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.
Abstract

Register allocation is a long studied optimization in compiler construction because it provides great opportunity for improving execution time. Adaptive compilation is a relatively new technique that uses repeated compilation and search to find effective parameters for compiler optimizations. We examine the priority-based graph-coloring register-allocation algorithm in the context of an adaptive compiler. The priority-based algorithm was selected because it is well known, but little information exists on how it should be tuned to produce good results or how it compares with competing algorithms.

We show that adaptive compilation can be used to improve the performance of a priority-based allocator. Aggressive tuning through adaptive compilation enables us to fairly compare against the Chaitin-Briggs algorithm for register allocation. We found the standard priority-based allocator was, on average, 16-9% worse than Chaitin-Briggs. Adaptive compilation enabled the priority-based allocator to close this performance gap and slightly outperform Chaitin-Briggs by an average of 1%.
Acknowledgments

I would like to thank Keith Cooper, my thesis adviser, for his support and encouragement throughout this process. I was fortunate to have such a patient and intelligent guide to help me complete the research. I am also very grateful to the remainder of my committee. Devika Subramanian has taught me so much these past few years. Her insights and passion for learning make her a great mentor. Vivek Sarkar has been a tremendous inspiration. His enthusiasm is infectious and I have greatly benefited from working by his side.

Tim Harvey deserves a special thank you. He worked relentlessly to ensure that I made constant progress. Our frequent conversations were enjoyable and enlightening. I would never have been able to complete this endeavor without his help.

My fellow graduate students provided constant encouragement and support. Jeff Sandoval, Raj Barik, Raghavan Raman, Seth Fogarty, Nathan Tallent, Yi Guo, Mary Fletcher, Ryan Zhang, Anna Youssefi, and Dan Sandler have all made my time here enjoyable.

Dave Streuker, who wrote most of the latex code I used for my thesis.

My girlfriend Dana who: moved to Paris, completed her Master’s degree, did an internship in London, moved back to Houston, and got a job, all in the time it took to complete my thesis. Her support in this time has kept me healthy, happy, and more grateful than ever to have her back in Houston.

Finally I would like to thank my family. They have supported and encouraged me my whole life. My parents, Kathleen and David, and my sister Jessica have all positively shaped my future. I have them to thank for all of the advantages I have enjoyed.
Contents

1 Introduction 1

2 Related work 6
   2.1 Register allocation 6
   2.1.1 Work preceding graph-coloring allocators 7
   2.1.2 Graph-coloring allocators 9
   2.2 Adaptive compilation 14

3 Classic priority-based allocator 19
   3.1 Overview 19
   3.1.1 Live ranges 20
   3.1.2 Priority function 23
   3.1.3 Interference graph 25
   3.1.4 Machine model 26
   3.2 Key data structures 28
   3.2.1 Live ranges 28
   3.2.2 The interference graph 29
   3.3 The priority-based algorithm 32
   3.4 Building live ranges 41
   3.4.1 Data-flow analysis 43
   3.4.2 Building initial live ranges 43
   3.4.3 Local live ranges 47
   3.4.4 Marking loads and stores 47
   3.5 Live-range splitting 49
   3.5.1 The splitting algorithm 50
   3.5.2 Transferring live units between live ranges 51
   3.5.3 Updating live ranges after a split 52
   3.5.4 Updating the constrained lists 54
   3.5.5 Zero-occurrence live ranges 55
   3.6 Rewriting the code 56
   3.6.1 Local allocation and spill-code insertion 56
   3.6.2 Instantiating loads and stores at live range boundaries 57
   3.7 Limiting basic-block size 58
   3.8 Differences between the priority-based and Chaitin-Briggs allocators 58
4 Engineered priority-based allocator
4.1 Introduction ............................................. 60
4.2 Live-range trimming ...................................... 61
4.3 Rematerialization ......................................... 64
   4.3.1 Finding rematerializable values ....................... 65
   4.3.2 Splitting live ranges into rematerializable sections .... 68
   4.3.3 Spilling in the presence of rematerialization ......... 73
4.4 Aggressive code motion of loads and stores ................ 76
   4.4.1 Replacing store/load pairs ........................... 76
   4.4.2 Cleanup ............................................... 80
4.5 Optimistic simplification .................................. 81
4.6 Smart loading into copies .................................. 84
4.7 Limiting the number of splits .............................. 85
4.8 Enhanced register promotion ............................... 87

5 Adaptive priority-based allocator .......................... 89
5.1 Adaptive compilation framework ........................... 89
5.2 Priority-based allocator search space ....................... 91
   5.2.1 Allocation parameters ................................. 92
   5.2.2 Search space size ................................... 97
   5.2.3 Neighborhood structure ............................... 99
5.3 Search algorithm .......................................... 99
   5.3.1 Generic random-restart impatient-hillclimber .......... 100
   5.3.2 Search state ......................................... 101
   5.3.3 Random-restart impatient hillclimber in the adaptive compiler 102

6 Experimental results ...................................... 107
6.1 Classic .................................................. 109
6.2 Engineered ............................................... 110
6.3 Adapted .................................................. 114
   6.3.1 Performance results .................................. 115
   6.3.2 Parameter frequency .................................. 116
   6.3.3 Search efficiency ................................... 123
6.4 Allocation time .......................................... 125
6.5 Summary ................................................ 126

7 Conclusion ................................................. 129
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Live-range example</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>Comparison of live-range definitions</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>Comparison of live-range definitions</td>
<td>24</td>
</tr>
<tr>
<td>3.4</td>
<td>Coarse live ranges</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>Precise live ranges</td>
<td>27</td>
</tr>
<tr>
<td>3.6</td>
<td>Live-range definition (C code)</td>
<td>30</td>
</tr>
<tr>
<td>3.7</td>
<td>Live-unit definition (C code)</td>
<td>31</td>
</tr>
<tr>
<td>3.8</td>
<td>Example of load and store movement</td>
<td>38</td>
</tr>
<tr>
<td>3.9</td>
<td>Example of load and store movement</td>
<td>40</td>
</tr>
<tr>
<td>4.1</td>
<td>Live-range splitting</td>
<td>62</td>
</tr>
<tr>
<td>4.2</td>
<td>Briggs-style rematerialization</td>
<td>66</td>
</tr>
<tr>
<td>4.3</td>
<td>Priority-based allocator rematerialization</td>
<td>67</td>
</tr>
<tr>
<td>4.4</td>
<td>Rematerialization store problem</td>
<td>75</td>
</tr>
<tr>
<td>4.5</td>
<td>Aggressive code motion</td>
<td>77</td>
</tr>
<tr>
<td>4.6</td>
<td>Optimistic simplification motivation</td>
<td>82</td>
</tr>
<tr>
<td>4.7</td>
<td>Smart loading of copies</td>
<td>85</td>
</tr>
<tr>
<td>4.8</td>
<td>Zero-occurrence live ranges</td>
<td>86</td>
</tr>
<tr>
<td>4.9</td>
<td>Live-range trimming to reduce splitting</td>
<td>87</td>
</tr>
<tr>
<td>5.1</td>
<td>Main loop in the adaptive compiler</td>
<td>92</td>
</tr>
<tr>
<td>5.2</td>
<td>Structures representing the current state of a search.</td>
<td>103</td>
</tr>
<tr>
<td>6.1</td>
<td>Performance of classic priority-based allocator</td>
<td>111</td>
</tr>
<tr>
<td>6.2</td>
<td>Performance of engineered priority-based allocator (block size unlimited)</td>
<td>114</td>
</tr>
<tr>
<td>6.3</td>
<td>Performance of engineered priority-based allocator (block size 1)</td>
<td>115</td>
</tr>
<tr>
<td>6.4</td>
<td>Performance of adapted priority-based allocator</td>
<td>117</td>
</tr>
<tr>
<td>6.5</td>
<td>Frequency of registers used for local allocation</td>
<td>121</td>
</tr>
<tr>
<td>6.6</td>
<td>Search descent (rkf45)</td>
<td>124</td>
</tr>
<tr>
<td>6.7</td>
<td>Search descent (applu)</td>
<td>124</td>
</tr>
<tr>
<td>6.8</td>
<td>Allocation time</td>
<td>127</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Priority-based allocation parameters ......................... 97
5.2 Search space size ........................................... 98

