optimization on the other hand, incurs little or no compile-time overhead. Meta optimization subjects a compiler to a training phase, during which time the meta optimizer finds an effective policy. After the training phase is completed however, the compiler heuristic directly uses that policy to make its compilation decisions. Thus, the compiler policy could be trained "at the factory", and the learned heuristic could be shipped with the compiler.

3.3 Using Empirical Observations

The approach I take to meta optimization is evidence-based. I use actual observed metrics to drive the machine learning process. What then are the metrics by which we can judge the quality of a compiler policy? Because my goal is to create compiler policies that maximize end-to-end performance, the only metric I consider in this thesis is the actual runtime of the code a policy generates.

Alternatively, we could search for policies that generate the shortest, most succinct code sequences. Such an approach might be beneficial for compiling to embedded devices which are memory-limited. However, consider an optimization such as loop unrolling that increases the footprint of the generated code, yet also can yield large performance improvements. We could also consider minimizing cache misses, or the number of loads from memory. Such metrics would perform poorly for the many methods that infrequently touch memory.

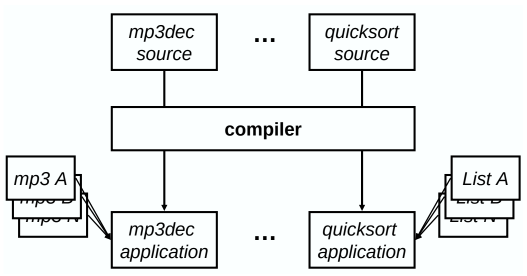
Because compilers and architectures are multi-faceted systems, we should not base optimization decisions on *indirect* performance criteria. The only way to know the *full* impact of a policy is to time its end-to-end performance. Other metrics may correlate with runtime performance, but they are generally inadequate predictors of performance. Of course, meta optimization is a general approach, and depending on the goals of the compiler, one may wish to use other metrics to guide the learning process. The next section abstractly discusses four different methodologies for training compiler policies.

3.4 Training Compiler Policies

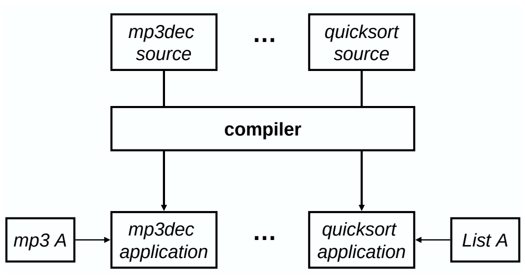
Many people view a compiler as a tool that can effectively compile a diverse set of input programs. The compiler will ideally generate high-quality code for applications ranging from multimedia applications and integer codes to computationally intensive scientific applications. In addition, users expect the compiler-generated executables to perform well on a wide range of program inputs (*e.g.*, users expect their mp3 players to work well for a diverse collection of music).

We can use meta optimization techniques to train a compiler policy to perform well on a broad spectrum of applications. Later chapters will expound upon the notion of a *training set* of data, but for now it will suffice to mention that the machine learning algorithms create heuristics that are catered to this data. In other words, we can include in the training set, data from applications on which we want a learned compiler policy to excel. If the training set includes abundant data from a diverse set of applications, the learned policy will be general-purpose.

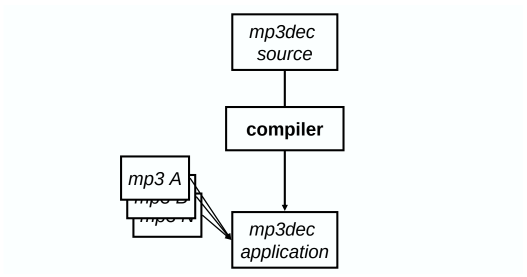
Later chapters use an approach that I call *Multiple Applications Single Input* (MASI) to train general-purpose compiler policies. The MASI approach, as shown in Figure 3-2(b) trains a policy using data drawn from multiple applications, each of which runs a single input. This thesis does not consider the *Multiple Applications Multiple Inputs* (MAMI)



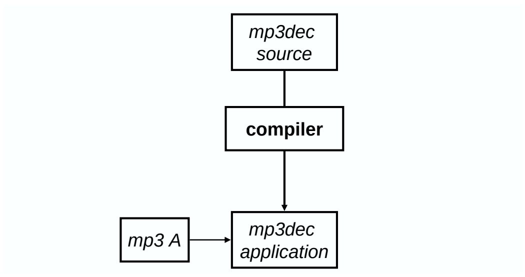
(a) Multiple Applications Multiple Inputs (MAMI).



(b) Multiple Applications Single Input (MASI).



(c) Single Application Multiple Inputs (SAMI).



(d) Single Application Single Input (SASI).

Figure 3-2: The various ways in which to train a compiler policy. The MAMI and MASI approaches shown in (a) and (b) train a general-purpose policy that will perform well across a suite of applications. The SAMI and SASI approaches specialize compiler policies for a single application.

approach shown in part (a), which uses data drawn from multiple applications, each of which runs multiple inputs, to induce policies. The MAMI approach would increase training time, but would likely lead to more robust solutions.

Of course, we can also use meta optimization to specialize the compiler for a *single* application. In this case, the training data set is comprised of data that comes from a single application. This thesis experiments with a *Single Application Single Input* (SASI) training model, which caters the compiler to a particular application that runs a fixed input (*e.g.*, an mp3 player that plays a single mp3). This mode of operation, which is shown in Figure 3-2(d) is beneficial for embedded systems developers who target platforms with tight resource constraints. Though the compiler policy is trained using a single input, I empirically show that the learned policy performs well when other inputs are applied. A *Single Application Multiple Inputs* (SAMI) approach as part (c) shows would likely improve the

Mnemonic	**Description**
MAMI	Meta optimization extracts training data by first instrumenting multiple applications, and running multiple input data sets on the resulting executables.
MASI	Meta optimization extracts training data by first instrumenting multiple applications, and running each executable on a single input data set. This dissertation uses MASI to create general-purpose compiler policies.
SAMI	Meta optimization extracts training data by first instrumenting a single application, and running the resulting executable on multiple input data sets.
SASI	Meta optimization extracts training data by first instrumenting a single application, and running the resulting executable on a single input data set. This dissertation uses SASI to create application-specific compiler policies.

Table 3.1: The various approaches to training compiler policies.

performance of the resulting application on novel inputs, but this thesis does not consider such experiments.

The reader who is familiar with compiler technology may notice similarities between SASI and SAMI approaches and standard feedback-directed optimizations (FDO). FDO is an orthogonal approach that first compiles an application with *instrumentation* that is designed to log details of the application's execution. By running the instrumented executable on a *training* input data set (or sets), FDO *profiles* the application by noting various vital characteristics of the execution (*e.g.*, which methods were hottest, which paths of control flow were most active, etc.). The application is then recompiled and the compiler uses the profile log to specialize code generation. The goals of FDO and meta optimization are much different: FDO collects a large amount of data that fixed compiler policies use to specialize optimization for an application, and meta optimization uses data to create compiler policies. Table 3.1 summarizes the various ways in which we can train compiler policies.

The next chapter introduces the machine learning concepts that are necessary to understand the remainder of the thesis.

Chapter 4

Searching and Pattern Recognition

This chapter introduces the machine learning concepts and techniques that are used throughout this thesis. Later chapters will bolster the ideas presented here. I begin by describing various techniques for searching through large problem spaces. Section 4.2 then introduces *supervised learning*, which seeks to create predictive models given large amounts of data. Finally, this chapter ends with a description of *policy search*, an approach that directly finds effective policies. I use both supervised learning and policy search for meta optimization. The reader who is already familiar with machine learning can safely skip this chapter.

4.1 Searching

The freely available compiler `gcc` has over 25 command-line options that dictate how the compiler should optimize a given input program. The resulting performance of an application critically depends on the settings to these command-line options. One could use *searching* techniques to find effective settings to these options. Searching is the process of looking through a problem space (*e.g.*, the set of all possible optimization command-line settings) with the goal of returning a good solution to the problem (*e.g.*, a good setting to the options for a given input program). The first technique we discuss, exhaustive search, will return the optimal solution to the problem. However, because exhaustive search is often extremely inefficient, researchers have devised techniques that find good solutions with

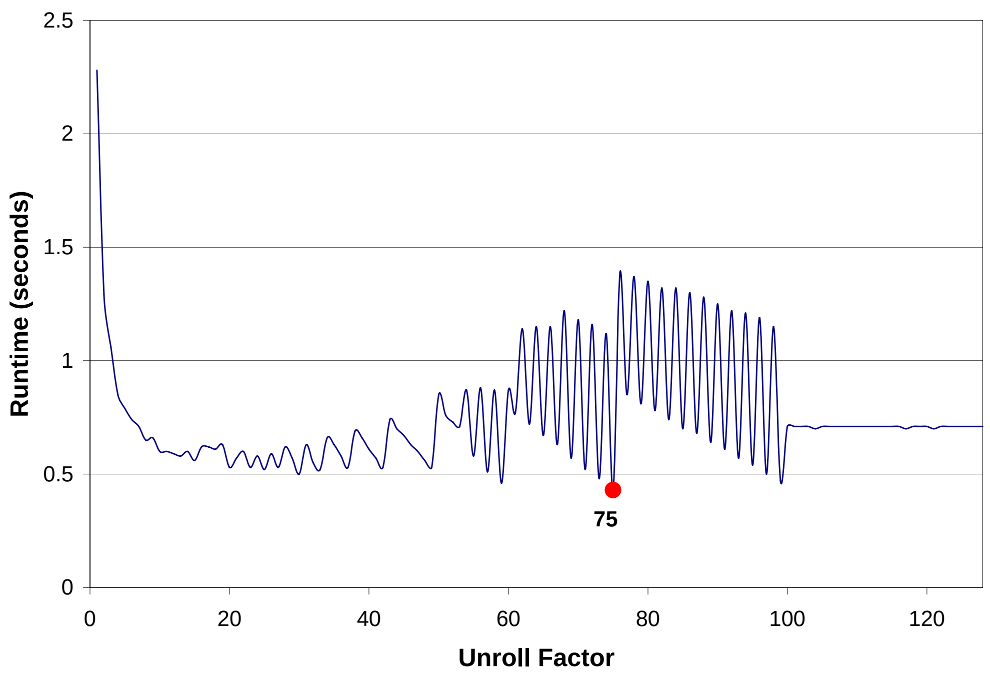


Figure 4-1: Performance of a matrix multiply kernel with respect to the loop unrolling factor. There is a wide variance in performance across the unroll factors from one to 128. Performance plateaus when the loop is fully unrolled. Note the abundance of local minima. This chart was gathered using the Open Research Compiler (version 2.1) at optimization level `-O3` on an Itanium 2 ® processor.

much less computational effort. This section discusses two such techniques, both of which are inspired by processes in the natural world.

4.1.1 Exhaustive Search

As the name, "exhaustive search" suggests, this conceptually simple class of algorithms iteratively, and exhaustively combs the search space. Consider loop unrolling, which is a highly effective and almost universally implemented optimization. Loop unrolling exposes additional instruction level parallelism (ILP) to the compiler by replicating the loop body $N-1$ times, where N is called the *unroll factor*. Chapter 6 discusses loop unrolling in detail, but for now it should suffice to mention that the variance in performance across the range of unroll factors can be quite large for loop-intensive applications.

It is feasible to exhaustively search for the unroll factor that maximizes the performance of a given application.[1] Figure 4-1 graphs the resulting runtime of the SPEC 1992 [89] matrix multiply benchmark for the unroll factors from one to 128. Despite the presence of multiple local minima, exhaustive search easily isolates the best unroll factor for this small benchmark (75). However, when combined with other loop optimizations that interact with the loop unroller — such as *loop tiling*, *array padding*, *software pipelining*, and *data prefetching* — the space of parameter settings quickly explodes. For instance, consider searching over the possible parameter settings for the following trio of optimizations: loop unrolling ($U \in 1...128$), loop tiling ($T \in 1...100$), and array padding ($P \in 1..10$). It would take a considerable amount of time to evaluate all 128,000 combinations.

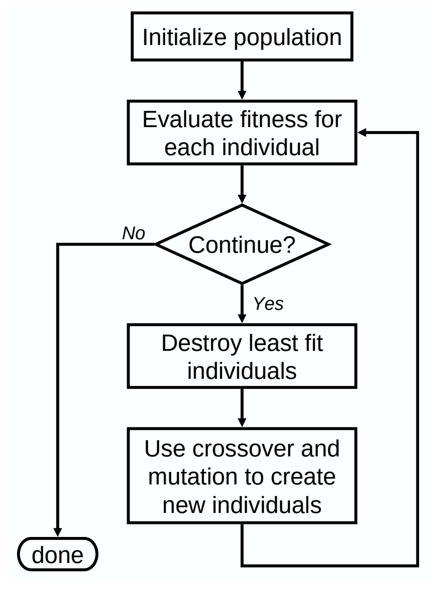
Exhaustive search has the advantage that it does not get trapped in local minima. However, exhaustive search is generally not an option for searching through the vast space of compiler solutions — it is too inefficient.

4.1.2 Genetic Algorithms

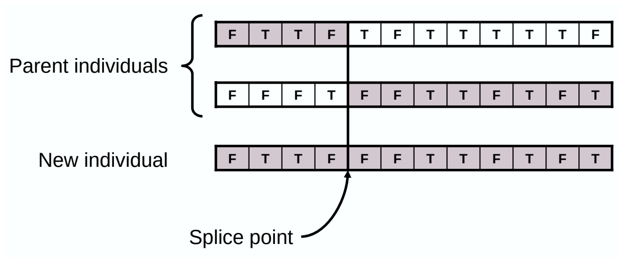
Genetic algorithms (GAs) — which are instances of so-called evolutionary algorithms — are loosely based on Darwinian evolution [43, 8, 33]. GAs seek to improve searching efficiency by maintaining and propagating good local decisions; they are iterative search algorithms, and keeping with the analogy to evolution, each iteration is referred to as a *generation*. The algorithm maintains a *population* of *individuals*. Each individual is represented by a string of bits or characters, which encodes the actions that an individual will perform. The algorithm assigns each individual a score, or *fitness*, that is commensurate with how beneficial the individual's actions are.

In most implementations of GAs, the individuals with the highest fitness scores are most likely to survive and make it to the next generation. In other words, the algorithm weeds unfit individuals from the population at the end of each generation. Akin to sexual reproduction, the GA creates a new individual by splicing together substrings from two — usually highly-fit — individuals. GA proponents maintain that this act, referred to as *crossover*, preserves and combines good local solutions to a problem. The algorithm randomly *mutates*

[1]While unroll factors can technically span the range of nonnegative integers, issues such as instruction caching and register pressure practically limit the range.


Initialize population
Evaluate fitness for each individual
Continue?
No
Yes
Destroy least fit individuals
Use crossover and mutation to create new individuals
done

(a) General flow.



(b) Example representation.

Figure 4-2: Genetic algorithm flow and representation. Part (a) shows the general flow of genetic algorithms, and part (b) shows an example representation for a GA individual. Each of the three individuals shown in (b) is composed of a character string, which can for example represents possible compiler command line settings. The algorithm splices together the representations of two individuals to create a new individual as shown in (b).

individuals in the population — typically with low probability — to help escape from local minima.

Figure 4-2(a) diagrams the general flow of genetic algorithms. GAs begin by creating an initial population of randomly created individuals. The algorithm then determines the fitness level for each individual in the population. The function that evaluates an individual's fitness determines which properties to favor; GAs sort the individuals in a population according to fitness and then they destroy a (usually fixed) percentage of the least fit individuals. To replenish the population for the next generation, a GA will create new individuals by crossing over the representations of randomly chosen pairs of individuals as shown in Figure 4-2(b).

For example, ACOVEA is a fairly popular software package that uses GAs to tune compiler command-line settings to optimize the performance for a given application [59]. For this application, the Boolean characters that comprise the individuals in Figure 4-2(b) determine which optimizations to enable. For instance, the first character of an individual's representation might determine whether to instruct the compiler to unroll loops. ACOVEA, like the experiments I perform in this thesis, is driven by empirical tests; for each individual in the population, the package compiles the input application according to the individual's representation. The package then runs the resulting binaries to determine which individuals are fittest (the faster the binary, the fitter the individual).

4.1.3 Hill Climbing and Simulated Annealing

Although I do not use hill climbing or simulated annealing in any of the experiments performed in this thesis, I describe them because of their widespread usage in compiler resource assignment problems [16]. Unlike genetic algorithms, hill climbing maintains a single solution, S. Here S can be any sort of representation, including a vector of coefficients, or a string of characters as with genetic algorithms [33]. On each iteration, hill climbing perturbs the current best solution, S, to create a modified solution, S'. If S' is a better solution than S — according to the objective function — then S' becomes the current best solution. The algorithm repeats this process until a user-defined stopping criterion is reached.

Figure 4-3 shows the potential pitfalls of hill climbing. The figure shows a randomly created 2-dimensional solution space. The contour shows the cost of being in any particular state, and the goal is to find the state with the lowest overall energy.[2] There are three local minima in the graph. If the hill climbing algorithm starts in the state labeled 'start', then as the black dotted line shows, hill climbing will stop at the local minimum labeled 'HC'. Even worse, if the algorithm were to start in any completely flat region of the contour, it would make no progress.

Beam search is a modification to hill climbing that increases the likelihood of finding the global minimum. Beam search, like a GA, maintains a set of solutions [33]. The search technique then proceeds by iteratively modifying and testing each solution. The most promising solutions are retained, and these comprise the "beam". Another widely used technique to avoid getting trapped in local minima involves *random restarts.* In Figure 4-3, for example, if we were to restart the algorithm in multiple random initial states, the

[2]I use the generic term 'energy' to specify any quantity worth minimizing (or maximizing).

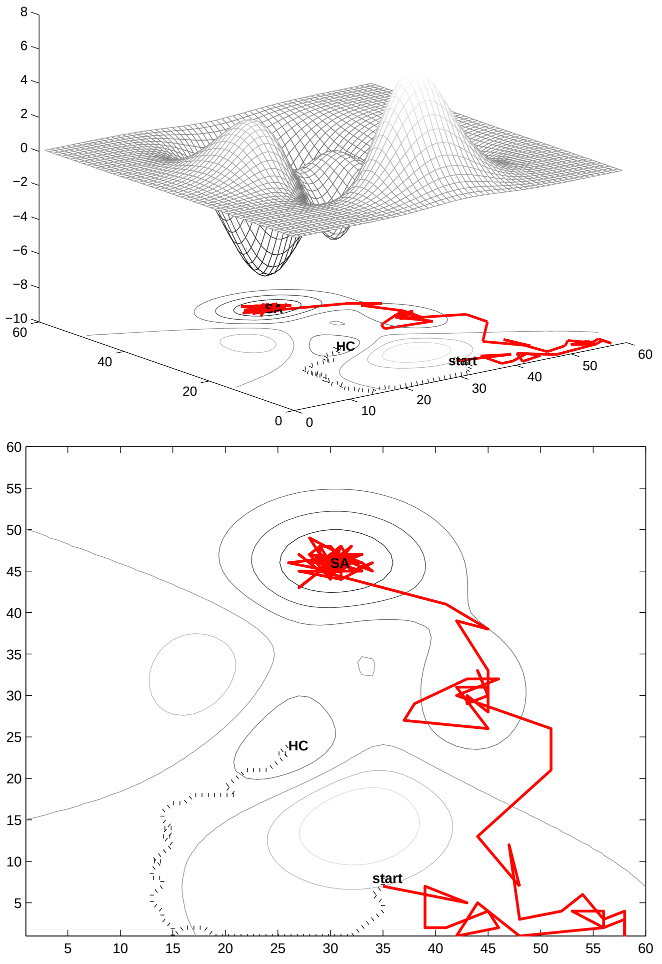


Figure 4-3: A graphical depiction of hill climbing and simulated annealing. The contour plot shows an illustrative search space in which there are three local minima. The black, dotted path, which begins at 'start' and ends at 'HC' corresponds to the exploration of the hill climbing algorithm. The red path, which corresponds to a run of simulated annealing algorithm, also begins at 'start' and explores a local minimum before eventually terminating at the true global minimum. To improve visualization, I omit many intermediate steps of the simulated annealing algorithm.

likelihood of finding the global minimum would increase.

Simulated annealing (SA) is a widely used algorithm that is similar to hill climbing yet not as prone to stopping in local minima [33]. As such it has become an effective tool for solving layout problems [16]. Simulated annealing is a searching technique that is inspired by the metallurgical annealing process. Just as with hill climbing, SA maintains a single current solution S, and furthermore, any perturbed state S' that results in a lower energy configuration will supplant S as the current best solution. However, a simulated annealing algorithm will probabilistically move to states of higher energy than the current best state. More specifically, it will move to a worse state according to the following probability:

$$P_{move} = e^{\frac{-\Delta E}{T}}$$

where ΔE is the difference in energy between the current state and the potential successor state of higher energy, and T — often referred to as the temperature — is a parameter that decreases over time according to a user-defined schedule. At the beginning of the algorithm's execution, high values of T will encourage exploration. The effect of decreasing T over time is that the algorithm will eventually converge to some minimum. Figure 4-3 shows the path of exploration in this search space for SA. The SA exploration path, which follows the red line, also starts at the location 'start'. The algorithm explores a local minimum but eventually escapes and finds the true global minimum (labeled 'SA').

4.2 Supervised Learning

This section transitions from searching to pattern recognition. Supervised learning is an approach to pattern recognition that operates on a database of examples [33, 84]. As the name suggests, supervised learning requires a "supervisor" to correctly construct the database. This section discusses *regression* and *classification*, both of which can be framed as supervised learning techniques. I also describe tradeoffs that a designer must balance between model complexity and generalization performance.

4.2.1 Regression

I introduce supervised learning by describing regression. Not only is regression probably more familiar to the reader, but it is also a form of supervised learning. Regression attempts to fit a function to a series of data. Assume we are presented with the following data:[3]

[3]Matlab generated the following data according to the function, $y = 1 + 2.5x^2 + \varepsilon$, where ε represents a Gaussian noise component of $0.25N(0, 1)$.

x	-1.0	-0.8	-0.6	-0.4	-0.2	0.0	0.2	0.4	0.6	0.8	1.0
y	3.66	2.16	2.07	1.60	1.26	1.33	1.18	1.23	1.86	1.99	3.62

Adhering to supervised learning nomenclature, we will refer to this data set as our *training set*, and we will call each of the eleven points that comprise the training set, *training examples*. Each training example $\langle x_i, y_i \rangle$ is composed of a *feature* x_i and a corresponding *label* y_i.

Our goal in this case is simple: find the function that minimizes the sum of squared error for the examples in our training set. As mentioned above, we know that the data for this example is roughly quadratic, so we will search for a solution to $y = b+mx^2$. Linear algebra provides an elegant solution for approximating this simple function:

$$\begin{bmatrix} b \\ m \end{bmatrix} = (A^T A)^{-1} A^T y, \qquad \text{where } A = \begin{bmatrix} 1 & -1.0^2 \\ 1 & -0.8^2 \\ 1 & -0.6^2 \\ 1 & -0.4^2 \\ 1 & -0.2^2 \\ 1 & 0.0^2 \\ 1 & 0.2^2 \\ 1 & 0.4^2 \\ 1 & 0.6^2 \\ 1 & 0.8^2 \\ 1 & 1.0^2 \end{bmatrix}, \text{ and } y = \begin{bmatrix} 3.66 \\ 2.16 \\ 2.07 \\ 1.60 \\ 1.26 \\ 1.33 \\ 1.18 \\ 1.23 \\ 1.86 \\ 1.99 \\ 3.62 \end{bmatrix}$$

The calculation yields $b = 1.09$, and $m = 2.27$. Using these parameters we attain an error of:

$$\left\| A \begin{bmatrix} 1.09 \\ 2.27 \end{bmatrix} - y \right\|^2 = 0.88.$$

These coefficients b and m define our model. For instance, if we were presented with a new data point x_q, we could predict y_q using the equation $y_q = 1.09 + 2.27{x_q}^2$. Figure 4-4 plots the training data alongside the values predicted by this model.

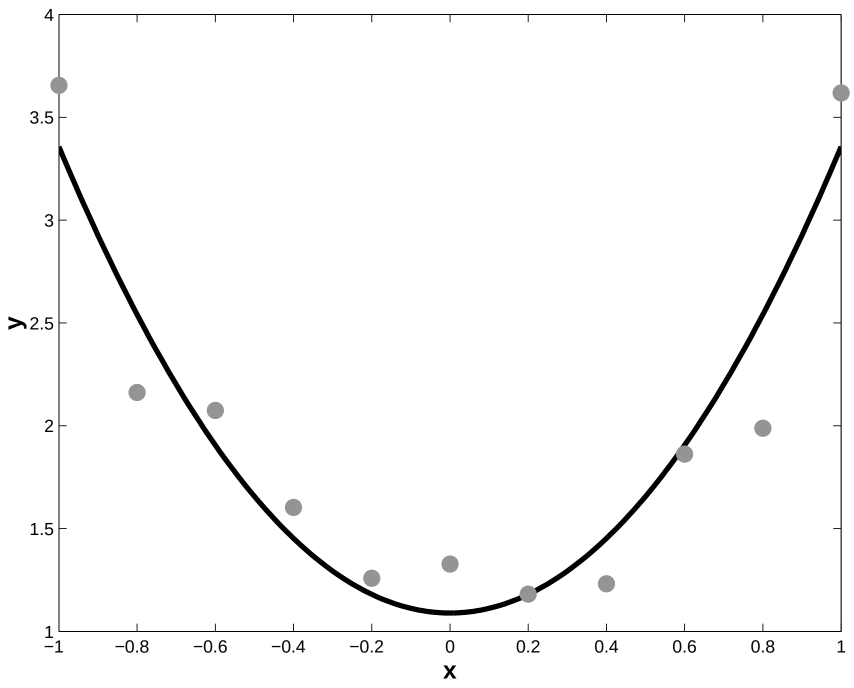


Figure 4-4: Regression example. The gray circles correspond to the training set of examples, and the solid line represents the values predicted by the learned model.

For the above example, we sought a function, $y_i \leftarrow f(x_i)$, where x_i and y_i are real-valued scalars. It is not difficult to extend the function so that it considers a vector of inputs, $y_i \leftarrow f(\vec{x_i})$. The machine learning community refers to the inputs, $\vec{x}$, as a *feature vector*. We can create arbitrarily complex feature vectors to construct more sophisticated models. For example, we can create a 2nd-order polynomial model by creating a feature vector of the form $\vec{z_i} = \left[\begin{array}{ccc} 1 & x_i & x_i^2 \end{array}\right]$. The matrix A then becomes:

$$A = \left[\begin{array}{c} \vec{z_0} \\ \vec{z_1} \\ \vdots \\ \vec{z_N} \end{array}\right],$$

where N is the number of examples in the training set. Again, $(A^T A)^{-1} A^T y$ provides the best least-squares linear fit for the examples in the training set. Notice that the feature vector can contain elements that are non-linear in the original feature space. In general, it is possible to arbitrarily transform the original feature space into a higher-dimensional

Model	**Training Error**	**Validation Error**
$y = b + mx$	2.79	2.51
$y = b + mx^2$	0.88	0.50
$y = b + mx^2 + nx$	0.82	0.65
$y = b + mx^9 + nx^8 + \ldots + vx^2 + wx$	0.04	1.50

Table 4.1: Choosing an appropriate model.

feature space, $\left[\begin{array}{cccc} \phi(x_i) & \varphi(x_i) & \ldots & \psi(x_i) \end{array}\right]$, where ϕ, φ, and ψ are called *basis functions.* For instance, in some cases it may be sensible to use a *log* transform to squash the inputs.

