RICE UNIVERSITY

Adaptive Compilation and Inlining

by

Todd Waterman

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Keith D. Cooper, Professor, Chair
Computer Science

Ken Kennedy, Professor
Computer Science

Devika Subramanian, Professor
Computer Science

Tim Warburton, Assistant Professor
Computational and Applied Mathematics

Houston, Texas

December, 2005
INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.
Adaptive Compilation and Inlining

Todd Waterman

Abstract

Adaptive compilation uses a feedback-driven process to leverage additional compilation time into improved executable performance. Previous work on adaptive compilation has demonstrated its benefit at an inter-optimization level. This dissertation investigates the ability of adaptive techniques to improve the performance of individual compiler optimizations.

We first examine the ability to use adaptive compilation with current commercial compilers. We use adaptive techniques to find good blocking sizes with the MIPSpro compiler. However, we also observe that current compilers are poorly parameterized for adaptive compilation.

We then construct an adaptive inlining system that demonstrates the potential of adaptive compilation to improve individual optimizations. We design the inliner to accept condition strings that determine which call sites are inlined. We develop an adaptive controller for the inliner based on a detailed understanding of the search space that the condition strings provide.

Our adaptive inlining system consistently finds good sets of inlining decisions and outperforms static techniques. In addition, we demonstrate the inability of static techniques to provide a universal inlining solution and the necessity of adaptive inlining. Adaptive inlining demonstrates the capacity of adaptive compilation to improve the performance of a single, carefully designed optimization.
Acknowledgments

This work would not have been possible without my advisor, Keith Cooper. His guidance was critical to the development of this dissertation, as well as my understanding of both compilers and computer science as a whole.

My committee members Ken Kennedy, Devika Subramanian, and Tim Warburton each provided valuable insights that helped shape this document.

Many members of the compiler groups at Rice helped me with my work and my education as a graduate student. Daniel Chavarria, Anshu Dasgupta, Jason Eckhardt, Alex Grosul, Tim Harvey, Cheryl McCosh, John Mellor-Crummey, Apan Qasem, Jeff Sandoval, Steve Reeves, and Lind Torczon all deserve mention. Tim Harvey deserves special recognition for always having time to discuss ideas, explain a concept, provide direction, or just talk. Cheryl McCosh also deserves credit for helping me preserve my sanity at the expense of her own.

Jack Davidson and Anne Holler provided the INLINER tool that much of my work was built on top of.

Finally, my parents, George and Bitsy, and my sister, Molly, were invaluable. They provided tons of support, sent me cookies, and kept me going through the long process of getting my degree. I can’t thank them enough.
# Contents

Abstract i

List of Illustrations v

List of Tables viii

1 Introduction 1

2 Related Work 8

2.1 Adaptive Compilation ........................................... 8

2.2 Procedure Inlining ............................................. 12

3 Investigating Adaptive Compilation using the MIPSPro Compiler 18

3.1 Introduction ...................................................... 18

3.2 ATLAS and Other Work ........................................ 20

3.3 Adjusting Blocking Size ....................................... 21

3.4 Determining Blocking Size ................................... 28

3.5 Conclusion ..................................................... 29

4 Building an Inliner for an Adaptive System 31

4.1 Introduction ...................................................... 31

4.2 Background .................................................... 32

4.3 Building a Flexible Inliner ................................... 33

4.3.1 Condition String Properties ............................... 35

4.3.2 Gathering Dynamic Information ........................... 38
5 Building an Adaptive Inlining System

5.1 Introduction .......................................................... 54
5.2 Finding an Objective Function ........................................ 55
5.3 Preventing Repeated Evaluations ..................................... 59
5.4 Choosing a Condition String Structure ............................... 63
5.5 Developing an Adaptive Approach ................................... 65
5.6 Results ........................................................................ 71
5.7 Tuning the Hill Climber .................................................. 76
5.8 Examining Sets of Decisions ............................................. 82
5.9 Future Work .............................................................. 90
5.10 Conclusions ............................................................. 93

6 Conclusion ..................................................................... 95

Bibliography ..................................................................... 97
Illustrations

3.1 Running times for $500^2$ arrays ........................................ 23
3.2 Running times for $1000^2$ arrays ........................................ 23
3.3 Running times for $1500^2$ arrays ........................................ 23
3.4 Running times for square matrices ..................................... 24
3.5 Relative running times for square matrices ......................... 25
3.6 L1 cache misses for square matrices ................................... 26
3.7 L2 cache misses for square matrices ................................... 26
3.8 Running times for varying M ............................................. 26
3.9 Running times for varying N ............................................. 27
3.10 Running times for varying K ............................................ 27
3.11 Search time required ..................................................... 29

4.1 Inlining and compile times for vortex ................................ 42
4.2 Source program size for vortex ......................................... 43
4.3 Executable size for vortex ................................................ 44
4.4 Execution times for vortex ................................................ 45
4.5 Sweep of statement count and loop nesting depth .................. 47
4.6 Sweep of statement count and static call count ..................... 47
4.7 Sweep of statement count and constant parameter count ........... 48
4.8 Execution times for bzip2 .................................................. 49
4.9 Execution times for vpr .................................................... 49
4.10 Execution times for parser ............................................... 50
4.11 Execution times for mpeg2-encode ........................................ 50
4.12 Sweep of statment count and loop nesting depth ...................... 51

5.1 Timer accuracy for a SC sweep of vortex .............................. 59
5.2 Timer accuracy as a percentage for a SC sweep of vortex .......... 60
5.3 Timer accuracy for a SC sweep of bzip2 ............................... 60
5.4 Timer accuracy as a percentage for a SC sweep of bzip2 .......... 61
5.5 Timer accuracy for a SC sweep of parser ............................. 61
5.6 Timer accuracy as a percentage for a SC sweep of parser .......... 62
5.7 Running time for Vortex using different inlining strategies ........ 72
5.8 Running time for Vortex using different inlining strategies (zoomed) ... 73
5.9 Standard deviation in results for Vortex using different inlining techniques ................................................................. 74
5.10 Evaluations required for Vortex using different inlining techniques .. 74
5.11 Normalized execution time of benchmarks using various inlining methods ................................................................. 76
5.12 Effect of patience on the Vortex benchmark ......................... 77
5.13 Effect of patience on the Parser benchmark ......................... 78
5.14 Effect of patience on the Bzip2 benchmark ......................... 78
5.15 Effect of patience on the Gzip benchmark .......................... 79
5.16 Effect of patience on the Mcf benchmark .......................... 79
5.17 Execution time vs. Executable size for vortex ..................... 83
5.18 Execution time vs. Source lines for vortex ........................ 84
5.19 Execution time vs. Executable size for parser ..................... 85
5.20 Execution time vs. Executable size for bzip2 ..................... 85
5.21 Execution time vs. Executable size for gzip ....................... 86
5.22 Execution time vs. Executable size for mcf ....................... 86
5.23 Execution time vs. Source lines for parser ........................ 87
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.24</td>
<td>Execution time vs. Source lines for bzip2</td>
<td>87</td>
</tr>
<tr>
<td>5.25</td>
<td>Execution time vs. Source lines for gzip</td>
<td>88</td>
</tr>
<tr>
<td>5.26</td>
<td>Execution time vs. Source lines for mcf</td>
<td>88</td>
</tr>
<tr>
<td>5.27</td>
<td>Execution time vs. Dynamic calls inlined for vortex</td>
<td>89</td>
</tr>
<tr>
<td>5.28</td>
<td>Execution time vs. Dynamic calls inlined for parser</td>
<td>90</td>
</tr>
<tr>
<td>5.29</td>
<td>Execution time vs. Dynamic calls inlined for bzip2</td>
<td>91</td>
</tr>
<tr>
<td>5.30</td>
<td>Execution time vs. Dynamic calls inlined for gzip</td>
<td>91</td>
</tr>
<tr>
<td>5.31</td>
<td>Execution time vs. Dynamic calls inlined for mcf</td>
<td>92</td>
</tr>
</tbody>
</table>
Tables

4.1 Varying inlining results for vortex ......... 41

5.1 Bounds for condition string parameters ....... 67
5.2 Division of the SC parameter for bzip2 using different distributions . 70
5.3 Summary of benchmarks used ................. 75
5.4 Frequency each neighbor was chosen as the downward step ......... 81
Chapter 1

Introduction

Fortran, the first successful high-level language, was designed in 1954 by John Backus's group at IBM. The goal of Fortran was to simplify program development while retaining the high level of performance provided by programs written in assembly. Backus believed that if the first Fortran system had failed to produce efficient programs the adoption of the language would have been seriously delayed [7]. Therefore, the primary concern in the design of the Fortran compiler, as well as other early compilers, was the production of high-quality executables which executed as quickly as their assembly-coded counterparts.

Over the last forty years, the advances predicted by Moore's Law [34] have radically altered the landscape of compilers depicted by Backus. Increases in processor speed have led to a de-emphasis on execution time in many situations. Users frequently use scripting languages such as MATLAB or Python to facilitate the rapid development of programs at the expense of execution time. Many programs run quickly enough that users no longer bother turning off the debug flag when compiling a program for release. Arguably, the most important feature for a compiler to gain widespread use, after the generation of correct code, is the quality of the warning and error messages.

While compiler optimization has become less significant in some areas, there remain several important areas where high-quality optimization is critical. Scientific codes are constantly pushing the boundary of computation. Weather, earthquake, and atomic-level simulations are often run at a given granularity of elements and time. When processor speeds increase or the compiler produces better code, scientists raise
the level of granularity to garner more precise results.

Scientific programming is not the only area where the role of an optimizing compiler remains important. Web servers and mail filters are widely used programs where performance is critical; optimization can minimize the amount of hardware that companies must dedicate to these services. Mainstream applications such as Microsoft Office that target a large user base also benefit from a good compiler. If optimization can reduce the processor and memory demands of the program, then the system requirements can be lowered to allow more widespread distribution. Finally, the embedded systems community cares greatly about optimizing for speed, space, and power.

Though the rapid advancements in computer performance have diminished the importance of compiler optimization for certain applications, they have at the same time enhanced the ability of a compiler to produce higher-quality executables. The high speed of newer processors allows compilers to spend effectively more time analyzing and optimizing code than was previously possible. Unfortunately, many modern compilers do not take advantage of this opportunity.

Typically, compilers run a series of optimizations that try to improve various aspects of the input code in an attempt to improve the overall performance. Many of these transformations target problems which are so complex that it is not understood how to achieve an optimal solution or it may be that computing an optimal solution is an NP-complete problem. In both cases, computing an optimal solution to the problem is infeasible and will remain so regardless of increases in computer performance. Therefore, compilers use heuristics to solve these problems. Heuristics try to provide good common-case performance based on the compiler writer's intuition.

Over time, compilers have grown to include more passes and more extensive transformations have been added. The fundamental design of compilers, however, has not changed, despite major advances in computer architecture. When a compiler is moved to a faster machine, it simply produces the same quality of code as before requires
less time. Recent compiler research has examined more aggressive methods for leveraging the extra compilation time afforded by processor advances into higher quality executables [5, 41, 44]. One such approach is adaptive compilation.

Adaptive compilation is the iterative process of compiling a program, evaluating its performance, and using the results of the evaluation to guide future compilations. Adaptive compilation is a general technique that can be applied in a variety of ways to solve many problems in compilation. For example, this approach can be used to find a good ordering of optimization passes, to determine the appropriate cache blocking size for a program, or to examine different methods of instruction scheduling.

The first step in adaptive compilation is performing an initial compilation or set of compilations and evaluating the resulting code. The initial compilations consist of running the compiler with some set of either default or random options. The code is then evaluated using an objective function. There are many possible objective functions that vary based on the goal of the adaptive process. When optimizing for speed, for example, the objective function could be the actual running time of the program, dynamic instruction counts, or a static estimate of the running time. If optimizing for space, however, the objective function could consist simply of the executable size or a combination of executable size and running time, actual or estimated.

After the initial compilation and evaluation phase, subsequent compilations are guided by feedback from the adaptive process. The result of applying the objective function to the first round of compilations is used to determine how future compilations are performed. If a compilation produces a good executable according to the objective function, then the adaptive system will make similar decisions in future compilations. Likewise, compilations that perform poorly teach the system what choices to avoid. This discovery of good and bad decisions can be modeled in several different ways; two possibilities are hill climbers and genetic algorithms [5, 28]. The feedback cycle can continue until one of several possible terminating conditions is met: a certain performance threshold is hit; several compilations have resulted in no
improvement; or a specified amount of time has elapsed.

The types of applications that require high-quality optimization irrespective of increases in processor performance are the same applications where the programmer is willing to invest significant amounts of compilation time. A large scientific code might be compiled once and then run for weeks. Commodity software is released infrequently, and a vendor may be more willing to spend a couple of extra hours or days compiling to produce a better executable. Programs placed onto an embedded device have a similar argument; new device models are shipped infrequently. In all of these cases, spending extra compilation time to produce a better executable makes sense.

The success of adaptive compilation is heavily dependent on the underlying compiler. The goal of this thesis is to demonstrate the ability of adaptive techniques to improve the efficacy of individual compiler optimizations, and illustrate the importance of high-quality, heavily-parameterized optimizations to the success of adaptive techniques. Adaptive compilation is a time intensive process that attempts to produce extremely efficient executables. An underlying compiler that performs only limited optimization or generates low-quality code works contrary to this purpose and is, therefore, unsuitable. In addition, since adaptive compilation improves executable performance by finding the right compiler configuration for a specific piece of code, it is critical that the compilation process can be changed in meaningful ways. This requires a compiler with a wide variety of parameters that impact optimization in meaningful ways. A compiler which produces excellent code but uses fixed optimizations that cannot be changed will not benefit from adaptive compilation.

We evaluated the potential of using current compilers in adaptive compilation through the construction of a system that adaptively determines blocking sizes for the SGI MIPSpro compiler [39]. We chose the MIPSpro compiler because it produces excellent code, contains high-level optimizations, and appeared to be more suitable to parameter tuning than other compilers that we had examined. We focused on high-
level optimizations for memory performance to allow our work to be compared against the ATLAS system [44]. ATLAS automatically produces high-quality linear-algebra kernels using a tuning system guided by expert knowledge to select algorithms and parameters; this makes it an excellent comparison to adaptive compilation, which attempts to use adaptive exploration to replace expert knowledge. The adaptive system’s tuning of blocking sizes was a result, not a goal. Blocking size was the only parameter of the MIPSpro compiler that we found which could be modified and produce results better than the compiler using default optimization. This observation is a comment on both the quality of MIPSpro compilation and the poverty of its parameterization.