6.1 Benchmarks used in collecting experimental results .......... 108
6.2 Parameters used in engineering version benchmarks .......... 112
6.3 Performance of the priority-based allocator ................... 116
6.4 Performance of the priority-based allocator on large routines 118
6.5 Frequency of multi-valued parameter settings .................. 120
6.6 Frequency of boolean-valued parameter settings ............... 122
6.7 Allocation time ............................................. 126
Chapter 1

Introduction

Despite advances in computer architecture, latency to main memory continues to be a major performance barrier. Accessing a value from a register is orders of magnitude faster than going to main memory. Unfortunately, practical considerations, such as switching complexity, limit the number of registers to a small value. Efficiently managing this limited resource is critical to achieving high performance. A compiler is typically responsible for deciding which values will be stored in a register and which will remain in main memory. Finding the perfect allocation of values to registers is an NP-complete problem, so algorithms for register allocation make simplifying assumptions and use heuristics to solve the problem in a reasonable amount of time. Register allocation remains an active area of research because of its great potential to improve performance.

Two well known algorithms for register allocation are the Chaitin-Briggs graph-coloring algorithm and Chow and Hennessy's priority-based algorithm [12, 22]. Although the algorithms are frequently cited as alternate approaches to register allocation, their relative performance is unknown. This thesis describes our recent work on priority-based register allocation. The allocator's performance is compared with a previously written and well tuned Chaitin-Briggs allocator. Both algorithms have been implemented in our research compiler and we show that priority-based allocation is only competitive with Chaitin-Briggs after extensive tuning through adaptive
compilation. In this thesis we describe how to build and tune the allocator, give detailed performance results, and provide the first fair comparison of the two competing algorithms.

Register allocation has been an important optimization since the first compilers were developed. Backus et al. describe the design of the register allocator used in the first FORTRAN compiler [5]. Though the IBM 704 had only 3 registers, considerable effort went into efficiently using those registers. The FORTRAN compiler was introduced in 1957 and many algorithms for register allocation have since been published. These algorithms can be broadly categorized as working over local or global scope. Local algorithms work with only a single basic block at a time, where a basic block is a maximal length of straight line code. In contrast, global algorithms work across an entire procedure. Global algorithms are generally considered to be more powerful.

This thesis focuses on a particular method of global register allocation called the priority-based approach. Priority-based allocation belongs to a group of algorithms known as graph-coloring register-allocation algorithms. These algorithms work by first finding the values, called live ranges, that can not be kept in the same register because their lifetimes overlap. These live ranges are said to interfere. A graph is built with each live range represented by a node and an edge between any live ranges that interfere. Finally, the allocator attempts to color the graph using k colors, where k is the number of machine registers, such that no nodes connected by an edge are given the same color. A successful coloring corresponds to a mapping of live ranges to registers.

The first complete description of a graph-coloring register allocator was given by Chaitin et al. in 1981 [19]. Briggs et al. later improved the design [12, 9]. Chow and Hennessy describe the first priority-based allocator in their 1984 paper [21]. Numerous other methods have been published for register allocation and continue to be published
today. Section 2 gives an overview of how these algorithms have evolved.

The approach taken by the priority-based algorithm is often contrasted with that of Chaitin-Briggs, yet there is no published side-by-side performance comparison. Register allocation is an important optimization, so we felt compelled to find out how the two algorithms compared. Making a fair comparison is difficult, because tuning a register allocator requires substantial effort. We already had a high quality implementation of a Chaitin-Briggs allocator; implementing a priority-based allocator enabled the direct comparison. We used adaptive compilation to automatically tune the priority-based allocator to ensure our inexperience with the algorithm had a minimal impact on the comparison. Our techniques allow us to present a fair performance comparison between two heavily cited algorithms which have so far escaped a true comparison.

Due to the age of the algorithm, there is no modern description of its implementation. This thesis seeks to fill this hole in the literature by describing our implementation of the algorithm using modern analysis techniques such as Single Static Assignment (SSA). We also describe how to integrate the global allocation with a local-allocation algorithm. Our description of the algorithm is more complete than anything else in the literature and should be detailed enough to aid programmers wishing to implement it in a compiler.

In the process of implementing the allocator, we found several areas where we could make some improvements. These changes seek to improve the quality of the allocation without fundamentally changing the algorithm. They include live-range trimming, rematerialization, aggressive code motion of loads and stores, optimistic simplification, and smart loading into copies. Except for rematerialization and smart loading, these changes were proposed after observing the behavior of the allocator and examining its output. They were designed to fix specific problems we saw. Although
they rationalize well on paper, we found they cause little change in actual performance when used in isolation. When used in the correct combinations, they have a greater impact and allow the priority-based allocator to match the performance of Chaitin-Briggs. Finding the correct combinations of settings is left to the adaptive compiler.

One of our initial observations about the priority-based algorithm was that its behavior could be adjusted through numerous parameters. These parameters include the maximum number of instructions in a basic block, the coloring heuristic, the number of registers to reserve for local allocation, the priority function, and the allocation of local registers. The potential to influence allocation quality by tuning these parameters led us to use tools from adaptive compilation [28]. We did not want our inexperience in tuning a priority-based allocator to bias the comparison in favor of Chaitin-Briggs. Previous work by Waterman has shown the effectiveness of adaptive compilation for finding good combinations of parameters in individual compiler optimizations [53]. We present our experience adding the priority-based algorithm to an adaptive compiler and show the effectiveness of an adaptive search for this algorithm. We added several more heuristics to the algorithm, including a different splitting method for live ranges and two new coloring heuristics, and also exposed the improvements discussed previously to the adaptive compiler so that it would have a rich set of parameters for improving performance.

In our early experiments with the adaptive compiler, we saw that the allocator would need different parameter settings for each function in a program. This need was primarily due to the number of registers that should be reserved for local allocation. Our infrastructure was only able to set parameters at the level of an entire program. We developed an algorithm that we could use to search for different compiler settings for each function without increasing the number of compile/execute iterations over the traditional search method. With this new algorithm we successfully used adaptive
compilation to tune the allocator.

