Tailoring Traditional Optimizations for Runtime Compilation

by

Anshuman Dasgupta

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

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Keith D. Cooper, Professor, Co-Chair
Computer Science

Ken Kennedy, Professor, Co-Chair
Computer Science

Yehia Massoud, Assistant Professor
Electrical and Computer Engineering

Rice University, Houston, Texas

September, 2006
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 3256681
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
Tailoring Traditional Optimizations for Runtime Compilation

Anshuman Dasgupta

Abstract

Runtime compilation, due to its online nature, presents unique challenges and opportunities to compiler designers. Since compilation occurs during program execution, a just-in-time compiler (JIT) must be judicious in expending compilation time. The literature on traditional, offline compilers describes numerous program transformation techniques that strive to increase execution efficiency. However, while optimization passes for static compilers are well understood and have been thoroughly investigated, many such transformation algorithms cannot be implemented on a JIT environment due to compilation-time constraints. Further, offline algorithms are not designed to exploit information available to an online compiler at program execution time.

The thesis of the research presented in this document is that program optimization techniques designed for traditional, offline compilers can be profitably adapted for a runtime compiler by effectively respecting the constraints imposed on compilation time and by exploiting the opportunities available in a runtime compilation environment. To that end, the dissertation explores the complexity of implementing program transformations for a runtime compiler and redesigns two optimization techniques for a JIT: register allocation and loop unrolling. The two transformations present contrasting challenges when they are included in a runtime compiler. While several offline, heuristic allocation algorithms achieve impressive results, they consume large amounts of compilation-time that are typically unacceptable for a JIT. We describe the design of two allocation algorithms that reduce allocation time while
preserving the advantages of strong techniques authored for offline compilers. An experimental evaluation of the new algorithms demonstrates their effectiveness on a runtime compilation environment.

While a runtime compiler is limited by the constraints imposed by its environment, compiling just prior to program invocation provides certain advantages over an offline compiler. In particular, it can examine information only available at program execution time. We describe the design of a lightweight runtime value-examining mechanism and a loop unrolling algorithm that work in tandem. Our experimental results indicate that the runtime unroller achieves significant improvements on floating point, scientific benchmarks.

In summary, thus, the research described in this dissertation demonstrates how compiler optimization algorithms can be effectively tailored for runtime compilation.
Acknowledgments

When asked to describe what graduate school was like, Jeff Nunemacher, my undergraduate Computer Science professor replied that it was, in Mark Twain’s words, the best and worst of times. On retrospect, I am apt to agree with him. Fortunately, however, the good times at Rice have considerably overshadowed the not-so-good. This is, in no small part, due to the friends and colleagues I had the pleasure of meeting at Rice. First, I would like to thank my advisors, Keith Cooper and Ken Kennedy for their support, guidance, and for fostering a wonderful environment for research. I would also like to thank Yehia Massoud, the third member of my thesis committee who was very helpful in offering valuable feedback. My fellow graduate students ensured that my graduate school experience was thoroughly enjoyable. The friendships I formed at the university will last a lifetime. I would especially like to thank Daniel Chavarría, Romer Gill, Tim Harvey, Anirban Mandal, Cheryl McCosh, Santashil Palchaudhuri, Apan Qasem, Jeff Sandoval, and Todd Waterman for all the conversations (and beer) we shared. Tim also served as an invaluable sounding board for research ideas and provided immensely useful feedback.

The professors of the Math/CS department in my undergraduate school – Ohio Wesleyan University – helped strengthen my interest in Computer Science. I am very grateful to Jeff Nunemacher, Kathy Radloff, and Alan Zaring for their teaching and mentoring.

This work would have been impossible without the support and encouragement of my family. Ma and Bapi, you have always been a source of inspiration to me and I can never fully express how fortunate I am to have such amazing parents. Didi and Rana, my sister and brother-in-law, have always offered me support, advice, and the feeling
that I was never far from home (in addition to unfettered access to their well-stocked fridge). Lastly, I would like to thank Semonti, my wife, who has patiently shared and endured the vagaries of my graduate school career. Her love, companionship, and encouragement have enriched my life and helped me become a better student and, more importantly, a better person.
To Ma and Bapi
Contents

Abstract i
Acknowledgments iii
List of Illustrations x
List of Tables xiii

1 Introduction 1
1.1 Redesigning traditional optimizations for runtime compilation . . . . . 4
   1.1.1 Incremental recomputation of program analyses . . . . . . . . 6
   1.1.2 Bartering optimization quality for compilation time . . . . 7
   1.1.3 Effective runtime loop unrolling . . . . . . . . . . . . . . 8

2 Related Work 10
2.1 Machine-independent code and JIT technology . . . . . . . . . . 10
2.2 Optimization passes in a JIT . . . . . . . . . . . . . . . . . . . 12
   2.2.1 Register allocation . . . . . . . . . . . . . . . . . . . . . . . . 12
2.3 Intermediate representations in runtime compilers . . . . . . . . 14
2.4 Incremental optimization . . . . . . . . . . . . . . . . . . . . . 15
2.5 Resource-constrained compilation . . . . . . . . . . . . . . . . 16
2.6 Loop unrolling . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
2.7 JIT annotations . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

3 Examining Strong Register Allocator Algorithms 21
3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
3.2 Overview of the allocator algorithms . . . . . . . . . . . . . . . . 22
4 Tailoring Graph-coloring Allocation for Runtime Compilers

4.1 Identifying the expensive components of the algorithm .................. 32
4.2 Interference graph building ................................................. 34
  4.2.1 Algorithm for identifying interferences .......................... 34
  4.2.2 Rebuilding the interference graph ................................ 35
4.3 Redesigning the interference graph builder for a runtime compiler .. 37
  4.3.1 Spill code insertion .................................................. 37
  4.3.2 The lossless (precise) allocator .................................. 38
  4.3.3 The lossy (imprecise) allocator .................................. 40
  4.3.4 Sources of imprecision in the lossy allocator ................. 48
  4.3.5 Issues common to both allocators ............................... 49
4.4 Experimental results ...................................................... 50
  4.4.1 Performance of the allocators in an offline compiler ........ 50
  4.4.2 Analysis of the imprecision in the lossy allocator ........... 55
  4.4.3 Performance of the allocators in a runtime compiler ........ 57
  4.4.4 The effect of input data size on runtime ....................... 59
4.5 Conclusions .............................................................. 64

5 Optimization Passes on a Runtime Compiler .............................. 65

5.1 Introduction ............................................................. 65
6 Effective and Efficient Strategies for Runtime Loop Unrolling

6.1 Introduction ............................................. 76
6.2 Loop unrolling ............................................ 77
6.3 Benefits of loop unrolling ................................ 77
6.4 Methodology and experimental framework .............. 80
   6.4.1 Two motivational examples ......................... 81
6.5 The case for effective loop unrolling at runtime ........ 83
   6.5.1 Performance enhancements ......................... 83
   6.5.2 Space savings ....................................... 88
6.6 Examining runtime program values: background .......... 89
6.7 Description of our runtime unrolling algorithm .......... 91
   6.7.1 A lightweight value examination mechanism for loop unrolling 93
   6.7.2 Choosing unroll factors ............................ 97
6.8 Experimental results ..................................... 98
   6.8.1 Opportunities exploited by runtime loop unrolling ..... 99
   6.8.2 Effect on application performance ................. 102
   6.8.3 Analyses of results ................................ 103
   6.8.4 Overhead of unrolling ............................. 105
   6.8.5 Post-unroll cleanup loop usage .................... 105
6.9 Conclusion ............................................... 109
7 Conclusion

7.1 Using strong register allocation techniques in a runtime environment  110
7.2 Exploiting runtime value information to conduct loop unrolling . . . 112

Bibliography 113
Illustrations

1.1 Optimization quality and Compilation Time ........................................ 7

3.1 Overview of the Chaitin-Briggs allocator ........................................... 22
3.2 Example tile tree: (a) CFG; (b) tiles overlaid on CFG; (c) the tile tree. 25
3.3 The Callahan-Koblenz Allocator ...................................................... 25
3.4 Compilation time for the register allocators on the SPEC integer,
    epic, and jpg benchmarks ............................................................ 30
3.5 Execution time for register allocated SPEC integer, epic, and jpg
    benchmarks .................................................................................. 30

4.1 Contribution of phases in Chaitin-Briggs to the total allocation time . 33
4.2 An interference graph constructed from a simple procedure ............... 35
4.3 Algorithm for constructing the interference graph .............................. 36
4.4 Spill code insertion ........................................................................ 38
4.5 Lossless algorithm for reconstructing the interference graph after spilling 39
4.6 Interference graph with edges partitioned into definition and use edges. 41
4.7 Lossy algorithm for reconstructing the interference graph after spilling 43
4.8 Compilation times with the unmodified, lossless, and lossy register
    allocators for the SPEC 2000 integer benchmarks. ............................ 51
4.9 Execution times for code allocated with the unmodified, lossless, and
    lossy register allocators for the SPEC 2000 integer benchmarks. ......... 52
4.10 Contribution of phases in the lossy allocator. The values are geometric means over all benchmarks. ........................................... 53

4.11 Contribution of phases in the lossless allocator. The values are geometric means over all benchmarks. ........................................... 53

4.12 Comparison of different phases in the three allocators. The values are geometric means over all benchmarks and are relative to the Chaitin-Briggs allocator. ................................................................. 54

4.13 Comparison of modified allocators with linear-scan on a runtime compiler ................................................................. 58

4.14 The graph shows the runtime of SPEC parser benchmark as the input size increases. ......................................................... 60

4.15 The graph shows the runtime of SPEC crafty benchmark as the input size increases. ......................................................... 61

4.16 The graph shows the runtime of SPEC gzip benchmark as the input size increases. ......................................................... 62

5.1 Results of running a single optimization pass on the LLVM JIT for benchmarks from the SPEC floating point suite. The results are averages over all benchmarks and are relative to the observed execution time of an unoptimized program. ........................................... 68

5.2 Results of running a single optimization pass on the LLVM JIT for benchmarks from the SPEC integer suite. The results are averages over all benchmarks and are relative to the observed execution time of an unoptimized program. ........................................... 70

6.1 Unrolling a loop by a factor of three. The unrolled loop on the right contains three copies of the original loop body followed by a cleanup loop ................................................................. 77
6.2 Functional Units on the Intel Pentium 4 processor ............... 80
6.3 Enabling better instruction scheduling due to loop unrolling .... 81
6.4 Execution time for umCEM application on input set 1 as the unroll factor for the innermost loop varies on a Pentium 4. The bar graph shows the relative improvements achieved by an unroll factor over the original loop. ........................................... 84
6.5 Execution time for umCEM application on input set 2 as the unroll factor for the innermost loop varies on a Pentium 4. The bar graph shows the relative improvements achieved by an unroll factor over the original loop. ........................................... 85
6.6 Runtime code generation mechanism in the LLVM JIT. The top panel depicts how the LLVM JIT generates code on-demand during application execution. The bottom panel shows how a sample procedure stub is generated. ..................................... 92
6.7 The runtime loop unrolling framework. ............................... 95
6.8 Overview of runtime unrolling algorithm. The control-flow graph before and after unrolling is shown in Figure 6.14 ................. 96
6.9 Compute unroll factor .................................................. 97
6.10 Breakdown of loop bounds in inner loops for our benchmark suite. ... 100
6.11 Breakdown of loops that the runtime loop unroller can examine. ... 101
6.12 Effect of runtime loop unrolling on benchmarks. This graph shows the performance of the Runtime-2 unrolling strategy. ............... 103
6.13 Effect of runtime loop unrolling on benchmarks. This graph shows the performance of the Runtime unrolling strategy. ............... 104
6.14 The control flow graph before and after loop unrolling is applied. In the right panel, cleanup loop blocks have been highlighted in gray. ... 106
## Tables

3.1 Compilation and Execution Times for the Chaitin-Briggs and Callahan-Koblenz allocator .................................................. 29

4.1 Extra spills and edges added by the lossy allocator .................. 56

5.1 Description of optimizations used in experiments to determine effectiveness of passes .................................................... 67

5.2 Optimizations ranked in descending order of performance for benchmarks from the SPEC floating point suite. The numbers are geometric means over all benchmarks and are relative to the performance of a program with no optimization (represented by “none” in the table). .................................................... 69

5.3 Optimizations ranked in descending order of performance for benchmarks from the SPEC integer suite. The numbers are geometric means over all benchmarks and are relative to the performance of a program with no optimization (represented by “none” in the table). .................................................... 71

6.1 apsi and umCEM program characteristics. apsi runtime characteristics have been gathered by running the application on the SPEC provided ref data set. .................................................... 82

6.2 Growth in bytecode representation due to offline unrolling of apsi as unroll factor increases .................................................... 89
6.3 Description of Benchmarks. SPEC descriptions have been summarized from [86].

6.4 Execution frequency of unrolled and cleanup loops
Chapter 1

Introduction

Programs written for the first generation of stored-program electronic computers, such as the EDVAC, were completely machine-specific. A program was written for a particular machine, encoded in machine-code on a punch tape, and converted by a mechanical tape-reader into an electronic signal. On later computers, the advent of assemblers eased the programmer’s task by enabling some degree of symbolic programming. Since then, we have steadily moved away from machine-specific encodings. Early compilers written for high-level languages — Backus’ breakthrough effort being the most famous [11] — allowed a programmer to author applications without considering the architectural features of the target machine. The high-level program, a machine independent representation, was translated to machine code by the compiler. Inspired by early efforts from the Smalltalk-80 and the UCSD p-system research groups [45, 23], the Java programming language, introduced in the 1990’s, continued the trend of divorcing program representations from machine characteristics. Java translated the source-code into a compact and portable intermediate format that could then be easily transported and executed on a diverse set of computing platforms.

In recent years, platform-independent program representations that are interpreted or compiled at runtime on target machines have grown in popularity. Java and C# are common examples of languages implemented on these frameworks. Three developments have contributed greatly to the increased deployment of such frameworks: the rising availability and usage of the Internet, the explosive growth in mobile consumer devices, and the drastic increase in processor power. Internet usage has in-
creased rapidly in the last decade and many expect the trend to continue [82, 33]. A portable program that can be run on a computer connected to the Net is immensely beneficial since it allows content providers the opportunity to personalize content and serve pre-compiled applications to the user. Mobile devices, specifically cell-phones and PDA’s, have registered an equally impressive growth and consequently have become ubiquitous [81]. Moreover, most cell-phones incorporate several non-telephonic applications such as schedulers, photo-management software, and MP3 players. The drastic gains in processing power seen on consumer-grade computer systems have also played a part in popularizing systems that support these portable, dynamically compiled representations. Modern processors allow the virtualization mechanisms of these frameworks to execute with reasonable efficiency. As a result of these developments, and due to the widely varying computing environments present on desktops and mobile devices, frameworks that enable portable software development have assumed great importance. Several vendors offer integrated environments for the development and execution of portable code. For instance, Sun’s Java 2, Enterprise Edition (J2EE), and Microsoft’s .Net platform allow development and execution of code on both desktop and servers. Both vendors also offer a modified version of their desktop infrastructure for mobile devices – Java 2, Mobile Edition (J2ME), and .Net Compact Framework (CFW) respectively. These environments allow the programmer to compose one program and subsequently execute that program on different desktop or mobile devices.

Compiling programs on these frameworks is a two-step process: the source-code is first compiled to a machine-independent representation called the bytecode.\(^1\) The bytecode is then transported to the target processor where it is interpreted by the vir-

\(^1\)So named because each opcode occupies a single byte. The term bytecode dates back to the Smalltalk-80 systems [45]; more recently, it has become almost synonymous with Java’s bytecode format. In this document, we shall use the term bytecode to refer to a portable representation used for runtime compilation.
While such portable representations are advantageous since they allow programs to be executed on a wide variety of architectures, the run-anywhere feature of the code comes with a cost. Specifically, bytecode interpretation, when compared to traditional, static compilation, is plagued with a distinct disadvantage: the performance of interpreting portable code is slower than the execution of a natively compiled version of the program that has been specialized for the target architecture. Two major factors contribute to the relative inefficiency of portable code: the overhead of interpretation and runtime compilation, and the genericity of the code that, by its very nature, prohibits important machine-specific optimizations. To alleviate these problems, the virtual machine invokes a runtime compiler, also popularly known as a just-in-time compiler (JIT), for frequently executed code.

The JIT translates the bytecode to architecture-specific machine code and, in the process, attempts to optimize the resulting assembly code. A JIT shares many goals with traditional, offline compilers—it attempts to optimize its translation to emit the most efficient machine code. However, in contrast to other compilers, a JIT must be acutely cognizant of compilation time. Since the compiler is invoked at runtime, it must optimize for the sum of compile time and runtime. This constraint requires the JIT to strike a fine balance between conducting strong optimizations that tend to be computationally expensive and faster, but less-effective techniques. The decisions taken in choosing these optimizations have a profound impact on program execution efficiency. Our work will examine and address this critically important issue.

Optimization strategies in a JIT is a well-researched topic and the literature describes many efforts to reduce the compile-time requirements of a runtime compiler. To compensate for the constraints on compilation time, several researchers have suggested discarding traditional transformation algorithms for simpler techniques. In contrast, we believe that we can reformulate strong algorithms in a manner that preserves their algorithmic advantage over simpler strategies and yet adapts their functionality for a runtime environment. To this end, we will design and evaluate
strategies to increase the efficiency of an important traditional, global optimization –
register allocation – while striving to compromise as little as possible on compilation
quality. In addition to tailoring optimizations to respect the constraints imposed by
online compilation, we would also like to focus on the opportunities available as a
consequence of invoking the compiler at runtime. Therefore, we shall explore trans-
formation techniques that attempt to improve applications by exploiting information
available only at execution time. In particular, our work will have three major objec-
tives:

• We shall examine computationally expensive register allocation strategies in
detail and compare the compilation and execution time characteristics of these
techniques. Further, we shall redesign an expensive algorithm for a critically
important JIT-optimization – register allocation – to improve its compilation
time. We will use incremental techniques to reduce the time needed to compute
program analyses. This will allow us to save precious compilation time.

• A dynamic environment, such as a JIT, allows a compiler to examine and exploit
program information available only at runtime. We will compare the effective-
ness of different compiler optimizations in a JIT. We will use our examination
of compiler transformations to select a technique that can utilize runtime in-
formation to its benefit. We will then modify this transformation for runtime
compilation usage and evaluate our technique.

• We shall explore the trade-offs incurred between compilation time, observed
execution time, and optimization quality as a result of applying our techniques.

1.1 Redesigning traditional optimizations for runtime com-
pilation

While compiler literature is replete with descriptions of numerous optimizations,
many of these algorithms primarily focus on improving the quality of the optimized
code. In designing several of these transformations, compiler authors have utilized intricate mathematical techniques in their algorithms. Many of these techniques are computationally intensive. For an offline compiler, compilation time has no direct impact on application efficiency, so a reasonable increase in compile time can be tolerated for an optimization that improves the code. In a JIT environment, however, where compilation time is added to application runtime, expensive optimizations are feasible only if they provide consistent and large improvements. Thus, JITS typically exclude most expensive optimization techniques.

This challenge has led JIT authors to focus on the design and implementation of optimizations that use only modest amounts of compile time. Often, these faster algorithms use starkly different strategies than traditional optimizations, thereby sacrificing optimization quality for compilation speed. For instance, graph-coloring register allocation is a widely used technique in offline compilers. However, most JITS abandon the traditional, proven techniques that operate by analogy to graph coloring in favor of less expensive algorithms that are also less effective. These algorithms are intentionally constrained by the desire to reduce compile time. As a result, they generally produce less optimal code. After examining and contrasting these techniques, we realized that such a drastic redesign of an optimization is unnecessary and counter-productive for many compilation scenarios. In the work described in this document, we plan to leverage techniques used by mature, strong optimizations. In the first part of the dissertation, we shall reformulate an optimization - graph coloring register allocation - so that it is fast enough to use in a runtime compiler. Our reformulation sacrifices some optimization quality in favor of efficiency, but it preserves the essential flavor and, consequently, the proficiency of the graph-coloring allocators. Our approach yields the right balance between compilation efficiency and optimization efficacy and makes graph coloring allocation more attractive for use in

\[\text{The Hotspot server compiler is a notable exception - it conducts a Briggs-style register allocation pass and uses splitting and constraints on register coloring to gain efficiency [64].}\]
a runtime compiler. Our initial experiments, as detailed in Section 4.4, will validate the effectiveness of our strategy.