In this case we knew to learn a quadratic model (because I generated the data according to a quadratic model). In general however, one does not know the distribution from which the training data comes. Choosing an appropriate model is a balancing act. Table 4.1 shows the ramifications of increasing model complexity. The models in the table range from a line to a 9th-order polynomial. Notice how the error on our training set decreases as we increase model complexity. Intuitively this makes sense: the more complex models have more flexibility to fit the data in the training set. However, the use of unnecessarily complex models is not advised. To explain why, let me introduce the concept of a *validation set.* A validation set is a collection of examples that was not used to train the model; testing the performance of the model on the validation set allows us to estimate its general predictive accuracy. The following data is drawn from the same distribution as the examples in the training set:

x	-0.9	-0.7	-0.5	-0.3	-0.1	0.1	0.3	0.5	0.7	0.9
y	3.05	2.08	1.46	0.95	1.01	1.12	1.14	1.50	2.22	2.98

The third column of table 4.1 shows the sum of squared error for each of the models on this set of data. Notice how the quadratic model has the lowest error on the validation set. The linear model is too simple to express the desired function well, and the higher-order polynomial models are too flexible and *overfit* the data. Figure 4-5 graphically depicts overfitting for the example. The gray circles in the figure are the training examples. The $9th$-order polynomial latches on to these data, leading to a low error on the training set. The added complexity allows the model to fit the added noise component, which hurts the model's accuracy on the validation set (represented by the red squares). In contrast, the

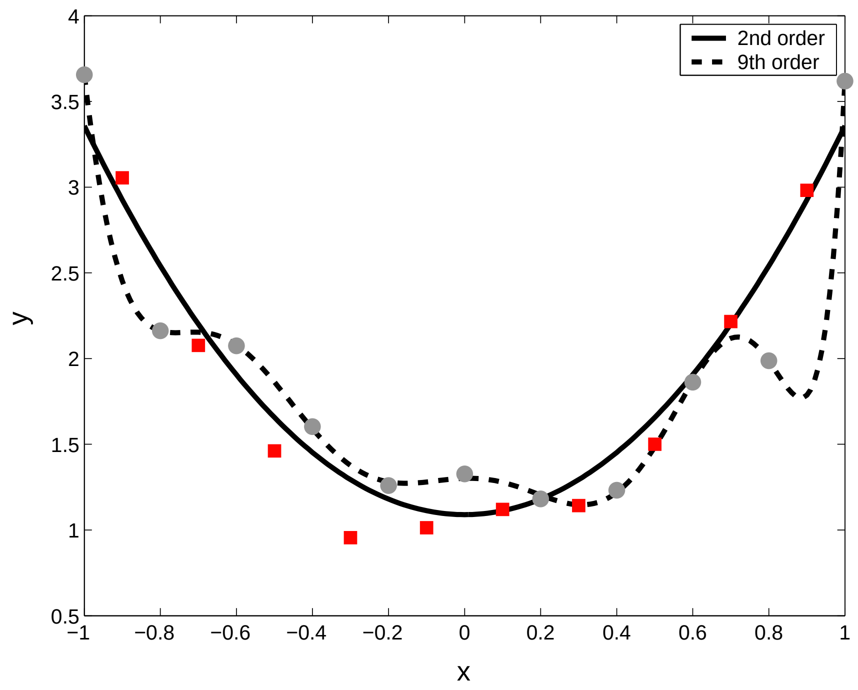


Figure 4-5: Model complexity and overfitting. The gray circles represent the training examples, and the red squares represent the validation set of examples. The 9th-order polynomial overfits the training data, leading to a poor validation error.

quadratic model performs slightly better on the validation set than it does on the training set. *Validation*, which I formalize in Section 4.2.3 refers to the use of a validation set to gauge the generality of a learned model.

4.2.2 Classification

Supervised classification is similar to regression. Classification also uses a set of training examples. Again, each training example $\langle \vec{x}_i, y_i \rangle$ is composed of a feature vector $\vec{x}_i$ and a corresponding label y_i. For classification however, $y_i \in 0, 1, \ldots, M$. In other words, each example is categorized into one of M classes. This section describes *binary classification*, where $y_i \in \{0, 1\}$.

The feature vector contains measurable characteristics of the object under consideration. The label indicates how the object should be classified. Training a classifier often involves finding a function, $y \leftarrow f(\vec{x})$, that maps from feature vectors to labels so that the overall

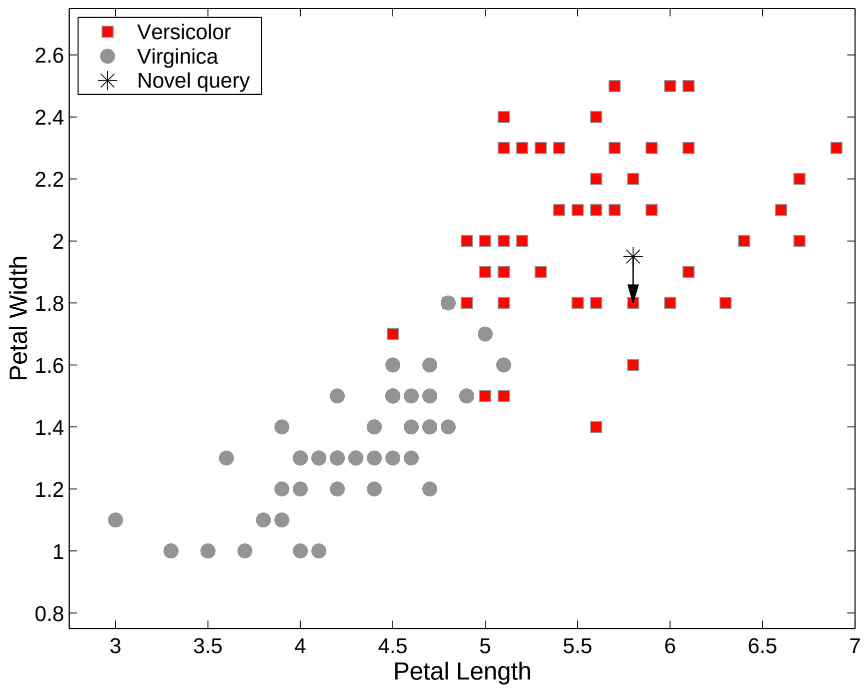


Figure 4-6: Binary classification of irises with nearest neighbor.

classification error is minimized on the training set. For classification, the machine learning community uses a different definition of error than that which one would apply for regression. An example is either classified correctly or incorrectly. Therefore, the error for example i is,

$$E(i) = \begin{cases} 1 & : \text{ if } f(\vec{x_i}) \neq y_i \\ 0 & : \text{ otherwise.} \end{cases}$$

and the error on a set of N examples becomes,

$$error = \frac{1}{N} \sum_{i=1}^{N} E(i) \tag{4.1}$$

To clarify the supervised learning approach with an example, consider the data in Figure 4-6. These data correspond to irises of two different types: Iris versicolor, and Iris verginica, which I represent as orange squares and gray circles respectively in the figure.[4] Based on two features of a flower — the width and the length of the flower's petals — it is fairly easy for a human to correctly discriminate irises of these different types.

[4]This data is part of the University of California Irvine Machine Learning Repository [76].

How can we train a machine learning algorithm so that it can correctly discriminate between objects of different types? One possible approach uses the simple and intuitive *nearest neighbor* algorithm, which is represented as a database of N training examples, $\{\langle \vec{x}_i, y_i \rangle \mid 1 \leq i \leq N\}$. Nearest neighbor computes a label for a *novel* example (*i.e.*, an example that is not in the training set) by inspecting the label of the nearest example in the database. One could use many different metrics to measure the distance between two examples, but a common approach uses Euclidean distance in the feature space.

Thus, the mapping f from feature vectors to labels using Euclidean distance is defined as,

$$f(\vec{x_q}) \equiv y_{i^*} \mid i^* = \underset{1 \leq i \leq N}{\operatorname{argmin}} \{||\vec{x_i} - \vec{x_q}||^2\},$$

where x_q is a novel example.

The asterisk in Figure 4-6 shows a query for a novel example. The arrow in the figure points to the training example that is nearest to the query example using Euclidean distance. The nearest neighbor algorithm would therefore predict that the novel example is an Iris versicolor, because this is the label of the closest training example.

Nearest neighbor, of course, is only one of many supervised learning algorithms devised by the machine learning community. The next section discusses techniques to gauge the performance of a supervised learning algorithm.

4.2.3 Validation for Supervised Learning

The simplest approach, which I mentioned in Section 4.2.1 is called *validation*. It randomly splits the full set of labeled training examples D into two disjoint sets: a training set D_T that is used to train the classifier, and another set D_V that is used test the prediction accuracy of the classifier [33, 84]. In other words, the classifier uses D_T to train a model f, and the prediction error of the classifier is obtained by applying Equation 4.1 to the examples in D_V. Because the examples in the validation set are novel to the classifier, the prediction error of the classifier over all the examples in the validation set provides an estimate of the generalization error [33]; in other words, validation gives us an idea how well a classifier will perform in the real world where it will likely encounter unfamiliar cases.

While the results of this simple approach to validation are informative, leave-one-out cross-validation (LOOCV) likely yields more accurate generalization estimates [33]. LOOCV is an iterative process that iterates N times, where N is the size of the training dataset ($|D|$). On each iteration i, the technique removes the i^{th} example from D, trains the classifier using the remaining $N-1$ examples, and then determines how well the resulting classifier categorizes the left-out example. The predicted generalization accuracy becomes the number of correctly classified left-out examples divided by the total size of the training set, N.

While LOOCV is more robust than simple validation, it is also much more computationally demanding, and thus, LOOCV is not always a feasible approach. For instance, the policy search experiments that I present in Chapter 5 require much simulation time, making LOOCV an unappealing option because it would effectively increase simulation time by a factor of N.

Just as with the regression example I presented in Section 4.2.1, overly complex models tend to overfit the data in the training set. Thus a classification algorithm may choose to sacrifice prediction accuracy on the training set for improved accuracy on novel examples. Different learning algorithms use different mechanisms to manage complexity, but the main approach remains the same: bias a learning algorithm toward simpler models. Genetic programming, which I describe in Chapter 5, uses *parsimony* to favor shorter expressions. The support vector machine classifier that I describe in a later chapter uses *regularization* to limit complexity. Validation and LOOCV help a designer adjust the model parameters to avoid overfitting.

4.3 Reinforcement Learning

In reinforcement learning, an *agent* learns how to adapt to its environment through trial-and-error experimentation [9, 33, 84]. The agent periodically receives a *reinforcement* signal that indicates how beneficial its past decisions were. The reinforcement signal is often *delayed*, which means the agent may make several decisions before it learns how efficacious those decisions were. The goal of the reinforcement learning agent is to use intermittent reinforcement signals to learn the circumstances under which particular actions are desirable.

In some ways, reinforcement learning is more familiar than the other techniques because many problems humans approach fit nicely into this paradigm. As a concrete, real-life example, consider the process of brewing beer. The total brewing process takes several weeks to complete, and at various points during the process the brewer can adjust the flavor, aroma, and mouthfeel by taking certain actions (*e.g.*, adding ingredients, reducing the temperature of the fermentation vessel, adding carbonation). Unfortunately, however, creating the perfect batch of beer is a trial-and-error process, and the brewer cannot really tell if the beer is going to be tasty or subpar for weeks.

To formalize the reinforcement learning approach, at any given time t, the agent (*e.g.*, the brewer) receives some indication about the current state of the environment, $\vec{s_t}$ (*e.g.*, the current state of the beer). Based on $\vec{s_t}$ the agent will perform an action, a_t (*e.g.*, adding an ingredient). The environment responds to the action by transitioning to state $\vec{s}_{t+1}$, and additionally, the system may provide a reinforcement signal, $r(\vec{s_t})$ (*e.g.*, the taste of the beer) that indicates how appropriate the agent's decision was given the circumstances with which it was faced.

There are two main approaches for solving reinforcement learning problems: the first strategy attempts to determine the utility of being in a given state, while the second strategy searches directly for policies that perform well in the agent's environment [51]. Instances of the latter strategy are policy search methods, which the experiments in the next chapter employ. Policy search methods maintain and adjust an explicit representation of a policy, continuing to tweak the policy as long as the agent's performance improves [84]. The mechanism for adjusting an agent's policy depends largely on how the policy is represented, and therefore, I will wait until the next chapter — where I use policy search to find good priority functions — to provide a concrete description of this operation.

Chapter 5

Policy Search for Meta Optimization

This chapter shows that a machine learning approach known as policy search can automatically create effective compiler policies. In particular, policy search is the best approach for constructing *priority functions*, which are prevalent in optimizing compilers. The next section defines priority function. Section 5.2 describes the specific policy search approach that I use to perform the experiments in this chapter. I discuss my experimental methodology in Section 5.3. Sections 5.4 and 5.5 provide two case studies and Section 5.7 concludes.

5.1 Priority Functions

Many compiler heuristics have a focal point: a single *priority* or *cost* function often dictates the efficacy of a heuristic. A priority function — a function of the factors that affect a given problem — measures the relative importance of the different options available to a compiler algorithm.

Put simply, priority functions prioritize the options available to a compiler algorithm. Recall the list scheduling example in Chapter 2, where a priority function assigns a weight to each instruction in the scheduler's dependence graph, dictating the order in which to schedule instructions. The list scheduler proceeds by scheduling *ready* instructions in priority order. In other words, if two instructions are ready to be scheduled, the algorithm will favor the instruction with the higher priority. The scheduling algorithm hinges upon the priority function. Apart from enforcing the legality of the schedule, the scheduler entirely relies on the priority function to make all of its performance-critical decisions.

Mathematically, a priority (or cost) function is a function, $y \leftarrow f(\vec{x})$, where $\vec{x}$ is a vector of features that might be useful for prioritizing options, and $y \in \mathbb{R}$. For the experiments that I perform in this chapter, $\vec{x}$ is a composition of real-valued and Boolean-valued features.

To give the reader a feel for the ubiquity of priority functions, the following list describes some common priority-based heuristics:

- **Hyperblock formation**: Later in this chapter I use the formation of predicated hyperblocks as a case study.

- **Meld scheduling**: Abraham et al. rely on a priority function to schedule across region boundaries [1]. The priority function is used to sort regions by the order in which they should be visited.

- **Modulo scheduling**: In [83], Rau states that "there is a limitless number of priority functions" that can be devised for modulo scheduling. Rau describes the tradeoffs involved when considering scheduling priorities.

- **Register allocation**: Many register allocation algorithms use priority functions to determine which variables to spill if spilling is required. I use register allocation as a case study in this chapter.

- **Loop invariant code motion**: Trimaran's IMPACT compiler uses a cost function to determine whether to hoist an invariant out of a loop [94]. Aggressively hoisting invariants can negatively affect performance by causing extra register pressure.

- **Clustered scheduling**: Özer et al. describe an approach to scheduling for architectures with clustered register files [78]. They note that the choice of priority function has a "strong effect on the schedule." They also investigate five different priority functions [78].

- **Resource allocation**: Gordon et al. use a cost function to distribute computation on a multicore architecture [39].

This is not an exhaustive list of applications, as many other important compiler optimizations employ cost functions of the sort mentioned above.

Compiler writers typically revert to tedious trial-and-error experimentation to find effective priority functions. For instance, Bernstein et al. manually identified three priority functions for a register allocation heuristic [12]. By applying the three functions to a suite of benchmarks, they found that a register allocator's effectiveness is highly dependent on the priority function the compiler uses.

This chapter shows that a machine learning approach known as policy search can automatically create effective priority functions for two important priority-based compiler heuristics: hyperblock formation, and register allocation. The rational for tuning priority functions is based on the insight that these small functions are crucial for performance. I experimentally verify this assumption in this chapter.

Policy search can be used to cater a priority function to a specific input program by using the SASI or SAMI training approaches. More importantly, the techniques can use a MASI or MAMI training approach to find a general-purpose function that works well for a broad range of applications. In this mode of operation, policy search can perform the tedious work that is currently performed by engineers. For both of the case studies I describe in this chapter, policy search was able to at least match the performance of human-generated priority functions. In some cases it created superior priority functions.

The next section describes the representation and operation of the policy search method that I use to perform the experiments presented in this chapter.

5.2 Policy Search with Genetic Programming

Policy search is a form of reinforcement learning that searches for policies that work well in an *agent's* current environment. In the context of this chapter's material, the agent refers to the compiler heuristic that we wish to construct. For priority-based compiler heuristics, the policy is the priority function.[1]

Many practitioners of reinforcement learning use lookup tables or neural networks to represent policies. In this chapter I take an alternate approach that uses genetic programming (GP), which is a type of "evolutionary" algorithm [41, 73]. GP is well-suited to the problem of fine-tuning priority functions:

- GP is especially appropriate when the relationships among relevant variables are poorly understood [56]. Such is the case with compiler policies, which often feature uncertain tradeoffs. Today's complex systems also introduce uncertainty.
- GP can handle large state spaces [73]. As I mention above, many reinforcement learning approaches use tables to represent state-action pairs. Unfortunately, the state spaces of the policies for which I search in this chapter are far too big to store in a table.
- GP is a distributed algorithm. With the cost of computing power at an all-time low, it is now economically feasible to dedicate a cluster of machines to searching a solution space.
- GP solutions are human readable. The individuals on which GP operates are parse trees which can easily be converted to free-form arithmetic equations. Other machine learning representations, such as neural networks, are not as comprehensible.

GP maintains a population of parse trees [56]. For this problem, each parse tree is an expression that represents a priority function. As with natural selection, expressions are chosen for reproduction (called crossover) according to their level of fitness. Expressions that best solve the problem are most likely to have progeny. The algorithm also randomly mutates some expressions to innovate a possibly stagnant population.

GP, like genetic algorithms, is a type of evolutionary algorithm; and as such, GP's flow is identical to that of genetic algorithms. As Figure 5-1(a) shows, the algorithm begins by creating a population of initial expressions. We can seed the population with human-generated

[1]Throughout the remainder of this chapter I interchange the terms, *priority function*, *policy*, *expression*, and *individual*.

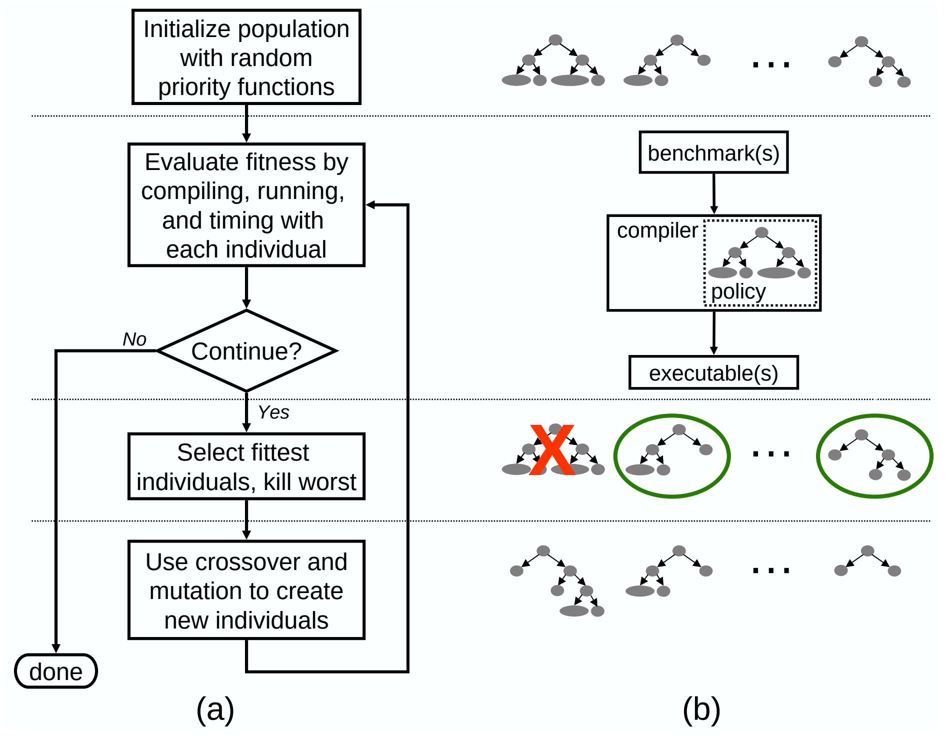
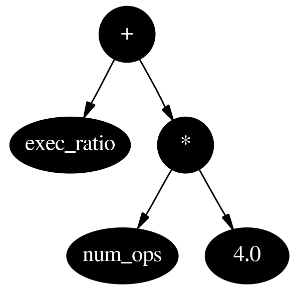

Initialize population with random priority functions
Evaluate fitness by compiling, running, and timing with each individual
Continue?
No
Yes
Select fittest individuals, kill worst
Use crossover and mutation to create new individuals
done
(a)
benchmark(s)
compiler
policy
executable(s)
(b)

Figure 5-1: Genetic programming flow. Part (a) shows the flow of GP policy search to find compiler priority functions, and part (b) graphically depicts the operation of each phase.

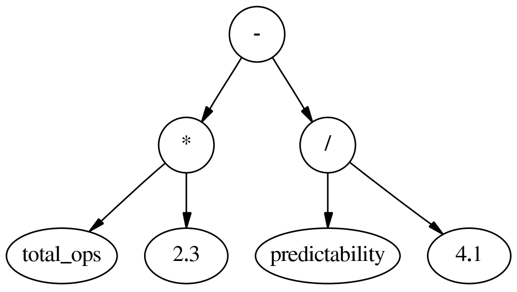
heuristics, or we can simply start with randomly generated expressions.[2] The algorithm then determines each expression's level of fitness. In this context, compilers that produce the *fastest* code are fittest. To determine the fitness level of an expression, the algorithm compiles and runs a set of benchmarks using the expression as a heuristic's priority function (as shown in Figure 5-1(b)). The priority functions that, on average, lead to the fastest compiler-generated executables are the fittest. Once the algorithm reaches a user-defined limit on the number of generations, the process stops; otherwise, the algorithm proceeds by probabilistically choosing the best expressions for crossover. Some of the offspring undergo mutation, and the algorithm continues.

Unlike other evolutionary algorithms, which use fixed-length string representations, GP's expressions are variable in length and free-form. Figure 5-2 provides several examples of genetic programming individuals. Variable-length individuals do not artificially constrain evolution by limiting representable solutions. However, without special consideration, indi-

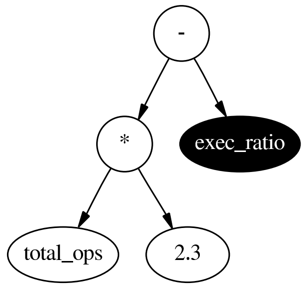
[2]For the experiments I perform in this chapter, seeding the initial population with hand crafted policies had little effect on the outcome.



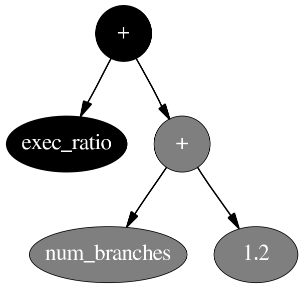
(a) GP individual



(b) GP individual



(c) Crossover



(d) Mutation

Figure 5-2: GP individuals. Part (a) and (b) show examples of GP individuals. Part (c) provides an example of a random crossover of the individuals in (a) and (b). Part (d) shows a mutation of the expression in part (a).

viduals grow exponentially during crossover and mutation.

My system rewards *parsimony* by selecting the smaller of two otherwise equally fit expressions [56], which helps limit model complexity.[3] Parsimony also facilitates inspection by human engineers, and without it, expressions quickly become unintelligible. In addition to fine-tuning policies, I envision this work to function as a tool for compiler writers and architects to identify important combinations of features.

Figure 5-2, part (c) provides an example of crossover, the method by which two expressions reproduce. Here the two expressions in (a) and (b) produce offspring. Crossover works by selecting a random node in each parent, and then swapping the subtrees rooted at those

[3]It may even be worthwhile to enforce parsimony even more strongly by favoring good-performing terse expressions over better-performing complex expressions, but this work does not consider such an approach.

nodes.[4] The crux of GP is that crossover propagates 'good' subexpressions.

GP is more likely to select expressions with favorable subexpressions for crossover, thereby further disseminating fit subexpressions. Our system uses tournament selection to choose expressions for crossover. Tournament selection chooses N expressions at random from the population and selects the one with the highest fitness [56]. N is referred to as the tournament size, and small values of N reduce selection pressure; expressions are only compared against the other $N - 1$ expressions in the tournament.

Finally, part (d) shows a mutated version of the expression in (a). Here, a randomly generated expression supplants a randomly chosen node in the expression. For details on the mutation operators I implemented, see [8].

To find general-purpose expressions (*i.e.*, expressions that work well for a broad range of input programs), the learning algorithm learns from a *set* of training programs. To train on multiple input programs, I use the technique described by Gathercole in [36]. The technique — called dynamic subset selection (DSS) — trains on subsets of the training programs, concentrating more effort on programs that perform poorly compared to the baseline heuristics. DSS reduces the number of fitness evaluations that need to be performed in order to achieve a suitable solution. Because my system must compile and run benchmarks to test an expression's level of fitness, fitness evaluations for this problem are costly.

The next section describes the methodology that I use throughout the remainder of the chapter.

5.3 Methodology

My system uses policy search to automatically search for effective priority functions. I search for effective policies by wrapping the iterative framework of Figure 5-1(a) around the compiler and architecture. As Figure 5-1(b) reveals, I replace the policy component

[4]Selection algorithms must use caution when selecting random tree nodes. If we consider a full binary tree, then leaf nodes comprise over 50% of the tree. Thus, a naive selection algorithm will choose leaf nodes over half of the time. For this reason I employ depth-fair crossover, which equally weighs each level of the tree [52].

of a compiler heuristic with an expression parser and evaluator. This allows policy search to quickly test the efficacy of a priority function in the population, by compiling, running, and evaluating the fitness of the applications in the training set using said expression as the policy. The expressions that create the fastest executables for the applications in the training set are favored for crossover.

Notice that the selection of applications to include in the training set is an important aspect of the training process. For instance, an embedded systems developer might want to specialize the compiler to perform extremely well on the application under development; thus, the developer would only include a single application in the training set. I use the Single Application Single Input (SASI) training approach for program specialization. SASI assigns fitnesses based on an application's performance on a single input. To create general-purpose policies I use the Multiple Application Single Input (MASI) training approach. The system will find policies that perform well, on average, on the set of applications in the training set. Please refer to Section 3.4 for a description of the SASI and MASI training approaches.

For both the SASI and MASI experiments I perform, my system uses *whole-program* execution time to assign fitnesses. This approach focuses on frequently executed procedures, and therefore, it may slowly converge upon general-purpose solutions (*i.e.*, it will cater a policy to the (usually small) subset of code that accounts for most of the execution time of the application(s)). However, when using a SASI approach, we actually want to focus on the 'hot' methods, and thus, this evaluation of fitness works well. The alternative is to perform method-level timing and average the performance over all of the methods in the program. While this approach could facilitate the search for general-purpose solutions, it would also require a compiler instrumentation pass to insert timers that would measure the runtimes of individual methods.

Table 5.1 shows the GP expression primitives that my system uses. Careful selection of GP primitives is essential. We want to give the system enough flexibility to potentially find unexpected results. However, the more leeway we give GP, the longer it will take to converge upon a general solution. Notice in particular that there are two primitives that return constants: `rconst`, and `bconst`. Whenever the GP system generates an expression

Real-Valued Function	**Representation**
$Real_1 + Real_2$	(add $Real_1$ $Real_2$)
$Real_1 - Real_2$	(sub $Real_1$ $Real_2$)
$\begin{cases} Real_1/Real_2 & : \text{ if } Real_2 \neq 0 \\ 0 & : \text{ if } Real_2 = 0 \end{cases}$	(div $Real_1$ $Real_2$)
$Real_1 \cdot Real_2$	(mul $Real_1$ $Real_2$)
$\begin{cases} Real_1 \cdot Real_2 & : \text{ if} Bool_1 \\ Real_2 & : \text{ if not} Bool_1 \end{cases}$	(cmul $Bool_1$ $Real_1$ $Real_2$)
$\sqrt{Real_1}$	(sqrt $Real_1$)
$\begin{cases} Real_1 & : \text{ if} Bool_1 \\ Real_2 & : \text{ if not} Bool_1 \end{cases}$	(tern $Bool_1$ $Real_1$ $Real_2$)
Returns real constant K	(rconst K)
Returns real value of arg from environment	(rarg arg)

Boolean-Valued Function	**Representation**
$Bool_1$ and $Bool_2$	(and $Bool_1$ $Bool_2$)
$Bool_1$ or $Bool_2$	(or $Bool_1$ $Bool_2$)
not $Bool_1$	(not $Bool_1$)
$Real_1 < Real_2$	(lt $Real_1$ $Real_2$)
$Real_1 > Real_2$	(gt $Real_1$ $Real_2$)
$Real_1 = Real_2$	(eq $Real_1$ $Real_2$)
Returns Boolean constant	(bconst $\{true, false\}$)
Returns Boolean value of arg from environment	(barg arg)

Table 5.1: GP primitives. My GP system uses the primitives and syntax shown in this table. The top segment represents the real-valued functions, which all return a real value. Likewise, the functions in the bottom segment all return a Boolean value.

with an `rconst` terminal, it will assign the terminal a uniformly generated number on the unit interval (0,1). The system can compose multiple constants via arithmetic operators to create constants outside the unit interval. In addition, random mutation of the `rconst` terminal perturbs the constant according to a Gaussian distribution. Similarly, `bconst` terminals are randomly initialized with a Boolean constant, and mutation may choose to toggle that value.