Adaptive selection of blocking sizes using the MIPSpro compiler produces results that compare well against the ATLAS system. For large data sets, the matrix-multiply computation, when compiled by the MIPSpro compiler using the default blocking size heuristic, can take more than twice as long as code produced by the ATLAS system. Adaptively selecting the blocking size keeps performance within fifteen percent of ATLAS regardless of the array size. Better results might have been possible if not for the limited tunability of the SGI compiler. This result shows both the promise of adaptive compilation and the limitations current compilers impose on adaptive compilation. Chapter 3 discusses the results of the adaptive blocking system and the limitations of current compilers when used as part of an adaptive compiler.

The work on adaptively selecting blocking sizes showed the potential benefit of adaptive compilation, but also illustrated how the design of current optimizations limits the opportunity. Optimizations are usually designed to operate in a specific manner and rarely have parameters that can change their performance in a meaningful way. One of the results of this dissertation is to show that, in order to increase adaptive compilation’s effectiveness, optimizations must expose more decisions to external control. We know of no work on how to parameterize transformations and the impact such decisions have on code quality. To explore these issues, we developed
a highly-parameterized inliner for use with an adaptive compiler. Despite several well-known studies of inlining [8, 15, 33, 37], the selection of procedures to inline is a poorly understood, complex, problem. It is typically solved by heuristics which consider a variety of factors. The ability to parameterize all of these factors makes inlining an ideal problem to solve using adaptive techniques.

The inliner was designed to expose the decision making process to the adaptive system through a highly-expressive parameter interface. The adaptive system specifies to the inliner a list of conditions that must be satisfied for inlining to occur. The list of conditions consists of procedure and call-site properties that are compared against either static values or other program properties and then combined in conjunctive normal form. Example properties include procedure length, number of call sites, dynamic call count, and whether or not the procedure is a leaf procedure. Having a large number of properties that can be arbitrarily combined creates a huge search space that provides enormous opportunity for the adaptive system to improve the code, but also produces the challenging problem of finding good condition strings. Chapter 4 presents our highly-parameterized inliner and demonstrates its potential to improve code quality.

The construction of a tunable inliner is insufficient for validating adaptive inlining as a technique. We fit our inliner into an adaptive system, which produces excellent inlining results in reasonable time, in Chapter 5. Our adaptive system uses a randomized hill climber to explore a carefully constrained search space. We crafted the search space using experimental results and our knowledge of inlining to allow the hill climber to find good results without getting bogged down sifting through poor solutions. The adaptive inliner finds a set of inlining decisions, specific to each program, that produce faster executables than those produced using either uninlined code or a static heuristic. The magnitude of improvement, which can be large, varies based on the opportunities for inlining within a program. The important result, however, is that adaptive inlining consistently realizes the potential of inlining to improve code
performance.

This dissertation demonstrates how adaptive techniques can be used to improve the efficacy of a complex compiler optimization. This requires both constructing the optimization to be amenable to adaptive compilation and building an adaptive system tailored to the resulting search space. We built an adaptive system for procedure inlining and the result is an adaptive inliner that can maximize the potential of inlining in a way not possible for static techniques.
Chapter 2

Related Work

Before embarking on a course of research it is critical to understand how the path being pursued fits into the greater body of work. Investigations into adaptive compilation and its reliance on a highly-parameterized optimizing compiler need to begin with the examination of prior work in adaptive compilation and its origins. Furthermore, before constructing an adaptive inliner it is imperative to understand previous research into inlining and the techniques employed.

2.1 Adaptive Compilation

Adaptive compilation is a term used to describe the very general process of iteratively compiling and evaluating a program and using feedback to improve future compilations. Research in adaptive compilation has recently increased as constantly improving processor performance has made the time requirements of the iterative process relatively less expensive. Though research into adaptive compilation per se is a recent phenomena, there exists older work which supports the area.

The roots of adaptive compilation can be traced back to optimizations based on profile information. Profiling consists of executing a program on a representative input program or set of programs and recording control-flow information to assist in optimization [9, 23]. Originally profile information was used to assist programmers in hand-optimizing their programs, but more recently it is typically integrated into compilers to assist with scheduling and classical optimization [12, 22].

The ATLAS system uses techniques similar to adaptive compilation in the generation of high-quality linear algebra kernels [44]. Traditionally, the basic linear algebra
subroutines (BLAS) are hand-optimized for each architecture to ensure excellent performance. ATLAS provides equivalent BLAS performance without requiring repeated hand-optimization. Instead, an expert user produces a single parameterized version of each kernel. When the ATLAS system is installed on a specific platform, a large number of experiments are run to determine the best parameters for each kernel. When trying to produce codes of extremely high quality, the ATLAS system can be viewed as an intermediate step between optimizing by hand and using an adaptive system. Hand optimization requires large amounts of programmer time, but little, if any, compilation time. ATLAS reduces the amount of time required of expert users, but takes several hours of exploratory compile time at install time. A successful adaptive system for the same problem would not require the time of a programmer, but might require even more computing time to find a high-quality executable.

Cooper, Scheilke and Subramanian make one of the earlier entries into what could be considered a true adaptive compilation system [17]. They use a genetic algorithm to determine which optimizations should be performed within a compiler, and in what order, to minimize executable size. Performing the same code transformations in varying orders can produce executables with different performance characteristics. They explore different compilation orders using a genetic algorithm to discover a sequence of optimizations that produces smaller executables. While exploring compilation orders, they observed that different sequences of optimizations could produce better results with regards to execution time than the traditional sequence of optimizations. They also demonstrated that the best sequence varies between programs. These results hint at adaptive compilation’s promise and illustrate the program-specific nature of adaptive compilation for the sequence-finding problem.

The exploration of the sequence-finding problem has continued in the work of the Rice group [18, 5]. They demonstrate the complexity of adaptive compilation in general and the determination of the best possible sequence of optimizations specifically. The search space of optimization sequences is immense and cannot be completely
explored. They present several methods for partially exploring the search space and show how their results improve upon traditional methods. They are still examining whether their results can be used to produce more general rules and various methods for making their techniques more practical.

Kulkarni et al. have performed similar work using genetic algorithms to determine optimization order [32, 31]. The genetic algorithm that they use is quite similar to Cooper et al. [17]. Their work is different from that of the Rice group in two key ways: they find optimization sequences on a function-specific basis and they focus on minimizing the number of versions of the program that need to be evaluated. The reduce search times an average of 65% by only executing codes if a functionally equivalent code has not already been executed. It is important to note that several of their techniques do not work in conjunction with interprocedural optimizations.

Dr. Options is a system that addresses the related problem of determining the best set of compiler options for the PA-RISC compiler [24]. Dr. Options recommends which compiler optimizations should be enabled for a specific program to give optimal performance. Dr. Options uses a variety of heuristics that rely on data-flow analysis, profile information and details provided by the user to recommend whether or not different compiler options should be enabled. Dr. Options does not iterate like the Rice work, but is still an adaptive system which adjusts parameters to its input through the use of profile information.

Triantafyllis et al. present an approach to determining the correct set of compiler options that falls between the work of the previous two groups [42]. They use a representative set of codes to determine which optimizations tend to be the most profitable and to discover correlations between different optimizations. When optimizing a specific program, frequently-called functions are subject to an iterative compilation process that is guided and constrained by the information gathered from a representative data set. This work shows how understanding of the problem can make adaptive compilation more practical; they can quickly find a good solution by biasing their
search towards solutions that perform well on the reference set. However, their work deals with a limited initial set of parameters, and there may be difficulties expanding it to deal with the massive search spaces seen in other work.

Kisuki, Knijnenburg, and O'Boyle apply adaptive techniques to the selection of blocking sizes and unrolling factors for scientific codes [28, 29]. They use a variety of search techniques to discover a good blocking size and unrolling factor. They achieve good performance compared to unoptimized code, but do not compare against code that has undergone traditional high-level transformations. Adaptive compilation is an expensive approach that should only be performed when it performs better than less-expensive traditional methods.

Zhao, Childers, and Soffa take an analytical approach to solving the same problem addressed by Kisuki, Knijnenburg and O'Boyle [47]. They develop models to estimate the impact of different loop optimizations on cache performance for individual programs. They combine their models and use them to perform transformations only when they are deemed profitable. This results in improved performance over the previous approach of always applying the transformation; it has a much lower compile-time cost than adaptive compilation. However, the models are specific to the effect that is targeted. It is not yet clear how much of the analysis carries over to other optimizations.

Stephenson et al. use genetic algorithms to evolve heuristics for several compiler optimizations [41]. They search for better general-purpose heuristics instead of attempting to produce different heuristics for individual input programs. The programmer provides a set of parameters that they believe are important to the optimization being addressed. A genetic algorithm then tries combining the parameters in a variety of different heuristics and evaluating them on a suite of test programs. They produce a new heuristic for hyperblock formation that performs nine percent better than the previous heuristic on average, but success with other optimization heuristics is limited. Their approach could be used to successfully tune heuristics for specific
applications though it would be expensive. Much of Stepenson et al.'s work builds on the earlier work of Motwani et al. [35] who evolved a heuristic for simultaneously performing register allocation and instruction scheduling.

Despite the recent spate of activity in adaptive compilation, the area is still one with more questions than answers. The majority of current research focuses on exploring a select number of features that are exposed by current compilers. Future research needs to investigate the construction of complete adaptive systems including how a compiler can be built to facilitate adaptive compilation. This issue is the primary focus of my thesis work. A deeper understanding of how to efficiently explore the enormous search spaces created by adaptive systems is also necessary.

2.2 Procedure Inlining

Procedure inlining is the process of replacing a call site with a copy of the procedure body. Obvious benefits of inlining include eliminating the overhead of procedure calls and increasing the scope of local optimizations both of which can result in a faster program. The primary drawback of inlining is increased executable size and compilation time. One of the first references to procedure inlining occurs within Allen and Cocke's 1972 survey of optimizing transformations [4]. They discuss several different methods for linking procedures, one of which is an open linkage that involves substitution of the procedure body into the call site. Knuth's study of Fortran programs mentions the potential benefit of an open linkage when a procedure call occurs within an inner loop [30].

Over the past 30 years, there has been a great deal of research into procedure inlining. A large number of papers have discussed how to implement inlining, when it should be performed, and its impact for different languages and on different platforms. However, despite all of this research, there is still no consensus on how and when inlining should be performed.

Scheißler's work is one of the first to present an algorithm for guiding inline sub-
stitution [37]. His research focused on the benefit of eliminating the overhead of a procedure invocation as opposed to the ability to increase the scope of optimization. His algorithm greedily inlined the most frequently called procedures while not exceeding certain program and procedure size limitations. Scheifler observes only a minimal improvement in execution time, but mentions that procedure inlining could facilitate other optimizations and allow the construction of programs from many small procedures without loss in performance.

Procedure inlining is also performed in Hecht’s SIMPLE-T compiler with similar results [27]. The SIMPLE-T compiler inlines procedures with a single call site and no special properties. Hecht reports a 20% reduction in procedure calls and a one to two percent reduction in static instruction count.

The Experimental Compilation System (ECS) built at IBM research introduced the methodology of the General Purpose Optimizing (GPO) compiler and was one of the first compilers to place a heavier emphasis on inline substitution [3, 26]. The ECS was designed to allow high-quality compilation of a variety of large programs that could be written in PL/I, ALGOL, COBOL, or FORTRAN. Procedure inlining was a critical part of the compiler, since many input programs could consist of a large number of smaller procedures, and inline substitution could increase the scope of local optimizations.

Ball conducted a more thorough examination of how inlining can improve the efficacy of other optimizations [8]. He observed that procedure inlining can expose more opportunities for constant propagation and dead code elimination when some parameters passed to the procedure are constant. He used a heuristic to estimate how much each procedure benefited from inlining in the context of constant propagation and dead code elimination, and used these results to guide inlining decisions. Previous work had used inlining as an enabling transformation, but Ball was the first to use these benefits to guide inlining instead of just relying on their occurrence.

Chow examined the benefits of procedure integration in conjunction with his
UOPT optimization tool for the S-1 computer [13]. He believes that inlining offers several major benefits to optimization and register allocation with only minor drawbacks. His beliefs are supported by results which show that code which has been both inlined and optimized performs on average over fifteen percent better than code that has only been optimized.

Davidson and Holler developed the INLINER tool, a program that performed source-to-source inlining of C programs [19, 20]. They describe the difficulties of performing inlining at the source level and how they may be overcome. The INLINER tool was used as the base system for our inlining tool. The source-to-source nature of their tool allowed them to make observations generalized to several different platforms.

Davidson and Holler restricted inlining to limit register pressure – they never inlined a procedure if all of the variables tagged with the REGISTER keyword could not be given a physical register. This produced both faster and smaller programs than performing unrestricted inlining. Their specific algorithm has been antiquated by the advent of more complex register allocators and the diminished use and importance of the REGISTER keyword, but it still underscores the importance of considering the impact on register allocation when making inlining decisions.

Finally, Davidson and Holler investigated the belief that the increased code size resulting from inlining could decrease performance. When performing unrestricted inlining they never observed severe code explosion or degraded program performance due to larger executables. ¹ They examined cache and memory system performance for both inlined and uninlined codes and found that performance was often slightly better for the inlined versions. However, they did note that inlined code could sometimes run slower than the original by increasing the overhead of procedure calls on frequent paths.

Cooper, Hall, and Torczon also noted that inline substitution can sometimes have

¹It should be noted that we have observed extreme code explosion in today's large object-oriented programs.
negative effects. They examined the interaction of inlining with optimizing FORTRAN compilers [15, 16]. Inlining was performed on several scientific codes and the resulting code was optimized using different commercial compilers. The results were compared against code that had been optimized, but not inlined. They observed inconsistent improvements and degradations from inlining across both applications and compilers. They discovered that inlining can have detrimental effects such as increased register pressure and more floating point interlocks that counteract and overcome the traditional benefits. They also noted that optimization helps to mitigate the growth in object code size due to inlining.