1.1.1 Incremental recomputation of program analyses

A compiler optimization pass generally consists of an analysis phase followed by a transformation phase. Moreover, some optimizations need to iterate over the two phases several times before emitting the final code. The analysis phase is often expensive and contributes greatly to the runtime of the optimization. By minimizing the time required for analysis, we can considerably increase the efficiency of the compiler pass. Therefore, we redesigned such an algorithm to use the analysis results computed on the first iteration of the pass to derive the analysis for subsequent iterations. In particular, we worked with a graph-coloring register allocator because it is known to contribute to application performance and because the cost of analysis and building the interference graph is well known to dominate the total cost of the technique. This results in a substantial reduction of compilation time as little or no loss in code quality. We have designed and evaluated two variants of an incremental strategy: lossless and lossy updates.

**Lossless incremental updates:** Lossless updates are a redesign of the original algorithm to reduce the compilation time of the optimization without compromising on its accuracy. We note that while these techniques will work especially well on runtime compilers, they can also be used for offline compilation. Further, they may be of particular interest in adaptive schemes that repeat an optimization several times to find the best setting of some parameter.

**Lossy incremental updates:** In contrast, algorithms redesigned with lossy updates also reduce compilation time but are less accurate than the original algorithm and thus, in general, can produce worse code. However, we have strived to minimize the degradation of the optimization results. Lossy incremental algorithms are well
suited for runtime compilation where efficient compilation is critical in decreasing application runtime.

We must emphasize that both kinds of updates are safe – they preserve the semantics of the original program. As we shall see in Chapter 4, we have implemented both strategies for the Chaitin-Briggs graph-coloring register allocator.

1.1.2 Bartering optimization quality for compilation time

For a particular compiler pass, we view the optimization quality attainable by different algorithms as a linear continuum. Consider the algorithms used to implement register allocation. As shown in Figure 1.1, less optimal strategies lie on the left of this continuum while stronger techniques lie on the right. As depicted on the diagram, compile-time efficient allocators would generally fall on the left and global, stronger allocators would fall on the right. On this scale, we have observed (as confirmed in Chapter 3) that there is a marked separation between algorithms designed for runtime compilers and algorithms designed for static compilation. Indeed, traditional optimizations in an offline compiler consistently outperform JIT optimizations. Unfortunately, traditional optimizations are relatively slow and are, thus, unsuitable
for runtime compilers. To address these shortcomings, we will redesign traditional algorithms to consume less compilation time by exploiting incremental strategies. To this end, we are prepared to trade-off some optimization quality. We, thus, intend our redesigned algorithms to reside between the space occupied by weaker JIT optimizations and traditional, strong optimizations. Finally, we shall explore the costs and the benefits of exchanging compilation time for optimization quality as application runtime increases.

1.1.3 Effective runtime loop unrolling

As described in the previous section, a key challenge in implementing compiler transformations for online compilers is the compilation time constraint posed by the runtime system. However, such a dynamic environment also provides unique opportunities for a JIT. Since the online compiler is invoked just before program execution, it has access to runtime information not available to a traditional, static compiler. Therefore, we wished to examine compiler transformations that can capitalize on the presence of execution-time program information. To this end, we decided to compare the effectiveness of different compiler passes on a JIT. We present this comparison in Chapter 5. Encouraged by the performance of one optimization – simple loop unrolling – we decided to modify the transformation to gather and exploit program values available only when the program executes. The advantages of our approach were three-fold. First, we were able to unroll loops more effectively at runtime. Offline compilers, in the absence of accurate profiling data, have to be conservative in choosing unroll factors for symbolic loop bounds. Our runtime loop unrolling algorithm was able to select unroll factors based on runtime program values. Second, since most JITs generate machine code lazily – they process a procedure only if it is invoked at runtime, our unrolling algorithm only unrolled loops in executed procedures. This approach is advantageous over an ahead-of-time strategy that unrolls loops eagerly potentially leading to code bloat and, consequently, increased bytecode transmission
time. Lastly, the runtime unroller, in contrast to an offline bytecode compiler, can accommodate architecture-specific characteristics in its code generation.

The rest of this document is structured as follows: Chapter 2 presents related work and will place our work in the context of current and past research. We will describe some preliminary experiments in Chapter 3 that have served as the motivation for much of this work. Chapter 4 will describe the design and implementation of two incremental approaches to register allocation for a runtime compiler. The following chapter contains an evaluation of the effectiveness of different optimization passes in a runtime compiler. Our goal in this chapter was to examine and select a transformation pass that can be modified to exploit program value information available only at runtime. In Chapter 6, we present the design and evaluation of a runtime loop unrolling algorithm and a lightweight value examination technique. Lastly, we will present our conclusions in Chapter 7.
Chapter 2

Related Work

Our work can be summarized as runtime-compilation research. Therefore, in contrasting our research with prior work, we shall describe various efforts that have attempted to reduce compilation time in a dynamic compiler. Since the first part of our research focuses on a particular global optimization – register allocation – we will examine research that intended to implement this and other global transformations for a JIT. In more general terms, our examination of strategies for JIT register allocation explores the complexity of designing optimizations for runtime compilers that have strict constraints on resources available to the compiler. Therefore, we shall also consider prior work done in compilation for resource-constrained environments and describe techniques such as annotations that attempt to reduce the cost of a dynamic compiler. Further, we will explore prior research that attempts to exploit information available at runtime to improve program performance. Specifically, we shall describe research conducted on loop unrolling – the focus of the second half of this dissertation.

2.1 Machine-independent code and JIT technology

With the introduction of the Java programming language in the last decade, Sun Microsystems popularized the concept of portable code that can be interpreted on diverse hardware platforms. However, the idea of a dynamically executed, architecture-independent, program representation predates Java. The UCSD p-system, developed in the 1970s allowed programs to be moved to a new hardware environment without rewriting. In contrast to contemporary virtual machines, this system provided an
abstract operating-system layer for programs across different architectures [23]. The p-System, however, did not implement a runtime optimizer. The Smalltalk system, developed by Xerox PARC, introduced much of the terminology and concepts that are used in current-day machine-independent frameworks [45]. Programs authored by Smalltalk users would be compiled to a bytecode representation which could then be interpreted by a virtual machine implemented on different architectures.

Researchers have long explored the merits of optimizing code at runtime. Early work in this field was motivated by the optimization opportunities inherent in dynamic languages. As narrated in [9], runtime compilers have evolved from their humble beginnings in APL to full-blown and sophisticated just-in-time compilers for Java. Several studies have highlighted the advantages of runtime optimization over pure interpretation of bytecode [41, 39, 40, 2]. Importantly, these studies have argued that the runtime system must be judicious in invoking the dynamic compiler. If the JIT is called indiscriminately, the overhead of the compiler swamps the benefit of producing faster native code. Therefore, many execution systems conduct a cost-benefit analysis before deciding to compile using the JIT. For instance, the IBM Java JIT is invoked when the number of method calls crosses a threshold [90]. \(^1\) Similarly, the HotSpot virtual machine interprets the bytecode until it detects a frequently executed region. The VM then triggers the JIT on that region, thereby compiling it to machine code [68]. The Dynamo system instruments the code and inserts counters in potentially important portions of a program and interprets the code until a counter exceeds a preset value [12]. That event triggers an optimization of the code. Recent work in the runtime optimization field has evaluated and advocated the use of more complex cost-benefit analyses [59, 7].

\(^1\)The paper describes their production JIT system, not the Jalapeno research compiler [19].
2.2 Optimization passes in a JIT

As we discussed in the previous section, careful JIT-invocation decisions greatly affect the runtime of the program. But, once triggered, the JIT too must contribute to keeping the dynamic compilation costs as low as possible. JITs are generally structured like conventional compilers – they subject the input to a series of pre-determined passes. JITs, however, focus more on reducing compilation time than other compilers. Consequently, JIT developers are more prone to choosing optimizations that are efficient, sacrificing some optimization quality in the process. This concern with compilation-time is illustrated by an analysis of current-day runtime compilers – literature on the HotSpot, Intel, and the IBM JITs reveal that they implement several local optimizations, preferring them over stronger, global techniques [68, 90, 1]. We shall focus on a crucial JIT optimization – register allocation – for which global algorithms exist but are considered too expensive for indiscriminate use on a JIT.

2.2.1 Register allocation

Register allocation is a critical pass in a compiler and consequently is a well-studied problem. The term is, in effect, used\(^2\) to describe two related mechanisms: allocation and assignment. Allocation consists of deciding which values in the intermediate representation will be stored in machine registers. Assignment is the process of mapping allocated values to the registers available on the architecture. Optimal register allocation has been proved to be NP-complete [84]. As a result, allocation is usually performed by a heuristic-driven algorithm. Early attempts at allocation consisted of simple, local algorithms [11, 62]. Near-optimal allocation of registers assumed increasing importance with the ever-widening disparity between processor and memory speeds. To tackle the NP-complete nature of the problem, some researchers modeled register allocation as a conflict graph colored using heuristics [61, 43, 28, 31]. Chaitin

\(^2\)or, more accurately, abused
et al. presented the first paper comprehensively describing a graph-coloring register allocator [28, 27]. Bernstein et al. and Bergner et al. subsequently added improvements to Chaitin's technique [14, 13]. Briggs et al. redesigned the Chaitin allocator to delay spill decisions until later on in the allocation process. This can lead to an improved coloring of the graph. In evaluating our JIT improvements, we shall adapt the refinement of Chaitin's algorithm formulated by Briggs et al. [18]. Chow proposed a different allocation technique based on graph-coloring [31]. He assigned priorities to live ranges by examining the uses, definitions, and moves between registers and memory. In contrast to Chaitin and Briggs, Chow constructs block-level live ranges. One perceived disadvantage of a Chaitin-style allocator is its inability to consider “topological” information about a program while allocating registers. Callahan and Koblenz attempted to address this shortcoming by constructing a hierarchical representation of the input program [21, 42]. Their algorithm then uses this data structure to identify good spill locations. In Chapter 3, we shall discuss the hierarchical allocator in more detail.

Most JITs implement simpler register allocation algorithms than those described above. Linear-scan techniques are particularly popular among JIT developers. Polletto and Sarkar devised a linear-scan algorithm that was faster than a graph-coloring allocator [76]. As the name suggests, the allocation proceeds by linearly scanning live ranges and mapping these ranges to registers. Recent research by Traub et al. and Mossenbock and Pfeiffer has refined this strategy [92, 71]. Since linear-scan techniques are generally more efficient than graph coloring, they are attractive to implement on a JIT. However, as is often the case, the improved compilation speed comes with a performance penalty. Graph-coloring allocation tend to outperform linear-scan methods [76, 92]. By redesigning strong allocation techniques, we intend to capture the best elements from both algorithms – reduced compilation time as well as proficient register allocation.

In addition to graph-coloring and linear-scan, many other approaches to register
allocation have been proposed [51, 15]. This document, however, will not describe these techniques.

2.3 Intermediate representations in runtime compilers

Machine-independent representations are used by popular runtime frameworks for portable code such as the Java Virtual Machine and the Common Language Runtime. The imposed portability of the bytecode, however, can prove to be a major obstacle for machine-specific optimizations. In Java, for instance, operands in the bytecode are arranged on a stack. While this leads to extremely compact code by enabling implicit operand names, it can also cause difficulties in conducting standard optimizations. Several researchers have attempted to address this concern. The Soot framework converts Java bytecode into a three-address format (called Jimple) or a SSA representation (Shrimple) [79]. CACAO is a runtime compiler that translates bytecodes into a register oriented representation. Their representation is similar to SSA format [56]. The Jalapeno compiler implements a virtual machine for Java. The compiler first constructs a three-address code based intermediate representation from Java bytecode. The authors state that this conversion helps in conducting machine specific transformations [19]. Moreover, when the compiler identifies a candidate for aggressive optimizations, it conducts SSA based transformations on the code.

Some researchers have experimented with an alternate solution to this problem. Krintz proposes the use of a hybrid representation that encodes frequently executed methods in SafeTSA [6] form while preserving the Java bytecode format for other code sections [57]. Amme et al. suggested similar encoding techniques at the Java class level [5]. Cartwright designed a novel scheme to incorporate SSA information into bytecodes [26]. While this representation is transparent to regular virtual machines, a suitably aware JIT can extract the SSA representation to conduct optimizations. A similar issue arises in decompiling x86 floating-point code. Cooper et al. handled this problem in the Vizer system [35].
For our experiments, we have used the LLVM compiler infrastructure [60]. This project is currently under active development at University of Illinois and suits our purposes particularly well. LLVM encodes programs in a language-independent, SSA-based intermediate representation and provides both a static and runtime compiler. It is well-designed, modular, and serves as a good research compiler.

2.4 Incremental optimization

Our primary goal is to reduce compilation time without making large compromises on code quality. To this end, we will redesign global allocation algorithms to incrementally update their analyses at the end of each iteration of the algorithm. Consequently, this will reduce the time spent on recomputing the analyses. Our research was motivated by related work in this area for offline compilers. Cooper et al. suggest incrementally refreshing inter-procedural analyses when a program unit is modified [36]. This technique reduces the cost of recompiling source code. Pollock and Soffa incrementally recalculate data-flow information and identify optimizations that are affected by program edits [77]. In a runtime compilation context, Burke et al. suggest incrementally marking procedures whose compilation has been invalidated by dynamic class loading [19]. In memory managed runtime systems, such as Java, automatic garbage collection is important to delivering good application performance. The literature describes several incremental garbage collection algorithms [49, 70]. These systems were designed to decrease the running time of the garbage collector thereby reducing pauses in program execution.

Our work shares similar concerns with these researchers – we too would like to reduce compilation time by avoiding the need to recompute program analyses information. We also wish to minimize the time spent away from program execution in a runtime environment. However, unlike the described research, we focus on incremental techniques to conduct traditional, global transformations that aim to improve the quality of compiled code in a runtime environment.
2.5 Resource-constrained compilation

Much of the prior research in designing compiler passes for a time constrained environment such as a Java JIT, has focussed on creating new optimization algorithms. For instance, Yang et al. describe an allocation algorithm for extended basic blocks [98]. Chen et al. implement a non-iterative loop invariant code motion pass that propagates region constants to outside the loop [30]. Other researchers have focussed on selectively optimizing frequently executed regions of the program [91, 50, 64]. Their work highlights a key constraint in runtime compilation: if a program unit is optimized, then its expected execution time must be carefully considered in deciding on the degree of optimization that should be conducted.

In recent years, some research has been conducted into reducing memory requirements for compiler optimizations. The popularity of mobile devices that can run portable code, such as Java, has ignited interest in designing memory efficient runtime compilers. Pliss and Mathiske describe a JVM strategy for evicting previously compiled code based on a low-overhead profiling technique [75]. They designed their technique specifically for memory constrained environments such as mobile phones where cache size can be severely limited. Zhang and Krintz study compile-only virtual machines and examine which compiled methods the virtual machine should discard in a memory-constrained environment [100]. They also evaluate strategies that attempt to determine the optimal time to unload a method. Some runtime compilers attempt to reduce code size by judiciously invoking the JIT – since bytecode is a compact representation, the compiler can preserve storage. However, interpreting the bytecode is often expensive and therefore may not be suitable for many environments. Chen et al. tackled the memory limitation problem by designing a garbage collector that reduces the dynamic memory consumption of a program by lazily allocating and compressing objects [29]. McDowell and colleagues implemented the Java virtual machine on a specially designed operating system and reduced its resident memory consumption for operation on embedded processors [65].
Typically, the first generation of runtime compilation platforms for mobile devices did not contain a JIT – code was executed strictly by interpretation. However, recent advances in storage and computational capacity have led to an interest in implementing JITs for embedded processors. An important infrastructural advance in this area is the release of Sun’s CLDC HotSpot framework that contains a runtime compiler for mobile devices [69]. Sun’s white-paper advocates the use of JIT compilation to increase application efficiency thereby reducing the device’s power consumption.

2.6 Loop unrolling

In contrast to the resource constraints imposed on a runtime compiler described in the previous sections, a JIT can exploit unique opportunities for optimization that arise due to its proximity to the program execution environment. Our research will describe the development of one such optimization – loop unrolling – for a runtime compiler. Therefore, in this section, we shall explore prior research conducted in loop unrolling.

Loop unrolling is a classic optimization that has been described in early compiler literature [3]. Unrolling eliminates branches from the instruction stream thus allowing the instruction scheduler \(^3\) to be more aggressive in rearranging instructions. The technique can also enhance the efficacy of other optimizations. For instance, unrolling enables loop fusion and the elimination of register-to-register copies emitted by scalar replacement [54]. However, excessive unrolling of a loop can adversely affect program performance. When a loop is unrolled, register pressure in the loop generally increases. Greater register demand can lead to more spills being inserted in the loop thereby reducing the performance of the code. Further, since instructions are duplicated in the additional copies of the loop created by unrolling, the transformation can have an adverse affect on instruction cache performance [96].

\(^3\)Unrolling helps both hardware as well as software instruction schedulers
The number of times the loop is unrolled, also known as the *unroll factor*, is critically important. As a result, many researchers have focussed on mechanisms to automatically determine good unroll factors [25, 24, 55]. Sarkar presents an analytic model to determine the best unroll factor for a set of nested loops based on an objective function that computes the cost incurred for every choice of the unroll vector. Sarkar’s technique also inserted fewer cleanup loops than an unroll-and-jam algorithm thereby potentially increasing application performance. The effectiveness of the algorithm was tested on SPEC 95 floating point benchmarks and achieved a 8% speedup over the original program [83]. Stephenson and Amarasinghe used machine learning techniques to predict the optimal unroll factor [87].

Unrolling loops is also important in the context of runtime compilation. Vaswani and colleagues used profiling data to guide loop unrolling for frequently executed loops in the .NET runtime compiler [93]. The HotSpot Java server compiler conducts unrolling to increase code efficiency and enable auxiliary optimizations [68]. Muth et al. use value profiling techniques to monitor values stored in register locations. The distribution of those values are utilized in a cost-benefit analysis to conduct specialization and other optimizations including loop unrolling [72]. Our runtime unrolling approach differs from Muth et al., in that it does not continuously monitor program variables. Instead, it uses a very lightweight mechanism to examine program values at runtime and uses those values to unroll loops in JIT. Further, our technique also examines memory locations in addition to register values potentially leading to greater opportunities for improvements. The DyC compiler allowed programmers to add annotations to C programs that were used by the compilation system to optimize the program at runtime [46]. DyC specialized the program by using runtime information about program values. In addition to several other optimizations, it conducted a *complete loop unrolling* transformation by using runtime values of loop iteration counts. While the DyC approach is related to our strategy described in Chapter 6, it involves additional user input – in particular, a programmer must modify the ap-
plication source code. Therefore, unlike our runtime loop unrolling technique, DyC is not completely transparent to the end-user. Further, in contrast to our unroller, it focuses on fully unrolling the loop which may not be the most desirable unroll factor for the transformed loop.

2.7 JIT annotations

To reduce dynamic compilation cost, some research has focussed on exploiting the two-phased bytecode compilation process. Researchers hope to increase the efficiency of the runtime compiler by shifting some of the burden of JIT optimization to the bytecode compiler. To this end, some infrastructures allow the bytecode compiler to transmit information to the JIT via annotations. Annotations are descriptive data attached to the bytecode and are typically used to convey source-level information to the runtime compiler. For example, Java's annotations, specified in the J2SE standard, and Microsoft's .NET attributes allow meta-data to be transported from the bytecode compiler to the JIT [67, 63]. This additional information can be used effectively at runtime to reduce JIT compilation time or enable more complicated optimizations. Reig advocates the use of annotations to improve code-generation [80]. Krintz and Calder recommend using annotations to reduce startup time and assist in various optimizations [58]. They implemented annotation-aware optimizations on the Intel Open Runtime Platform and transmitted hints to the backend optimizer to improve the efficiency and quality of register allocation, flow-graph generation, inlining, and selective optimization. Azevedo et al. also argue that an annotation-based system can be beneficial to a JIT [10]. Their system computes ranks for virtual registers at bytecode compilation time and uses that information at runtime to guide register-allocation. Ronne and colleagues propose an extension of the SafeTSA representation [6] to convey virtual register ranks and kill sets to the register allocator [94]. The authors also demonstrate that their representation can assist escape analysis. Haldar proposed a mechanism to verify the integrity of data-flow analyses passed as annotations [47].
Qian et al. annotated the bytecode with an analysis of array references [78]. The JIT utilized these annotations to improve the code by eliminating bounds checks for certain array accesses.