My system creates an initial population that consists of 399 randomly generated expressions of varying heights using the primitives in Table 5.1 and features extracted by the compiler writer. There is no restriction on what features a compiler writer could include; however, for efficiency's sake, the compiler writer should only include program characteristics that she thinks may be important for forming good priority functions.

In addition to the randomly generated expressions, I seed the initial population with the

Parameter	**Setting**
Population size	400 expressions
Number of generations	50 generations
Generational replacement	22% of expressions
Mutation rate	5%
Tournament size	7
Elitism	Best expression is guaranteed survival.
Fitness	Average speedup over the baseline on the suite of benchmarks.

Table 5.2: GP parameters. This table shows the GP parameters I used to collect the results in this section.

compiler writer's best guess. In other words, I include the priority function distributed with the compiler. For two of the three optimizations presented in this chapter, I found that the seed was quickly obscured and weeded out of the population as more favorable expressions emerged. In fact, for hyperblock selection, which I discuss later in this chapter, the seed had little impact on the final solution. These results suggest that one could use policy search to construct priority functions from scratch rather than trying to improve upon preexisting functions.

Table 5.2 summarizes the parameters that I use to collect results. I chose the parameters in the table after a moderate amount of experimentation. I give the GP system 50 generations to find a solution. For the benchmarks that I surveyed, the time required to run for 50 generations is about one day per benchmark in the training set.[5] Because fitness evaluations are so costly — requiring a compile and run of a benchmark — my system memoizes benchmark fitnesses.

After every generation the system randomly replaces 22% of the population with new expressions created via the crossover operation presented in Section 5.2. Only the best expression is guaranteed survival. Typically, GP practitioners use much higher replacement rates. However, since I use dynamic subset selection, only a subset of benchmarks is evaluated in a generation. Thus, I need a lower replacement rate in order to increase the likelihood that a given expression will be tested on more than one subset of benchmarks. The system mutates roughly 5% of the new expressions. Finally, I use a tournament size of 7 when selecting the fittest expressions. This setting causes moderate selection pressure.

[5]We ran on 15 to 20 machines in parallel for the experiments in Section 5.4 and Section 5.5.

The following two sections present case studies and build upon the methodology I described above. I include results for both of the case studies in their respective sections.

5.4 Case Study I: Hyperblock Formation

This section describes the operation of policy search in the context of a specific compiler optimization: hyperblock formation. I first describe the *predication* paradigm on which hyperblocks rely, and then I discuss factors (or features) that might be important when creating a priority function for a hyperblock formation heuristic. I conclude the section by presenting experimental results for this optimization.

5.4.1 Branching *v.* Predication

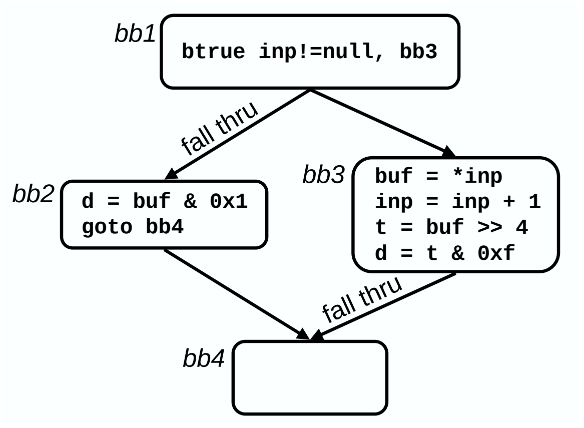
Processors fetch and execute the instructions of a basic block in order. Consider the example code in Figure 5-3(a), and its corresponding control flow graph in part (b). The target of the branch instruction on line (1) depends on whether or not `inp` is `null`. Figure 5-3(c) shows a simple 5-stage processor pipeline based on the pedagogical DLX processor described in [81]. The processor cannot resolve the branch condition until the EX stage, which means the processor begins executing two instructions (in the IF and ID stages) without knowing if they should be executed.

Figure 5-3(c) shows the default mode of operation, where the processor fetches and executes instructions in order: by the time the processor resolves the branch condition — and hence knows whether the branch should *fall through* to the next instruction, or jump to a different address — it has already begun executing the instructions on lines (2) and (3). If the branch condition (`inp != null`) is false, the processor does not miss a beat: it is already executing the correct instructions. If on the other hand, the branch condition turns out to be true, the processor must *nullify* the partially executed instructions in the IF and ID stages because they should not be executed; the processor can then continue by fetching and executing the correct instructions at *bb3*.

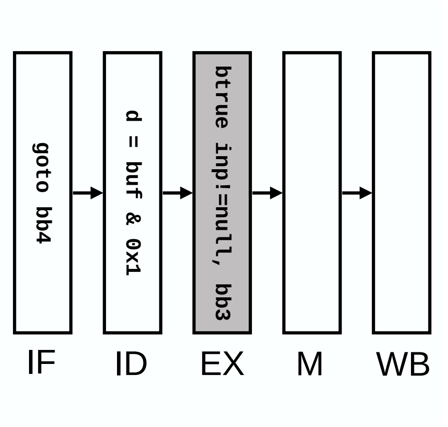
With each processor revision, computer architects increased the pipeline depth and the

```
(1) bb1: btrue inp!=null, bb3
(2) bb2: d = buf & 0x1
(3)      goto bb4
(4) bb3: buf = *inp
(5)      inp = inp + 1
(6)      t = buf >> 4
(7)      d = t & 0xf
(8) bb4:
```

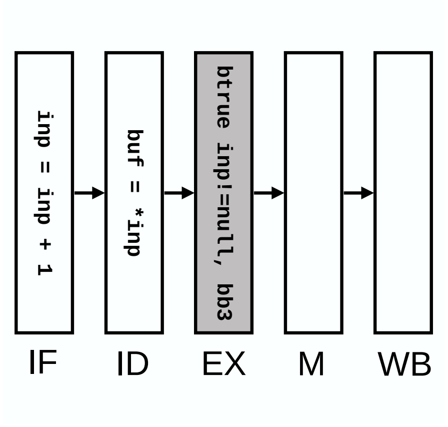
(a) Sample IR



(b) Control flow graph



(c) Fall through execution



(d) Branch predicted execution

Figure 5-3: Influence of control flow on processing. Part (a) shows a simple code fragment, and (b) shows the corresponding CFG. Part (c) shows a simple processor pipeline to illustrate the effects of branches. The processor might have already partially executed instructions it should not have because the branch direction is not resolved until the EX stage. Part (d) shows how branch prediction can predict likely instructions to fetch and execute.

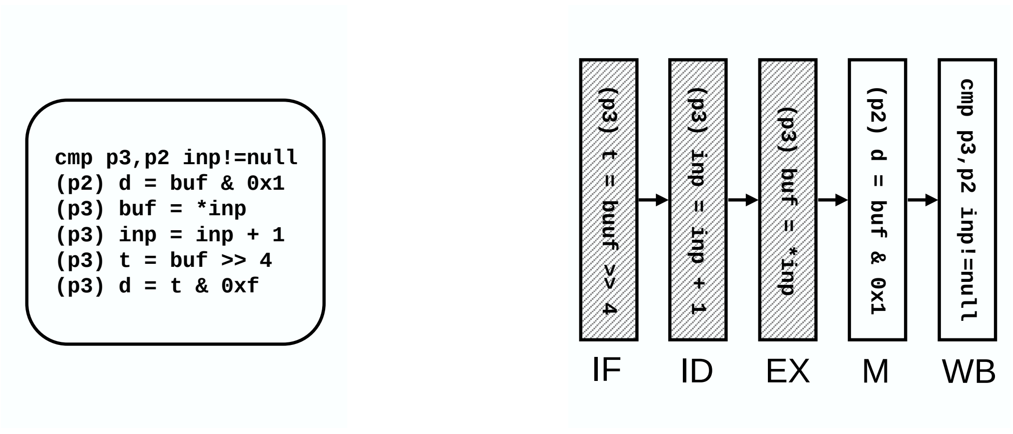
number of instructions executed per cycle, which exacerbates the potential penalty of nullifying wrongly executed instructions. Fortunately, computer architects quickly realized that for many branches, the branch outcome is predictable. Branch prediction allows the processor to speculatively execute instructions along a *likely* path of control flow (*i.e.*, a likely sequence of instructions during execution). See [81] for a detailed discussion of branch prediction.

When the processor's branch predictor is correct, execution proceeds smoothly, and there is no need for the processor to nullify instructions. For instance, Figure 5-3(d) shows the pipeline that results when the branch predictor predicts that the processor should branch to *bb3*. However, when the prediction is incorrect, the processor once again executes instructions on a path that should not have executed; it has to *nullify* these mispredicted instructions, restore the processor state, and begin executing the instructions that should have been executed.

Due to its small scale, the simple processor pipeline diagrammed in Figure 5-3 is only of academic interest. Consider the costs of misprediction on a real modern-day processor: the Pentium® 4 architecture features 20 pipeline stages, and because it is a superscalar processor (meaning it can execute multiple instructions per pipeline stage) it has to nullify up to 126 in-flight instructions when it mispredicts [47]. Improved branch prediction algorithms obviously increase processor utilization. Unfortunately, some branches are inherently unpredictable, and hence, even the most sophisticated prediction algorithm would fail. For such branches, *predication* may be a fruitful alternative.

Rather than rely on branch prediction, predication allows a processor to simultaneously execute instructions from the taken and fall-through paths of control flow. The processor nullifies all instructions in the incorrect path. In this model, a *predicate operand* guards the execution of every instruction. If the value of the operand is true, then the instruction executes normally. If however, the operand is false, the processor nullifies the instruction, preventing it from modifying processor state.

Using a process dubbed if-conversion, a predicating compiler merges disjoint paths of execution into a predicated hyperblock [67]. A hyperblock is a predicated single-entry, multiple-exit region. Figure 5-4(a) shows the hyperblock corresponding to the control-flow in Figure 5-3(a). Here, `p2` and `p3` are mutually exclusive predicates that are set according to the branch condition in part (a). When `inp` is `null` the `cmp` instruction sets `p2` to true, and `p3` to false. Thus, although *all* the instructions in this example hyperblock technically execute, only the instruction that is guarded by `p2` modifies processor state; the processor nullifies all the instructions that are guarded by `p3`. Likewise, when `inp` is not `null` `p3` becomes true and `p2` false.



```
cmp p3,p2 inp!=null
(p2) d = buf & 0x1
(p3) buf = *inp
(p3) inp = inp + 1
(p3) t = buf >> 4
(p3) d = t & 0xf
```

(a) A hyperblock

(b) Model of execution

Figure 5-4: Hyperblocks and predicated execution. Part (a) shows the hyperblock that results from the control flow in Figure 5-3(b). Part (b) shows how all instructions in this example execute to completion, though only instructions that are guarded by true predicates are allowed to modify processor state. For instance, if `p3` were false, then the processor would ignore the instructions in the shaded instruction slots.

Figure 5-4(b) shows the execution of the hyperblock in part (a) on a simple pipeline. The processor executes all the instructions in the hyperblock, but only instructions that are guarded by a true predicate are allowed to modify processor state. The example illustrates the case where `p2` is true. Although predication can effectively expose ILP, simply predicating everything will diminish performance by saturating machine resources with useless instructions. If `p2` is true all of the time for instance, the instructions guarded by `p3` consume valuable execution slots. An appropriate balance of predication and branching can drastically improve performance.

5.4.2 Feature Extraction

In the following list I give a brief overview of several features that are useful to consider when trying to balance predication versus traditional control flow. Again, a *path* refers to a sequence of basic blocks connected by edges in the control flow graph:

- **Path predictability**: Predictable branches incur no misprediction penalties, and thus, should probably remain unpredicated. Combining multiple paths of execution into a single predicated region uses precious machine resources [67]. In this case, using machine resources to parallelize individual paths is typically wiser.

- **Path frequency**: Infrequently executed paths are probably not worth predicating. Including the path in a hyperblock would consume resources, and could negatively affect performance.

- **Path ILP**: If a path's level of parallelism is low, it may be worthwhile to predicate the path. In other words, if a path does not fully use machine resources, combining it with another sequential path probably will not diminish performance. Because predicated instructions do not need to know the value of their guarding predicate until late in the pipeline, a processor can sustain high levels of ILP.

- **Number of instructions in path**: Long paths use more machine resources, and if predicated, may slow execution. On the other hand, the cost of misprediction is relatively high for short paths. If the processor mispredicts on a short path, the processor has to nullify all the instructions in the path, *and* the subsequent control-independent instructions fetched before the branch condition resolves.

- **Unbalanced paths**: Since every instruction in a hyperblock executes, long paths effectively delay the time to completion of short paths.

- **Number of branches in path**: Paths of control through several branches have a greater chance of mispredicting. Therefore, it may be worthwhile to predicate such paths. On the other hand, including several such paths may produce large hyperblocks that saturate resources.

- **Compiler optimization considerations**: Paths that contain hazard conditions (*i.e.*, pointer dereferences and procedure calls) limit the effectiveness of many compiler optimizations. In the presence of hazards, a compiler must make conservative assumptions. Depending on the details of the target architecture, the code in Figure 5-3(a) might benefit from predication. Without architectural support, the load from `*inp` cannot be hoisted above the branch. The program will behave unexpectedly if the load is not supposed to execute and it accesses protected memory. By removing branches from the instruction stream, predication affords the scheduler freer code motion opportunities. The predicated hyperblock in Figure 5-4 allows the scheduler to rearrange memory operations without control-flow concerns.

- **Machine-specific considerations**: A heuristic should account for machine characteristics. For instance, the number of parallel resources and the branch delay penalty are decisive factors.

Clearly, there is much to consider when designing a heuristic for hyperblock selection. Many of the above considerations make sense on their own, but when they are put together, contradictions arise. Finding the right mix of criteria to construct an effective priority function is nontrivial for human designers.

5.4.3 Trimaran's Heuristic

I use the Trimaran infrastructure to evaluate the ideas in this chapter [94]. Trimaran is a freely available integrated compiler and Explicitly Parallel Instruction Computing (EPIC) processor simulator. Trimaran's IMPACT compiler creates predicated hyperblocks [67, 68]. The IMPACT compiler begins by transforming the code so that it is more amenable to hyperblock formation [67]. IMPACT's algorithm then identifies acyclic paths of control that are suitable for hyperblock inclusion. Park and Schlansker detail this portion of the algorithm in [80]. A priority function — which is the critical calculation in the predication decision process — assigns a value to each of the paths based on characteristics such as those I described in the last subsection [67]. Some of these characteristics come from runtime profiling.

IMPACT uses the priority function shown below:

$$h_i = \begin{cases} 0.25 & : \text{ if } path_i \text{ contains a hazard.} \\ 1 & : \text{ if } path_i \text{ is hazard free.} \end{cases}$$

$$d\_ratio_i = \frac{dep\_height_i}{\max_{j=1\rightarrow N} dep\_height_j}$$

Feature	Description
Registers	64 general-purpose registers, 64 floating-point registers, and 256 predicate registers.
Integer units	4 fully-pipelined units with 1-cycle latencies, except for multiply instructions, which require 3 cycles, and divide instructions, which require 8.
Floating-point units	2 fully-pipelined units with 3-cycle latencies, except for divide instructions, which require 8 cycles.
Memory units	2 memory units. L1 cache accesses take 2 cycles, L2 accesses take 7 cycles, and L3 accesses require 35 cycles. Stores are buffered, and thus require 1 cycle.
Branch unit	1 branch unit.
Branch prediction	2-bit branch predictor with a 5-cycle branch misprediction penalty.

Table 5.3: Architectural characteristics. This table describes the EPIC architecture on which I evaluate policy search. This model approximates the Intel Itanium architecture.

$$o\_ratio_i = \frac{num\_ops_i}{\max_{j=1 \to N} num\_ops_j}$$

$$priority_i = exec\_ratio_i \cdot h_i \cdot (2.1 - d\_ratio_i - o\_ratio_i) \tag{5.1}$$

The heuristic applies the above equation to all paths in a predicatable region. Based on a runtime profile, *exec_ratio* is the probability that the path is executed. The priority function also penalizes paths that contain hazards (*e.g.*, pointer dereferences and procedure calls). Such paths may constrain aggressive compiler optimizations. To avoid large hyperblocks, the heuristic is careful not to choose paths that have a large dependence height (*dep_height*) with respect to the maximum dependence height. Similarly it penalizes paths that contain many instructions (*num_ops*).

After IMPACT's algorithm assigns path priorities, it proceeds by merging together the paths with the highest priorities into a predicated hyperblock. The algorithm stops merging paths when it has consumed the target architecture's estimated resources.

Feature	**Description**
dep_height	The maximum instruction dependence height over all instructions in path.
num_ops	The total number of instructions in the path.
exec_ratio	How frequently this path is executed compared to other paths considered (from profile).
num_branches	The total number of branches in the path.
predictability	Average path predictability obtained by simulating a branch predictor (from profile).
predict_product	Product of branch predictabilities in the path (from profile).
avg_ops_executed	The average number of instructions executed in the path (from profile).
$unsafe\_JSR$	If the path contains a subroutine call that may have side-effects, it returns *true*; otherwise it returns $false$.
$safe\_JSR$	If the path contains a side-effect free subroutine call, it returns *true*; otherwise it returns $false$.
mem_hazard	If the path contains an unresolvable memory access, it returns *true*; otherwise it returns $false$.
max_dep_height	The maximum dependence height over all paths considered for hyperblock inclusion.
total_ops	The sum of all instructions in paths considered for hyperblock inclusion.
num_paths	Number of paths considered for hyperblock inclusion.

Table 5.4: Hyperblock selection features. The compiler writer chooses interesting attributes, and the system evolves a priority function based on them. Some of these parameters require profiling the code before compilation. I also include the min, mean, max, and standard deviation of path characteristics. This provides some global information to the greedy local heuristic.

5.4.4 Experimental Setup

This section discusses the experimental setup for optimizing Trimaran's hyperblock selection priority function. Trimaran is an integrated compiler and simulator for a parameterized EPIC architecture. Table 5.3 details the specific architecture over which I applied policy search. This model resembles Intel's Itanium® architecture.

Recall that a priority function is a function, $y \leftarrow f(\vec{x})$, where $\vec{x}$ is a feature vector and y is a real-valued priority. I modified Trimaran's IMPACT compiler by replacing its hyperblock formation priority function (Equation 5.1) with my GP expression parser and evaluator. This allows IMPACT to read a GP policy, $f_{GP}(\vec{x})$, and evaluate it based on the values of human-selected features ($\vec{x}$) that might be important for creating effective priority functions.

Table 5.4 describes these features.

The hyperblock formation algorithm passes the features in the table as parameters to the expression evaluator. For instance, if a GP individual contains a reference to *dep_height*, the path's dependence height will be used when the expression is evaluated. Most of the characteristics in Table 5.4 were already available in IMPACT. I also extract the minimum, maximum, mean, and standard deviation of all path-specific characteristics in the table.

I added a 2-bit dynamic branch predictor to Trimaran's processor simulator and I modified the compiler's profiler to extract branch predictability statistics. Lastly, I enabled the following compiler optimizations: function inlining, loop unrolling, backedge coalescing, acyclic global scheduling [22], modulo scheduling [96], hyperblock formation, register allocation, machine-specific peephole optimization, and several classic optimizations.

5.4.5 Experimental Results

I use the benchmarks in Table 5.5 to test policy search. I include all of the Trimaran certified benchmarks [94].[6] The benchmark suite also includes many of the Mediabench applications [62]. The build process for ghostscript proved too difficult to compile, and I had to exclude the remainder of the Mediabench applications because the Trimaran system does not compile them correctly.[7]

I begin by presenting results for application-specialized heuristics. Following this, I show that it is possible to use policy search to create general-purpose heuristics.

Specialized Priority Functions

One can employ policy search using the SASI approach to create a priority function that is catered exactly to one application. Figure 5-5 shows that policy search is extremely effective

[6]Due to preexisting bugs in Trimaran, I could not get 134.perl to execute correctly, though [94] certified it.

[7]I exclude cjpeg, the complement of djpeg, because it does not execute properly when compiled with some priority functions. My system can also be used to uncover bugs!

Benchmark	**Suite**	**Description**
codrle4 decodrle4	See [15]	RLE type 4 encoder/decoder.
huff_enc huff_dec	See [15]	A Huffman encoder/decoder.
djpeg	Mediabench	Lossy still image decompressor.
g721encode g721decode	Mediabench	CCITT voice compressor/decompressor.
mpeg2dec	Mediabench	Lossy video decompressor.
rasta	Mediabench	Speech recognition application.
rawcaudio rawdaudio	Mediabench	Adaptive differential pulse code modulation audio encoder/decoder.
toast	Mediabench	Speech transcoder.
unepic	Mediabench	Experimental image decompressor.
085.cc1	SPEC92	gcc C compiler.
052.alvinn	SPEC92	Single-precision neural network training.
179.art	SPEC2000	A neural network-based image recognition algorithm.
osdemo mipmap	Mediabench	Part of a 3-D graphics library similar to OpenGL.
129.compress	SPEC95	In-memory file compressor and decompressor.
023.eqntott	SPEC92	Creates a truth table from a logical representation of a Boolean equation.
132.ijpeg	SPEC95	JPEG compressor and decompressor.
130.li	SPEC95	Lisp interpreter.
124.m88ksim	SPEC95	Processor simulator.
147.vortex	SPEC95	An object oriented database.

Table 5.5: Benchmarks used. The set includes applications from the SpecInt, SpecFP, and Mediabench benchmark suites, as well as a few miscellaneous programs.

on a per-benchmark basis. The dark bar shows the percentage speedup (over Trimaran's baseline heuristic) of each benchmark when run with the same input data set with which it was trained. The light bar shows the speedup attained when the benchmark processes a data set that was not used to train the priority function. I refer to the alternate data set as the *test* data set, which is essentially a validation input set for gauging how well the learned policy applies to unseen data.

Intuitively, in most cases the training input data achieves a better speedup. Policy search, as I have applied it, finds priority functions that excel on the training input data. The test input data likely exercises different paths of control flow— paths which may have been unused during training. Nonetheless, in every case, the application-specific priority function outperforms the baseline. The mpeg2dec benchmark is a particularly interesting

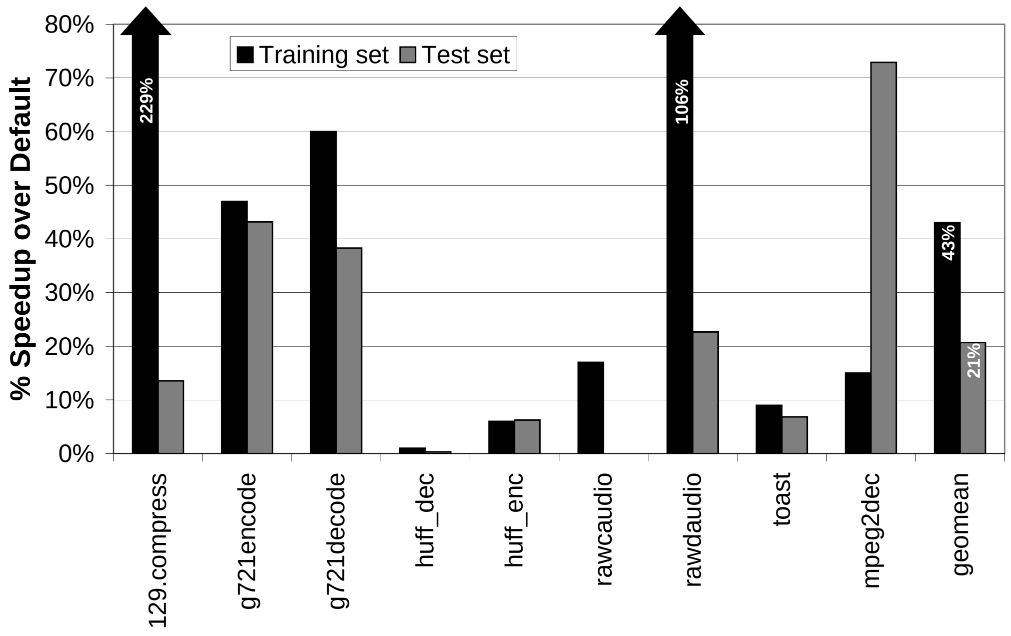


Figure 5-5: Hyperblock specialization using the SASI training approach. This graph shows the speedups obtained by training on a per-benchmarks basis. The dark colored bars are executions using the same data set on which the specialized priority function was trained. The light colored bars are executions that use an alternate, or *test* data set.

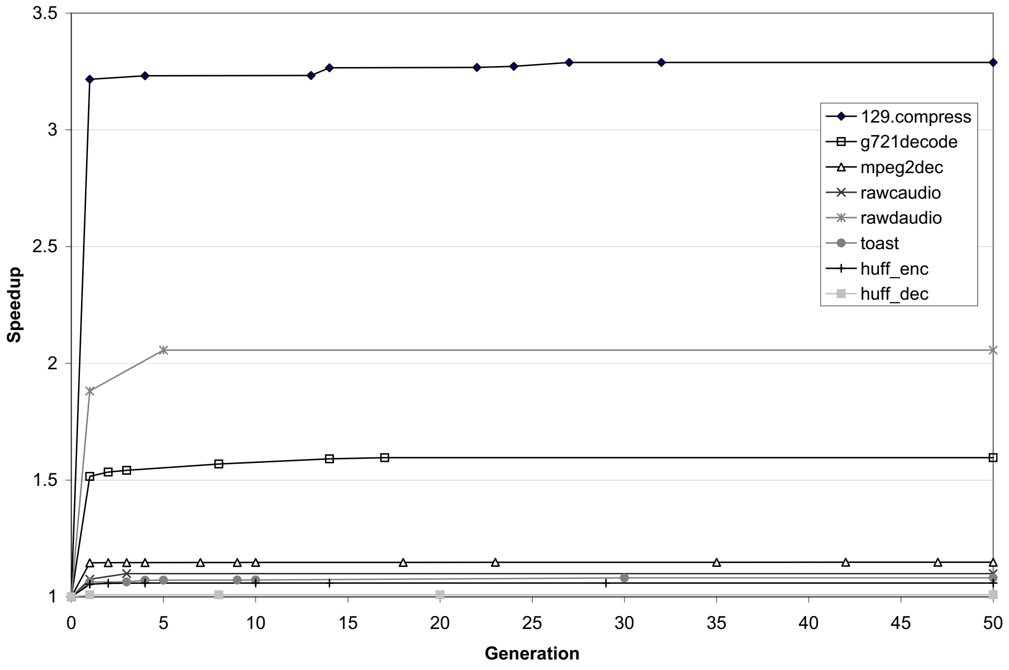


Figure 5-6: Hyperblock formation evolution. This figure graphs the best fitness over generations. For this problem, policy search quickly finds a priority function that outperforms Trimaran's baseline heuristic.

case. Policy search found a priority function that improves the performance of mpeg2dec's method `Decode_MPEG2_Intra_Block` by a factor of four. That method accounts for 10% of the total execution with the training set, and a whopping 43% of the execution with the test set. Because the method accounts for a much larger fraction of the total execution time in the test set, the learned priority function has a much larger impact.

Figure 5-6 shows fitness improvements over generations. In many cases, policy search finds a superior priority function quickly, and finds only marginal improvements as the evolution continues. In fact, the baseline priority function is quickly obscured by GP-generated expressions. Often, the *initial* population contains at least one expression that outperforms the baseline. This means that by simply creating and testing 399 random expressions, I was able to find a priority function that outperformed Trimaran's for the given benchmark.

