Tuning an Adaptive-Compilation Search Space
with Loop Unrolling

by

Jeffrey A. Sandoval

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

Master of Science

APPROVED, THESIS COMMITTEE:

Keith D. Cooper, Professor, Chair
Computer Science

Devika Subramanian, Professor
Computer Science

Joseph R. Cavallaro, Professor
Electrical and Computer Engineering

Houston, Texas

April, 2007
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.

UMI®

UMI Microform 1441855
Copyright 2007 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P. O. Box 1346
Ann Arbor, MI 48106-1346
Tuning an Adaptive-Compilation Search Space with Loop Unrolling

Jeffrey A. Sandoval

Abstract

This thesis demonstrates that careful selection of compiler transformations can improve the output and reduce the compile-time cost of adaptive compilation. Compiler effectiveness depends on the order of code transformations applied. Adaptive compilation, then, uses empirical search to tune the transformation sequence for each program. This method achieves higher performance than traditional compilers, but often requires large compilation times. Previous research reduces compilation time by tuning the search process. This thesis, instead, tunes the search space by adding a loop unroller, addressing a deficiency in our compiler. Despite increasing the search-space size, this change results in more effective and efficient searching. Averaged across nine benchmarks, the adaptive compiler produces code, more quickly, that is 10% faster. Unfortunately, implementing a loop unroller is non-trivial for our low-level intermediate language. Therefore, this thesis also contributes an algorithm for identifying and unrolling loops in the absence of high-level loop structures.
Acknowledgments

I would like to thank Keith Cooper, my thesis advisor, for his guidance and insight throughout this research project. I have really enjoyed working with him and learning from him. My thesis committee members, Devika Subramanian and Joseph Cavallaro, have also shared their guidance and expertise. I would like to thank them for their comments and suggestions, which have helped improve my work considerably.

Tim Harvey has been a great friend and mentor since I arrived at Rice. He has always generously offered his time and knowledge to help me; I extend my gratitude for all of his efforts. I would also like to acknowledge the contributions and support of many other Rice students, faculty, and alumni, including Steve Reeves, Linda Torczon, Yi Guo, Yuan Zhao, Cheryl McCosh, Anshuman Das Gupta, Todd Waterman, Dave Peixotto, Mary Fletcher, Mack Joyner, Ryan Yang Zhang, Nathan Tallent, John Garvin, Yuri Dotsenko, and Apan Qasem.

I extend my thanks to the National Science Foundation for providing financial support, through grant ITR-0205303, “ITR: Building Practical Compilers Based on Adaptive Search.”

I am very thankful for the support and encouragement I received from members of the Rice Graduate Christian Fellowship. I especially appreciate my friendship with Chirag Patel.

Finally, I am indebted to my family for all of their love and inspiration, both now and while I was growing up. My fiancée, Monica Van Horn, has been very patient with my intermittent progress, empathetic with my frequent frustrations, and wonderful support throughout my entire research project. My parents, Gerald and Joyce, and my siblings, Elizabeth, Steven, and Patrick, have always taught me to do my best and to not give up. I am thankful for their confidence in me, and I appreciate all of the ways in which they have shaped my life.
# Contents

Abstract ii
Acknowledgments iii
List of Illustrations vi
List of Tables viii

1 Introduction 1
1.1 Background ........................................ 2
1.1.1 Traditional Compilation ....................... 2
1.1.2 Adaptive Compilation .......................... 4
1.2 Tuning the Search Space ........................... 7

2 Related Work 10
2.1 General Loop Unrolling ............................ 10
2.2 Control Flow Graph Cycle Detection ............. 15
2.3 Induction Variable Analysis ...................... 19
2.4 Adaptive Compilation ............................. 20

3 Loop Unrolling 31
3.1 Motivation ......................................... 32
3.2 Background ........................................ 35
3.2.1 Obstacles to Unrolling Unstructured Loops .... 41
3.3 Loop Unrolling ..................................... 49
3.3.1 CFG Analysis and Naive Unrolling ............. 51
3.3.2 SSA Analysis and Clean Unrolling .......... 57
3.4 Final Considerations .................................................. 71
  3.4.1 Hardware Looping Construct .................................. 71
  3.4.2 Loop-Analysis Shortcomings .................................... 72

4 Experimental Results .................................................. 75
  4.1 Benchmarks ........................................................... 76
  4.2 The Hill Climber ..................................................... 80
  4.3 Fixed-Sequence Loop Unrolling ................................... 82
  4.4 Adaptive Compilation with Loop Unrolling ....................... 84
    4.4.1 Search Effectiveness ........................................... 85
    4.4.2 Search Efficiency ............................................... 87
    4.4.3 Pass Frequency ................................................ 93

5 Conclusion ............................................................... 96
  5.1 Future Work .......................................................... 98

A Additional Data .......................................................... 101

Bibliography ............................................................... 107
Illustrations

3.1 Average pass frequency ........................................... 34
3.2 Simple loop unrolling ........................................... 38
3.3 Loop unrolling with cleanup loop ............................... 39
3.4 Types of loop syntax ............................................. 41
3.5 Fundamental Irreducible Graph .................................. 42
3.6 Irreducible Graph Complexities ................................... 44
3.7 Ambiguous loop .................................................. 48
3.8 Possible single loop .............................................. 48
3.9 Possible nested loops ............................................. 48
3.10 Naive loop unrolling ............................................. 52
3.11 Original loop ................................................... 52
3.12 Cloned loop body twice ......................................... 53
3.13 Chained the cloned loop bodies ................................. 53
3.14 Completed naive unrolling ...................................... 53
3.15 A loop with the MRI=1 property ................................. 58
3.16 Effect of naive unrolling on MRI ............................... 60
3.17 Iteration lookahead with MRI=2 ............................... 60
3.18 Clean unroll, by two ........................................... 62
3.19 Computing guard expressions for the unrolled body ........... 65
3.20 Loop with code on the back path ............................... 69
3.21 Using a direct jump to handle code on the back path ........... 69
3.22 Using cloning to handle code on the back path .................. 69
Tables

3.1 Five best sequences for adpcm_coder ........................................ 33
3.2 Frequency of loop peel in top sequences ................................. 33

4.1 The nine floating point benchmarks used for all experiments ....... 78
4.2 Inner Loop Statistics ............................................................. 79
4.3 Peel and unroll frequencies .................................................... 94

A.1 The alphabet for the adaptive compiler; the top portion of the table
indicates the existing transformations, while the bottom portions
indicates the extension provided by this thesis .................................. 101
A.2 The five best sequences found by the hill climber for each benchmark 102
Chapter 1

Introduction

This thesis demonstrates that careful selection of compiler transformations can improve the output and reduce the compile time cost of current adaptive compilation techniques. Because compiler effectiveness is dependent on the order in which code transformations are applied, adaptive compilation has successfully employed empirical search to discover effective transformation sequences for individual input programs. Naively applied, however, this method often requires an unreasonable amount of time, because it reduces to the problem of searching for optimal points in a search space of exponential size. Related research focuses on changing or improving the search algorithms, pruning the search space, or reducing the cost of evaluating a single point [1, 31, 45, 18, 57, 40]. These methods have been very successful, but they treat the set of compiler optimizations as fixed. This thesis, instead, targets a deficiency in the set of compiler optimizations. By augmenting the search space with a new transformation, loop unrolling, the adaptive compiler is able to exploit opportunities that it previously could not. This represents a change in the search space that, despite increasing the size, facilitates more effective and efficient searching. Across nine benchmarks, the new search space allows the adaptive compiler to produce code, often in less time, that is an average of 10% faster.

Unfortunately, adding a loop unroller is non-trivial within the context of our adaptive compiler. A secondary contribution of this thesis is providing an algorithm for identifying and unrolling loops in a low-level intermediate language, where high-level loop structures have been lost. The mechanics of loop unrolling are generally discussed from a high-level perspective. Thus, loop unrolling is often ignored as a re-
search problem and, instead, treated as an engineering problem. However, this thesis targets loop unrolling for the lowest level loop syntax, unstructured loops expressed at an assembly-language level, and therefore requires the combination of two complex analysis phases for loop unrolling to be safe in the general case. Furthermore, the flexible ordering of transformations in an adaptive compiler can create unusual loop structures that previously described methods do not address. This thesis illustrates the difficulty of unrolling loops in this situation, and shows the necessary steps for handling the general case.

The remainder of this chapter presents a brief overview of the background of adaptive compilation. First, several issues with traditional compilation are identified. The next section explains how adaptive compilation can address those problems. Finally, the chapter concludes by examining the implications of changing the set of transformations used in the adaptive compiler.

1.1 Background

Understanding the concepts presented in this thesis requires a brief examination of the issues with both existing traditional and adaptive-compilation techniques. While adaptive compilation successfully addresses the problem of sub-optimal output from traditional compilation, it comes with a high compile-time cost. Reducing the cost of adaptive compilation is the focus of this thesis.

1.1.1 Traditional Compilation

Although high-level programming languages reduce development and maintenance costs as well as increase portability, hand-coded assembly programs often achieve higher performance than compiled programs. Because a compiler bridges the gap between software and hardware, challenges from both domains restrict compiler effectiveness. The two main problems traditional compilers face are eliminating programming-language overhead and responding to the fast rate of microprocessor development.
Programming Languages  General high-level abstractions, although helpful for writing large and complex programs, contribute a large amount of overhead to the actual work done by the program. For example, multi-dimensional arrays provide a convenient model for representing data, but they often lead to complex address calculations in the final executable. Similarly, object oriented programming provides useful structure and abstraction, encouraging code reuse, but it fills the final executable with procedure calls that impede performance. Unfortunately, many of the optimizations necessary for eliminating these overheads must solve NP-complete problems [15]. In practice, then, these optimizations employ "predictive heuristics" that can only approximate the solutions [57]. Compilers generally work well in the many cases that are amenable to the underlying heuristics, but, not surprisingly, we observe many cases in which the compiler cannot produce code that is optimal. Developers must either accept this sub-optimal result or invest a large effort in tuning their code by hand. Often, the developers most concerned with performance will avoid high-level features of the programming language, or they will selectively perform some optimizations by hand. For example, high-performance numerical codes usually only use the most basic features of the lowest-level programming languages available (C or FORTRAN). Additionally, these codes often contain loops that are unrolled by hand and routines that are inlined by hand. Selective use of programming language features and manual application of high impact transformations can result in acceptable performance. Unfortunately, these practices result in forfeiting the very benefits promised by high-level languages: programmer productivity.

Microprocessor Technology  Microprocessor technology changes every few months, easily surpassing the development rate of effective optimizing compilers. With such rapid changes in the hardware, a mature optimizing compiler is often not available for a given architecture; in addition, it is sometimes not feasible to develop a powerful optimizing compiler for some specific architectures because the hardware may become
obsolete before the compiler is in production use. Lack of powerful compilers often leaves developers with one option: using simple compilers that act as translators from machine-independent representation to machine code. Although the compiler can still perform machine-independent optimizations well, important machine-dependent optimizations such as cache optimization, instruction selection, and register allocation are often inadequate for producing optimized code. The code will run correctly, but it will not take full advantage of the processor resources.

Although it is desirable to write software with high level abstractions, the problems described above introduce serious limitations on reaching this goal. In both cases, compiled code can suffer greatly from an inefficient use of machine resources for a given algorithm and architecture. In a world that demands faster response, higher throughput, less energy usage, and reduced production and operating expenses, efficient use of machine resources is critical. The next section describes adaptive compilation, an existing method for solving the problems that have been illustrated in this section.

1.1.2 Adaptive Compilation

The adaptive-compilation framework discussed in this section is based on the work of Grosul, from Cooper’s research group at Rice University [30]. His work solves the problems discussed in the previous section by applying artificial-intelligence techniques in the compilation process. Although Grosul showed a performance improvement in output code of up to 40% over traditional compiler techniques, the compile time cost of this method is often unacceptably large.

Traditional compilers apply the same fixed sequence of optimization passes to every input code. This sequence is based on compiler writers’ expert knowledge of code optimization and is designed to exploit opportunities for optimization that are common in many programs. This approach has been dominant in production compilers, because in the past this approach was able to achieve performance near 100% of peak processor utilization. Today’s machines, however, are much more
complex, and processor utilization is affected by many factors that are difficult to predict. Consequently, today's compilers are rarely able to output code that achieves near peak utilization. Often, high processor-utilization is only achievable with hand coding and significant manual tuning. In response to this, researchers began to re-evaluate the fixed-sequence optimization approach.

Studies indicate that some input programs can be improved substantially with adaptive compilation, a method that customizes the optimization sequence for each given input code and target architecture [21]. Adaptive compilation is successful because compiler transformation effectiveness is determined not only by the characteristics of the input program, but also by the behavior of other transformations in the compilation process. The problem can be viewed as a set of optimization passes that create and destroy opportunities. A pass creates opportunities by transforming the code in such a way that a previously non-existent or unidentifiable opportunity is now visible in the code. Conversely, a pass destroys opportunities simply by transforming the code in such a way that the opportunity either no longer exists or is unidentifiable. Opportunities can be created or destroyed directly as the intended consequence of a pass, or they may be indirectly created or destroyed as an unintended consequence of a pass. For example, weak strength-reduction converts some multiplication operations into shift and addition operations. The direct benefit of consuming this opportunity is a potential savings in execution time and power consumption. Unfortunately, this type of strength reduction may destroy opportunities for tree-height reduction and algebraic reassociation, because a commutative operation (multiplication) is replaced with a non-commutative operation (shift). Algebraic reassociation exhibits similar behavior. Reassociation reorders large expressions containing commutative operations, attempting to increase the effectiveness of common subexpression elimination. Inadvertently, this reordering may destroy existing opportunities for common subexpression elimination.

The fundamental problem that adaptive compilation addresses is that multiple
optimization opportunities are often mutually exclusive; taking advantage of one opportunity destroys another. This means that the behavior of an individual compiler pass is affected by every pass that was applied before it. Similarly, every later pass is affected by the decisions made in the current pass. Therefore, each local decision in an individual compiler pass is actually a global decision in the whole compiler, potentially affecting the entire compilation process. The adaptive compiler explores the trade-offs between different global decisions, searching for the most effective set of decisions. Because a single sequence of optimization passes is not optimal for every input code, an adaptive search can tune a sequence towards the optimal sequence for every specific input code. Fortunately, different sequences can eliminate many of the same inefficiencies from a program, because different orderings of optimization passes result in a similar global effect. Thus, there may be many distinct sequences that achieve similar performance results.

Since the number of phase orderings for a compiler is finite (assuming that the length of the optimization sequence is bounded), the problem could theoretically be solved with a naive brute-force approach of enumerating all solutions and returning the best. In practice this approach is intractable, but several early studies used this method on small subsets of the search space to gain insight into the characteristics of the full search spaces [6, 17]. These studies produced promising results, suggesting several important points. First, different input codes respond to the same optimization sequence in very different ways; in other words, an effective sequence for one input code may not be effective for another input code. This fact implies that adaptive compilation is required for every input program, and a universally good sequence probably does not exist. Second, many optimization sequences for a given input code are nearly as effective as the optimal sequence for that code [6]. This important discovery led to the use of genetic algorithms and hill climbers for effectively exploring the entire search space. Even though it is impractical to enumerate the complete space, it is beneficial to use an intelligent search method to explore a subset of the
space; with enough searching, we can make a strong statistical argument that the best solution discovered within the explored subset is within a small delta of the optimal solution.

Even with intelligent searching methods, however, the cost of adaptive compilation is still high enough to warrant active research. Most of the current research, including this thesis, focuses on reducing the high compile-time cost of adaptive compilation. The method presented in this thesis does not modify the search algorithms, however. Instead, it modifies the search space to facilitate a more effective and efficient search.

1.2 Tuning the Search Space

Intelligent searching does not completely alleviate the high compile-time cost of adaptive compilation. Two factors contribute to this cost: the number of sequences explored and the time required to evaluate a single sequence. The main approaches for reducing compile time correspond to the following factors. First, several methods attempt to reduce the number of sequences explored by focusing the search in regions of the space that are predicted to contain good solutions [1, 31]. Next, some methods prune redundant or ineffective sequences from the search space [45]. Finally, other methods attempt to reduce the time needed to evaluate a given sequence by predicting or computing the execution time without executing the program [18, 57, 40]. All of these methods, however, assume that the search space is fixed. In other words, they ignore the possibility of changing the set of transformations in the compiler.

This thesis, in contrast, examines the effects of modifying the adaptive compilation search space. Specifically, this work adds a loop-unroller pass to our compiler and shows that the search methods are both more effective and more efficient when applied to this new search space. Changing the set of transformations in the compiler will change both the size and contents of the search space. Adding a transformation will increase the search space size, potentially increasing the search time, while removing a transformation will decrease the search space, potentially decreasing the search time.
To complicate matters, however, changing the set of transformations also changes the quality of sequences that the compiler can produce. For example, removing a very powerful transformation, even though it will shrink the search space, is probably a bad idea, because the compiler will no longer be able to leverage the effectiveness of that transformation. This thesis shows that, despite the increase in the search space size, adding a new transformation can be beneficial if it creates new sequences that are more effective than those in the old search space.

The choice of a loop unroller originated as our response to an unexpected behavior of the adaptive compiler. On several benchmarks, our adaptive compiler selected the loop-peel transformation unusually often, up to 70% of the time. Though our intended use of loop peel was to prepare the code for loop unswitching [16] or Cytron et al. style code motion [23], the adaptive compiler found a secondary, unintended use for loop peel: reducing loop overhead. This suggested the need for a stronger transformation: loop unrolling. While each successive application of loop peel can potentially reduce loop overhead for a single iteration, the more powerful loop-unrolling transformation can reduce loop overhead for all of the iterations. This unexpected use of loop peel motivated the focus of this thesis, writing a loop unroller to improve the adaptive compiler.

In the literature, loop unrolling is most often discussed from a high-level source code perspective\textsuperscript{1} [5, 26, 46, 50, 39, 16]. Although this approach allows a very clear presentation of the concept, it is not helpful for the compiler writer who must implement loop unrolling in the context of a low-level intermediate representation. Loop unrolling at such a low level of abstraction exhibits increased complexity, because high-level loop structures and induction variables are not explicitly labeled as they are in high-level programming languages. In addition, our adaptive framework applies compiler passes in unexpected contexts; therefore, our loop unroller must be

\textsuperscript{1}Occasionally, loop unrolling is discussed from a low-level perspective, but in these cases the complex implementation details are not addressed [25, 36]
able to handle the unstructured control flow that can result. This thesis addresses these problems, providing an algorithm for loop unrolling in this context.

Adding a new pass to the adaptive compiler enlarges the search space. However, this thesis demonstrates that, because the opportunity for loop-unrolling often exists in our benchmarks, we are able to search this new larger space just as effectively. Evaluation of the new space considers two metrics, the quality of the code produced and the speed of the search. Results show that the adaptive compiler finds solutions, often in less time, that execute an average of 10% fewer instructions. Although this thesis only measures the effects of adding loop unrolling to the search space, the same methods can be applied as an evaluation function for any compiler transformation. For example, instead of focusing on new transformations, existing transformations could be evaluated and eliminated if they turn out to be unnecessary in the presence of other overlapping transformations. Taken to the extreme, this methodology could be leveraged by compiler writers to select optimizations for a general purpose, statically ordered optimizing compiler.