Our research shares common threads with much of the prior work described above. However, our approach is more aggressive in reducing the computational burden of a global transformation while maintaining its algorithmic superiority over weaker techniques. Prior work in the field has focussed on drastically weakening global optimizations and providing hints and limited pre-computed information via annotations to the backend compiler. In contrast, we will redesign a strong, global register allocation algorithm with an emphasis on reducing compilation time while preserving its proven advantages in optimization quality over compile-time efficient, but weaker algorithms. We expect our techniques to be used in conjunction with selective optimization strategies described in Section 2.5. Our work will lower the threshold that allows such a selective runtime technique to choose strong optimization algorithms. Further, we shall explore strategies to incorporate another traditional transformation – loop unrolling – in a runtime environment. By unrolling loops at runtime, in contrast to conducting the optimization at compile-time, we wish to exploit information on program values that is available only during program execution.
Chapter 3

Examining Strong Register Allocator Algorithms

3.1 Introduction

Register allocation is a critical transformation conducted by a compiler. The allocator maps values in the input program to a limited set of machine registers. Register allocators typically take a representation of a program (typically in the form of assembly code or an intermediate format after instruction selection) as input. This representation does not impose any architectural limitations on the number of registers – values are contained in locations known as virtual registers. It is the allocator’s responsibility to map the theoretically unlimited virtual registers into a finite number of machine (or physical) registers. Moreover, while conducting this mapping, it needs to maintain the semantics of the program. Graph coloring register allocators construct an interference graph that represents the safety constraints needed to preserve the program’s meaning. Program values are represented by nodes in the interference graph and edges between nodes imply that those values cannot share a physical register because they are simultaneously live. Values that cannot share a physical register are said to interfere with each other. In our preliminary explorations, we have evaluated two computationally intensive register allocation algorithms: Chaitin-Briggs and Callahan-Koblenz. We intended to evaluate these algorithms for their suitability for runtime compilation.

Both the Chaitin-Briggs and Callahan-Koblenz allocators construct such an interference graph for each procedure in the program and then attempt to color it.

---

1This is collaborative work with Jason Eckhardt. He implemented the Callahan-Koblenz allocator for LLVM.
However, the two graph coloring algorithms use significantly different techniques to construct and color their interference graphs and to spill registers. To understand and highlight the impact of these differences in allocation decisions, we present a summary of the algorithms in the next two sections.

3.2 Overview of the allocator algorithms

3.2.1 Chaitin-Briggs allocator

As the name suggests, the Chaitin-Briggs allocator is based on Chaitin’s classical graph coloring allocator. In describing their algorithm, Briggs et al. identify several major phases in their allocator. Our implementation faithfully follows the implementation described in the paper except we do not need to mark and number live ranges (Briggs et. al call this the “Renumber” phase) since this information is already available in the static single assignment form (SSA) based representation we use. The major phases, as depicted in Figure 3.1 and described in [?] are:

1. **Build the Interference Graph**: The initial phase identifies interferences by constructing live ranges and adding interferences between these ranges. This phase consists of building live sets via an iterative data-flow algorithm.

2. **Coalesce**: The allocator then removes copies between registers if the source and the destination registers do not interfere. The build and coalesce phases are
repeated in tandem until no more coalescing can be conducted. The compile-time implications of this iterative process and strategies to minimize its impact are described in Chapter 4.

3. *Calculate Spill costs and Simplify.* To make an informed spill decision, the allocator estimates the relative expense of spilling a node. These phases calculate spill costs for every node in the interference graph and then order the nodes by pushing them on a stack after removing these nodes from the graph. The Simplify phase first removes all trivially colorable nodes—i.e. nodes that have fewer neighbors than than the number of available physical registers. If it reaches the point where no such node remains in the graph, then this phase consults the spill heuristic, chooses the node with the lowest spill cost, and pushes that node onto the stack. The process is repeated until the graph is empty and all nodes have been placed on the stack.

4. *Select:* The allocator tries to color the graph by repeatedly popping a node from the stack, inserting it into the graph, and attempting to assign it a color. If all colors have already been exhausted by its neighbors, then the node is marked for spilling and left uncolored. This late placement of the coloring phase distinguishes Briggs's algorithm from Chatin's.

5. *Spill code insertion:* If any nodes were marked for spilling by the previous phase, then the graph could not be colored. As a result, spill code is inserted for those nodes and the allocator is restarted on the modified program.

The Briggs allocator marks nodes to be spilled at a later stage than Chaitin's algorithm. The authors call this procedure optimistic coloring since the algorithm defers the spilling of a node in the hope that it will become colorable.
3.2.2 Callahan-Koblenz allocator

The Chaitin-Briggs allocator suffers from one major drawback: it does not pay attention to program structure, or code shape, while allocating registers. Several attempts have been made to address this shortcoming and impart "geographical" information to the allocator. The Callahan-Koblenz allocator attempts to extend Chaitin's allocator by using a hierarchical data structure to represent the control flow characteristics of the program. This allows the allocator to be more prudent in its placement of spill code. The hierarchical allocator approaches spill-code placement in a more measured manner than Chaitin-Briggs – by analyzing program structure, it attempts to schedule spills in less frequently executed portions of the program.

While building the program representation, the Callahan-Koblenz allocator focuses on loops and conditionals since they can be helpful in identifying good locations for spill code placement. The code shape is captured by constructing a tile tree. Each tile in this tree represents a region of code such as a loop or conditional. Given any two tiles in the tree, either one must be contained by the other or the two must be completely disjoint. The hierarchical nature of the tile tree is thus especially useful to the allocator in scheduling spill code away from high-frequency code regions. In Figure 3.2, we have shown a tile-tree derived from a simple control-flow graph. In the figure, the set blocks(T) contains all basic blocks which belong to tile T, but not to any subtiles of T. The allocator uses these tiles to split live ranges – each live range is thus dissected into tile sections which allows the allocator to make finer-grained allocation and spill decisions. In particular, the allocator can decide to spill a live range in one tile while allocating it to a register in another or allocate the live range to different registers in different tiles.

Figure 3.3 depicts the overall structure of the Callahan-Koblenz allocator. The algorithm proceeds by constructing the tile-tree and then examining the tiles twice – a bottom-up traversal followed by a top-down traversal.
Figure 3.2: Example tile tree: (a) CFG; (b) tiles overlaid on CFG; (c) the tile tree.

Figure 3.3: The Callahan-Koblenz Allocator
The bottom-up traversal

During this phase, the allocator visits each tile $T$ in postorder and examines values that are both live and referenced in the tile.

1. *Interference graph construction:* For each tile, an interference graph is built. The graph is similar to a standard Chaitin-Briggs interference graph with one major exception: variables that are not referenced in the tile are not assigned a node in the graph. (We abbreviate such Live-But-Not-Referenced live ranges as “LBNR”). In contrast to the biased coloring technique of Chaitin-Briggs that attempts to assign the same color to the source and destination register of a register-to-register copy, this allocator uses a different strategy. When a copy instruction is encountered, the allocator assigns preferences to each live range.

2. *Incorporate subtile summaries:* Since the algorithm makes a post-order walk through the tile tree, before a tile is processed, all of its subtiles have already been encountered. The allocator adds the allocation decisions and preferences made at each subtile into the parent tile’s interference graph.

3. *Color:* Next, the allocator attempts to color nodes in the graph while trying to respect color-preferences. If a preference is satisfied, it may be propagated to other nodes. Note that the top-down phase can alter the colors assigned to a node by this phase. Therefore, the color assigned to a node is called a “pseudo-color” (there are $k$ pseudo-colors, corresponding to the $k$ physical registers).

4. *Summarize:* After a tile $T$ is processed, the allocation decisions taken at a subtile is conveyed to its parent tile. In particular, the information passed contains all tile-global variables allocated to registers, all tile-globals allocated to memory, and *tile summary variables* (TSV). Each TSV corresponds to a set of tile-local variables that were allocated the same color, so that the local allocation is represented in a very compact form. Conflicts involving tile-locals are stored in terms of their associated TSVs.
The top-down traversal

During the top-down traversal, the algorithm visits each tile in pre-order. This phase uses the local information gathered during the bottom-up phase to make final allocation decisions. Spill code is introduced at tile boundaries to reconcile differences in each tile's allocation.

1. **Rebuild**: Reconstruct the interference graph for T directly from its summary information.

2. **Incorporate parent summaries**: Conflicts for LBNRs that were excluded in the first phase are now added to the graph for consideration, if they received a register in the parent. Certain preferences are also setup based on the parent's allocation.

3. **Color**: A final coloring is performed, binding pseudo-colors to physical registers. As before, coloring decisions are influenced by any preferences.

4. **Summarize**: Save T's allocation and preference information to be passed down to its subtiles.

5. **Spill code**: Spill code is introduced at the tile boundaries, which may not be the same tile where a particular spill decision was made. Spill instructions could be loads, stores, or register-to-register copies, depending on the potentially differing allocations and assignments of a live range in T and its parent.

### 3.3 Our experimental methodology

All the experiments described in this document used a common framework. We chose LLVM as our compiler infrastructure since it was modular, flexible, and very well documented [60]. It suited our needs especially well since LLVM provides two types of compilers: a dynamic, JIT-driven compiler as well as a static compiler. We
compiled and evaluated our benchmarks on an Intel Pentium 4, 3.2GHz processor with 1 GB of memory running Linux. Throughout this document, reported times are the sum of the system and user times consumed by a process.

### 3.4 Experiments and evaluation

In assessing these allocators, our goal was to examine how strong algorithms could be adapted to a runtime environment. We wished to compare the results of the both algorithms to a JIT-style, speedy compile-time algorithm. Thus, we ran three allocators on each benchmark we tested: the Chaitin-Briggs allocator, the Callahan-Koblenz allocator, and the linear-scan allocator provided along with LLVM. Our results are summarized in Table 3.1 and Figures 3.4 and 3.5.

As can be seen from the results, both graph-coloring allocators produced better code than the linear scan algorithm. The allocation generated by Chaitin-Briggs performed around 5.98% faster than that produced by linear scan. We observed that Callahan-Koblenz was successful in scheduling loads and stores to less-frequently executed regions of the program. Overall, Briggs-allocated code ran 5.5% faster than the linear-scan allocator while the hierarchical allocator registered an improvement of 10.6% over linear-scan.

#### 3.4.1 Compilation time and runtime

From our experiments, we concluded that the Callahan-Koblenz allocator is an improvement over the original Chaitin-Briggs allocator. In comparison to Chaitin-Briggs, however, the allocator maintains a greater amount of bookkeeping information. This overhead is reflected in its allocation time. As shown in Figure 3.4, our implementation of the Callahan-Koblenz allocator is significantly slower than our Chaitin-Briggs allocator – on average, it allocates code 222% slower while the Chaitin-Briggs is around 33% slower than linear-scan. A part of this inefficiency can be attributed to our inexperience with this allocator, especially since there is no de-
<table>
<thead>
<tr>
<th>Test</th>
<th>Compilation</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CB</td>
<td>CK</td>
</tr>
<tr>
<td>gzip</td>
<td>2.49</td>
<td>9.61</td>
</tr>
<tr>
<td>mcf</td>
<td>1.29</td>
<td>5.38</td>
</tr>
<tr>
<td>crafty</td>
<td>2.02</td>
<td>12.81</td>
</tr>
<tr>
<td>parser</td>
<td>1.56</td>
<td>5.84</td>
</tr>
<tr>
<td>eon</td>
<td>1.55</td>
<td>3.49</td>
</tr>
<tr>
<td>gap</td>
<td>1.17</td>
<td>2.08</td>
</tr>
<tr>
<td>bzip2</td>
<td>3.34</td>
<td>13.84</td>
</tr>
<tr>
<td>twolf</td>
<td>1.53</td>
<td>4.36</td>
</tr>
<tr>
<td>epic</td>
<td>3.34</td>
<td>11.87</td>
</tr>
<tr>
<td>jpg</td>
<td>1.95</td>
<td>5.26</td>
</tr>
<tr>
<td>TOTAL</td>
<td>1.33</td>
<td>3.22</td>
</tr>
<tr>
<td>GEOM. MEAN</td>
<td>1.90</td>
<td>6.34</td>
</tr>
</tbody>
</table>

Table 3.1: Compilation and Execution Times for the Chaitin-Briggs(CB) and Callahan-Koblenz(CK) allocator. Reported times are relative to the Linear-scan allocator.

detailed design document for this allocator in compiler literature. We had to consult the original authors on many points. But, aside from efficiency losses due to implementation inexperience, the hierarchical allocator also conducts many more computations and transformations than Chaitin-Briggs: in addition to the work done in constructing the interference graph and coloring, it builds the tile tree, iterates over the tree twice, and maintains a host of bookkeeping information. These extra mechanisms all contribute to an increase in compile-time.

The Chaitin-Briggs algorithm, though considerably slower than the linear-scan algorithm is simpler in design than Callahan-Koblenz. Therefore, we concluded that the Chaitin-Briggs allocator lends itself more towards efficiency improvements than
Figure 3.4: Compilation time for the register allocators on the SPEC integer, epic, and jpg benchmarks

Figure 3.5: Execution time for register allocated SPEC integer, epic, and jpg benchmarks
Callahan-Koblenz. These experiments confirmed that though the Callahan-Koblenz allocator can produce better code than Chaitin-Briggs, the 5% decrease in program runtime would be negated by the considerable increase in compile time. We contend that we can decrease the time taken by the Chaitin-Briggs allocator considerably thereby greatly enhancing the utility of the algorithm on a runtime compiler.

3.4.2 Conclusions

Our experience with these allocators provided us with several insights into choosing a computationally intensive, global algorithm to implement in a runtime allocator. First, we realized that in a runtime compiler, the utility of spending additional compile time to improve code must be compensated by the code improvement made by the optimization. That is, the runtime gains of the code must at least equal the extra compilation time. Using this criteria, it is difficult to justify including Callahan-Koblenz in a runtime compiler. Second, we realized that to reduce compilation time drastically, we must not only revamp the strong algorithms but must also be prepared to surrender some optimization potential to purchase compilation efficiency. Lastly, the experiments confirmed that our focus on register allocation was well-placed. We were encouraged by the improvement in runtime of the graph-coloring algorithms over a simpler strategy. The results indicated that register allocation gave us ample opportunities to improve the program at runtime. These lessons served us well — it provided us with the motivation to design and evaluate JIT strategies for traditional optimizations. We also used our understanding of the two allocators to guide our design goals for algorithms described in the next chapter.

\footnote{We would like to note that additions to the base Chaitin-Briggs algorithm such as rematerialization [17], interference region spilling [13], and passive splitting [37] may well eliminate the difference in performance between the two allocators.}
Tailoring Graph-coloring Allocation for Runtime Compilers

Our observations in the previous chapter led us to carefully consider strategies that reduce compilation time for global register allocators. We realized that global register allocation can considerably decrease the runtime of the allocated code. However, as traditionally implemented, the additional compile time required for a global allocator would greatly diminish the runtime gains achieved by an improved register allocation. Moreover, our experiments in Chapter 3 indicated that Chaitin-Briggs was more attractive than Callahan-Koblenz to serve as a register allocator for a runtime compiler. Consequently, we focussed on redesigning the base Briggs algorithm to increase its efficiency.

4.1 Identifying the expensive components of the algorithm

As described in Section 3.2.1, while the Chaitin-Briggs algorithm passes through 6 major phases while allocating registers. In order to effectively redesign the algorithm for a JIT, we wished to profile the algorithm to understand the time-consumption of the different phases. For most programs, the interference graph builder is the most expensive component – its worst-case asymptotic bound is $O(n^2)$, where $n$ is the number of live ranges in the program. However, since experimental results can sometimes differ from those suggested by complexity analyses, we conducted a

---

1Surprisingly, Chaitin's original papers do not provide a complexity analysis. Briggs provides experimental evidence to confirm that interference graph construction is the most expensive component.
Figure 4.1: Contribution of phases in Chaitin-Briggs to the total allocation time. The values are geometric means over all SPEC integer benchmarks.

number of experiments to measure the relative performance of the 6 phases. Figure 4.1 presents the results of those experiments. We timed the running time of the different components and computed the percentage of time spent in each phase. As displayed in table, our results show that, as predicted by the complexity analysis, the interference graph builder is the most expensive component. On average, it consumes about 72% of the allocation process. Therefore, from these experiments, we conclude that reducing the time requirements of the builder should increase allocation efficiency. Note that the building phase might be revisited several times by the algorithm. In particular, if spilling or coalescing occurs, the interference graph is rebuilt. This observation is crucial in understanding the contribution of the building phase towards the total expense of register allocation. In the next few sections, we shall describe how we redesigned the interference construction algorithm to effectively increase the allocator's efficiency.
4.2 Interference graph building

The Chaitin-Briggs algorithm models allocation as a graph-coloring problem. It first builds an interference graph that denotes the safety constraints that the allocator must respect. These constraints, essential for maintaining the semantics of the program, are called interferences. The interference graph is an undirected graph that consists of nodes and edges. Nodes represent the live ranges in the program. An edge between two nodes indicates that the two corresponding live ranges interfere with each other. Chaitin in [28] defines the term interference as: *two names interfere if one of them is live at a definition point of the other*\(^2\). If two live ranges interfere, then they cannot share the same physical register. Thus, the register allocator must preserve safety by ensuring that the live ranges are allocated different registers. Figure 4.2 depicts an interference graph constructed from a simple procedure. We shall revisit the structure of the interference graph while devising alternate strategies for graph-coloring register allocation. To fully understand the impact of our changes to the allocator, it is important to examine the base register allocation algorithm — specifically the interference graph construction mechanism. As we have already discussed, graph construction is the most expensive component of the allocator and we wish to reduce its cost. We will therefore outline this algorithm in the next section.

4.2.1 Algorithm for identifying interferences

When a program is provided as input to the allocator, it invokes the interference graph builder. The builder proceeds by identifying interferences between live ranges for each procedure and then constructing the graph to reflect the interferences. Figure 4.3 contains the pseudo-code of the construction algorithm. Recall that two live ranges interfere if one is live at the other's definition point. First, the algorithm calculates live set information by using a classic, worklist-driven data-flow algorithm. This

\(^2\) Chaitin’s "names" are derived from connected components of a virtual register’s def-use chains and are analogous to live ranges
Procedure P:

c = ...  
b = ...  
a = 0  
e = load integer from memory  
d = b + c  
f = e + 10  
g = f * 50 + a  
h = d + g  
return h

Figure 4.2: An interference graph constructed from a simple procedure

analysis annotates each block in the procedure with the set of live-ranges that are live at the block's beginning and at its end. Next, the graph constructor iterates over every block in reverse; it examines the last instruction in the block and traverses backwards to the first instruction. At every instruction, the algorithm updates the set of values that are live at that point in the program incrementally from the previously known set. Live ranges that are defined in that instruction are deleted from the live set and live ranges that are used in the instruction are added to the set. To build the actual graph, the process adds interferences between a value defined in the current operation and all values live at the definition.

4.2.2 Rebuilding the interference graph

After the interference graph is built, the graph-coloring allocator proceeds as shown in Figure 3.1. Note that during the allocation process, the interference graph information might become outdated. As a result, the allocator must rebuild the interference graph. Rebuilding occurs due to two reasons: spill code insertion and coalescing. When spill code is added to the program, new live ranges are created and the spilled live
Build_Live_Sets(Procedure P)

Use an worklist algorithm to compute live-in and live-out sets for each block in P

Build_Interference_Graph(Procedure P)

call Build_Live_Sets(P);

for every block B in P

  Initialize set Current_Live to the LiveOut set of block B

  for every instruction I in the block in reverse order

    for every definition D in I

      add an interference from D to every element

      in Current_Live - {D} creating nodes if necessary

    for every definition D in I

      remove D from Current_Live

    for every use U in I

      add U to Current_Live

Figure 4.3 : Algorithm for constructing the interference graph
ranges are deleted. The allocator needs updated interference information for these ranges before it can safely allocate physical registers to them. Note that deleting the original version of the spilled live ranges is critical for the spill to have its effect, namely lowering the degree of its neighbors. Therefore, at the end of the spill phase, the algorithm rebuilds the interference graph, re-running the algorithm in Figure 4.3. Coalescing can also make the interference graph imprecise. Briggs describes this issue comprehensively in his dissertation [16]: when two values are coalesced, the allocator constructs an approximate set of interferences for the combined live range. This approximation, though safe, may be overly conservative. Rebuilding the interference graph after coalescing corrects this problem. Moreover, both coalescing and spilling can invalidate the live sets. Thus, the interference graph builder must rebuild live sets before rebuilding the graph.

4.3 Redesigning the interference graph builder for a runtime compiler

Our analyses and experiments led us to focus on increasing the efficiency of the interference graph construction process. To this end, we decided to design two allocators. These allocators correspond to the two strategies – lossy and lossless – that we discussed in Section 1.1.1. The distinction between the two allocators is important: the lossless allocator will preserve the precision of the interference graph while decreasing the time taken to update the graph. In comparison, the lossy allocator will introduce some imprecision in the interference graph. However, this will allow the allocator to significantly reduce allocation time.