Cooper, Hall, and Kennedy also investigated procedure cloning in a way that relates to procedure inlining [14]. They observe that the performance overhead from procedure calls comes not just from the direct cost of the call site, but also from the limitations it places on optimization. They present an algorithm for procedure cloning to improve optimization; however, their methodology and results should also be considered when investigating inlining.

Richardson and Ganapathi perform a comparison of interprocedural analysis and inlining, concluding that inlining is a better technique as long as it is practical considering executable size and compile time constraints [36]. The view of interprocedural analysis and inlining as strictly competing techniques is misguided – they both have situations where they are appropriate and beneficial. They also fail to consider some of the potential benefits of interprocedural analysis due to limitations of their compilers, for example, the improvement in single-procedure constant propagation from MOD analysis [25]. However, Richardson and Ganapathi are correct in observing that in many cases procedure inlining can provide some of the benefits that can be gained from interprocedural analysis. They also observe that the two primary benefits from inlining for their test cases occur from the elimination of call overhead and improved register allocation.

Hwu and Chang discuss inline function expansion for C programs guided by profile
analysis [33]. They use profile information to build a weighted call graph that indicates both how many times each procedure is called and how frequently each call site is visited. They begin inlining the most frequent call sites as long as they do not cause control stack overflow until a specific increase in instruction size is reached. They achieve good results in reducing the number of dynamic calls executed in the final program while limiting the growth in code size. They do not examine the interplay of inlining with other optimizations. Chang, et al. extend this work and discuss many of the implementation details [11].

Dean and Chambers try to determine the benefits of inlining procedures by experimentally inlining the routine [21]. They consider the cost of inlining to be the increase in code size and the benefits to be the number of instruction executions that can be eliminated after inlining. They determine how many instructions can be eliminated from a procedure at a specific call site by actually inlining the call and performing optimization. They then create a database entry so if the procedure is called again under similar circumstances a simple lookup can be performed. They do not examine how inlining a call site affects the surrounding section of code.

Manuel Serrano gives a very good overview of previous approaches towards inlining in “Inline expansion: when and how?” [38]. He also presents his own algorithm for inlining, implemented in a Scheme compiler, which inlines a procedure if it is less than a certain multiple of operations of the call itself. As the algorithm recurs inside procedures, this factor decreases to limit code growth, but the interaction of inlining with optimizations is not considered.

Zhao and Amaral developed an adaptive inliner for the Open Research Compiler [48]. Their inliner is not 'adaptive' in the adaptive compilation sense. They classify input programs as small, medium, or large and apply a less restrictive heuristic for the selection of procedures to inline to smaller programs. This performs better than the use of a single heuristic for all programs. This result is encouraging for the success of a true adaptive inlining system.
There has been a great deal of work dealing with procedure inlining over the past thirty years, but there is little consensus on inlining. Researchers have shown procedure inlining to be both a technique of great promise and limited opportunity. Many different methods for selecting call sites to inline have been presented. These varying results are produced because inlining is a technique where the efficacy and proper approach depend heavily on the input program, underlying compiler, and architecture. This makes inlining an ideal optimization to subject to an adaptive system.
Chapter 3

Investigating Adaptive Compilation using the MIPSpro Compiler

3.1 Introduction

Adaptive compilation is the process of iteratively compiling a program, evaluating its performance, and using the results of the evaluation to guide future compilations. The previous chapter presented prior work that demonstrated that adaptive compilation can improve program performance by finding a good set of optimizations to apply and an order in which to apply them. However, optimization selection and order are not the only aspects of compilation that can be controlled effectively by an adaptive process. A goal of our research is to demonstrate that adaptive compilation can be successful in controlling other aspects of compilation. We investigate if an adaptive system can use the parameters exposed by current commercial compilers to yield improved performance.

This chapter focuses on a specific experiment in adaptive control of compilation: finding good loop blocking sizes for the MIPS R10000 processor. The experiment harnesses the MIPSpro compiler into an adaptive system. The adaptive control chooses blocking sizes; it uses command-line parameters to the MIPSpro compiler to enforce those choices. With feedback-driven iterative refinement, the adaptive system finds the best blocking factor for a particular input code and a particular problem size. To evaluate the effectiveness of the adaptive system, we compare its performance on the benchmark program DGEMM against the performance of the equivalent routine in the ATLAS library. The ATLAS version chooses a blocksize dynamically as a function of input problem size; it was produced by a team of experts in tuning linear-algebra
codes. The feedback-driven system is able to produce results close to those obtained by ATLAS, without the investment of time by domain-specific experts.

Two critical factors led us to choose the MIPS R10000 and its compiler. First, the MIPSpro compiler produces, in general, high-quality code for the R10000. Other compilers that we tried could not come close to the performance of the compiled and distributed ATLAS library codes—a necessity for fair comparisons against ATLAS. Second, the MIPSpro compiler provides command-line flags that allow the user to control blocking sizes, which greatly impact the performance of scientific codes. The combination of a strong base compiler and command-line flags that let the adaptive system control blocking make the MIPSpro compiler a good choice for this study. We used this setup to examine the impact of user-level options on the effectiveness of the MIPSpro compiler. We evaluated performance using linear algebra kernels and compared the results against kernels optimized using the ATLAS system.

Our experiments show that the baseline MIPSpro compiler does a good job competing with the ATLAS system for smaller data sets. As array sizes grow, the effectiveness of the standard compiler’s loop-blocking algorithm deteriorates. Adaptive choice of blocking sizes with the MIPSpro compiler produces results that are close to those of ATLAS even for larger data sets. These results are achieved through the variation of a single option to the compiler – blocking size.\textsuperscript{1} Our adaptive technique rapidly finds a good blocking size while only exploring a small subset of the set of all reasonable values. During our experimentation, we also discovered shortcomings in the MIPSpro compiler’s mechanism for controlling blocking size that motivated the work in Chapter 4.

This chapter describes both the blocksize experiments on the MIPS machine and our conclusions. The next section describes ATLAS and related work. Section 3.3 explores the potential of varying optimization decisions using a single target code. The

\textsuperscript{1}In our experiments, adjusting other command-line parameters to the MIPSpro compiler had no consistent, measurable improvement on DGEMM, beyond the obvious ones, such as \texttt{-g} versus \texttt{-o3}.\textsuperscript{
data we gathered drove development of an adaptive technique discussed in section 3.4. We conclude in section 3.5.

3.2 ATLAS and Other Work

A beginning motivation for our work is the ATLAS system [44]. ATLAS tries to achieve hand-coded performance for linear algebra kernels on different processors without a programmer having to modify the code for each processor. Each kernel is modified and parameterized once for all processors by a programmer. Then, when the system is installed on a particular machine, experiments are run to determine the proper parameters for the kernel. The performance of the ATLAS kernels may sometimes fall short of hand-optimized versions that take advantage of special features on a specific processor. However, ATLAS will often outperform hand-coded kernels since they are rarely rewritten for each possible processor configuration.

Hand-coded programs achieve excellent performance at the expense of a large amount of human time to optimize the program. ATLAS trades off some of this human time for processor time and still delivers high-quality executables. Our approach attempts to take this tradeoff a step further. ATLAS still requires a developer to examine and optimize each kernel included in the system. Our system uses adaptive compilation to replace both the processor-independent hand tuning and the processor-dependent automatic tuning done in ATLAS. The ATLAS-optimized kernels provide a good comparison point for our work and help us to determine if we can use additional processor time to replace hand tuning without significantly harming performance.

Yotov et al. modify the ATLAS system to determine proper parameters using a model-driven approach instead of running experiments [46]. This substantially reduces the CPU time necessary to install ATLAS on a target architecture and provided comparable performance on two of the three machines tested. It does not decrease the amount of hand tuning required for each kernel.
Knijnenburg et al. investigate automatic choice of blocking sizes [29]. They examine the interaction of blocking size and unrolling factor to find an ideal combination, using an adaptive-compilation scheme. Their work uses source to source transformations as opposed to adjusting compiler parameters. We expand upon their work by examining how iteratively determining blocking size compares with the ATLAS system and the default algorithms in the MIPSpro compiler and by placing it in the context of general adaptive compilation. In our experiments, adaptive selection of unrolling factors in conjunction with blocking size never improved upon the results achieved by letting MIPSpro select the unrolling factor automatically.

Profile-driven optimization also executes code on sample input to gather information and improve performance. However, it uses that knowledge in a very different way than do our adaptive compilers. Profiling instruments code and then executes it in order to provide a more detailed picture of how control flows through the program [9, 12]. This information is then used to provide more accurate analysis to the optimizer. In contrast, adaptive compilation uses the results of execution as feedback for choosing parameters and optimizations in subsequent compilations. Profiling usually executes only a single version of the program to provide information for optimization, while adaptive compilation repeatedly optimizes and executes code to iteratively enhance performance. Profiling uses the results of execution to identify control-flow patterns in the executing code. Adaptive compilation uses the results of execution to measure the effectiveness of particular optimization strategies and parameters.

3.3 Adjusting Blocking Size

Proving adaptive compilation profitable for the MIPSpro compiler requires demonstrating that adaptively chosen compiler parameters can yield better executable performance than the standard compiler. We evaluated the compiler’s adaptability by examining the performance impact of various options on kernels from the basic linear
algebra subprograms (BLAS). Since these kernels are scientific programs, we focused on parameters that affected high-level loop transformations. Initial experimentation showed that several parameters impacted performance, but only one, blocking size, resulted in better performance than the default compiler when varied. Therefore, we further examined the effects of adjusting blocking size.

Blocking is a memory hierarchy optimization that reorders array accesses to improve temporal and spatial reuse [1, 45]. Blocking can be targeted at any level of the memory hierarchy, but is most frequently targeted at reducing cache misses. The reduction of cache misses allows blocking to vastly reduce the running time of scientific codes. The MIPSpro compiler automatically performs blocking when the loop nest optimizer is invoked. The compiler allows the user to manually specify the blocking size, overriding its own choice.

To determine the potential benefits for adaptively determining blocking size, we explored the effects of various block sizes on running time. We exhaustively examined the performance of the BLAS kernel DGEMM, a general matrix-matrix multiply routine, for different block sizes and varying array sizes. We first modified the DGEMM kernel by hand to eliminate a single conditional that resided within the multiplication loop nest. The conditional prevented multiplication of rows of the matrix by zero elements. The check was unnecessary for correctness and hurt performance on dense matrices by preventing the compiler from performing any high-level optimizations on the loop nest. Experimentation suggests that this conditional was also removed from the ATLAS DGEMM kernel.

The MIPSpro compiler was run using blocking sizes from one to one hundred, squared. We compared the results against the automatically-blocked code as well as a version of DGEMM tuned by the ATLAS system. All tests were executed on a 195 MHz MIPS R10000 with 256MB memory, a 32 KB L1 data cache, and a 1 MB unified L2 cache.

Figures 3.1, 3.2, and 3.3 show the running times of the different methods for
Figure 3.1: Running times for $500^2$ arrays

Figure 3.2: Running times for $1000^2$ arrays

Figure 3.3: Running times for $1500^2$ arrays
various square array sizes. The results for the ATLAS system and the MIPSpro compiler using internally determined blocking are straight lines, since they do not vary the blocking size. The results of compiling using the MIPSpro compiler and manually selecting a blocking size vary substantially. Selecting a poor blocking size can cause substantially worse performance. In addition, as array size increases the range of good blocking sizes decreases and the penalty for selecting a bad blocking size is magnified.

The data from these figures also suggests that the performance of the MIPSpro compiler with tuned blocking size can remain close to the performance of the ATLAS system as array size increases, while performance of the standard MIPSpro compiler deteriorates. This trend is better shown in figure 3.4. The performance of the standard MIPSpro compiler diverges from the performance of ATLAS and the MIPSpro compiler with adaptively tuned blocking after the array size reaches 800 squared. This can also be clearly seen in figure 3.5, which shows the relative running times with respect to the ATLAS system. After the array size reaches 900 squared, compiling using the standard MIPSpro compiler blocking scheme results in more than twice the running time obtained using ATLAS. In contrast, tuning to find an ideal blocking size keeps results within 10 to 15 percent of ATLAS.
The reason behind the poor performance of the standard MIPSpro scheme quickly becomes evident when examining figures 3.6 and 3.7. When DGEMM is compiled with the standard MIPSpro compiler, it incurs significantly more L1 cache misses than with either ATLAS or a tuned blocking size. The standard MIPSpro compiler uses a rectangular blocking scheme that grows as array size increases. When the array size becomes too large, the block size becomes too large for the L1 cache, and there is a drastic increase in the number of cache misses. Cache misses, however, do not explain the performance difference between the MIPSpro compiler with tuned blocking and ATLAS. Tuned blocking has fewer L1 and L2 cache misses than ATLAS, but has worse performance. This difference is probably due to special tuning performed by ATLAS specific to the DGEMM kernel.

We also examined the performance of DGEMM on non-square matrices. DGEMM multiplies an M by K matrix by a K by N matrix to yield an M by N matrix. In our previous experiments M, N and K were always equal, but we also tried holding two of the three dimensions to a value of 1000 and varying the third dimension from 500 to 2000. The results of these additional experiments can be seen in figures 3.8, 3.9, and 3.10. There are some minor differences from the results of the square matrices; ATLAS performs worse than expected when the K dimension is larger. The
Figure 3.6: L1 cache misses for square matrices

Figure 3.7: L2 cache misses for square matrices

Figure 3.8: Running times for varying M
general trend, however, remains the same. The DGEMM executables produced by the MIPSpro compiler are substantially slower than their ATLAS counterparts, while adaptively tuning the blocking size allows executable performance to remain close to ATLAS levels.
3.4 Determining Blocking Size

To realize the promise of adaptive compilation, we must not only show that it can improve performance, but also demonstrate that we can find an excellent parameter with a reasonable amount of work. The previous section shows that correctly setting the parameter for blocking size can lead to better running times for the DGEMM kernel. However, it does not discuss how the blocking size that produces these results can be determined. This section examines how to quickly determine the appropriate blocking size for a program.