This thesis provides two main contributions, described in detail in Chapters 3 and 4. First, Chapter 3 presents an algorithm for unrolling loops when high-level loop-structure information is not available. Second, Chapter 4 demonstrates a method for evaluating the effects of adding a new transformation to an adaptive compiler. The results of adding a loop unroller to the Rice University research adaptive compiler indicate that careful selection of the transformations that are included in the adaptive compiler can improve both the effectiveness and efficiency of the search methods, despite the increase in the search space size.
Chapter 2

Related Work

Because this thesis examines several distinct lines of research, the related literature can be grouped into four disjoint categories: general loop unrolling, control-flow-graph (CFG) cycle analysis, induction variable detection, and adaptive compilation.

2.1 General Loop Unrolling

Loop unrolling is a classic transformation that is mentioned frequently in the literature, though it is often not discussed in detail. This section examines the literature that deals with loop unrolling, identifying the relationship between the previous work and this thesis. The main shortcoming of the following research is that it usually does not address the implementation details of loop unrolling.

One of the earliest publications mentioning loop unrolling is by Allen and Cocke in 1972 [5], although it is likely that programmers had already been applying loop unrolling by hand before this. Knuth, for example, described the technique of “loop doubling”, which is simply the manual application of loop unrolling by a factor of two [43]. Allen and Cocke provided a high level description and example for loop unrolling, but did not discuss the implementation details. They mentioned that complete unrolling may be done, if the loop iteration count is a small constant. They identified the main advantages as reduced loop overhead and increased ILP. Because of the increased code size, and cleanup loop overhead, the authors suggested that the decision to unroll and the factor to unroll by should be determined based on the several factors: the loop body size, the execution frequency, and whether a cleanup loop is actually needed. Finally, Allen and Cocke mentioned that unrolling may be
applied to outer loops, but in this case it is usually combined with loop jamming, or fusion. Unlike loop unrolling, which is safe in all cases, unroll-and-jam can be unsafe in the presence of dependences.

Dongarra and Hinds examined the profitability of applying loop unrolling, by hand, to performance-critical inner loops in the linear algebra routines DAXPY and DDOT [26]. They identified three advantages to loop unrolling: reduced loop overhead, increased functional unit usage, and increased straight-line code. The main disadvantage is code growth; if the loop body grows too large, then it may no longer fit into the instruction cache. They provided results from a large variety of machines and compilers, nearly all indicating that unrolling can improve performance by over a factor of two. As expected, higher loop trip counts resulted in more benefit. Finally, they warned that loop unrolling should not be applied blindly, since many pieces of the code will not benefit from (or will be harmed by) unrolling. They suggested that this transformation should be done automatically in future compilers.

The popularity of vector machines increased in the 1970’s and 1980’s as a mechanism for achieving high performance on regular, scientific codes. These machines were very complex, as were the vectorizing compilers needed to utilize the hardware. In 1987, Weiss and Smith used loop unrolling and a primitive form of software pipelining to achieve pipelined scalar performance comparable to that of the CRAY-1S vector architecture. Their motivation was to show that advanced scalar compilation techniques targeting a pipelined architecture can be used instead of the more expensive vector architecture and vectorizing compiler. The authors identified the reduction in loop overhead as a benefit of unrolling, but argued that the largest advantage is improved scheduling. In the presence of data dependences, they suggested a simple form of scalar replacement to reduce the number of memory accesses. In the absence of data dependences, however, the memory operations may be reordered with a simple basic-block scheduling algorithm, achieving a very efficient schedule. Unfortunately, the downside of unrolling is increased register pressure and increased code size. Us-
ing a simulator with 256 registers and a 512 word instruction cache, increased from the base system that has 8 registers and a 64 word instruction cache, they showed speedups that match those obtained when using the vector capabilities of the system. Further analysis showed that 32 registers is sufficient for most of their loops, and the default instruction cache only degraded the speedup slightly. The effectiveness of loop unrolling in this paper suggested that hardware designs should focus on increasing register file and cache sizes, instead of supporting long vector operations.

As superscalar and VLIW architectures became more popular in the 1980’s and 1990’s, the need for increasing instruction level parallelism (ILP) became apparent. Mahlke et al. argued that increasing computational resources (functional units) in a processor will result in little performance improvement without aggressive loop unrolling and register renaming [46]. They showed that unrolling, by itself, is insufficient for increasing ILP, because it leaves output and anti dependencies in the unrolled loop body. These dependences limit a compact instruction schedule. With register renaming, however, each reassignment to a register is instead allocated a new register name. Although this increases register pressure, it allows in a much more efficient instruction schedule. The authors reported a 5.1 speedup when loop unrolling and register renaming was applied to a set of 40 loop nests from a variety of supercomputing applications. Although the register pressure increased by a factor of 2.6, nearly all loops required fewer than 128 registers. Again, this paper suggested that loop unrolling can be very effective, if the register file is large enough.

Despite the comprehensive range of literature on loop unrolling by 1996, Davidson and Jinturkar discovered that production compilers often fail to exploit the full potential of loop unrolling [25]. They conducted a survey of loop unrolling for a variety of production compilers and architectures, concluding that loop unrolling was not uniformly implemented. Many compilers, in fact, only considered trivial loops for unrolling (single basic blocks), yet their studies showed that many important loops are more complex. They argued that loop unrolling should be applied on a low-level
intermediate representation late in the optimization process, after other optimizations have been performed. They noted that sophisticated loop analysis is required to identify anything more than single basic block loops, but they ignored the details because they were able to leverage existing technology in their compiler backend. Their main argument for applying loop unrolling late in the compilation process was that the most appropriate unroll factor can be more easily determined, since the code is near its final machine code form. Their heuristic for determining unroll factor simply choose the largest unroll factor between 0 and 15 that will likely create a final loop that will fit in the instruction cache. Interestingly, the unrolling is applied after register allocation, so they did not experience issues with increased register pressure. This, however, may limit the effectiveness of their approach by preventing better instruction scheduling. They proposed a more aggressive approach to unrolling than previously attempted, stressing the importance of handling loop bounds that are not known at compile time, as well as loop bodies that contain control flow. Their results showed that aggressive loop unrolling was able to achieve a 10 to 20 percent improvement in execution time over a simple and naive approach, with some increases as much as 40 to 50 percent.

While Davidson and Jinturkar argued that the unroll factor is most easily determined late in the compilation process, Sarkar proposed a method that allows loop unrolling to be applied at an earlier stage [50]. Sarkar's method aims at reducing the complexity of the code produced by the unroll-and-jam transformation on perfectly nested loops. He provided a cost model that considers estimates on ILP, register pressure, and cache constraints. This cost model is used to analytically determine the most effective unroll factors for a set of nested loops. Experimental results indicated that the cost model always achieved better performance than fixed unroll factors. Additionally, the cost model never resulted in performance degradation. For the SPEC95FP benchmarks he examined, Sarkar concluded that register locality affected performance the most.
Kennedy and Allen identified many loop transformations for optimization in their 2002 textbook [39]. They held an interesting perspective on loop unrolling, treating it as a method for eliminating register copies arising from the use of scalar replacement on loop-carried data dependences. They presented an algorithm, applicable to high-level source code, that unrolls loops to eliminates these copies. Kennedy and Allen focused on unroll-and-jam as a viable optimization for high-performance computing, instead of plain unrolling. Unroll-and-jam is applied in the context of two nested loops; instead of unrolling the inner loop, the outer loop is unrolled. This creates two inner loops, which are then fused together. Like loop unrolling, unroll-and-jam creates a larger loop body that can improve instruction schedules. However, the iteration count of the outer loop is reduced. A significant benefit comes from improved locality in the inner loop, however, as many codes reuse data between the outer loop iterations. Unfortunately, loop-fusion is a complex transformation that requires sophisticated data-dependence analysis to ensure safety; such analysis beyond the scope of this thesis. Consequently, this thesis did not attempt to implement unroll-and-jam in the context of a low-level intermediate representation. Instead, only unrolling for inner loops is considered. Kennedy and Allen suggest that unroll-and-jam is inherently a high-level transformation; they use FORTRAN for their examples and present their algorithms as source-to-source transformations.

Next, in contrast to Kennedy and Allen’s high-level approach, Hennessy and Patterson discussed loop unrolling from the perspective of the computer architecture, a very low-level approach [36]. In the context of a MIPS-like assembly language, they demonstrated the use of loop unrolling for reducing loop overhead and increasing ILP. They explored the effects of loop unrolling on instruction scheduling for a simple five stage pipeline, showing the ability of unrolling to hide long instruction latencies. Hennessy and Patterson indicated that advanced compiler technology is required for loop unrolling at the assembly level—mainly calculating the data dependence graph—but they did not provide details about how that technology is implemented. Further,
they did not discuss the problem of identifying loops in the control flow graph, or recognizing induction variables and loop bounds, though they mentioned that loops of unknown bounds require a cleanup loop.

Finally, Cooper and Torczon discussed loop unrolling from a high-level perspective, using algol-like source code for examples [16]. They noted the potential reduction in operations executed, but suggested that the primary goal is creating better code shape for other optimizations. Specifically, improved instruction scheduling arises from the increased size of the loop-body; the schedule may achieve higher ILP and more effectively hide memory and branch latencies. Cooper and Torczon identified increased code size as the main drawback, and mention the potential degradation of instruction cache performance. They noted the difference between known and unknown loop bounds, providing a clever optimization for the cleanup loop. Finally, they mentioned that loop unrolling can eliminate copy operations arising from loop-carried data dependences. Cooper and Torczon’s discussion of loop unrolling is very high level, using source code for examples and omitting details on loop detection and induction variable analysis. This thesis, on the other hand, must address loop unrolling from a low level code representation, explicitly identifying induction variables.

2.2 Control Flow Graph Cycle Detection

CFG cycle detection is required for loop unrolling because it allows the basic blocks comprising each loop to be identified. In a high-level programming language, loop bodies are usually directly notated in the syntax, allowing easy identification. In low-level code, however, loops are constructed with a set of conditional branches that create a cycle. This cycle is visible in the CFG and can be detected with various graph-analysis algorithms. Though cycle detection is not sufficient analysis for general loop unrolling, as shown in Chapter 3, it is an important first step.

Interval analysis, attributed to Cooke and Allen in 1969 and 1970, is the earliest method for identifying loops in a flow graph [13, 14, 4]. Interval analysis is a technique
for collapsing a flow graph into nested groups of nodes, iteratively, until a single node is reached. Although this technique is often used for computing data-flow analysis information, a property of the collapsing rules is that they preserve the loop structures and nesting relationships within the interval graph. Unfortunately, the main drawback of this method is that it only works on reducible flow graphs. If a flow graph is not reducible, then the algorithm fails, producing little useful information about loop structures.

In 1972, Tarjan published his work describing the properties of depth-first search on a graph [54]. This work is important because it provided a foundation for later work on cycle analysis, but it did not directly address the problem of cycle analysis. Tarjan provided an algorithm for constructing a depth-first spanning tree (DFST) and detecting “back edges” in this spanning tree, edges that create cycles in the spanning tree. For reducible graphs, the back edges in the DFST unambiguously identify the headers of loops. Finally, Tarjan presented an algorithm for computing maximal strongly connected components (SCC) in a graph. This did not solve the problem of cycle detection, because it cannot distinguish nested loops. The SCC algorithm is used in induction variable analysis, however [63, 20].

Hecht and Ullman introduced T1-T2 analysis in 1972, independently of the work of Cocke and Allen [35]. They showed that this framework for analyzing flow graphs is equivalent to interval analysis. Their method is similar to interval analysis in that it collapses the flow graph down to a single node. They identified the base structure that is present in a flow graph if and only if it is a non-reducible graph. Tarjan then improved upon Hecht and Ullman's algorithm in 1973, publishing a method for determining the reducibility of a flow graph [55]. This method is based on his earlier work on depth-first search, and runs in a slightly better time bound than the T1-T2 method.

The loop-detection research that existed through the early 1970's, though applicable in most cases, had one major drawback: the algorithms fail in the presence
of irreducible loops. One approach to solving this problem is node splitting, where portions of an irreducible loop body are peeled for each header of the loop until a single header for the loop exists. Node splitting is beyond the scope of this thesis, but the main disadvantage is unnecessary growth in code. Several other approaches to handling irreducible loops appeared in the 1990’s. All of these publications focused on analysis techniques for identifying the nesting relationship of reducible and irreducible loops.

In 1996, Sreedhar et al. presented the DJ Graphs algorithm for identifying the nesting relationship of reducible and irreducible loops [52]. This algorithm uses the dominator tree [56], augmented with special “join” edges that represent the edges in the original control flow graph that are not part of the dominator tree. The DJ Graphs algorithm works by analyzing each level in the DJ Graph, from the bottom up. First, reducible loops are detected from join edges for which the destination dominates the source; these loops are collapsed into a single node. After all reducible loops are identified and collapsed on a given level, Tarjan’s SCC algorithm [54] is used on the nodes at the current level or greater to detect irreducible loops. Each non-trivial SCC is collapsed as an irreducible loop. This algorithm, instead of giving up in the presence of irreducible loops, is able to provide a reasonable nesting relationship between reducible and irreducible loops.

Independently of Sreedhar et al., Havlak published a similar algorithm for identifying reducible and irreducible loop nestings in 1997 [33]. Havlak’s method builds upon Tarjan’s work [55] by detecting the presence of irreducible loops, and correctly determining a valid nesting relationship with other loops. Each back edge in the DFST is treated as a loop header, just as in Tarjan’s algorithm, and the loop body is detected by “tracing” backwards through the CFG from the back edges. For reducible loops, all backwards paths from the back edges will eventually reach the loop header, because the header dominates all other nodes in the loop body. Irreducible loops cause problems, however, because the presence of additional loop entry nodes allows
the backward trace to continue past the loop body boundary. Havlak solved this by using a fast algorithm for calculating ancestor information in the DFST. The traces are terminated as soon as a node that is not a descendant of the header is reached. This node is identified as another entry to the loop, and the loop is marked as irreducible. Havlak recognized that the nesting relationship in the presence of irreducible loops depends on the DFST that is used. The problem occurs when both a reducible and irreducible loop share the same header. To maximize the number of reducible loops that the algorithm discovers, Havlak proposed a node splitting algorithm (for analysis purposes only) that forces all reducible loops to be discovered. The drawback of this algorithm, however, is that it may provide an unintuitive nesting structure of irreducible loops.

Ramalingam provided an in-depth survey of the prevalent cycle-detection algorithms in a 1999 journal publication [49]. This paper compared three CFG loop-detection algorithms: DJ Graphs, Havlak, and Steensgaard. Ramalingam identified that the Havlak algorithm incorrectly claimed to be almost linear; in support, he illustrated a case that causes quadratic behavior. The problem is that, for an irreducible loop, incoming edges for additional headers are re-pointed to the representative header for the loop. This means an edge may be processed multiple times, if there is a deep nesting of irreducible loops. Ramalingam provided a fix to this problem, by allowing these edges to be ignored until the analysis reaches the outermost irreducible loop in which such an edge enters the loop. Ramalingam also showed that the DJ Graphs algorithm is quadratic in the worst case, because of the repeated application of Tarjan's SCC algorithm for each level. The worst case occurs when there is a long chain in the dominator tree that is not part of a loop. These nodes will be repeatedly examined in the SCC algorithm. The author provides a fix that prevents such nodes from ever being examined, in a similar approach to Havlak's algorithm. Finally, Ramalingam considered Steensgaard's algorithm, which instead works top down (the previous algorithms all work bottom up) [53]. Steensgaard's algorithm identifies all non-trivial
SCCs (outer loops) first. After removing the back edges, the algorithm then identifies all SCC's that are nested within the outer loops. This process is repeated until no more inner loops are found. Ramalingam pointed out that irreducible graphs can be constructed such that all three algorithms produce different nesting forests. Further, they identify differing numbers of loops! He proposed the use of Steensgaard's algorithm in place of Tarjan's SCC algorithm within the DJ Graphs algorithm. Though Steensgaard's algorithm is quadratic in the worst case, it would be operating on subgraphs of the entire graph. This change would allow a very fine grained nesting of irreducible loops. Ramalingam suggested that the DJ Graphs algorithm produces a more intuitive loop nesting structure than Havlak's algorithm. However, he also suggested that Havlak's algorithm is simpler and easier to implement, because it does not need the dominator tree.

2.3 Induction Variable Analysis

In languages that support a strict DO-loop syntax, as described in Chapter 3, induction variable information is available with little or no analysis. In languages without strict loop syntax, however, identifying induction variables requires complex analysis. For loop unrolling, this analysis must identify loop-invariant values as well as values with invariant (or predictable) increment or decrement per iteration. Before the appearance of static single assignment (SSA) form in 1991 [22], most induction variable analysis relied on “ad hoc pattern recognition algorithms” [63]. This method was complex and usually only worked on the most common cases that arise in practice. In 1992, however, Wolfe published a simple algorithm that detects cycles in the SSA graph using Tarjan's SCC detection algorithm [63]. Wolfe identified characteristics of cycles in the SSA graph that represent several different types of induction variables: linear, polynomial, geometric, flip-flop, periodic, wrap-around, and monotonic. Wolfe's simple approach replaced the need for complex recognition algorithms for each type of induction variable. Wolfe did not provide an explicit algorithm, but in-
stead described the characteristics that must be identified for each type of induction variable. Wolfe’s motivation was that induction variable information can be used for improving the precision of dependence analysis.

In 2001, Cooper et al. published an SSA based algorithm for operator strength reduction (OSR) that uses an adaptation of Wolfe’s algorithm. Using Wolfe’s algorithm to initially detect the induction variables, the OSR algorithm then replaces expressions based on induction variables with new induction variables. The name “strength reduction” comes from the ability of this algorithm to replace multiplication operations in loop bodies with weaker (i.e., less complex and often faster) addition operations. An expression containing multiplication between an induction variable and a loop-invariant value, for example, can be replaced with a new induction variable that requires only an addition operation. A secondary goal of this algorithm is to minimize the number of redundant induction variables. The final step of the OSR algorithm, linear function test replacement, accomplishes this goal by eliminating induction variables that are used only for determining control flow. Though loop unrolling does not need the full OSR algorithm, the induction variable analysis algorithm is used for this thesis. This algorithm precisely identifies loop-invariant values and values of invariant increment or decrement (induction variables).

2.4 Adaptive Compilation

The literature on adaptive compilation has increased dramatically over the last several years. While the adaptive compilation systems that have been proposed have a range of differences, there are several fundamental elements that most systems have in common. First, a system usually tunes a set of compiler decisions for an individual input program. Second, the tuning process is usually guided by feedback from some sort of evaluation function (such as running the compiled program). Finally, this feedback process is usually iterative, allowing the system to achieve better results with more time. Given the similar structure in most adaptive compilation systems,
a rough classification is as follows:

**Individual Parameter Tuning** This category represents adaptive compilation systems that attempt to find the best parameter, or set of parameters, for an individual optimization pass. A simple example in this category is empirically determining the unroll factor for a specific loop body. Often, if the target transformation affects disjoint pieces of the code, different parameters may be tuned in parallel. This parallel tuning is possible, for example, when determining the unroll factor for two unrelated loop bodies. This flexibility comes at a cost, however, because it usually means the search space size will be dependent upon the input code. Implementing a system in this category often requires intimate compiler support to provide an interface for the fine grained tuning of parameters.