4.3.1 Spill code insertion

Before we examine the modified allocators, consider the spill insertion mechanism in a Chaitin-Briggs allocator. Since the modified allocators update the interference graph after spilling, our examination of spill mechanisms will introduce the terminology that
we use in describing the lossless and lossy allocators. Figure 4.4 shows the insertion of instructions for spilled live ranges. The leftmost panel in the diagram depicts the code before spilling. After a spill load or store is inserted, the original live range is essentially split up into multiple live ranges. The spiller creates an uniquely named temporary register for each of these smaller live ranges which are then used to rename references to the spilled ranges. \( T_1 \) and \( T_2 \) are the temporary registers in the figure. Next, the spilling algorithm differentiates between uses and definitions. The middle panel shows the changes made to the code if a use register \( R_1 \) in instruction I is spilled. Since \( R_1 \) is used, but not defined in the instruction, a load to the temporary register \( T_1 \) is inserted before the instruction. \( R_2 \), on the other hand, is defined by the instruction. Therefore, a store to the temporary \( T_2 \) is inserted after the definition. In the descriptions of our allocators, we shall use the term temporary register to refer to the newly created live ranges (\( T_1 \) and \( T_2 \) in our examples). For consistency, we shall use \( R \) to refer to a spilled register and \( T \) to refer to temporary registers.

### 4.3.2 The lossless (precise) allocator

Recall that while allocating registers, if a live range is spilled, it is removed from the program. Moreover, several new live ranges are introduced in spill instructions. Consequently, the graph does not have information about the newly created values and it also contains spurious interferences from the spilled node. Therefore the graph is rebuilt after spilling. We postulated that we may be able to reduce allocation time by not rebuilding after spilling. Instead, we will incrementally update the interference
; The procedure has been renamed and spill locations have been
determined
for every block B in the procedure
  Initialize Current_Live to the LiveOut set of block B
  iterate backwards through every instruction I in B
    Last_Live = Current_Live
    for every definition D in I
      remove D from Current_Live
    for every use U in I
      add U to Current_Live
    if a load must be inserted for temporary register T
      add an interference between T and every member of Current_Live
    if the load services multiple instructions
      add an interference between T and all live ranges defined
        between I and the last use of T
    update infinite spill cost registers
  if a store must be inserted for temporary register T
    add interferences between T and every member of Last_Live
  update infinite spill cost registers
  remove nodes corresponding to all spilled ranges from the graph

Figure 4.5: Lossless algorithm for reconstructing the interference graph after spilling

To reconcile the graph with the modified program,
To incrementally update the graph, the lossless allocator uses a strategy similar
to the algorithm outlined in Figure 4.3. The spiller in this allocator, just like the
spiller in the standard Chaitin-Briggs algorithm, iterates through every basic block
searching for a spilled register. However, we made two important changes to the spill mechanism. First, it iterates backwards through each block looking for a spilled value. Second, while iterating backwards, it maintains the set of values live at the current instruction. It removes a value to the live-set on encountering a definition and inserts a value if it encounters a use. The currently live set is initialized to the live-out set before iteration begins. Recall that we are concerned with updating the interference graph on spilling a register. As described in the previous section, when a value is spilled, the spiller replaces the use or the definition with a newly created temporary register. The problem we now face is that the temporary register is not represented in the graph. To remedy this, our incremental allocator adds an interference between the temporary register and all values that are live at that point in the basic block. For both loads and stores, the live range of the temporary register spans the original instruction. Thus, the spiller must be careful to add values that are used (and therefore live for preceding points in the block) in that instruction. Figure 4.5 outlines the pseudo-code for the modified spiller.

The modified algorithm we described avoids the need to call the rebuild process if a value is spilled. After the incremental updates, the interference graph remains updated and is ready for another iteration of the allocation process. While we have described the crux of the changes required to make the allocation process incremental, the lossless allocator encounters a few other obstacles that must be overcome. These issues are common to both our lossless and lossy allocators and we have discussed the approaches of both allocators in Section 4.3.5.

4.3.3 The lossy (imprecise) allocator

Our success with the lossless allocator, as shown in Section 4.4, indicated that incremental methods can improve the runtime of register allocator. However, we wished to make the allocator more competitive for JIT usage. Therefore, we were willing to trade-off some degree of allocation proficiency to reduce the running time of the
Algorithm. To decrease allocation time, we decided to augment the representation of the interference graph. The unmodified interference graph contains two major data structures—a symmetrical bit matrix and a collection of edge sets. The bit matrix indicates whether two nodes in the graph interfere. Each node in the graph, \( N \), holds an edge-set that lists the nodes that \( N \) interferes with. We added additional information to the edge-sets—each edge contains a tag indicating the type of the edge. We classified each edge in an interference graph as a definition edge or use edge.

To comprehensively define these terms, let us reconsider the algorithm described in Figure 4.3. The procedure for building an interference graph traverses the program, identifies live ranges, and adds interferences between them. A careful examination of the algorithm yields that there exist three distinct scenarios when an interference edge is added. If the algorithm added an edge between live range \( L_1 \) and \( L_2 \), then either:

1. The algorithm discovered that \( L_2 \) is live at a definition point of \( L_1 \), in which case the edge \( < L_1, L_2 > \) gets classified as a definition edge, or
2. The algorithm discovered that \( L_1 \) is live at a definition point of \( L_2 \), in which case the edge \( < L_2, L_1 > \) gets classified as a definition edge, or

3. The algorithm discovered that both cases 1 and 2 occurred – \( L_1 \) was live at \( L_2 \)'s definition point and \( L_2 \) was live at \( L_1 \)'s definition point. In this case, both edges are classified as definition edges.

A use edge is an edge that has not been classified as a definition edge. The algorithms described in this document focus solely on definition edges. In our initial construction of the interference graph, we used these specifications to categorize the edges. Consider the program that was displayed in Figure 4.2. Figure 4.6 shows the interference graph for the program with edges partitioned into the two categories. This modification is similar to transforming the undirected interference graph into a constrained directed graph.\(^3\) This seemingly minor distinction between edges holds the key to a faster algorithm. In the next few paragraphs, we shall use tuple notation to denote edges only when a distinction between use and definition edges is warranted.

**Updating the graph after spilling**

Once the enhanced interference graph has been constructed, the lossy allocator utilizes the additional information embedded in the graph to guide post-spill incremental updates of the graph. Let a live range \( R \) be spilled at an instruction \( I \) in block \( B \) of the program, and a new temporary register \( T \) created in its place. The allocator must, as before, compute the interference edges for \( T \). Prior to inserting the spill instruction, our algorithm iterates backwards through the block and locates the previous definition point. Let \( D \) be the live range defined at that location. The allocator adds all of \( D \)'s definition edges as interference edges for \( T \). Further, it classifies these edges as

\(^3\)Specifically, the directed graph, \( DG \), must maintain the one-one mapping between itself and an undirected graph – if \( < n_1, n_2 > \in DG \Rightarrow < n_2, n_1 > \in DG \). This is similar in structure to Cooper and Simpson’s containment graph [37] but encodes very different semantics.
; The procedure has been renamed and spill locations have been
determined
for every block B in the procedure
iterate through every instruction I in B
if a load must be inserted for temporary register T
locate the last definition in the block prior to I
if such a definition D is found
add the edge (T, D) to the graph
if D is a temporary register, set D to the register name before renami
for every definition edge <D, E> in the interference graph
add the edge (T, E) to the graph
if D is a copy instruction, add an edge between the source and T.
else if no such definition exists
for every value L in the LiveIn set of block B
add edge (T, L) to the graph
if a store must be inserted for temporary register T
T is defined in instruction I. Let D be the register name of T before
renaming
for every definition edge <D, E> in the interference graph
add the edge (T, E) to the graph
mark all edges originating from T as definition edges
if the spill load services multiple instructions
add interferences between T and all definitions till the last use of T
remove nodes corresponding to all spilled ranges from the graph

Figure 4.7: Lossy algorithm for reconstructing the interference graph after spilling
definition edges for \( T \). If a definition point is not found before the beginning of the block is encountered, then the algorithm adds all members of the live-in set of \( B \) as definition edges for \( T \). This procedure results in generating a safe but conservative estimate for the interference edges of node \( T \). Figure 4.7 outlines the lossy spill algorithm.

In the next few sections, we shall prove two lemmas that confirm that our updates are safe for spill loads and stores that service one instruction. Lemma 1 proves that given a safe interference graph, if a definition point is found in the block while iterating backwards from \( P \), then the addition of these edges to \( T \) is safe. Lemma 2 proves that if a definition point is not encountered before the beginning of the block and all edges from the live-in set is added to \( T \), then the interference graph is safe after the update. Since the updates occur before register assignment, the registers referred to in the lemmas are virtual registers.

**Preliminary definition:** An interference graph for a procedure \( P \) is considered safe iff the following condition holds:

Given any two live ranges \( L_1 \) and \( L_2 \) in \( P \), \( L_1 \) and \( L_2 \) interfere \( \Rightarrow \exists \) edge (\( L_1, L_2 \)) in the interference graph

**Invariant 1:** For a node \( N \) in the interference graph, the set of definition edges contains the nodes of all values live at \( N \)'s definition points. Note that due to the classification of edges described in Section 4.3.3, this proposition is true after the initial construction of the interference graph.

**Lemma 1**

*Given:*

- a spilled register \( R \) renamed to \( T \) at instruction \( I \) in basic block \( B \)
- \( I \) contains the only use of \( T \) before spill code insertion
- a safe interference graph before processing the spill
- \( D \) is the first register definition in \( B \) found by iterating backwards from \( I \)
If the edge \((T, D)\) and all of \(D\)'s definition edges are added to \(T\), the resulting node for \(T\) contains all interferences needed to ensure register allocation safety. Further if all added edges are marked as definition edges originating from \(T\), Invariant 1 is preserved by this update.

**Proof:** We shall prove this by contradiction. Let us first consider what it means for the update not to be safe. The update is unsafe iff after the edges are added, the graph does not contain an edge between \(T\) and a live range \(L\) even though \(L\) and \(T\) interfere. This follows from the definition of safety in an interference graph. Let \((T, L)\) be such an edge that is not added to \(T\) after the update.

First, note that \(L\) cannot be equal to \(D\) since we know that \((T, D)\) is added to the graph.

Further, if \(L\) and \(T\) interfere, it implies that \(L\) was live at the definition point of \(T\). Recall that if a spill store is being inserted, then the definition point of \(T\) is \(I\). If, however, a spill load is inserted, a new instruction defining \(T\) will be inserted just before \(I\). In both cases, remember that our algorithm updates interferences before inserting the spill instruction.

Since the algorithm adds all the definition edges of \(D\) to \(T\), our assumption about the absence of \((T, L)\) in the updated graph implies that \(< D, L >\) cannot be definition edge in the graph before processing the spill. Our graph was safe before processing the spill. Thus, if \(< D, L >\) was not a definition edge before the spill, and since Invariant 1 holds before the update, this implies that \(L\) was not live at the definition point of \(D\). If \(L\) interferes with \(T\), however, and it is not live at the definition point of \(D\), then \(L\) must have been killed (i.e. defined) in block \(B\) between \(T\)'s and \(D\)'s definition points. But, we know that \(D\) is the first definition of a register in block \(B\) while iterating backwards from \(I\) and that \(L\) is not \(D\). Thus, we reach a contradiction. It also follows that since Invariant 1 holds before the update and that no registers are killed between the definition points of \(T\) and \(D\), marking the added edges as definition edges for \(T\) preserves the invariant.
Lemma 2

Given:

- a spilled register $R$ renamed to $T$ at instruction $I$ in basic block $B$
- $I$ contains the only use of $T$ before spill insertion
- a safe interference graph before processing the spill
- there are no register definitions from the beginning of $B$ till $I$

If all edges in $\text{live} \cap \text{in}(B)$ is added to $T$, then the resulting node for $T$ contains all interferences needed to ensure register allocation safety. Further, Invariant 1 is preserved by this update.

Proof: Again, as in Lemma 1, the update is unsafe iff after the edges are added, the graph does not contain an edge between $T$ and a live range $L$ even though $L$ and $T$ interfere. Now, if $L$ interferes with $T$ and there are no definitions between instruction $I$ and the beginning of the block, then $L$ must be in the live-in set of block $B$. Thus, adding all members of the live-in set of $B$ will ensure the presence of an edge between $T$ and all ranges that interfere with $T$. Note that since there are no values killed between the beginning of $B$ and $I$, marking all values in $\text{live-in}(B)$ as definition edges for $T$ will preserve Invariant 1.

Lemma 1 and Lemma 2 prove that our updates are safe for spills that service one instruction. We, however, need to consider the situation when a load services multiple instructions.

Spill loads servicing multiple instructions

If the allocator decides to spill a live range, it must iterate through the code and insert spill code – loads before a use and stores after a definition. \(^4\) The placement of

\(^4\)We use Harvey’s suggested modification to Briggs’ algorithm [48] but the same updates can be conducted on Briggs’ original algorithm.
stores is simple – a store is inserted after every definition of a spilled range. However, load placement is more intricate. By inserting a load, the spiller essentially marks the beginning of a new live range. In certain situations, multiple uses of the same live range are present in separate instructions that are situated close together. In this case, the spiller tries to schedule one load for those uses. Thus, after spilling, a single load can service multiple uses of a live range. To account for this, we need to change our algorithm to incrementally update the interference graph after spilling.

On inserting a spill load that services multiple uses of a live range, the incremental algorithms iterate forwards through the block till the last use of the live range in the block and add all definitions encountered as use interferences for the newly created temporary live range. Both the lossy as well as the lossless allocator performs this update to ensure safety. Note that since this scenario occurs only if there are no deaths in between the load and the uses, the algorithms are only required to track live range definitions while iterating. For these updates, they need not be concerned about removing dead live ranges.

Lemma 3

Given:
- a spilled register $R$ and a temporary register $T$ created in its place at instruction $I$ in basic block $B$
- $T$ is used (read-from) in instruction $I$ and a spill load is placed immediately preceding instruction $I$
- $T$ is also used in instruction $J$ that appears after $I$ in block $B$

We know from Lemmas 1 and 2 that if $T$ was used solely in $I$ then after we update the interferences for $T$, the interference graph will be safe. We need to show that after the update, if we add all live range definitions between $I$ and $J$, then we will end up with a safe interference graph and Invariant 1 will be preserved.
Proof: Consider an interference caused by the occurrence of $T$ in $J$ that is not present after the update. This can only be caused by a live range that was created in between $I$ and $J$, i.e., it's definition point lies in between $I$ and $J$. Thus, if the algorithm adds all definition points in between $I$ and $J$, we will capture all interferences caused by the use of $T$ in $J$. We know from Lemmas 1 and 2 that Invariant 1 is preserved for the updates described in Section 4.3.3. In this case, there are multiple uses of $T$. However, the presence of additional uses of $T$ does not affect the values live at its definition point. Thus Invariant 1 is preserved as shown in Lemmas 1 and 2. In this lemma, we do not specify the lack of live range deaths between the uses of $T$ as a pre-condition since it affects strictly the precision, and not the safety of the updates.

The three preceding lemmas prove that our modifications of the interference graph do not compromise on the correctness of register allocation. Specifically, we have proven that after our updates, the interference graph contains at least all the edges required to preserve safety during register allocation.

4.3.4 Sources of imprecision in the lossy allocator

Though we have proven that the lossy allocator constructs safe interference graphs, it may add unnecessary edges to the graph. There are three sources of imprecision. First, remember that after spilling, the algorithm searches backwards for a definition point. It then adds all interferences for the definition (say D) as interferences for the temporary live range. However, $D$ might interfere with more registers than the temporary live range. Specifically, if the death of a live range occurs in between $D$ and the spill instruction then $D$ will interfere with that live range while the temporary live range will not. This situation arises only for spill loads since for stores, the temporary live range is defined in the instruction. $D$ can also be defined in multiple locations which may lead to the addition of extraneous interferences. However, JIT instruction selectors are designed to be fast and typically reserve new virtual registers for each
defined value. Therefore, we do not expect multiple definitions to be a major source of imprecision. In LLVM, we encounter this imprecision mainly for definitions by copy instructions that are generated to eliminate phi-nodes. Second, if the definition point is a copy instruction, then the source of the copy instruction does not interfere with the definition. However, we are conservative and add an edge between the source and T. This edge is superfluous if the source dies between the copy instruction and the spill instruction. Finally, another source of imprecision arises from adding edges between values in the live-in set and T. Again, such an edge is extraneous if the death of the live-in value occurs before the spill instruction in the block.

4.3.5 Issues common to both allocators

Updating infinite spill cost registers

While constructing the graph, the Chaitin-Briggs allocator identifies all live ranges that do not span the death of a live range and assigns them “infinite” spill cost. These ranges are never spilled since doing so will not decrease register pressure and consequently cannot make the graph more colorable. However, when a spill instruction is inserted, the spill creates a temporary live range which has a short span. As a result, a death is introduced in the instruction stream and this might cause some unspilled live ranges to no longer have infinite spill cost. These infinite-spill-cost live ranges are recomputed in the original Chaitin-Briggs allocator while rebuilding the graph. However, our modified algorithms do not reconstruct the graph after spilling. Therefore, the lossy and lossless allocators ensure that all registers that are live at the insertion point are deleted from the infinite spill cost set. The lossless allocator uses the set of currently live ranges that it maintains to erase ranges from the infinite-spill cost set. The lossy allocator uses the interferences it adds to the temporary register to delete ranges from the infinite-spill cost set. Moreover, if a spill load services multiple uses, both allocators remove all intervening definitions between the load and the last use from the infinite spill cost set.
Updating live set information after spilling

When a live range is spilled, the allocators follow the original Chaitin-Briggs algorithm – they replace the reference to the original range by a newly created, temporary register and insert the appropriate spill instructions. This invalidates live set information. In Chaitin-Briggs, live set information is reconstructed when the interference graph is rebuilt. Since the modified allocators do not rebuild the graph after spilling, they must update the live sets to reflect the post-spill changes. Thus, after a live range is spilled in block B, the modified allocators erase the live range from the live in and live out sets of every block in the procedure. Notice that the temporary live ranges are local to the block they are created in. Therefore, these values can be ignored by the live-in and live-out sets.

4.4 Experimental results

In the next few sections, we shall present the results of our experiments that evaluated the performance of the lossless and the lossy allocators.

4.4.1 Performance of the allocators in an offline compiler

Our first goal was to examine how effectively the modified allocators reduced allocation time. We measured the performance of three allocators: the Chaitin-Briggs unmodified algorithm, the lossy Chaitin-Briggs allocator, and the lossless Chaitin-Briggs allocator. In our first experiment, we statically compiled our benchmarks with the three allocators and compared the compilation times. Figure 4.8 shows the results of the experiment. As the graph indicates, both the lossless and the lossy allocator performed well – they decreased allocation time when compared to the unmodified Chaitin-Briggs allocator. On average, the lossless allocator decreased allocation time by over 11% while the lossy allocator reduced allocation time by 32%. This is a significant decrease over the original Chaitin-Briggs algorithm. Next, since the lossy
Figure 4.8: Compilation times with the unmodified, lossless, and lossy register allocators for the SPEC 2000 integer benchmarks. Displayed compilation times are relative to the unmodified Chaitin-Briggs allocator.

allocation algorithm can add superfluous edges to the interference graph, we were interested in comparing the execution times of the lossy allocator and the unmodified allocator. The results shown in Figure 4.9 confirm that the overly conservative allocation increases the runtime of the allocated program. However, we were pleased to note that the runtime only increases by around 1%. We will provide a more thorough measurement of the imprecision of the lossy allocator in the next few sections. Note that these two experiments measure the performance of the allocators in a static compilation environment. Our intention in designing these experiments was to understand the potential of the modified allocators. We were heartened by the sharp reduction of allocation time and the competitive performance of the code generated by the lossy allocator. However, there is one major difference between allocation time
Figure 4.9: Execution times for code allocated with the unmodified, lossless, and lossy register allocators for the SPEC 2000 integer benchmarks. Displayed execution times are relative to the benchmark allocated with the unmodified Chaitin-Briggs allocator.

Measurements obtained in a static and runtime compiler. In most runtime compilers, procedures are compiled on-demand. If a procedure is not invoked at runtime, then the JIT does not translate that procedure to native code. The offline compiler, in contrast, compiles all procedures oblivious to their utilization at runtime. Hence, for a particular benchmark, we expect to see the register allocator operating on fewer procedures in a dynamic compiler than suggested by these results. The measurements in Figure 4.8 thus provide an upper bound on the compile-time improvements we anticipate in a dynamic environment.