Examining every potential blocking size, as done in the previous section, is rarely necessary to find the size that produces the best results. Instead, the space of potential blocking sizes can be explored intelligently. Examining only a few blocking sizes and using those results to choose future sizes to examine allows the best blocking size to be determined in a fraction of the time that an exhaustive search would require.

Our approach begins by finding the result for a blocking size of 50. It then begins sampling higher and lower block sizes in increments of ten as long as the results stay within 10 percent of the best results seen so far. After this stage is complete the area around the best result found is examined in detail: the five block sizes larger than the best result, and the five block sizes smaller than the best result are all examined. Of these eleven block sizes, the one with the best running time is chosen.

When this algorithm was tested on the DGEMM benchmark it always resulted in selecting the best possible blocking size. The amount of time required to determine the blocking size compared to an exhaustive approach can be seen in figure 3.11. A more aggressive approach could determine a good blocking size in even less time, trading less compile time for slightly more execution time. However, this is contrary to the adaptive compilation approach, which believes in using additional compile time to improve executable performance.

Expending CPU time to determine the correct blocking size parameter may also not be necessary for each program compiled. Since the performance of blocking is
based on the dimensions of the arrays and not the values contained within, the ideal blocking size needs to be determined only once for each set of dimensions. The first time a program is compiled with a specific set of dimensions the correct blocking size can be determined and stored in a table. Whenever subsequent programs are compiled with the same array dimensions the blocking size parameter can be retrieved from the table.

3.5 Conclusion

Our investigation of adaptive compilation using the MIPSpro compiler shows the potential benefits of adaptively tuning parameters. Experiments revealed that adaptively selecting the appropriate blocking size for the DGEMM kernel provides performance near the level of the ATLAS system. In comparison, the performance of the standard compiler drops off considerably for larger array sizes. A good blocking size can also be found quickly by intelligently exploring only a small portion of the search space once for each array size.

The MIPSpro compiler, however, is not sufficient to make adaptive compilation a generally applicable technique. Many heuristic decisions are made during optimiza-
tion, but these decisions are not accessible to adaptive techniques. The success of adaptive compilation on a wide range of applications will require the design of compilers that expose a carefully selected set of parameters that can significantly alter performance in different ways.
Chapter 4

Building an Inliner for an Adaptive System

4.1 Introduction

The adaptive compilation work on loop blocking in the previous chapter showed some of the potential of adaptive compilation, but also presented many challenges and questions. Current compilers fail to expose enough performance-changing parameters to make adaptive compilation successful on a wide range of input codes. In addition, as the parameterization of compilers increases, adaptive compilers must find a way to quickly and efficiently explore the growing search space. Further progress in adaptive compilation requires the development of highly-parameterized optimizations with complex search spaces.

To continue this research, we built an adaptive inlining system. Procedure inlining, the process of replacing a call site with a copy of the called procedure's body, makes an excellent optimization to explore in an adaptive context for two reasons. Inlining is capable of greatly reducing the running time of programs by eliminating the overhead of procedure calls and increasing the scope of optimizations. However, despite a great deal of research in the area, there is still no consensus on how to best determine which procedures to inline.

There are also two practical advantages to examining procedure inlining. The construction of a source-to-source inliner allows us to examine adaptive techniques without building an entire compiler. Furthermore, a source-to-source system allows us to examine adaptive inlining on a variety of architectures with several different underlying compilers. Since part of the motivation for our work is to show the general applicability of adaptive compilation, this approach is of great benefit.
The first step in building an adaptive inlining system is the construction of a procedure inliner that exposes a variety of ways to make inlining decisions. We modified an existing source-to-source inliner [19] to handle ANSI C and to accept condition strings that determine the call sites to be inlined. These condition strings allow a variety of program properties to be used in guiding inlining, and allows them to be combined in many ways. We explain how the underlying inliner was designed and demonstrate that it exposes sufficient opportunities for an adaptive inlining system in this chapter.

4.2 Background

Procedure inlining alters code performance in several ways. The direct benefit of procedure inlining is eliminating the overhead of making the call. When a procedure is called parameters must be evaluated and transferred, registers must be saved, an activation record must be created, and control must jump to the called function. There is additional overhead when returning from the function: another control transfer, restoring register values, deleting the activation record, and passing the return value. The process of inlining eliminates this overhead.

Inlining's other major benefit is improving the efficacy of other optimizations. Many compiler optimizations work on an intraprocedural scope. Inlining produces larger procedures and increases the area that can be analyzed and transformed by these optimizations. Dead code elimination, constant propagation, and redundancy elimination are just some of the transformations that can benefit from inlining [2, 6, 10, 43].

The primary disadvantage to inlining is the increase in code size that arises when a single procedure is converted into many segments of inlined code. It has been suggested that large executables produced by inlining will suffer performance degradation due to poor instruction-cache performance, but researchers have thrown doubt on this assertion for certain systems [20]. However, large code is in itself undesirable:
many applications do care about executable size, and since code growth increases compilation time, compilation may become infeasible.

There has been a great deal of prior work on inlining which is discussed in detail in section 2.2. Procedure inlining has been investigated on a wide variety of architectures for many different programming languages. The heuristics used have focused on eliminating overhead, enhancing other optimizations, or minimizing code. Even algorithms with a similar focus vary greatly in the details of their approach. The results of all of this research have shown that inlining is often a profitable technique, but that it sometimes has little impact or even a negative effect. Unfortunately, the research has not found clear heuristics to predict when inlining is profitable or how to apply it in specific situations for maximal benefit. This makes inlining an ideal problem to be addressed using adaptive methods.

4.3 Building a Flexible Inliner

Our source-to-source inliner was built from the Davidson and Holler's INLINER tool [19]. The INLINER tool performs source level inlining of Kernighan and Ritchie-style C code. Inlining decisions are guided by concerns of register pressure; a procedure is never inlined if inlining would create a procedure with more variables marked with the register keyword than the number of physical registers on the machine. Modifying the INLINER tool to be suitable for adaptive inlining required two major changes.

The first change was updating the INLINER tool to accept ANSI-style C. Almost all modern C code is written using the ANSI style, and the adaptive inliner needs to accept ANSI C code so it can be evaluated on current benchmarks. The actual conversion of INLINER code to accept ANSI-style code, though laborious, is of limited intellectual interest.

The second change to the INLINER tool was simpler in many aspects, but of much greater interest from a research perspective. The method that determines the
call sites to be inlined was modified to be suitable for adaptive inlining. The original INLINER tool made inlining decisions based on a single heuristic. The goal of the new method was to provide extremely fine granularity and great flexibility in determining which procedures to inline.

The obvious method to provide flexibility in the specification of call sites for inlining is to directly specify call sites for inlining: Give all of the call sites in a program distinct labels and then specify a set of labels at the command line. Unfortunately, there are two major problems with such a solution.

The first problem is that the set of call sites in a program does not remain constant throughout the process of procedure inlining. Whenever a call site is inlined, new call sites can potentially be created. If procedure A calls procedure B which in turn calls procedure C, inlining the call site of B within A will create a call to procedure C within A. This problem can be quite complex on programs with large, deep, call graphs. This makes it essentially impossible to directly label all of the call sites in a program and specify them for inlining at the command line.

In addition to the mechanical difficulties of directly specifying call sites for inlining, it is also undesirable from a denotation standpoint as well. Specifying call sites directly does not allow any higher meaning to be gathered from inlining decisions. For example, if we find a good set of procedures to inline for a specific program there is no way to take the solution and try applying it to a different program. This could also result in a very difficult search space for the adaptive system.

Therefore, we chose to specify call sites for inlining indirectly. We did this by specifying values for different properties that meaningfully distinguish call sites which must be met for inlining to occur. This is done at the command line through the use of condition strings. Condition strings specify the criteria which must be met inline a call site. A condition string can consist of multiple individual conditions. Each condition can contain multiple call site properties, constants, basic arithmetic, and a comparison operator ($<$, $>$, or $=$). All of the conditions in a condition string are
combined in disjunctive normal form to allow full expressibility.

To evaluate a condition string, the inliner first makes a pass over the whole program, gathering statistics about the code. These properties, in turn, become values that the condition string can reference to make inlining decisions. The set of call sites in the input program that satisfy the complete condition string are inlined. Inlining decisions are made beginning with procedures located at the leaves of the call graph and proceeding toward the root, with procedure properties updated as decisions are made. Therefore, the inlining decisions are not independent; inlining one call site can impact whether or not another call site is inlined. The following example invocation of the inliner would integrate all procedures inside foo.c which are fewer than 25 lines long or where the call site occurs within a loop and the procedure is fewer than 100 lines long:

```
inliner -C 'sc < 25 | lnmd > 0, sc < 100' foo.c
```

### 4.3.1 Condition String Properties

The availability of a large number of call site properties that can be referenced in the condition string is critical to the success of adaptive inlining. Adaptive compilation should benefit from being able to explore a large search space of possible sets of decisions with different performance characteristics. Therefore, having many properties which can be used to make inlining decisions gives the adaptive system greater flexibility and potential for improvement. The downside of a large number of parameters is that the search space becomes larger, so exploring the space can become more complex and time intensive. The addition of parameters with no relation to making profitable inlining decisions is therefore undesirable. Finding a good set of properties for the inliner to consider is an important goal of our research. Thus, we would prefer to add parameters of questionable value than accidentally omit valuable parameters. We can deal with potential complications of the search space when designing the adaptive system.
Properties which can be used within the inliner's condition string are listed below.

**Statement count - sc:** A procedure's statement count is an obvious program property to consider when making inlining decisions. Statement count is the number of C statements contained within a procedure and represents, roughly, the size of the procedure. Procedure inlining often increases object code size. One common approach to inlining is to only integrate small procedures, since this minimizes code growth, and small procedures frequently have the most to gain from an increased scope of optimization.

**Loop nesting depth - 1nd:** The loop nesting depth property returns the number of loops that surround the call site within the calling procedure. Optimizing compilers often place greater emphasis on improving the quality of code within loops under the assumption that code within a loop nest executes more frequently. Therefore, it may be desirable to inline call sites within loops, because the call site will be executed frequently, magnifying any benefits of inlining.

**Static call count - scc:** The static call count is the number of distinct call sites where a procedure is invoked before any inlining occurs. If the static call count of a procedure is small, each call site can be integrated and the original procedure eliminated. In particular, a procedure with just one call site can be inlined with little or no increase in code size. Procedures with a low static call count can consequently be inlined with fewer disadvantages. However, since our inliner can perform inlining on a per call site basis, procedures with many call sites can still be inlined where necessary.

**Parameter count - pc:** A procedure's parameter count is the number of variables that are passed into that procedure. Procedures with a high parameter count have a greater invocation cost due to the cost of passing parameters. In some cases, the cost of calling a small procedure with many parameters can be greater than simply executing the procedure inline [38]. The cost of passing parameters can
be especially high when conversion is required; for example, \texttt{short} parameters in C.

\textbf{Constant-parameter count - cpc:} Constant-parameter count is the number of parameters that are constant at a given call site. Procedure calls with constant parameters can often benefit more from other optimizations after inlining. Inlining in this case has a similar effect to procedure specialization. This parameter provides an estimate of the additional optimization opportunities which will be exposed if the call site is inlined. This is similar to the approach of Ball's work [8].

\textbf{Calls in procedure - clc:} Calls in procedure is the number of procedure calls that are made from within the procedure that is a candidate for inlining. This parameter was introduced as a method for detecting leaf procedures. Leaf procedures are good candidates for inlining for two distinct reasons; they are often small and easily inlined, and a majority of the total execution time often occurs in leaf procedures.

\textbf{Dynamic call count - dcc:} Dynamic call count is the number of times that a call site is executed during a profiling run of the program. Inlining a frequently executed call site has more potential for improving program performance than does inlining an infrequently executed call. Call sites with an extremely high dynamic call count can provide a substantial performance benefit when inlined, simply through the elimination of the procedure call overhead. Previous research has shown the potential of using profile information to guide procedure inlining [33, 12].

One advantage of having this type of flexible parameter system is that we can duplicate the algorithms used in previous studies on inlining. This allows us to compare previous algorithms against each other and with our adaptive technique.
4.3.2 Gathering Dynamic Information

Evaluating condition strings requires determining the value of the various condition string properties for each call site. The appropriate value for most of these properties can be determined using simple static analysis inside of the inliner. However, determining the dynamic call count of a call site requires profiling the program.

We collected profiles using the gcov utility. gcov is a tool used to provide coverage and profiling information in conjunction with gcc. The process begins with the compilation of the program using the flags -g, -pg, -fprofile-arcs, and -ftest-coverage. The program is then executed with a training data set, which results in the generation of files containing profile information. gcov is then invoked on each of the source files in the program and uses the profiling information to produce annotated source files. These annotated source files list the number of times a source code line is executed at the beginning of each line and can be used as inputs to the inliner.

Invoking the -g option causes the inliner to accept the annotated source files as input. We modified the scanning phase of the inliner to strip off the annotations at the beginning of each line and associate the frequency counts with the appropriate procedure calls. The inliner uses this information when a condition string requires a call site's dynamic call count.

4.4 Experimental Validation of the Inliner

The previous sections in this chapter describe our construction of a parameterized, source-to-source inliner. The overall goal of this work is to design an adaptive system that can find a good set of inlining decisions for a specific program. Accomplishing this requires the construction of an adaptive harness for the inliner which repeatedly invokes the inliner with a variety of condition strings to discover one with the desired performance. However, for this to be successful the space of inlining decisions exposed by our inliner must contain high-quality solutions, and we must have a method for
finding these solutions.

In this section, we experimentally examine the potential of our inliner with two goals in mind. First, we want to validate the inliner’s design and demonstrate that high-quality sets of inlining decisions can be found with our high-level condition string specification. Second, we seek a better understanding of the space of inlining decisions to aid in the construction of the adaptive system. We accomplish this by exploring the space of inlining decisions. The space is too large to be exhaustively explored, so we perform partial explorations using parameter sweeps and the selection of specific points.