**Compiler Flag Tuning** This type of adaptive compilation system attempts to determine the most effective set of compiler flags for a given program. Most compilers support the common `-01, -02, -03 ...` flags, while many compilers give additional control for the user to toggle specific optimizations on and off. Additionally, some compilers provide flags that control the optimization parameters, such as the loop-unroll factor, the data-alignment factor, or the inlining depth. Adaptive compilation systems in this category search over all possible compiler flags and parameters (or, at least the flags and parameters that have the largest impact on performance), tuning them separately for each input file. This category is similar to the previous category, except that it is only able to tune parameters that are already exposed by the compiler. Thus, implementation usually doesn't require modification of the compiler internals. Unfortunately, unlike the previous category, the unit of compilation (often the entire file) is indivisible. That is, the same parameters are applied to the entire compilation unit. The advantage of this restriction is that it ensures the search space is fixed for all input programs.
Phase Ordering This final category comprises adaptive compilation systems that attempt to determine both the best selection and best ordering of optimization passes for a given input file. This category searches over the set of all of the permutations of optimization passes, allowing repeats. Thus, the search space is fixed for all inputs, assuming that that set of optimizations is fixed. Implementation requires significant compiler support to control the order of optimization passes. Additionally, engineering the compiler to allow arbitrary ordering of optimization passes is difficult, because many passes make assumptions about the input and output of that pass that restrict certain orderings. These restrictions either need to be respected or eliminated.

To begin, the framework used in this thesis falls into the category of a phase ordering system. The Rice University research compiler addresses the optimization pass ordering constraints through a modular design. This design requires each optimization pass to rebuild the necessary analysis information, and then leave the code in clean state. With this as a comparison system, the related work can now be examined.

Whitfield et al. propose a framework that allows analytical determination transformation interactions [62]. They present Gospel, a transformation description language that allows a formal specification of the preconditions and actions for a transformation. The authors show how analysis of the preconditions for each transformation reveal the enabling and disabling interactions between transformations. While this work isn't technically an adaptive compilation system, it recognizes the impact of transformation interactions. The authors provide empirical results that support their analytical predictions. The drawback of this approach is that the description language only works for a restricted set of transformations. Specifically, transformations requiring fix-point computation cannot be expressed in this framework. Additionally, the results of this work offer no indication of the best order in which a set of transformations should be applied. The non-existence of interactions, however, could be used for pruning the search space in a phase-ordering adaptive compiler.
The Automatically Tuned Linear Algebra Software (ATLAS) project is one of the most common examples of adaptive compilation [61]. At the core, ATLAS is a library that implements Basic Linear Algebra Subprograms (BLAS). During installation on a particular system, however, the ATLAS libraries are empirically tuned to achieve very high performance; in many cases, the automatically tuned library can achieve performance comparable to, or better than, the vendor provided hand-coded implementation. This system, to the final user, appears to be a fully-automatic individual parameter tuning approach. The performance critical sections of the library, however, have been manually parameterized to allow automatic tuning. That is, the project required significant initial manual development time to enable the automatic tuning at a later time. To repeat this procedure for a different application or library would require a similar initial development effort to manually parameterize the code. Other approaches, including this thesis, try to be more general.

In 1999, Chow and Wu present a system that determines the best set of compiler flags for the IA64 research compiler [12]. This framework can only tune the command line flags that are available to the user, which usually indicate whether or not an optimization should be applied. Some flags accept parameters, also. The authors apply “fractional factorial design” (FFD) to determine which configurations should be run to get a clear picture of the transformation interactions. Even though the system does not allow transformations to be reordered, there still exists interactions between transformations. FFD allows the a good set of compiler flags to be discovered without testing every possible configuration.

One of the early applications of genetic algorithms was done by Shielke, in Cooper’s research group [19]. Compiling for reduced code size, the authors showed that the genetic algorithm produced code up to 40% smaller than with a typical fixed sequence. Additionally, since they broke ties by execution time, the resulting code was up to 26% faster. Their experiments allowed them to create a new fixed sequence that favored smaller executables. This new sequence beat the original fixed sequence in
code size always, and sometimes in running time. Further tuning was almost always possible with the GA, however, suggesting that a universal sequence does not exist.

In 2001, Granston and Holler describe Dr. Options, a system that recommends compiler flags for Hewlett-Packard’s PA-RISC compiler [29]. The recommendations, made for an individual application, are based on information from the user, compiler analysis, and profile information. This approach emphasizes an interactive process between the developer and the compiler. Because the system limits itself to a single compilation and execution, to collect profile information, it is not technically an iterative adaptive compilation system. The system shares many of the same goals as other adaptive compilation systems, however, and is successful in tuning the flags to achieve an improvement over the default options.

A 2002 journal paper by Cooper et al. provides a very good motivation for adaptive compilation, as well as an outlook for future work [21]. They suggest that compiler writers have been successful in writing strong compilers for a single architecture, or “retargetable” compilers that are not strong optimizers. Unfortunately, combining the two properties has been elusive. An analytical approach to compiler design is difficult, because the effects of passes are very difficult to predict without actually compiling the code. Thus, adaptive compilation empirically “computes” the solution to the design problem. The authors compare two search algorithms, the genetic algorithm (GA) and the hill climber (HC). The HC gets within 15% of the optimal very quickly, but then unfortunately slows down. Although the GA takes longer, it is able to get much closer to the optimal. An enumeration of a restricted search space suggests that good solutions, within 1% of optimal, are very rare (about 1 in 30,000). Further, the hill climber behavior suggests that local optima are common. The authors argue that this result favors using GA over the HC. This decision, however, would clearly be affected by the search budget.

Knijnenburg et al. present an approach to adaptive compilation that automatically determines loop tile sizes and unroll factors for unroll-and-jam [40]. They focus
mainly on small computation kernels, but show preliminary results for entire applications. Their system performs source-to-source transformations, which are then fed to the backend native compiler. They restrict the unroll factor to be between 1 and 20, and the tile sizes to be between 1 and 100. Thus, their search space is, at most, 200,000 points. The authors examine several search algorithms, but show that the random search is often just as effective as the more intelligent search methods. By only considering square tile sizes, they are able to reduce their search space size by a factor of 100. In this new space, the “Pyramid” or “Grid” search method works the best. This algorithm evaluates a sparse grid of points over the search space, and then performs more detailed localized search near the best points from the grid. Finally, the authors show that static cache models can be used to further prune the search space. This approach focuses on reducing the execution time for adaptive compilation by ignoring configurations that are predicted to exhibit poor performance.

Building upon Knijnenburg et al., Fursin et al. examine the impact of tuning optimization parameters for an entire program [27]. This paper considers three optimizations: array padding, loop unrolling, and loop tiling. The authors show that their adaptive compilation techniques can achieve, on average, a 20% speedup with only 15 search iterations.

Cooper and Waterman extend the work of ATLAS by examining the capability of a more general approach to empirical tuning [15]. ATLAS requires significant programmer effort to parameterize each routine, enabling automatic tuning during installation of the library. Cooper and Waterman show that, by tuning the blocking factors through the MIPSpro compiler command line options, performance comparable to ATLAS can be achieved. These results support the argument for a more general approach, because it does not require the initial parameterization of the source code. While this result is promising, the authors warn that current compilers do not expose enough parameters to make this sort of general adaptive compilation possible.
Triantafyllis et al. recognize that the "predictive heuristics" used in compilers have major drawbacks [57]. While usually effective on average, the heuristics often fail to correctly predict the profitability of a transformation for an individual piece of code. The authors credit adaptive compilation as a solution to this problem, but criticize the large compile times required by these methods. They propose an aggressive pruning of the search space at compiler-construction time that results in a small tree of interrelated optimization configurations. The child relationship in the tree indicates that, if the parent configuration was effective, then it is likely that the child configuration will also be effective. The search space comprises a set of command line and internal options for the Intel Itanium compiler. At compile time, the configurations at the top level of the tree are evaluated with a static performance estimator. The configuration tree is then traversed in a greedy fashion, never backtracking. This approach results in a very practical approach to adaptive compilation, achieving significant performance increases without incurring unacceptable compile times. The nature of the aggressive pruning, however, inherently limits the potential improvement from adaptive compilation.

Almagor et al. published a detailed study on the characteristics of the adaptive compilation search spaces in 2004 [6]. The authors argue that building effective adaptive compilers requires a "fundamental understanding of the search spaces and their properties." To gain this understanding, they exhaustively enumerated five subsets of the full search spaces for three small programs. The spaces contained sequences of length 10, selected from a pool of five transformations. The study resulted in several interesting observations about the search spaces. The spaces contain many local optima, some very good and some very bad. The distance from a random point to a local optimum is very small, however, at most 16 in the smaller space. (Exploration algorithms on the large space also show this characteristic.) Next, the distribution of solutions can be sufficiently modeled with a random probe of only 1000 sequences. Finally, the distribution of solutions differs between input programs and compiler
transformations. Using the knowledge about the small spaces, the authors design algorithms to effectively search the space. Their results show that an impatient hill climber with randomized restarts can find very good solutions on both the small space as well as the large.

Cooper’s research group published a second paper on search space structure, building on the work of Almagor et al. [17]. The same datasets are used from the enumerated spaces, but more detailed analysis provides further insight into the choice of search algorithms. The main findings suggest that the search spaces contain many local minima, but the probability of a random sequence being within 1% of the optimal is very low. Further, they discovered the presence of “phase transitions” in the probability of finding a solution within x% of the optimal. For example, in the benchmark fmin, very few sequences were within 2.6% of the optimal, while many good sequences existed just past that threshold. In exploring the full space, the authors considered three algorithms: random probing, the hill climber (HC), and the genetic algorithm (GA). All three algorithms were able to improve upon the standard fixed sequence, but the distinguishing factor was the search cost. The benchmarks split into two categories: those for which the hill climber dominated the cost-efficiency, and those for which the random probe was equally cost-efficient. This suggests that some spaces contain many good solutions, allowing random probe to be a feasible option, while other spaces contain relatively few good solutions, requiring the intelligent search strategy of the hill climber. While the genetic algorithm always found solutions of comparable quality, it was also the most time consuming method.

Kulkarni et al describe VISTA, a phase-ordering adaptive compilation system very similar to the Rice University system [44]. VISTA provides an interactive user interface that allows the developer to control and monitor the compilation process. Their system includes a genetic algorithm that searches for effective compilation sequences. They propose several techniques for reducing the compilation time, such as detecting redundant sequences and identifying equivalent function instances. Using careful
analysis of the enabling/disabling effects of each transformation pass, they are able to further reduce the sequences that need to be evaluated. This work shows that up to 84% of evaluations can be eliminated with their techniques. They also show that the GA can be tuned to find the same quality solutions with only one third of the generations. They achieve this by customizing the GA parameters, such as the initial population, the sequence length, and the available transformation pool.

One of the main problems with adaptive compilation is the high compile time cost, associated with evaluating each compilation sequence in the search. Cooper et al. solve this issue with a method called virtual execution [18]. Instead of executing the program for each possible sequence, the authors propose that the profile data from a single execution can be used to "calculate" an estimation of the running time resulting from any compilation sequence. They show how to correlate block counts from an unoptimized execution trace to the transformed code. For passes which don't change the control flow graph, this is trivial. For passes that do, such as loop peeling, this requires a complex analysis pass that is not guaranteed to correctly determine the running time. Thus, this method is estimated virtual execution (EVE). Results show that EVE is within 2-3% correct 95-99% of the time, and can be used as an effective evaluation function. Experiments with the hill climber show that EVE does not degrade the effectiveness of the search, as it finds solutions equivalent to those found by actually running the program for each sequence. The efficiency of the search, however, increases significantly. The search times drop from 5% to 85% of their original duration.

Cooper et al. also address the usability of adaptive compilation systems, arguing that it is a major obstacle to widespread acceptance of this method [18]. Their system incorporates a graphical user interface (GUI) that eases configuration of the search algorithms and parameters. Additionally, this GUI provides live feedback while the search proceeds, allowing the user to monitor progress. The authors suggest, from experience, that close monitoring is vital to achieving good results. Additionally, they
provide a “stop” button. Since the search methods are anytime algorithms, the search may be halted at any time and the best result up to that point will be returned.

Waterman applied adaptive compilation techniques to the problem of procedure inlining [58]. Tuning inlining decisions adaptively, using executing time as the feedback metric, Waterman showed that significant improvement can be gained over static inlining heuristics. This approach tunes the parameters for a single optimization.

Agakov et al. describe an adaptive compilation system very similar to that used in this thesis [1]. Using the SUIF compiler from Stanford [32], they propose improvements to the search algorithms for the phase ordering problem. Unlike the work in this thesis, Agakov et al. use a source-to-source framework that relies on a “back-end” native compiler. Similar to Almagor et al. [6], they exhaustively enumerate a smaller search space to gain insight into their search algorithms. The smaller space contains sequences of length 5, taken from a pool of 14 transformations. Their results indicate that the genetic algorithm and random probing methods both achieve good results, but also require a very large number of search iterations. To reduce this time, they apply machine learning techniques to allow a biased random sampling method to focus the search in the profitable regions of the space. With 1000 search evaluations, they “train” two models. The Independent Identically Distributed (IID) model records the probability that a certain transformation will appear in a good sequence. The Markov model records the probability that a certain transformation will follow some other transformation. Several different models are created using a collection of input programs and data files. Each input program is classified according to 36 program features, and the model most closely resembling the input program is used. The results are very promising, indicating that the learned models can reduce the search time by an order of magnitude.

Building on their previous work, Kulkarni et al. show that it is sometimes possible to completely enumerate the search space [45]. They recognize that the space of “function instances”, for their compiler, is much smaller than the search space. They
analytically determine transformation interactions and use that information to prune the search space accordingly. Detecting equivalent function instances reduces the time required to execute the program. Out of the 111 functions in their benchmark suite, they are able to completely enumerate the search space for all but two. Given this large amount of data, they are then able to empirically determine the probability that a transformation will enable or disable another transformation. Using this enabling/disabling information that has been empirically determined, they show how it can be used to improve the efficiency of their "batch compiler".
Chapter 3

Loop Unrolling

Traditional compiler writers have long understood the importance of tuning a compiler’s set of optimizations to exploit the opportunities that are available in the code [51]. Following this methodology, Section 3.1 begins this chapter by presenting the case for adding a loop unroller to the Rice University adaptive compiler. In the context of adaptive compilation, this careful tuning will lead not only to improved compiler output, but also to reduced compile times.

Most programs utilize loops that perform similar operations (the loop body) multiple times (the iteration count). As a result, nearly all programming languages provide a method for writing loops. Section 3.2 shows how differing loop syntax can represent the same behavior, but that the structure of the syntax has a direct impact on the complexity of compiler analysis and transformation. With the highest-level loop syntax, loop unrolling only requires simple analysis and is generally straightforward. Thus, the mechanics of loop unrolling are generally ignored from a research perspective and treated, instead, as an engineering problem. However, this thesis targets loop unrolling for the lowest-level loop syntax, unstructured loops, and therefore requires the combination of two complex analysis phases for loop unrolling to be safe in the general case. Furthermore, the flexible ordering of transformations in an adaptive compiler can create unusual loop structures that previously described control flow graph (CFG) analysis methods do not address. Section 3.2.1 illustrates several of these loop anomalies, and proposes a method for handling them.

After presenting the semantics of the loop unrolling transformation, Section 3.3 describes an algorithm that identifies and safely unrolls unstructured loops. This
approach combines previous work of two distinct areas of research, CFG analysis [49] and induction variable detection in the static single assignment (SSA) graph [63]. The unrolling algorithm is presented in two steps, each step consisting of an analysis phase and a transformation phase. Section 3.3.1 describes the first step, naive unrolling, which requires only CFG analysis. Finally, Section 3.3.2 presents the second step, clean (or general) unrolling, which requires induction variable analysis in addition to the CFG analysis. Clean unrolling, because of the extra analysis, is usually much more powerful than naive unrolling.

Finally, Section 3.4 concludes this chapter with a brief examination of several additional considerations for loop unrolling. First, some architectures provide a hardware looping construct that introduces subtle implications to the proposed loop-unrolling algorithm. Next, there are cases in which the loop-unrolling algorithm fails. It is worth noting that some cases not handled by the loop unrolling analysis could be addressed by applying the proper enabling transformations before loop unrolling is applied. In a traditional fixed-sequence compiler, this would be the desired action. Chapter 4, on the other hand, shows that the correct enabling transformations are automatically discovered by the search algorithm in an adaptive compiler.

3.1 Motivation

During initial experiments with the adaptive compiler, visual inspection of the results indicated that the adaptive compiler often selected loop peel many times consecutively. For example, consider the top five strings, from experiments based on Grosul’s work, for adpcm_coder, shown in Table 3.1 [30]. Each of these strings contain loop peel, notated with the letter p, in seven of the ten positions. These results are not specific to adpcm_coder, although this benchmark exhibits the most extreme results; Table A.2 in the appendix presents similar results for the other eight benchmarks.

\(^1\)Refer to Table A.1 in Appendix A for a complete listing of the compiler transformations and their alphabet letter mappings.
Table 3.1: The five best sequences found by the hill climber for `adpcm_coder`

<table>
<thead>
<tr>
<th>String</th>
<th>Instruction Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>ppppcczpc</td>
<td>8 983 004</td>
</tr>
<tr>
<td>pppppclpc</td>
<td>8 983 004</td>
</tr>
<tr>
<td>pppppclpd</td>
<td>8 983 744</td>
</tr>
<tr>
<td>pppppclpg</td>
<td>8 983 744</td>
</tr>
<tr>
<td>pppppclpm</td>
<td>8 983 744</td>
</tr>
</tbody>
</table>

Table 3.2: Frequency of loop peel in sequences within 1% of best

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Loop-peel Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm_coder</td>
<td>42.8%</td>
</tr>
<tr>
<td>adpcm_decoder</td>
<td>21.8%</td>
</tr>
<tr>
<td>applu</td>
<td>40.0%</td>
</tr>
<tr>
<td>matrix300</td>
<td>13.1%</td>
</tr>
<tr>
<td>rkf45</td>
<td>41.5%</td>
</tr>
<tr>
<td>seval</td>
<td>15.3%</td>
</tr>
<tr>
<td>solve</td>
<td>30.5%</td>
</tr>
<tr>
<td>svd</td>
<td>24.0%</td>
</tr>
<tr>
<td>tomcatv</td>
<td>4.8%</td>
</tr>
<tr>
<td>average</td>
<td>26.0%</td>
</tr>
</tbody>
</table>

A clearer quantification of this irregularity is seen by examining the frequency of loop peel in the best sequences for all benchmarks. Table 3.2 presents the loop peel frequency for sequences within 1% of the best, showing that many of the benchmarks exhibit a high frequency of loop peel. The average occurrence rate of loop peel is 26%. Figure 3.1 plots the average frequency of each transformation, showing that the average frequency of loop peel is much higher than that of any of the other transformations. This behavior by the adaptive compiler demonstrates an unusual bias towards selecting loop peel. Because the adaptive compiler only selects transformations that are profitable, there must be some advantage in applying loop peel more than once. As the next paragraph describes, the profitability comes from an unintended source.
Loop peel was originally written as an enabling transformation for loop unswitching [16] or Cytron et al. style code motion [23]. There are two reasons why this intended use does not explain the high occurrence rate of loop peel, however. First, neither of the target transformations were ever implemented in our compiler; loop peel could not possibly be creating opportunity for a transformation that does not exist. Second, even if the target transformations were present, we would expect that only one or two applications of loop peel would be needed to enable the code for later optimization. With occurrence rates of up to 70%, loop peel is not behaving as an enabling transformation. Instead, the adaptive compiler found a secondary, unintended use for loop peel: reducing loop overhead. Each successive application of loop peel can potentially reduce loop overhead for a single loop iteration. This explains why the adaptive compiler would select the transformation so frequently; if no other opportunities were available, then loop peel could at least remove a small number of instructions of loop overhead.