Figure 4.10 and Figure 4.11 depicts the running times of the lossy and the lossless allocator broken down into its different components. When compared to Figure 4.1,
Figure 4.10: Contribution of phases in the lossy allocator. The values are geometric means over all benchmarks.

Figure 4.11: Contribution of phases in the lossless allocator. The values are geometric means over all benchmarks.
Figure 4.12: Comparison of different phases in the three allocators. The values are geometric means over all benchmarks and are relative to the Chaitin-Briggs allocator.
the diagram demonstrates that the gains in allocation efficiency were due to a significant reduction in interference graph construction time. We also present a comparison of the running time of the phases in the three allocators in Figure 4.12. As the graph demonstrates, the lossless allocator reduced interference-graph construction time by around 19.6% over the Chaitin-Briggs allocator. In comparison, the lossy allocator provided significantly larger savings of 53.3% in interference construction time over Chaitin-Briggs. It is interesting to note that the Spill, Simplify, and Select phases of the lossless and the lossy allocators had slightly worse performance than the corresponding phases in Chaitin-Briggs. The additional time incurred was due to the book-keeping computations that these allocators perform in these phases. Further, as described in Section 4.4.2, the lossy allocator can insert extraneous edges in the graph that may lead to extra spilling. Consequently, the lossy allocator may, on occasion, insert more spill code in the program than the other two allocators. This results in the relatively greater time consumed by the Spill phase of the lossy allocator in comparison to the other allocators.

4.4.2 Analysis of the imprecision in the lossy allocator

Superfluous edges

The lossy allocator, in its bid to increase allocation efficiency, may add more edges to the interference graph than the unmodified algorithm. In our next set of experiments, we wish to measure the imprecision of the lossy allocator. We compiled the SPEC integer benchmarks using two allocation algorithms – the Chaitin-Briggs and the lossy allocators – and compared the interference graphs that were produced for every procedure in the benchmark. We tallied the number of edges that were produced by both allocators. Our results, as displayed in Table 4.1, show that the lossy allocator adds a moderate number of extra edges to the graph. On average, the allocator added around 10% superfluous edges.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CB Spills</th>
<th>Lossy Spills</th>
<th>Lossy/CB</th>
<th>CB Edges</th>
<th>Lossy Edges</th>
<th>Lossy/CB</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>1061</td>
<td>1032</td>
<td>1.028</td>
<td>18852</td>
<td>17020</td>
<td>1.108</td>
</tr>
<tr>
<td>vpr</td>
<td>6665</td>
<td>6469</td>
<td>1.030</td>
<td>103334</td>
<td>90136</td>
<td>1.147</td>
</tr>
<tr>
<td>gcc</td>
<td>61214</td>
<td>58601</td>
<td>1.045</td>
<td>953856</td>
<td>857890</td>
<td>1.111</td>
</tr>
<tr>
<td>mcf</td>
<td>378</td>
<td>378</td>
<td>1.000</td>
<td>10098</td>
<td>9056</td>
<td>1.115</td>
</tr>
<tr>
<td>crafty</td>
<td>10477</td>
<td>10084</td>
<td>1.039</td>
<td>190942</td>
<td>169614</td>
<td>1.126</td>
</tr>
<tr>
<td>parser</td>
<td>4136</td>
<td>3974</td>
<td>1.041</td>
<td>75250</td>
<td>70822</td>
<td>1.063</td>
</tr>
<tr>
<td>eon</td>
<td>17559</td>
<td>17368</td>
<td>1.011</td>
<td>218664</td>
<td>202294</td>
<td>1.081</td>
</tr>
<tr>
<td>gap</td>
<td>23620</td>
<td>23041</td>
<td>1.025</td>
<td>395012</td>
<td>370348</td>
<td>1.067</td>
</tr>
<tr>
<td>vortex</td>
<td>8583</td>
<td>8459</td>
<td>1.015</td>
<td>225194</td>
<td>217736</td>
<td>1.034</td>
</tr>
<tr>
<td>bzip2</td>
<td>1264</td>
<td>1204</td>
<td>1.050</td>
<td>16640</td>
<td>15210</td>
<td>1.094</td>
</tr>
<tr>
<td>twolf</td>
<td>11627</td>
<td>11204</td>
<td>1.038</td>
<td>183962</td>
<td>166196</td>
<td>1.107</td>
</tr>
<tr>
<td>GEOM. MEAN</td>
<td>6436.8</td>
<td>6636.0</td>
<td>1.03</td>
<td>106752.7</td>
<td>117355.6</td>
<td>1.10</td>
</tr>
</tbody>
</table>

Table 4.1: Extra spills and edges added by the lossy allocator when compared to the Chaitin-Briggs (CB) allocator.
Additional spill code

The addition of extra edges by the lossy allocator affects the colorability of the interference graph. Since some nodes have more edges than in the regular Chaitin-Briggs algorithm, those nodes might be more difficult to color. Consequently, the lossy allocator may generate more spill code than the unmodified algorithm. We measured the amount of spill code that was generated by both allocators. We present our comparison in Table 4.1. Compared to the unmodified allocator, the lossy allocator, on average, generated around 3% more spills. As Table 4.1 shows, the relative increase in edges is greater than the relative increase in spills. This occurs primarily because spilled nodes in the lossy allocator contain a higher number of extraneous edges as compared to unspilled nodes. In future work, we will exploit this feature of the lossy allocator by modifying code shape to reduce the number of extra spills.

4.4.3 Performance of the allocators in a runtime compiler

Since we designed the lossy algorithm for runtime compilation, our most important experiment measures the performance of the allocator in a dynamically compiled environment. We present a summary of our results in Figure 4.13. As can be seen, both the lossy and the lossless allocators outperformed the Chaitin-Briggs algorithm on all the benchmarks. The results demonstrate that the significant decrease in compilation time for the lossy allocator more than compensated for the marginal increase in execution time. Note that, as is the case with dynamic compilation, not every procedure in a benchmark was compiled. However, the incremental techniques reduced compilation time considerably. On average, the lossless allocator reduced observed runtime (compile time + execution time) time by 3% when compared to the Chaitin-Briggs allocator. The lossy allocator registered more impressive results — it reduced observed runtime by 8% on average.
Comparison with linear-scan

To gauge how effective the lossy allocator was in comparison to a JIT specific algorithm, we compared its performance to the linear-scan algorithm that was bundled with LLVM. The LLVM linear-scan algorithm extends [71] by removing the need to reserve spill registers and adding the ability to propagate spill code into instructions [44]. In our experiments, the lossy algorithm outperformed linear-scan on 9 of the 11 SPEC integer benchmarks. On 2 SPEC benchmarks—eon, and twolf—the allocator performed worse than linear-scan. On these benchmarks, graph-coloring required significantly more compilation time than linear-scan and thus the improved allocation could not compensate for the substantive compile-time handicap. Note that on both benchmarks, the lossy allocator performed better than Chaitin-Briggs. Second, on 3 benchmarks: vpr, gcc, and crafty the lossy allocator outperformed linear-scan which, in turn, bested the Chaitin-Briggs technique. This result emphasizes the success of redesigning a strong allocation technique for runtime compilation.
Further, we note that the input data size plays a major role in the relative performance of the allocators in a runtime environment. These results were obtained by running the SPEC benchmarks on large data sets. In the next section, we will examine the relationship between input data complexity and allocator performance.

4.4.4 The effect of input data size on runtime

The results obtained by our experiments highlight a difficult decision that JIT compiler designers must make. Optimizations that are expensive to conduct may improve the dynamically compiled code for current and future invocations. However, the additional time consumed by these optimizations may outweigh the advantages of executing strongly optimized code. In general, the longer a program continues to execute, the greater the advantages of expending additional time to optimize the code. In other words, the marginal value of an extra unit of time spent on optimizing a program increases with the execution time of the program. We wished to understand the impact of program running time on the relative performance of our allocators. Therefore, we gradually increased the input complexity of our benchmarks and measured the performance of code allocated by the linear-scan, Chaitin-Briggs, lossless, and lossy algorithms on a runtime compilation environment. Figure 4.14 shows the results of our experiments for 3 benchmarks — gzip, parser, and crafty.

The performance of the allocators exhibits an interesting progression. For smaller input sizes, the linear-scan allocator outperforms all other allocators. For these sizes, the overhead of graph-coloring register allocation swamps the benefits afforded by stronger allocation. As the input size to parser increases, procedures in the program are executed more frequently. Progressively, the more proficiently allocated code begins to recoup the extra time it ceded during optimization. We were pleased to note that the lossy allocator starts outperforming the linear-scan JIT much before

\footnote{or function if different optimization algorithms can be selected for each function of the program}
Figure 4.14: The graph shows the runtime of SPEC parser benchmark as the input size increases. Crossover points are marked by dashed vertical lines.
Figure 4.15: The graph shows the runtime of SPEC crafty benchmark as the input size increases. Crossover points are marked by dashed vertical lines.
Figure 4.16: The graph shows the runtime of SPEC gzip benchmark as the input size increases. Crossover points are marked by dashed vertical lines.
its Chaitin-Briggs counterpart. We wish to highlight three key features of these results. Consider the results from the parser benchmark shown in Figure 4.14. The benchmark parses natural language sentences. The data size, therefore, is measured in the number of lines provided as input. As can be seen from the graph, the cross-over point between linear-scan and the lossy allocator is around 2400 lines of input. In contrast, the program allocated with the Chaitin-Briggs allocator surpasses the linear-scan variant at around 4000 lines of input. Moreover, the lossless allocator consistently performs better than unmodified Chaitin-Briggs. Its cross-over point occurs at approximately 3600 lines. The performance curve demonstrates how program runtime changes the choice of the best optimization algorithm in a runtime compiler. Second, our allocation algorithms considerably reduces the switch-over point between the compile-time efficient linear-scan technique and a graph-coloring algorithm. As a result, the modified allocators can be profitably invoked even on programs that do not run for extended periods of time. Lastly, note that for all 3 benchmarks, the lossy allocator maintains its dominance over the lossless and unmodified graph-coloring allocators even for larger input sets. This indicates that the extra spill-code in the lossy allocated program did not occur in frequently executed regions of the benchmark. This is partly fortuitous (and, therefore, may not occur with all programs) and further extends the performance benefits of the lossy allocator. However, as we have discussed in the previous section, the lossy allocator inserts only a marginal amount of additional spill code. Thus, we have noticed this same progression in other benchmarks.

To summarize, we have successfully increased the suitability of the graph-coloring register allocator for a runtime compiler and made it more attractive for a runtime environment. In closing, we would like to point out that we have considered the whole program as the optimization unit for our allocation algorithms. However, since JITs compile each function on-demand, some compilers select different allocators for each function. In that case, our techniques and conclusions can just as easily be applied
to a smaller unit of code since we have considerably reduced the barrier for invoking graph-coloring register allocation.

4.5 Conclusions

Our experiments show that our redesign of the Chaitin-Briggs allocator was successful. We constructed two allocators – the lossy and lossless allocators – by modifying the base Chaitin-Briggs algorithm. The lossless allocator reduced allocation time but did not compromise on allocation quality. The lossy allocator, in contrast, sacrificed some allocation efficacy to decrease allocation time. As our experiments show, both allocators were able to increase allocation efficiency. Moreover, the lossy allocation only slightly increased application execution time leading to significant savings in observed execution time on a runtime compilation environment. These results confirm that our algorithms can occupy the middle ground between JIT optimizations and traditional optimizations designed for offline compilers.
Chapter 5

Optimization Passes on a Runtime Compiler

5.1 Introduction

In the previous chapters, we have looked at a global optimization – register allocation – for which proficient, offline algorithms exist but are deemed too expensive for runtime usage. We thus devised an algorithm for a runtime compiler that reduced the time consumption of the allocator while maintaining most of the proficiency of its offline counterpart. The approach highlighted a critical issue that plagues a JIT designer – a compiler transformation invoked at application runtime must be necessarily stingy with compilation time. In the following chapters, we focus on a contrasting feature of the runtime compilation process. While an online compiler is constrained by execution time considerations, the dynamic compilation environment also provides opportunities not available to a classical, offline compiler. In particular, a JIT compiler can use application characteristics that can be collected only at runtime. We wanted to explore the possibility of exploiting this additional information to improve application performance in a runtime compilation environment. Our approach is cataloged in the next few sections.

Since its inception, the structure of a traditional, offline compiler has remained relatively unchanged. It takes an intermediate representation of the program, invokes a series of optimization passes, and finally emits an executable or object file. Runtime compilers also tend to use this “assembly-line” arrangement of optimization passes to emit executable code. An unintended consequence of this structure is that it allows us to invoke a transformation pass in isolation and analyze its effectiveness. ¹ Since our

¹For simplicity, we do not consider interactions between passes. Inter-pass interactions can be
goal in this thesis was to tailor traditional compiler passes for runtime compilation, we wished to determine which compiler transformations were the best candidates for a JIT compiler. Further, in keeping with our intentions stated in the last paragraph, we wanted to focus on transformations that could exploit runtime information to improve code quality. To this end, we conducted experiments that measured the quality of code produced by a transformation pass.

5.2 Evaluating compiler passes

In addition to the compiler framework, LLVM contains a number of optimization passes. Table 5.2 describes the LLVM optimizations that were used in our experiments. All optimizations were global and thus operated at a procedural level. In order to determine the profitability of each compiler pass, we conducted experiments that measured the effectiveness of a pass in isolation. This set of experiments was an important step in understanding the potential each transformation. We wanted to identify a transformation that would be adapted successfully on a runtime compiler. To this end, we compiled benchmarks into LLVM bytecode without conducting any ahead-of-time optimizations. We then executed the benchmark on the LLVM JIT with a single optimization enabled and measured the observed execution time. We measured the performance of these transformations on programs on programs from the SPEC suite of benchmarks. We divided the test programs into two groups of benchmarks based on whether the programs primarily conducted floating point or integer computations. We used benchmarks from both the SPEC 95 and SPEC 2000 benchmark collections. Note that in our results section, the we report improvements on observed execution time – the sum of runtime compilation and application execution time.

extremely important as shown in [4, 97]. However, our main intention in running these experiments was to gauge the effectiveness of different passes. Unlike other studies in the literature, our primary goal was not to determine which combination of passes emit the most efficient code.
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Pass Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adce</td>
<td>Aggressive dead code elimination</td>
</tr>
<tr>
<td>gcse</td>
<td>Global common sub-expression elimination</td>
</tr>
<tr>
<td>cprop</td>
<td>Constant propagation</td>
</tr>
<tr>
<td>ivsimp</td>
<td>Induction variable simplification</td>
</tr>
<tr>
<td>dce</td>
<td>Dead code elimination</td>
</tr>
<tr>
<td>licm</td>
<td>Loop invariant code motion</td>
</tr>
<tr>
<td>icomb</td>
<td>Instruction combining</td>
</tr>
<tr>
<td>lwall</td>
<td>Lower Allocations</td>
</tr>
<tr>
<td>lunrl</td>
<td>Simple single basic-block loop unrolling</td>
</tr>
<tr>
<td>lwgc</td>
<td>Lower garbage collection intrinsics</td>
</tr>
<tr>
<td>lwce</td>
<td>Lower constant expressions</td>
</tr>
<tr>
<td>lwpk</td>
<td>Lower packed datatypes</td>
</tr>
<tr>
<td>lwin</td>
<td>Lower invoke and unwind instructions</td>
</tr>
<tr>
<td>lwsw</td>
<td>Lower switch statements</td>
</tr>
<tr>
<td>lwsl</td>
<td>Lower select statements</td>
</tr>
<tr>
<td>pre</td>
<td>Partial redundancy elimination</td>
</tr>
<tr>
<td>scra</td>
<td>Scalar replacement of aggregates</td>
</tr>
<tr>
<td>scfg</td>
<td>Simplify the CFG</td>
</tr>
<tr>
<td>sccp</td>
<td>Sparse conditional constant propagation</td>
</tr>
<tr>
<td>tre</td>
<td>Tail recursion elimination</td>
</tr>
<tr>
<td>none</td>
<td>No optimizations</td>
</tr>
</tbody>
</table>

Table 5.1: Description of optimizations used in experiments to determine effectiveness of passes
Figure 5.1: Results of running a single optimization pass on the LLVM JIT for benchmarks from the SPEC floating point suite. The results are averages over all benchmarks and are relative to the observed execution time of an unoptimized program.

5.2.1 Impact on floating point benchmarks

Figure 5.1 shows the results of our experiments for benchmarks from the SPEC floating point suite – it depicts the average performance gains obtained by enabling an optimization pass in the JIT.

The optimizations listed in Table 5.2 contain a simple loop unrolling algorithm that was packaged with LLVM. This transformation, abbreviated “lunrl” in the experiments, unrolls inner loops that consist of one basic block with compile-time constant loop bounds. As can be seen in the graph, the extremely simple loop unrolling algorithm performs admirably in comparison to other optimizations – on average, it reduces observed execution time by around 3.2%. This result was encouraging as it suggests that loop unrolling can be profitable for scientific benchmarks. Further, an
<table>
<thead>
<tr>
<th>Rank</th>
<th>Pass</th>
<th>Observed Exec. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pre</td>
<td>0.949</td>
</tr>
<tr>
<td>2</td>
<td>lunrl</td>
<td>0.962</td>
</tr>
<tr>
<td>3</td>
<td>ivsimp</td>
<td>0.972</td>
</tr>
<tr>
<td>4</td>
<td>lwsl</td>
<td>0.988</td>
</tr>
<tr>
<td>5</td>
<td>licm</td>
<td>0.989</td>
</tr>
<tr>
<td>6</td>
<td>lwce</td>
<td>0.992</td>
</tr>
<tr>
<td>7</td>
<td>lwsw</td>
<td>0.998</td>
</tr>
<tr>
<td>8</td>
<td>scfg</td>
<td>0.998</td>
</tr>
<tr>
<td>9</td>
<td>lwin</td>
<td>0.999</td>
</tr>
<tr>
<td>10</td>
<td>adce</td>
<td>0.999</td>
</tr>
<tr>
<td>11</td>
<td>dce</td>
<td>0.999</td>
</tr>
<tr>
<td>12</td>
<td>lwall</td>
<td>1.000</td>
</tr>
<tr>
<td>13</td>
<td>none</td>
<td>1.000</td>
</tr>
<tr>
<td>14</td>
<td>tre</td>
<td>1.000</td>
</tr>
<tr>
<td>15</td>
<td>lwpk</td>
<td>1.001</td>
</tr>
<tr>
<td>16</td>
<td>sccp</td>
<td>1.001</td>
</tr>
<tr>
<td>17</td>
<td>cprop</td>
<td>1.001</td>
</tr>
<tr>
<td>18</td>
<td>scra</td>
<td>1.001</td>
</tr>
<tr>
<td>19</td>
<td>lwgc</td>
<td>1.003</td>
</tr>
<tr>
<td>20</td>
<td>icomb</td>
<td>1.004</td>
</tr>
<tr>
<td>21</td>
<td>gcse</td>
<td>1.007</td>
</tr>
</tbody>
</table>

Table 5.2: Optimizations ranked in descending order of performance for benchmarks from the SPEC floating point suite. The numbers are geometric means over all benchmarks and are relative to the performance of a program with no optimization (represented by "none" in the table).
analysis of the results indicate a few important characteristics of these experiments. Note that loop transformations perform particularly well. Table 5.2 lists the transformations in descending order of their performance. In addition to loop unrolling, partial redundancy elimination (which is effective on loop structured control-flow), induction variable simplification, and loop invariant code motion improved benchmark runtimes significantly. This effect is partly due to the code shape of the tested benchmarks – scientific programs tend to conduct many of their computations via loops.

5.2.2 Impact on integer benchmarks

![Effectiveness of optimizations on a JIT](image)

Figure 5.2: Results of running a single optimization pass on the LLVM JIT for benchmarks from the SPEC integer suite. The results are averages over all benchmarks and are relative to the observed execution time of an unoptimized program.