With these tools in hand, we were ready to answer the question that was the initial motivation for this study: Chaitin-Briggs vs. priority-based, which algorithm is better? We are able, for the first time, to show a fair comparison since both algorithms have been implemented in the same compiler. The Chaitin-Briggs version is an existing implementation in our compiler infrastructure. We show a comparison between this algorithm and the versions of the priority-based algorithm presented in this thesis. The results show that Chaitin-Briggs beats the untuned priority-based allocator by an average of 16–9%, depending on the number of registers available. As the number of registers decreases, its performance relative to Chaitin-Briggs improves. The engineered version is only slightly worse Chaitin-Briggs – except when using few registers where it is 14% worse on average. The adapted version does slightly better than Chaitin-Briggs in all register settings.

In our quest to answer our initial question we ended up with five contributions to the research community. This thesis describes these contributions:

1. Implementation of the priority-based register-allocation algorithm as first described by Chow and Hennessy.

2. Engineered version of the priority-based algorithm aimed at improving the quality of allocation.

3. Tuned version of the priority-based allocator produced by running an adaptive-compilation search.

4. Algorithm for running an adaptive compiler so that each individual function in a program is searched in unison.

5. Direct comparison of the priority-based algorithm with the Chaitin-Briggs algorithm for register allocation.
Chapter 2

Related work

This thesis is a synthesis of two different research areas: register allocation and adaptive compilation. Register allocation has been studied extensively in the literature, while adaptive compilation is a relatively new area of study. This thesis is one of few works that uses prior research in both areas to tackle the problem of register allocation.

2.1 Register allocation

Techniques for register allocation can be roughly grouped into two categories: local and global. Local allocators work at the level of a basic block, while global allocators work over an entire procedure. Even in the early compilers, particular attention was paid to the efficient use of registers. It is tempting to think that early attempts at register allocation focused on local allocation, and only after that problem was satisfactorily solved did work begin on larger scopes. However, the literature shows this clearly is not the case. From the beginning, compilers were concerned with achieving the best possible allocation. As the field matured a variety of methods were proposed for both local and global allocation. This section reviews the evolution of register allocation techniques up to the latest improvements to graph coloring allocators. Register allocation continues to be an active area of research, but developments not involving graph coloring allocators are omitted from this thesis.
2.1.1 Work preceding graph-coloring allocators

Backus et al. describe their method of register allocation in the first FORTRAN compiler [5]. Their method was quite sophisticated and involved a Monte-Carlo simulation of the branches in the program to estimate the frequency of the execution paths. The simulation could be improved by frequency annotations from the programmer. Register allocation was done by regions, where a region could contain at most one loop. The regions were allocated in order of frequency so that the most common path would receive the best allocation. Information about the allocation was recorded at region and basic-block boundaries so that the correct loads and stores could be inserted at the boundaries.

Other early work on register allocation focuses on local allocation. An often cited method for local allocation was invented by Best in the mid 1950s, according to Backus [4]. The algorithm was rediscovered by Belady in a study of page replacement algorithms [7]. He proposed the algorithm as an offline page replacement algorithm to measure the effectiveness of competing online algorithms. Best’s method provides a way to choose which value to spill when we have to spill during local allocation. To select a value, we look at all the values currently in a register and spill the one that is referenced farthest in the future. This policy will produce an allocation that is optimal in the number of loads required. It does not explicitly deal with how to minimize the total number of memory operations.

Horowitz et al. give a method for local allocation of index registers that they prove is optimal in the number of memory references for straight line code [35]. They solve the problem by using a directed graph to model the possible contents of the registers at each instruction in the program. Each node in the graph is a possible mapping of values to registers. The edges are weighted by the cost of getting from one configuration to another, where the cost is based on how many registers where
changed in the move and, whether or not the values in the changed registers have been modified. An optimal solution to the register allocation problem is a shortest path in the graph from the starting to an ending configuration. They recognize that the graph can become very large, and give an algorithm to find the shortest path without generating the entire graph. The same idea was used by Hsu et al. in 1989 to give an optimal local algorithm for allocating general purpose registers [36].

An early attempt at global allocation is described by Day [31]. Day presents two methods for global allocation of registers. The first assigns a single data item to a register for an entire region. This is the same method for global allocation used by Lowry and Medlock in the FORTRAN H compiler [45]. The second method allows for multiple data items to be assigned to a register as long as they do not interfere. He shows how to formulate both methods as integer programming problems and provides three ways to solve these problems: one optimal and two heuristics. In order to formulate the problem he needs to assigns profits for keeping an item in a register. He suggests using control flow information combined with usage and definition counts to estimate the profit of keeping an item in a register. His suggestion for profit is very similar to what Chow and Hennessy use as the priority function in their allocator [21]. His optimal solution maximizes the total profit.

Another early method for global allocation is given by Beatty [6]. He proposes an algorithm that integrates local allocation with global allocation and assignment. He builds on the earlier work by Day who suggested assigning multiple global values that do not interfere to a single register. Beatty attempts to improve on Day’s suggestion by integrating high quality local allocation with global allocation across an entire region. His key technique is to separate local allocation from local assignment. He begins with a phase of local allocation, which marks the values that should be given a register only at a basic block level. He follows that by a global allocation and
assignment and then finishes with local assignment. The separate passes allow Beatty to make local allocation decisions without overly constraining the global allocator by premature assignment. He can then use global assignment to assign good registers for the values selected by local allocation. The difficult part for Beatty is how to represent the interaction between local allocation and global assignment. He struggled with finding the correct mechanism to represent the interaction and it took a full man-year of time to write the allocator. Reviewing his methods suggests the need for a strong conceptual framework, such as graph coloring, for global allocation and assignment.

2.1.2 Graph-coloring allocators

The first global graph-coloring register allocator was described by Chaitin et al. in a 1981 paper [19] and refined in a 1982 publication [18]. In this paper Chaitin describes how to construct the interference graph, perform copy coalescing (called subsumption in the paper), and choose values for spilling. The algorithm works by building an interference graph where each node is a virtual register and the edges represent interferences. Two virtual registers interfere if one is live at the definition point of the other. Copies are coalesced and the interference graph is rebuilt. This iterates until no more copies can be coalesced. Allocation continues by removing unconstrained nodes from the graph, those with fewer than \( k \) neighbors, because there will always be a color available for those nodes. If the algorithm reaches a point where no more nodes can be removed, it chooses a node to spill and rewrites the code with the spills inserted. If any node has to be spilled, the interference graph is rebuilt and colored again. The algorithm terminates when the graph can be colored without spilling any nodes.

Two years after Chaitin described his graph coloring allocator, Chow and Hennessy proposed the priority-based approach to register allocation that is the focus of
this thesis [21, 22]. Larus and Hilfinger describe their implementation of the priority-based allocator in the SPUR Lisp compiler [44]. They were the first to propose limiting the basic block size in order to obtain a more precise interference graph. Also, they used a machine model where values can only be referenced from registers; Chow and Hennessy assume that instructions can reference values directly from memory. Larus and Hilfinger report achieving a good allocation, but do not provide any comparison with other global techniques.