Exploring portions of the inlining decision search space helps us better understand inlining and guides our work on building an adaptive system. These experiments inform us how effective different parameters of the condition string are in isolation, what range of values is appropriate for the parameter, and how each parameter affects different types of input programs. This information can provide insight into why different inlining techniques have varying levels of impact in different situations. More importantly, however, we use this information to help guide our adaptive system. The space of inlining decisions that the adaptive system must explore is immense; a basic understanding of this space allows us to build an adaptive system capable of finding good condition strings in much less time. Our previous experience with finding good compilation sequences supports this notion [5].

All experiments in this section, as well as later in the dissertation, were performed on a dual-processor 1 GHz G4 MacIntosh running OS X Server. Each processor has a 256kB L2 cache and 2MB L3 cache. Benchmarks were first inlined with whichever condition string was being evaluated, and the resulting code was compiled using gcc 3.3 with -03 enabled. Inlining within gcc was disabled using the -fno-inline-functions flag. We ran each experiment individually on an unloaded machine using only one processor.

We begin by discussing our experimentation with a single benchmark, vortex, in
detail. We then present data for other benchmarks at a higher level.

4.4.1 The Vortex Benchmark

We began exploring the impact of different inlining decisions using the vortex benchmark. Vortex is an object-oriented database program that is written in C and is part of the SPEC CINT2000 benchmark suite. Vortex was the first program examined for several reasons. The object-oriented construction of vortex means that the program contains a large number of procedures. This produces more opportunity for inlining to improve performance and provides a large set of possible inlining decisions to be explored. In addition, vortex is also a relatively large benchmark, consisting of almost seventy-thousand lines of source code. This reduces the likelihood of a simple inlining solution, for example, where performance depends on inlining a single routine, and ensures that code growth is an issue. Finally, using a well-recognized benchmark like vortex gives our experiments legitimacy and makes comparisons against others’ work easier.

Vortex was first tested without inlining, using gcc's inliner, and with a couple of standard heuristics to obtain baseline data about the benchmark’s performance. Table 4.1 shows the time required to compile each version of vortex as well as the size of the executable produced and its execution time. All of the results presented use -O3 optimization, but it is also important to consider the structure of the input files when comparing results. Vortex consists of 66 C files. gcc usually compiles each file individually and combines them at link time. However, our inliner examines all of the source files at once to allow inlining across files and produces a single output file. A single input file is advantageous to gcc, since it increases the scope of some optimizations and presents more opportunity for inlining. In order to guarantee that any improvements observed by our inliner come from inlining alone, we use our inliner to produce a single C file with no inlining performed for gcc. These are the results titled “single file” in the table. The results show that a single input file improves
<table>
<thead>
<tr>
<th>Version of Vortex</th>
<th>Comp. Time</th>
<th>Run time</th>
<th>Exec. size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Many files, no inlining</td>
<td>50.65s</td>
<td>22.53s</td>
<td>596kB</td>
</tr>
<tr>
<td>Many files, gcc inlining</td>
<td>52.05s</td>
<td>22.27s</td>
<td>608kB</td>
</tr>
<tr>
<td>Single file, no inlining</td>
<td>47.23s</td>
<td>20.66s</td>
<td>592kB</td>
</tr>
<tr>
<td>Single file, gcc inlining</td>
<td>465.40s</td>
<td>17.73s</td>
<td>1968kB</td>
</tr>
<tr>
<td>&quot;sc&lt;25</td>
<td>sc&lt;100, scc=1</td>
<td>sc&lt;175, lnd&gt;0&quot;</td>
<td>356.45s</td>
</tr>
<tr>
<td>&quot;sc&lt;10</td>
<td>sc&lt;200, scc=1</td>
<td>sc&lt;200, lnd&gt;0&quot;</td>
<td>219.92s</td>
</tr>
</tbody>
</table>

Table 4.1: Varying inlining results for vortex

running time due to both increased scope of other optimizations and the inliner \(^1\), but that compile time drastically increases when using the gcc inliner on a single file. It is also worth noting that inlining has little impact when applied to separate files.

The final two results shown in table 4.1 are condition strings given to our inliner. The first string represents a set of inlining conditions taken from Hall’s study of Fortran inlining [15]. The inliner integrates all procedures that meet one of three conditions: fewer than 25 statements, fewer than 100 statements and only a single call site, or fewer than 175 statements and the call is contained within a loop. This condition string produces an executable that runs slower than gcc’s default inliner. The second condition string represents a modified version of the first with different cutoff points. It was obtained through a small amount of manual experimentation and is not intended to represent an ideal tuning. The second condition string produces an executable which runs faster than both the first string and gcc’s inliner, is significantly smaller, and requires far less compile time. Two important points should be taken from these simple experiments. First, inlining can significantly increase the speed of some programs. Vortex runs in twenty-percent less time when inlined with the second

---

\(^1\)The GCC inliner does not perform cross-file inlining
Figure 4.1: Inlining and compile times for vortex

condition string versus single-file, no inlining. Second, tuning parameters can have a major impact. The simple adjustment of values between the two strings produced a smaller executable which ran ten percent faster and compiled almost 40 percent faster.

After these initial experiments, we progressed to performing parameter sweeps. We began by varying X in the condition string “sc<X”, which inlines all procedures containing less than X statements. Statement count was chosen as the subject of our experimentation because it (or the analogous line count) is used both in prior work on inlining heuristics and in current compilers [15, 40]. In addition, it is the most reasonable parameter to apply independently over a wide range of values due to the wide variation in procedure’s statement counts.

Our parameter sweep varied the maximum statement count inlined from one up to 400. The parameter sweep was stopped at 400 due to the combination of extremely long compile times and minimal additional changes in performance. The time required
to inline and compile the different versions of vortex is shown in Figure 4.1. The inliner runs quickly throughout the sweep (making the inline time hard to distinguish from the x-axis in Figure 4.1). Compilation time, however, grows dramatically as more procedures are inlined. "sc < 1" requires less than a minute to compile after inlining, while "sc < 401" takes almost 50 minutes. The increase in compilation time closely follows the increase in both source code size and executable size that is shown in Figures 4.2 and 4.3.

The similarity between Figures 4.2 and 4.3 is also worth noting. Source-code growth as more procedures are inlined is to be expected, but there has been some conjecture that this would not necessarily translate to an increase in executable size. This theory stems from the idea that much of the added source code could be eliminated by an optimizing compiler. The combination of Vortex and gcc does not show any evidence of this effect.

The impact on execution time of the maximum statement count inlined, shown in
Figure 4.3: Executable size for vortex

Figure 4.4, is the most interesting result of our parameter sweep. The graph exhibits multiple local minima, at a variety of plateaus, which indicates that a simple hill climbing technique may not be sufficient for finding the ideal value for some parameters in even simple condition strings. ² The lowest point in the graph occurs when the condition string is “sc < 141”, but “sc < 12” might be a more interesting data point, because inlining procedures containing less than twelve statements produces a fast executable and local minimum with a relatively small amount of inlining. “sc < 141” produces a good result but is probably needlessly inlining many procedures. A much smaller executable with equivalent or better running time might be constructed by using a condition string with “sc < 12” combined with other conditions that more accurately target the remaining procedures that need to be integrated.

² Recall that hill climbing, with a little local exploration, worked for blocksize choice with DGEMM on the MIPSpro compiler.

There has been speculation that excessive inlining would result in large executables
with poor instruction cache performance and consequently longer execution times. Because the executable is so large, we would have expected “sc < 141” to produce an executable that performs poorly, but this was not the case. We never observed a large increase in execution time as a result of inlining. Large amounts of inlining produces large executables which require long compilation times, but execution times did not suffer. This behavior was not observed with vortex, but that does not mean that it cannot happen, or that inlining can be exercised with impunity.

Inlining can increase execution time, and exponential code growth is a concern, for compile time if not for cache effects. Inlining Vortex with the condition string “clc < 4” took approximately half an hour and produced C file of almost 1.9 million lines. gcc was not able to compile the resulting code due to memory constraints.

Exponential explosion of code size is an important concern for an adaptive inliner, since we want to avoid explorations that fall into these black holes. Complicating the issue is the fact that good solutions and terrible solutions can exist in very close
proximity to each other in the search space. The performance of the “clc < 4” condition string was discovered by accident when we were testing the clc parameter for correctness. At the same time, we evaluated the “clc < 3” condition string. The two condition strings had the following performance:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Comp. Time</th>
<th>Run time</th>
<th>Exec. size</th>
</tr>
</thead>
<tbody>
<tr>
<td>clc &lt; 3</td>
<td>163.99</td>
<td>15.31</td>
<td>932</td>
</tr>
<tr>
<td>clc &lt; 4</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

The condition string “clc < 3” has one of the fastest execution times that we have discovered for vortex. These two cases illustrate the extreme difficulty of exploring the space of condition strings to find a good set of inlining decisions. Inlining all procedures that contain fewer than three procedure calls produces an excellent result, but when the boundary is raised to four, exponential blowup occurs. A sophisticated adaptive system is needed to adequately explore this complex search space. This example also suggests the need for a mechanism to cut off inlining based on code growth since the inliner ran for over 30 minutes.

We also performed sweeps of pairs of condition string properties to continue to gain a better understanding of the search space and its potential. We paired statement count against statement count of a procedure inside a loop, statement count of a procedure with a single static call site, and the number of constant parameters passed to the procedure. The resulting graphs can be seen in Figures 4.5, 4.6, and 4.7. The graphs continue to show several local minima and maxima as was observed in the one-dimensional sweeps, but they also display a certain amount of continuity. The graphs lack the chaotic behavior observed in our work on adaptively selecting the ordering of optimizations [5]. This suggests that, from an algorithmic perspective, exploring the space of inlining decisions may not be as difficult as finding good scalar optimization sequences.
Figure 4.5 : Sweep of statement count and loop nesting depth

Figure 4.6 : Sweep of statement count and static call count
4.4.2 Overview of other Benchmarks

We have taken a detailed look at the space of possible inlining decisions for the vortex benchmark. We have discovered that there is great potential for adaptive inlining with the vortex benchmark, and that our inliner is capable of exposing that opportunity. This section examines other benchmarks in lesser detail to confirm that these discoveries are not specific to vortex.

We performed statement count sweeps on four other benchmarks; the bzip2, vpr, and parser benchmarks from SpecINT as well as the mpeg2-encode program from Mediabench. These sweeps can be seen in figures 4.8, 4.9, 4.10, and 4.11. These graphs confirm the trends that we saw when examining vortex. The graphs display variation over parameterization that suggests an adaptive approach.

It is important to realize that the minima of each graph are distinct. The best condition strings found vary between benchmarks. Procedure inlining is not an optimization that can be solved with a unified heuristic [41]: an adaptive approach
Figure 4.8: Execution times for bzip2

Figure 4.9: Execution times for vpr
Figure 4.10: Execution times for parser

Figure 4.11: Execution times for mpeg2-encode
that finds program-specific conditions will achieve better sets of inlining decisions. The differences between benchmarks become more evident when comparing sweeps across multiple parameters. Figure 4.12 shows a sweep of statement count against statement count inside of a loop for the bzip2 benchmark. When compared against the equivalent graph for vortex, it is clear that both benchmarks display a similar amount of continuity and smoothness, but the parameter values that produce good solutions vary greatly between them.

The magnitude of change varies significantly between the graphs in this section. This is to be expected; procedure inlining can only improve performance when there is opportunity. We see great improvements in vortex because it is an object-oriented benchmark with a large number of procedures that are called during the course of the program. Inlining has limited effect on mpeg2-encode, which contains only a small number of procedures and spends most of its time in a single kernel.

Overall, the experiments in this section show the potential of an adaptive inlin-
ing system and confirm the suitability of our inliner for such a system. The inliner provides a large search space that contains high-quality sets of inlining decisions. Furthermore, good inlining decisions vary between benchmarks, necessitating an adaptive approach. These experiments also provide insight into how to design the adaptive control algorithm as will be shown in the next chapter.

4.5 Future Work

The current inliner is sufficient for use in an adaptive inlining system. It allows us to make inlining decisions based on several different properties. These properties describe different aspects of the call site and candidate procedure. The large search space that they provide can be explored using adaptive techniques and can yield significant improvements in the running time of input programs. However, there are still ways in which the inliner could potentially be improved to provide better results and the ability to achieve them more efficiently.

The primary method for improving the inliner is to increase the expressiveness of the condition strings. The condition-string system allows great flexibility in forming conditions, but is limited to the properties supported by the inliner. Additional condition-string properties would provide more ways to specify a set of inlining decisions. This could increase the likelihood of the adaptive system restricting inlining to only the performance-critical call sites.

There are several properties that could allow the creation of additional high-quality condition strings. Properties that attempt to take a higher-level approach to the benefits of inlining could be helpful; for example, an estimate of the number of instructions which become useless upon inlining a call site [8]. More complex profiling properties could also be helpful, such as detecting when a procedure has infrequently called paths. Finally, general properties might allow condition strings to make helpful comparisons. Average procedure size could be used in comparison with the statement count of a procedure.
4.6 Conclusion

In this chapter, we present the design and analysis of a procedure inliner for use in an adaptive inlining system. We develop a condition-string system for the inliner that utilizes several different program properties. This varies significantly from the traditional method of compiler parameterizations that rely on flags. The condition-string system provides a large, diverse, search space of possible inlining decisions which makes the inliner suitable for adaptive techniques. Experiments have shown that this system can produce a significantly better set of inlining decisions than a traditional inliner. The next chapter will show how the inliner is placed into an adaptive system to efficiently find good inlining solutions.
Chapter 5

Building an Adaptive Inlining System

5.1 Introduction

We presented a source-level procedure inliner, which exposes inlining decisions through the use of condition strings, in the previous chapter. Our initial experiments have shown that the flexibility of this inliner allows better results to be obtained than can be achieved using a traditional inliner. However, we have not yet presented a method for achieving these results in reasonable time. This chapter shows how we built an adaptive inlining system using our inliner.

The construction of an effective adaptive inlining system requires the understanding of several different, complex issues. We need to make decisions on how to evaluate different solutions, how to constrain the space of possible solutions, and how to explore the constrained space. These decisions were sometimes made through examination of prior work and intuition followed by experimental validation. At other times, experimentation was performed that led us to a specific approach. The methodology itself is an important aspect of this chapter. It justifies our approach, and also provides guidance in the design of future adaptive compilation systems.