As can be seen in Figure 5.2 and Table 5.3, the results obtained on the integer
<table>
<thead>
<tr>
<th>Rank</th>
<th>Pass</th>
<th>Observed Exec.</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>adce</td>
<td>0.936</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>icomb</td>
<td>0.982</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>gcse</td>
<td>0.983</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>lws1</td>
<td>0.993</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>pre</td>
<td>0.993</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>lwsw</td>
<td>0.996</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>lwpk</td>
<td>0.998</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>tre</td>
<td>0.998</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>lwce</td>
<td>0.999</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>scra</td>
<td>1.000</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>dce</td>
<td>1.000</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>lwin</td>
<td>1.000</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>none</td>
<td>1.000</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>lunrl</td>
<td>1.001</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>licm</td>
<td>1.001</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>lwall</td>
<td>1.004</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>sccp</td>
<td>1.005</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>cprop</td>
<td>1.005</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>lwgc</td>
<td>1.007</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>ivsimp</td>
<td>1.014</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>scfg</td>
<td>1.028</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: Optimizations ranked in descending order of performance for benchmarks from the SPEC integer suite. The numbers are geometric means over all benchmarks and are relative to the performance of a program with no optimization (represented by “none” in the table).
benchmarks differ significantly from those from the floating point intensive programs. In stark contrast to the floating point results, optimizations that focus on loop nests were relatively ineffective on integer benchmarks – they failed to improve application performance to as large a degree as on floating point benchmarks. For instance, on the integer benchmarks, simple loop unrolling, lunnr1, recorded a slight degradation in performance as did loop invariant code motion and induction variable simplification. Partial redundancy elimination improved the performance of integer benchmarks on average. However, it could only reduce execution time by 0.7% on average. In contrast, on floating point benchmarks, the benefits from conducting pre were much more pronounced with an average reduction of 5.1% in execution time.

5.2.3 Opportunities exploited by simple loop unrolling

We were impressed by the effectiveness of the simple loop unrolling transformation on floating point benchmarks – it reduced observed execution time by 3.8% on average. In contrast, however, on integer benchmarks, it degraded application performance very slightly on average. We wished to examine further this difference in loop unrolling efficacy between the two sets of benchmarks. To that end, we measured the number of loops unrolled by the algorithm. On average, lunnr1 unrolls 0.08% (it unrolls only 4 loops out of 4965) of the inner loops in the integer benchmarks and 6.2% on floating point benchmarks. We were stricken by the significant difference in the number of loops unrolled. Since the integer benchmarks were not amenable to unrolling, the transformation did not lead to any performance gains. After further analysis, we were able to more fully understand the reasons for this disparity in opportunities. Most loops encountered by the JIT in the integer benchmarks did not contain loop bounds that are compile time constants. Therefore, the unrolling algorithm was limited in its ability to conduct unrolling. Further, there were structural differences (differences in code shape) between the floating-point and integer benchmarks. A larger proportion of loops in the integer benchmarks did not contain an induction variable. Specifically,
induction variables were absent in 47.85% of the integer loops as compared to in 27.89% of loops in the floating-point benchmarks. This underscores the differences in the nature of computations present in the two groups of benchmarks. In the integer benchmarks, several loops iterate over pointer structures such as trees and linked-lists and consequently lack induction variables. The floating-point benchmarks, in contrast, encode more structured scientific algorithms and loops in these programs therefore tend to be counting loops that contain induction variables.

Moreover, we compared our results with those obtained by Stephenson and Amarasinghe [88]. Their choice of benchmarks were similar to ours – in particular, they too conducted loop unrolling for programs in the SPEC suite. Though their evaluation goals differed from ours – they used machine learning techniques to compute good unroll factors – they obtained very similar results for the SPEC integer and floating-point benchmarks. In the experiments described in [88], unrolling proved to be largely ineffective on SPEC integer benchmarks when compared to the floating point benchmarks. For instance, their nearest neighbor algorithm degrades performance on 2 SPEC integer benchmarks, increases performance very slightly (≤ 1%) on 5 integer benchmarks, and increases performance by around 2% on one. In comparison, their unrolling strategy achieves better results on SPEC floating-point benchmarks. Note that most SPEC floating point benchmarks are authored in Fortran while SPEC integer benchmarks are written in C. The authors conclude that the source code language is an important feature for choosing unroll factors and suggest that Fortran code tends itself to easier analysis than programs written in C due to possible aliasing in C arrays. They also observe that Fortran applications are more prone to be scientific in nature thereby making them better suited for loop unrolling. We agree with the authors’ insights on the difference between loop unrolling effectiveness on floating-point versus integer benchmarks. For the reasons listed in this section, we shall focus on loop unrolling for scientific, floating point applications.
5.2.4 Findings and conclusions

As a consequence of these experiments, we decided to adapt inner loop unrolling for scientific benchmarks in a runtime compilation environment. After examining the results of our experiments, we noticed that, as stated in the previous section, loop based optimizations performed well on floating point, scientific benchmarks. In addition to loop unrolling, we also considered adapting partial redundancy elimination (PRE) for a JIT. However, it was unclear to us how the optimization would benefit from information available at program runtime. In particular, we found loop unrolling to be attractive for several reasons:

- Inner loop unrolling is a relatively simple modification of the program as we will see in Section 6.2. Therefore, we expect to conduct the transformation without consuming excessive compilation time.

- The transformation does not require intensive program analyses. A generic version of inner loop unrolling (again, demonstrated in Section 6.2) can be conducted on almost all counting loops without compromising safety. As we have demonstrated in previous chapters, program analysis can be a major impediment to compile-time efficient transformations. By eliminating the need for such analyses, loop unrolling looks promising for use on a runtime compiler.

- The loop unrolling mechanism shown in the experiments used a very simple algorithm. It was constrained to operate only on loops contained in a single basic block with compile-time constant loop bounds. In spite of these constraints, simple loop unrolling performed impressively in comparison to other, more algorithmically complicated, transformations. Consequently, the results suggested that a more complex loop unrolling technique augmented with runtime information could potentially be profitable on a JIT.

- Scientific applications are an important class of programs. A runtime compilation framework provides an attractive environment for many distributed,
computationally-intensive applications. We believe, therefore, that it is important to focus on making these programs efficient in a runtime compilation environment. Since scientific programs are loop-intensive and tend to conduct most of their computations in loops, a loop specific transformation such as unrolling can potentially increase the performance of such applications.

In the next chapter, we shall describe the design and implementation of a runtime loop unrolling strategy.
Chapter 6

Effective and Efficient Strategies for Runtime
Loop Unrolling

6.1 Introduction

Computer programs tend to spend a majority of their runtime executing a small proportion of the code [66, 85]. This is especially true for numerical and scientific programs. Further, these applications are generally well-structured and frequently executed code resides within loops. Optimizing loops, therefore, is important and may lead to improvements in application performance. Consequently, compiler literature describes many loop transformations that attempt to minimize loop execution time. As we have shown in the previous chapters, tailoring traditional compiler techniques towards runtime compilation can be extremely fruitful. In this chapter, we will focus on another traditional optimization – loop unrolling. In the next few sections, we will craft an unrolling strategy for runtime compilation. Through our exploration of loop unrolling strategies described in this Chapter, we make three important contributions:

- We design and implement a loop unrolling technique for runtime compilation that exploits runtime value information on loop bounds to guide code transformation

- We describe the design and implementation of a lightweight value examination mechanism that has extremely low overhead. This mechanism greatly increases the effectiveness of loop unrolling as compared to offline (ahead-of-time) unrolling techniques

- We show that such an unrolling strategy can be profitable and effective – it can
for (i = 0; i < N; i++)
   A = A + B[i]
endfor

for (i = 1; i < N-2; i+=3)
   A = A + B(i)
   A = A + B[i+1]
   A = A + B[i+2]
endfor
for (; i < N; i++)
   A = A + B[i]
endfor

Figure 6.1: Unrolling a loop by a factor of three. The unrolled loop on the right contains three copies of the original loop body followed by a cleanup loop

significantly reduce observed execution time for many applications

6.2 Loop unrolling

Loop unrolling is a transformation that has been cataloged in early compiler literature and continues to be implemented in almost all contemporary compilers. Unrolling a loop consists of duplicating the loop body multiple times. Figure 6.3 shows a loop transformed by unrolling. The number of times the loop body is duplicated is called the unroll factor of the loop. The term unroll factor is used ambiguously in the literature. There is some confusion about whether the phrase refers to the number of loop bodies duplicated during unrolling or to the total number of loop bodies after unrolling. We shall use the second connotation – an unroll factor of 1 will indicate that the loop has not been unrolled.

The unrolling mechanism, as evident in Figure 6.3, is simple – it consists of duplicating the loop body and adjusting the loop bounds. Additionally, a cleanup loop may be inserted as shown in the example. The cleanup loop is necessary for cases in which the unroll factor does not completely divide the iteration count.

6.3 Benefits of loop unrolling

Loop unrolling can be beneficial for a number of reasons. Unrolling reduces the number of runtime branches executed. Most modern architectures contain sophisticated
branch prediction hardware. Consequently, the presence of a branch results in extra overhead required by the hardware to predict the direction of the branch. For instance, the Pentium 4 optimization manual recommends that branches be removed, if possible, to avoid the overhead associated with branch history table lookup. Furthermore, reducing the number of dynamic branches can lead to fewer branch mispredictions. This is especially important on deeply pipelined machines that are currently prevalent. On modern architectures, many branch prediction mechanisms leverage the runtime behavior of a branch to attain highly accurate prediction rates [99, 74]. During program execution, the outcome of a branch is recorded in a branch history bit register. Due to the limited size of this register, the branch predictor cannot accurately predict branches that exhibit predictable behavior but whose patterns exceed the register size. By reducing the iteration counts of such loops, loop unrolling can potentially increase branch prediction accuracy.

Branches on several simpler processors, such as DSPs, are followed by delay slots — instructions that are executed independent of the branch outcome. Scheduling instructions in these slots may prove to be difficult for the compiler. Thus, on these architectures, unrolling may prove to be beneficial by reducing the number of unused branch delay slots. Unrolling also reduces the number of instructions executed in an iteration of a loop. In addition to branch removal, the transformation enables the removal of compare instructions required by branches and also instructions that update loop induction variables. The elimination of these instructions is beneficial for a super-scalar processor since it reduces the load on the functional units of the machine. Consider the execution units of a Pentium 4 processor shown in Figure 6.2. The issue ports for this machine dispatch instructions for execution. For instance, port 0 can either dispatch a FP move instruction or a branch/logic instruction on the first half of a processor cycle. Similarly, Port 1 can either issue an integer arithmetic instruction or a FP arithmetic instruction in the first half of a processor cycle. Therefore, decreasing the number of compares, branches, and induction variable updates
in the loop reduces the contention for issue slots on such a machine. This can lead to gains in program performance.

Unrolling a loop can also serve as an enabling optimization for an offline instruction scheduler. Since the unrolling mechanism exposes more instructions to schedule in a loop body, the scheduler can use these additional instructions to effectively hide instruction latencies. For instance, consider the program shown in Figure 6.3 for a hypothetical architecture where loads have a latency of 3 cycles, and arithmetic operations and stores have 2 cycle delay. The original code (shown in the leftmost panel) – a simple loop with an iteration count of 40 – does not provide the scheduler with opportunities to hide the latency of the memory operations. In contrast, the code unrolled with a factor of 4, displayed in the middle panel, allows the scheduler to effectively mask the latency of the loads and stores. The code after scheduling, shown in the right panel, is much more efficient and will lead to significant performance gains on the architecture.  

There are two notable impediments to aggressive unrolling of a loop. Unrolling increases the number of virtual registers required by the loop body. Depending on the register allocation algorithm, this may, in turn, lead to additional spill code in the loop. Therefore, the unroller must be carefully consider the degradation of performance that may be caused by register spills due to an overly aggressive strategy. Second, the post-unroll, expanded loop body may exceed the capacity of the instruction cache potentially increasing cache misses and fetch requests to main memory. The compiler, therefore, has to be judicious in making unroll decisions.

---

1The exact performance gains will depend on additional architecture characteristics such as whether the processor is super-scalar and pipelined, the organization of the functional units, the latencies of branch instructions, and whether the architecture can dynamically reschedule instructions.
6.4 Methodology and experimental framework

We wish to investigate the effectiveness of inner loop unrolling for runtime compilation. In particular, we wish to devise an unroll strategy that will be effective and profitable in a just-in-time environment. Runtime compilation is a time-critical process and therefore it excludes some strategies that were described above. Specifically, the cost of an intensive search for unroll parameters or the calculation of an intricate analytical framework proves to be prohibitive for a JIT. While we will be constrained by these considerations, we also wish to leverage the advantages of compiling in a dynamic environment. We wish to exploit the extra context provided to a JIT via runtime monitoring. A JIT environment can identify values that are frequently assigned to variables in a program. In a loop unrolling context, we can use this information to guide our unrolling strategy.

As described in Chapter 2, we use LLVM to conduct our experiments. Our loop unroller is invoked for procedures or methods that are invoked at runtime. \(^2\) Since

\(^2\)Our unrolling strategy shall focus on innermost loops in the code since merely unrolling such
<table>
<thead>
<tr>
<th>Cycle Number</th>
<th>Original Scheduled Code</th>
<th>Unrolled Code</th>
<th>Unrolled Scheduled Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LOOP 40 TIMES</td>
<td>LOOP 10 TIMES</td>
<td>LOOP 10 TIMES</td>
</tr>
<tr>
<td>0</td>
<td>LOAD</td>
<td>LOAD1</td>
<td>LOAD1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>COMPUTE1</td>
<td>LOAD2</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>STORE1</td>
<td>LOAD3</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>LOAD2</td>
<td>LOAD4</td>
</tr>
<tr>
<td>4</td>
<td>COMPUTE</td>
<td>COMPUTE2</td>
<td>COMPUTE1</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>STORE2</td>
<td>COMPUTE2</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>LOAD3</td>
<td>COMPUTE3</td>
</tr>
<tr>
<td>7</td>
<td>STORE</td>
<td>COMPUTE3</td>
<td>COMPUTE4</td>
</tr>
<tr>
<td>8</td>
<td>END LOOP</td>
<td>STORE3</td>
<td>STORE1</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>LOAD4</td>
<td>STORE2</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>COMPUTE4</td>
<td>STORE3</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>STORE4</td>
<td>STORE4</td>
</tr>
<tr>
<td>12</td>
<td>END LOOP</td>
<td>END LOOP</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.3: Enabling better instruction scheduling due to loop unrolling

the current version of LLVM is a compile-only JIT, it generates machine code for a procedure when it is first invoked. At that point, the bytecode representation of the procedure is lowered to a machine code format in memory. Subsequent calls to the procedure are handled by the machine code. In the next sections, we will present unrolling strategies that work within this framework.

6.4.1 Two motivational examples

To enable a discussion of runtime loop unrolling in more depth, we would like to present two examples that will elucidate the advantages of our approach. First, consider the application apsi from the SPEC CPU 2000 benchmark suite. apsi is a weather-prediction application that computes environment related metrics. The program consists of a moderately large number of procedures and loops. However, loops can prove to be profitable. In contrast, to realize fully the advantages of unrolling the outer loops, the compiler must conduct additional transformations after unrolling [73, 22].
<table>
<thead>
<tr>
<th></th>
<th>apsi</th>
<th>umCEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of procedures</td>
<td>125</td>
<td>6</td>
</tr>
<tr>
<td>Procedures executed</td>
<td>68</td>
<td>6</td>
</tr>
<tr>
<td>Proportion of procedures executed</td>
<td>54.40%</td>
<td>100%</td>
</tr>
<tr>
<td>Loops in code</td>
<td>408</td>
<td>110</td>
</tr>
<tr>
<td>Inner loops in code</td>
<td>291</td>
<td>76</td>
</tr>
<tr>
<td>Number of loops executed</td>
<td>111</td>
<td>17</td>
</tr>
<tr>
<td>Proportion of loops executed</td>
<td>27.21%</td>
<td>15.45%</td>
</tr>
<tr>
<td>Compile-time ascertainable loop bounds</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>Proportion of loop bounds ascertainable only at runtime</td>
<td>96.07%</td>
<td>99.09%</td>
</tr>
</tbody>
</table>

Table 6.1: apsi and umCEM program characteristics. apsi runtime characteristics have been gathered by running the application on the SPEC provided ref data set.

only a select number of these loops and procedures are executed at runtime. Further, the input data strongly determines which segments of code are invoked. Table 6.4 displays the static and runtime characteristics of apsi. As can be seen from the table, the program only executes 38.14% of the loops (and around 54.4% of the procedures) for the SPEC-provided ref input set. In addition, input parameters to apsi provide the length of the input matrix which in turn dictates the iteration counts of frequently executed loops.

Second, consider the umCEM scientific application developed by Dr. Tim Warburton at Rice University. umCEM solves Maxwell’s equations in complex geometries with inhomogeneous media[]. It uses an iterative method to continually refine the computation. The main kernel in umCEM consists of a matrix-matrix multiply implemented as a triply nested set of loops. The runtime of the program is dominated by the multiply routine. The multiply routine is called from several call sites in the program. The loop bounds of the three nested loops are passed in as parameters to the procedure. For every input set, umCEM calls the multiply routine with a different set of parameters. In particular, the iteration count for the inner loop is controlled by procedure parameters. Note that since the parameters to the routine are determined by input characteristics, an offline compiler cannot utilize information about loop bounds to guide its loop unrolling strategy.
The dynamic nature of these applications impedes the effectiveness of an offline loop unroller (invoked by an ahead-of-time compiler). There are two major obstacles encountered by an offline unrolling strategy. Such an unroller, in the absence of profiling data, will find it difficult to identify loops that are promising candidates for unrolling because program input characteristics dictate which procedures and loops are executed at runtime. Consequently, the unroller may decide to unroll all loops. However, this may lead to a significant increase in bytecode size that, in turn, will adversely affect bytecode transmission time. Moreover, an offline algorithm may not be able to aggressively unroll the loop. Since the iteration counts of frequently executed loops in \texttt{apsi} depend on input data, the offline algorithm has to be conservative in selecting loop unroll factors. If the unroll factor chosen by the algorithm exceeds the actual iteration count for an invocation of the loop, then code in the cleanup loop body – shorn of the advantages of loop unrolling – will execute. Therefore, an offline algorithm generally chooses a low unroll factor to allow most invocations of the loop to benefit from the unrolled code. In contrast, a runtime loop unrolling algorithm can exploit information about program input data to effectively unroll the loop and thus more effectively tailor the code for execution. A runtime strategy also allows the JIT to only unroll loops in procedures that are executed at runtime. This eliminates the potential bytecode bloat due to an indiscriminate unrolling of procedures by an offline compiler.

6.5 The case for effective loop unrolling at runtime

6.5.1 Performance enhancements

In our initial explorations, we decided to investigate the viability of runtime inner loop unrolling. To this end, we varied the unroll factor for the \texttt{umCEM} application for different input sets. Our hardware setup consisted of running the application on a Intel Pentium 4, 3.2 Ghz processor with 512 MB main memory. In our initial experiments, we used the GNU compiler, gcc. Reported times are a sum of system
and user time accrued by the program. Figures 6.4 and 6.5 depict the results of these experiments.

Figure 6.4: Execution time for umCEM application on input set 1 as the unroll factor for the innermost loop varies on a Pentium 4. The bar graph shows the relative improvements achieved by an unroll factor over the original loop.

We would like to highlight two key features of these results. First, note that the
Figure 6.5: Execution time for umCEM application on input set 2 as the unroll factor for the innermost loop varies on a Pentium 4. The bar graph shows the relative improvements achieved by an unroll factor over the original loop.

efficacy of an unroll factor changes with different input sets. Therefore, the results suggest that choosing unroll factors based on input data can be profitable. This result is important because it follows that a runtime unrolling strategy that exploits
input characteristics can significantly improve application performance. Second, the bar graphs in Figures 6.4 and 6.5 indicate that a runtime unrolling algorithm can be more suitable than its offline counterpart. An offline unroller can be too conservative in unrolling loops and thereby may be unable to exploit the full potential of unrolling. For instance, an unroll factor of 2 – a factor likely to be chosen by an offline unroller – in input set 1 achieves an improvement of 32% over the original loop. However, as the graph shows, a more aggressive strategy yields better improvements. The optimal unroll factor – 27 in this case – achieves a 55% improvement over the original loop and an additional 33% improvement over an unroll factor of 2. Alternatively, however, an overly aggressive offline unrolling strategy may be misled by flawed or stale profile data and can, consequently, degrade application performance. The optimal unroll factor for the first input set – 27 – performs relatively poorly for input set 2 as shown in Figure 6.5.\footnote{This occurs because the loop bounds are different for the two different inputs – an unroll factor of 27 for input set 2 leads to excessive execution of the cleanup loop.} Therefore, using profiling data from previous runs of the application to guide offline unrolling strategies may prove to be disadvantageous, and even detrimental, to application performance. In addition, an ahead-of-time loop unrolling strategy for portable representations is unable to exploit architecture-specific information such as cache sizes and the number of registers available. Unrolling at runtime, in contrast, does not suffer from these drawbacks. Such an algorithm can carefully examine input data and architectural characteristics at runtime and then choose a suitable unroll factor for a specific execution of the application.