Once policy search has discovered a decent solution, the search space and GP operator dynamics are such that most "offspring" are worse, some are equal, and very few are actually better. This seems indicative of a steep hill in the solution space. In addition, multiple reruns using different initialization seeds reveal minuscule differences in performance. It might be a space in which there are many local maxima with similar fitness levels.

General-Purpose Priority Functions

This section discusses how we can use the MASI training approach to create a general-purpose hyperblock formation priority function.[8] Instead of creating a specialized priority function for each benchmark, in this section I aim to find one priority function that works well for all the benchmarks in the training set. To this end, I apply GP policy search over the entire training set using dynamic subset selection [36]. To create a training set, I divided the benchmarks in Table 5.5 into two sets:[9] a training set, and a validation set.

Figure 5-7 shows the results of applying the single best priority function to the benchmarks in the training set. The dark bar associated with each benchmark is the speedup over Trimaran's base heuristic when the training input data is used. This data set yields a

[8]The lack of inputs for the suite of benchmarks Trimaran compiles precludes using a MAMI approach.

[9]I chose to train mostly on Mediabench applications because they compile and run faster than the Spec benchmarks.

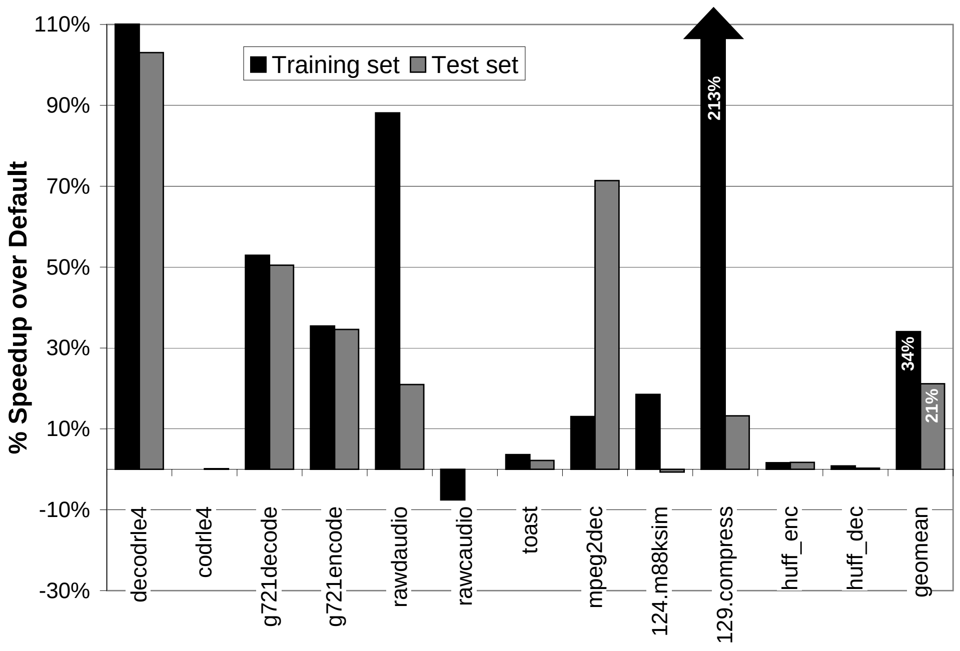


Figure 5-7: Training on multiple benchmarks using the MASI training approach. Policy search found a *single* priority function that works well for all the benchmarks in this graph. The dark bars represent speedups obtained by running the given benchmark on the same data that was used to train the priority function. The light bars correspond to a test data set.

34% improvement on average using the geometric mean (44% arithmetic). Note that the MASI training approach seeks the policy that attains the best *average* improvement on the training set. Policy search finds a priority function that performs well on average, but notice that it slightly degrades the performance of rawcaudio. The light bar shows results when the test input data is used. The overall improvement for this set is 21% (25% arithmetic). Again, mpeg2dec performs very well on the test set.

I then apply the resulting priority function to the benchmarks in the validation set. The machine learning community refers to this as validation. Since the benchmarks in the validation set are not related to the benchmarks in the training set, this is a measure of the policy's generality.[10]

Figure 5-8 shows the results of validation. This experiment applies the best priority function

[10]In Chapter 4 I discussed a more thorough validation technique called leave-one-out cross validation that would provide a better sense of a policy's generality. However, due to the computational demands of GP policy search, cross validation would have drastically lengthened the duration of these already long-running experiments.

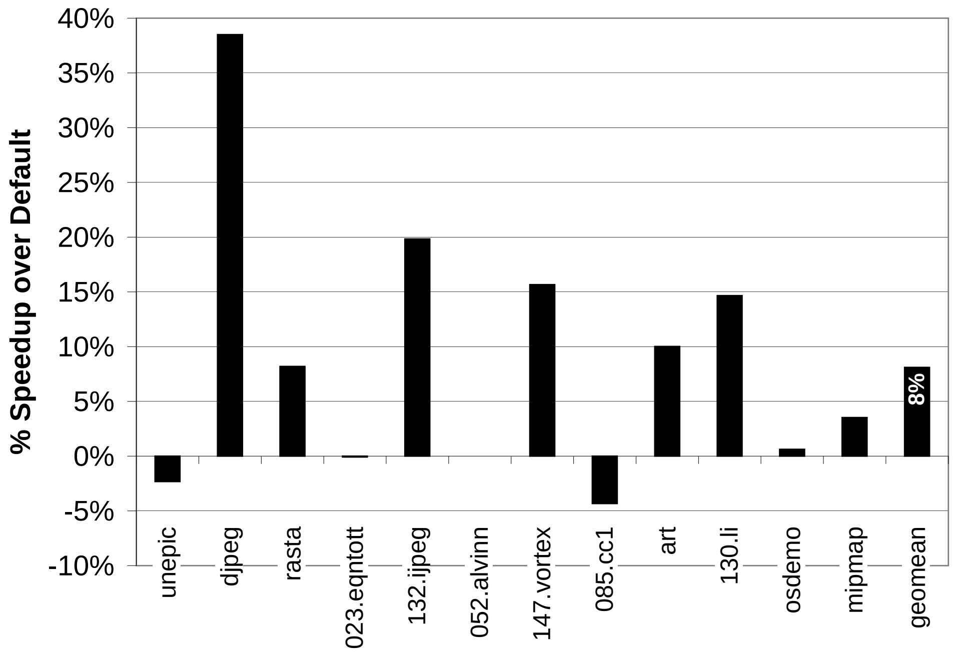


Figure 5-8: Validation of the general-purpose priority function. The best priority function found by training on the benchmarks in Figure 5-7 is applied to the benchmarks in this graph.

on the training set to the benchmarks in the test set. The average speedup on the test set is 8% (9% arithmetic). In three cases (unepic, 023.eqntott, and 085.cc1) Trimaran's baseline heuristic marginally outperforms the GP-generated priority function. For the remaining benchmarks, the heuristic policy search found is better.

The Best Priority Function

Figure 5-9 shows the best general-purpose priority function my system found for hyperblock selection. Because parsimony pressure favors small expressions, most of my system's solutions are readable. Nevertheless, I hand-simplified the expressions presented in this chapter for ease of discussion.

Notice that some parts of the expression have no impact on the overall result. For instance, removing the sub-expression on line (2) will not affect the heuristic; the value is invariant to a scheduling region since the mean execution ratio is the same for all paths in the region. Such 'useless' expressions are called *introns* in GP nomenclature. According to theory,

```
(1) (add
(2)  (sub (mul exec_ratio_mean 0.8720) 0.9400)
(3)  (mul 0.4762
(4)      (cmul (not mem_hazard)
(5)       (mul 0.6727 num_paths)
(6)       (mul 1.1609
(7)        (add
(8)         (sub
(9)          (mul
(10)          (div num_ops dep_height) 10.8240)
(11)         exec_ratio)
(12)        (sub (mul (cmul has_unsafe_jsr
(13)                        predict_product_mean
(14)                        0.9838)
(15)                   (sub 1.1039 num_ops_max))
(16)              (sub (mul dep_height_mean
(17)                        num_branches_max)
(18)                   num_paths)))))))
```

Figure 5-9: The best priority function my system found for hyperblock scheduling.

introns are useful for preserving good building blocks during crossover and mutation [56].

The conditional multiply statement on line (4) directly affects the priority function: it favors paths that do not have pointer dereferences (because the sub-expression in line (5) will always be greater than one). Pointers inhibit the effectiveness of the scheduler and other compiler optimizations, and thus dereferences should be penalized. The IMPACT group came to the exact same conclusion, though the extent to which they penalize dereferences differs [67].

The sub-expression on line (8) favors 'bushy' parallel paths, where there are numerous independent operations. This result is somewhat counterintuitive since highly parallel paths will quickly saturate machine resources. In addition, paths with higher *exec_ratio*'s are slightly penalized, which also defies intuition.

The conditional multiply expression on line (12) penalizes paths with unsafe calls (*i.e.*, calls to subroutines that may have side effects). Once again this agrees with the IMPACT group's reasoning [67].

Because Trimaran is such a large and complicated system, it is difficult to know exactly why the priority function in Figure 5-9 works well. This is exactly the point of using a methodology like meta optimization. The bountiful complexities of compilers and systems are difficult to understand. Also worthy of notice is the fact that policy search gets such

Level	**Access time**
Registers	0 cycles.
L1 cache	1 cycle.
L2 cache	5 cycles.
L3 cache	12-15 cycles.
Main memory	180-225 cycles

Table 5.6: Memory hierarchy access times for the Intel Itanium ®2 procesor [46].

good speedups, particularly on the training set, by changing such a small portion of the compiler.

5.5 Case Study II: Register Allocation

This case study shows that we can use GP policy search to automatically create register allocation priority functions. The motivation behind register allocation is the widening gap between processor speeds and main memory. Table 5.6 shows access times for Intel's Itanium 2 processor in processor cycles for the various levels of the memory hierarchy. Almost all processors have a small set of registers that the processor can access very quickly — faster than any other level of the memory hierarchy. Unlike with the other levels however, the compiler explicitly manages register access.

Register allocation is an optimization that tries to allocate processor registers to program variables. This optimization has a large effect on performance because registers are so much faster to access than the other levels of the memory hierarchy. Because it is often the case that the register allocator cannot fit all of a program's variables in registers, many register allocation algorithms use cost functions to determine which variables to spill to memory when spilling is required. For instance in priority-based coloring register allocation, the priority function is an estimate of the relative benefits of storing a given variable in a register [23].

Priority-based coloring first associates a *live range* with every variable. A live range is the composition of basic blocks through which the associated variable's value must be preserved. The algorithm then tries to allocate each live range to a register in priority order. Trimaran's compiler uses Chow's register allocation approach and the following functions to compute the priority of a given live range, lr [23]:

$$savings_i = w_i \cdot (LDsave \cdot uses_i + STsave \cdot defs_i) \tag{5.2}$$

$$priority(lr) = \frac{\sum_{i \in lr} savings_i}{N} \tag{5.3}$$

Equation 5.2 computes the savings of allocating the live range in basic block i. The terms $LDsave$ and $STsave$ are estimates of the execution time saved by keeping the associated live range in a register for references and definitions respectively. The terms $uses_i$ and $defs_i$ represent the number of uses and definitions of the live range in block i, and w_i is the estimated execution frequency for the block.

Equation 5.3 sums the savings over the N blocks that compose the live range. Thus, this priority function represents the savings incurred by accessing a register instead of resorting to memory. The algorithm then tries to assign registers to live ranges in priority order. Please see [23] for a complete description of the algorithm. For our purposes, the important thing to note is that the success of the algorithm depends on the priority function.

The priority function described above is intuitive— it assigns weights to live ranges based on the estimated execution savings of register allocating them. Nevertheless, as I will describe below, my system automatically finds functions that improve the heuristic by up to 11%.

5.5.1 Experimental Results

I collected these results using the same experimental setup that I used for hyperblock selection. Again, I use Trimaran and I target the architecture described in Table 5.3. However, to more effectively stress the register allocator, I trim the number of general-purpose registers and floating-point registers to 32 apiece.

I modified Trimaran's Elcor register allocator by replacing its priority function (Equation 5.2) with the GP policy search parser and evaluator. The register allocation heuristic described above essentially works at the basic block level. Equation 5.3 simply sums and normalizes the priorities of the individual basic blocks. For this reason, I stay within the algorithm's framework and leave Equation 5.3 intact.

Table 5.7 shows a subset of the features that I extracted for register allocation.

Feature	**Description**
region_weight	The number of times the region was executed (from profile).
live_ops	The number of live operations in the live range.
num_defs	The number of definitions created in the live range.
num_uses	The number of uses in the range.
num_refs	The number of references in the range.
store_latency	The latency of storing a value to memory.
load_latency	The latency of loading a value from memory.
move_latency	The latency of moving a value from one register to another.
num_calls	The number of procedure calls made in the block.
num_refs	The total number of references made in the block.
num_forbidden_regs	The number of registers that are not available to the live range (because it interferes with an allocated live range).
reg_file_size	The number of registers in the bank associated with the type of variable being considered.
entry_edges	The number of entry edges to the live unit.
exit_edges	The number of exit edges from the live unit.
inf_lranges	The number of interfering live ranges.
num_lus	The number of basic blocks in the live range.
avg_refs	The average number of references per basic block in the live range.
stddev_refs	The standard deviation of the number of references per basic block in the live range.
has_single_ref	If the block has a single reference this returns *true*, otherwise it returns *false*.
has_single_block	If there is only one block in the live range, this returns *true*, otherwise it returns *false*.
is_pass_through	Returns *true* if there are no live references in this block, and *false* otherwise.
is_gpr	Returns *true* if the live range corresponds to an integer type variable.
is_fpr	Returns *true* if the live range corresponds to a floating-point variable.
is_pr	Returns *true* if the live range corresponds to predicate variable.

Table 5.7: Register allocation features. This table describes the features I extracted to learn a register allocation policy. Some of these features come directly from profile information.

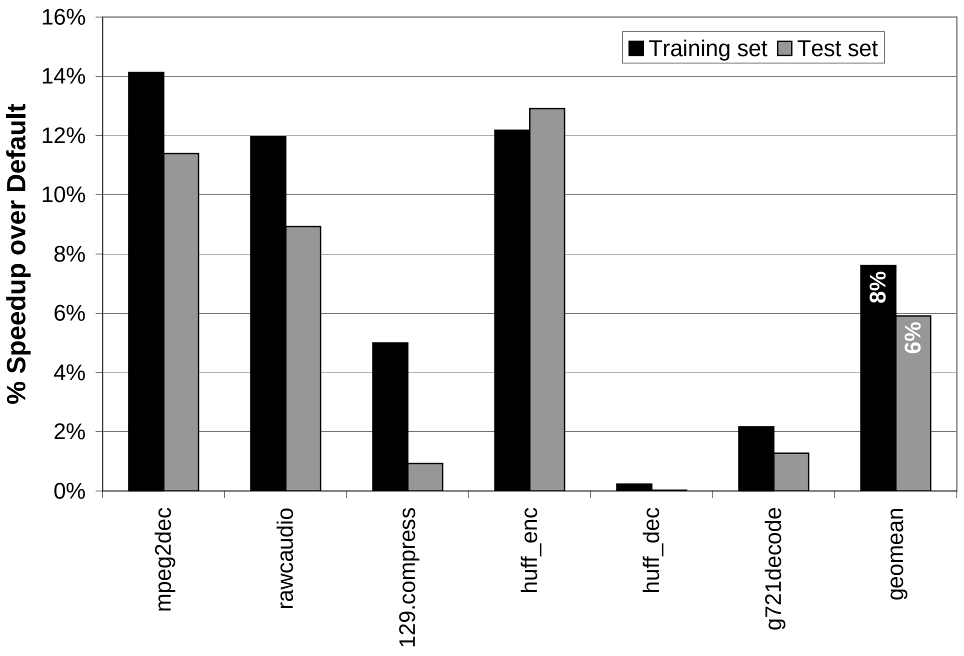


Figure 5-10: Register allocation specialization using SASI training. This graph shows speedups obtained by training on a per-benchmarks basis. The dark colored bars are executions using the same data set on which the specialized priority function was trained. The light colored bars are executions that use a test data set.

Specialized Priority Functions

These results indicate that policy search works well, even for well-studied heuristics. Figure 5-10 shows speedups obtained by specializing Trimaran's register allocator for a given application using the SASI training approach. The dark bar associated with each application represents the speedup obtained by using the same input data that was used to specialize the heuristic. The light bar shows the speedup when the benchmark processes a test data set. Again, because of differences in profiles, the huff_enc benchmark performs slightly better on the test data than it does on the training data.

However, overall the training input data outperforms the alternate input data. In the case of register allocation however, we see that the disparity between speedups on training and novel data is less pronounced than it is with hyperblock selection. This is because hyperblock selection is extremely data-driven. An examination of the general-purpose hyperblock formation heuristic reveals two dynamic factors (*exec_ratio* and *predict_product_mean*) that are critical components in the hyperblock decision process.

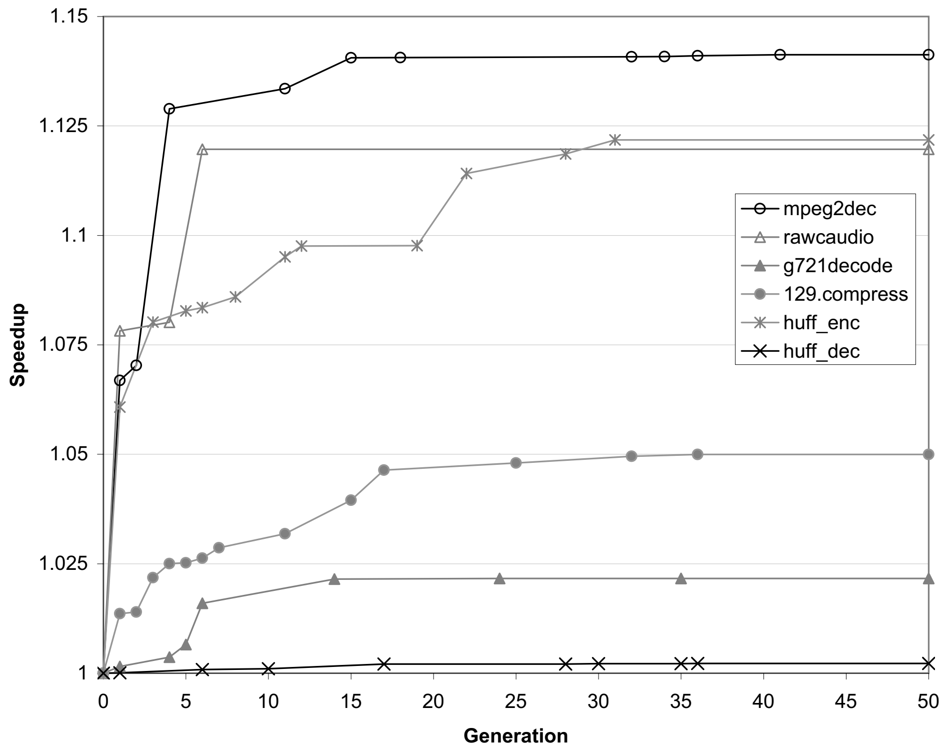


Figure 5-11: Register allocation evolution. This figure graphs fitness over generations. Unlike the hyperblock selection evolution, these fitnesses improve gradually.

Figure 5-11 graphs fitness improvements over generations. It is interesting to contrast this graph with Figure 5-6. The fairly constant improvement in fitness over several generations seems to suggest that this problem is harder to optimize than hyperblock selection. Additionally, unlike the hyperblock selection algorithm, the baseline heuristic typically remained in the population for several generations.

General-Purpose Priority Functions

Just as in Section 5.4.5, I divide the benchmarks into a training set and a test set and use the MASI training approach.[11] The benchmarks in Figure 5-12 show the training set for this experiment. The figure also shows the results of applying the best priority function to all the benchmarks in the set. The dark bar associated with each benchmark is the speedup over Trimaran's baseline heuristic when using the training input data. The average for this data set is 3%. Again, we see that policy search attempts to maximize overall performance, and

[11]This experiment uses smaller test and training sets due to preexisting bugs in Trimaran. It does not correctly compile several of the benchmarks when targeting a machine with 32 registers.

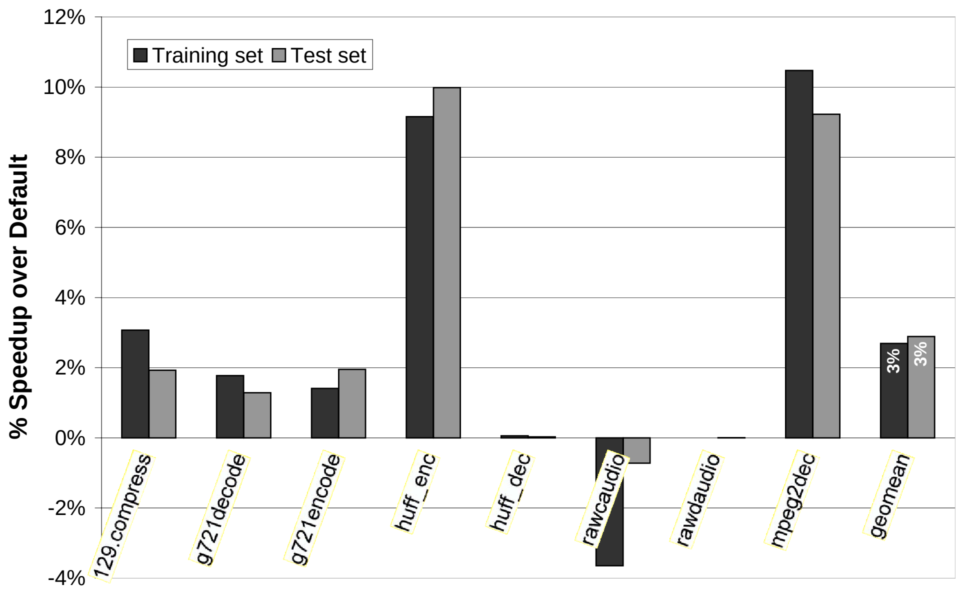


Figure 5-12: Training a register allocation priority function using the MASI training approach. Policy search used all the benchmarks in this figure to find a general-purpose priority function. The single best priority function was applied to all the benchmarks. The dark bars represent speedups obtained by running the given benchmark on the same data that was used to train the priority function. The light bars correspond to a test data set.

as a result, the performance of rawcaudio suffers. This data point is particularly interesting because using a SASI approach, policy search improved the effectiveness of the register allocator by 12% for rawcaudio. On a test data set the best priority function also attains an average speedup of 3%, which indicates that register allocation is not as susceptible to variations in input data.

Figure 5-13 shows the validation results for this experiment. The figure shows the percent speedup (over Trimaran's baseline) achieved by applying the single best priority function to a set of benchmarks that were not in the training set. The learned priority function outperforms the baseline for all benchmarks except decodrle4 and 132.ijpeg. Although the overall speedup on the cross validation set is only 2%, this is an exciting result. With about a week's worth of intensive processing, my technique improved upon the popular Chow and Hennessey [23] register allocation approach.

Figure 5-14 provides another perspective on these results. The data in the graph compares the machine-learned register allocation priority function against a simple function that returns a constant value (and thus, equally weighs all spill candidates) on the vali-

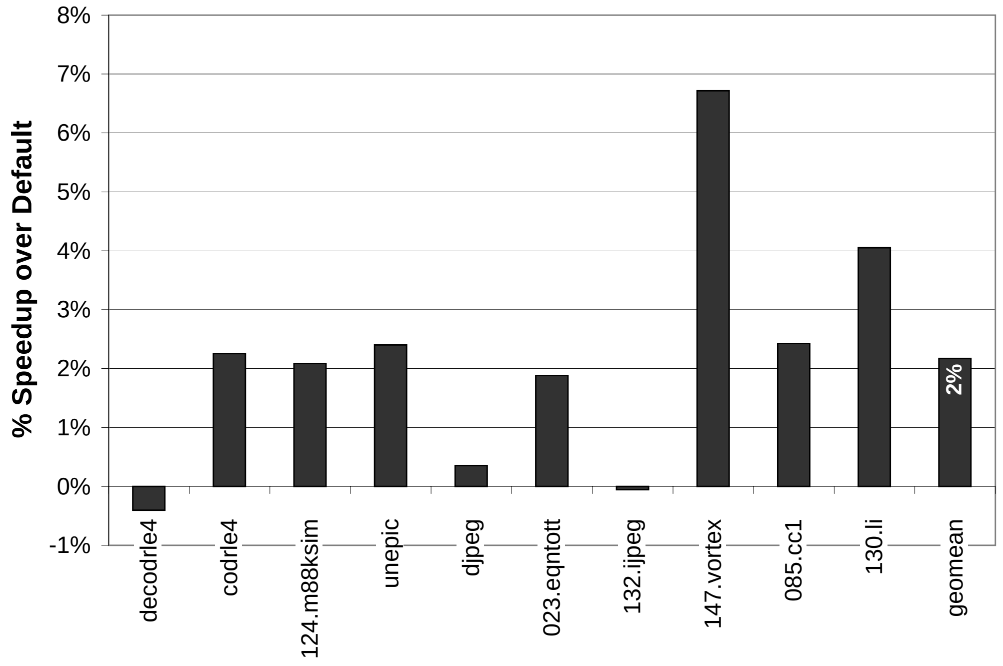


Figure 5-13: Validation of the general-purpose register allocation priority function. The best priority function found by policy search is applied to the benchmarks in this graph.

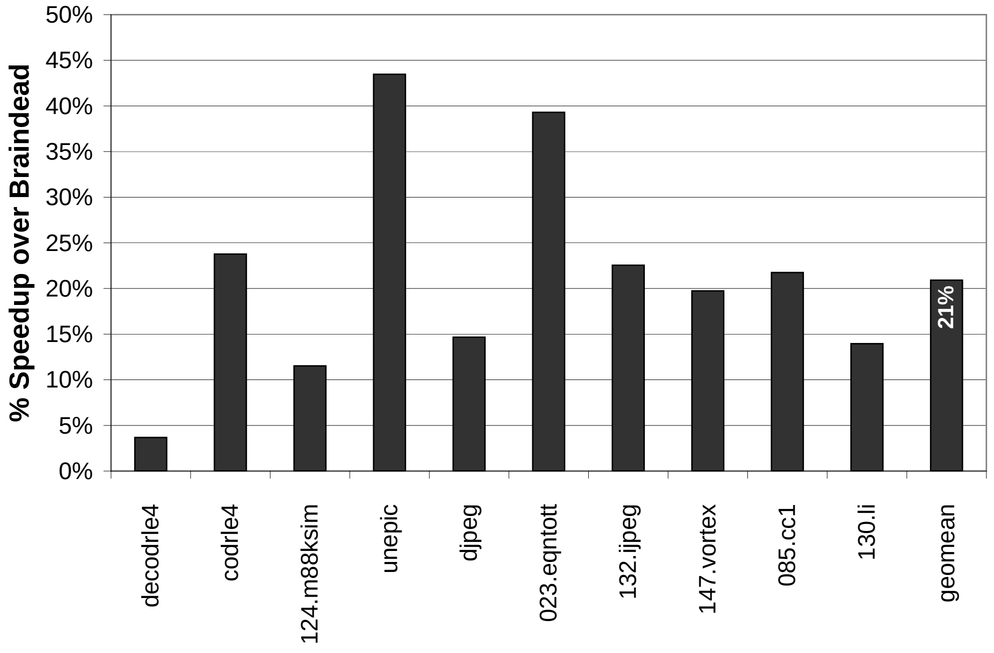


Figure 5-14: Speedups over a simple register allocation policy. This graph shows the importance of the register allocation priority function. When compared against a priority function that returns a constant value, the machine-learned function achieves large speedups.

```
(add
 (mul
  (mul (tern (eq num_refs live_ops)
              (sub (div move_latency 0.3949) 0.8015)
              (mul
               0.6297
               (sqrt (mul
                      (sub (mul live_ops reg_file_size)
                            (sub 0.1345 caller_benefit))
                      (mul (mul (tern
                                   (eq num_refs live_ops)
                                   0.6165
                                   caller_benefit)
                                  (add (sub
                                         (add
                                          (mul live_ops reg_file_size)
                                          0.1541)
                                         (mul 0.3049
                                               (sub 0.1345 caller_benefit)))
                                        num_uses))
                            (cmul has_single_ref 2.0 num_defs))))))
        (add (sqrt (mul 0.5113 live_ops)) caller_benefit))
  (add (sub
         (add (mul live_ops reg_file_size) 0.1541)
         (mul 0.3049
               (sub 0.1345 caller_benefit)))
         num_uses))
 (add 0.8655 caller_benefit)))
```

Figure 5-15: The best general-purpose priority function my system found for register allocation.

dation set. This graph highlights the impact that a small priority function can have on overall performance. This graph also suggests that policy search can find powerful priority functions.