Limited information has been published about how to improve the standard priority-based allocator. Chow shows how to extend the priority-based allocator to perform inter-procedural allocation [20]. Sorkin makes some suggestions for changing the priority-based allocator [49], but implementing his changes would fundamentally change the allocator. Sorkin suggests getting rid of the interference graph and using an alternate definition of coloring in which a live range can be colored if there is some color available in each of its blocks; it need not have a single color over the entire live range. The problem with this definition is that you must have either have a register swap instruction in your machine to handle cases where live ranges need to swap colors at some basic block boundary, or use a series of xor operations to swap the values. These changes were not implemented in this thesis, because they change the algorithm so that it is no longer a true priority-based graph coloring allocator. Finally, Kim et al. show how to adapt the priority-based allocator for architectures with predicated execution [37]. They modify the priority function to account for predicted usage and also show that increasing the granularity of the interference graph improves the quality of allocation.

In contrast to the priority-based allocator, numerous improvements have been suggested for the standard Chaitin allocator. Briggs et al. gave several improvements, including optimistic coloring [10], and rematerialization [11]. These topics are covered
in detail in Briggs's thesis [9] and summarized elsewhere [12]. Optimistic coloring delays spilling decisions until the coloring phase where the algorithm can tell precisely whether or not each node gets a color. Rematerialization finds values that can be recomputed instead of spilled and uses this knowledge to recomputed the value instead of loading it from memory if it can not be kept in a register. These two ideas were the inspiration for the optimistic simplification and rematerialization improvements added to our priority-based allocator. Taken together, these improvements form the basic Chaitin-Briggs allocator which is the baseline allocator for comparisons in thesis.

Briggs compares allocation time between the Chaitin-Briggs allocator and the priority-based allocator of Chow and Hennessy [9]. He notes that the comparison is not ideal because the allocators are implemented in different compilers. He compares the allocation time over a set of 181 FORTRAN routines. He finds that the priority-based allocator is faster than the Chaitin-Briggs allocator when used with small routines, but Chaitin-Briggs is faster for large routines. His evidence suggests an actual running time of $O(n^2)$ for the priority-based allocator and $O(n)$ for the Chaitin-Briggs allocator, where $n$ is the size of the routine.

Bernstein et al. describe three improvements to the Chaitin allocator: multiple heuristics for determining which node to spill, a greedy heuristic for ordering the removal of unconstrained nodes, and local cleanup after spilling to ameliorate the negative effects of “spill everywhere” behavior. Of these three ideas, the multiple heuristics for selecting nodes to spill is the most interesting in respect to our work, because it foreshadows the idea of adaptive register allocation. Bernstein et al. developed three heuristics for estimating the cost of spilling a node. These heuristics are used to select which node to spill when there are no more unconstrained nodes to remove from the graph. They attempt to color the graph three times, once using each heuristic. The heuristic that produces the smallest spill cost is then chosen as the “real” heuristic for
spilling. The algorithm spills according to this heuristic and the graph is rebuilt and coloring is attempted once again. They acknowledge that no heuristic is dominant in terms of always producing the best results and that the correct heuristic depends on the code being allocated. These ideas are very similar to what we accomplish with adaptive compilation, but we are working on a much larger scale with many heuristics instead of only three. They also mention that they compare against a priority-based allocator, however, they did not implement splitting, which makes the comparison unfair.

Global spilling of a live range is a known deficiency of the Chaitin allocator. Techniques for spilling a live range only in the areas where there is high register pressure have been proposed by several authors. Bergner et al. describe a method for reducing spill code by only spilling a live range over parts of the code, rather than the global method used by Chaitin [8]. They follow the optimistic coloring used by Briggs, but when they can not find a color for a live range, they spill only part of the live range. Instead of blindly spilling, they look at all interferences for the uncolorable node and remove all the edges for a specific color. Spill code must be inserted for the live range for any references that overlap with the live ranges whose edges were removed. The live range can then be assigned that color, since it will not interfere with the other live ranges of the same color. This process is repeated for any live range which can not be given a color in the select phase.

Cooper and Simpson give a method for reducing spill code using live-range splitting [27]. When the allocator has to spill, they see if it is instead profitable to split the live range by keeping it in memory for some part of its lifetime. A live range can be split if it completely contains another live range, because it can be kept in memory during the other live range’s lifetime. It is profitable to split a live range if it requires fewer loads and stores to split the live range than to spill everywhere.
When it comes time to spill, they look to see if either a set of live ranges already assigned the same color can be spilled across the live range, or if the current live range can be split across some other live ranges that are assigned the same color. If it is possible to split so that the live range can be given a color, and it is cheaper than spilling entirely, the live range is split. This method for splitting can be combined with Bergner’s interference region splitting because Cooper and Simpson first look to see if a live range can be split, and if not they spill, at which time Bergner could kick in and only spill in a region. Cooper and Simpson only split live ranges that contain other live ranges, where Bergner will split in regions of interference without requiring containment.

Callahan and Koblenz describe an enhancement to the Chaitin allocator that takes the structure of the program into account when making spilling decisions [13]. They partition the control flow graph into sets of basic blocks, called tiles, and then build an interference graph and color each tile separately. The tiles represent the control flow in the program, for example, the loop structure. They then combine the separate allocation of the tiles to get a complete allocation for the program. They keep track of spills and conflicts in each region, which allows them to place spill code in better positions, such as placing the spill at a tile boundaries when a variable will not be allocated in the sub-tile. Using the tiles also allows them to do splitting similar to Chow and Hennessy, except that the splits are at predefined points corresponding to the tiles. Unlike Chaitin they can keep a single live range in different physical registers in different parts of the code and connect those parts with a copy. Cooper et al.compare the performance of the Chaitin-Briggs and Callahan-Koblenz allocators in the LLVM compiler [24], and Eckhardt provides a detailed description of the algorithm in his thesis [32].
2.2 Adaptive compilation

Adaptive compilation, also known as iterative compilation or feedback-directed optimization, is a general method that uses repeated compilation and information feedback to optimize code for some desired metric. We follow the categorization given by Sandoval [48], and place adaptive compilation systems into three broad categories: systems designed to order a series of optimization passes, systems designed to tune a set of compiler flags, and systems designed to improve a single compiler optimization. The work presented in thesis uses adaptive compilation to improve the specific optimization of register allocation. In this section we survey several works in all three categories, beginning with systems designed to improve a single compiler optimization.

Stephenson et al. use machine learning techniques to fine tune various compiler heuristics [50, 51]. They use genetic algorithms to improve heuristics in a variety of optimizations, including: hyperblock formation for predicated execution, register allocation, and data prefetching. Their work in register allocation is particularly interesting because they tune the priority function from Chow and Hennessy's priority-based allocator. They report a mean speedup of 8% obtained by searching for a specific priority function for each benchmark. They also use their method to search for a good general purpose priority function that can be used across all benchmarks. A good general purpose priority function is more difficult for them to find; they report a mean speedup of 2% when using their general purpose function.

Cavazos, Moss and O'Boyle apply machine learning to register allocation in the Jikes RVM [14]. They present the notion of Hybrid Optimizations, where the compiler uses a heuristic to select between two different algorithms for register allocation. They identify a set of features from the input code and then use machine learning to build
a heuristic based on those features. The heuristic indicates which register allocation algorithm is more profitable to use for a particular function. They implemented their method in Jikes to select between the linear scan and Chaitin-Briggs algorithms. They ran a number of benchmarks and report an average improvement of 3% in total execution time over linear scan, and a 9% improvement over Chaitin-Briggs.