An interesting feature of these results is the drastic reduction in execution time obtained by unrolling \texttt{umCEM}. For instance, the optimal unroll factor for the first input set – unroll factor 11 – performs over 50% better than the original program. Further investigation yielded that a number of factors contribute to the significant improvements after unrolling. The procedure containing the unrolled loop is responsible for most of the computations conducted by the application. A profiled version of \texttt{umCEM}
indicates that the procedure accounts for 95.22% of total execution time. Thus, moderate improvements in this procedure results in large improvements in application runtime. Further, after unrolling, there are four factors that contribute towards increasing the efficiency of the procedure: a decrease in the number of branches and mispredictions, improvements in register allocation, and a decrease in the number of instructions. We shall analyze the decrease in branches and mispredicts in more detail in Section 6.8.3. It is interesting, however, to consider the impact of unrolling in umCSEM to the number of emitted instructions and the improvements in register allocation. In the original (not unrolled) program, out of a total of 20 instructions in the loop, there are 8 x86 assembly language instructions that are emitted to compute array accesses. While some of these instructions are loop invariant and consequently can be hoisted out of the loop, gcc decided to keep these computations in the loop to avoid the spill instructions that would be necessary to reload values if they were placed outside the loop. 4 The loop also contained 6 instructions that computed the loop control condition and branched back into (or exited) the loop. These instructions needed to be replicated only once per iteration. Therefore, unrolling the loop decreased the number of instructions required per iteration. For instance, while the compiler emitted 20 instructions to encode the original program, it needed only 27 instructions to encode the loop unrolled by the factor of 2. 5

Unrolling the loop also had an auxiliary effect on register allocation. For a small unroll factor, gcc's register allocator chose to spill array references to memory. However, for unroll factors greater than 8, gcc decided to place those references in registers. Unrolling affected the register allocator's decisions by increasing the number of

4This was compiled using the highest optimization level, -O3. The loop had high register pressure and therefore if some of these values were moved out of the loop and, consequently, farther away from their uses, they would have to be reloaded. A stronger operator strength reduction transformation would have been able to eliminate some of the instructions [32].

5While the unroll-by-2 eliminated the 14 described instructions, it added an extra instruction. This resulted in a savings of 13 instructions per loop iteration.
static uses of the array reference. Register allocators, as described in Chapter 3, typically use a heuristic that counts the number of static occurrences of a virtual register in the program. Register allocators tend to favor the allocation of virtual registers with higher usage frequencies to registers. Therefore, unrolling the loop resulted in those array references being placed in registers. The removal of spill code also reduced the number of instructions needed to encode the loop resulting in a further increase in efficiency. For instance, for an unroll factor of 11, only 62 instructions are needed to encode the loop. Since the loop was executed a large number of times, the elimination of spill code improved the runtime of the program.

6.5.2 Space savings

The apsi application exhibits another feature that reiterates the utility of runtime unrolling. As described in Section 6.4.1, it contains a large number of procedures and loops that execute conditionally based on input data. Applying an offline unrolling strategy to such an application can lead to a significant growth in bytecode size. We measured the growth in bytecode by unrolling all loops of apsi. Figure 6.5.2 shows the growth in bytecode size as the unroll factor increases. Such an indiscriminate unrolling of loops by the offline compiler led to a 22% increase in bytecode size for an unroll factor of 2 – from 186 KB to 417 KB. This is unfortunate since an increase in bytecode size may result in an increase in transmission time. An increase in code size can also pose additional constraints in storage-limited and memory-limited environments. Therefore, the offline compiler has to be conservative in unrolling the loops to limit the potentially large increase in program size. Further, the offline unroller cannot selectively unroll procedures and loops because of the dynamic nature of apsi. For these reasons, in an application such as apsi, a runtime unrolling technique has an opportunity to outperform an offline algorithm.
<table>
<thead>
<tr>
<th>Unroll Factor</th>
<th>Size (bytes)</th>
<th>Percentage Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>186592</td>
<td>0.00%</td>
</tr>
<tr>
<td>2</td>
<td>228524</td>
<td>22.47%</td>
</tr>
<tr>
<td>3</td>
<td>242371</td>
<td>29.89%</td>
</tr>
<tr>
<td>4</td>
<td>255605</td>
<td>36.99%</td>
</tr>
<tr>
<td>5</td>
<td>271095</td>
<td>45.29%</td>
</tr>
<tr>
<td>6</td>
<td>287344</td>
<td>54.00%</td>
</tr>
<tr>
<td>7</td>
<td>303499</td>
<td>62.65%</td>
</tr>
<tr>
<td>8</td>
<td>318945</td>
<td>70.93%</td>
</tr>
<tr>
<td>9</td>
<td>334395</td>
<td>79.21%</td>
</tr>
<tr>
<td>10</td>
<td>348974</td>
<td>87.03%</td>
</tr>
<tr>
<td>11</td>
<td>363845</td>
<td>95.00%</td>
</tr>
<tr>
<td>12</td>
<td>378187</td>
<td>102.68%</td>
</tr>
<tr>
<td>13</td>
<td>389078</td>
<td>108.52%</td>
</tr>
<tr>
<td>14</td>
<td>402361</td>
<td>115.63%</td>
</tr>
<tr>
<td>15</td>
<td>417325</td>
<td>123.66%</td>
</tr>
</tbody>
</table>

Table 6.2: Growth in bytecode representation due to offline unrolling of apsi as unroll factor increases

6.6 Examining runtime program values: background

As we have discussed in Section 6.5, values that are assigned to variables at runtime can be exploited to direct compiler transformations. This idea has been explored in compiler research. Due to the popularity of JITs in the last decade, some researchers have implemented value profiling techniques that record program values and use that information to guide program optimizations. Our approach shares similarities with prior research conducted in value profiling and program specialization.

Value profiling

Calder et al. introduced the term value profiling - they recorded the variance in program values for a number of benchmarks and demonstrated that optimizations can take advantage of the relatively low variance exhibited by many values [20]. Their scheme attempts to identify when profiling values have converged and discontinues subsequent profiling. Watterson and Debray devised a goal directed model of gathering program values by instrumenting a program offline and running it on representative
training data [95]. The authors described methods to reduce the overhead of value profiling by estimating the utility of a value profile for optimization. Arnold and Ryder used code duplication to generate value samples – the duplicated code contains instrumentation fragments and is executed on being triggered by a specified sample condition [8]. Their technique guarantees that only a bounded amount of time is spent in executing the instrumented version of the code.

**Runtime code specialization**

By virtue of compiling code just before execution, a JIT can use extra contextual information to guide the optimization process. To this end, several research efforts have focussed on specializing programs to exploit available runtime information. This class of optimizations is also referred to as *online feedback-directed* techniques. Early efforts by Deutsch and Schiffman in this area sped up method lookup by dynamically modifying the call-site and inserting a relinking check at method entry [41]. Their experiments indicate a hit-rate of 95% for the inline rewrite technique. Holzle and Ungar used inlining to reduce the number of dynamically dispatched calls for the object-oriented SELF-93 system [50]. They also inserted invocation frequency counters to determine which methods to optimize. By using these methods, they intended to reduce the pauses experienced by the user while running a program. Dynamo is a system developed by researchers at Hewlett-Packard Labs that interprets the executing program and identifies “hot traces” – frequently executed code fragments [12]. The traces are then optimized for fast execution. Their technique shares similarities with hardware trace-cache mechanisms seen on modern processors such as the Intel Pentium 4. Arnold et al.’s feedback-directed optimizations for Java is closely related to our work. They implement the instrumentation sampling infrastructure methods described in [8] for a Java JIT and use that to conduct runtime optimizations. Specifically, they conduct inlining, splitting, code positioning, and loop unrolling. However, unlike our framework, they do not examine runtime program values. They only con-
sider basic block and procedure frequencies. Therefore, they use a simplistic scheme to guide loop unrolling – unroll factors are doubled for frequently executed loops and halved for cold loops [7]. Schultz et al.'s JSpec tool examines caller-receiver pairs in a program and specializes classes for specific method invocations. This is similar to the procedure-cloning work described in an offline context by Cooper et al [34].

While the literature describes several approaches to value profiling and runtime specialization, our transformation strategy combines both these techniques for a particular optimization – loop unrolling. Our approach, unlike many strategies discussed above, examines program values at runtime and automatically transforms the code while the program is executing without any user intervention. It does not rely on representative, training input data or pre-execution profiling runs. Further, in contrast to the prior research described above, the technique we devise to examine program values is extremely lightweight and, as we shall show, adds practically no overhead to the observed execution time (i.e. runtime compilation + program execution time).

6.7 Description of our runtime unrolling algorithm

Our initial experiments described Section 6.4.1 highlighted the opportunities present for a runtime loop unrolling algorithm. We, thus, wanted to use the insights we acquired by an examination of umCEN and apsi to craft a JIT loop unroller that can exploit these opportunities and, consequently, increase application performance. Before describing our algorithm, it is helpful to understand the overall structure and workings of the LLVM JIT during program execution. The LLVM infrastructure includes an offline compiler that is used to generate LLVM bytecode representation from high-level code. The bytecode is a platform independent representation that can then be executed by the LLVM JIT. The JIT currently uses a compile-only strategy – when a procedure is first invoked, it produces relocatable machine code for that procedure and transfers control over to the compiled code. An important feature of the runtime compiler is that it compiles procedures on-demand. When a procedure is
Figure 6.6: Runtime code generation mechanism in the LLVM JIT. The top panel depicts how the LLVM JIT generates code on-demand during application execution. The bottom panel shows how a sample procedure stub is generated.

compiled, the JIT replaces all call-sites in the generated code with stubs that contain callbacks to a JIT routine. If such a stub is invoked, then transfer passes to the JIT compiler which then generates code for the called procedure. Further, it replaces the stub with a call to the newly compiled procedure so that future invocations can avoid the overhead of the JIT callback. This mechanism is shown in Figure 6.6. Note that machine code for a procedure is only generated once—the callback routine tracks if it had previously generated code for the procedure. The compile-once approach is typical behavior for a JIT and most runtime compilers for desktop and scientific platforms follow this strategy. Some JITs for mobile devices discard previously compiled code due to storage constraints and researchers have devised code-replacement strategies for these systems [100, 89].

We designed and implemented a loop unrolling algorithm in the LLVM framework.
Given an unroll factor, the algorithm generated an unrolled inner loop consisting of multiple copies of the original loop, and, if necessary, a cleanup loop. The algorithm generated unrolled loops similar to the simple example shown in Figure 6.3. Since we wished to use input data characteristics to guide our unrolling strategy, we also implemented a lightweight mechanism to examine program values at runtime. This mechanism was the critical component in determining the efficacy of the unrolling transformation. In the next few sections, we shall describe the value-examining mechanism in more detail.

6.7.1 A lightweight value examination mechanism for loop unrolling

To enable an effective unrolling of loops at runtime, we wanted to exploit information about values assigned to loop bounds at runtime. Thus, we decided to implement a mechanism that would examine loop values at runtime and subsequently, use that information, to guide the unrolling process. In crafting our value examining mechanism for loop unrolling, our design goals were:

- The mechanism should be transparent to the user: We wanted to avoid additional user input, such as source code modifications or offline profiling information, to guide our loop unroller. Our goal was to choose which loops to unroll as well as how much to unroll (i.e., choose the unroll factor) automatically at runtime.

- We wanted to ensure that the mechanism had extremely low overhead for two reasons: First, we wanted to minimize runtime compilation time to reduce observed execution time. Second, we wished to eliminate the potential penalty incurred by unrolling decisions. If the unroller chooses a loop that does not greatly impact program performance, we do not wish to expend significant compilation time to gather value information. This strategy contrasts with prior research discussed in Section 6.6 in which more compile-time intensive value profiling techniques were used.
The value examination mechanism we designed differs from value profilers in that it does not attempt to continuously monitor the flow of runtime values. Rather, it reads value information only when the loop unrolling mechanism is invoked. To ease discussion, we shall refer to this mechanism as the value examiner or the examiner. We shall use the term profiler to refer to the mechanisms described in Section 6.6.

To implement our lightweight value examiners, we augmented the LLVM JIT machine-code generation mechanism. Consider the framework shown in Figure 6.6. We modified the compilation process so that when a procedure is first invoked, the JIT shares information about the current runtime environment with the value examiner. The examiner records that information and also analyzes the procedure parameters and global variables. Note that at this point, it does not attempt to extract value information. Next, the loop unroller is invoked which examines the loops. On encountering a loop with runtime loop bounds, the unrolling algorithm consults the examiner for iteration count information. When faced with such a request, the value examiner analyzes type information on procedure parameters and global variables and ensures that reading the required runtime values is safe. If the analysis indicates it is safe to extract value information, the examiner reads information from the runtime environment and provides the loop unroller with loop bounds information. Using this information, the unroller chooses an unroll factor and transforms the loop. Information about the current environment is gathered by the JIT in an architecture-specific manner. For instance, on the x86 platform, parameter passing conventions dictate that procedure arguments be passed on the stack. Therefore, we use the value of the current stack pointer to extract runtime values of procedure arguments. The runtime loop unrolling framework is shown in Figure 6.7.

Selection of loops to unroll

In crafting our runtime unroll strategy, we decided to focus on loops in the program whose iteration range cannot be determined at compile-time. As we shall describe in
Figure 6.7: The runtime loop unrolling framework.

the next few sections, the runtime loop unrolling strategy we present only considered loops with bounds that are not compile time constants. We wished to specifically exploit situations in which the use of an offline compiler is limited. While an offline compiler can use symbolic unrolling for these kinds of loops, such an approach suffers from the drawbacks listed in Section 6.4.1. In contrast, an offline compiler can be much more effective on loops whose bounds are known at compile-time. Therefore, for purposes of unrolling loops for a JIT, we decided to ignore such loops and focus exclusively on loops with bounds that can vary at runtime.

**Ensuring safety in the value examiner**

Our goal in implementing a value examiner was to enable our loop unroller to tailor loops for loop bounds that are known only at runtime. Specifically, we focussed on loop bounds that are derived from procedure arguments. As can be expected, type information for arguments in procedures containing loops can be varied. Loop bounds have to be integral in type. While many procedures contain integer parameters that determine loop bounds, loop bound information can also be provided by more complex data structures such as integer pointers or compound data types such as C-style structs. Extracting information from these more complex data structures may prove to be beneficial. However, when extracting information from such data
UnrollLoop(Loop L, ExaminedValues EV)

; The procedure unrolls loop L and is provided with a set, EV, of
; runtime values collected by the value-examiner

let B be the loop block which contains a back-edge to the loop header
if there are multiple back edges
    return

examine the conditional instruction, I, that controls the back edge
if I compares an induction variable, IV, to a value E ∈ EV
    conduct safety check on E and return if it fails
    compute the iteration count, IC, of the loop based on the initial value
    of IV, and the values of E, and IV's increment in the loop

ComputeUnrollFactor(IC)

; unroll the loop
for J = 1 to unroll_factor - 1
    duplicate all blocks from the original loop
    change conditional instruction in B to an unconditional jump to the
    duplicated header
    for each duplicated block, DB
        change uses in DB of all values defined in the original loop to
        definitions in the duplicated loop

; attach cleanup loop
create the pre-cleanup loop block
duplicate all blocks from the original loop to create cleanup loop blocks
attach the loop exit block to pre-cleanup loop block
add conditional branches to post loop and cleanup preheader blocks
change uses in the cleanup blocks of all values defined in
the original loop to definitions in the cleanup loop

modify conditional branch operands in the pre-loop, loop exit, and the
pre-cleanup blocks to reflect the unroll_factor of the loop

Figure 6.8: Overview of runtime unrolling algorithm. The control-flow graph before
and after unrolling is shown in Figure 6.14

types, the value examiner must be careful not be violate safety constraints. For in-
stance, consider a examined variable that points to an integer in memory. The value
of the variable (i.e. the location pointed-to by the variable), however, is modified
ComputeUnrollFactor(IterationCount IC):
    ; compute unroll factor
    if IC <= 4
        set the unroll_factor to IC
    else for each integer UF = 16 down to 2
        if UF divides IC
            set the unroll_factor to UF
    else
        set the unroll_factor to 16

Figure 6.9 : Compute unroll factor

in the procedure prior to any uses and subsequently is used as a loop bound. Remember that the value examiner is passed the current runtime environment before procedure invocation. Therefore, if the examiner attempts to read the value of the integer pointer, it will initiate a memory access that may compromise safety. Thus, to ensure safety, the examiner analyzes the procedure and accesses memory if the safety analysis guarantees that the memory location for a function argument is not modified before being used as a loop bound. (i.e. it points to the same location) between the function invocation and its use as a loop bound. Recall that the value examiner only considers global values or function parameters with one level of indirection. Consider an examined value, $x$. The safety of the examination mechanism can be compromised if a variable – global value, function parameter, or a local variable – points to $x$ and the dereferenced value of that variable is changed. Consequently, the value examiner rejects all examined values whose storage location is assigned to another variable or stored to memory. the function entry point.

6.7.2 Choosing unroll factors

As shown in Figures 6.4 and 6.5, the choice of an unroll factor can have a tremendous affect on program performance. In the experiments described below, we used two
approaches to select unroll factors based on runtime information. Our first selection criteria was simple – we chose an unroll factor of 2 for every unrolled loop. By doing so, we wanted to gauge how such a simplistic choice of unroll factors would benefit application performance. In formulating our second selection strategy, we wished to observe if the benefits of a more aggressive runtime unrolling would overcome the potential penalties accrued due to factors that hinder loop unrolling – namely additional register pressure and instruction-cache performance. Therefore, we selected unroll factors by examining the value of the loop bounds on the first invocation of the loop. Given a loop bound, $L$, we completely unrolled the loop if the loop bound was $\leq 4$. For larger bounds, we checked if the bound was completely divisible by integers 16 through 2 and chose the factor that divided the loop bound. For all other loop bounds, we used a unroll factor of 16. Our selection strategy for unroll factors was driven by a desire to reduce the number of cleanup loop invocations. Further, we decided to limit the maximum unroll factor to 16 to avoid indiscriminate unrolling. Since we examine the loop bounds to determine the unroll factor, and loop bounds in certain programs can be significantly large (greater than 1000 for some programs in our benchmark suite), we wanted to ensure that the unroller imposes an upper limit on the amount of unrolling. We would like to note that while more complex unrolling strategies are documented in the literature, as seen in Section 2.6, our goal was not to evaluate different selection techniques for unroll factors. Rather, our research focused on the effect of using runtime values to conduct loop unrolling. Figures 6.8 and 6.9 provides an overview of our unrolling algorithm in pseudo-code.

6.8 Experimental results

We present an evaluation of our runtime unrolling strategy for 14 applications. We have discussed the characteristics of umCEM and apsi in Section 6.4.1. Table 6.3 provides an overview of the benchmarks. All benchmarks, except umCEM are from the SPEC floating point suite. We included all benchmarks from the SPEC 2000 floating
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>301.apsi</td>
<td>Weather prediction application</td>
</tr>
<tr>
<td>172.mgrid</td>
<td>Multigrid solver for a three dimensional potential field</td>
</tr>
<tr>
<td>umCEM</td>
<td>Solver for Maxwell’s equations in complex geometries with inhomogeneous media</td>
</tr>
<tr>
<td>125.turb3d</td>
<td>Simulator for isotropic, homogeneous turbulence in a cube</td>
</tr>
<tr>
<td>200.sixtrack</td>
<td>Simulator for particle accelerator</td>
</tr>
<tr>
<td>104.hydro2d</td>
<td>Solver for hydrodynamical Navier-Stokes equations</td>
</tr>
<tr>
<td>179.art</td>
<td>Image recognition via neural networks</td>
</tr>
<tr>
<td>168.wupwise</td>
<td>Solver for quantum chemodynamics</td>
</tr>
<tr>
<td>171.swim</td>
<td>Weather prediction application, shallow water modelling</td>
</tr>
<tr>
<td>183.equake</td>
<td>Earthquake simulator</td>
</tr>
<tr>
<td>177.mesa</td>
<td>3-D graphics library</td>
</tr>
<tr>
<td>188.ammp</td>
<td>Molecular dynamics computations</td>
</tr>
<tr>
<td>173.aplu</td>
<td>Computational Fluid Dynamics</td>
</tr>
<tr>
<td>103.su2cor</td>
<td>Computations of masses of sub-atomic particles</td>
</tr>
</tbody>
</table>

Table 6.3: Description of Benchmarks. SPEC descriptions have been summarized from [86].

point suite except four – 178.galgael, 183.faceres, 189.lucas, and 191.fma3d. These benchmarks are authored in FORTRAN-90. LLVM, currently, does not provide a FORTRAN-90 frontend. For the SPEC FORTRAN benchmarks, we converted the code into C by using the f2c utility. In addition to the SPEC 2000 benchmarks, we evaluated the runtime unroller on two benchmarks from the SPEC 95 benchmark suite – turb3d, and hydro2d, and also on the umCEM application described in Section 6.4.1. We chose to test our unrolling algorithm on these benchmarks since they seem well suited to benefit from online unrolling of loops. All the applications conduct scientific computations and contain several loops that dominate execution time.