Figure 5-15 shows the best general-purpose register allocation priority function that policy search found.

5.6 Effort and Training Time Involved

This section discusses the effort that is involved with heuristic creation using GP policy search. The GP system I created includes a training harness in which a developer can specify a training set: the harness reads a configuration file that describes the applications to include in the training set, as well as the commands needed to compile, run, and test the correctness of each application.

There are two major steps that are not automated: first, as I have already mentioned the designer needs to extract features that adequately distill the various "choices" available to the optimization. For most experiments I performed, this step was easy and required at most a few days of work. Then, the designer can simply replace the human-constructed priority function with a call to the GP evaluator library function, passing the vector of features as an argument.

The remainder of the learning process is entirely automated. Creating application-specific priority functions using the SASI approach required around a day of processing using 15 Linux, Pentium 4 workstations. Of course, the training time depends on the size and running time of the application on which they system trains. However, because I used a *simulated* machine for these experiments, which drastically slows down evaluation, a training time of one day is probably an overestimate.

For the experiments I perform in this Chapter, creating general-purpose priority functions using the MASI approach required around a week of processing using 15 Linux, Pentium 4 workstations. Again, training time is entirely unsupervised. For the MASI approach, it is important to emphasize that this is a *one-time process*. One can directly use the priority function that the GP policy search system has found; from this point on, there will be no additional compile-time overheads.

5.7 Conclusion

This chapter demonstrates two main points: first, while priority functions may represent a small portion of a compiler's code base, they can drastically affect performance; second, policy search with genetic programming can find effective priority functions with little human intervention. For the priority-based hyperblock formation algorithm, policy search attained huge performance increases ($> 20\%$) on the training set, and a substantial performance improvement on a validation set (8%). The speedups for register allocation were not as drastic, achieving roughly 3% improvements over the baseline. Nevertheless, policy search managed to automatically find a priority function that outperforms a well-studied human-crafted heuristic. That in itself is an exciting result.

Several aspects of GP make it amenable to policy search. GP created high-quality priority functions effectively despite the rather large set of features with which it was presented. Both case studies I present in this chapter make use of large feature sets. In a way, the crossover operator performs *feature selection* by propagating good building blocks: the important features are passed on from a fit GP individual to its progeny. In this regard GP was extremely easy to use. I thought little about what features were the most important, and instead just provided any feature that I thought *might* be useful for designing a policy.

As pleased as I am with this chapter's results, GP was somewhat frustrating at times. As Table 5.2 shows, there are a lot of knobs to tweak in GP; and the whole point of using policy search is that we do not want to have to manually tweak compiler heuristics! Fortunately, from my experience with GP, it was *fairly* robust to different parameter settings. In addition, my system was completely unsupervised: I chose parameter settings, started the system, and examined the best priority function some number of days later. My system definitely reduces the net level of tweaking.

One crucial aspect that this chapter fails to address is noise. GP relies on determining small differences in fitness. On a real machine, small variances are hard to determine because of noise (*i.e.*, subsequent runs of the same program will likely produce different fitnesses). Collecting results on a simulator is much easier because back-to-back runs of the same program will produce the same fitness. However, using a simulator introduces two more obstacles: first, simulator inaccuracies may produce misleading results, and second, simulation is much slower than native execution.

The experiments I perform in the next chapter use a real (noisy) machine and an alternate approach to automatic heuristic design.

Chapter 6

Supervised Learning for Meta Optimization

Policy search can be applied to search for any compiler policy. However, in some cases, supervised learning can more efficiently ‘solve’ hard compilation problems. This chapter uses loop unrolling — a well-known optimization for exposing instruction level parallelism — as a case study to demonstrate the effectiveness of supervised learning. Because loop unrolling indirectly affects so many aspects of system performance, it is difficult to model the appropriateness of the optimization. In this chapter I show that *near neighbor* (NN) classification and *support vector machines* (SVM) work remarkably well for predicting unroll factors. The best machine-learned classifier can predict with 65% accuracy the optimal unroll factor, and the optimal *or* second-best unroll factor 79% of the time.

I evaluate the implications of improved unrolling decisions using the Open Research Compiler (ORC) and an Itanium® 2 architecture. The best classifier achieves a 5% speedup (over ORC’s heuristic) for the SPEC 2000 benchmarks, and a 9% speedup for the floating pointing benchmarks in that suite. This chapter adds to the evidence that suggests machine learning techniques can model systems at least as well as human designers, and with much less effort.

The next section describes the loop unrolling optimization. Section 6.2 presents the methodology and infrastructure that I use for this chapter’s experiments. I discuss multi-class

```
void addScalar (char* a, int c, int l) {
     for(int i = 0; i < l; i++) {
        a[i] = a[i] + c;
     }
}
```

(a) Original version.

```
void addScalar (char* a, int c, int l) {
     int i = 0;
     for(; i < l % 4; i++) {
        a[i] = a[i] + c;
     }

     for(; i < l; i+=4) {
        a[i] = a[i] + c;
        a[i+1] = a[i+1] + c;
        a[i+2] = a[i+2] + c;
        a[i+3] = a[i+3] + c;
     }
}
```

(b) Unrolled version.

Figure 6-1: Loop unrolling example. The loops in part (b) result from unrolling the loop in part (a). The first loop in part (b) assures that the unrolled second loop will execute to completion.

classification in Section 6.3. Section 6.4 provides concrete results, Section 6.5 describes a methodology for identifying the most salient features of an optimization, and finally I conclude in Section 6.6.

6.1 Loop Unrolling

Loop unrolling is a well known transformation in which the compiler replicates a loop body some number of times. Figure 6-1 illustrates the operation of this optimization. Loop unrolling decreases the number of times the backward branch (back to the top of the loop) is encountered during the execution of a loop. This can be particularly important for architectures that have high branching overhead. However, loop unrolling is primarily used to enable other optimizations, many of which target the memory system. For example, unrolling creates multiple static memory instructions corresponding to dynamic executions of a single operation. After unrolling, these instructions can be rescheduled to exploit memory locality. If the loop accesses the same memory locations on consecutive iterations, many of these references can be eliminated altogether with scalar replacement. Another method to reduce memory traffic utilizes a wide memory bus to transfer multiple words with a single load or store operation. Unrolling is key to exposing adjacent memory references [30, 60] so that they can be merged into a single wide reference.

Arguably, the most important aspect of loop unrolling is its ability to expose instruction level parallelism (ILP) to the compiler. After unrolling, the compiler can reschedule the operations in the unrolled body to achieve overlap among iterations. Such a scheme was first used in the Bulldog compiler [34] and is still important in compiling for machines that support a high degree of ILP. Typically, unrolling is combined with other transformations that increase the size of the scheduling window. Examples include trace scheduling [34] and hyperblock formation [69].

Loop unrolling is an interesting optimization because it indirectly affects many aspects of system performance: it affects the efficacy of the instruction scheduler, the software pipeliner, the register architecture, and the memory system. Because its impact is mainly noticed in secondary effects, it is difficult to decide when this optimization is appropriate. Superficially, loop unrolling appears to be an optimization that is always beneficial. However, in many cases it will impair performance. The following non-exhaustive list considers some possible drawbacks to loop unrolling:

- The most acknowledged detriment of unrolling is that code expansion can degrade the performance of the instruction cache.
- Added scheduling freedom can result in an increase in the live ranges of variables, resulting in additional register pressure. Since memory spills and reloads are typically long latency operations, this can negate the benefits of unrolling.
- Control flow also complicates unrolling decisions. If the compiler cannot determine that a loop may take an early exit, it will actually have to add control flow to the unrolled loop which may negate — or at the very least neutralize — the benefits of unrolling [75].
- Some compilers aggressively speculate on memory accesses. Execution time will increase if the scheduler chooses to speculatively hoist unrolled memory accesses that dynamically conflict.

The scheduler, the register allocator, and the underlying architecture interact in nontrivial ways; loop unrolling increases the aggressiveness of certain optimizations, which, depending on the circumstances, may adversely affect other important optimizations and reduce

Feature
The loop nest level.
The number of operations in loop body.
The number of floating point operations in loop body.
The number of branches in loop body.
The number of memory operations in loop body.
The number of operands in loop body.
The number of implicit instructions in loop body.
The number of unique predicates in loop body.
The estimated latency of the critical path of loop.
The estimated cycle length of loop body.
The language (C or Fortran).
The number of parallel "computations" in loop.
The maximum dependence height of computations.
The maximum height of memory dependencies of computations.
The maximum height of control dependencies of computations.
The average dependence height of computations.
The number of indirect references in loop body.
The minimum memory-to-memory loop-carried dependence.
The number of memory-to-memory dependencies.
The tripcount of the loop (-1 if unknown).
The number of uses in the loop.
The number of definitions in the loop.

Table 6.1: A subset of features used for loop classification. These characteristics are used to train the classifiers.

overall performance. The only way to truly know what will work is to empirically evaluate decisions; even human-designed models must be evaluated at some point to determine their effectiveness. It is the goal of this research to use empirical observations to train a learning algorithm how to make informed decisions.

6.2 Methodology and Infrastructure

This section describes the methodology and infrastructure that I use to perform the loop unrolling experiments. In this section I describe my general approach, the compiler and platform I use, the loop instrumentation process, and the benchmarks that I use to test the efficacy of my techniques.

6.2.1 Supervised Learning

I use a machine learning approach known as supervised learning to induce effective loop unrolling heuristics. As described in Section 4.2.2, supervised learning uses a training set of examples to learn a function f that maps a feature vector $\vec{x_i}$ to a prediction y_i:

$$y_i \leftarrow f(\vec{x_i}) \tag{6.1}$$

Unlike the experiments that I performed in the last chapter, all elements of a feature vector $\vec{x_i}$ are real-valued, and the predictions are integer values within a constrained range: $y_i \in \{1, 2, \ldots, 8\}$. The limit on the range of predictions is due to the fact that a substantial number of benchmarks run incorrectly for unroll factors beyond eight.[1]

In my experiments, the feature vector contains loop characteristics such as the trip count of the loop, the number of operations in the loop body, the programming language the loop is written in, etc. I extract a feature vector for each unrollable loop in my suite of benchmarks. Table 6.1 shows a subset of the features that the experiments in this chapter use. In all, I collected 38 features for these experiments, but as I later discuss, all of these features are not needed; we can use feature selection techniques to find a small subset of these features that classifies the data extremely well.

In addition to the feature vector, I also extract a training label for each unrollable loop in the benchmark suite. The training label indicates which (mutually exclusive) unroll factor results in the highest performance for the training example. Labeling the data is relatively straightforward: I measure the total runtime of each loop using eight different unroll factors $(1, 2, \ldots, 8)$, and the label for the loop is the unroll factor that yields the best performance. Thus, for each example loop i, I extract a vector of characteristics $\vec{x_i}$ that describes the loop, and a label y_i that indicates what the empirically found best action for the loop is. The task of a classifier is to learn a good function f using a set of N training examples $\{\langle \vec{x}_i, y_i \rangle \mid 1 \le i \le N\}$.

While supervised learning is trained offline, one can easily incorporate the learned classifier into a compiler.

[1]Technically, any policy should be correct; my approach exposed bugs in the ORC compiler.

6.2.2 Compiler and Platform

I use the Open Research Compiler (ORC v2.1) [77] — an open source research compiler that targets Itanium architectures — to evaluate the benefits of applying learning to loop unrolling. ORC is a well-engineered compiler whose performance rivals commercial compilers. The experiments in this chapter target a 1.3 GHz Itanium 2 server running Red Hat Linux Advanced Server 2.1. I use `-O3` optimizations for all of these experiments. However, I disable the software pipelining pass because it inhibits the operation of the loop unroller.

6.2.3 Loop Instrumentation

A training example for supervised learning consists of a feature vector and its corresponding label. This section describes the machinery I implemented to acquire the label for a given loop.

Because this chapter is concerned with loop optimizations, I wrote an *instrumentation pass* that inserts code to measure the runtime of innermost loops. For every loop in a program, the instrumentor assigns a unique counter identifier and an associated memory location to store timer information. Immediately before execution of the program reaches an innermost loop, the instrumentation code captures the processor's cycle counter and places it in the loop's associated memory location. When the loop exits, the cycle counter is again captured; the instrumentation code uses the saved initial timer value and the final timer value to compute the total running time of the loop.

I invested much engineering effort to minimize the impact that the instrumentation code has on the execution of the program. I initially inserted procedure calls to an instrumentation library that started and stopped the loop timers. This methodology proved to be extremely intrusive since the caller-saved register allocator spilled many variables on each call to the instrumentation library. My current loop instrumentor inserts assembly instructions that start and stop the loop timers. This lightweight model allows the instruction scheduler to bundle instrumentation code with a loop's prologue and epilogue code. Furthermore, the instrumentor does not significantly affect register usage.

At all exit points in the program a call is made to an instrumentation library to store to disk the cumulative running time of each loop in the program. The information on disk then is just a list of loop identification numbers and associated cumulative running times.

To gather the label y_l for a given innermost loop l, I must determine which unroll factor $u \in \{1, 2, \ldots 8\}$ yields the best performance for the loop. Thus, I have to determine the runtime $r_{l,u}$ for each loop l using each of the eight unroll factors; then, for a given loop l, I can assign the label y_l to be the unroll factor u that produced the fastest runtime for the loop:

$$y_l = \operatorname*{argmin}_{1 \le u \le 8} \{r_{l,u}\}$$

My strategy for collecting labels finds, in bulk, the runtimes $r_{l,u}$ for a given unroll factor u. For instance, I collect all of the runtimes $r_{l,8}$ at the same time by setting the unroll factor to be 8 for all the innermost loops in a program. This strategy, while it is the easiest and most efficient data labeling approach, is clearly imperfect. For example, consider the case where there are two unrollable loops in the same method. It is likely the case — because of caching and register allocation effects — that the optimal unroll factors for each loop are dependently linked, and thus by finding labels (and making predictions) in isolation we will miss these cases. Nevertheless, my simple strategy provides good performance results.

Because noise on a real machine introduces measurement uncertainty, I assign $r_{l,u}$ to be the median of 30 trials of running loop l with unroll factor u. To further mitigate measurement noise, I only use loops that run for at least 50,000 cycles. For instance, if a loop that is only run for a few thousand cycles sits on the edge of an instruction cache boundary, a single cache miss would comprise a significant portion of the total runtime of the loop.

It is not possible to measure loop runtimes without affecting the execution in some way. Although I cannot quantitatively state the impact that the instrumentation code has on a program's execution, I am encouraged by the fact that I was able to induce highly effective heuristics with measurements gathered using this infrastructure.

6.2.4 Effort Involved

This section discusses the effort that is involved with heuristic tuning of the sort I employ in this chapter. The instrumentation of ORC — which I was completely unfamiliar with at the time — was the most demanding task, and required about two weeks of intensive work. Collecting the labels was somewhat time consuming since I ran each benchmark 30 times for each of the unroll factors, but this step was completely unsupervised and only took a little longer than a week. Finally, when I had my training dataset, I could quickly prototype several popular learning algorithms in Matlab.

The machine learning algorithms train and classify relatively quickly. The training phase requires anywhere from a few milliseconds to around 30 seconds, depending on the learning algorithm I employ. The prediction phase is extremely fast as well: the Matlab implementations required at most 5 milliseconds per prediction.

The effort that was involved is worthwhile. Now that my infrastructure is in place, quickly retuning the unrolling heuristic to match architectural changes will be trivial. I will simply have to collect a new labeled dataset, which is a fully automated process, and then I can apply the learning algorithm of my choice. Contrast this with the tedious, manual retuning efforts currently employed today. Furthermore, I am in the position to create heuristics for other loop optimizations such as loop tiling and strip mining.

6.2.5 Benchmarks Used

I extracted training examples from 72 benchmarks taken from a variety of benchmark suites. I use benchmarks from SPEC 2000,[2] SPEC '95, and SPEC '92 [89]. For SPEC benchmarks such as swim, where the application appears in two different SPEC suites, I include the newest version only. In addition, I train with Mediabench applications, benchmarks from the Perfect suite, and a handful of kernels. The training benchmarks span three languages (C, Fortran, and Fortran90). For each benchmark I only use loops that ORC can unroll

[2]Please note that I have excluded two SPEC 2000 benchmarks: ORC cannot compile 252.eon because it is a C++ program, and 191.fma3d because it does not compile correctly with my instrumentation library (it creates a different number of loops depending on the unroll factor, and thus features and labels cannot be correlated).

and whose “optimal” unroll factor is measurably better than the average ($1.05x$) over all unroll factors up to eight.

There are many different classification techniques that one could employ. The next section describes two techniques that work well for predicting loop unrolling factors (and have also been shown to work well for a wide range of problems).

6.3 Multi-Class Classification

This section describes two multi-class classification algorithms. I begin by describing near neighbor classification, a conceptually simple, but highly effective technique. I then describe support vector machines, a statistical learning algorithm that is currently a popular technique within the machine learning community.

6.3.1 Near Neighbor Classification

Chapter 4.2.2 described the nearest neighbor technique. This section describes a simple twist to nearest neighbor that worked quite well for the experiments I conduct in this chapter.

Near neighbor (NN) classification is an extremely intuitive learning technique [33].[3] The idea of the algorithm is to construct a database of all $\langle \vec{x}_i, y_i \rangle$ pairs in the training set. A label (unroll factor) can be computed for a novel example simply by inspecting the labels of the examples that lie within a fixed radius of novel example. This is a sensible approach for assigning loop unroll factors: the compiler should treat similar loops similarly. I use Euclidean distance as the similarity metric. The distance between database entry $\vec{x_i}$ and a novel loop with feature vector $\vec{x}_{novel}$ is $\|\vec{x}_{novel} - \vec{x_i}\|$. The feature vectors are normalized to weigh all features equally; otherwise, features with large values such as loop tripcount would grossly outweigh small-valued features in the distance calculation. More specifically, I normalize feature vectors so that each dimension of the vector has zero mean and unit standard deviation over all of the examples in the training set.

[3]This technique is also referred to as nearest neighbors within a radius.

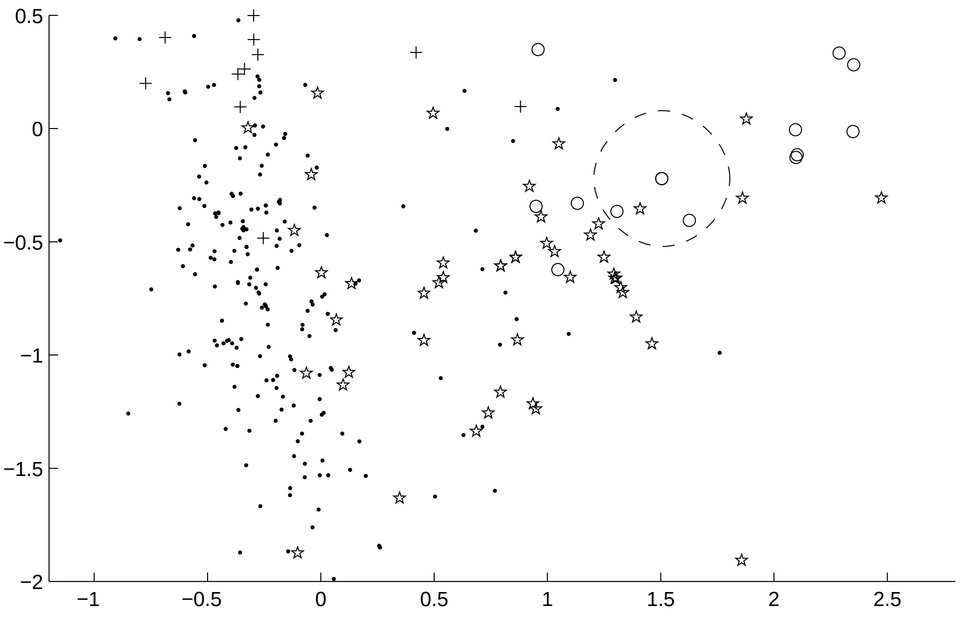


Figure 6-2: Near neighbor classification. This figure depicts the near neighbors algorithm on real unroll data. Note that this graphic is only meant to illustrate the idea of near neighbors: this figure only considers four classes, whereas the remainder of the chapter considers classification into eight classes; to further improve visualization, I only include examples where the given unroll factor is at least 30% better than the other three; I use linear discriminant analysis to find a good 2-dimensional plane on which to project the multi-dimensional data, and thus the axes are a linear combination of the higher-dimensional axes [33].

The graph in Figure 6-2 visually depicts the operation of NN on real loop data (but it is important to point out that this is only a pedagogical example). Each of the points in the figure represents a loop from my suite of benchmarks. Points represented by pluses, circles, stars, and dots correspond to unroll factors one, two, four, and eight respectively. Because there are too many dimensions in the original feature space to graphically depict (equivalent to the number of features in Table 6.1), I have reduced the dimensionality by projecting loops from the original feature space — each of which is represented by a feature vector ($\vec{x_i}$) — onto a plane[4]. Again, however, this example is only meant to highlight the operation of NN: the experiments I perform in this chapter operate in a 10-D space, as Section 6.5 explains.

The near neighbors algorithm makes predictions for a new point based on the labels of points

[4]To find a 'good' plane onto which to project the data, I use the linear discriminant analysis algorithm described in [33]. Note that the axes of the graph correspond to a linear combination of the dimensions in the original feature space.

that lie within a specified radius of the new point. For all NN experiments I use a radius of 0.3, the value of which I chose by manually inspecting the distances to training examples for about 100 random queries. In Figure 6-2, the query point centered by the dotted circle has three neighbors that lie within the specified radius. The algorithm predicts that the unroll factor for the query point is the same label as the most commonly occurring label among the near neighbors. In this case, the algorithm would predict an unroll factor of two, represented by circles in the figure.

Near neighbors can also assign a confidence to a query. If the vast majority of near neighbors share the same label, then the confidence of the query is high. Alternatively, there are cases when there is no clear winner — or even no near neighbors — which corresponds to a low confidence. In these cases, I simply assign the unroll factor based on the label of the single nearest neighbor, but more elaborate schemes are certainly possible. One can imagine a tool that automatically detects outliers by setting low confidence examples aside. An engineer could then visually inspect outlier loops to determine why they are hard to classify.

Note that NN classification is trivial to train: one simply has to populate a 'database' of examples. Though the training time of a classifier is not a paramount concern (since training the classifier is done offline), the time it takes for the resulting classifier to make predictions is important (since this task will be performed by the compiler at compile time). NN classifies a new example by performing a linear scan of the examples in the training set. For small training sets like mine, the lookup is extremely fast: with over 2,500 examples in the database, the linear-time scan takes less than 5 ms. Lookup time is far outweighed by compiler fixed-point dataflow analyses. Furthermore, advances in the area of approximate near neighbor lookup permit fast access (sublinear time in the size of the database) to databases on the order of hundreds of thousands of examples, so I expect the NN method to scale well with database size [38].

6.3.2 Support Vector Machines

A detailed description of support vector machines (SVMs) is beyond the scope of this thesis, so I only describe the high-level ideas of the algorithm here. The operation of an SVM is shown in Figure 6-3. There are two unique aspects of SVMs: first, an SVM

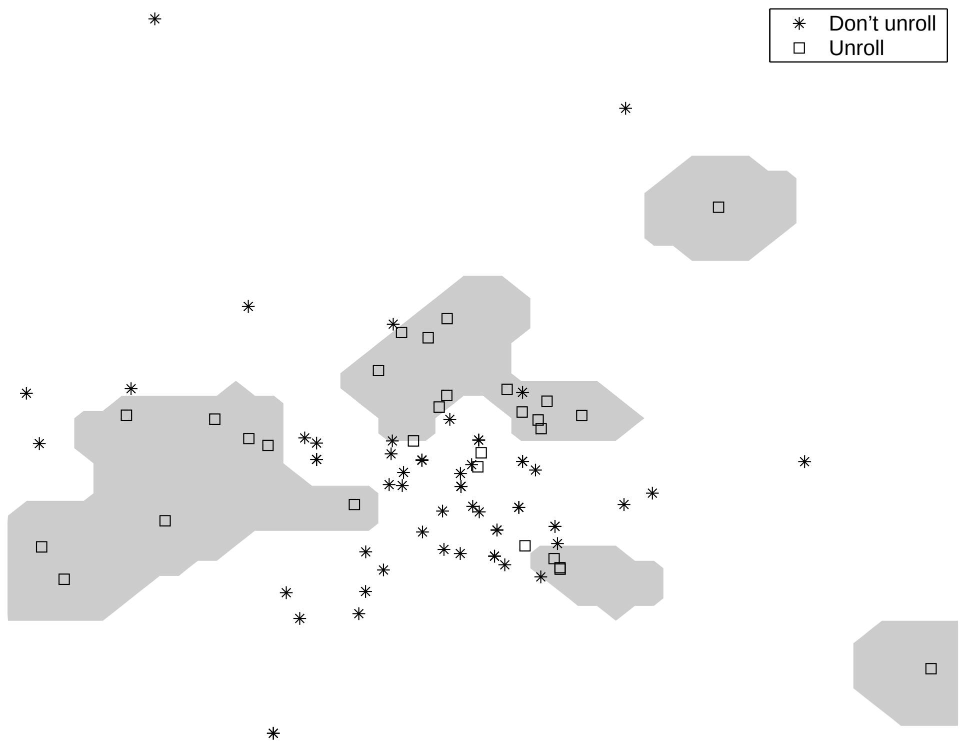


Figure 6-3: Support vector machine classification. This figure illustrates the operation of a support vector machine. To improve visualization in this example I linearly project the original high-dimensional feature space to a 2-dimensional plane; I show here a case with binary classification, and I only consider training examples where there is at least a 30% performance improvement. The SVM non-linearly maps this pedagogical 2-dimensional space into a higher dimensional space (using a radial basis kernel function [33]). The SVM then finds the linear boundary in the high dimensional space that maximally separates data from distinct classes.

maps the original D-dimensional feature space (using a non-linear function) to a higher-dimensional space where it is easier to 'separate' data, and second, in this transformed space the SVM attempts to find linear boundaries that maximally separate the classes, subject to a *regularization* parameter. The regularization parameter adjusts the complexity of the learned model, which means that an SVM does not necessarily try to minimize the errors on the training set. Proponents of SVMs claim that this prevents overfitting the training data, and thus will more likely generalize better to novel examples.

SVMs are binary classifiers, and thus some work must be done to use them in a multi-class classification context. While there are many ways to do this, one common method uses *output codes* [31]. Output codes associate a unique binary code to each label. For example,

Prediction Correctness	**NN**	**SVM**	**ORC**	**Cost**
Optimal unroll factor	62%	65%	16%	1x
Second-best unroll factor	75%	79%	37%	1.07x
Third-best unroll factor	84%	85%	58%	1.15x
Fourth-best unroll factor	90%	91%	71%	1.20x
Fifth-best unroll factor	93%	93%	87%	1.31x
Sixth-best unroll factor	96%	96%	91%	1.34x
Seventh-best unroll factor	98%	98%	96%	1.65x
Worst unroll factor	100%	100%	100%	1.77x

Table 6.2: Accuracy of predictions for the nearest neighbors algorithm, an SVM, and ORC's heuristic. This table shows the cumulative percentage of correctly made predictions by each algorithm. For instance, the SVM predicts the optimal *or* nearly-optimal unroll factor 79% of the time. The *Cost* column shows the *average* runtime penalty over all loops in the training set for mispredicting (as compared to the optimal factor).

one possible set of codewords for a three-class problem is,

class	h_1	h_2	h_3
1	1	0	0
2	0	1	0
3	0	0	1

Now, the problem has been transformed into many binary classification problems. In the case of the above example, I would train three binary classifiers, each of which would use the binary partition induced by the codewords. Thus, classifier h_1 would learn to discriminate examples in class 1 from examples in classes 2 and 3. A query's "code" representation is formed by concatenating the binary classifier predictions. The multi-class prediction is the class label corresponding to the closest codeword (in Hamming distance) to the query's code. The approach outlined above is also referred to as *one-versus-all*. Error correcting codewords can provide better results by using more bits than necessary to describe each label, but for simplicity I do not use such encodings.