While loop peel can reduce the loop overhead for a single iteration, the more powerful loop-unrolling transformation can reduce loop overhead for the entire loop. In other words, for a loop with \( n \) iterations, loop peel can decrease the dynamic instruction count by \( O(1) \) and loop unrolling can decrease it by \( O(n) \). By adding
a loop unroller to our compiler, we expect the adaptive compiler to discover that a single application of unroll is more effective than multiple applications of peel. This should result in a decrease of the frequency of loop peel. Additionally, for loops with high iteration counts, the use of loop unrolling should improve performance beyond the improvement possible with loop peel. That is, the compiler will be able to reduce loop overhead much more effectively with unrolling than with peeling.

Furthermore, loop unrolling also functions as an enabling transformation, similar to loop peeling. A larger loop body is created through cloning, potentially providing subsequent optimization passes with more context for optimization. Peeling and unrolling both clone the loop body, and therefore may create many of the same optimization opportunities. However, the cloned blocks resulting from peel will be outside of the loop body, while the cloned blocks resulting from unroll will be within the loop body. Thus, the opportunities created by unrolling may have a much larger impact on performance. Consequently, we expect the adaptive compilation search to find better compilation sequences than previously possible.

### 3.2 Background

Loop unrolling is a simple, but powerful, loop transformation that increases the size of the loop body while reducing the loop iteration count. This is accomplished by executing multiple iterations of the original loop in a single iteration of the unrolled loop. The number of iterations that are combined in a single unrolled iteration is specified by the unroll factor. Loop unrolling has several effects [16, 36, 5, 60, 46]:

**Positive Effects of Unrolling**

**Reduced loop overhead** Because the total iteration count decreases with loop unrolling, the number of loop-exit tests and conditional branches will decrease. Further optimization can potentially reduce the induction-variable increments as well.
**Increased ILP** In the absence of control flow, loop unrolling creates longer segments of straight-line code. This can increase the amount of instruction level parallelism, potentially resulting in improved instruction scheduling for superscalar or VLIW processors. In the presence of ambiguous memory references, however, the instruction scheduler may be constrained [60].

**Increased optimization context** Loop unrolling can expose optimization opportunities, across loop iterations, that were previously hidden. Redundant expression elimination is likely to be quite effective in loops with consecutive array accesses. Mahlke *et al.* show that ILP is not improved much by unrolling alone. They suggest applying additional optimizations, such as register renaming, strength reduction, and tree height reduction [46]. Alexander *et al.* show how loop unrolling can be used to exploit a wide data bus [3]. Unrolling may expose several consecutive single-word memory accesses that can be combined into a single multi-word access, reducing the number of memory accesses. Opportunities for scalar replacement may be exposed with loop unrolling [8].

**Negative Effects of Unrolling**

**Increased code size** The executable code size will increase with loop unrolling, because cloning the loop body adds duplicated code to the program. Weiss *et al.* show that increased instruction cache misses may result if unrolling increases the code size too much [60]. Fortunately, instruction-cache sizes have increased dramatically since Weiss *et al.* published their study in 1987. It is likely that the issue of increased code size is not as important today.

**Increased register pressure** Loop unrolling, by itself, will not increase the register pressure, because the cloned iterations do not overlap. However, ILP will be severely limited unless register renaming is applied. By renaming registers in the cloned iterations, the false dependences disappear and the instruction
scheduler is free to exploit any existing ILP. Once the instructions from different iterations are overlapped, the register pressure will increase. If the register pressure becomes too high, the register allocator may need to insert additional spill code.

**Cleanup loop overhead** Loop unrolling, in the general case, requires a special cleanup loop to handle cases when the dynamic iteration count is not a multiple of the unroll factor. At run time, a conditional expression must be checked to determine whether or not the cleanup loop is needed. This conditional expression lies on the critical path of the program and cannot be avoided unless the iteration count is known at compile time. If the dynamic iteration count is low, and the loop is nested within other loops, then the cleanup loop overhead may be significant. For example, if the dynamic iteration count is always less than the unroll factor, then the unrolled loop will never be executed. That is, the benefits of unrolling will not be realized, but some of overhead will be experienced.

For loop unrolling to be profitable, the positive effects must outweigh the negative effects. In general, this means that the decreased loop overhead, increased opportunity for instruction level parallelism (ILP), and improvement from other optimizations must outweigh the increased code size, increased register pressure, and cleanup loop overhead. The unroll factor controls the degree to which these effects are exhibited. In other words, a large unroll factor can reduce loop overhead much more than a small unroll factor, but a small unroll factor is less likely to induce extra register spilling. Therefore, the main problem of loop unrolling is often viewed as selecting an appropriate unroll factor.

A simple approach for loop unrolling limits the unroll factor based on the size of the loop body; this effectively bounds code growth and register pressure increases. Unfortunately, it also bounds the reduction in loop overhead. Other, more elaborate methods, such as that proposed by Sarkar, attempt to estimate the effects of loop
for \( i = 1 \) to 97 step 4 do
\[
a(i) = b(i) + c(i);
\]
\[
a(i+1) = b(i+1) + c(i+1);
\]
\[
a(i+2) = b(i+2) + c(i+2);
\]
\[
a(i+3) = b(i+3) + c(i+3);
\]
endfor

Figure 3.2: The simple loop, shown on the left, is unrolled by a factor of four on the right.

unrolling to determine the most effective unroll factor [50]. Qasem et al. argue that a fine-grained tuning approach is most effective, suggesting that the best unroll factor should be determined independently of other unroll factors [48]. However, because this thesis examines loop unrolling in the context of transformation ordering, instead of parameter tuning, the primary goal is not to determine the most effective unroll factor for each loop. Instead, loop unrolling is applied uniformly to all inner loops, using an unroll factor of four. To achieve higher unroll factors, the adaptive compiler can simply apply the transformation multiple times. This approach allows loop unrolling to fit into the transformation ordering framework, while retaining the ability to perform at least a coarse-grained tuning of the unroll factor.

Loop unrolling is best understood by first examining high-level source code. Consider the loops in Figure 3.2. For the original loop on the left, the total iteration count is 100 and the loop body contains one statement. Unrolling this loop by a factor of four produces the loop on the right. This new loop body contains four statements, but the iteration count has been reduced to 25. Notice that the increase in the loop body size and the decrease in iteration count correspond to the unroll factor.

The loop bounds are not always known at compile time, however. The loop on the left in Figure 3.3 presents a situation where the iteration count is \( n \). Loop unrolling can still be applied in this case, as shown in the code on the right, but care must be
for i = 1 to (n - 3) step 4 do
  a(i) = b(i) + c(i);
  a(i+1) = b(i+1) + c(i+1);
  a(i+2) = b(i+2) + c(i+2);
  a(i+3) = b(i+3) + c(i+3);
endfor

for i = 1 to n step 1 do
  a(i) = b(i) + c(i);
endfor

Figure 3.3: Because the original loop on the left contains unknown loop bounds, the unrolled loop on the right must execute a cleanup loop to handle any remaining iterations.

taken to produce correct code for all values of n. Specifically, the unrolled body of the loop must be followed by a cleanup loop that handles remaining iterations when the iteration count is not a multiple of the unroll factor. For example, if n = 103, then the unrolled body would execute 25 iterations (100 iterations of the original loop) and the cleanup loop would execute the three remaining iterations.

Although these examples are relatively simple, they represent the predominant convention for notating loops in high level source code. Nearly all programming languages provide syntax for representing loops that consist of a loop body and an iteration mechanism, such as an induction variable and test or iteration count. The examples in Figures 3.2 and 3.3 contain a type of loop known as a DO-loop or a FOR-loop, as indicated on the left in Figure 3.4. This syntax clearly indicates the loop body (enclosed by the DO and END DO keywords) and the induction variable and upper bound (specified by i = 1, 100). Loop unrolling is straightforward for these types of loops because the syntax defines all of the required information. An example of a DO-loop is the FORTRAN DO-loop.
As shown in the middle in Figure 3.4, the same loop can be represented as a WHILE-loop. This type of loop indicates the loop body (enclosed by the WHILE and END WHILE keywords), but it does not indicate the induction variable or loop bounds. Instead, a simple loop-exit test is provided; extra induction variable analysis is required to perform loop unrolling because the loop-exit test does not indicate the induction variable increment. In fact, a while loop may not even use an induction variable, and therefore can't always be unrolled. While loops are included in C, and the general case of a C for-loop is a WHILE-loop.

Finally, the last type of loop, a GOTO-loop, is shown on the right in Figure 3.4. A GOTO-loop encodes the same loop behavior as the other two examples, but the new syntax does not clearly indicate the loop body or the induction variable. For this type of loop, loop unrolling requires complex analysis to identify the loop body and the induction variables.

Although each loop structure in Figure 3.4 represents the same semantics as the others, the differing syntax exposes different amounts of information. The leftmost DO-loop reveals both the loop body and the induction variable information, while the rightmost GOTO-loop reveals neither. In the middle, the WHILE-loop only indicates the loop body. In general, it is easy to transform code between representations from the left to the right. Such a transformation represents a loss of information in the syntax of the program. In reality, the information is not lost, it is just hidden. Thus, to transform code from a representation on the right to one on the left, extra analysis is required to discover the information that is hidden. Control flow analysis is needed to identify the loop body, and induction variable analysis is needed to identify the induction variable and determine the iteration count. Further, it is not always possible to transform a loop from a form on the right to a form on the left. For example, loops without induction variables cannot be transformed into a DO-loop.

Although the GOTO-loop syntax is rarely used in high level programming languages, assembly language and many low-level intermediate representations use this
unstructured syntax for representing loops. Often, assembly languages do not provide syntax for DO-loops or WHILE-loops, so all loops must be translated into GOTO-loops. ILOC, the intermediate representation used for this thesis, represents all loops as GOTO-loops. Thus, this thesis focuses on unstructured loops, because the loop unroller needs to operate on the ILOC representation. As indicated, loop analysis on unstructured loops is much more complex than on structured loops that encode information about the loop. The next section illustrates the difficulty of the analysis that is required for identifying the loop body of unstructured loops.

3.2.1 Obstacles to Unrolling Unstructured Loops

This section describes several problems that arise while identifying and unrolling unstructured loops. First, because this situation requires control flow analysis to identify the loop bodies, irreducible loops can cause many traditional approaches to fail [33]. Next, there exist some ambiguous unstructured loop nestings for which control flow analysis is not enough to determine the true meaning of the code. Using induction variable information, however, in addition to the CFG analysis, is enough to properly identify the meaning of the underlying code. Finally, there are cases where loop unrolling may require extra care to prevent inefficiencies in the output code.
Handling Irreducible Loops

Cocke and Allen presented interval analysis, the earliest method for identifying loops in a CFG, in 1969 and 1970 [13, 14, 4]. This method iteratively collapses the CFG into smaller derived graphs through the application of simple reduction rules, ultimately reaching a single node. The derived graphs reveal information about the structure of the original graph, specifically loop nestings. Unfortunately, this method only works on a specific class of flow graphs, reducible graphs. Irreducible graphs, on the other hand, are the class of flow graphs for which interval analysis fails. These graphs get their name because the reduction rules in interval analysis are unable to reduce the graph to a single node. Hecht and Ullman identify the subgraph, shown in Figure 3.5, that is present in all and only irreducible graphs [35]². An irreducible loop, therefore, is a loop that contains this subgraph. One informal definition of an irreducible loop is a loop that contains multiple entry points. The cycle involving nodes B and C can be entered through either of those nodes. Instead of having a single header that dominates all nodes in the loop, an irreducible loop has multiple headers that collectively dominate all nodes in the loop [52].

²This irreducible subgraph may not be obvious in the original CFG; however, after interval analysis creates a sequence of derived graphs, this subgraph will always become apparent in an irreducible graph. This graph is what prevents the interval analysis from continuing.
From the perspective of loop unrolling, an irreducible loop poses challenges in recognizing the induction variable. Wolfe defines that a linear induction variable, in SSA form, has a single phi-node at the header of the loop. This corresponds to the merge of possible values for the induction variable at the head of the loop. The possible values comprise the initial value for the first iteration and the incremented value for later iterations. If a loop contains multiple entry points, however, the induction variable will require phi nodes at each entry point. This corresponds to the merge of the initial value at each entry point with the value of the induction variable on later iterations. Figure 3.6 provides an example of an irreducible loop, in SSA form. Several characteristics of this example are identified to help persuade the reader that the analysis of irreducible loops is difficult.

First, the loop behavior can be completely determined by the path through which the loop was entered. Figure 3.6 shows that the loop with blocks A and B can be entered from either of the blocks. The landing pads for each of those header nodes are denoted with labels P.A and P.B. A landing pad is the only predecessor of a loop header that is not contained in the loop body. Landing pads can always be constructed by creating a new predecessor of the loop header. Then all edges originating outside of the loop body that point to the loop header are redirected to this new landing pad. The landing pads P.A and P.B each define the parameters that control the loop: the initial value of the induction variable, the loop-invariant induction variable increments, and the loop upper bound. That is, each landing pad sets these loop-controlling parameters to possibly distinct values. Thus, the loop behavior, such as the iteration count and induction variable increment, depends entirely on which entry path is taken into the loop.

Next, Wolfe’s method for identifying induction variables relies on a linear induction variable having a single phi node at the header of the loop. The loop in Figure 3.6 clearly contains multiple phi nodes. While the variable i is still an induction variable, it requires extra analysis to determine that the phi nodes all correspond to entry points
Figure 3.6: Irreducible Graph Complexities
in the loop. These phi nodes simply allow the induction variable to take on different initial values, depending on the entry path into the loop. With proper recognition of these extra phi-nodes, the induction variable analysis algorithm behaves as before. The induction variable will still be an SCC in the SSA-graph, and each non-phi-node operation must be a addition or subtraction operation between the induction variable and a region constant. The algorithm for identifying region constants, however, is much more difficult and requires considerable modification.

The method described by Cooper et al. for identifying region constants relies on the dominance relation [20]. Block A dominates block B if block A lies on every path from the start of the procedure to block B. The dominance relation is reflexive, but strict dominance is not; that is, a block does not strictly dominate itself. Thus, a value is region constant if its definition is in a block that strictly dominates the header of the loop. This ensures that the definition is outside of the loop, and therefore invariant within the loop. For an irreducible loop, this definition of region constant must be extended to account for multiple loop headers. In general, to be region constant a variable must not change value within the loop body. For reducible loops, the strict dominance property ensures this. Because irreducible loops have multiple headers, however, the dominance relationship only works when the definition strictly dominates all headers of the loop. As shown in Figure 3.6, the definitions for variables x, y, and z do not strictly dominate all loop headers. Therefore, the definition of region constant for irreducible loops must be extended with the idea of collective dominance, which was introduced by Sreedhar et al. to describe irreducible-loop headers [52]. A value is region constant in an irreducible loop if it is collectively dominated by several definitions outside of the loop body. That is, each loop header must have an incoming definition. Further, in SSA form a region constant may be part of an SCC, because of the phi-nodes potentially required for each loop header. To distinguish this SCC from an induction variable, the SCC must not contain any modification operations. In other words, the SCC may only contain phi-nodes. If this property can be verified,
then the SCC is a region constant. In the example, the variables $x$, $y$, and $z$ can be identified as region constants, because each variable is in an SCC that contains only phi-nodes.

Finally, the state of the induction variable at the loop exit conditional may differ depending on whether the induction variable increment has been fully applied or not. For example, consider the case in Figure 3.6 where $x$ and $y$ are both assigned the value 1. If the loop enters through header A, then the induction variable will have been incremented by two at the time it reaches the loop exit conditional. If the loop enters through header B, however, the induction variable will have only been incremented by one. This inconsistency makes the calculation of upper bounds for unrolled loops very difficult.

Because of the complications described above, as well as the low frequency of irreducible loops occurring in practice [16], the algorithm in this thesis chooses to ignore irreducible loops. Several algorithms have been developed over the years for identifying reducible and irreducible loops [53, 52, 33], and Ramalingam suggests that the DJ Graphs algorithm by Sreedhar et al. produces the most intuitive loop nesting [49]. Thus, this thesis uses the DJ Graphs algorithm to identify all loops in the program, allowing the irreducible loops to be skipped for the purposes of loop unrolling.

Handling Ambiguous Control Flow

Disregarding the infrequent occurrence of irreducible loops, control flow analysis generally does very well at identifying loops. One exception, however, is when multiple loops share a common header node. The sharing creates an ambiguous loop nesting relationship that cannot be precisely identified without further analysis. In other words, control flow analysis alone is not enough to determine the exact loop nesting structure. Existing methods identify loops based on header nodes, and therefore will treat a set of nested loops with a shared header as a single loop. Although this in-
accuracy may be acceptable for region based transformations that use loop nestings as a guide to program structure, it becomes problematic when the loop is the focus of the transformation. The following paragraphs illustrate this problem, and propose a solution that leverages induction variable analysis to increase the accuracy of the loop identification.

Consider the loop in Figure 3.7. The maximal strongly connected component (SCC) contains nodes A, B, and C. However, there are two smaller SCCs within the maximal SCC. A and B create an SCC, as do A and C. Nodes B and C both have a distinct back edge to the header, block A. We could classify this as a single loop, with multiple paths through the loop body. This scenario, the most likely, is depicted in Figure 3.8. The corresponding pseudocode is shown below the flow graph. The node E represents an empty block, which could be removed by a control flow cleanup pass to create the ambiguous graph in Figure 3.7. This situation is easily created by the creation of superblocks [37].

Unfortunately, the same flow graph in Figure 3.7 might also represent a set of nested loops. Figure 3.9 gives an example of the flow graph and pseudocode that could create this situation. This time, the inner loop comprises nodes A and B, while the outer loop adds nodes E and C. Nodes A and E are the headers for the inner and outer loops, respectively. The inner loop contains a break condition that exits both the inner and outer loop nestings. This example shows that the header of the outer loop is empty. Removing the empty block E with control flow cleanup results in the graph in Figure 3.7. While this example is unlikely in practice, it could happen after a series of transformations optimize the original code.