We developed an adaptive inlining system that uses a randomized hill climber to consistently find good sets of inlining decisions. The hill climber explores a search space that is restricted to a fixed-order condition string and a limited number of evaluation points. Each set of decisions is evaluated using a single execution of the code to balance the search time required against timer accuracy in evaluating the objective function. Several descents of the hill climber are used to ensure good results, and limited patience can be used to find results quicker with only a minor degradation
in performance.

Our experiments show that adaptive inlining produces faster executables than either no inlining, or a standard, static inliner. The magnitude of these improvements, though significant in some cases, is not the primary result of the experiments, because the improvements are highly dependent on the specific benchmark and the opportunities for inlining therein. Instead, the results are important because they show that adaptive inlining consistently capitalizes on the potential of procedure inlining. Procedure inlining is a complex optimization that depends on a variety of factors and remains poorly understood despite significant research. Our adaptive system overcomes these issues, which limit prior techniques that use a fixed approach, and realizes the benefits that procedure inlining is capable of producing.

Sections 5.2 and 5.3 discuss our design of an objective function for the hill climber and how to limit the time spent evaluating potential solutions. Section 5.4 presents the fixed-order condition string that constitutes our exploration space. Section 5.5 develops the hill climber that explores the space and presents results that validate its efficacy. Section 5.6 examines tuning the hill climber to find results using fewer evaluations, while section 5.7 examines the properties of the good sets of inlining decisions we find. Section 5.8 discusses potential improvements to the system and Section 5.9 concludes.

5.2 Finding an Objective Function

An adaptive system uses a feedback loop to search for better solutions to the problem it is trying to solve based on previous results. This requires an objective function that determines the quality of a result obtained by the system. In the case of our adaptive inlining system, we need to be able to compare different sets of inlining decisions and determine which is better. An adaptive system cannot utilize feedback without an objective function. Consequently, the first step in building an adaptive inlining system is devising an objective function.
Traditionally, procedure inliners have tried to minimize execution time of the program while only incurring a small increase in code size. The motivation for wanting to minimize execution time is obvious, but the desire to limit code growth is slightly more complex. Code size itself can be extremely important in certain situations such as embedded applications. However, the use of code growth as a metric in making inlining decisions stems from less direct concerns.

Inliners frequently use code size as a factor in inlining decisions for two reasons. First, unbridled inlining can result in severe code growth that can make compilation infeasible. We saw this in our experiments when going from “clc < 3” to “clc < 4” with the Vortex benchmark. Extreme code growth remains a concern for an adaptive inlining system because the downside risk in compilation time can be huge. The second reason code growth is used as a metric is the belief that too much code growth results in decreased performance. However, in the previous chapter we observed that a set of inlining decisions that results in substantial code growth can also produce a faster executable. Traditional inliners limit code growth because of concerns over its impact on performance. This is not an issue for an adaptive system that can evaluate how inlining decisions impact performance more directly.

Therefore, our only concern with code size is to prevent drastic code explosion from a set of inlining decisions, thereby ensuring that the code will compile. This growth can be prevented with a cap on how much inlining is allowed. Since the only purpose of this cap is to prevent code explosion, it should be loose enough so that it does not exclude a set of inlining decisions that results in both significant code growth and improved performance. We strike from consideration any set of inlining decisions that results in the inlined source code size being more than ten times greater than the original source code size. With this restriction in place, our objective function can be concerned solely with the execution speed of the program.

Previous work in adaptive compilation has used instruction counts as the basis of their objective functions [5]. Instruction counts work well as an objective function
for an adaptive compiler that finds program-specific sets of scalar optimizations. Instruction counts eliminate the imprecision inherent in using raw running times and are accurate performance predictors for scalar optimizations which do not impact the memory system. Unfortunately, instruction counts are not suitable as the basis for the objective function in this work. Inlining can radically change the code shape and size of a program. This can impact program performance without changing the instruction counts. For example, changes in code layout, branch behavior, and procedure calls can all impact performance.

Therefore, our objective function relies uses actual running times. Using execution times for an objective function raises two concerns: the overhead incurred by executing the code, and the precision of running times as an objective function. The overhead of running the program as part of the objective function is unfortunate, but not fatal. Current adaptive systems typically execute the code as part of the objective function: systems that rely on instruction counts must still normally execute the code to gather them. Training data can be used to keep the executions short, and strategies to reduce the number of executions necessary are an area of active research, including this work. However, we need to determine if running times are reliable and if timers are precise enough to be used as the objective function for an adaptive inliner.

The accuracy and precision of the objective function for an adaptive system is critical. Adaptive systems rely on feedback provided through the objective function to discover better results over time. An imprecise objective function could cause the system to make incorrect choices, leading to the exploration of the wrong sets of inlining decisions and eventually lesser results. Typically, the accuracy and precision of running times is improved through repeated runs. Since repeated executions add significantly to the expense of an adaptive system, we need to determine how many runs are necessary to provide sufficient accuracy to the adaptive system. In addition, since each execution adds to the overhead of the system, we do not want to perform
any more executions than necessary.

We determined the number of executions necessary to ensure that our objective function is precise through experimentation. We performed statement count sweeps on three different SpecINT benchmarks using the training data set: vortex, bzip2, and parser. These three benchmarks were selected because of their varied program properties: vortex is an object-oriented database program, bzip2 performs data compression, and parser is a recursive-descent parser. Experiments were performed on a 1 GHz G4 processor with 2MB cache and 2GB memory. Each point in the sweep was executed five times and the variation in time for a single evaluation up to an average of five was compared.

The variation in running times for the three benchmarks is shown in Figures 5.1, 5.3, and 5.5. The average of five trials is used as a baseline in the graphs. Lines show the absolute difference in time reported when fewer than five trials are used. When only a single trial is used the greatest divergence from using an average of five trials is slightly less than a quarter of a second. Variation decreases significantly when a second trial is used, and lesser decreases are observed as the number of trials is further increased. Since there will always be some variation, we need to determine how much variation is acceptable.

We also examined the variation in running times as a percentage of the total running time of the program to provide further insight. These results can be seen in Figures 5.2, 5.4, and 5.6. The greatest deviation we see between a single trial and the average of five trials is still less than 0.7 percent, and a 0.2 percent deviation is far more typical. Additional trials, as stated earlier, further reduce the deviation, however, no method that relies on running times can be completely precise. Furthermore, any deviation could possibly have adverse effects on the adaptive compiler.

Given the high degree of precision observed even when using only a single evaluation, we decided to use just a single execution of the code using the training data set to evaluate the objective function. Repeated trials would lead to greater precision,
but, the precision of a single trial was high enough that any lost improvements would be minor. This potential improvement must be weighed against the additional cost of repeated executions. Execution of the code is usually the most expensive part of the adaptive process, and repeated trials would substantially increase the total time. If we have a fixed time budget for performing adaptive inlining, it seems obvious that better results can be obtained using a single evaluation for the objective function and having time to explore more sets of inlining decisions through additional descents in other parts of the space.

5.3 Preventing Repeated Evaluations

In the previous section, we determined that the objective function for the adaptive inlining system should consist of executing the inlined program and reporting the running time. Even though an adaptive system can afford the overhead of executing code, it is important to avoid superfluous executions. We need a mechanism to avoid
Figure 5.2: Timer accuracy as a percentage for a SC sweep of vortex

Figure 5.3: Timer accuracy for a SC sweep of bzip2
Figure 5.4: Timer accuracy as a percentage for a SC sweep of bzip2

Figure 5.5: Timer accuracy for a SC sweep of parser
executing identically inlined codes more than once.

Depending on the specific adaptive algorithm used, it is likely that a single condition string is examined more than once. Repeated executions can be avoided in this case with a simple lookup mechanism, but, this is not our only concern. It is also possible for different condition strings to produce the same set of inlining decisions and consequently identical executables. We want to avoid performing multiple evaluations of a single set of decisions.

We solve this problem by creating a signature every time we inline a program. The inliner examines individual call sites, applies the condition string and then inlines the call site if appropriate. Call sites are examined in a deterministic order proceeding from the leaves to the root of the call graph. The signature is a binary string constructed by adding a one to the string whenever a call site is inlined and a zero to the string whenever a call site is not inlined. Since the order in which call sites are examined is deterministic, two inlined programs are identical if they have the same
signature.

It is important to note that signatures can only be used to determine whether two inlined codes are identical. Once a single different decision is made it can have a cascading effect on future inlining decisions. Inlining a call site can create new call sites that don’t exist in other versions of the program. Therefore, it is impossible to draw any conclusions from similar but non-identical signatures.

Inlining decision signatures allow us to avoid redundant executions and also reduce the number of compilations necessary for the adaptive system. We perform inlining as a source-to-source transformation at the beginning of the compilation process. If the signature has already been seen, then the remainder of the compilation and execution process can be aborted and the previous results can be used. This helps ensure the practicality of the adaptive inlining system and allows more distinct sets of inlining decisions to be examined in a fixed amount of time.

5.4 Choosing a Condition String Structure

The goal of our adaptive inlining system is to explore the decision space exposed by the inliner to find an extremely high-quality set of inlining decisions. The condition-string system used by our inliner provides a massive decision space. This space contains sets of inlining decisions that produce excellent results, as shown in the previous chapter. However, the immense size of the decision space has drawbacks as well.

An extremely large decision space can make it harder for an adaptive system to find a good result in a reasonable amount of time. A large space increases the possibility of spending significant time exploring poor sets of decisions. For example, the inliner will allow you to inline only procedures of greater than a certain size. It is improbable that such a condition string will yield good results, and it is also likely that it will result in exponential code growth. Therefore, we want to limit the size of the adaptive inliner’s exploration space to facilitate easier searching and prune obviously poor sets of inlining decisions.
We simplified the space of possible inlining decisions by restricting condition strings to a basic form. The adaptive inlining system explores only conditions of the form:

\[
\text{"sc} < A \mid \text{sc} < B, \text{ln}d > 0 \mid \text{sc} < C, \text{sc}c = 1 \mid \text{cl}c < D \mid \text{cp}c > E, \text{sc} < F \mid \text{d}cc > G'\]

This form of condition string was chosen based on previous inlining work and the experiments of the previous chapter.

The first three clauses in the restricted form have long been used as inlining heuristics. Inlining procedures with less than a specified instruction count has several potential advantages. Small procedures cause less code growth when inlined and are often frequently executed leaf procedures. Inlining small procedures attempts to generate maximum benefit for minimum code growth. Using statement count in conjunction with the procedure's static call count or the loop nesting depth of the call site is an attempt to selectively inline somewhat larger procedures for maximum benefit. Typically, code within loops is executed more frequently, increasing the benefit of inlining calls inside loops. Procedures with a static call count of one don't necessarily have greater benefit when inlined, but don't result in code growth since the original procedure can be eliminated after inlining.

The fourth clause inlines procedures that contain less than a specified number of procedure calls. This clause was shown to produce some extremely good results when used independently in the previous chapter. Inlining procedures that contain only a few calls to other procedures has the intent of inlining leaf procedures in the call graph. Leaf procedures are good candidates for inlining because they are frequently where much of the work of the program is performed.

The fifth clause in our condition string attempts to inline call sites based on the potential for increased optimization. A significant amount of work has been done on various heuristics for this purpose. Our approach is to inline procedures with
greater than a specified number of constant parameters. Procedures with constant parameters often benefit more from inlining, since the constant-valued parameters provide more opportunity for constant propagation and other optimizations. We pair the number of constant parameters condition with a limitation on procedure size to mitigate against code explosion.

The final clause inlines call sites based on their dynamic call count. This clause was added based on the various work done using profiling information to guide inlining decisions. Inlining the most frequently executed call sites should be advantageous, since any benefits of inlining the call site will be magnified.

Successful adaptive inlining requires a large space of inlining decisions that contain good solutions for different programs. Constraining the space in this way limits the set of possibilities that the algorithm can consider and steers it away from potentially fruitless paths. This results in a more manageable exploration space which still contains a great number of good solutions. This will be demonstrated later in the chapter. Now, we must devise an adaptive approach for exploring this constrained space.

5.5 Developing an Adaptive Approach

We carefully constructed a search space of inlining decisions suitable for adaptive exploration. The search space is large and expressive enough to find good sets of inlining decisions for a variety of programs, while also being constrained enough to be searched efficiently. In order to gain the benefits of this search space, however, we must implement an effective adaptive technique for exploring the space.

We use a hill climber with randomized restart to explore the search space. We use a hill climbing approach for two reasons. First, hill climbers have been used with great success in previous work on adaptive compilation [5]. Second, the preliminary explorations of the search space in the previous chapter suggest a relatively smooth space. Search spaces with smooth slopes are ideal for hill climbing, since the algorithm
can follow the slope to a good result.

A hill climber is a greedy algorithm for exploring the search space. We begin by selecting a point in the search space at random and evaluating it. Then, we begin evaluating the neighbors of the first point until one is discovered to be a better point in the space. We then shift focus to this new point and begin exploring its neighbors. Some hill climbers examine all neighbors before selecting the best downward step. We select the first downward step since it requires fewer evaluations, and the search space has several good points. This process continues until we reach a local minimum. We perform several such descents from different random points and use the best local minimum that we discover.

The first step in building a hill climber for this problem is defining the notion of a neighbor. We use a condition string with seven numerical parameters defined in the previous section. We define the immediate neighbors of a specified point to be those arrived at by increasing or decreasing a single parameter, resulting in a set of 14 potential neighbors for each point.

We also need to set bounds on each of the parameters in the condition string before using the hill climber. This is important both because it provides a limited area within which to generate a random point and eliminates nonsensical values (e.g., comparing statement count against negative values). We attempt to set the bounds conservatively enough that they do not eliminate good decision sets.