SVMs take longer to train than the NN algorithm (around 30 seconds for my data), but once the classifier has been constructed, unroll factors for novel examples can be predicted quickly. For a good description of the operation of SVMs please see [33].

6.4 Experiments with Multi-Class Classification

In this section I describe the operation of a multi-class classifier for loop unrolling. More specifically, I attempt to classify loops into one of eight categories, corresponding to unroll factors one through eight. Recall that an unroll factor of one leaves the loop rolled.

As mentioned in Section 6.2, I first collect the amount of time it takes for each unroll factor to execute each unrollable loop in my suite of benchmarks. The unroll factor that requires the fewest number of cycles to execute a given loop is the label for that loop. For all experiments in this chapter I use the MASI approach for collecting data. In other words, the loop examples come from multiple applications, and each label in the training set is computed using a single static application input. I do not use the full set of 38 features that I extracted. Instead, as I will discuss in Section 6.5, I use the most "informative" ten features for classification.

I train the NN algorithm by populating the database with examples. The predicted unroll factor for a novel loop will be the most common unroll factor of the loops within a radius of 0.3. For the SVM, I obtained the C and Matlab SVM implementation distributed at [92]. The toolkit contains functions for tuning, training, and testing the accuracy of an SVM. The SVM experiments presented in this section use a radial basis kernel function [33, 92].

Table 6.2 shows the accuracy of the learning algorithms and ORC's heuristic. Using leave-one-out cross validation (LOOCV) I find that 65% of the time the SVM finds the optimal unroll factor, and 79% of the time it chooses either the optimal *or* the nearly-optimal solution. The rightmost column in the table shows the *average* cost associated with mispredicting over all loops in the training set. We can tenuously conclude from the table that 79% of the time, SVM classification is within 7% of the optimal performance (with this dataset). The NN algorithm performs slightly worse, performing optimally 62% of the time. However, this should only be looked at as an approximation of how well the algorithms would perform if used in a compiler: I filter out many noisy examples to collect these numbers, but in the end-to-end experiments I perform later, the learned compiler policy has to make predictions for *all* loops.

The histogram in Figure 6-4 shows the distribution of "optimal" unroll factors. The optimal

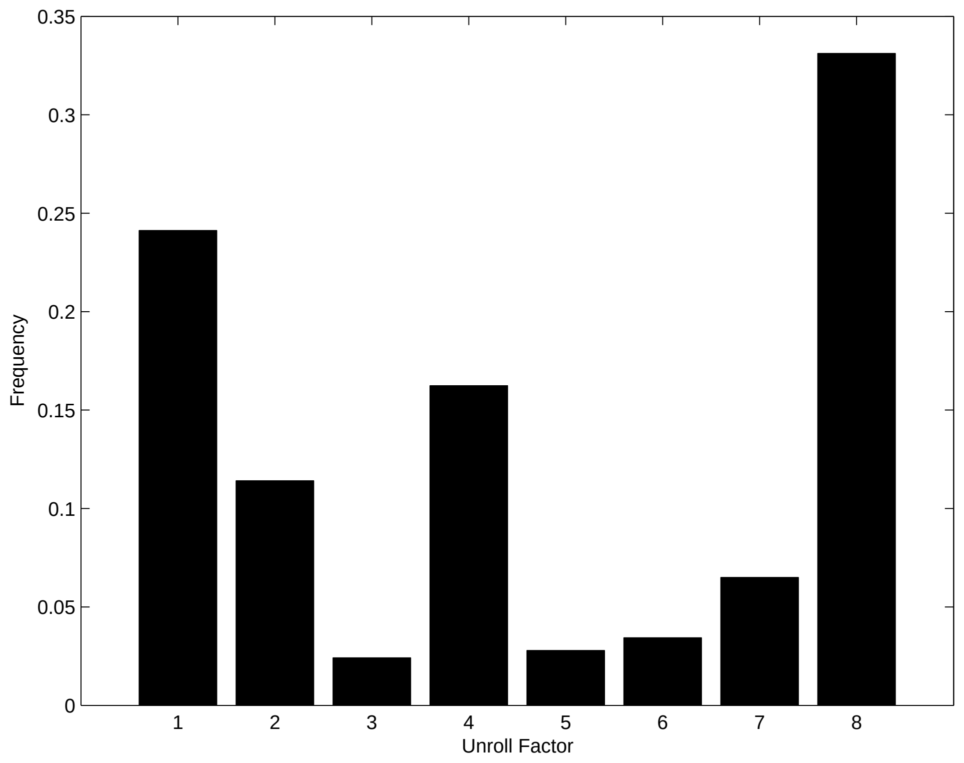


Figure 6-4: Histogram of "optimal" unroll factors. This figure shows the percentage of loops for which the given unroll factor is optimal. The histogram data was collected from over 2,500 loops (spanning several benchmark suites).

unroll factor for each loop is the class label for the loop; however, because my data collection technique is imperfect, the class label does not always correspond to the optimal unroll factor. An interesting observation is that non-power-of-two unroll factors are rarely optimal for this dataset. The figure also indicates that no one loop unrolling factor is dominantly better than the others.

6.4.1 Realizing Speedups

In this section I see if improved unrolling classification accuracy yields program speedups. For these experiments, I compile the SPEC 2000 benchmarks using the learned classifiers to predict an unroll factor for each loop. Note that I train the learning algorithms using the MASI approach. In other words, the corpus of training examples comes from a suite of 72 applications, each of which processed a static input file. Then, I determine the generalization ability of the policies using an approach that is similar to LOOCV: for each benchmark in

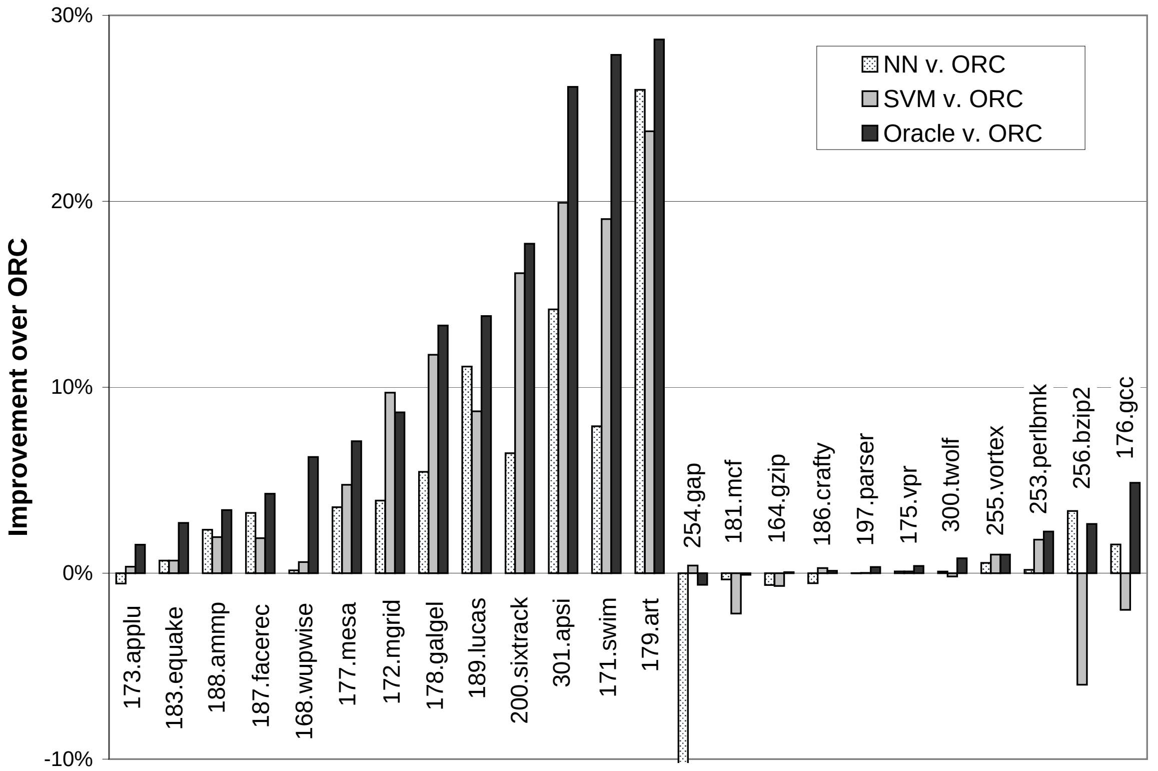


Figure 6-5: Realized performance on the SPEC 2000 benchmarks. Both NN and an SVM achieve speedups on 19 of the 24 benchmarks. The SVM achieves a 5% speedup overall, and it boosts the performance of all SPECfp benchmarks, leading to a 9% overall improvement. Near neighbors performs slightly worse, boosting the performance by about 4%. The rightmost bar shows the speedup that an "oracle" would attain (7.2% average).

the SPEC 2000 suite, I gauge how well a heuristic induced by training on the examples from the remaining 71 benchmarks in the training set performs on it. This tells us how well the learned compiler policy performs on loops that it has not seen before. In addition, the learned classifier makes predictions for *all* unrollable loops when compiling a benchmark. I do not instrument the compiled code for the experiments in this section. Instead I use the UNIX `time` command and the median of three trials to measure whole-program runtimes.

Figure 6-5 shows the performance improvement of my method over ORC's unrolling heuristic. The figure also shows the speedup that the compiler could obtain if an "oracle" were to make its unrolling decisions. The oracle here simply refers to a classifier that uses the extracted training labels to make predictions. Thus, the oracle makes perfect predictions according to the training data set. The SVM achieves a speedup on 19 of the 24 SPEC benchmarks. Overall my technique attains a 5% average speedup on the SPECs, and a 9%

speedup when only the SPECfp benchmarks are considered.

The benchmarks on the left hand side of the graph come from the SPECfp benchmark set. These applications are highly amenable to loop optimizations, and thus, meta optimization substantially increases the performance of the loop unroller. The benchmarks on the right hand side of the graph are from the SPECint application set. The loops in these benchmarks contain abundant control flow which hinders the effectiveness of the loop unrolling optimization. For these benchmarks, meta optimization is slightly outperformed on average by ORC's human-constructed constructed heuristic (by about one half percent), and the oracle only achieves a 1% speedup overall.

Note that the oracle is slightly outperformed in a couple of cases because my data collection methodology is necessarily imperfect. In addition to working within a generally noisy environment, I assume that the optimal unroll factor of a particular loop does not depend on the unroll factors of the other loops. While this assumption may not be entirely correct, it simplifies the data collection process (I can collect the runtimes for all loops given a particular unroll factor in the same run). The overall performance of the oracle legitimizes my assumptions.

6.5 Feature Selection

This section focuses on finding the most informative features for discriminating unroll factors. I take two approaches to feature selection in this section. The first method uses information theory to score the information content of a feature, and the second method greedily chooses features that match a given classifier for a given training set. For both approaches I let f_i represent an individual feature from the set of 38 features — 22 of which Figure 6.1 lists — I extract for each loop $F = [f_1 \dots f_{38}]$.

6.5.1 Mutual Information Score

The *mutual information score* (MIS) measures the reduction in uncertainly in one variable (*e.g.*, a particular feature f_i) given information about another variable (*e.g.*, the best unroll

Rank	Feature	MIS
1	# floating point operations	0.19
2	# operands	0.186
3	instruction fan-in in DAG	0.175
4	live range size	0.16
5	# memory operations	0.148

Table 6.3: The best five features according to MIS.

Rank	Nearest neighbor	Error
1	# operands	0.48
2	live range size	0.06
3	critical path length	0.03
4	# operations	0.02
5	known tripcount	0.02

Table 6.4: The top five features chosen by greedy feature selection for the nearest neighbor algorithm. The error numbers reported here are for the training set, hence the low error rates for these classifiers.

Rank	SVM	Error
1	# floating point operations	0.59
2	loop nest level	0.49
3	# operands	0.34
4	# branches	0.20
5	# memory operations	0.13

Table 6.5: The top five features chosen by greedy feature selection for the SVM algorithm. The error numbers reported here are for the training set, hence the low error rates for these classifiers.

factor u) [33]. The MIS adapted for the loop unrolling problem is given by,

$$I(f_i) = \sum_{\phi \in V_{f_i}} \sum_{y \in \{1 \ldots 8\}} P(\phi, y) \cdot \log_2 \left(\frac{P(\phi, y)}{P(y) \cdot P(\phi)} \right)$$

where V_{f_i} represents the set of values that f_i can assume. I use the MIS to determine the extent to which knowing the value of a loop characteristic reduces the uncertainty about the desired loop unroll factor. Informative features receive higher scores than uninformative features. I bin the values of continuous features before estimating the probability mass functions used to compute the MIS.

Table 6.3 shows the five features with the highest MIS.

6.5.2 Greedy Feature Selection

There are several problems with MIS, most notably that it does not tell us anything about how features interact with each other. In addition, even though the score is a metric for information content, it does not guarantee that the features will be useful for a particular classifier.

Greedy feature selection identifies features that perform well for a given classifier and a given training dataset. Given a feature set, $F = \{f_1 \ldots f_D\}$, where D is the feature dimension, the simple algorithm starts by choosing the *single* best feature, $b_1 \in F$, for discriminating the training dataset (using a particular classifier). The algorithm proceeds by choosing a second feature, $b_2 \in F$, that together with b_1, best discriminates the training dataset. In each iteration of the algorithm a new feature is chosen that minimizes the training error given the features that have already been selected. The algorithm halts after a user-defined number of features have been selected.

Tables 6.4 and 6.5 show the five best features for my dataset according to greedy feature selection using the nearest neighbor and SVM algorithms, respectively. Notice that the choice of classifier affects the list of features deemed to be the most informative. Also note that the ideal setting of the radius for the NN algorithm depends on the number of features I include; for instance, the 0.3 radius setting would not perform well if only a single feature was considered. For this reason I use the nearest neighbor algorithm — which bases predictions on the label of the single nearest neighbor to a query — instead of NN.

I used the ten features that correspond to the union of characteristics in Tables 6.3–6.5 to perform the classification experiments presented in Section 6.4. Whenever possible, it is preferable to use a small number of features when training a classifier. Uninformative features can ‘confuse’ a learning algorithm or lead to overfitting of the training data. In addition, learning algorithms are generally more efficient when shorter feature vectors are used.

Notice that the number of instructions in the loop body appears only once in Tables 6.3–6.5, and relatively far down the list. I highlight this fact because this feature is the de facto standard when discussing unrolling heuristics [75]. According to both feature selection methods applied above, there are many other features that are more useful for discriminating unroll factors. The features that are listed in this section are not entirely surprising; arguments could be made to support their predictive values. It would be more difficult however, to determine why the features found by greedy selection are *jointly* informative.

6.6 Conclusion

Compilers rely on models to make informed decisions. While humans can generate highly effective models, the number of person hours required to create them may be prohibitive. This chapter presented novel research on the automatic creation of compiler policies using supervised machine learning techniques. I used empirical evidence to teach two simple machine learning algorithms how to make informed loop unrolling decisions.

The learned classifiers predict loop unrolling factors with good precision. Using LOOCV to find the generalization ability of the classifier, the algorithm is able to predict the optimal unroll factor for a given loop 65% of the time. Furthermore, it predicts the optimal or the nearly optimal solution 79% of the time. I translate these results into speedups on a real machine. Using the Open Research Compiler and targeting the Itanium 2 architecture, I find that the learning algorithms improve the performance of 19 of the 24 benchmarks in the SPEC 2000 benchmark set. When I focus solely on loop unrolling, the technique achieves a 5% improvement on the SPEC benchmarks, while improving the SPECfp benchmarks by 9%.

One argument against using machine learning is that the compiler writer must extract the features with which the learning algorithms are trained. However, most of the features that I collected were readily available in the ORC infrastructure; and as with the experiments I presented in the last chapter, I thought little about which features to include, and I extracted any feature that I thought *might* be pertinent to discriminating loop unrolling factors. Because supervised learning operates offline on a corpus of training examples, it is possible to thoroughly analyze the information content of the features. I used two forms of feature selection to identify the most salient features for predicting unroll factors.

Another potential argument against using machine learning to optimize policies is that extracting features and labels takes time. By far the hardest part of this research was writing the loop instrumentor. If meta optimization techniques become successful, I would expect compiler infrastructures to export generalized timers to support supervised learning. With the timers in place, only patience is required to create a training data set. While the actual training of the classifiers takes less than a few seconds, it does take time to acquire the labels. However, collecting the labels is a completely unsupervised process, and requires far less effort than constructing an unrolling heuristic by hand. Unlike the GP policy search technique I presented in the last chapter, I spent little time tweaking the machine learning algorithms. Therefore, while the performance results are satisfying, I also excited about the complexity ramifications of my research.

Perhaps the biggest drawback of the multi-class approach I use in this chapter is how its predictions are confined to the limits of the labels with which the supervised learning algorithm is trained (*e.g.*, my learned classifiers will never predict unroll factors greater than eight). While this limitation may prevent the technique from being used for some compiler optimizations, the vast majority of optimizations already have imposed constraints. For my experiments, I set my limits to the greatest unroll factor for which all of the loops in my training set compile correctly.

In the previous chapter I had the benefit of working within a noiseless simulator. This chapter shows that, with care, learning techniques can apply in noisy environments. To combat noise, I assign class labels by using the median of 30 trials for each loop at each unroll factor. As with the experiments in the previous chapter, the loop unrolling policy is trained at the factory, and thus the machine-learned policy incurs little compile time overhead. The next chapter describes a technique that enables ordinary users to contribute to the data collection process in a self-improving, deployed system.

Chapter 7

Collaborative Data Collection

In order to adequately train compiler policies using machine learning, we have to collect a large amount of training examples, and these examples have to be drawn from a diverse set of applications. Furthermore, as we have seen, applications that we include in the training set tend to perform better than novel applications. Thus, when we perform at-the-factory training, the compiler vendor essentially has to guess the applications in which its clientele is interested. This chapter introduces an approach that mitigates the stringent data requirements of machine learning. The next section provides a high-level view of my approach, which I call *collaborative compilation*, and Section 7.2 provides implementation details.[1] Section 7.3 presents two proof-of-concept experiments that I perform on a prototype collaborative compiler. Section 7.4 discusses privacy concerns, and Section 7.5 concludes.

7.1 Collaborative Compilation

Languages such as Java and C# are popularizing a new paradigm of compilation. Under a traditional processing approach, the software developer ships a pre-compiled binary that is ready for native execution on the user's machine. Even though the application may be used multiple times by millions of people, it is only compiled *once*. Modern-day interpreted languages run within virtual machines (VMs) which actually feature just-in-time (JIT)

[1] I performed this work jointly with Ben Wagner. Please see his Master's thesis for a more complete description of collaborative compilation [95].

compilers that quickly generate native code on demand. Thus, *every time* a user runs an application, the compiler is invoked.

Because the just-in-time (JIT) compiler in a modern-day virtual machine (VM) performs compilation at runtime, it cannot afford to thoroughly analyze program behavior and evaluate multiple compilation options. The speed-of-compilation constraint makes JITs increasingly reliant on effective and efficient heuristics. We can harness the frequent compiler invocations associated with VM processing to facilitate the machine learning data collection task.

As more data is added to a machine learning training set, the ability of a learning algorithm to generalize typically increases. In addition to the quantity of data used, the quality of the data is also extremely important. Previous attempts to learn compiler heuristics extracted training data from standardized benchmark suites such as SPEC [89]. Computer benchmarking has always been controversial because it is difficult for a handful of benchmarks to simulate a "real world" workload. There is no reason to believe that a compiler that is tuned to perform well on standardized benchmarks will perform well on applications that a common user is interested in running.

Of course, we could ask compiler users to endure a time-consuming training process, and to assemble a representative training set of applications and input data sets. Using this approach, application developers could customize their compilers to perform well on the applications and systems for which they are developing. Serious software vendors may choose to employ meta optimization techniques, but casual programmers will probably not subject themselves to the additional work required to train a compiler. Typical computer users — who may not even know what a compiler is — will be even less inclined to optimize the performance of their favorite applications. A typical user will not use meta optimization techniques unless the process is completely transparent, non-disruptive, timely, and effective. I propose *collaborative compilation*, which satisfies these requirements and enables typical users to automatically apply meta optimization.

Collaborative compilation asks regular users to contribute to the machine learning data collection task. A collaborative compiler, which I refer to as a client, *may* insert low-overhead ($\sim$1%) instrumentation code that collects data that is suitable for training a

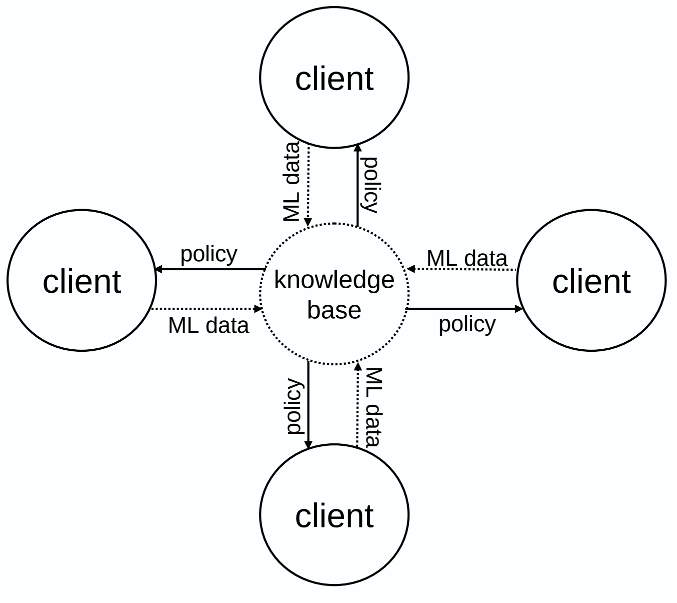


Figure 7-1: Collaborative compilation. The compiler *may* instrument a small fraction of the generated code to collect data that is used to train a learning algorithm. Users accept a small runtime overhead as a fair cost for access to the community "knowledge base". An offline learning algorithm creates a policy using data that is contributed to the community knowledge base by collaborative users. Because collaborative users are essentially training a learning algorithm to work well for the applications and data sets they care about, they will reap the rewards of contributing to the knowledge base.

machine learning algorithm. The instrumentation code measures the effectiveness of a given optimization throughout the execution of the VM. Before the VM exits, it contributes the data that it collected to a centralized *knowledge base*. Collaborative users will likely receive good karma from agreeing to contribute their data: these users are training the learning algorithm to perform well on code in which they are interested. I show in this chapter that the reward that a collaborative user reaps may well be worth the small instrumentation overhead.

Figure 7-1 shows a high-level view of a collaborative compilation. When a collaborative user runs an application, the client VM will fetch the best-so-far compiler heuristics — which may be cached locally — and use them to make compilation decisions. At the same time the client may choose to instrument some of the methods in the user's application. When the client VM exits, it sends any data it collected to a central repository, where it is incorporated into a growing training set. Thus, users of a collaborative compiler will accept a small runtime overhead as a fair cost for access to the community knowledge base.

Communities can range in size from millions of users to a single user taking advantage of the repeated use of a working set of programs. Perhaps specific collaborative communities could be established for users that share common processing interests (*e.g.*, a scientific computing community). In large communities, users have access to a much larger corpus of training data than they could generate on their own.

A successful collaborative compiler must have the following attributes:

- Transparent: users will not endure a separate training phase. Learning must be performed online and incrementally. A collaborative system transparently trains the compiler while it runs applications.

- Low overhead: people will not use this system if the instrumentation overhead negates the benefits of access to the community knowledge base. Collaborative systems use two techniques to reduce the overhead so that it is unnoticeable.

- Timely: users will not tolerate a long training process. Collaborative systems, which have access to a vast community corpus of training data, can quickly reap performance benefits.

- Accurate: the goal is to isolate and measure the effect of the proposed heuristic. In other words, there must be a mechanism for discarding noisy measurements. In addition, runtime measurements must be independent of system workload. Collaborative systems employ two techniques to identify and filter noisy measurements.

- Effective: users will only be inclined to use collaborative systems if doing so improves the performance of their applications. Proof-of-concept experiments show that these techniques can lead to substantial performance improvements.

This chapter evaluates a prototype collaborative compiler that successfully tackles these issues. The next section describes the instrumentation required to collect machine learning training data.

7.2 Collaborative Instrumentation

The high-level goal of collaborative compilation is to enable ordinary users to transparently train their JIT compilers to perform well on the applications and systems in which they are interested. The major component of collaborative compilation focuses on extracting empirical performance data that is suitable for training machine learning techniques. The data requirement varies depending on whether the compiler uses a policy search approach or a supervised learning approach.

Consider the supervised learning approach that the last chapter discussed. For each loop in an application I extract a feature vector and an empirically found label. The label indicates which unroll factor is the best for the corresponding loop. We can collaboratively extract this information using a JIT: extracting features is still easy, but now we have to devise a strategy to determine the the loop's label online, without adversely affecting the user.

I will build on the (somewhat naive) collaborative approach to on-the-fly data gathering shown in Figure 7-2. The client first randomly chooses a method to instrument, in this case `addScalar` (shown in part (a)). After the client extracts a feature vector to describe the loop in the method — which could include system information to distinguish different processing platforms — it creates two new versions of `addScalar`: `addScalar'` and `addScalar''`, which are shown in parts (c) and (d), respectively. The loop in `addScalar''` is unrolled, whereas the same loop is left intact in `addScalar'`. Figure 7-2 (b) reveals that the instrumentor wraps timers around the calls to `addScalar'` and `addScalar''` in the original method. In addition, the original method is updated to randomly choose between executing one of the two newly-created versions, or executing the original code. I use the sampling technique described by Liblit et al. in [63] to reduce the overhead of choosing which version to execute.

With this instrumentation in place, the client can then determine, for example, whether unrolling is beneficial for a given loop by *repeatedly sampling* these new versions, `addScalar'` and `addScalar''`. Sampling, in this context, refers to measuring the runtime of a single invocation of one of the newly-created methods. The instrumentor assigns each instrumented method a set of storage cells, which I call the "accumulator array", to keep track of runtime

```
public void addScalar(char* a, int c, int n) {
   for(int i = 0; i < n; i++) {
      a[i] = a[i] + c;
   }
}
```

(a) Original version.

```
public void addScalar(char* a, int c, int n) {
   if(shouldWeSample()) {
     if(pickVersion()) {
       start_timer;
       addScalar'(a, c, n);
       stop_and_accumulate_timer;
     } else {
       start_timer;
       addScalar''(a, c, n);
       stop_and_accumulate_timer;
     }
     return;
   }

   for(int i = 0; i < n; i++) {
      a[i] = a[i] + c;
   }
}
```

(b) Original version, sampling added.

```
public void addScalar'(char* a, int c, int n) {
   for(int i = 0; i < n; i++) {
      a[i] = a[i] + c;
   }
}
```

(c) Rolled version.

```
public void addScalar''(char* a, int c, int n) {
   int i = 0;
   for(; i < n % 4; i++) {
      a[i] = a[i] + c;
   }

   for(; i < n; i+=4) {
      a[i] = a[i] + c;
      a[i+1] = a[i+1] + c;
      a[i+2] = a[i+2] + c;
      a[i+3] = a[i+3] + c;
   }
}
```

(d) Unrolled version.

Figure 7-2: (Naive) Instrumentation by a collaborative compiler. The client randomly selects — according to a user-defined rate — a small subset of methods to instrument. When a method is selected, the system creates three versions of the method. First, as shown in (b), the system inserts code in the original method which will occasionally sample from two newly-created methods, which are shown in (c) and (d). The methods in (c) and (d) are optimized in two different ways, and both are timed to note runtime disparities between the two optimization approaches.

statistics.