The point of considering these situations is not to belabor the contrived examples. Rather, it is to point out the inherent limitations of using control flow analysis exclusively. The additional information provided by induction variable analysis can help disambiguate these examples. For example, the flow graph in Figure 3.8 can be identified as a single loop if the two back edges are controlled by the same induction
START:
  i=0
  n=100
  A: c = ((i mod 2) == 0)
     BR c, B, C
  B: /* Task A */
     c = (i < n)
     JUMP E
  C: /* Task B */
     c = (i < n)
     JUMP E
  E: BR c, A, END
END:

START:
  i=0; j=0
  n=100; m=100
  E:
    A: i = i + 1
    c = (i < n)
    BR c, B, C
    B: /* Inner loop task */
       c = Interrupt?()
       BR c, END, A
    C: i = 0
      j = j + 1
      c = (j < m)
      BR c, E, END
  END:
variable conditional expression. It is important to note that this analysis only helps on loops that are controlled by an induction variable counter. The problem of disambiguating general nestings of WHILE-loops cannot be solved with induction variable analysis. This limitation can be ignored in the context of this thesis, however, because clean unrolling cannot be applied to loops that are not controlled by an induction variable.

**Preventing Inefficient Unrolling**

A final obstacle for unrolling unstructured loops is the possibility of producing inefficient code. Consider the loop in Figure 3.7 when it is a single loop (as suggested by Figure 3.8). Because the loop exit test exists in two blocks, the unrolling algorithm will choose one of those branches on which to unroll the loop. Thus, the loop exit branch in block B will be eliminated through the block chaining in the unrolled body. The exit branch in block C will remain, however, appearing in every cloned iteration in the unrolled loop body. This exit branch would act as a break condition in the loop, possibly limiting the effectiveness of other optimizations. If the loop unrolling algorithm were to recognize that the two exit branches compute the same exit condition, however, the branch could be reverse-hoisted to achieve the code in Figure 3.8. This transformation partially removes the effects of superblock creation, but results in a more efficient unrolled loop body.

### 3.3 Loop Unrolling

Section 3.2.1 showed why loop detection of unstructured loops is a difficult problem; therefore, the main challenge for loop unrolling in this context is the analysis required for loop detection. To facilitate clear understanding of the loop unrolling algorithm proposed in this thesis, the next two sections present the algorithm as a series of smaller analysis and transformation steps.

The analysis can be broken into two individual pieces, CFG analysis to detect
the loop bodies and SSA-graph analysis to detect the induction variables and region constants. Given this information, the compiler can safely apply the loop unrolling through a series of four code transformations. The entire algorithm can be summarized in the following steps:

1. [Analysis] CFG Loop Detection
2. [Transformation] Block Cloning
3. [Transformation] Block Chaining
4. [Analysis] SSA-Graph Induction Variable Detection
5. [Transformation] Loop Bound and Guard Condition Calculation
6. [Transformation] Exit-Branch Elimination

This thesis makes a distinction between naive unrolling and clean unrolling or general unrolling. Naive unrolling only duplicates the loop body, while clean unrolling also adjusts the loop bounds and eliminates exit branches in the unrolled loop body. Naive unrolling may create larger extended basic blocks that may enable other optimizations, such as redundancy elimination. An extended basic block is a sequence of blocks in the CFG such that each block, except the first, has a single predecessor [2]; in other words, an extended basic block is a maximal path in the CFG that contains no control flow joins. Naive unrolling does not necessarily reduce loop overhead, however. The advantage of naive unrolling is that it only needs partial information for unrolling, hence the name. Specifically, naive unrolling does not require induction variable information. Naive unrolling can be applied either without performing SSA-graph analysis, or in cases in which SSA-graph analysis determines that a linear induction variable is not present. Section 3.3.1 presents the first three steps of the loop unrolling algorithm, effectively a naive unrolling algorithm.
Extending the naive unrolling algorithm with steps 4-6, Section 3.3.2 presents the clean unrolling algorithm. With induction variable information from the SSA-graph analysis, the compiler is then able to adjust the loop bounds and eliminate the exit branches.

### 3.3.1 CFG Analysis and Naive Unrolling

This section presents a naive unrolling algorithm, which is simply the first three steps of the entire loop unrolling algorithm. This algorithm is naive because it only uses CFG analysis and does not rely on SSA-graph analysis. The information needed for naive unrolling corresponds to the WHILE-loop described in Figure 3.4, because naive unrolling only requires information about the loop body. Figure 3.10 shows a high-level source code example of naive unrolling. Without the induction variable information, the compiler must insert the loop test condition before each unrolled iteration to preserve correctness. Although the loop branching overhead has not been reduced, the transformed code has a longer extended basic block that may be amenable to other transformations.

The high-level source code example in Figure 3.10 is included only as an illustration of the semantics of naive loop unrolling. Because this thesis focuses on unstructured loops in low-level intermediate representation, the rest of this section uses the CFG in Figure 3.11 as a running example. The loop depicted in the CFG is a single loop nest with a clear entry and exit block, indicated by the blocks \textit{START} and \textit{END} respectively. The example is simple for clarity, but the algorithms in this section apply to loops of arbitrary size and control flow. The rest of this section traces the naive unrolling of this example loop by an unroll factor of two. After the CFG analysis to determine the loop body, naive unrolling consists of cloning the loop body and then chaining the cloned iterations together.
while $i \leq 100$ do
    $a(i) = b(i) + c(i)$;
    $i = i + 1$;
    if $i > 100$ then break;
endw

while $i \leq 100$ do
    $a(i) = b(i) + c(i)$;
    $i = i + 1$;
    if $i > 100$ then break;
endw

Figure 3.10: The while loop, shown on the left, is naively unrolled by a factor of four on the right.

Figure 3.11: Original loop
CFG analysis

The CFG-analysis phase is simply the use of an existing algorithm for identifying loop bodies in a CFG. There are many algorithms to choose from in the literature [13, 14, 4, 35, 54, 55, 53, 52, 33, 49]. Any loop-detection algorithm may be used, but Ramalingam suggests that the DJ Graphs Algorithm, by Sreedhar et al., produces the most intuitive nesting relationship between reducible and irreducible loop [49, 52]. Therefore, this thesis uses DJ Graphs as a plug-in component that identifies loop bodies. Section 3.2.1 illustrates some theoretical deficiencies of this type of CFG analysis algorithm, but in practice it is usually sufficient for correctly identifying loop bodies.
Cloning the loop body

Algorithm 1 describes a procedure for cloning the body of a loop, given the loop structure and the unroll factor. Line 1 loops over the unroll factor, creating the correct number of copies to create the unrolled body. The loop in lines 2-4 creates a copy of each block in the loop body and stores the pointer in the clone field. Next, lines 5-14 clone the edges. It is important to note that only edges originating in the loop body are cloned; no incoming edges are cloned. If the edge is completely contained in the loop body (that is, if the source and destination of the edge are both in the loop body), then the edge is created within the new cloned loop body. If the destination of the edge is outside of the loop body, then the new edge is pointed to the original destination. The result of the LoopClone() procedure is to create a set of $F$ clones of the loop body that have no incoming edges, but maintain the correct internal and outgoing edges. Figure 3.12 shows the result of applying LoopClone() with an unroll factor of two on the original loop from Figure 3.11.

Chaining the Cloned Loop Bodies

After cloning the loop body $F$ times, the next step connects the separate clones together to form a "chain". This algorithm, described in Algorithm 2, creates the unrolled loop body by re-pointing the edges in the individual cloned regions. Line 2 stores the head block for the $i^{th}$ cloned region, and line 3 stores the head block for the "next" cloned region, the $(i + 1)^{th}$ region with wrap-around. Then, the edges entering the $i^{th}$ header are redirected to the $(i + 1)^{th}$ header. Since the LoopClone() procedure in Algorithm 1 does not clone incoming edges, all predecessors for the $i^{th}$ header must originate in the $i^{th}$ cloned region. Further, since they point to the header, they must be back edges (i.e., edges that create a cycle by providing a path to the header of a loop). Thus, this algorithm breaks the cycle in each cloned region and redirects control flow to the next cloned region. Finally, the last cloned region wraps around and redirects control to the first cloned region. This last step restores
Algorithm 1: LoopClone(Loop, F): Cloning a loop body

1 \textbf{for} $i = 1 \ldots F$ \textbf{do}
2 \hspace{1em} \textbf{for} $N \in \text{Loop.blocks}$ \textbf{do}
3 \hspace{2em} $N\.clone[i] \leftarrow N\.copy()$;
4 \hspace{1em} \textbf{endfor}
5 \textbf{for} $N \in \text{Loop.blocks}$ \textbf{do}
6 \hspace{1em} \textbf{for} $E \in N\.suc$ \textbf{do}
7 \hspace{2em} $D \leftarrow E\.dest$;
8 \hspace{2em} \textbf{if} $D \in \text{Loop.blocks}$ \textbf{then}
9 \hspace{3em} addEdge($N\.clone[i], D\.clone[i]$);
10 \hspace{2em} \textbf{else}
11 \hspace{3em} addEdge($N\.clone[i], D$);
12 \hspace{2em} \textbf{endif}
13 \hspace{1em} \textbf{endfor}
14 \hspace{1em} \textbf{endfor}
15 \textbf{endfor}
the loop cycle, but now it contains $F$ iterations instead of one. Figure 3.13 shows
the result of applying $\text{LoopChain()}$ with an unroll factor of two on the cloned loop
from Figure 3.12. The resulting loop shows the structure of the unrolled loop body,
although it still has no incoming edges.

<table>
<thead>
<tr>
<th>Algorithm 2: LoopChain(Loop, $F$): Chaining Cloned Loop Bodies</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 for $i = 1 \ldots F$ do</td>
</tr>
<tr>
<td>2 $N_1 \leftarrow \text{(Loop.header).clone}[i]$;</td>
</tr>
<tr>
<td>3 $N_2 \leftarrow \text{(Loop.header).clone}[(i \mod F) + 1]$;</td>
</tr>
<tr>
<td>4 for $E \in N_1,\text{pred}$ do</td>
</tr>
<tr>
<td>5 $\text{repointEdge}(E, N_2)$;</td>
</tr>
<tr>
<td>6 endfor</td>
</tr>
<tr>
<td>7 endfor</td>
</tr>
</tbody>
</table>

Completing the Naive Unrolling

The last step for naive unrolling is simply to re-point the loop entry edge from the
original loop header to the unrolled loop header. Figure 3.14 illustrates the final
result of naive unrolling. It is important to note that the resulting code will be
correct in the general case. This transformation is safe because the loop exit test
and branches are preserved between iterations in the unrolled body. In this example,
the original loop is now unreachable code, and has been grayed out in the figure.
The original loop could have been used as one of the iterations in the unrolled body
instead of creating an extra clone. It is presented this way, however, because the
general unrolling algorithm, presented in Section 3.3.2, uses the original loop as the
cleanup loop.
3.3.2 SSA Analysis and Clean Unrolling

The naive unrolling algorithm of the last section results in a loop body containing larger extended basic blocks, potentially increasing optimization opportunity. However, the loop overhead has not been decreased, and none of the individual blocks have increased in size. This section shows how to eliminate the loop-exit branches within the body of the unrolled loop, allowing fusion of the unrolled iterations. This type of unrolling will be referred to as clean unrolling, or general unrolling, because it supports induction-variable based loops in the general case. A cleanup loop is utilized to handle cases for which the dynamic iteration count is either unknown at compile time or is not a multiple of the unroll factor.

Minimum Remaining Iterations

The intuition behind clean unrolling follows from understanding the minimum remaining iteration (MRI) property. Every loop has at least one exit condition that controls loop termination. The loop iterates while the exit condition evaluates to false, and program control transfers to the next statement after the loop body when the condition evaluates to true. From the opposite perspective, every loop also has a continuation condition; while this condition evaluates to true, the loop continues to iterate. The truth of this continuation condition, at the head of a loop (the first block in the loop body), must always evaluate to true (otherwise control would not have been transferred to the loop header). As the loop body finishes an iteration, it then checks the condition again. If it remains true, then control transfers to the head of the loop for another iteration. This condition must be dependent on some operation in the loop body; otherwise, the loop would never terminate once started. Therefore, because the continuation condition is checked at the end of every iteration, it ensures that at least one more iteration is remaining before it begins another iteration by transferring control to the head of the loop. Thus, the continuation condition ensures that the minimum remaining iterations for the loop is at least one. This ensures cor-
rect code, as the loop body is only executed when at least one more iteration remains in the loop. This results in the loop executing each iteration in a simple "one at a time" manner. The program decides whether or not to execute the next iteration immediately before it will be executed. An example of this concept is depicted in Figure 3.15; the continuation condition is essentially checking to see whether at least one more iteration remains before it transfers control back to the header of the loop. Otherwise, the loop is terminated and the program control continues after the loop body.

The MRI property is simply a conservative minimum bound, guaranteeing that there is at least a certain number of iterations remaining. The bound is not necessarily a tight bound, however, and there may be more than the given number of iterations remaining. Since most loops usually execute more than one or two iterations, it is quite common for the actual minimum remaining iterations to be much larger than one. For example, consider the loop in Figure 3.2 with a known iteration count of 100. In this case, before the first iteration of the loop, the precise MRI is 100. As the loop progresses, the actual MRI decreases until the MRI is zero, causing the loop to
terminate. The important point is that for the majority of the iterations, the actual MRI is greater than one. This is the property that loop unrolling exploits.

**Iteration Lookahead**

Continuing with the example from Figure 3.15, consider what happens when naive unrolling is applied to this loop. Figure 3.16 shows the result. Again, control transfers to the next iteration only when the “1 more?” condition evaluates to true. What if we could somehow look into the future, and determine that two more iterations were remaining? If we had a “2 more?” conditional expression, then we could use that in place of the conditional check after block A2. Then, whenever control transferred to the head of the unrolled body, we would know that at least two more iterations remained. Thus, the conditional after block A1 is no longer necessary. Transforming the code accordingly leads to the CFG in Figure 3.17. This technique will be referred to as *iteration lookahead*, because it allows several iterations to be executed as a group. Instead of using the original continuation condition expression that represents an MRI of one, a new lookahead continuation conditional expression is used that represents an MRI of greater than one. In more formal terms, the iteration lookahead is trying to determine whether a tighter minimum bound exists. If so, the tighter bound can be used to optimize the unrolled loop body by removing the incremental continuation conditionals.

The final step in clean unrolling is simply to connect the unrolled loop body and the original loop body with the correct continuation conditional expressions, as shown in Figure 3.18. The lookahead continuation conditional is used after the START block, checking whether enough iterations remain to even enter the unrolled body. If not, the original loop, or *cleanup loop* is executed. The cleanup loop, an exact clone of the original loop, uses the original continuation conditional, while the unrolled loop uses the lookahead continuation conditional. As a final step after the lookahead continuation conditional terminates the unrolled loop, the code must then
Figure 3.16: Effect of naive unrolling on MRI

Figure 3.17: Iteration lookahead with MRI=2
check the original continuation conditional again. The lookahead conditional may fail for an MRI greater than one, for example, but a single iteration may still remain. This corresponds to a number of remaining iterations that is less than the lookahead factor, or unroll factor. If at least one iteration remains, the cleanup loop must be entered.

**Deriving Lookahead Expressions**

The previous paragraphs have shown how to use the continuation conditional lookahead expression to complete the clean-unrolling transformation, but have treated the lookahead as though it is knowledge from an oracle. Fortunately, many loops exhibit predictable behavior for the continuation conditional, allowing the compile-time derivation of a lookahead conditional expression. The most common case of a continuation conditional is a comparison between a linear induction variable and a loop-invariant “upper bound”; the lookahead expression simply compares the upper bound with the induction variable incremented to appear as though it were several iterations in the future. Essentially, the lookahead expression is pre-computing the value of the induction variable for one or more iterations in the future. The more common approach results in a new upper bound that is calculated by decrementing the original upper bound by the induction variable increment multiplied by the unroll factor.

This technique only requires that the continuation conditional have a predictable pattern. Therefore, continuation expressions with non-integer, or even non-linear induction variables, can be transformed into lookahead expressions. The techniques proposed by Wolfe to recognize non-linear induction variables could be applied here [63].

**Detecting Induction Variables and Region Constants**

Detecting the induction variables (IV) and region constants (RC) for each loop requires analysis on the SSA graph of the program. This technique was first introduced
Figure 3.18: Clean unroll, by two
by Wolfe in 1992 [63], and was later used for operator strength reduction by Cooper
et al. in 2001 [20]. Using Tarjan's strongly connected component algorithm on the
SSA graph reveals the linear induction variables for each loop [54]. Each induction
variable is identified as an SCC in the SSA graph that contains a single phi-node at
the loop header and only loop-invariant linear increment operations (such as addition
or subtraction). Then, region constants can be identified as SSA definitions that are
not induction variables and are defined in a block that strictly dominates the loop
header. The dominance relation ensures that the value is loop-invariant, because it
must be defined before control enters the loop body.

Identifying Loop Exit Branches

Loop exit branches can be identified by examining all edges that leave the loop body.
Commonly, there is a single exit branch that corresponds to the continuation condi-
tional branch that determines whether or not another iteration of the loop should be
executed. In some cases, there are multiple exit branches, which correspond to loops
with break conditions. Therefore, loop unrolling requires that the correct loop exit
branch be identified.

Conditional branches in ILOC can be constructed in one of three styles, depending
on how the comparison operation and condition code are evaluated. This requires a
chain of up to three operations; one handles the comparison, another evaluates the
condition code, and the final actually executes the branch. Additionally, the oper-
ations in this chain are not necessarily in close proximity to one another; they may
be scattered throughout different blocks in the loop. Using the use-def chains that
are created during SSA construction provides an easy mechanism for tracing through
the conditional branch operation chain. Use-def chains simply link an SSA-use to the
single SSA-definition that provides the value. When the actual operands for the con-
ditional branch are located, they can be compared with the IV and RC information. If
the comparison operation is performing a standard less-than or greater-than compari-
son ($<$, $>$, $\leq$, $\geq$), and one operand is a region constant while the other is an induction variable, then this conditional branch represents the continuation conditional for the loop.

**Calculating New Conditional Expressions**

After the continuation conditional branch has been identified, a new upper bound for the loop can be calculated. Additionally, the unrolled body requires two additional guard expressions before and after the unrolled body. Figure 3.19 shows, from a low level, what the final unrolled code will look like for a small example loop. The three test expressions are shown in blocks T0, T1, and T2. These block labels correspond with the labels in the flow graph from Figure 3.18. Conditional T0 can be referred to as the new upper bound for the loop, because it determines if another iteration of the unrolled loop should execute. Conditional T1 is a guard condition, determining whether the MRI holds for the first iteration. If not, the cleanup loop is executed immediately. Finally, conditional T2 is another guard condition, determining whether any iterations remain after the unrolled loop exits. If so, the cleanup loop is executed; otherwise, the execution of the loop is complete.

Figure 3.19 shows the general case where the induction variable is incremented both before and after the loop exit conditional is executed. The entire increment for a single iteration is ($x + y$). Instead of having a back edge directly associated with the exit conditional, there is actually a *back path*. This indicates that additional code will be executed after the continuation conditional branch is executed, but before control branches back to the loop header.