Waterman applied adaptive compilation techniques to the specific compiler optimization of procedure inlining [53]. He describes an implementation of procedure inlining that exposes inlining decisions through a set of command line parameters. He uses adaptive compilation to find parameter settings that encourage the inliner to make good decisions and produce quality code. Waterman shows that exposing more decisions to the adaptive compiler through parameters increases its effectiveness in finding good solutions. He reports execution time reduction of up to 30% using the adaptive inliner.

Cavazos and O’Boyle use adaptive compilation to tune an inlining heuristic in the Jikes RVM [15]. They take an existing heuristic with five parameters and use a genetic algorithm to tune the values of those parameters. The genetic algorithm uses a fitness function that assigns fitness to a group of parameter settings based on performance across a set of training benchmarks. The parameter values learned in this training set are then used to compile a set of test benchmarks. They report an average reduction in total execution time of 37% on the test benchmarks using their adaptively tuned heuristic.

Early work in using adaptive compilation for optimization pass ordering was done by Cooper, Schielke, and Subramanian, who used genetic algorithms to find sequences of compiler optimizations to reduce code size [26]. They report a reduction in code size between 20-80% for a variety of benchmarks. A single sequence was used for all files in a program. They also experimented with several benchmarks in searching for
a unique sequence for each file in the benchmark, but found it had little improvement over the sequence found for the whole program.

Cooper et al. describe their experience exploring the search space of compilation sequences [2, 25]. They give results for exhaustively enumerating several search spaces of sequences of length 10 chosen from 5 transformations. They show that the search spaces have many local minimum, which makes them amenable to search by a random-restart hillclimber. They compare the effectiveness of several search algorithms in a larger search space of sequences of length 10 chosen from 16 transformations. They report results for searching with genetic algorithms, random-restart impatient hillclimbers, and a greedy construction method. They note that all the techniques consistently find improvements to the code. They observe that the impatient hillclimbers can find solutions with fewer evaluations than the genetic algorithms, and the solutions found are nearly as good as those found by genetic algorithms.

Kulkarni et al. describe VISTA, an interactive compilation system that uses search to find good sequences of compiler optimizations [41, 40]. They use genetic algorithms as their method of search. VISTA is able to search for a different sequence for each function in a benchmark and they found that each function often needed significantly different optimization sequences. They also worked to reduce the search time in multiple ways, such as detecting redundant sequences, identical code, and equivalent but not identical code.

Agakov et al. use machine learning techniques to focus the search of an adaptive compiler [1]. Their method is independent of the search space, but they show results for reducing the search time to find good compiler optimization sequences. They model what a good sequence looks like to guide the search towards those sequences. They proposed two models: an Independent Identically Distributed (IID) model which assumes that the occurrence of an optimization pass in the sequence
is independent of all other passes in the string, and a Markov model which assumes that the probability a pass occurs in a sequence depends only on the predecessor pass in the sequence. They then used machine learning techniques to learn these models from a small training set. They use the learned models to bias the search towards good sequences. They report that using the models to guide the search helps to find good sequences quickly, with an average speedup of 22% after only two iterations of search.

Kulkarni et al. describe a method for exhaustive exploration of the search space for compiler optimization passes [42]. They find that the search space can be pruned to allow exhaustive exploration. They use two methods for pruning the search space: dormant phase detection, and duplicate function instance detection. Dormant phases are the passes that do not change the code. Once a dormant phase is detected, the search down that sequence path can be pruned. Duplicate function instances occur when two sequences of passes generate the same code. They are detected by hashing several aspects of the optimized code. Once duplicate function instances are detected, the search paths for the two sequences that generated the instances can be merged. Using these two techniques, they were able to exhaustively search the pass ordering space for all but two of the 111 functions in their benchmarks.

Kulkarni et al. extend their earlier work on exhaustive exploration of the optimization pass search space to use dynamic execution estimates to evaluate the quality of a pass ordering [43]. They use a simulator for an embedded processor to collect dynamic information, so it is very important for them to reduce the number of times a function is measured. They use a combination of dynamic execution counts on the basic blocks and static estimates of run time for instructions in a block to produce the dynamic estimates. They only run a function through the simulator if the optimizations have produced a previously unseen control flow graph. Once the function is run
through the simulator, they know precisely how many times each block is executed and use that information to weight the static estimates of run times within the basic block.

Compiler flag tuning is traditionally done by hand, but several researchers have applied adaptive techniques to attack this problem. Chow and Wu describe a method for finding combinations of compiler switches to improve execution time in an IA64 research compiler [23]. Their method applies "fractional factorial design" to reduce the number of combinations that must be tested. They begin with a small set of combinations to test. A program is compiled with all of these combinations and the performance is measured. The results of these experiments are used to estimate the effects of the switches and the interactions between them. These estimates are used to determine whether any future combinations need to be tested. With their method, they are able to estimate the performance improvements due to various compiler switches without exhaustive testing of all combinations of parameters.

Triantafyllis et al. describe a method for adaptively tuning compiler parameters with a focus on quick compilation time [52]. They search for a set of parameters that minimizes execution time in a compiler for the Intel Itanium processor. To reduce compile time, they limit the search to configurations estimated to be good candidates for exploration. These candidates are organized into a tree, which is searched at compile time. The search proceeds by testing all configurations at a given depth and greedily selecting the best at that depth for further exploration while pruning all other paths leading from that depth. They further reduce execution time by using a static estimate of the performance gained from a set of parameters. They report an average speedup of 5.3% on a series of benchmarks at an average cost of 88.4% increased compile times.
Chapter 3

Classic priority-based allocator

This chapter presents the priority-based approach to global register allocation invented by Chow and Hennessy [22, 21]. The algorithm in this thesis is based on these previous publications, as well as the description of the implementation in the SPUR Lisp compiler by Larus and Hilfinger [44]. Our implementation closely follows the original algorithm, but is presented here in greater detail. We deviate from the original in two main ways: our definition of live ranges and use of Static Single Assignment Form for analysis. The algorithm, its data structures, and our differences are explained in detail below.

3.1 Overview

The priority-based algorithm maps the problem of register allocation to that of finding a $k$ coloring of a graph, where $k$ is the number of machine registers available to the allocator. The algorithm works by first finding all of the live ranges in the program. A live range is a set of program points over which a variable is live. Once the live ranges are found, an interference graph is built which models the allocation constraints. The nodes of the graph are live ranges and an edge exists between any two live ranges that can not be kept in the same register because they are both live at the same point in a program. Each live range is assigned a priority to indicate the importance of keeping it in a register.
The graph is colored using $k$ colors. Live ranges in the graph are assigned colors in priority order. If a live range becomes uncolorable because all of the available colors have been assigned to its neighbors, then the live range is split in an attempt to reduce its number of interferences. Splitting creates new live ranges which are inserted into the graph. If a live range cannot be further split, it is removed from the graph and will not be given a register. Allocation continues until all live ranges have either been assigned a register or removed from the graph. The next sections discusses live ranges and the interference graph in detail.