The parameters based on statement counts are set as the result of a fast statement count sweep. We evaluated several inlining decisions of the form "sc < X", giving X an initial value of ten. X is repeatedly doubled and the string reevaluated until the program can no longer be compiled due to memory constraints, the code size has grown to ten times that of the original program, or no additional procedures are inlined compared to the previous string. The previous value of X is then used as the maximum for the statement count parameter, while zero is used as the minimum. The other statement count parameters are set to a multiple of this value, since they
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Minimum Value</th>
<th>Maximum Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>Set by fast SC sweep</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>10*Set by fast SC sweep</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>10*Set by fast SC sweep</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>Set by fast SC sweep</td>
</tr>
<tr>
<td>G</td>
<td>Set by fast DCC sweep</td>
<td>Max. call site frequency</td>
</tr>
</tbody>
</table>

Table 5.1: Bounds for condition string parameters

are restricted in other ways.

The bounds for the call count and constant-parameter-count parameters are based solely on our experience during initial experiments. Call count shows good results with values up to three; however, “c1c < 4” exhibited exponential code growth with some programs. Therefore, we set the upper bound of the parameter to three and the lower bound to one. Similarly, we saw no significant benefit in tests when raising the constant parameter count required beyond three. We set the lower bound of the constant parameter count parameter to zero and the upper bound to three.

We set the upper bound of the profile-based parameter using the logic that it is unnecessary to have values higher than any dynamic call count in the program. During profiling, the highest dynamic call count of any call site is recorded and used as the upper bound of the procedure. The lower bound is then set using a similar procedure to setting the upper bound of the statement-count-based parameters: the minimum dynamic call count required for inlining is repeatedly reduced until one of the three conditions specified earlier occurs. A summary of these bounds can be seen in Table 5.1.

Once we have determined bounds for the search space and defined neighbors, it is
possible to begin using a hill climber to adaptively find good solutions in the search space. However, at this point, it will not yet work effectively because of two obstacles. First, although each individual parameter has been constrained to a reasonable range, not all combinations are reasonable. Frequently, a start point can be selected that has several parameters set to extremely admissible values. This results in explosive code growth and forces the inliner to select another start point. Second, several of the parameters have extremely large bounds, and single-stepping between discrete values results in an extremely slow search. For example, the most frequent call site in vortex is executed over 80 million times. The lower bound will be set at under 1000 executions. This provides essentially 80 million discrete points for just the dynamic call count parameter. Trying to find a good value for dcc by adjusting the parameter by one is completely impractical.

We make the hill climber practical by providing two solutions to combat these problems. We restrict start points to a subset of the total search space, and define each parameter to have only a limited number of points of exploration. For each parameter with a large range of values, we allow 21 points of exploration with ordinals zero through 20. This still provides a search space of just over 49 million points. Our first experiments divided the points of exploration linearly across the space of possible points, however, we quickly realized that this was problematic. Many parameters have an extremely large search space, but the interesting values tend to be grouped together at lower values. A simple quadratic distribution is a natural alternative, but succumbs to the reverse issue: points are grouped too closely together at the low end, and too sparsely at the high end. We therefore use a quadratic distribution with a linear coefficient: \( value = c_1 x^2 + c_2 x \). Currently, we have set the linear coefficient \( c_2 \) to a constant value of five. The quadratic coefficient \( c_1 \) is parameter specific and automatically set to generate the correct minimum and maximum values. Table 5.2 shows how these different approaches divide the parameter space for \( sc \) in vortex.

It is important to note that final point in Table 5.2 for the hybrid scheme dif-
fers from the linear and quadratic methods. This difference was not intended; when calculating the coefficient for the quadratic term we inadvertently floored the value to the nearest integer, and the error was not detected until after significant experimentation. The error was not detected because it has an insignificant effect on the adaptive inliner.

We immediately reevaluated the hill climber with the "correct" coefficient on several benchmarks after detection of the error; the results were essentially unchanged. This result is not surprising. The purpose of the selection of points is to limit the exploration space and distribute points across the space in a sensible method. The error results in a slightly smaller maximum value for a set of points, but with the same overall distribution. The exact maximum value was never considered to be critical, as indicated by the rough way in which it was chosen. We continue to use a floored coefficient value for all of our experimentation, since it has negligible impact on the adaptive inliner and allows use to preserve our previous results.

We are able to determine distribution points for the dynamic-call-count parameter differently using additional information. Since we know the number of times every procedure in the program is called, we can use percentiles to determine points of exploration. For example, if there are 1000 call sites with an execution frequency between the previously determined maximum and minimum, the first point of exploration would be the dynamic call count of the most executed call site of those thousand call sites. The second point of exploration would be the dynamic call count of the 50th most executed call site (recall that we discretize the space into 21 points). This gives a good distribution of points across the space. A similar technique cannot be used for the statement count parameters, since the statement count of a procedure will change during the process of inlining.

Increasing the granularity of each parameter in the space allows the hill climber to explore the adaptive space efficiently without getting bogged down in minute changes, but does not solve the problem of poor start point selection. The range of each pa-
<table>
<thead>
<tr>
<th>Ordinal</th>
<th>Linear</th>
<th>Quadratic</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>256</td>
<td>13</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>512</td>
<td>51</td>
<td>58</td>
</tr>
<tr>
<td>3</td>
<td>768</td>
<td>115</td>
<td>123</td>
</tr>
<tr>
<td>4</td>
<td>1024</td>
<td>205</td>
<td>212</td>
</tr>
<tr>
<td>5</td>
<td>1280</td>
<td>320</td>
<td>325</td>
</tr>
<tr>
<td>6</td>
<td>1536</td>
<td>461</td>
<td>462</td>
</tr>
<tr>
<td>7</td>
<td>1792</td>
<td>627</td>
<td>623</td>
</tr>
<tr>
<td>8</td>
<td>2048</td>
<td>819</td>
<td>808</td>
</tr>
<tr>
<td>9</td>
<td>2304</td>
<td>1037</td>
<td>1017</td>
</tr>
<tr>
<td>10</td>
<td>2560</td>
<td>1280</td>
<td>1250</td>
</tr>
<tr>
<td>11</td>
<td>2816</td>
<td>1549</td>
<td>1507</td>
</tr>
<tr>
<td>12</td>
<td>3072</td>
<td>1843</td>
<td>1788</td>
</tr>
<tr>
<td>13</td>
<td>3328</td>
<td>2163</td>
<td>2093</td>
</tr>
<tr>
<td>14</td>
<td>3584</td>
<td>2509</td>
<td>2422</td>
</tr>
<tr>
<td>15</td>
<td>3840</td>
<td>2880</td>
<td>2775</td>
</tr>
<tr>
<td>16</td>
<td>4096</td>
<td>3277</td>
<td>3152</td>
</tr>
<tr>
<td>17</td>
<td>4352</td>
<td>3699</td>
<td>3553</td>
</tr>
<tr>
<td>18</td>
<td>4608</td>
<td>4147</td>
<td>3978</td>
</tr>
<tr>
<td>19</td>
<td>4864</td>
<td>4621</td>
<td>4427</td>
</tr>
<tr>
<td>20</td>
<td>5120</td>
<td>5120</td>
<td>4900</td>
</tr>
</tbody>
</table>

Table 5.2: Division of the SC parameter for bzip2 using different distributions
rameter was selected to allow independently reasonable values. Several parameters taken in conjunction may render a start point unsuitable, despite having each individual parameter within admissible bounds. We therefore restrict the space from which starting points can be selected. Each parameter is normalized to a value between zero and twenty. We then require the parameters for a start point to satisfy the following condition: \( A^2 + B^2 + C^2 + D^2 + E^2 + F^2 + G^2 \leq 20^2 \), where A through G are the normalized parameters. This creates a start space where a single parameter can be at its maximum value or several parameters can have relatively large values, but eliminates the many unsuitable start points where many parameters have initially high values. A solution can still fall outside these bounds through the descent of the hill climber.

It is interesting to note how different our hill climber and search space are from those in the work on adaptively determining the order of scalar optimizations [5]. Our search space is potentially much larger since we include numerical values of program properties. At the same time, neighbors are defined differently, and our search space has far fewer neighbors for a specific point. These differences in the search space lead to differences in the hill climber where we worry about setting bounds for parameters and restricting the selection of start points. This demonstrates how adaptive techniques must be tailored to individual optimizations.

5.6 Results

We evaluate this refined hill climber using the same system that was used in previous sections of this chapter. We ran experiments ranging from a single descent of the hill climber to the best of twenty descents and compared results with those obtained without inlining or using the inliner provided by GCC. We begin by presenting detailed results with the Vortex benchmark, which is representative, and then show a summary for several other benchmarks.

We used our adaptive inliner to perform one hundred descents on the Vortex
Figure 5.7: Running time for Vortex using different inlining strategies

benchmark to find a good set of inlining decisions. The endpoints of each descent were used to provide an average of expected results when performing a single descent, best of five, best of ten, or best of twenty descents. Since we performed a total of one hundred independent descents this gives us 20 best of five, 10 best of ten, and 5 best of twenty descents. The results of these runs are compared against no inlining and GCC inlining, both using a single, combined, source file in Figure 5.7. GCC inlining performs substantially better than no inlining, and our adaptive approach provides significantly more improvement than the GCC inliner. The benefit of performing more descents is less apparent, but can be seen better in Figure 5.8.

The adaptive inliner shows good results even when only a single descent is used, but the final result improves as more descents are performed. The biggest change comes from performing five descents instead of only one, which fits our intuition. Performing multiple descents not only achieves better execution times on average, but also leads to more consistent results. Figure 5.9 shows the standard deviation of
the best executable found using differing numbers of descents with the hill climber. Performing only a single descent not only provides slightly worse results on average, but the likelihood that the best set of inlining decisions found is significantly worse than what is attainable is far greater.

The improved performance and reliability of performing multiple descents with the hill climber strongly encourage such an approach. However, the drawback of performing more descents is increasing the number of evaluations and time required. Figure 5.10 shows the number of evaluations required for performing a varying number of descents on the Vortex benchmark. It is important to perform enough descents with the hill climber to ensure a good result, but not to perform so many descents that the evaluation time becomes prohibitive. Our results with Vortex suggest that performing five descents of the hill climber is a good compromise.

We conducted the same experiments that were performed on the Vortex benchmark on several other programs as well. We examined four other benchmarks from
Figure 5.9: Standard deviation in results for Vortex using different inlining techniques

Figure 5.10: Evaluations required for Vortex using different inlining techniques
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Source lines</th>
<th>Procedures</th>
<th>Avg. procedure size</th>
</tr>
</thead>
<tbody>
<tr>
<td>vortex</td>
<td>67220</td>
<td>999</td>
<td>12.2</td>
</tr>
<tr>
<td>parser</td>
<td>11393</td>
<td>425</td>
<td>10.67</td>
</tr>
<tr>
<td>bzip2</td>
<td>4649</td>
<td>76</td>
<td>19.97</td>
</tr>
<tr>
<td>gzip</td>
<td>8616</td>
<td>146</td>
<td>13.52</td>
</tr>
<tr>
<td>mcf</td>
<td>2423</td>
<td>92</td>
<td>6.67</td>
</tr>
</tbody>
</table>

Table 5.3: Summary of benchmarks used

SpecINT in detail, in addition to vortex. These benchmarks were selected for their diversity. Table 5.3 summarizes some of the basic properties of these benchmarks. It is important to note, however, that the benchmarks vary not only in their superficial properties, but also in their purpose and design: vortex is an object-oriented database program, bzip2 and gzip are computationally-intensive compression programs, parser is a recursive descent parser; and mcf performs vehicle-depot scheduling.

We do not present individual graphs of the results from these benchmarks here, but they reinforce the conclusions reached with the Vortex data. Figure 5.11 summarizes the results we saw for the different benchmarks. We compare execution time of the benchmarks with no inlining, gcc inlining, or adaptive inlining using a hill climber with five descents. These are not the best possible results for the adaptive inliner; we have observed slightly better results for each of the benchmarks when using more descents. However, these results demonstrate that an adaptive inlining system can consistently outperform a static inlining system using a reasonable approach.

In this section, we have demonstrated the ability of an adaptive inlining system to find a superior set of inlining decisions with a limited number of evaluations using a randomized hill climber in a carefully constructed search space. Adaptive inlining remains an expensive technique, however, and the next section examines how to
Figure 5.11: Normalized execution time of benchmarks using various inlining methods achieve the best possible results with limited evaluations.

5.7 Tuning the Hill Climber

The hill climber developed in the previous section effectively finds good sets of inlining decisions for various programs. However, the time required to find a good solution is still significant. Further reducing the number of evaluations and descents required to find a good solution will increase the practicality of adaptive inlining. This section examines methods to improve the efficacy of our hill climber.

The first technique we examine is using limited patience during a hill climber’s descents. Normally, a hill climber will examine every neighbor if necessary in an attempt to find a better point in the search space. When a hill climber has limited patience, it will only examine a specified percentage of the neighbors before terminating the search. This can be detrimental to the results of a specific descent, but allows more descents to be conducted in a limited amount of time since each descent
requires fewer evaluations. Limited patience is a key factor in the efficiency of hill climbers for finding good optimization sequences [5].

We tried varying levels of patience with our hill climber to discover the impact of patience in finding better solutions faster. We compare 100% patience against 50% and 25% patience for the benchmarks from the previous section using one, five, ten, and twenty restarts of the hill climber. The results were averaged out over a hundred total restarts. We examine 50% and 25% patience since they provide sufficient savings to make limited patience worthwhile and lower values would only examine one or two neighbors.

The results of our experiments with limited patience can be seen in Figures 5.12 through 5.16. The different graphs show a common and expected theme: if the adaptive system can only use a small number of evaluations running several descents, limited patience is preferable. However, if a great number of trials can be afforded, then running several patient descents provides the best results.
Figure 5.13: Effect of patience on the Parser benchmark

Figure 5.14: Effect of patience on the Bzip2 benchmark
Figure 5.15: Effect of patience on the Gzip benchmark

Figure 5.16: Effect of patience on the Mcf benchmark
The results in Figure 5.14 are an anomaly which differs from those obtained for the other benchmarks; better results are obtained using an approach with the same number of descents and 50 percent patience, instead of 100 percent patience. Obviously, this should not be the case, but such results can occur with randomized greedy algorithms. Since the two sweeps were executed independently, they had different start points for each of their descents. Therefore, the descents with 50 percent patience could have started at initial points capable of reaching better inlining solutions.