The T0 test is required for ensuring the MRI property for the unrolled body is greater than or equal to the unroll factor. The induction variable, as identified with the SSA-graph analysis, is represented as a chain of operations throughout the loop body. The operations in this chain determine the increment for the induction variable during each iteration of the loop. The operation chain will contain a single phi-node
HEAD:
  i += x
  c = (i < n)
  if (!c) goto EXIT
  i += y
  goto HEAD
EXIT:

T1:  c = (i < (n - 3x - 2y))
     if (c) goto UNROLL
     goto CLEANUP
UNROLL:
  i += 4x
  i += 3y
T0:  c = (i < (n - 3x - 3y))
     T2IV = i
     T2RC = n
     if (!c) goto T2
     i += y
     goto UNROLL
T2:
  c = (T2IV < T2RC)
  if (!c) goto EXIT
  i = i + y
  goto CLEANUP
CLEANUP:
  i += x
  c = (i < n)
  if (!c) goto EXIT
  i += y
  goto CLEANUP
EXIT:

Figure 3.19: The original loop on the left is unrolled by four on the right. This example shows how the T1 and T2 continuation expressions are computed.
at the header of the loop. Because the induction variable has been determined to be linear, this SSA operation chain will only include addition and subtraction between the induction variable and region constants. Region constants may either be a loop-invariant variable or a literal constant. Often times the IV increment will contain only a single increment operation, but sometimes (for example, in the case of previous unrolling) the increment will contain multiple increment operations. The entire chain represents the increment, however, and needs to be considered when calculating the new upper bound.

By "reversing" the induction variable increment and applying it to the previous upper bound, which is a region constant, the new upper bound is derived. The reversed increment must be applied multiple times, determined by the unroll factor. This calculation, in the case of literal constants, can be computed at compile time and placed in the landing pad of the loop. For the typical case involving loop-invariant values, the new upper bound must be computed at run time. Although Figure 3.19 does not show it, this computation can be placed in the landing pad of the loop. Because the landing pad of the loop dominates every other block in the loop, the upper bound can safely be used in the unrolled body of the loop. The new upper bound for the example is $(n - 3x - 3y)$, or more clearly $(n - 3(x + y))$. More generally, the upper bound is calculated as $(n \pm (f - 1)(incr))$ where $f$ is the unroll factor and $incr$ is the total increment per iteration. The $\pm$ should simply be the reverse of the original increment operation, as described above.

An induction variable, by definition, will only have increments of the form $IV = IV \pm RC$ or $IV = RC + IV$ (subtraction is not commutative). The region constants are either the result of a load immediate, or their definition strictly dominates the loop header. If the RC falls into the latter category, then the value will be available at the end of the loop landing pad. This is guaranteed because the definition of the RC strictly dominates the loop header, and the landing pad is the immediate dominator of the loop header. Thus, the original RC name may be used in the
reversed induction variable computation. However, if the RC is the result of a load immediate, then this definition does not necessarily dominate the landing pad. To handle load immediate operations safely, they must be duplicated in the landing pad. The duplicate operation should store the result to a new name to preserve the SSA property of a single definition for each name.

Block T1 contains the guard expression determining whether there are enough iterations to enter the unrolled loop. That is, T1 ensures that the MRI is correct for the first iteration of the unrolled loop. This conditional branch replaces the unconditional jump at the end of the original landing pad. The same process for calculating T1 can be used as for T0, except we need to take extra care to handle the order of the increments and conditional expressions in the loop body. The upper bound for T1 used in the example is \((n - 3x - 2y)\). This differs from the bound used in T0 because this calculation resides outside of the loop (before the induction variable has been incremented). Therefore, we need to account for a partial iteration. The general formula for the upper bound for T1 is \((n \pm (f - 1)x \pm (f - 2)y)\), where \(f\) is the unroll factor, \(x\) is the increment before the conditional expression and \(y\) is the increment after the conditional. Again, the \(\pm\) is determined by reversing the original operation in the increment.

Finally, block T2 contains the expression determining whether the cleanup loop is needed after the unrolled body exits. Fortunately, this conditional expression is already provided, since it is the original upper bound of the loop. Leaving the conditional expression in the unrolled loop likely creates partially dead code [42]. This could be a performance inefficiency, however, since the result is only used after the unrolled loop terminates. On the other hand, copying the computation to the T2 block is not safe, because the arguments for the conditional expression could be modified before reaching T2. Thus, the best solution is to copy the operands for the conditional expression at their original location, but place the comparison operation and branch in the T2 block. This is captured by storing \(i\) and \(n\) in the temporary
names $T2IV$ and $T2RC$. Then, the comparison and branch is placed in block T2.

The final concern is code on the back path. As shown in the example, the $1 \leftarrow y$ statement is executed on the back path. A more general example is shown in Figure 3.20. The unrolling algorithm has two options to preserve the correct behavior when transferring control from the unrolled body to the cleanup loop. First, control may jump directly to the back path in the cleanup loop, instead of jumping to the header. This case, shown in Figure 3.21, is easier to implement, but has the side effect of turning the cleanup loop into an irreducible loop. The second option is to clone the entire back path code, executing that before jumping to the cleanup loop header. The cloning method, shown in Figure 3.22, is also depicted in the example in Figure 3.19. The experiments for this thesis use the direct-jump method, however.

Up until this point, the code has been in SSA form. Calculating and storing the new upper bound with new names preserves the SSA namespace. The analysis requires SSA form, and the upper bound calculations are easily inserted while the code is in SSA form. The actual block cloning and chaining transformation would require significant effort to preserve the SSA namespace, however. Each cloned block would need new phi nodes, and all definitions and uses would need to be renamed to maintain correct SSA form. To simplify the code transformation, therefore, the analysis is expected to be completed before any blocks are cloned. Once the analysis is complete, and the upper bound is calculated and inserted, the code can be restored to the original namespace. Then, the blocks can be cloned without concern for the properties of the namespace.\footnote{In an intermediate representation that enforces the SSA namespace at all times, the programmer has no choice but to preserve SSA during the transformation phases of loop unrolling.}

**Completing the Process**

The last section described the process for calculating the new upper bounds and guard conditionals for the unrolled loop and the cleanup loop. This section provides the
Figure 3.20: Loop with code on the back path

Figure 3.21: Using a direct jump to handle code on the back path

Figure 3.22: Using cloning to handle code on the back path
remaining steps in completing the unrolling, as well as an overview of the structure of the unrolled body.

A cloned and chained unrolled body, as produced by the naive algorithm, can be transformed into a clean unrolled body by using the upper bound information calculated in the previous section. First, the continuation conditionals for each cloned iteration in the unrolled body are removed, except for the last. The arguments for the last continuation conditional are saved in temporary names for the T2 expression. The last continuation conditional expression is then converted to use the new T0 upper bound, which was computed previously and stored in a new register name.

Next, the T1 conditional expression is placed in the landing pad of the original loop. The true direction of the branch directs control to the unrolled body, while the false direction of the branch directs control to the original loop. Thus, the original loop becomes the cleanup loop.

The T2 block is then created. The T2 conditional expression is constructed, based on the temporary names that were saved from the original continuation conditional. Code on the back path is either cloned, or the true direction of the T2 branch is pointed directly to the back path in the cleanup loop. Finally, the exit branch in the unrolled body is pointed to the T2 block.

This concludes the unrolling algorithm. The code is now unrolled safely and cleanly, handling arbitrary upper bounds and induction variable increments. It is important to note, however, that the unrolling algorithm presented here does not apply any optimizations on the new unrolled body. To exploit the new and potentially larger program context for optimization, other transformations must be applied. Dead code elimination should be applied to remove the remaining calculations from the original continuation conditionals [38]. A control flow cleanup pass should be applied to remove any unnecessary branch operations [16]. Value numbering, algebraic reassociation, operator strength reduction, loop-invariant code hoisting, and peephole optimization may also be profitable, depending on the loop body [10, 9, 20, 41, 47, 24].
In general, however, loop unrolling generates a new region of code that is potentially amenable to any optimization pass sequence. Adaptive compilation exploits this property, and the results are presented in the next chapter.

3.4 Final Considerations

This section concludes the chapter with a brief examination of several additional considerations for loop unrolling. First, some architectures provide a loop operation in the instruction set, which provides hardware support for loops. This scenario introduces subtle implications to the proposed loop-unrolling algorithm. Next, the loop-unrolling algorithm is not always able to successfully unroll a loop. This section concludes by identifying and examining cases which cause the algorithm to fail.

3.4.1 Hardware Looping Construct

Some architectures provide a hardware looping construct that enables efficient execution of loop bodies in hardware. This is accomplished through an additional hardware instruction that specifies the iteration count and the address of the last operation in the loop body. The loop induction variable and backwards branches are all handled by the hardware, resulting in very efficient loop execution. There are subtle implications to this thesis if such an operation is considered.

First, if a compiler writer is implementing a compiler backend for a target architecture that includes a hardware looping operation, then the methods in this thesis can be used to translate general GOTO-loops into the correct output format. Utilizing the looping operation requires the same analysis that is needed to identify and unroll loops. This approach allows an existing compiler to be extended to efficiently support a new target architecture.

Next, if the compiler's intermediate representation includes a loop operation, then some of the analysis will not be needed to perform loop unrolling. The looping operation explicitly identifies the beginning and end of the loop body, as well as
the total iteration count of the loop. Thus, SSA-graph analysis is no longer needed. However, control flow analysis is still required to identify the entire loop body. Instead of computing a lookahead continuation conditional, the unrolling algorithm only needs to determine if the iteration count (either literally or symbolically provided by the looping operation) is a multiple of the unroll factor. This determines whether a cleanup loop is needed. The unrolled loop body is then modified to iterate a reduced number of times.

Finally, the advantages of loop unrolling may be diminished in the presence of a hardware looping operation. Much of the loop overhead has been reduced or eliminated through efficient hardware execution, so unrolling will not likely reduce loop overhead further. However, loop unrolling still provides an increased optimization context. Other optimizations should have more opportunity to improve the code because of the larger loop body. Thus, unrolling still has the potential to improve performance in the presence of a hardware looping operation.

3.4.2 Loop-Analysis Shortcomings

The loop-unrolling algorithm in this chapter was designed specifically to handle the complexities encountered with unstructured loops. However, the analysis is still limited, missing some opportunities for loop unrolling. This section presents these shortcomings, showing why the algorithm cannot handle them.

This algorithm ignores irreducible loops, performing loop unrolling on reducible loops only. Section 3.2.1 introduces irreducible loops, and describes in detail the difficulties of unrolling irreducible loops. Node splitting could be used to handle irreducible loops by transforming each occurrence into a collection of reducible loops [34]. Because irreducible loops arise infrequently in practice, it was not worth the engineering effort to support node splitting.

Ignoring irreducible loops, the control flow analysis generally does an excellent job identifying loop bodies. Sections 3.2.1 and 3.2.1 present several small examples
where the control flow analysis falls short, but the main limitations of loop unrolling arise in the induction variable analysis. If the induction variables or loop-invariant values cannot be identified, then the loop cannot be cleanly unrolled (although, naive unrolling is still applicable).

The main problem stems from the loop-invariant, or region constant (RC) values being hidden. The algorithm identifies an RC as a value whose definition strictly dominates the header of the loop, or the result of a load immediate operation. For example, an expression with constant valued operands is not recognized as loop-invariant, even though it is. Also, any loop-invariant expression that is defined within the body of the loop will not be correctly identified, because the definition does not strictly dominate the loop header. Thus, constant propagation and loop-invariant code motion are passes that can increase the effectiveness of the RC identification. Cooper et al. make this observation for their operator strength reduction algorithm [20]. Fortunately, the adaptive compiler is capable of finding the correct enabling transformations for loop unrolling.

Next, induction variables (IVs) are sometimes missed by the analysis. A linear IV is defined as a cycle in the SSA graph that contains a single phi-node at the header (merging the initial value of the IV with the value computed on the previous iteration), and addition and subtraction operations between the IV and RCs. If the RCs have not been properly identified, then the IV may not be properly identified. To the analysis, it will look like the increment for this cyclic variable is potentially different on each iteration of the loop.

Induction variables may also be missed if they are updated along different paths in the loop body. This would create either multiple back edges going into the phi node of the loop, or another phi node in the body of the loop. Both of these cases are not captured by the definition of an IV. If the increment for the IV was the same on every path in the loop, it would behave like an IV. However, extra analysis is required to catch this. Wolfe identifies this case in his work on induction variable detection.
for dependence analysis [63].

Finally, even if the IVs and RCs have been successfully identified, the loop exit test must conform to certain properties for the unroller to be successful. First, there must be a conditional branch such that one path exits the loop and one remains in the loop. Second, the direction of the branch should be determined by a standard comparison operator ($<$, $>$, $\leq$, $\geq$) between an IV and an RC. The algorithm uses SSA use-def chains to identify the conditional expression and their operands, allowing them to be scattered throughout the loop body. If the conditional expression uses logical operators, such as AND or OR, then the loop exit branch will not be identified as a candidate for loop unrolling. A simple pre-pass for converting these logical expressions into a series conditional branches would allow the algorithm to identify the correct exit branch (if one of the correct form exists) for loop unrolling.
Chapter 4

Experimental Results

As described in Chapter 3, the main motivation for this thesis is addressing a deficiency in the set of optimizations in the Rice University research adaptive compiler. Namely, the previous compiler did not have a transformation that could effectively reduce loop overhead. This chapter presents experimental results that show the positive impact of adding a loop unroller to the adaptive compiler.

After Section 4.1 provides a brief overview of the benchmarks used in the experiments and Section 4.2 describes the search algorithm, Section 4.3 shows the effects of using loop unrolling in a fixed compilation-sequence. Although the main focus of the thesis is on an adaptive compiler, the fixed-sequence results provide evidence that loop unrolling at least has the potential to improve performance for the selected benchmarks. The results are inconsistent, however, also showing that loop unrolling can hurt performance significantly. Therefore, these fixed-sequence results suggest that loop unrolling is well suited for use in an adaptive compiler, because the adaptive compiler only selects transformations when they are found to be profitable.

Next, Section 4.4 presents the results of adding the loop unroller to the adaptive compiler. The effects are measured according to two metrics, effectiveness and efficiency. First, effectiveness measures the quality of the compilation sequence found by the adaptive compilation search, regardless of the amount of time required for the search. This section compares the effectiveness of the same search algorithm, with the same search parameters, on differing search spaces. Results show that eight of the nine benchmarks exhibit an improvement in effectiveness when the loop unroller is added. Next, efficiency measures the amount of search time (in search iterations)
required to find a solution of a given quality. Therefore, it is most intuitive to view this metric as an "efficiency curve," plotting solution quality against search time. A quickly descending curve indicates that the search is very efficient, requiring very few search iterations to find a high quality solution. Despite the increase in the size of the search space when loop unrolling is added, seven of the nine benchmarks show an increase in search efficiency. These benchmarks show either an immediate improvement in efficiency, or an improvement after a small number of search iterations. Only two benchmarks show a degradation in search efficiency.

Finally, to address the original motivation for this thesis, Section 4.4.3 presents results measuring the frequency of each transformation in the top sequences. Without the loop unroller, the adaptive compiler selects loop peel nearly 26% of the time. With the loop unroller, however, the loop peel frequency drops to 15%, while the loop unroll frequency is 11%. These results indicate that loop unrolling, as predicted, reduces the need for multiple applications loop peel.

4.1 Benchmarks

All of the experimental results for this thesis measure performance on the benchmarks described in this section. Table 4.1 presents an overview of the nine benchmarks, written in both C and FORTRAN77\(^1\). Because this thesis targets scientific computing and embedded systems, the set of benchmarks comprises various numerical and signal processing programs. These classes of programs are performance critical and often run for very long periods of time, justifying the extra compile time required for adaptive compilation.

All of the benchmarks operate on floating-point data sets, usually performing regular computation over large amounts of data. Consequently, nested loops are very common in these benchmarks; Table 4.2 displays the loop statistics for the inner loops

\(^1\)The ILOC compiler only supports front ends for C and FORTRAN77, eliminating FORTRAN90 programs from experimentation
in each benchmark. Although this set of benchmarks is biased towards programs that rely heavily on iteration-based loops, Section 4.4 shows that this does not guarantee that loop unrolling will always improve performance.

Each benchmark is executed with a default training dataset for input. This dataset represents a small input size that still accurately captures the characteristics of the program execution. Grosul showed that for benchmarks of this type, changing the input dataset has little impact on the effectiveness of the search algorithm for the Rice University adaptive compiler [30]. That is, a good compilation sequence, discovered by running the adaptive compiler on a given program and training dataset, is also a good sequence for the same program with a different dataset. Using the smallest data set that supports this property helps reduce the time spent in the execution phase of the adaptive compiler. However, the success of this approach relies on the behavior of the benchmarks, as well as the type of optimization passes that are included in the compiler. For other compilers and input programs, it may be important to identify one or more datasets that are characteristic of typical program input. Recent work by Fursin et al. examines in more detail the impact of datasets on the effectiveness of adaptive compilation [28]. They show that multiple datasets can be used for adaptive compilation, when a single representative dataset does not exist, to discover a compilation sequence that is effective across many different inputs.

This thesis excludes benchmarks that do not contain loops and benchmarks for which loop unrolling cannot identify and unroll the loops. For example, many programs with linked data structures include loops that do not iterate over an induction variable. Although naive unrolling may be applied in this situation, it is left as future work. For these types of benchmarks, loop unrolling does not affect program performance, because it cannot transform the code. The quality of solutions in the search space with loop unrolling will be no better than without loop unrolling.