3.1.1 Live ranges

A live range is a set of program points over which a variable is live. A variable $v$ is live at a point $p$ if it is used at $p$ or there is a definition free path leading from $p$ that reaches a use of $v$. Every name in the original program that is both defined and used belongs to at least one live range. A name may belong to more than one live range if there are disjoint regions where the variable is live. An example program and its live ranges is shown in Figure 3.1. Part (a) shows the original program and Part (b) shows the same code with variable names rewritten with live range names. The bottom block shows how a single source-variable can be mapped to two different live ranges. The definition of $y$ in the bottom block begins a new live range (D) because it kills – that is, redefines – the earlier definition of $y$ (live range B) in the top block. In contrast, the definition of $x$ in the right block does not start a new live range because the definition of $x$ at the top is still live; it reaches a use in the bottom block. The algorithm for computing live ranges is discussed in Section 3.4.

The priority-based algorithm uses an imprecise definition of live ranges. A live range is a set of blocks rather than a set of program points. For example, in Figure 3.1, the live ranges for A and B consist of all four blocks, and the live ranges C, D, E, and
Figure 3.1: Live-range example. Part (a) shows the original program, Part (b) shows the code rewritten with live range names.

F are the single basic blocks in which they occur. A precise definition of live ranges would specify that A and B end at the first instruction of the bottom block and do not overlap with live ranges C, D, and E. Each block that is part of the live range has some associated information such as the number of uses and definitions in that block. The block and its associated information is called a live unit.

Although we build live ranges at the basic block level, our definition of live ranges differs from the one given by Chow and Hennessy. Their original definition does not
allow for a source name to map to more than one live range. Instead, they have a
one-to-one mapping between live ranges and source names. They build live ranges
using live and reaching sets, but we build live ranges using Static Single Assignment
Form (SSA). Their live ranges contain any blocks where the original variable is both
live and reached by some definition or use. This allows for a single live range to
contain several non-contiguous sets of blocks. We use SSA to separate a single source
name into possibly multiple live ranges each of which is a contiguous set of blocks.

Chow and Hennessy trade a reduction in the number of nodes in the interference
graph for a loss of precision. Our definition allows for more precise interferences at
the cost of more nodes in interference graph. We use our definition of live ranges
because we are interested in quality of allocation rather than fast compilation time.
The increased precision of the interference graph allows us to produce a better alloca­
tion, while remaining in the framework of a priority-based allocator. An example
of the differences is show in Figure 3.2. The interference graph created by using our
definition of live ranges is 2-colorable, while the one created from the Chow and Hen­
nessy definition is not 2-colorable; the live range for x has not been split even though
it has two disjoint definitions.

Although not intuitively obvious, there are some cases where using the classic
definition of live ranges can produce a graph with fewer interferences. This happens
when a live range has a use followed by a definition in the same basic block. In this
case, we consider the second definition to start a new live range, while Chow and
Hennessy call it the same live range. The additional live range increases the number
of interferences in that block. Figure 3.3 shows a concrete example.

Even in light of this example, we use our live range definition for three reasons.
First, we want to expose more choices to the allocator, so we aggressively split live
ranges into their separate pieces. Second, we expect basic blocks to be small so the
Figure 3.2: Comparison of the live-range definitions used by Chow and Hennessy (right) and this thesis (left). The definition used in this thesis splits the x live range into two pieces, which creates an interference graph that is 2-colorable. The classic definition of live ranges does not split x and its interference graph is not 2-colorable.

cases where our definition hurts us should be reduced, since it is only a problem where the definition and use are in the same block. Finally, after running the preliminary optimization passes, there are no cases in any of our benchmark programs where this harmful situation occurs.

3.1.2 Priority function

The priority-based algorithm assigns a priority to each live range indicating how important it is to keep the live range in a register. The priority is based on how many loads and stores will be saved if the live range is kept in a register instead of memory. The priority function is defined by equations 3.1, 3.2, and 3.3. Priority is computed by summing the individual priorities of all of the live units in the live range and
then normalizing by the number of live units. The motivation behind normalizing is to assign smaller priorities to larger live ranges because they tie up a register over a larger span of blocks. Thus for two live ranges with similar unit priorities but different sizes, the priority function assigns higher priority to the smaller live range. The unit priorities are defined by equation 3.2, and are determined by the number of loads and stores saved if the live range is given a register, minus the number of loads and stores needed if the unit is a boundary of the live range. Section 3.4.4 discusses how to compute when loads and stores are needed at the boundary. The priority of a unit is weighted by the loop nesting depth of that unit so that more frequently executed units contribute to a higher priority. We follow the example of Briggs and assume
that each loop executes 10 times [9].

\[
\text{Priority}(lr) = \sum_{lr.\text{units}} \frac{P(\text{unit})}{|lr.\text{units}|} 
\tag{3.1}
\]

\[
P(\text{unit}) = (\text{unit.uses} + \text{unit.defs} - \text{Moves(\text{unit})}) \times 10^{\text{unit.loop.depth}} 
\tag{3.2}
\]

\[
\text{Moves(\text{unit})} = \begin{cases} 
1 & \text{unit needs a load or a store, but not both} \\
2 & \text{unit needs both a load and a store} \\
0 & \text{otherwise}
\end{cases} 
\tag{3.3}
\]

### 3.1.3 Interference graph

Once live ranges are found, the algorithm builds an interference graph that contains a node for every live range. An edge is added between any two live ranges that span the same block to signify that they cannot be kept in the same register.

Figure 3.4 shows the interference graph for the live ranges in Figure 3.1. The live ranges A and B interfere with all other live ranges; C, D, and E interfere with each other; E interferes with only A and B.

The size of a basic block has a direct impact on the number of edges in the interference graph, because live ranges are defined over sets of basic blocks. Splitting a large basic block into several smaller blocks can reduce the number of edges if two live ranges that previously shared the larger block no longer overlap. Figure 3.5 shows the effects of splitting basic blocks. In this case, we limit the block to a single instruction. Limiting blocks to one instruction gives statement-level granularity of interferences. The increased granularity can make the graph easier to color, but it also increases the size of several data structures associated with a live range. Larus and Hilfinger were the first to suggest limiting the size of basic blocks to improve the quality of allocation [44]. They experimented with block sizes of 2, 5, 10, 20, and
Figure 3.4: Control-flow graph with the code in the live-range namespace (left). The interference graph for this code (right). This graph is not 3-colorable.

100. They found that smaller block sizes generally result in better allocation. We discuss how to choose the limit for basic-block size in Chapter 5 by using an adaptive compiler to find the best size. The effect on the quality of allocation of limiting block size is discussed in Chapter 6.

3.1.4 Machine model

Chow and Hennessy assume a machine model where all values have a home location in memory and instructions can operate directly on values in memory. In
Figure 3.5: Control-flow graph with the code in the live-range namespace with blocks restricted to contain on instruction (left). The interference graph for this code (right). This graph is 3-colorable.

their model, register allocation is only an optimization, not a necessity. In contrast, our intermediate representation is a low level RISC-like assembly language where all operands must be in registers. We begin with an unlimited number of virtual registers and map them to the specified number of machine registers. This is the same model used by Larus and Hilfinger in their implementation of the priority-based allocator. We run the allocator for a specified number of machine registers, and we must reserve
at least two registers for generating spill code for any unallocated live ranges. Since the algorithm does not iterate, these registers are reserved before allocation begins. Thus, we actually color the interference graph with $k - r$ colors, where $r$ is the number of registers reserved for spilling and local allocation.