At runtime, the inserted instrumentation code measures the total elapsed time of each method invocation and stores the value in the accumulator array, which I show in Figure 7-3(a). The square of the elapsed time, t^2 is also accumulated and a method-specific sample counter N is incremented. With this information, a client can compute the mean and standard deviation of a method's runtime samples.

If there are enough samples to determine that one version is statistically better than the other, the client will create a machine learning label for this loop. The client assumes that one version is clearly better if there is at least a 3% performance difference between the two versions. In the prototype compiler, the label takes the form of a runtime ratio that indicates which of the two optimized versions was the best.

Before exiting, the client computes the average runtime for both versions. Based on runtime averages it will compute a label that indicates which version was the fastest. The feature vector(s) and label(s) collected during execution are then sent to the knowledge base. A machine learning algorithm uses the aggregated data to induce a classifier for predicting when to apply the optimization.

7.2.1 Coping with Noise

For really large communities, the naive approach may be sufficient. The law of large numbers would allow the knowledge base to accurately estimate the relative benefits of one heuristic over another.[2] For instance, for reinforcement learning approaches, the knowledge base could accumulate samples until it has a high confidence in the relative benefit of one heuristic over another [84]; and for supervised learning, the knowledge base could aggregate all training examples that share the same feature vector, and statistically compute the most common label among them.

However, for small communities, the general approach described above suffers from one primary problem: system noise. In order to be successful for small communities, the system

[2]This assumes that the distribution of noise has zero mean and is symmetric, which the experiments in this chapter indicate.

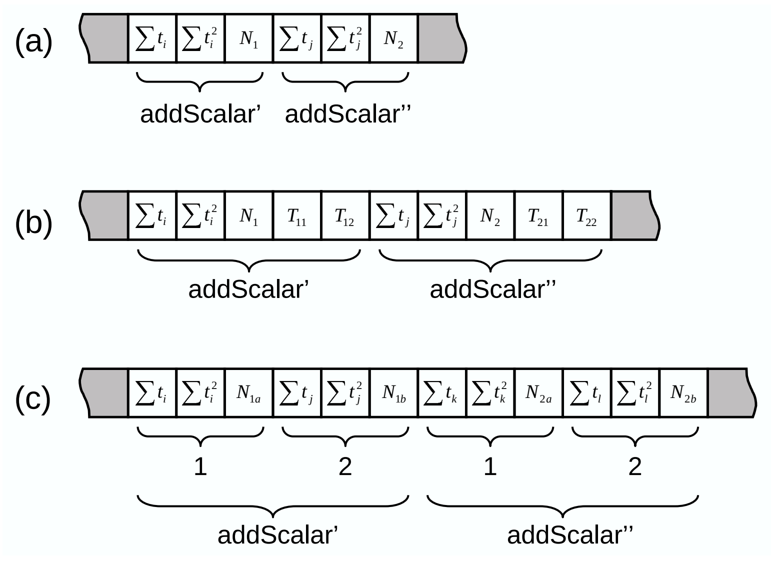

(a)
$\sum t_i$ $\sum t_i^2$ N_1 $\sum t_j$ $\sum t_j^2$ N_2
addScalar'
addScalar''
(b)
$\sum t_i$ $\sum t_i^2$ N_1 T_{11} T_{12} $\sum t_j$ $\sum t_j^2$ N_2 T_{21} T_{22}
addScalar'
addScalar''
(c)
$\sum t_i$ $\sum t_i^2$ N_{1a} $\sum t_j$ $\sum t_j^2$ N_{1b} $\sum t_k$ $\sum t_k^2$ N_{2a} $\sum t_l$ $\sum t_l^2$ N_{2b}
1
2
1
2
addScalar'
addScalar''

Figure 7-3: Various accumulator arrays. The accumulator array for the example in Figure 7-2 under various modes of operation. In its most basic mode, the system accumulates the sum, the sum of squares, and the total number of samples for each version (a). With this information it is possible to compute the mean and variance of the method runtimes. Part (b) shows the form of the accumulator array for median filtering (median of three). Part (c) shows the array when using redundant versions. Not shown is a simple extension of (b) and (c) that combines median filtering and redundant versions.

must be able to isolate the impact of a heuristic, even if the heuristic only slightly affects performance. Needless to say, it would be difficult to learn a heuristic if its effect were less than the noise tolerance. My target for noise tolerance is 3%, which is more ambitious than the 5% tolerance that I, along with other researchers have used in previous work [72, 90]. In other words, if one version is 3% better (or worse) than another version, the system recognizes this as an informative machine learning training example.

Java virtual machines (JVMs) are complicated systems. At any given time a JVM is juggling several threads. A thread switch during a timed sample may completely skew the runtime average of a method's samples. Furthermore, methods can be swapped out on the fly with better-optimized versions that can lead to big boosts (or dips) in performance. The garbage collector can also drastically change the runtime performance of a method. A copying garbage collector, for instance, may change the alignment of code arrays, which can cause sizable performance changes. Even for identical code copies, different addresses can yield disparate branch predictor and instruction cache performance [50].

In addition to virtual machine noise, a client also has to deal with operating system (OS) and computer architecture noise. It is not uncommon for any number of hardware or software faults to occur during the execution of an application. OS context switches and page faults, for instance, can add significant amounts of noise to runtime samples.

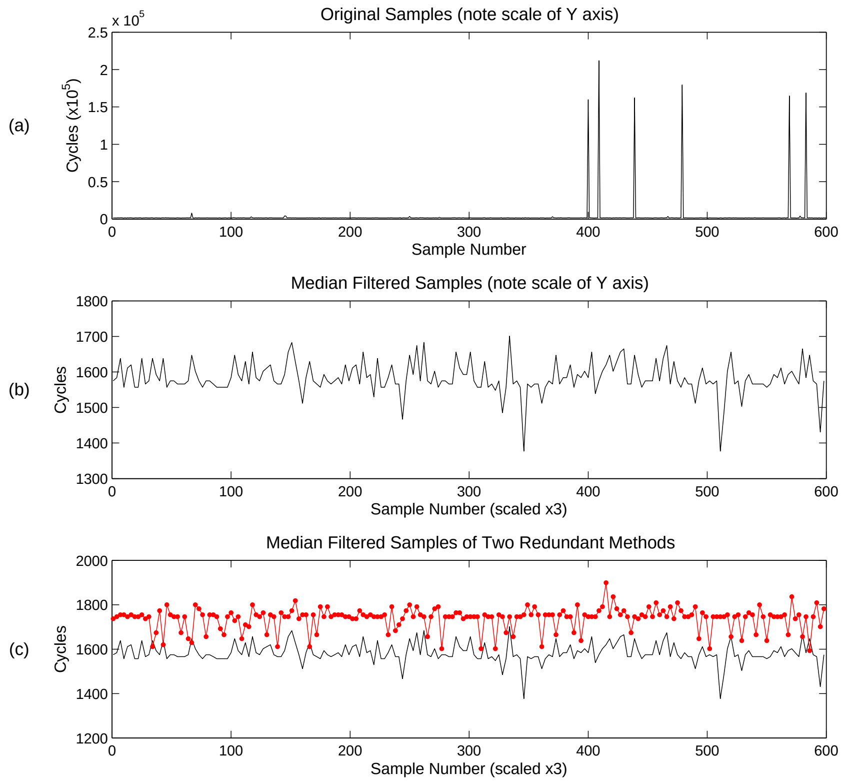


Figure 7-4: A typical series of method-level samples. These charts plot the runtime (in CPU cycles) of invocations of a typical method over time. The chart in part (a) does not use median filtering, and thus we can see several erroneous spikes in runtime. Such transient measurements can skew the runtime average since they are often orders of magnitude higher than average. The data in part (b) shows a median-filtered version of the graph in part (a) (note the scale change of the Y-axis). Finally, the graph in part (c) shows how the mean runtime of two identical methods can differ solely because of code placement. A client should reject the samples in (c) because of the large performance difference between the two identical copies.

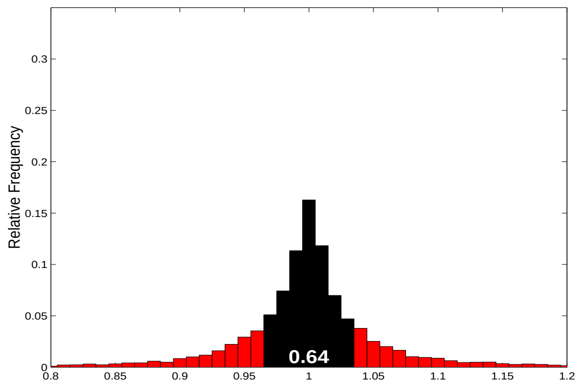
Figure 7-4 (a) plots the runtime samples of a method that does a constant amount of work per invocation. Any given sample corresponds to an invocation of the method. Each invocation has a corresponding runtime, shown on the Y-axis, which describes how many CPU cycles it took to execute the method in that invocation. The raw data in the top chart has several transient spikes in runtime. Because these surges are often orders of magnitude greater than the average sample times, they can significantly affect the computed mean and variance of the runtime.

It is simply not possible to eliminate noise in the virtual machine. Identifying and discarding noisy samples is a much better approach. Collaborative systems employ two online techniques that effectively filter noisy samples: median filtering, and creating redundant methods. As Figure 7-4(b) shows, median-of-three filtering nicely discards infrequent spikes in method runtimes (note the scale of the Y-axis). The accumulator array for median filtered collaborative compilation is shown in Figure 7-3(b). In this scenario samples are not immediately accumulated; when three samples have been collected the system identifies and accumulates the median value.

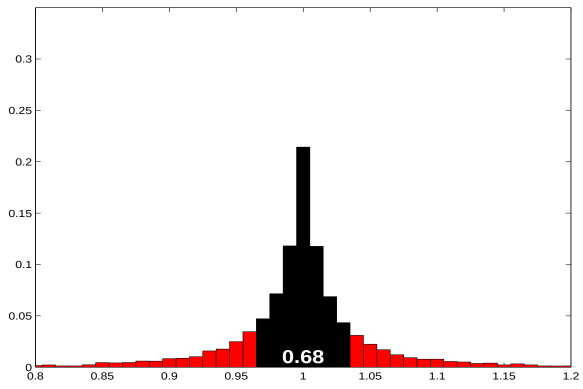
While median filtering effectively eliminates transient spikes in runtime, it will not help identify noise due to differences in memory alignment. Figure 7-4(c) shows that the performance of two *identical* methods can differ solely because of their placement in memory. For instance, code alignment can affect instruction caching behavior and the effectiveness of the branch predictor [50].

In addition, the reader may wonder how workloads that vary with invocations (*e.g.*, have drastic phase shifts [88]) will affect collaborative compilation. For example, an invocation of a sorting method that sorts an array of length 10 will have a much smaller runtime than an invocation of the method with an array of length 10,000. A collaborative system might believe that one version of a method is superior to another only because it was repeatedly tested on abnormally small workloads. To detect such cases, a client creates redundant versions of a method. The only difference between copies of a version is where they are placed in memory. The system uses these duplicate copies to gauge the level of noise for a set of samples. For instance, the client can discard samples where there is a large average runtime discrepancy between the two identical copies. Figure 7-3(c) depicts the accumulator array for the redundant methods.

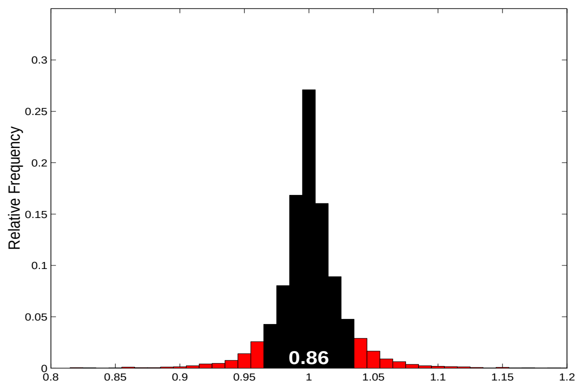
While the aforementioned safeguards cannot completely eliminate noisy examples, they do have an immediate impact as seen in Figure 7-5. The histogram empirically demonstrates that a client, with median filtering and redundant methods, can effectively reduce measure-



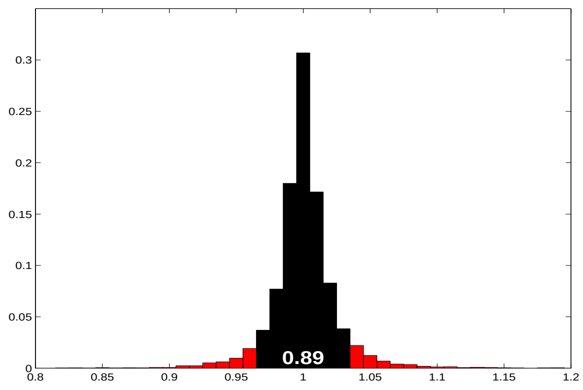
(a) No filtering.



(b) Median filtering.



(c) Redundant methods.



(d) Redundant methods & median filtering.

Figure 7-5: Checking the accuracy of collaborative training labels. This figure shows histograms of runtime ratios obtained when 10,000 pairs of identical methods (from the DaCapo benchmark suite) are compared. For any given trial, because the methods are identical, we expect the runtime ratio to be 1. The histogram in (a) uses no filtering, and thus, only 64% of the labels are within 3% of 1. Using median filtering only, as in part (b), 68% of the labels are within 3%. Filtering with redundant methods vastly reduces noise: part (c) shows results when using redundant methods only, and part (d) shows that 89% of the labels lie within 3% tolerance when using both redundant methods and median filtering. This figure highlights the fact that, although clients cannot eliminate system noise, they can effectively filter noisy labels. The load on each machine varied unpredictably during the experiment.

ment error. The histogram plots the labels of training examples — essentially runtime ratios — obtained when comparing thousands of *identical* methods from the DaCapo benchmark suite [28]. For this experiment a client randomly chooses methods to instrument. It then creates two identical versions and measures the mean runtime for each throughout the execution of the benchmark to which it belongs. It finally computes a runtime ratio by dividing the mean of one method by the mean of the other method. Notice that most of the ratios are tightly clustered around 1, which is the desired value for this experiment.

The figure is especially compelling because the labels come from Pentium III and Pentium 4 machines that are shared by several research groups, and the load on each machine varied unpredictably (from unloaded to completely loaded) during the experiment. The data in Figure 7-5(a) shows that no filtering of the data yields poor results. Only 64% of the data are clustered within 3% of 1. This could be due to variations in calling context, differences in memory alignment, VM noise, OS noise, or miscellaneous architectural noise. Part (b) shows that median filtering can provide modest improvements in accuracy. The use of redundant methods is by far the most effective technique for reducing noise, as part (c) indicates. Finally, by combining median filtering and redundant methods, a client ensures that 89% of the runtime ratios are within ±3% of 1. Previous machine learning researchers have successfully used a threshold of at least 5% — which contains 96% of the labels in this experiment — to gauge whether one version is better than another [72, 90] (See Section 6.2.5).

There are two components to a VM's total runtime: compilation time, and method runtime. When a long runtime is expected the user may wish to instruct the VM to aggressively compile an application (*e.g.*, the `-server` option on Sun's virtual machine). In other words, for long running programs, it is more appropriate to minimize the "steady-state" runtime at the expense of compile time. The timers discussed above only measure method runtimes, and therefore, they are only appropriate for learning to improve steady state performance.

However, many users are interested in quick-running programs where compile time must be minimized. My infrastructure provides a mechanism for computing the *total* runtime of a method, which includes the time spent in compilation. Thus, in addition to the label, a client can also report the compilation rate of the optimization in bytecodes per cycle. In contrast to collecting runtime samples, a collaborative client can only afford to collect a small number of compile time samples per instrumented method (because compilation is very expensive). Thus, the knowledge base will not use this information until it has aggregated enough compile time samples to make accurate predictions of the means.

Rate	Total
0.01	1%
0.05	8%
0.10	13%
0.20	24%
0.10 & capped at 1	2%

Table 7.1: Runtime overhead of collaborative compilation averaged over 12 complete runs of the specjvm98 benchmark suite [89]. The rate column shows the rate at which methods under compilation are selected to be collaboratively compiled. The total column contains the total overhead of the collaborative system, including the cost of compiling redundant methods, and of sending any gathered data to a collaborative server. The last row shows the overhead when at most one method is instrumented per run. For large enough communities, the instrumentation rate could be set such that there would be virtually no overhead.

7.2.2 Implementation Details

I have extended JikesRVM, which is an aggressive open source virtual machine that features a JIT compiler [6]. The prototype client can instrument multiple methods per run of an application. I reduce the amount of bookkeeping that the system must perform by allowing only one thread and one method invocation to sample at a given time. Thus, if the system were sampling a recursive method, subsequent recursive calls to that method would not trigger the timer. The system times all computation that a method performs, including the time spent in any methods it might call.

7.2.3 Runtime Overhead

This section discusses the overhead associated with collaborative compilation. Table 7.1 lists the average overhead of the system over 12 complete runs of the specjvm98 benchmark set [89]. I computed the overhead numbers using the user time reported by the `time` command. These numbers include the cost of compiling four additional methods for each collaboratively instrumented method and connecting to the knowledge base over the network.

Later I discuss an experiment where the prototype compiler automatically tests the efficacy of two key parameters for tuning the method inliner in Jikes RVM. The effectiveness of these settings is highly dependent on which method is being compiled. As a result, a client will sometimes try a setting that reduces the performance of the instrumented method. Because the client will infrequently sample the instrumented methods, however, the overall performance is hardly affected. The numbers in Table 7.1 account for occasionally making bad decisions.

For all the experiments I perform in this chapter, I set the instrumentation rate at 10% and cap the number of instrumented methods at one. In other words, with these settings, the system will never instrument more than one method per run. This approach also eliminates the possibility of instrumenting two different methods that are linked to each other in the program call graph (*i.e.*, one method either directly or indirectly calls another method). The runtime overhead of using this option is small, as the last row of Table 7.1 indicates.

In the traditional compile, profile, feedback cycle, gathering data is an expensive proposition. These feedback-directed systems were designed to extract as much data as possible during the profiling phase. In contrast, a JIT is invoked every time a user runs an application on a VM. Thus, unlike a traditional feedback-directed system, a collaborative compiler does not have to gather data in wholesale. Instead, a client can afford to collect a small amount of information per invocation of the JIT. For large communities, a client can make do with instrumenting a negligible number of methods per VM invocation, which will drive the client's overhead toward zero.

7.3 Case Study: Improving Steady-State Performance

This section serves as a proof-of-concept validation of collaborative compilation. The experiment I present here investigates improving the steady-state performance of the DaCapo benchmark suite [28], and uses the SASI approach to application specialization.

Some users are more concerned with steady-state performance than they are about compilation overhead. This section describes a simple experiment with the method inliner in Jikes RVM. Method inlining can have a large effect on the resulting performance of a Java application. Small accessor methods are commonly used in Java, which can lead to relatively high calling overheads for small amounts of computation. Inlining of such methods can lead to substantial performance gains. However, if the compiler overzealously inlines method calls, instruction bloat and register pressure could degrade overall performance. For instance, my system found that inlining the calls of the primary method (`TrapezoidIntegrate`) of `series` in the Java Grande benchmark suite [49] leads to a 27% slowdown.

There are two key inlining parameters that adjust the aggressiveness of the JikesRVM inliner. One parameter sets the maximum inline depth, and the other restricts the size of methods the inliner will consider for inlining. In this experiment I used the system to compare the relative benefits of three options: using the default settings, doubling the values of both parameters, and halving the values of both parameters.

When the client selects a random method for instrumentation, it chooses to compare random pairs of these three options. At the completion of a collaborative execution, the VM sends the contents of the accumulator array to the centralized server. The centralized server catalogs and maintains a database of training examples. The server then constructs an application-specific database of the best per-method, inlining parameter settings. For each benchmark, I collect around 600 training examples, which roughly corresponds to running the application (over and over) for ten hours on a single machine. Note that this is a SASI approach because I repeatedly run each benchmark with the same input file to populate the database.

Because this experiment aims to maximize steady-state performance, I compare against a VM that compiles all methods at the highest level of optimization. I also use the so-called `replay` system in the virtual machine, which is essentially a profile-directed system. I verified that aggressive compilation and the use of profile-directed feedback in the comparison system is faster (in the steady-state) than allowing the VM to incrementally re-optimize methods. In other words, I compare against the *fastest* steady-state strategy. To collect the steady-state runtime, I iteratively run each benchmark three times and take the fastest of the last two runs. It is important to note that in `replay` mode, the VM compiles all methods upfront, and thus compilation time does not factor into these results (*i.e.*, no compilation is performed in the second and third iterations of the benchmarks).

Figure 7-6 shows the gains attained by collaborative compilation on the DaCapo benchmark suite (version 050224) [28].[3] The results are averaged over ten runs of each application and

[3]My version of the RVM cannot run `xalan`, `batik`, or `chart` because of preexisting bugs.

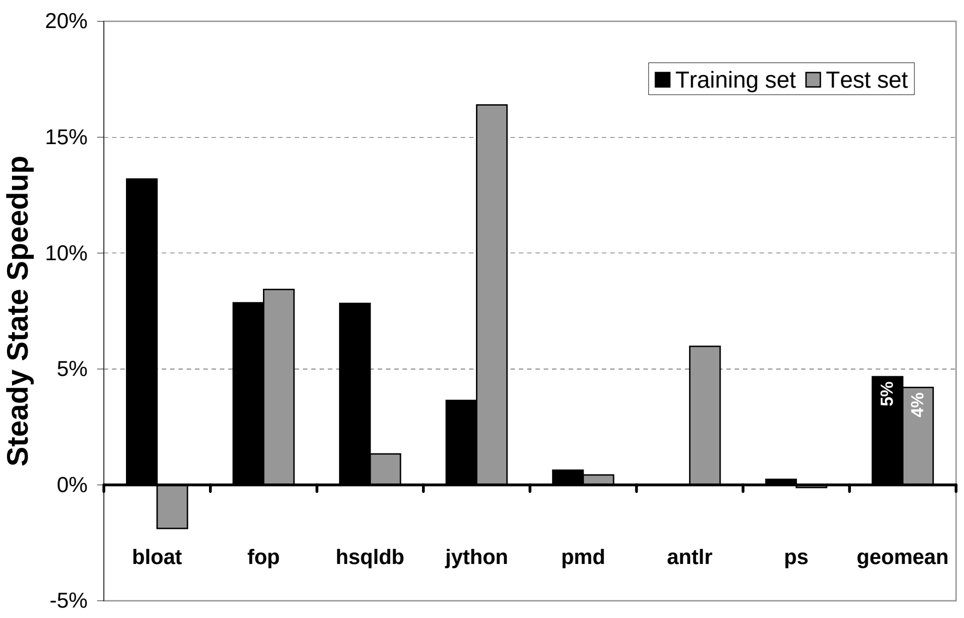


Figure 7-6: Steady-state performance of collaboratively tuned benchmarks. The knowledge base was populated with training examples for each benchmark (around 600 examples for each benchmark) running the default dataset. As expected, access to the knowledge base significantly increases the performance of these applications on the default dataset. However, it also speeds up the alternate dataset, significantly in some cases.

input data set. This experiment was performed on a single 1.8 Ghz Pentium M machine with 512Mb of memory running Ubuntu Linux. The training input data set was exclusively used to populate the collaborative database. The black bar indicates the performance of each application on the training data set (which corresponds to the 'default' input in DaCapo). We see that the collaborative database leads to substantial improvements for four of these benchmarks, and furthermore, it does not slow down a single application. On this input set, the database leads to a 5% improvement. Interestingly, when we apply the test dataset, shown by the the gray bars, the performance is actually better for three of the benchmarks. I use the 'small' input as the test data set. Because this input set processes less data than the training input (for the most part), the memory system is not a bottleneck, which increases the net impact that improved inlining settings have on performance. Overall, the database leads to a 4% improvement on the test dataset.

7.4 Reducing Privacy Concerns

The virtual machine obviously has access to all of the code that is run by a user. Therefore, privacy-conscious users may understandably be wary of using a system that communicates abstracted descriptions of the programs it runs to a centralized server. I have done several things to allay such concerns.

First, if the collaborative community were to adopt a reinforcement-style learning algorithm such as policy search, the only data that a client would communicate to the knowledge base is a reinforcement signal — in the form of a runtime ratio — that indicates whether the agent's actions were beneficial.

Second, in the event that the community chose to use supervised learning, the data that is sent only summarizes *high-level abstractions* of the code that the virtual machine has run, as well as a label which indicates the configuration that results in the best performance. Furthermore, the system only collects per-method measurements for small slices of the program. It would be difficult to piece together the structure of a program based on data reported to the centralized server. I acknowledge that the proof-of-concept experiment communicates fully-qualified method names, which is a definite privacy leak. As the system matures, I will replace these names with abstract feature vectors. This means that there may be conflicts (*i.e.*, two different methods might have the same feature signature), but it also has the advantage that there will be no problems with versioning (*i.e.*, there might be several methods named `Foo.foo` that have entirely different structures).

For users who do not want others to have any idea of the appearance of the programs they run, the collaborative compiler can be configured to run locally. No data is communicated to the outside world.

7.5 Conclusion

Collaborative compilation has the potential to alleviate the data collection requirement imposed by meta optimization. A potential drawback of the meta optimization methodology is that it requires a separate training phase, which is performed "at the factory." However, such factory settings may not work well in the field: subtle differences between architectures and operating systems can negate the improvements of at-the-factory training. In contrast, collaborative infrastructures offer a diverse pool of training data from which a general heuristic can be constructed. Furthermore the process of training a collaborative heuristic is much simpler from the user's perspective than the process employed by previous researchers: simply running the application trains the heuristic.

The work presented in this chapter is an important first step toward collaborative compilation. I show that the sampling overhead is quite small with respect to the potential gains of such a system. The data extraction mechanism is effective because it focuses on the programs (and input data sets) that users are actually running. In addition, users have access to a potentially much larger knowledge base than they would be able to generate on their own.

While the prototype collaborative compiler resides in a JIT, one could easily apply the techniques described in this chapter to traditional compilation. In fact, a traditional compilation framework would pose fewer challenges as much of the engineering effort was spent dealing with VM noise and speed-of-compilation constraints.

Chapter 8

Related Work

The idea of applying machine learning techniques to difficult systems problems is becoming increasingly accepted. In fact, many researchers have applied machine learning to compilation. This chapter describes the work that is most related to the research I presented in the previous chapters.

The next section describes searching approaches that have been applied for program specialization. This mostly corresponds to SASI training. Remember, there is no guarantee that applying the parameter settings that maximize the performance of the specialized application will enhance the performance of any other program. The following section describes MASI approaches, which attempt to find a unique heuristic that performs well for a wide range of programs. This chapter concludes by describing work that relates to collaborative compilation.

8.1 Program Specialization

The *Superoptimizer* is among the first approaches to program specialization [70]. Superoptimization exhaustively searches for programs with certain properties, such as the smallest sequence of instructions that performs a certain task. For instance, in [70], the Superoptimizer takes as input an assembly specification of a function, and then it creates all possible assembly programs that contain less instructions than the input. Each of these computer-generated sequences is run on an array of inputs and the outputs are tested against precomputed outputs that are known to be correct. While this quick-and-dirty correctness check is not perfect, it significantly increases search speed. A much slower Boolean test, which validates the correctness of a sequence bit-by-bit, can be used to validate short sequences found by the quick-and-dirty test. Superoptimizers often find extremely clever (and shorter) sequences that perform the same computation as the input program. However, because pointer arithmetic drastically complicates the search, this approach is only feasible for short input functions that do not access memory [70].

Bernstein et al. manually identified three priority functions for choosing spill variables [12]. The authors mention that these priority functions are applicable under different scenarios. For each method, their technique performs a compile-time search for the priority function that spills the fewest variables to memory. This simple approach increased the effectiveness of their register allocator; the cost of the compile-time search in this case was limited to roughly a $3x$ slowdown of the register allocator.