Because the search time is limited, however, it is still important to consider benchmarks in which loop unrolling does not increase program performance. For these
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Language</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm.coder</td>
<td>C</td>
<td>audio encoder that converts from 16-bit PCM to 4-bit ADPCM</td>
</tr>
<tr>
<td>adpcm.decoder</td>
<td>C</td>
<td>audio decoder that converts from 4-bit ADPCM to 16-bit PCM</td>
</tr>
<tr>
<td>applu</td>
<td>FORTRAN77</td>
<td>fluid dynamics simulation</td>
</tr>
<tr>
<td>matrix300</td>
<td>FORTRAN77</td>
<td>matrix multiplication for varying matrices</td>
</tr>
<tr>
<td>rkf45</td>
<td>FORTRAN77</td>
<td>Runge-Kutta-Fehlberg solver for ordinary differential equations</td>
</tr>
<tr>
<td>seval</td>
<td>FORTRAN77</td>
<td>cubic spline function evaluator</td>
</tr>
<tr>
<td>solve</td>
<td>FORTRAN77</td>
<td>solver for systems of linear equations</td>
</tr>
<tr>
<td>svd</td>
<td>FORTRAN77</td>
<td>computes the singular value decomposition of a rectangular matrix</td>
</tr>
<tr>
<td>tomcatv</td>
<td>FORTRAN77</td>
<td>vectorized mesh generator</td>
</tr>
</tbody>
</table>

Table 4.1: The nine floating point benchmarks used for all experiments

benchmarks, the search time spent evaluating compilation sequences that contain loop unroll is “wasted,” because it cannot improve the final search result. Thus, search efficiency can be negatively affected by adding a transformation. A decrease in search efficiency means that, given a bounded amount of time, the search may not find as good of a solution. Thus, both search efficiency and search effectiveness can be hurt by adding loop unrolling. To measure this effect, two benchmarks that respond poorly to loop unrolling are included in the experiments. Both solve and svd exhibit performance degradation when loop unrolling by a factor of four is applied. These benchmarks provide efficiency results that would be similar to a benchmark for which loop unrolling has no effect. Despite the potentially harmful effect of including loop unrolling in the search for these benchmarks, Section 4.4 shows that the search effectiveness and efficiency do not significantly degrade.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>File</th>
<th>Inner Loops</th>
<th>Average Operations</th>
<th>Average Blocks</th>
<th>Average Nesting Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm_coder</td>
<td>adpcm_coder</td>
<td>1</td>
<td>210.0</td>
<td>30.0</td>
<td>0.0</td>
</tr>
<tr>
<td>adpcm_decoder</td>
<td>adpcm_decoder</td>
<td>1</td>
<td>172.0</td>
<td>25.0</td>
<td>0.0</td>
</tr>
<tr>
<td>applu</td>
<td>blts</td>
<td>4</td>
<td>111.3</td>
<td>2.0</td>
<td>4.3</td>
</tr>
<tr>
<td></td>
<td>buts</td>
<td>5</td>
<td>87.6</td>
<td>2.0</td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td>jactd</td>
<td>1</td>
<td>11425.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>jacu</td>
<td>1</td>
<td>8901.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>rhs</td>
<td>22</td>
<td>422.1</td>
<td>2.0</td>
<td>2.3</td>
</tr>
<tr>
<td>matrix300</td>
<td>saxpy</td>
<td>1</td>
<td>57.0</td>
<td>2.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>sgemm</td>
<td>8</td>
<td>49.0</td>
<td>2.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>sgemv</td>
<td>3</td>
<td>53.3</td>
<td>2.0</td>
<td>0.0</td>
</tr>
<tr>
<td>rlf45</td>
<td>fehl</td>
<td>6</td>
<td>57.5</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>rkfs</td>
<td>5</td>
<td>47.8</td>
<td>2.6</td>
<td>0.6</td>
</tr>
<tr>
<td>seval</td>
<td>spline</td>
<td>4</td>
<td>97.3</td>
<td>2.0</td>
<td>0.0</td>
</tr>
<tr>
<td>solve</td>
<td>decomp</td>
<td>8</td>
<td>33.6</td>
<td>2.1</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>solve</td>
<td>2</td>
<td>40.0</td>
<td>2.0</td>
<td>1.0</td>
</tr>
<tr>
<td>svd</td>
<td>svd</td>
<td>26</td>
<td>43.5</td>
<td>1.5</td>
<td>1.6</td>
</tr>
<tr>
<td>tomatv</td>
<td>tomatv</td>
<td>12</td>
<td>134.8</td>
<td>2.4</td>
<td>1.2</td>
</tr>
</tbody>
</table>

Table 4.2: Inner loop statistics for each benchmark; includes loop count, average static operation count, average static block count, and average loop nesting depth
4.2 The Hill Climber

The search algorithm that is used for the adaptive compilation experiments is the hill climber. Because the framework for this thesis builds upon Grosul's work, in Cooper's research group [30], the detailed justification for selecting the hill climber over other search algorithms is not explored in this thesis. Briefly, the hill climber was found to be the most cost-effective search algorithm; it usually found good solutions with relatively little effort.

The hill climber works by selecting a random compilation sequence in the search space. It first evaluates this sequence, where evaluation is determined by the objective function. Often, the objective function is execution time, so the evaluation phase requires that the program be compiled and executed. The objective function for this thesis measures dynamic instruction count for the abstract ILOC instruction set. Since the goal is to minimize instruction count, the search is actually descending valleys instead of climbing hills.

Next, the hill climber begins evaluating "neighbor" sequences. A neighbor sequence is simply any sequence of hamming-1 distance; that is, any sequence differing in one position is a neighbor. The hill climber evaluates neighbors, looking for a better sequence. If one is found, the hill climber "steps" to this new sequence and repeats the process. Each step represents the progress of finding a slightly better solution.

At some point, the hill climber discovers that no neighbors exhibit better results. This represents a local minimum. The sequence is not guaranteed to be a global minimum, nor is it guaranteed to be a good sequence. Thus, the hill climber begins the process again with a new random sequence as the starting point. Previous work shows that, given enough random restarts, the hill climber is likely to find a solution that is within a small delta of the optimal [6].

The hill climber algorithm has several parameters that control the search behavior. First, the number or randomized restarts indicates how many times the hill
climber will restart after finding a local minimum; this thesis uses 20 randomized restarts. Next, the patience value, a percentage value between 0 and 100, controls the behavior of an optimization to the search. Instead of examining all neighbors of a given sequence, an impatient hill climber only examines a subset of the neighbors. This subset is randomly selected, and the first sequence that is better is selected for continuing the search. This thesis uses 20% patience; thus, only 20% of the neighbors will be examined before determining that a sequence is a local minimum. Grosul determined that 10% is a good patience value [30]. Initial experiments for this thesis indicated that 20% is more effective for the search space with loop unrolling, however. Perhaps this suggests that the larger search space requires higher patience for larger search spaces.

The search space size and structure are determined by the number of optimizations in the compiler and the length of the compilation sequence. The size of the space is $|A|^N$, where $A$ is the set of optimization passes and $N$ is the length of the sequence. The number of neighbors for each sequence, also determined by $A$ and $N$, is $N \ast (|A| - 1)$. There two parameters are of most interest in this thesis, because they will change as the set of optimizations increases. By adding the loop unrolling, the size of the search space will increase, as will the number of neighbors for each sequence. The Rice University compiler originally had 16 transformations; adding the loop unroller makes 17. The sequence length is held constant at 10. Fixing the sequence length and adding a transformation increases the search space at the growth rate of a tenth-order polynomial, since the size would be $|A|^{10}$. Fixing the set of optimizations and increasing the sequence length grow the search space size at the rate of an exponential function, since the size would be $16^N$.

Even though the search space growth rate is polynomial with regard to the number of optimization passes, the practical implications are significant because the degree of the polynomial is high. For example, increasing the set of optimizations from 16 to 17 causes the search space to grow from $16^{10} = 1,099,511,627,776$ to
$17^{10} = 2,015,993,900,449$, a 1.8x increase. Adding a single transformation to the Rice University adaptive compiler results in nearly doubling the size of the search space. Thus, the compiler writer should be confident that the new transformation will improve the compiler enough to offset the growth in the search space.

### 4.3 Fixed-Sequence Loop Unrolling

Figure 4.1 presents the impact of adding a loop unroller to a fixed-sequence compiler. For each benchmark, the leftmost bar shows the dynamic instruction count for code compiled with the standard fixed-sequence. The standard sequence, as defined by Almagor et al., represents a typical sequence of optimization passes that would be applied in a fixed-sequence production compiler [6]. The other measurements are normalized to the performance of the standard sequence. The right bar for each benchmark indicates the performance achieved by applying unrolling with a factor of four, followed by the standard sequence. Unrolling improves performance up to 8% on matrix300 and 2% to 6% on seval and tomcatv. Unfortunately, it degrades performance up to 24% on applu, rkf45, solve, and svd, mainly due to small iteration counts in many inner loops. Both adpcm.coder and adpcm.decoder show no change due to a subtle naming issue that prevents unrolling from finding induction variables. Manually coalescing copies before unrolling would fix the problem. This isn’t necessary in an adaptive compiler, however, because the next section shows that the search can automatically find the correct enabling transformations.

Unfortunately, the average impact of loop unrolling is a 5.5% degradation in performance. There are two reasons for this. First, instead of selecting an unroll factor based on the loop body, the loop unroller applies a fixed unroll factor to each loop. The unroll factor was four for these experiments. Qasem et al. show that performance can be improved by choosing different unroll factors for each loop [48]. Second, the fixed-sequence with loop unrolling was not tuned to effectively support loop unrolling. That is, loop unrolling was not always enabled or exploited. adpcm.coder
Figure 4.1: Impact, on dynamic ILOC instruction count, of loop unrolling in fixed-sequence

and adpcm_decoder are examples where unrolling was not properly enabled. Other benchmarks indicate that the opportunities created by loop unrolling were not always completely taken advantage of. For example, applu and rkf45 both show performance degradation in the fixed-sequence result, but Section 4.4 shows that loop unrolling can improve performance on these benchmarks, given the appropriate sequence.

Sufficient tuning of the compilation sequence, combined with an effective unroll-factor heuristic, would likely result in improvement of the fixed-sequence results presented in this section. Since the primary focus of this thesis is not to build an effective loop unroller for a fixed-sequence compiler, however, the given results are sufficient. Instead, the next section shows that the adaptive compiler is able to determine a more effective compilation sequence for each benchmark without programmer intervention.
4.4 Adaptive Compilation with Loop Unrolling

The previous section showed that loop unrolling, in a fixed-sequence, produces a wide range of positive and negative changes in performance. The average performance change was negative, however, and would have required significant tuning to improve the results. This section shows the positive impact of adding loop unrolling to the adaptive compiler. The adaptive compiler is able to automatically determine effective compilation sequences that contain loop unrolling. On average, the adaptive compiler produces code that is about 10% faster with the loop unroller than without. This shows that adaptive compilation is able to leverage the positive benefits of unpredictable transformations without suffering from the negative effects.

Further, this section introduces the important balance between search space content and search space size. The search space content determines the quality of a solution that can be found, while the search space size determines the amount of time the search requires to find a good solution. As shown in Section 4.2, adding a new transformation to the adaptive compiler increases size of the search space very quickly. Increasing the number of transformations from 16 to 17 results in a search space that is 1.8 times larger. For the search efficiency not to suffer, the solutions in the new space must improve upon those in the old space. Results presented in this section indicate that adding the loop unroller improves the search-space content for the benchmarks examined. Thus, efficiency plots for seven of the nine benchmarks show improvement when loop unrolling is available in the search.

Finally, measurements showing the frequency of each transformation in top sequences support the original hypothesis that loop unrolling is more effective at reducing loop overhead than loop peel. When the loop unroller was added to the adaptive compiler, the frequency of loop peel dropped from 26% to 15%. Loop unrolling was selected 11% of the time. Slight changes in the frequency of the other transformations can also be seen, indicating that loop unroll and loop peel require different transformations to exploit the opportunities that they create.
Figure 4.2: Impact on dynamic ILOC instruction count, of loop unrolling in fixed-sequence and hill climber

4.4.1 Search Effectiveness

Figure 4.2 plots the effects of adding the loop unroller to the adaptive compiler. Loop unrolling is added as a new optimization pass that unrolls all inner loops by a factor of four; multiple applications of unroll are used to achieve higher unroll factors. The graph measures dynamic instruction count, and each bar is normalized to the standard fixed-sequence. The two leftmost bars, labeled by Standard and Standard+U, indicate the fixed-sequence without and with loop unrolling. The fixed-sequence results are included for comparison. The two rightmost bars, labeled by HC and HC+U, indicate the original hill climber and the hill climber with loop unrolling. Measurements are grouped by benchmark, with the average of all the measurements plotted in the rightmost group.

There are several interesting things to note about Figure 4.2. First, the original hill climber improves the code by an average of 7% over the standard fixed-sequence.
When loop unrolling is added, the average improvement jumps to nearly 17%. This shows that, across the benchmarks tested, loop unrolling improves the effectiveness of the adaptive compiler.

Second, the hill climber with unrolling outperformed the hill climber without unrolling for all but one benchmark, solve. One possible reason for this result is that the larger search space usually contains solutions of higher quality than the original search space. For solve, it turns out that the unroll factor of four is a poor choice, since many loops in solve have iteration counts of three or less. An alternative explanation of this result is that the original search space may contain solutions of equal or better quality, but the hill climber is not able to find them in the given amount of time. This scenario is closely related to search efficiency, and will be examined in more detail in the next section.

Next, all benchmarks that showed no improvement with loop unrolling in the fixed-sequence now show significant improvement in the hill climber. For example, adpcm_coder and adpcm_decoder both showed no change in performance with the fixed-sequence, because loop unrolling was not able to identify the loop induction variables. The hill climber, however, discovered an enabling transformation that made loop unrolling possible. Results for these two benchmarks are significant, because the hill climber without loop unrolling finds little or no improvement beyond the standard sequence. Two other benchmarks, applu and rxf45, show performance degradation with unrolling in the fixed-sequence, but the hill climber finds better results with unrolling. All of these results show that the adaptive compiler is able to automatically find an effective context for loop unrolling. In other words, the adaptive compiler is able to tune a compilation sequence containing loop unroll such that unrolling is enabled before it is applied, and successfully exploited after.

Two final benchmarks, svd and solve, are interesting because the best sequences discovered by the hill climber do not contain loop unroll. This shows that the adaptive compiler is able to detect cases where loop unrolling is not profitable and still find
effective sequences without using this transformation. The next section discusses search efficiency in more detail. These two benchmarks are of interest, because they show the impact on search speed from adding a transformation that is not profitable.

4.4.2 Search Efficiency

As shown in Section 4.4.1, the adaptive compiler is able to find more effective sequences when the loop unroller is in the transformation pool. The average improvement of roughly 10% suggests that the larger search space contains better solutions than the original search space. Another important evaluation of the adaptive compiler is search efficiency. This section presents the impact on search efficiency of adding the loop unroller to the adaptive compiler. Results indicate that for seven of the nine benchmarks, even though the search space is larger, the hill climber with loop unrolling finds better solutions in less time.

Search effectiveness does not provide a complete measure of the overall performance, because search time is not considered. Each hill climber experiment was run with the same parameters: 20% patience\(^2\) with 20 random restarts. However, the number of search iterations, or sequence evaluations, each experiment performs is dependent on the discovery of local minimum. For example, if the hill climber discovers a local minimum\(^3\) very early in the search, then that descent will be terminated with a very low evaluation cost. The final solution could still be of high quality, however, if the initial random point was of high quality, or the search steps showed very large improvements over the initial point. This scenario exhibits the best search efficiency, because it found a high quality solution with very little effort. On the other

\(^2\)The exact patience was 32 neighbors for both versions of the hill climber, which results in 20% patience when including loop unrolling and 21.3% patience when not including loop unrolling.

\(^3\)The patience factor determines how many neighbors the hill climber evaluates before deciding a solution is a local minimum. Therefore, the hill climber may incorrectly classify a solution as a local minimum. Regardless, the hill climber terminates the search after the patience is exhausted, and restarts with a new random solution.
hand, if the hill climber takes a large number of steps before it terminates at a poor local-minimum, then the evaluation cost will be very high. Further, if each step only showed small improvement, then the search efficiency would be very poor. As seen by these examples, search efficiency is not solely dependent upon the search parameters, although these will definitely have an effect. Rather, when the parameters are held constant, the search efficiency is dependent upon the size and content of the search space. Therefore, search efficiency must be carefully measured when evaluating the search performance of two different search spaces.

Figures 4.3 - 4.11 plot search efficiency curves for each of the nine benchmarks. The horizontal axis plots work, measured as sequence evaluations or search iterations. The vertical axis plots average sequence quality, measured as the dynamic instruction count of the compiled executable. The performance is normalized to the best sequence found for each benchmark. Each curve represents the average performance achieved for a given number of evaluations across 20 independent hill climber descents. The solid line, labeled as $HC$: Standard, indicates the average efficiency for the original hill climber. The dotted line, labeled as $HC$: Unroll, indicates the average efficiency for the hill climber with loop unrolling. In comparing the efficiency curve between the original search space and the loop unrolling search space, there are three categories of relationships:

**Immediate Improvement** On some benchmarks, the addition of loop unrolling allows the hill climber to exhibit improved efficiency for the majority of the duration of the search. Benchmarks in this category are adpcm.coder, adpcm.decoder, and matrix300. These benchmarks exhibit a very strong opportunity for loop unrolling that requires very little tuning.

**Delayed Improvement** Some benchmarks show an improvement in efficiency only after a small number of search evaluations. That is, for a small number of evaluations, the presence of low-quality sequences with loop unrolling prevents
the hill climber from finding better results. However, given a sufficient “learning period,” the hill climber discovers the effective context for loop unrolling. After this threshold, the search achieves better quality sequences than without loop unrolling. Benchmarks in this category are rkf45, seval, svd\textsuperscript{4}, and tomcatv. These benchmarks present opportunities for loop unrolling, but they are also susceptible to performance degradation when the code resulting from loop unrolling is not properly optimized.

**Degradation** Finally, some benchmarks show, for the entire duration of the search, a degradation in search efficiency when loop unrolling is added. In other words, to find a sequence of a given quality, the search with loop unrolling requires more work than the search without loop unrolling. This result suggests that, when applied to these benchmarks, loop unrolling is either ineffective or causes performance loss. In both cases, the hill climber wastes time evaluating sequences that contain loop unrolling. These evaluations contribute to the work without improving solution quality, reducing efficiency. Benchmarks in this category are applu and solve.

The final category is labeled as *Degradation*, because the quality of the solution returned in a practical amount of time is worse when loop unrolling is used. Efficiency measures the amount of time it takes to find a solution of a given quality. Given more time, it is possible that the search could find more effective solutions. Thus, for solutions of very high quality (that have not yet been found), it is possible that the search with loop unrolling will have better efficiency. For instance, consider the extreme example of giving the adaptive compiler enough time to completely enumerate the search space. Clearly, the optimal solution will be discovered. If the larger search space contains solutions of quality higher than the best solution in the smaller

\textsuperscript{4}Based on the efficiency curve, svd belongs in this category; however, the search effectiveness results indicate that svd does not exhibit opportunity for loop unrolling by a factor of four.
Figure 4.3: Search efficiency for adpcm_coder

Figure 4.4: Search efficiency for adpcm_decoder

Figure 4.5: Search efficiency for applu

Figure 4.6: Search efficiency for matrix300

Figure 4.7: Search efficiency for rkh45

Figure 4.8: Search efficiency for seval
Figure 4.9: Search efficiency for solve

Figure 4.10: Search efficiency for svd

Figure 4.11: Search efficiency for tomcatv
space, then the larger search space will exhibit better efficiency for those solutions. The search in the smaller space can never reach those solutions, so the efficiency for reaching a solution of that quality is infinite. If this is the case, then the benchmark technically falls into the category of Delayed Improvement. In effect, however, the search is subject to real time constraints. Thus, in practice, benchmarks in this category exhibit search efficiency degradation when loop unrolling is added to the compiler.

There is a logical correlation between the search efficiency and the search effectiveness for most benchmarks. Benchmarks that fall in the Immediate Improvement or Delayed Improvement efficiency categories also show improved search effectiveness. This is expected, because the search effectiveness is a measure of the best solution discovered in the given search time. Since the efficiency, for the best solutions found, was better with loop unrolling, then the quality should be better as well. There are two exceptions to this generalization, however. First, according to the efficiency plot, svd falls into the category of Delayed Improvement. However, the effectiveness results for svd show that the standard search actually found a better sequence than the search with loop unrolling. A possible explanation for the improved search efficiency is that, because the search spaces are different, the larger search space was more amenable to the search technique. Or, it is possible that the initial random starting points were of higher quality for the standard search. The second anomaly is applu. This benchmark falls in the Degradation efficiency category, but the search effectiveness results indicate the hill climber with loop unrolling achieves better results than without. This may occur because the search efficiency plot is an average. Even though the absolute best sequence found was better, the average sequence found was worse. This just means that the larger search space contains better solutions, but it takes the search additional time to find them.