3.2 Key data structures

The two most important data structures used in the priority-based algorithm are the interference graph and the live-range representation. The design and contents of these structures are discussed below.

3.2.1 Live ranges

The live range structure is used throughout allocation. It is the basic unit of allocation and contains information about the structure of the live range as well as the type of value it represents (so that the correct loads and stores can be generated).

Figure 3.6 shows the basic structure of a live range. The blocks field is a bit vector indicating which basic blocks are part of the live range. We use a unique integer to identify each basic block. Using the bit vector makes it easy to determine when two live ranges occupy the same basic block by taking the intersection of the blocks sets. Each live range also contains a list of live units which are the wrapper structures for each basic block in the live range. The live unit contains a pointer to the basic block along with some extra information such as the number of uses and defs in the block for that live range, a flag to indicate that the first reference of the live range in that block is a definition, and whether or not the live range is live on exit from that block. The structure for a live unit is shown in Figure 3.7.

Each live range also holds the information needed for assigning colors to the live
range. The forbidden set is all of the colors which cannot be assigned to the live range because they have already been assigned to one of its neighbors in the interference graph. This set is also used to decide when a live range must be split because it has no more colors available.

A live range may be split several times, creating new live ranges in the process. Splitting complicates some of the bookkeeping because the number of live ranges is not fixed. When splitting a live range, we found it useful to track the original live range at each split. We call the set of live ranges that were originally part of the same live range a family. The live units originally assigned to a live range may be distributed to other members of the family in the course of the algorithm. Section 3.5 describes the splitting algorithm in detail.

Some live-range data is shared across a single family. The primary examples are the blockmap and unitmap. The blockmap maps a basic block to the live range in the family that owns that basic block. Initially, all blocks in the live range are mapped to the original live range. This map is updated as the live range gets split. It is useful in a variety of places, but the main use is for mapping an occurrence of a variable to a color when rewriting the code, as discussed in Section 3.6. The unitmap maps basic blocks to live units and is used primarily for fast access to a live unit given its basic block. Since a live unit is only assigned to one live range at a time, this data can be shared among all live ranges in the same family.

3.2.2 The interference graph

The interference graph is stored implicitly across all live ranges, rather than being kept in one central structure. Each live range maintains a set of live ranges with which it interferes, called the interferences field in Figure 3.6. A live range may change its interference set many times over the course of the algorithm as it or its
struct LiveRange
{
    /* fields unique to an individual live range */
    BitVectorSet blocks; /* basic blocks making up this LR */
    List<LiveUnit*> units; /* live units making up this LR */
    BitVectorSet forbidden; /* forbidden colors for this LR */
    Set<LiveRange*> interferences; /* set of live range interferences */
    Color color; /* color assigned to this LR */
    Def_Type type; /* value type (float, double, int, etc) */
    RegisterClass rc; /* allowed class of registers */
    int id; /* unique id for this live range */
    int orig_lrid; /* original variable for this live range */
    float priority; /* priority for this to be in a register */
    bool is_candidate; /* is it considered for allocation */

    /* fields shared by all live ranges in a single family */
    /* maps from block --> live range */
    Map<BasicBlock*, LiveRange*> blockmap;

    /* keeps track of all the live ranges in the family */
    List<LiveRange*> splits;

    /* maps from block --> live unit for that block */
    Map<BasicBlock*, LiveUnit*> unitmap;

    /* fields used for engineering enhancements */
    bool rematerializable;
    Operation* remat_op;
    int num_colored_neighbors;
    bool is_local;
    int simplified_neighbor_count;
    bool is_simplified;
}

Figure 3.6: Live-range definition (C code)
neighbors are split. The structure used for storing interferences must be chosen with care. It should efficiently support all of the following operations: insertion, deletion, iteration, membership test, and intersection.

Ideally, the insertion, deletion, and membership test would take constant time, and iteration would take time proportional to the number of items in the set. The number of interferences can be quite large, so the structure should be space efficient. The total number of live ranges is not fixed over the course of the algorithm, which makes the choice of data structure more difficult. We experimented with using linked lists and sorted sets for storing interferences before creating our own data structure that we call lazy sets. Lazy sets use a bit vector to note membership and a linked list for iteration. Insertion updates the bit vector and adds to the list. Deletion is lazy and only updates the bit vector. The element will be removed from the linked list the next time we iterate over the set. Iteration walks over the linked list, deleting any element that is not present in the bit vector. This design supports constant time insertion, deletion and membership test. Iteration will be proportional to the number of elements except for the first iteration after an item has been deleted from
a set. The bit vectors are a space efficient representation and can be expanded as the number of live ranges grow.

### 3.3 The priority-based algorithm

The priority-based register-allocation procedure is shown in Algorithm 1. The algorithm works by first building the initial live ranges based on the input program. The live ranges are then separated into two groups based on the number of interferences. Any live range with fewer than $k$ interferences, where $k$ is the number of colors, can always be assigned a color because there will be at least one color not assigned to one of its neighbors. These live ranges are called *unconstrained*. The remaining live ranges are added to the *constrained* set. We then assign colors to the constrained live ranges in priority order. At each step, we compute the priority for any live ranges whose priority is unknown. A live range has an unknown priority either because this is the first time we have seen it in the constrained set, or we have previously split the live range in which case the priority needs to be recomputed because the contents of the live range have changed. Any live range that is completely uncolorable because there is no color available in any of the its blocks is removed from the interference graph. These live ranges will have spill code inserted for all their definitions and uses during the rewriting phase discussed in Section 3.6.

The live range with the highest priority is selected and removed from the constrained set. It is assigned a color and the color is added to the *forbidden* sets of all its neighbors. If any of these sets now contain all of the colors, that live range will have to be split. Splitting a live range will create new live ranges and possibly shuffle some live ranges between the constrained and unconstrained sets. After the neighbors have been split, the selection process repeats, and a new top-priority live
range is assigned a color. This continues until all of the constrained live ranges have been assigned colors or removed from the interference graph. Finally, all remaining unconstrained live ranges are assigned a color.

**Algorithm 1**: Priority-based register allocation

| Data: constrained, unconstrained - sets of live ranges |
| Data: k - number of colors |
| live.ranges ← BuildInitialLiveRanges() |
| constrained ← \{lr ∈ live.ranges : |lr.interferences| ≥ k\} |
| unconstrained ← \{lr ∈ live.ranges : |lr.interferences| < k\} |
| while constrained ≠ 0 do |
| for lr ∈ constrained do |
| compute priority for lr if not previously computed |
| if lr.priority ≤ 0 then |
| mark lr to be spilled |
| DeleteFromInterferenceGraph(lr, constrained, unconstrained) |
| end |
| end |
| choose live range with maximum priority, lr* ∈ constrained |
| AssignColor(lr*) |
| SplitNeighbors(lr*, constrained, unconstrained) |
| end |
| for lr ∈ unconstrained do |
| AssignColor(lr) |
| end |

A key step in the algorithm is splitting the neighbors of the highest-priority live range after it has been assigned a color. Splitting is done using Algorithm 2. A worklist is initialized to contain all neighbors of the top live range. Each element in the worklist is processed to see if it needs to be split because its forbidden set has become completely full. If the forbidden set is full, we first check to see if the entire live range is uncolorable. If the live range contains at least one block where there is a color available, we attempt to split the live range. Splitting produces a new live range lr' containing a subset of the blocks from the live range being split. The splitting algorithm constructs the new live range so that it is guaranteed to be colorable. The
details of splitting are discussed in Section 3.5. Splitting the live range changes its
number of interferences, which may make an unconstrained live-range constrained, or
vice versa. Also, the new live range created from the split needs to be inserted into
the correct constrained set. The $\text{UpdateConstrainedLists}$ algorithm performs this
task and is discussed in Section 3.5.4

After splitting, the live range $lr$ that was originally uncolorable may require further
splitting. We check to see if the remainder of the live range interferes with the top
priority live range, and if so we add it to the worklist. The live range will be split
as many times as necessary until it no longer interferes with the top live range or
becomes uncolorable. If after splitting it no longer interferes with the top live range,
then it must have at least one color available since it was the color assignment to the
current top live range that filled its forbidden set.