Iterative Compilation, which is a SASI program specialization technique, refers to any number of iterative searching methods to find near optimal command line parameter settings for a given application [14, 53, 55, 54, 27, 79, 59]. Iterative Compilation is an increasingly popular technique because of its simplicity and sheer ability to highly optimize an application. Agakov et al. add an interesting twist to iterative compilation: they increase the speed of iteration by combining MASI supervised learning techniques that can quickly identify and skip over command line settings that are likely to perform poorly [2]. ACOVEA, a freely-downloadable compiler tuning package, uses a SASI approach and genetic algorithms to find high-quality command line settings for an application [59].

Recall that the passes of an optimizing compiler are not well coordinated with downstream passes. One alternative to tweaking optimization heuristics to achieve symbiosis among the passes is to rearrange the order in which the passes are applied. Cooper et al. at Rice University use genetic algorithms (GAs) and the SASI approach to find good phase orders for application specialization [25, 26, 24]. Their compiler implements ten different optimizations, which can be run multiple times and in any order. They fix the maximum number of passes to be run at 12, leading to a search-space size of 10^{12}. Their GA assigns a fitness to each individual based on how compact the generated code is; shorter generated code sequences correspond to higher fitness levels. Thus, we expect the length of code generated by the best individuals to monotonically decrease with generation number. Intuitively, phase ordering can benefit greatly from encouraging the survival of effective local solutions; it may likely be the case that some optimizations expose key properties of the code that downstream optimizations can exploit. Their work shows that GAs, which purportedly preserve good local solutions, are far more efficient than random sampling for this problem [25]. Other researchers have conducted similar studies with different learning techniques and infrastructures [82, 58].

Many other compiler researchers have also creatively used genetic algorithms for program specialization. Steven Beaty used genetic algorithms to search for effective instruction schedules [10, 11]. His work used a GA to enumerate instructions for scheduling; an off-the-shelf list scheduling algorithm uses the instruction enumerations to schedule instructions legally, and in a good priority order.

Researchers have also investigated using genetic algorithms for compiling to embedded systems. Imtiaz Ahmad et al. follow the same scheduling approach described in [10] for compacting microcode [3]. Martin Grajcar extends [10] and [3] by scheduling across branches [40]. Gary Grewal et al. use GAs to compile for a Digital Signal Processor (DSP) [42]. Their system first compiles code for a simple and easily-targeted machine model, which they call the *clean machine*. A genetic algorithm then searches for good translations from clean machine instructions to target DSP instructions. Their work focuses on instruction selection and register allocation.

Gradient-based hill climbing and simulated annealing approaches have been successfully applied to solve 2-D and 3-D bin packing problems to which layout and scheduling problems map. For instance, it is quite common to use simulated annealing for automating VLSI circuit layout (*e.g.*, [87, 97, 85, 48]). Recent work has used simulated annealing to statically distribute processing workload on multi-core architectures [39]. For a good survey on approaches to solving layout problems, see [16, 93].

8.2 Meta Optimization Related Work

Instead of searching for effective application-specific solutions, the approaches I describe here search for quality, general-purpose compiler heuristics using MASI training. Of course, meta optimization can also be used to specialize the compiler for a given application as I explained in Chapter 5; and some of the program specialization approaches I described in the last section could have been applied to meta optimization (though they were not) by including the notion of a diverse training set.

Calder et al. provide the earliest example of meta optimization [17]. They used supervised learning techniques to fine-tune static branch prediction heuristics. They employed neural networks and decision trees to construct effective static branch prediction heuristics. Their machine-learned techniques achieved a prediction rate of 80%, while state-of-the art program-based prediction heuristics could only achieve 75% accuracy. There are two important distinctions that distinguish my approach to supervised classification from theirs: first, branch prediction is a binary problem that is simpler than the multi-class problem my work considers; and secondly, their problem has the benefit that instrumentation code to determine branch direction will not affect the direction to which branches are resolved. In other words, they were able to work with a noiseless dataset. Even in a controlled environment, back-to-back runs of a program on a real machine will result in different program runtimes. To deal with noise for learning with empirical runtime data, I used repeated sampling and median statistics to assign class labels to the training data set.

Monsifrot et al. use a classifier based on "boosted" decision tree learning to determine which loops to unroll [72]. Their work considers binary classification, leaving the choice of unroll factors up to a compiler heuristic. Doing so, their learned classifier correctly predicts 86% of the loops in their benchmark suite. My work takes a multi-class approach that not only predicts if a loop should be unrolled, it also predicts the best unroll factor for the loop. Judging by the histogram in Figure 6-4, simply unrolling all the time will achieve 77% accuracy, and while unrolling may be better than not unrolling for a given example, Table 6.2 shows that choosing the wrong unroll factor can severely limit performance.[1]

[1] I use a different infrastructure than that presented in [72]. Therefore, the reader should not directly compare the 77% accuracy achieved by always predicting unroll on my infrastructure against the binary classification results [72] presents.

Cavazos and Moss use supervised learning to improve the compilation speed of a Java JIT compiler [19]. They train a learning algorithm to recognize when the compiler can forgo scheduling a basic block without sacrificing much performance. The resulting JIT compiles code faster while retaining 90% of the performance of scheduling every basic block. For a similar reason, Cavazos et al. use supervised classification techniques to choose between two different optimizations: they learn when it is appropriate to use a compile-time intensive graph coloring allocator instead of the fast linear-scan allocator [20]. While these two problems are interesting and well-formed, they are again both binary problems.

Cavazos et al. use iterative compilation based on GAs to search the space of command line settings for a Java Virtual Machine [21]. In particular, they use their system to search for good settings to five command line arguments that affect method inlining.

Moss et al. use supervised learning to learn how to schedule instructions from a ready worklist [74]. Because of the combinatorial blowup of measuring all permutations of instructions — in order to extract the training data examples — it is unclear whether a supervised approach applies to this problem. The authors achieve good results, but they admit that it was difficult to cast the problem into a form suitable for supervised learning.

Shortly thereafter, McGovern et al. used the TD(λ) reinforcement learning approach to learn to schedule instructions in a basic block [71]. The TD(λ) heuristic performed slightly worse than the hand-crafted commercial scheduler against which they compared. This approach to meta optimization resembles my policy search experiments; while policy search and TD(λ) are both forms of reinforcement learning, policy search is the simpler approach. Policy search simply looks for a policy that works well in the compiler's environment, whereas TD(λ) tries to estimate the utility of being in a given state in the environment and bases decisions on the utilities. In addition, I recognized the critical importance of priority functions, and the applicability of policy search to automatically creating them; this thesis shows results of policy search for two priority functions. I describe a third experiment in [91] that uses policy search to find a policy for data prefetching. These three experiments show that policy search is a generally applicable technique.

Shun Long et al. use classification to select among a set of optimizations that historically performed well for similar code [66]. Their creative approach learns how to effectively predict which combinations of transformations are applicable for a given code segment. They performed their experiments concurrently with the multi-class experiments I conducted in this thesis. One drawback of the methodology presented in [66] is the size and similarity of the benchmarks on which the system was tested: they used small kernels, many of which are quite similar to one another, to test their methodology.

8.3 Collaborative Compilation Related Work

Diniz and Rinard implemented a system for improving the dynamic, or steady-state performance of an application [32]. Their system creates multiple versions of a method and periodically samples each to see which version is the fastest for a given computation. More recently, Lau et al. use a similar approach to Diniz's for automatically adapting the steady-state performance of IBM's product Java VM [61]. The key difference between my work and these related projects is that my data is intended to be contributed to a collaborative community of users, and thus the data we extract must be representative of possible *future* runtimes. The dynamic approaches employed by Diniz and Lau do not necessarily care why one version is better than another; their systems can attain speedups as long as one version *is* better than another. In theory, by learning when to apply a given optimization, a collaborative approach can improve the total runtime of an application, not just the steady-state runtime.

Arnold, Welc, and Rajan, maintain an application-specific database of profile information to improve the performance of a user's applications [7]. Their system collects profile information, in the form of coarse timer information, across multiple runs of a given application that helps identify — early in the application's execution — which methods are likely to become "hot". Their system, like mine, gradually accrues data that helps improve future invocations of the program. Essentially, their system provides a good estimate of the expected future runtime (in terms of baseline-compiled time) of a given method. Thus, their work is exceptionally effective at improving the startup costs of a VM because it does not spend time compiling ephemeral methods (and it can also aggressively compile long-lasting methods). My work is orthogonal to theirs; the information my system gathers would improve their results, and likewise their information would improve mine. Another major differentiation is the scalability of the two different approaches: collaborative compilation can support large communities, whereas the work in [7] is designed exclusively for local usage.

Ben Liblit et al. have implemented a system for isolating bugs. Their system instruments user programs at-the-factory with statistically sampled code that tracks the values of key program variables. Users pay a slight runtime overhead in exchange for helping the software developer identify bugs [65, 64, 63]. Their system, like collaborative compilation, benefits from the analysis of large amounts of aggregated data.

Chapter 9

Conclusion

This thesis introduces a systematic process for automatically creating compiler heuristics. The meta optimization approach takes advantage of the typical two-phased construction of compiler optimizations. First an optimization's correctness phase ensures that the compiler will not generate incorrect code, and then the policy phase determines which of the legal actions to take. Focusing optimization effort on the policy has several advantages:

- Any policy is legal. This is an important attribute, because above all else, users trust the compiler to faithfully maintain the semantics of an input program.
- Policies are critical to performance. In this thesis I have closely examined several compiler heuristics. I have shown that a good compiler policy can easily provide a factor of two speedup over poor policies.
- Policies are prevalent in compilers. My research has automatically constructed heuristics for hyperblock formation, register allocation, data prefetching, loop unrolling, method inlining, phase ordering,[1] and adaptive recompilation. Many other optimizations could also benefit from my approach.

Priority functions are an important subclass of policies. Many suboptimal heuristics for NP-hard problems are priority-based, meaning small priority functions make all of the

[1]Phase ordering research is joint work with Diego Puppin [82].

performance-critical decisions. My thesis describes a simple and effective approach called policy search, that automatically creates excellent priority functions. My genetic programming-based implementation found better priority functions — sometimes much better — than the best human-constructed priority functions, and with virtually no human intervention.

Policy search can 'solve' a wide range of compiler problems. For instance, it would be extremely difficult to cast the search for priority functions into a supervised learning problem (and to extract a labeled training data set). However, for cases in which it is easy to create a labeled training data set, supervised learning is the most effective and efficient approach, and it extends to problems that require categorization into multiple classes. This dissertation describes an experiment whereby I create a highly effective loop unrolling policy using multi-class supervised learning.

With a labeled training set in place, I have shown how to use feature selection to identify the most informative program characteristics for a given optimization policy. Such an approach enables the compiler writer to extract *any* feature he or she thinks *might* be important for categorization. Because of the strange interactions between various compiler passes and the architecture, determining which features are important is nontrivial for humans. Feature selection techniques not only improve the quality of the generated heuristics, they further reduce the amount of time and effort required to generate quality compiler policies. If my techniques are proven to be widely accepted, compiler infrastructures may someday provide 'feature extraction' mechanisms, much like current compilers provide generic data flow analysis packages.

Meta optimization also has the potential to aid with architecture design-space exploration. The tightly-coupled nature of compilers and computer architectures necessitates co-design. However, because it is currently difficult to re-tune compiler optimizations for new architectures, computer architects cannot adequately test the true potential of their innovations. Meta optimization can allow architects to quickly test the system-level impact of their designs. Similarly, compiler writers can use these techniques to automatically re-target new or emerging architectures.

While the training phase may require several days of unsupervised processing, the machine-learned policies will not measurably increase compilation time. The compiler vendor can train the heuristics “at-the-factory”, and deploy the compiler with the learned policies. While my techniques can (and do) construct general-purpose compiler heuristics that apply to unseen benchmarks, in general, the learned heuristics will perform best on the applications on which they were trained. Thus, in order to build general machine-learned compiler heuristics one has to collect a large amount of data, drawn from a diverse set of programs (and preferably, drawn from programs that users actually run).

In light of this, I have introduced collaborative compilation, a new paradigm for self-improving compilers that has the potential to satisfy the stringent data requirements of machine learning algorithms. By distributing the data collection process over a large community of users, collaborative compilation will experience no shortage of training data. Furthermore, the users of a collaborative community drive the learning process: ordinary users train the machine learning algorithms to perform well on *their* important applications and input data sets.

I firmly believe that the meta optimization techniques that my colleagues and I are pursuing will fundamentally change compiler development. Today, most compiler developers do not start from scratch when targeting a new architecture. Instead, developers leverage existing compilers, and adapt them to the task at hand. Nevertheless, in order to attain high levels of performance, all of the policies must be re-tuned. I believe that automatic heuristic tuning based on empirical evaluation will become prevalent, and that designers will someday intentionally expose algorithm policies to facilitate machine learning optimization.

That said, we still have much work to perform before meta optimization is adopted by the general compiler community. If we are to popularize meta optimization we need to build tools that compiler writers will want to use: we need to write feature extraction libraries, generalized timers for creating labels, and toolkits for feature selection and automatic heuristic generation. Once this infrastructure is in place, applying meta optimization will be painless.

Apart from the “user interface”, there are still many interesting problems left to investigate. To my knowledge, all meta optimization approaches have looked at optimizations in isolation. In other words, no one has tried to simultaneously learn two or more symbiotically linked optimizations. This is an interesting direction for future work.

Among the more pressing items that the community has to address is collecting accurate measurements without affecting the object under measurement. Modern architectures are helping our cause by exporting many architectural profiling mechanisms; however, architectural timers and counters are generally expensive (in terms of processor clock cycles) to access. This issue becomes even more critical when measuring small sections of code.

While collaborative compilation has great potential, the approach is in its infancy. Much work is needed to actually deploy a collaborative compiler. Because collaborative heuristics reside in JIT compilers, where speed of predictions is a concern, computationally intensive machine learning algorithms will negate any performance benefit from making correct predictions: we have to find accurate and *fast* predictors. We must also investigate all of the privacy concerns that come with a data sharing system. Future work will focus on finding high-payoff JIT optimizations that will benefit from meta optimization and collaborative compilation. Additionally, we will study the community aspect of collaborative systems. Can collaboratively trained heuristics adapt to dynamically changing user communities? Can general-purpose heuristics be induced in diverse computing communities?

The results presented in this dissertation lead me to believe that machine learning techniques can optimize policies at least as well human designers. Meta optimization has the potential to simultaneously reduce the human effort required for compiler design, and increase the attainable performance of tomorrow's complex systems.

Bibliography

[1] S. G. Abraham, V. Kathail, and B. L. Deitrich. Meld Scheduling: Relaxing Scheduling Constaints Across Region Boundaries. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29)*, pages 308–321, 1996.

[2] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using Machine Learning to Focus Iterative Optimization. In *International Symposium on Code Generation and Optimization*, New York, New York, Mar. 2006.

[3] I. Ahmad, M. K. Dhodhi, and K. A. Saleh. An Evolution-Based Technique for Local Microcode Compaction. In *Microprocessors and Microsystems*, volume 19, pages 467–474, 1995.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers : Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[5] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

[6] B. Alpen, C. Attanasio, J. Barton, M. Burke, P. Cheng, J. Choi, A. Cocchi, S. Find, D. Grove, M. Hind, S. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. Russell, V. Sarkar, M. Serrano, J. Shepherd, S. Smith, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. In *IBM Systems Journal*, volume 39(1), pages 211–238, February 2000.

[7] M. Arnold, A. Welc, and V. Rajan. Improving Virtual Machine Performance Using a Cross-Run Profile Repository. In *Proceedings of the SIGPLAN '05 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[8] W. Banzhaf, P. Nordin, R. Keller, and F. Francone. *Genetic Programming : An Introduction : On the Automatic Evolution of Computer Programs and Its Applications.* Morgan Kaufmann, 1998.

[9] A. G. Barto, R. S. Sutton, and C. Watkins. Learning and Sequential Decision Making. In M. Gabriel and J. Moore, editors, *Learning and Computational Neuroscience : Foundations of Adaptive Networks.* M.I.T. Press, Cambridge, Mass, 1990.

[10] S. J. Beaty. Genetic Algorithms and Instruction Scheduling. In *Proceedings of the 24th Annual International Symposium on Microarchitecture (MICRO-24)*, November 1991.

[11] S. J. Beaty. *A Study of Instruction Scheduling Using Genetic Algorithms.* PhD thesis, Colorado State University, Mechanical Engineering Department, 1991.

[12] D. Bernstein, D. Goldin, M. Golumbic, H. Krawcyk, Y. Mansour, I. Nahshon, and R. Y. Pinter. Spill Code Minimization Techniques for Optimizing Compilers. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 258–263, 1989.

[13] D. Bernstein, M. Rodeh, and I. Gertner. On the Complexity of Scheduling Problems for Parallel/Pipelined Machines. *IEEE Transactions on Computers*, 38(9):1308–1313, September 1989.

[14] F. Bodin, T. Kisuki, P. Knijnenburg, M. F. O'Boyle, and E. Rohou. Iterative Compilation in a Non-Linear Optimization Space. In *Workshop on Profile and Feedback Directed Compilation*, 1998.

[15] D. Bourgin. *Losslessy compression schemes.* http://hpux.u-aizu.ac.jp/hppd/hpux/Languages/codecs-1.0/.

[16] J. Cagan, K. Shimada, and S. Yin. A Survey of Computational Approaches to Three-Dimensional Layout Problems. *Computer Aided Design*, 34(8):597–611, 2002.

[17] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-Based Static Branch Prediction Using Machine Learning. In *ACM Transactions on Programming Languages and Systems (TOPLAS-19)*, volume 19, 1997.

[18] S. Carr and K. Kennedy. Improving the Ratio of Memory Operations to Floating-Point Operations in Loops. In *ACM Transactions on Programming Languages and Systems (TOPLAS-16)*, November 1994.

[19] J. Cavazos and J. E. B. Moss. Inducing Heuristics to Decide Whether to Schedule. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004*. ACM, 2004.

[20] J. Cavazos, J. E. B. Moss, and M. F. O'Boyle. Hybrid Optimizations: Which Optimization Algorithm to Use? In *15th International Conference on Compiler Construction (CC 2006)*, Vienna, Austria, Mar. 2006.

[21] J. Cavazos and M. F. O'Boyle. Automatic Tuning of Inlining Heuristics. In *11th International Workshop on Compilers for Parallel Computers*, January 2006.

[22] P. Chang, D. Lavery, S. Mahlke, W. Chen, and W. Hwu. The Importance of Prepass Code Scheduling for Superscalar and Superpipelined processors. In *IEEE Transactions on Computers*, volume 44, pages 353–370, March 1995.

[23] F. C. Chow and J. L. Hennessey. The priority-based coloring approach to register allocation. In *ACM Transactions on Programming Languages and Systems (TOPLAS-12)*, pages 501–536, 1990.

[24] K. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Exploring the Structure of the Space of Compilation Sequences Using Randomized Search Algorithms. In *Los Alamos Computer Science Institute Symposium*, 2003.

[25] K. Cooper, P. Scheilke, and D. Subramanian. Optimizing for Reduced Code Space using Genetic Algorithms. In *Languages, Compilers, Tools for Embedded Systems*, pages 1–9, 1999.

[26] K. Cooper, D. Subramanian, and L. Torczon. Adaptive Optimizing Compilers for the 21^{st} Century. In *Los Alamos Computer Science Institute Symposium*, 2001.

[27] K. Cooper and T. Waterman. Investigating Adaptive Compilation using the MIPSpro Compiler. In *Los Alamos Computer Science Institute Symposium*, 2003.

[28] The DaCapo Benchmark Suite. *http://ali-www.cs.umass.edu/DaCapo/gcbm.html.*

[29] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization.* Birkhäuser, 2000.

[30] J. W. Davidson and S. Jinturkar. Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 186–195, Orlando, FL, June 1994.

[31] T. Dietterich and G. Bakiri. Solving Multiclass Learning Problems via Error-Correcting Output Codes. *Journal of Artificial Intelligence Research*, 2:263–286, 1995.

[32] P. C. Diniz and M. C. Rinard. Dynamic Feedback: An Effective Technique for Adaptive Computing. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 71–84, 1997.

[33] R. Duda, P. Hart, and D. Stork. *Pattern Classification.* Wiley-Interscience, 2001.

[34] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* The MIT Press, Cambridge, MA, 1985.

[35] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[36] C. Gathercole. *An Investigation of Supervised Learning in Genetic Programming.* PhD thesis, University of Edinburgh, 1998.

[37] P. B. Gibbons and S. S. Muchnick. Efficient Instruction Scheduling for a Pipelined Architecture. In *Proceedings of the ACM Symposium on Compiler Construction*, volume 21, pages 11–16, 1986.

[38] A. Gionis, P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th Conference on Very Large Data Bases*, pages 518–529, February 1999.

[39] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, USA, October 2002.

[40] M. Grajcar. Conditional Scheduling for Embedded Systems Using Genetic List Scheduling. In *13th International Symposium on System Synthesis (ISSS)*, 2000.

[41] J. J. Grefenstette, C. L. Ramsey, and A. Schultz. Learning Sequential Decision Rules Using Simulation Models and Competition. *Machine Learning*, 5:355–381, 1990.

[42] G. W. Grewal and T. C. Wilson. Shake and Bake: A Method of Mapping Code to Irregular DSPs. In *10th International Conference on VLSI Design*, 1997.

[43] J. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, 1975.

[44] Intel. *http://www.intel.com.*

[45] Intel. Hyper-threading technology. *Intel Technology Journal*, 6(1), February 2002.

[46] Intel. *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*, May 2004.

[47] Intel. *IA-32 Intel Architecture Software Developer's Manual*, March 2006.

[48] S. Jajodia, I. Minis, G. Harkhalakis, and J.-M. Proth. CLASS: Computerized LAyout Solutions Using Simulated Annealing. *International Journal of Production Research*, 30(1):95–108, 1992.

[49] Java Grande. *http://www.epcc.ed.ac.uk/javagrande/.*

[50] D. A. Jiménez. Code Placement for Improving Dynamic Branch Prediction Accuracy. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 107–116, New York, NY, USA, 2005. ACM Press.

[51] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[52] M. Kessler and T. Haynes. Depth-Fair Crossover in Genetic Programming. In *Proceedings of the ACM Symposium on Applied Computing*, pages 319–323, February 1999.

[53] T. Kisuki, P. Knijnenburg, and M. F. O'Boyle. Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation. In *In Proceedings of Parallel Architectures and Compilation Techniques*, pages 237–246, 2000.

[54] P. Knijnenburg, T. Kisuki, K. Gallivan, and M. F. O'Boyle. The Effect of Cache Models on Iterative Compilation for Combined Tiling and Unrolling. In *Concurrency and Computation: Practice and Experience*, volume 16, pages 247–270, 2004.

[55] P. Knijnenburg, T. Kisuki, and M. F. O'Boyle. Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation. In *The Journal of Supercomputing*, volume 24, pages 43–67, 2003.

[56] J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, 1992.

[57] U. Kremer. Optimal and Near-Optimal Solutions for Hard Compilation Problems. *Parallel Processing Letters*, 7(4):371–378, 1997.

[58] P. Kulkarni, D. Whalley, G. Tyson, and J. Davidson. Exhaustive Optimization Phase Order Space Exploration. In *International Symposium on Code Generation and Optimization*, New York, New York, Mar. 2006.

[59] S. R. Ladd. ACOVEA : Analysis of Compiler Options via Evolutionary Algorithm. http://www.coyotegulch.com/products/acovea/index.html.

[60] S. Larsen and S. Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 145–156, Vancouver, BC, June 2000.

[61] J. Lau, M. Arnold, M. Hind, and B. Calder. Online Performance Auditing: Using Hot Optimizations Without Getting Burned. In *PLDI '06: Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, 2006.

[62] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In *International Symposium on Microarchitecture*, volume 30, pages 330–335, 1997.

[63] B. Liblit, A. Aiken, and A. Zheng. Distributed Program Sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, California, June 9–11 2003.

[64] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug Isolation via Remote Program Sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, California, June 9–11 2003.

[65] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable Statistical Bug Isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Chicago, Illinois, June 12–15 2005.

[66] S. Long and M. O'Boyle. Adaptive Java Optimisation Using Instance-Based Learning. In *International Conference on Supercomputing*, 2004.

[67] S. A. Mahlke. *Exploiting Instruction Level Parallelism in the Presence of Branches.* PhD thesis, University of Illinois at Urbana-Champaign, Department of Electrical and Computer Engineering, 1996.

[68] S. A. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *International Symposium on Microarchitecture*, volume 25, pages 45–54, 1992.

[69] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proc. 25th Annual International Symposium on Microarchitecture*, pages 45–54, Portland, OR, December 1992.

[70] H. Massalin. Superoptimizer: A Look at the Smallest Program. In *ASPLOS-II: Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, pages 122–126, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[71] A. McGovern and J. E. B. Moss. Scheduling Straight-Line Code Using Reinforcement Learning and Rollouts. In *Proceedings of Neural Information Processing Symposium*, 1998.

[72] A. Monsifrot, F. Bodin, and R. Quiniou. A Machine Learning Approach to Automatic Production of Compiler Heuristics. In *Artificial Intelligence: Methodology, Systems, Applications*, pages 41–50, 2002.

[73] D. E. Moriarty, A. C. Schultz, and J. J. Grefenstette. Evolutionary Algorithms for Reinforcement Learning. *Journal of Artificial Intelligence Research*, 11:199–229, 1999.

[74] J. E. B. Moss, P. Utgoff, J. Cavazos, D. Precup, D. Stefanovi, C. Brodley, and D. Scheeff. Learning to Schedule Straight-Line Code. In *Proceedings of Neural Information Processing Symposium*, 1997.

[75] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[76] D. Newman, S. Hettich, C. Blake, and C. Merz. UCI repository of machine learning databases, 1998.

[77] Open Research Compiler. *http://ipf-orc.sourceforge.net*.

[78] E. Ozer, S. Banerjia, and T. Conte. Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register Filee Microarchitectures. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-24)*, pages 308–315, 1998.

[79] Z. Pan and R. Eigenmann. Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning. In *International Symposium on Code Generation and Optimization*, New York, New York, Mar. 2006.

[80] J. C. H. Park and M. S. Schlansker. On Predicated Execution. Technical Report HPL-91-58, Hewlett Packard Laboratories, 1991.

[81] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kauffman, 1996.

[82] D. Puppin, M. Stephenson, S. Amarasinghe, M. Martin, and U. M. O'Reilly. Adapting Convergent Scheduling Using Machine Learning. In *Proceedings of the '03 Workshop on Languages and Compilers for Parallel Computing*, College Station, TX, 2003.

[83] B. R. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-24)*, November 1994.

[84] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.

[85] R. A. Ruttenbar. Simulated Annealing Algorithms: An Overview. *IEEE Circuits and Devices Magazine*, 5(1):19–26, 1989.

[86] K. Scott. On Proebsting's Law. Technical Report CS-2001-12, University of Virginia, 2001.

[87] C. Sechen. *VLSI Placement and Global Routing Using Simulated Annealing*. Kluwer Academic Publishers, Boston, MA, USA, 1988.

[88] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and Exploiting Program Phases. *IEEE Micro*, 23(6):84–93, 2003.

[89] SPEC.org. *http://www.spec.org*.

[90] M. Stephenson and S. Amarasinghe. Predicting Unroll Factors Using Supervised Classification. In *International Symposium on Code Generation and Optimization*, San Jose, California, Mar. 2005.

[91] M. Stephenson, M. Martin, U. M. O'Reilly, and S. Amarasinghe. Meta Optimization: Improving Compiler Heuristics with Machine Learning. In *Proceedings of the SIGPLAN '03 Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.

[92] J. Suykens and J. Vandewalle. Least Squares Support Vector Machine Classifiers. *http://www.esat.kuleuven.ac.be/sista/lssvmlab/*.

[93] R. Tessier. *Fast Place and Route Approaches for FPGAs*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1998.

[94] Trimaran. *http://www.trimaran.org*.

[95] B. Wagner. Collaborative Compilation. Master's thesis, Massachusetts Institute of Technology, 2006.

[96] N. Warter. *Modulo Scheduling with Isomorphic Control Transformations*. PhD thesis, University of Illinois at Urbana-Champaign, Department of Electrical and Computer Engineering, 1993.

[97] D. F. Wong, H. W. Leong, and C. L. Liu. *Simulated Annealing for VLSI Design*. Kluwer Academic Publishers, Boston, MA, USA, 1988.