Finally, because solve and svd appear to show no improvement, or even a performance degradation when unrolling is applied, they provide good insight into the
search behavior for benchmarks in which unrolling is not possible. solve shows a slight efficiency drop when loop unrolling is included in the search; svd, on the other hand, shows a slight improvement. These results suggest that adding a new transformation may not have a large impact on search efficiency, if the transformation exploits new opportunities that are present in the benchmarks. However, it is likely that search efficiency will start to degrade if the search space grows too large without a sufficient increase in solution quality in the search space.

4.4.3 Pass Frequency

Section 3.1 presented some preliminary experimental results that showed the high occurrence rate of loop peel. On average, the adaptive compiler selected loop peel 26% of the time in sequences within 1% of the best for each benchmark. Table 4.3 shows the frequency measurements for loop unrolling in addition to loop peel, and includes results for the standard hill climber and the loop-unrolling hill climber. When loop unrolling is present, the loop peel frequency drops from 26% to 15%. Most benchmarks show a large reduction in the frequency of loop peel; adpcm_coder is the most drastic, with loop peel dropping from 42.8% to 1.1%. tomcatv shows a drop from 4.8% to 0.0%. Three benchmarks show an increase in loop-peel frequency: seval, solve, and svd. Two of these, however, showed poor response to loop unrolling in Section 4.3. Loop unrolling, on average, is selected 11.5% of the time. All benchmarks, except for the two for which loop unrolling was not effective, use loop unrolling over 10% of the time. Section 4.4.1 claims that the adaptive compiler is able to discover effective compilation sequences that do not use a transformation when it is not profitable. The loop-unroll frequency for solve and svd are both under 1%, providing strong support for this claim.

Figure 4.12 presents a summarized graph of the average frequency for each transformation, for both the standard hill climber and the loop-unrolling hill climber. Figures A.1-A.10 in Appendix A provide the detailed frequency plots for each bench-
<table>
<thead>
<tr>
<th></th>
<th>Peel Frequency</th>
<th>Unroll Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Standard</td>
<td>With Unroll</td>
</tr>
<tr>
<td>adpcm_coder</td>
<td>42.8%</td>
<td>1.1%</td>
</tr>
<tr>
<td>adpcm_decoder</td>
<td>21.8%</td>
<td>12.0%</td>
</tr>
<tr>
<td>applu</td>
<td>40.0%</td>
<td>10.0%</td>
</tr>
<tr>
<td>matrix300</td>
<td>13.1%</td>
<td>12.9%</td>
</tr>
<tr>
<td>rkf45</td>
<td>41.5%</td>
<td>13.8%</td>
</tr>
<tr>
<td>seval</td>
<td>15.3%</td>
<td>20.6%</td>
</tr>
<tr>
<td>solve</td>
<td>30.5%</td>
<td>31.3%</td>
</tr>
<tr>
<td>svd</td>
<td>24.0%</td>
<td>33.3%</td>
</tr>
<tr>
<td>tomcatv</td>
<td>4.8%</td>
<td>0.0%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>26.0%</strong></td>
<td><strong>15.0%</strong></td>
</tr>
</tbody>
</table>

Table 4.3: Loop peel and loop unroll frequencies for the standard search space and the search space augmented with loop unrolling.

mark. On average, the frequency of most of the optimization passes remains similar between the two search spaces. The only major differences are with loop peel and loop unrolling, as expected. For those two transformations, the reduction in loop peel is roughly equivalent to the increase in loop unroll. The slight changes in the frequency of the other transformations indicate that loop unrolling requires different enabling and cleanup transformations than loop peeling requires. It would be interesting to analyze these changes in frequency, as it might offer insight into which passes are needed to effectively optimize unrolled code. This is left for future work, however.
Figure 4.12: Average Pass Frequency
Chapter 5

Conclusion

Adaptive compilation is an effective method for improving compiler output significantly beyond the output of traditional fixed-sequence compilers. Because adaptive compilation reduces to the problem of searching for optimal points in a search space of exponential size, this method often requires an unreasonable amount of time. Most research focuses on improving the search techniques and pruning the search space, successfully decreasing the compilation time. This thesis, on the other hand, demonstrated that careful selection of compiler transformations is a critical step in improving the output and reducing the compile time cost of current adaptive compilation techniques. An exemplification of this concept, described throughout the thesis, is the story of tuning the Rice University research adaptive compiler by adding a new optimization, loop unrolling.

The high occurrence of loop peel in the best compilation sequences discovered by the adaptive compiler suggested that the compiler would benefit from the addition of a new optimization for reducing loop overhead. Loop unrolling is often treated as an engineering problem, because it is straightforward in the context of a high-level intermediate representation (IR). For a low-level IR, however, unrolling presents unique challenges, because the loop body and induction variables are often hidden. This thesis illustrated the difficulties with loop unrolling in this low-level context, and provided a complete algorithm for solving the problem. The algorithm utilizes previous research on loop detection in flow graphs and induction variable detection in SSA-graphs. Combining information from these two analysis phases allows the unrolling transformation to be implemented safely in the general case.
Simply implementing a loop unroller and adding it to the compiler did not end the story; the effect of this change needed to be assessed. Evaluating a compiler transformation in a fixed sequence compiler has been an important practice since the emergence of compilers. The primary evaluation metric is usually the performance impact on output code. Production compiler writers are certainly concerned with the running time of a transformation, but asymptotically inefficient algorithms are rarely acceptable. In contrast, adaptive compilation is a means of trading compilation time for improved performance. Consequently, compilation time, or more precisely search time, becomes a very important evaluation criteria. This thesis recognized the two important metrics for evaluating an adaptive compiler: effectiveness and efficiency. Effectiveness measures the quality of the best compilation sequence that the compiler finds. Efficiency measures the amount of time it takes to find a given sequence. Both metrics are important, because they reflect that any improved quality achieved by the adaptive compiler may come at a cost of compilation time.

To determine the benefit of adding the loop unroller to the adaptive compiler, both metrics were measured on a suite of nine benchmark programs. The effectiveness of the adaptive compiler improved an average of 10%, while the efficiency improved for seven of the nine benchmarks. Only two benchmarks showed a slight decrease in search efficiency, corresponding to the extra time spent evaluating sequences that showed no improvement. These benchmarks are not well-disposed to loop unrolling, however, as they showed performance degradation from loop unrolling in a fixed sequence.

In conclusion, this thesis showed that adding a new transformation to an adaptive compiler can have both positive and negative effects on the two important metrics, effectiveness and efficiency. Adding a transformation increases the size of the search space, altering the delicate balance between the potential increase in effectiveness and decrease in efficiency. Because the choice of transformations in an adaptive compiler has a strong impact on its performance, selecting the best set of transformations is
an important method for tuning an adaptive compiler.

5.1 Future Work

While this thesis answers questions about the underlying mechanics of adaptive compilation, it also exposes new questions and potential research opportunities. Although these issues are beyond the scope of this thesis, they remain interesting research problems that are avenues of future work.

This thesis demonstrates a technique that could be used for evaluating the overall utility of an optimization pass. The experiments in this thesis only consider adaptive compilation performance, but the method could easily be extended to traditional compiler evaluation. The same adaptive compilation framework could be used to measure effectiveness and efficiency of each transformation, and comparing the results might indicate a ranking of importance of each optimization. The advantage of this approach over the traditional fixed sequence approach is that it makes careful consideration of the interactions between optimizations. The full potential of an optimization pass is only realized when detrimental interactions are avoided and beneficial interactions are exploited.

The size and contents of the adaptive compilation search space correspond directly with the search effectiveness and efficiency. While this thesis showed that the careful addition of a new transformation can improve the search effectiveness without significantly hurting efficiency, future work may show that some transformations can be removed. For example, the Rice University compiler uses several versions of value numbering that operate on different scoping levels, from single basic blocks to the entire procedure. Because there is much overlap in the effects of these different versions of value numbering, their presence may unnecessarily increase the size of the search space. In other words, removing one or more of the value numbering passes from the adaptive compiler may shrink the search space size without diminishing the quality of the search space contents. By adapting the methods in this thesis, the
result of removing an optimization pass from the compiler could be evaluated. If the hypothesis is correct, removing the partially redundant value numbering passes should result in an improved search efficiency without a significant loss in the search effectiveness.

Naive unrolling, described in Chapter 3, is used as the first step for general unrolling. In cases where the general unrolling cannot be applied—for example, when the loop is not controlled by a test between a linear induction variable and loop-invariant value—then naive unrolling may still offer some performance benefit. Naive unrolling may create a larger extended basic block that might be amenable to a speculative scheduling algorithm. Examining the improvement achievable with this method is an interesting research problem.

Because the main focus of the thesis wasn’t loop unrolling, many interesting opportunities to study this transformation were neglected. Several interesting questions arose that would be worth looking into. The measurements of the frequency for each transformation showed changes after the loop unroller was added. Presumably, the changes in frequency of other transformations indicate that a slightly different emphasis of transformations are needed for enabling the unroller and optimizing unrolled code. This change could potentially be identified with a statistical or data mining technique. It would be interesting to determine which transformations are more important in the presence of loop unrolling. Additionally, the unroller was allowed to be applied at any position in the compilation sequence. This flexibility may not be necessary; that is, the adaptive compiler may be just as effective if it were only allowed to apply unrolling in a single fixed position. The framework implemented for this thesis provides a solid foundation for exploring many of these hypotheses.

Finally, the experimental results in this thesis measure dynamic instruction count for the abstract ILOC instruction set. This approach creates stability in the measurements, because the simulation is deterministic. The main argument against using dynamic instruction count as an objective function is that it does not account for cache
effects. Grosul argued that dynamic instruction count was sufficient for his work, however. He showed that dynamic instruction count correlates strongly to actual execution time [30]. The main reason he was able to show this is that the optimizations in the Rice University research compiler do not target the memory hierarchy. Instead, their main goal is usually reducing dynamic instruction count through mechanisms such as redundancy elimination or loop-invariant code motion. Thus, these transformations rarely impact cache behavior. Since this thesis extends Grosul's work, it followed the assumption that dynamic instruction count is a sufficient objective function. Loop unrolling, however, may have a larger impact on instruction cache behavior than the existing transformations. Therefore, it would be beneficial to repeat Grosul's experiments to measure the ability of dynamic instruction count to predict actual running time. If the measurements do not correlate as well when loop unrolling is included, it would be beneficial to re-examine the behavior of the hill climber using execution time as the objective function.
## Appendix A

### Additional Data

| c | Constant Propagation [59] |
| d | Dead Code Elimination [38] |
| g | Global Value Numbering [7] |
| l | Partial Redundancy Elimination (PRE) [47] |
| m | Renaming (for PRE and LCM) [16] |
| n | Control Flow Cleanup [16] |
| o | Peephole Optimization [24] |
| p | Loop Peeling [16] |
| r | Algebraic Reassociation [9] |
| s | Register Coalescing [11] |
| t | Operator Strength Reduction [20] |
| u | Local Value Numbering [10] |
| v | SCC-based Value Numbering [10] |
| x | Dominator-based Value Numbering [10] |
| y | Extended-block Value Numbering [10] |
| z | Lazy Code Motion (LCM) [41] |

| U | Loop unrolling |

Table A.1: The alphabet for the adaptive compiler; the top portion of the table indicates the existing transformations, while the bottom portions indicates the extension provided by this thesis.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>String</th>
<th>Instruction Count</th>
<th>String</th>
<th>Instruction Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcmcoder</td>
<td>ppppppczpc</td>
<td>8 983 004</td>
<td>uCrodxxUc</td>
<td>7 397 342</td>
</tr>
<tr>
<td></td>
<td>pppppp1lp</td>
<td>8 983 004</td>
<td>tCrodxxUc</td>
<td>7 397 342</td>
</tr>
<tr>
<td></td>
<td>pppppp1ld</td>
<td>8 983 744</td>
<td>mCrodxxUc</td>
<td>7 397 638</td>
</tr>
<tr>
<td></td>
<td>pppppp1lg</td>
<td>8 983 744</td>
<td>zCrodxxUc</td>
<td>7 397 638</td>
</tr>
<tr>
<td></td>
<td>pppppp1lm</td>
<td>8 983 744</td>
<td>1CrodxxUc</td>
<td>7 397 638</td>
</tr>
<tr>
<td>adpcmdecoder</td>
<td>prpcoldpdp</td>
<td>7 717 116</td>
<td>uC1UrUvUcp</td>
<td>6 197 160</td>
</tr>
<tr>
<td></td>
<td>prpcoldpdp</td>
<td>7 717 264</td>
<td>yC1UrUvUcm</td>
<td>6 197 160</td>
</tr>
<tr>
<td></td>
<td>prpcoldpdc</td>
<td>7 717 412</td>
<td>yC1UrUvUcp</td>
<td>6 197 160</td>
</tr>
<tr>
<td></td>
<td>prpcoldpdc</td>
<td>7 717 412</td>
<td>yC1UrUvUcp</td>
<td>6 197 160</td>
</tr>
<tr>
<td></td>
<td>prpcoldpdc</td>
<td>7 717 412</td>
<td>yC1UrUvUcp</td>
<td>6 197 160</td>
</tr>
<tr>
<td>applu</td>
<td>zrpoddpdpv</td>
<td>266 067 724</td>
<td>rzoUc1tdpcm</td>
<td>258 055 438</td>
</tr>
<tr>
<td></td>
<td>vropppcpdv</td>
<td>266 119 216</td>
<td>rzoUctdpcm</td>
<td>258 394 142</td>
</tr>
<tr>
<td></td>
<td>vropdpdpv</td>
<td>266 152 677</td>
<td>rzoUctdpcm</td>
<td>259 706 787</td>
</tr>
<tr>
<td></td>
<td>zrpodpdpdv</td>
<td>269 990 972</td>
<td>rzoUctdpcm</td>
<td>259 706 787</td>
</tr>
<tr>
<td></td>
<td>zrpodpdpdv</td>
<td>270 027 254</td>
<td>rzoUctdpcm</td>
<td>259 878 740</td>
</tr>
<tr>
<td>matrix300</td>
<td>opdtxtcpdv</td>
<td>9 637 513</td>
<td>smvUvOxdv</td>
<td>7 202 981</td>
</tr>
<tr>
<td></td>
<td>opdtxtcpdv</td>
<td>9 637 928</td>
<td>ugpvUvOxdv</td>
<td>7 202 981</td>
</tr>
<tr>
<td></td>
<td>opdtxtcpdv</td>
<td>9 637 928</td>
<td>umpvUvOxdv</td>
<td>7 202 981</td>
</tr>
<tr>
<td></td>
<td>opdtxtcpdv</td>
<td>9 639 528</td>
<td>umpvUvOxdv</td>
<td>7 202 981</td>
</tr>
<tr>
<td></td>
<td>opdtxtcpdv</td>
<td>9 639 528</td>
<td>umpvUvOxdv</td>
<td>7 202 981</td>
</tr>
<tr>
<td>rkf45</td>
<td>ppoppopncdx</td>
<td>179 500</td>
<td>osUvUcpacd</td>
<td>169 150</td>
</tr>
<tr>
<td></td>
<td>ppoppopncdx</td>
<td>179 500</td>
<td>osUvUcpacd</td>
<td>169 150</td>
</tr>
<tr>
<td></td>
<td>ppoppopncdx</td>
<td>179 500</td>
<td>osUvUcpacd</td>
<td>169 152</td>
</tr>
<tr>
<td></td>
<td>ppoppopncdx</td>
<td>179 500</td>
<td>osUvUcpacd</td>
<td>169 152</td>
</tr>
<tr>
<td></td>
<td>ppoppopncdx</td>
<td>179 500</td>
<td>osUvUcpacd</td>
<td>169 152</td>
</tr>
<tr>
<td>seval</td>
<td>proxdtdtpv</td>
<td>674</td>
<td>Uoppzcxmd</td>
<td>636</td>
</tr>
<tr>
<td></td>
<td>proxdtdtpv</td>
<td>674</td>
<td>Uoppzcxmd</td>
<td>636</td>
</tr>
<tr>
<td></td>
<td>proxdtdtpv</td>
<td>674</td>
<td>Uoppzcxmd</td>
<td>636</td>
</tr>
<tr>
<td></td>
<td>proxdtdtpv</td>
<td>675</td>
<td>Uoppzcxvm</td>
<td>636</td>
</tr>
<tr>
<td>solve</td>
<td>gspxrcpovdv</td>
<td>816</td>
<td>tspoldpvcv</td>
<td>822</td>
</tr>
<tr>
<td></td>
<td>gspxrcpovdv</td>
<td>816</td>
<td>tspoldpvcv</td>
<td>822</td>
</tr>
<tr>
<td></td>
<td>gspxrcpovdv</td>
<td>816</td>
<td>tspoldpvcv</td>
<td>822</td>
</tr>
<tr>
<td></td>
<td>gspxrcpovdv</td>
<td>816</td>
<td>tspoldpvcv</td>
<td>822</td>
</tr>
<tr>
<td></td>
<td>gspxrcpovdv</td>
<td>816</td>
<td>tspoldpvcv</td>
<td>822</td>
</tr>
<tr>
<td>svd</td>
<td>vnrpydypc</td>
<td>3 823</td>
<td>ppyrodznvc</td>
<td>3 781</td>
</tr>
<tr>
<td></td>
<td>vnrpydypc</td>
<td>3 823</td>
<td>pmyrppdovp</td>
<td>3 796</td>
</tr>
<tr>
<td></td>
<td>vnrpydypc</td>
<td>3 825</td>
<td>pgrtppdvp</td>
<td>3 796</td>
</tr>
<tr>
<td></td>
<td>vnrpydypc</td>
<td>3 836</td>
<td>pymrpsodvp</td>
<td>3 801</td>
</tr>
<tr>
<td></td>
<td>vnrpydypc</td>
<td>3 857</td>
<td>pymrppdmdm</td>
<td>3 801</td>
</tr>
<tr>
<td>tomcatv</td>
<td>odtvstcdg</td>
<td>200 463 901</td>
<td>ouzUvcdts</td>
<td>182 288 125</td>
</tr>
<tr>
<td></td>
<td>odtvstcdg</td>
<td>200 463 901</td>
<td>ouzUvcdts</td>
<td>182 288 125</td>
</tr>
<tr>
<td></td>
<td>odtvstcdg</td>
<td>200 463 901</td>
<td>ouzUvcdts</td>
<td>185 548 510</td>
</tr>
<tr>
<td></td>
<td>odtvstcdg</td>
<td>200 463 903</td>
<td>ouzUgcdts</td>
<td>187 094 186</td>
</tr>
<tr>
<td></td>
<td>odtvstcdg</td>
<td>200 474 183</td>
<td>ouzUdcldts</td>
<td>187 094 186</td>
</tr>
</tbody>
</table>

Table A.2: The five best sequences found by the hill climber for each benchmark
Figure A.1: Average Pass Frequency

Figure A.2: Pass Frequency for adpcm_coder

Figure A.3: Pass Frequency for adpcm_decoder
Figure A.4: Pass Frequency for applu

Figure A.5: Pass Frequency for matrix300

Figure A.6: Pass Frequency for rkf45
Figure A.7: Pass Frequency for seval

Figure A.8: Pass Frequency for solve

Figure A.9: Pass Frequency for svd
Figure A.10: Pass Frequency for tomcatv
Bibliography


puter Science Institute, October 2003.


[31] Yi Guo, Devika Subramanian, and Keith D. Cooper. An effective local search algorithm for an adaptive compiler. In SMART '07: Workshop on Statistical and