Another method for tuning the hill climber used by the adaptive system is to bias the order in which neighbors are examined. If changing certain parameters has a higher tendency to yield a better set of inlining decision, then those parameters should be examined first. This observation results in finding good solutions with fewer evaluations, and when combined with limited patience also leads to better results.

Implementing a biased hill climber requires first determining which parameters most frequently improve results when changed. This trend needs to be benchmark independent, since biasing information will not be available for a program when it is being subjected to adaptive inlining. We ran 100 descents of the hill climber on five different benchmarks and recorded which neighbor was chosen every time a descent was made. The percentage of time each neighbor was chosen can be seen in Table 5.4.

The results in Table 5.4 are both surprising and significant. Two important observations can be made from the table. First, every possible condition change occurs a significant percentage of the time for at least some benchmarks. This indicates that all of the parameters of the adaptive inliner’s condition string assist in finding good sets of inlining decisions, validating the design of our condition string and showing that a variety of program properties need to be examined by the adaptive inliner and the problem with previous static techniques.

Second, the most frequently chosen parameters vary by benchmark. This result makes biasing unsuitable for the hill climber. The premise of biasing is that we can
<table>
<thead>
<tr>
<th>Step</th>
<th>Vortex</th>
<th>Parser</th>
<th>Bzip2</th>
<th>Gzip</th>
<th>Mcf</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC Increased</td>
<td>7.88%</td>
<td>11.17%</td>
<td>15.74%</td>
<td>16.56%</td>
<td>9.30%</td>
</tr>
<tr>
<td>SC Decreased</td>
<td>9.07%</td>
<td>19.68%</td>
<td>21.30%</td>
<td>15.95%</td>
<td>22.10%</td>
</tr>
<tr>
<td>Loop SC Increased</td>
<td>8.11%</td>
<td>10.64%</td>
<td>0.93%</td>
<td>2.45%</td>
<td>1.16%</td>
</tr>
<tr>
<td>Loop SC Decreased</td>
<td>8.35%</td>
<td>8.51%</td>
<td>1.85%</td>
<td>5.52%</td>
<td>3.49%</td>
</tr>
<tr>
<td>SCC SC Increased</td>
<td>13.60%</td>
<td>10.11%</td>
<td>23.15%</td>
<td>6.75%</td>
<td>20.93%</td>
</tr>
<tr>
<td>SCC SC Decreased</td>
<td>5.25%</td>
<td>8.51%</td>
<td>12.04%</td>
<td>7.36%</td>
<td>34.88%</td>
</tr>
<tr>
<td>CLC Increased</td>
<td>3.82%</td>
<td>4.26%</td>
<td>8.33%</td>
<td>6.13%</td>
<td>2.33%</td>
</tr>
<tr>
<td>CLC Decreased</td>
<td>3.82%</td>
<td>2.12%</td>
<td>2.78%</td>
<td>0.61%</td>
<td>2.33%</td>
</tr>
<tr>
<td>CPC Increased</td>
<td>3.58%</td>
<td>5.32%</td>
<td>2.78%</td>
<td>6.13%</td>
<td>2.33%</td>
</tr>
<tr>
<td>CPC Decreased</td>
<td>4.06%</td>
<td>1.59%</td>
<td>4.63%</td>
<td>14.72%</td>
<td>1.16%</td>
</tr>
<tr>
<td>CPC SC Increased</td>
<td>6.44%</td>
<td>3.19%</td>
<td>0.00%</td>
<td>4.91%</td>
<td>0.00%</td>
</tr>
<tr>
<td>CPC SC Decreased</td>
<td>3.34%</td>
<td>0.53%</td>
<td>0.00%</td>
<td>2.45%</td>
<td>0.00%</td>
</tr>
<tr>
<td>DCC Increased</td>
<td>18.85%</td>
<td>4.26%</td>
<td>1.85%</td>
<td>0.61%</td>
<td>0.00%</td>
</tr>
<tr>
<td>DCC Decreased</td>
<td>3.82%</td>
<td>10.11%</td>
<td>4.63%</td>
<td>9.82%</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

Table 5.4: Frequency each neighbor was chosen as the downward step.
examine a set of benchmarks to determine how frequently each parameter is chosen, and this ratio will hold true for other codes subjected to the adaptive inliner. However, since the variation amongst the benchmarks is so great, we cannot expect the average of this small set of benchmark codes to necessarily approximate the behavior of the hill climber for other programs.

The inability to use biasing in the adaptive system is actually an important revelation. If the same parameters were adjusted in the same ratio consistently across different benchmarks, then it would suggest the possibility of a universal solution to finding sets of inlining decisions. However, the large variation of parameters chosen between benchmarks suggests that a different inlining approach is necessary for different benchmarks. It reinforces our belief that an adaptive inlining system is the correct method for making inlining decisions.

In this section, we investigated how to tune the basic hill climber from the previous section to find better sets of inlining decisions using fewer evaluations. A hill climber with limited patience can conduct more descents in a fixed number of evaluations than a patient hill climber. This results in limited patience finding better sets of inlining decisions when only a small number of evaluations can be afforded, while a patient hill climber continues to be the best high-evaluation option. Biasing the hill climber is not viable, because of variance between benchmarks. This validates the importance of using an adaptive system for performing procedure inlining.

5.8 Examining Sets of Decisions

In the previous sections, we have shown how an adaptive inlining system can discover a good set of inlining decisions for a specific program and why an adaptive approach is necessary. In this section, we examine the different sets of good inlining decisions we discovered for each benchmark. This allows us to determine what similarities exist between different sets of decisions for the same benchmark, and also across benchmarks. Table 5.4 has already shown that good sets of inlining decisions across
Figure 5.17: Execution time vs. Executable size for vortex

benchmarks come from very different condition strings, but there may still be certain high-level similarities.

We first examine how executable size varies between sets of decisions. Figure 5.17 compares execution time against executable size on the vortex benchmark. Each point in the graph is the executable produced using a set of inlining decisions derived by a single descent of the hill climber. We see a wide variation in executable sizes that correspond to good sets of inlining decisions. However, amongst the good solutions, the smaller executables tend to have faster running times. This reinforces the idea that more inlining is not necessarily better. Instead, selecting just the right set of call sites to inline produces the best performance.

However, drawing results on the proper amount of inlining from executable size alone is dangerous. The largest executables in Figure 5.17 are not necessarily those with the most inlined code. For example, the version of vortex with the most inlined code could also allow the elimination of more dead code, resulting in a smaller exe-
Figure 5.18: Execution time vs. Source lines for vortex

cutable. Therefore, we also examine the relationship between execution time and the number of lines of source code after inlining is performed. The results for vortex can be seen in Figure 5.18. This data reinforces the conclusion from the previous graph that more inlining is not necessarily better.

We see similar results for the other benchmarks that we examine. Figures 5.19 through 5.22 show execution time versus executable size for the parser, bzip2, gzip, and mcf benchmarks. Figures 5.23 through 5.26 show execution time versus lines of source code for these same benchmarks. The graphs differ, as is to be expected, but they continue to reinforce the idea that better solutions do not result simply from more inlining.

We also examine how the number of procedure calls eliminated through inlining affects execution time. One might expect better performance from sets of inlining decisions that inline more procedure calls since eliminating call overhead is a direct improvement of procedure inlining. We compare execution time against the number
Figure 5.19: Execution time vs. Executable size for parser

Figure 5.20: Execution time vs. Executable size for bzip2
Figure 5.21: Execution time vs. Executable size for gzip

Figure 5.22: Execution time vs. Executable size for mcf
Figure 5.23: Execution time vs. Source lines for parser

Figure 5.24: Execution time vs. Source lines for bzip2
Figure 5.25: Execution time vs. Source lines for gzip

Figure 5.26: Execution time vs. Source lines for mcf
of dynamic calls removed by inlining for the \texttt{vortex} benchmark in Figure 5.27. There does not appear to be a correlation between more dynamic calls inlined and improved performance. In fact, for \texttt{vortex} the sets of decisions that eliminate fewer dynamic calls actually tend to perform better. Once again this data shows that more inlining does not necessarily lead to better performance. It also suggests, as did the benefit of the constant parameter count property, that much of the benefit from inlining does not come from direct effects, but instead from enabling other optimizations.

Figures 5.28 through 5.31 show the correlation between the number of dynamic calls inlined and execution time for the remaining benchmarks. These graphs show similar results as Figure 5.27. Looking at all of the graphs, however, causes you to notice how similar the total number of dynamic calls inlined is for the different sets of decisions for a given benchmark. For the \texttt{gzip} benchmark all of the sets of inlining decisions save one eliminate an almost identical number of procedure calls. (Due to the scale of the graphs, small changes in dynamic calls inlined cannot be perceived.)
This clustering of data exists to a lesser extent for the graphs of executable size and source lines as well. This suggests, as would be expected, a certain amount of underlying similarity between good sets of inlining decisions for a specific benchmark.

The results of this section reinforce what we have observed earlier in this chapter. The data shows that simple, somewhat intuitive, approaches do not work for inlining. Performing as much inlining as possible, or eliminating more dynamic procedure calls does not lead to better performance. Good sets of inlining decisions benefit from a variety of factors, and finding them for different benchmarks requires an adaptive approach.

### 5.9 Future Work

In this chapter, we have shown that an adaptive-inlining system can produce code superior to a traditional inliner in a reasonable number of evaluations. However, adaptive inlining is still a costly technique that could benefit from further refinement.
Figure 5.29: Execution time vs. Dynamic calls inlined for bzip2

Figure 5.30: Execution time vs. Dynamic calls inlined for gzip
Figure 5.31: Execution time vs. Dynamic calls inlined for mcf

There are several approaches to reduce the number of evaluations required to find a good inlining solution that can still be investigated.

We have shown that limiting the patience of the hill climber produces better sets of inlining decisions with fewer evaluations. Changes could be made to the way limited patience is implemented that might further improve results. Currently, the degree of patience is fixed; using variable patience might prove more profitable. For example, the hill climber could begin with a patience of 25 percent. As more downward steps are discovered, the level of patience could be increased. This would reflect the idea that as the hill climber is more successful, we are willing to spend more time trying to further improve the result found.

Another potential change to how patience is used in the hill climber would be to change the way patience is calculated. Currently, the patience of the hill climber is determined by the number of neighbors that can be examined before terminating the search. Patience could also be calculated based on the amount of compilation time
spent at a point in the search space. When examining results of the hill climber, it is sometimes apparent that the hill climber spends a great deal of time exploring the neighbors of a point which conducts massive amounts of inlining. Each neighbor requires a great deal of compilation time to evaluate, and the likelihood of a good solution springing from such a heavily inlined piece of code is low in our experience. Limiting patience based on compilation time would reduce the amount of time spent on such points, while at the same time it would allow more neighbors to be explored for points with modest amounts of inlining.

We could also explore the viability of using different search techniques for the adaptive inliner. Our adaptive inliner uses a randomized hill climber because of the success we observed in prior work [5]. The hill climber was excellent at finding good sets of inlining decisions using limited evaluations. An approach using genetic algorithms is probably unnecessary given the quality of the hill climbers results and the number of evaluations typically required to obtain good results from a genetic algorithm. However, it may be worth investigating other low-cost search techniques. Specifically, are there techniques that can take advantage of program properties to quickly find good sets of inlining decisions?

Further examination of search techniques is one method to potentially improve adaptive inlining; however, a better approach might be to investigate different basic forms of the condition string. Table 5.4 shows that all of the current parameters in the condition string are useful; however, there may be other worthwhile parameters. Exploring the use of other program properties in the condition string as well as different methods for combining them could lead to a search space that allows the hill climber to find good inlining decisions quicker.

5.10 Conclusions

The previous chapter showed the potential of adaptive inlining to find a significantly better set of inlining decisions than a traditional static technique. However, it does not
explore how an adaptive system can efficiently find sets of inlining decisions. This chapter capitalizes on the potential shown previously by constructing an adaptive inlining system that makes superior inlining decisions in limited time.

Our adaptive inliner uses a hill climber with randomized restart to explore the space of inlining decisions. The space of decisions is constrained using a standard condition string structure and the efficacy of points in the space is evaluated using execution time of the resulting code. When only a few evaluations can be afforded, running several descents with limited patience produces the best results, while a patient hill climber performs better given sufficient time. Overall, code optimized using adaptive inlining consistently outperforms uninned code or code inlined using a static technique.
Chapter 6

Conclusion

In this dissertation, we examine the use of adaptive techniques during program compilation and show that they can be used to improve the performance of specific optimizations. Successfully using adaptive techniques with compiler optimizations requires designing the optimizations in a flexible, adaptive manner and constructing an adaptive approach using a detailed understanding of the search space. We illustrate this through the development of an adaptive inliner.

Chapter 3 examines adaptive selection of blocking sizes using the MIPSpro compiler. We do this to evaluate the potential of current compilers in adaptive compilation. Adaptively selecting blocking sizes performed significantly better than the default blocking size heuristic and approached the results obtained by ATLAS. Results were limited, however, by the limited tunability of the SGI compiler.

In Chapter 4, we develop a flexible inliner suitable for adaptive inlining. The inliner accepts condition strings that determine which call sites are inlined. Condition strings provide a flexible inliner by combining various program properties in conjunctive normal form. The inliner exposes a large space of different inlining decisions with the potential to outperform static techniques.

We develop an adaptive technique for the inliner in Chapter 5. We carefully design a search space to limit exploration times without excluding high quality results. We then use a randomized hill climber to effectively explore the search space in a limited number of evaluations. This results in a practical adaptive inlining system that consistently outperforms a static inliner and realizes the potential of procedure inlining.
This dissertation makes several major research contributions. We demonstrate the ability to find good blocking sizes using adaptive techniques. We also show that current commercial compilers are poorly parameterized for adaptive compilation. This motivates us to develop an inliner with a well-designed parameter scheme that is both expressive and concise. We develop an adaptive controller for the inliner that finds good sets of inlining decisions and outperforms static techniques. Furthermore, we demonstrate that an adaptive approach is necessary to capitalize on the opportunities that inlining presents. This work validates our belief that some individual optimizations can benefit from adaptive techniques through intelligent parameterization, a good understanding of the search space, and designing an adaptive approach accordingly.
Bibliography