6.8.1 Opportunities exploited by runtime loop unrolling

We evaluated our unrolling algorithm by compiling the benchmarks to the LLVM bytecode representation. The bytecode was then executed by the LLVM runtime system which invoked the JIT as necessary. Loop unrolling (as we will see in Sec-
Figure 6.10: Breakdown of loop bounds in inner loops for our benchmark suite.

Section 6.8.5) requires a considerable modification of the intermediate representation’s control flow. To keep our analysis simple, we decided to invoke the unrolling algorithm only on certain types of inner loops. Specifically, the algorithm only considered loops with a single entry to the pre-header of the loop and a single exit out of the loop. Further, the runtime loop unroller ignore loops if the conditional that was responsible for the exit out of the loop did not involve an induction variable. The unroller also did not handle loops in which the loop controlling condition was computed in multiple basic blocks. Finally, the algorithm disregarded loop bounds that were contained in complicated data structures such as multi-dimensional arrays or arrays contained within structs. We imposed these constraints mainly to simplify the unrolling algorithm. A complex algorithm that considered all possible cases for loop unrolling could potentially consume additional compile time. Further, we wanted to verify if we could improve application efficiency by an examination of loops that fit our relatively strict criteria. As stated in Section 6.7.1, the lightweight value examiner extracts runtime information from global variables and procedure parameters. However, loop bounds can be contained in other locations such as procedure-local variables or constant expressions. Figure 6.10 shows a breakdown of loop bounds
Figure 6.11: Breakdown of loops that the runtime loop unroller can examine.

encountered in our benchmark suite. As can be seen from the graph, around 27% of the loops analyzed by the runtime unroller did not contain an induction variable. Since these loops are non-counting loops, information about their iteration counts are generally difficult to obtain. 6 7% of the loop bounds were constant expressions. As we mentioned in Section 6.7.1, we focussed solely on loops with values available only at runtime. Therefore, our algorithm ignored loops with compile-time constant loop bounds. Around a quarter of the loop bounds were contained in local variables whose values, by definition, are assigned or computed in the procedure. Since the value examiner extracted runtime values before a procedure started executing, these values were inaccessible to our runtime unroller. 41% of the loops contained loop bounds that could be successfully examined by the runtime unroller. However, due to the constraints we placed on the runtime loop unroller, it could not analyze many of these loops. Figure 6.11 provide a breakdown of loops whose bounds could be examined. As can be seen in the pie-chart, the runtime unroller could unroll around

---

6For instance, a common non-counting loop is a while loop iterating over a linked list. It is possible to unroll these loops but strategies to do so involve insertion of conditionals in the loop or a modification of the application's data structures [52].
27% of these loops.

### 6.8.2 Effect on application performance

Our baseline for comparison consisted of executing the applications with no unrolling of loops. We compare the two strategies described in Section 6.7.2 with the baseline program. In our graphs, the runtime unroller that always chooses an unroll factor of 2 is labeled “Runtime-2”. The more aggressive unroll-factor selection strategy is labeled “Runtime”. Figure 6.13 shows the results of our experiments for the “Runtime” strategy. Using the aggressive runtime unrolling strategy resulted in reducing execution time by 12.5% on average. In comparison, as the results in Figure 6.12, the Runtime-2 technique reduced execution time by 6.4% on average.

To further understand the effects of runtime unrolling, we measured the impact loop unrolling had on runtime hardware events. We used the Intel vtune utility to gather hardware counter information. In particular, we measured five characteristics: number of dynamic branches executed, the number of branch mispredictions, the number of loads and stores executed, and the number of instruction-cache misses incurred by the benchmarks. As can be seen in Figures 6.13 and 6.12, the runtime unrolling strategies resulted in a significant decrease in executed branches and branch misprediction rates. On average, Runtime and Runtime-2 reduced the number of executed branches by 12.7% and 7.8% respectively over the baseline. The decrease in branch mispredictions was also significant – 21.8% improvements on average for Runtime and 10.1% for Runtime-2.

---

7The Pentium 4 does not provide hardware counters that measure capacity, conflict, or compulsory misses in its first-level instruction-cache. As described in [53], one of the available performance counters can measure the trace cache misses caused by mispredicts from its internal branch prediction mechanism. However, no counter is provided to measure trace cache misses that occur due to other factors. We, therefore, measured the L2-cache read misses; note that the Pentium 4 L2 cache is a combined data and instruction cache.
Figure 6.12: Effect of runtime loop unrolling on benchmarks. This graph shows the performance of the Runtime-2 unrolling strategy.

Note that, as indicated in the experiments, unrolling reduces the number of loads and stores executed. This occurs because unrolling the loop eliminates induction variable updates and branch condition computations. That leads to a decrease in register pressure and the allocator can eliminate spill loads and stores that would have been inserted in the original loop.

6.8.3 Analyses of results

The Runtime-2 and the aggressive runtime unroller achieve similar results on mgrid. On further analyses of this benchmark, we determined that several loops that dominate execution time have a loop bound of 4. Therefore, both algorithms unroll these loops twice which in turn results in similar observed execution times. On wupwise, Runtime-2 performs considerably worse than Runtime. This is reflected in both the relative execution time and the occurrence of large number of branch mispredictions in Runtime-2. The degradation occurs because of a bad choice of unroll factors. We will examine this result in more detail in Section 6.8.5. The algorithm causes a
Figure 6.13: Effect of runtime loop unrolling on benchmarks. This graph shows the performance of the Runtime unrolling strategy.

degradation in 183.equake – application runtime increases by 1.15%. This occurs because the runtime unroller is unable to unroll any loops in the benchmark. The difference in execution time is slight in absolute terms – the unrolled version takes 1.09s more time to complete. The overhead of the runtime loop unroller is responsible for this degradation. Runtime loop unrolling was very beneficial for the umCEM application. As can be seen in Figure 6.13, unrolling led to a drastic improvement in all measured metrics. Importantly, execution time for the application was reduced by around 48.7% by Runtime and by around 28.6% by Runtime-2. Further investigation yielded that umCEM contains a matrix-matrix multiply subroutine that greatly dominates its execution time. The subroutine accounts for more than 95% of the application’s total execution time. Since the multiply kernel consists of a triply nested loop, unrolling of this loop led to a significant reduction in the kernel’s execution time, resulting, in turn, to a drastic improvement in application performance. The SPEC benchmark sixtrack also greatly benefits from runtime unrolling because of similar reasons. These two benchmarks contribute heavily towards the mean performance
of the runtime unroller. Therefore, in addition to the mean execution-time improvements displayed in the two graphs, we would like to include a few more measures of central tendency: if we exclude the two applications, the geometric mean of reduction in program execution time is 6.8% for Runtime and 3.9% for Runtime-2. Further, the $2^{nd}$ trimmed geometric mean of unrolled execution-times relative to the original program yields an improvement in program execution time of 8.4% for Runtime and 4.9% for Runtime-2. The efficacy of the runtime unroller varies significantly amongst the benchmarks in our testing suite. The improvements in execution time vary from a slight degradation of 1.2% in 183.equake to an improvement of 48.7% on umCEM. This variation is reflected in the standard deviation of execution-time improvements of 14.9% for Runtime and 10.0% for Runtime-2.

### 6.8.4 Overhead of unrolling

One of our primary goals was to ensure that the loop unrolling and the value examining mechanisms are efficient. We, thus, measured the compilation time required by the runtime unrolling algorithm. The results of these experiments show that the lightweight value examiner and the loop unroller consume very little compilation time – less than 0.05% of observed execution time in all benchmarks. Therefore, we were pleased to note that the runtime unrolling strategy can increase program performance without adversely affecting compilation time.

### 6.8.5 Post-unroll cleanup loop usage

The unrolling mechanism, as depicted in Figure 6.3 is simple: the original loop is replaced by a set of two consecutive loops. The first loop is the unrolled loop, the second loop contains the cleanup loop. The changes in the control flow graph are slightly more intricate. The left panel in Figure 6.14 shows a loop in LLVM before unrolling is applied. Note that we run the loop simplification pass in LLVM for easier analyses. The transformation ensures that there is a unique pre-header basic block (a
Figure 6.14: The control flow graph before and after loop unrolling is applied. In the right panel, cleanup loop blocks have been highlighted in gray.

basic block that contains a single control-flow edge into the loop header) for each loop in the code. The runtime loop unroller makes a number of changes to the structure of the loop. The right panel in Figure 6.14 shows the control-flow graph after loop unrolling. Note that unrolling adds 4 extra basic blocks and 7 extra edges to the loop. Many of the changes in the control-flow graph follow directly from the high-level changes shown in Figure 6.3 and need no explanation. The unroller inserts two edges that skip over the unrolled loop and the cleanup loop respectively. These edges, the large arcs in Figure 6.14, are traversed if the value of the loop induction variable is greater than the loop bound. In particular, these edges represent the decision to
either invoke or bypass the unrolled and cleanup loops.

Invoking the cleanup loop is generally not desirable and results in execution overhead. Note from Figure 6.14 that the cleanup loop can be executed along two paths. Program execution may lead to control being transferred along the edge from the Pre Loop block to the Pre Cleanup Loop. This corresponds to the case when the initial pre-loop value of the controlling loop induction variable exceeds the value of the loop bound. The cleanup loop can also be reached without bypassing the unrolled loop, via the Pre Cleanup Loop block. In this case, the loop bound is not completely divisible by the unroll factor and the cleanup loop must execute the remaining loop iterations. In both cases, cleanup loop execution leads to potentially unnecessary control flow and, consequently, additional branch execution. Also, since the cleanup loop does not offer the benefits of loop unrolling, we would prefer to minimize the executions of the loop.

To this end, we wanted to measure the invocations of the cleanup loop during program execution. Our goal was to check whether the selection of the unroll factor by our runtime loop unrolling mechanism led to extensive executions of the cleanup loop. Recall that the runtime unroller chose an unroll factor on the first execution of the procedure. Therefore, we wished to examine the effect of such an eager selection of unroll factors on cleanup loop invocations. We augmented the LLVM framework with a dynamic instrumenter that inserts code to count basic block executions at runtime. We, then, invoked the benchmarks and analyzed execution block frequencies. Table 6.4 shows a summary of the execution frequencies of the cleanup loops for the Runtime and the Runtime-2 strategies. An examination of the statistics displayed in the table helped us understand why the Runtime-2 strategy on the wupwise benchmark fared much worse than Runtime. As can be seen in the table, an unroll factor of 2 resulted in a significantly large number of cleanup loop iterations – the program executed half as many iterations as the unrolled loop. This resulted in the execution of additional conditional branches, which in turn increased the number of branch mis-
<table>
<thead>
<tr>
<th></th>
<th>Runtime</th>
<th></th>
<th></th>
<th>Runtime-2</th>
<th></th>
<th></th>
<th></th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unrolled</td>
<td>Cleanup</td>
<td>Total</td>
<td>Unrolled</td>
<td>Cleanup</td>
<td>Total</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(\times 10^6)</td>
<td>(\times 10^3)</td>
<td>(\times 10^6)</td>
<td>(\times 10^6)</td>
<td>(\times 10^3)</td>
<td>(\times 10^6)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>301.apsi</td>
<td>1826.9</td>
<td>75031.3</td>
<td>1901.9</td>
<td>6871.4</td>
<td>31408.2</td>
<td>6902.8</td>
<td>0.2755</td>
<td></td>
</tr>
<tr>
<td>172.mgrid</td>
<td>3167.9</td>
<td>119556.8</td>
<td>3287.5</td>
<td>4583.4</td>
<td>0</td>
<td>4583.4</td>
<td>0.7173</td>
<td></td>
</tr>
<tr>
<td>umCEM</td>
<td>131.0</td>
<td>11962.2</td>
<td>142.95</td>
<td>392.91</td>
<td>11974.6</td>
<td>404.88</td>
<td>0.3530</td>
<td></td>
</tr>
<tr>
<td>125.turb3d</td>
<td>196.0</td>
<td>0</td>
<td>196.01</td>
<td>784.05</td>
<td>0</td>
<td>784.05</td>
<td>0.2500</td>
<td></td>
</tr>
<tr>
<td>200.sixtrack</td>
<td>998.6</td>
<td>0.249</td>
<td>998.60</td>
<td>1497.9</td>
<td>0.112</td>
<td>1497.9</td>
<td>0.6667</td>
<td></td>
</tr>
<tr>
<td>104.hydro2d</td>
<td>107.5</td>
<td>10.03</td>
<td>107.53</td>
<td>161.28</td>
<td>5.015</td>
<td>161.28</td>
<td>0.6667</td>
<td></td>
</tr>
<tr>
<td>179.art</td>
<td>161.8</td>
<td>0</td>
<td>161.8</td>
<td>404.48</td>
<td>0</td>
<td>404.48</td>
<td>0.4000</td>
<td></td>
</tr>
<tr>
<td>168.wupwise</td>
<td>590.1</td>
<td>0</td>
<td>590.1</td>
<td>703.74</td>
<td>362752</td>
<td>1066.5</td>
<td>0.5533</td>
<td></td>
</tr>
<tr>
<td>171.swim</td>
<td>2137.5</td>
<td>0</td>
<td>2137.5</td>
<td>2137.8</td>
<td>1.335</td>
<td>2137.8</td>
<td>0.9999</td>
<td></td>
</tr>
<tr>
<td>183.equake</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>177.mesa</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>188.ammp</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>173.applu</td>
<td>44.74</td>
<td>2354.8</td>
<td>47.10</td>
<td>68.29</td>
<td>0</td>
<td>68.29</td>
<td>0.6897</td>
<td></td>
</tr>
<tr>
<td>103.su2cor</td>
<td>55.49</td>
<td>0.027</td>
<td>55.49</td>
<td>221.94</td>
<td>0.009</td>
<td>221.94</td>
<td>0.2500</td>
<td></td>
</tr>
<tr>
<td>MEAN</td>
<td>687.6</td>
<td></td>
<td></td>
<td>1302.4</td>
<td></td>
<td></td>
<td>0.5280</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.4: Execution frequency of unrolled and cleanup loops: the table enumerates iteration counts of the unrolled and cleanup loops after runtime loop unrolling. The last column lists the ratio of total iterations of Runtime loops over total iterations of Runtime-2 loops.

Predictions and the observed runtime of the application (as shown in Figure 6.12). Runtime-2 performed worse on this benchmark because of its inflexible strategy of choosing unroll factors. An unroll factor of 2 is unsuitable for wupwise since several loops that dominate its execution time have a loop bound of 3. Consequently, if those loops are unrolled twice, the cleanup loop executes once on every invocation of the loop. This leads to a relative degradation in performance. The benchmark highlights the drawbacks of a loop unrolling strategy that selects an unroll factor before examining program values. In stark contrast to the inflexibility displayed by the Runtime-2 technique, Runtime was able to choose an unroll factor of 3 for the dominant loops, thereby avoiding any invocation of the cleanup loops. This advantage in delaying the selection of unroll factors till runtime is reflected in the relative performances of
the two unrolling algorithms – Runtime improves the execution time of \texttt{wupwise} by 18.1% over Runtime-2.

6.9 Conclusion

As documented in this chapter, we presented a transformation technique that exploits the availability of information available only at program runtime. We described the design and implementation of a lightweight value examiner that extracts values of global variables and function parameters when a procedure is invoked. Further, we presented a loop unrolling technique that used the information gathered by the value examining mechanism to unroll loops efficiently and effectively at runtime. Our results on floating-point benchmarks indicated that runtime loop unrolling was successful in reducing observed execution time – on average, the technique reduced observed execution time by 12.5% on average.

The evaluation of our unrolling strategy demonstrates that an optimization technique can be profitably redesigned to take advantage of opportunities present on a runtime compilation environment. In previous chapters, we have described in detail the resource limitations imposed on a JIT due to its invocation during program execution. As a result of those limitations, runtime compilation is considered unsuitable in many program-execution environments. The increase in performance delivered by the runtime unroller emphasizes that runtime optimization techniques can play a part in offsetting the limitations faced by the runtime compiler. Therefore, exploiting these opportunities are critical in making runtime compilation more acceptable for use in a wide variety of applications.
Chapter 7

Conclusion

Runtime compilation has become increasingly popular due to the widespread availability and emergence of Internet-based programs and mobile devices. Therefore, effective strategies for a runtime compiler are critical to good performance for many applications. The research described in this dissertation tackles two contrasting features of online compilation. First, we demonstrate that the compilation-time constraints imposed by the runtime environment need not preclude the choice of a strong optimization algorithm. Our experiments show that by redesigning a critical optimization – global register allocation – for runtime usage, application performance can be significantly improved both over traditional offline strategies as well as techniques designed specifically to reduce compile-time. Second, the runtime compiler also presents unique opportunities not available to an offline compiler. The runtime unrolling mechanism described in the second part of the dissertation shows how an optimization can be tailored to profitably exploit runtime information and consequently improve program performance. In the next few sections, we shall discuss the contributions of this thesis and identify directions for future research.

7.1 Using strong register allocation techniques in a runtime environment

In recent decades, the difference between processor and memory speeds have increased significantly. Therefore, reducing the number of high-latency memory accesses is important in achieving better application performance. To that end, register allocation is a critical transformation that can be utilized by a compiler writer to maintain
program values in registers – the highest and fastest level of the memory hierarchy. Since allocation algorithms designed for offline use tend to consume moderately large amounts of compile timer, they are generally considered too expensive for a runtime compilation environment. However, the work described in this thesis has shown that a careful redesign can significantly reduce the compilation-time requirements of strong register-allocation algorithms with little or no reduction in allocation performance.

We have described the design of two graph-coloring allocation algorithms derived from the Chaitin-Briggs technique: the lossless and the lossy allocators. Both attempt to reduce the time required for the interference graph constructor – the most expensive component of Chaitin-Briggs. While the lossless allocator constructs the same interference graph as Chaitin-Briggs, the lossy allocator explored trading off some degree of precision for allocation speed. Our experiments show that such a tradeoff is beneficial in a runtime compilation context.

Our examination of register allocation techniques opens up two directions that can be explored in future work. In runtime compilation systems, we would like to study hybrid register allocation schemes in which the runtime compiler can choose from a group of different register allocation algorithms based on procedure characteristics. The lossy and lossless allocators significantly increases the spectrum of algorithms that the compiler can select from. The second application of these algorithms lies in an adaptive compilation scenario. Adaptive compilation consists of searching the optimization space by applying different sets of transformations [38]. An increase in the efficiency of applying the transformations can result in a significant reduction in search efficiency. Moreover, the incremental updates to the analyses phases described in this document can be fruitfully used to explore the optimization space more extensively. For these reasons, we would like to observe the behavior of an adaptive search strategy that uses the lossy and lossless algorithms, and the incremental updates described in the document.
7.2 Exploiting runtime value information to conduct loop unrolling

Our exploration of the effectiveness of different optimizations on a runtime compiler yielded that transformation techniques that focus on loops in scientific applications are profitable. Those results led us to examine whether the runtime environment can prove to be advantageous to optimization algorithms on a JIT. For runtime compilation to provide acceptable application performance, we believe that a runtime compiler must exploit information that is unavailable to its offline counterpart. In particular, the presence of program values in an online compilation environment allows the JIT opportunities to emit code that is tailored specifically for the current invocation of the program. To this end, we decided to focus on one optimization – inner loop unrolling – that showed promise in the results obtained by our effectiveness experiments. We described the design and implementation of a runtime value examiner that works in cooperation with a runtime loop unroller. This system does not require additional user input, additional information such as program profile information, or a manual modification of the program. Therefore, it is completely transparent to the user. The runtime unroller transforms loops whose bounds are unknown until runtime. As our experiments have shown, this strategy is both effective and efficient and results in increased program performance. Our focus on exploiting runtime loop bounds demonstrates that a runtime compiler can profitably optimize code that is generally difficult for an offline compiler to do in the absence of contextual information available only at program execution time.

Our exploration of runtime loop unrolling has piqued our interest in how different selection criteria for unroll factors can be redesigned for an online context. In future work, we wish to examine both analytical and adaptive selection algorithms for unroll factors. We are particularly interested in understanding how the extra information provided by our value examiner can be profitably used by these algorithms, and whether the selection techniques can be adapted for efficient runtime usage.
Bibliography


[96] Shlomo Weiss and James E. Smith. A study of scalar compilation techniques