<table>
<thead>
<tr>
<th>Algorithm 2: $\text{SplitNeighbors}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong> : $lr^*$ - a live range that was assigned a color</td>
</tr>
<tr>
<td><strong>Input</strong> : constrained</td>
</tr>
<tr>
<td><strong>Input</strong> : unconstrained</td>
</tr>
<tr>
<td>worklist $\leftarrow {lr \in lr^*.\text{interferences} : lr.is_candidate}$</td>
</tr>
<tr>
<td>while $\text{worklist} \neq \emptyset$ do</td>
</tr>
<tr>
<td>choose and remove $lr \in \text{worklist}$</td>
</tr>
<tr>
<td>if $lr.$forbidden = all possible colors then</td>
</tr>
<tr>
<td>if IsNotColorable($lr$) then</td>
</tr>
<tr>
<td>DeleteFromInterferenceGraph($lr$, constrained, unconstrained)</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>$lr, lr' \leftarrow \text{SplitLiveRange}(lr)$</td>
</tr>
<tr>
<td>live_ranges $\leftarrow$ live_ranges $\cup {lr'}$</td>
</tr>
<tr>
<td>$\text{UpdateConstrainedLists}(lr', lr, \text{constrained, unconstrained})$</td>
</tr>
<tr>
<td>if $lr^* \in lr.\text{interferences}$ then</td>
</tr>
<tr>
<td>worklist $\leftarrow$ worklist $\cup {lr}$</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>
When a live range cannot be further split to produce a new colorable live-range, it should be deleted from the interference graph and wholly spilled. Algorithm 3 shows how to remove a live range from the interference graph. As described above, the interference graph is not stored as a central structure; rather, each live range maintains its own set of interferences. To remove a live range from the interference graph, it is first marked as a non-candidate for further allocation. Then the removed live range is deleted from the interference sets of all of the live ranges with which it interferes. Removing an interference may decrease a live range’s neighbors to less than the number of colors being used. In this case, the live range becomes unconstrained and the appropriate sets are updated to reflect this change. As a final step, the deleted live range is removed from the constrained set. Only this set needs to be updated because the live range would not be deleted if it was unconstrained.

<table>
<thead>
<tr>
<th>Algorithm 3: DeleteFromInterferenceGraph</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: lr - a live range to be deleted from the interference graph</td>
</tr>
<tr>
<td><strong>Input</strong>: constrained</td>
</tr>
<tr>
<td><strong>Input</strong>: unconstrained</td>
</tr>
<tr>
<td>lr.is_candidate &lt;- false</td>
</tr>
<tr>
<td>for lr' in lr.interferences do</td>
</tr>
<tr>
<td>lr'.interferences &lt;- lr'.interferences - {lr}</td>
</tr>
<tr>
<td>if</td>
</tr>
<tr>
<td>unconstrained &lt;- unconstrained U {lr'}</td>
</tr>
<tr>
<td>constrained &lt;- constrained - {lr'}</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>constrained &lt;- constrained - {lr}</td>
</tr>
</tbody>
</table>

A live range becomes uncolorable when there is no register left in any of its basic blocks. This can be determined by a walk over the live range’s blocks. At each block, we check to see if there is a color available for the live range and if so we stop and indicate that the live range can be at least partially colored. If there is no color
available in any block, then the live range can not be even partially colored. One additional check should be made when walking over the live range’s blocks: there must be a color available in a block where there is actually a use or a definition of the live range. This condition ensures that we only continue to consider live ranges that may be profitably assigned a register. Otherwise, we could end up trying to split a live range that cannot be split because it contains no blocks we could move into a new live range. The procedure for checking colorability is shown in Algorithm 4.

<table>
<thead>
<tr>
<th>Algorithm 4: IsNotColorable</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong> : lr - a live range to determine colorability</td>
</tr>
<tr>
<td><strong>Data</strong> : used_colors- a mapping from blocks to a set of colors that have been assigned to live ranges extending over the block</td>
</tr>
<tr>
<td><strong>foreach</strong> live unit ∈ lr, call it unit <strong>do</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>return true</td>
</tr>
</tbody>
</table>

Several actions must be taken when a live range is assigned a color. A precise description is given in Algorithm 5. First a color must be selected for the live range. There are a variety of heuristics that can be used for choosing the color. Chapter 5 discusses several heuristics. The live range is also marked as a non-candidate for further allocation so that no attempt will be made to split the live range. Once the color is selected, the used_colors sets for each block in the live range are updated to note the selected color is no longer available. Additionally, each live range interfering with the live range being assigned a color must update its forbidden set to include the new color.
## Algorithm 5: AssignColor

**Input**: lr - a live range to be assigned a color  
**Data**: usecLcolors - a mapping from blocks to a set of colors that have been assigned to live ranges extending over the block

| lr.color ← pick a color  
| lr.is.candidate ← false  
| // update the used_colors set  
| foreach live unit ∈ lr, call it unit do  
| | used_colors[unit.block] ← used_colors[unit.block] ∪ {lr.color}  
| end  
| // update forbidden sets for interfering live ranges  
| foreach lr' ∈ lr.interferences do  
| | lr'.forbidden ← lr'.forbidden ∪ {lr.color}  
| end  
| // add memory operations needed at live range boundaries  
| InsertLoadsAndStores(lr) |

Once a live range is assigned a color, it may need some additional memory operations inserted at its boundaries. These loads and stores are needed to bring the value from memory to the assigned register and to return the value from the register to memory. They are only needed if the live range has previously been split. Section 3.4.4 describes how to compute when a load or store is needed at a live range boundary. We assume at this point that the necessary loads and stores have already been marked. Figure 3.8 taken from Chow and Hennessy’s paper shows an example live range and the loads and stores needed at its boundary [22].

We follow Chow and Hennessy’s recommendation of moving the loads and stores into predecessor or successor blocks whenever possible. Moving the loads allows for some stores internal to the live range to be deleted. Moving the memory operations up and down the edges rather than placing them at the bottom or top of the block means that any execution that flows straight through the live range does not need to execute the superfluous memory operations. At this point in the algorithm, we move the loads and stores onto the appropriate edges in the control flow graph. In a later
Figure 3.8: Example load and store insertion at live-range boundaries. The live range for X consists of the shaded blocks. A load in block (A) is needed for the upward exposed use. The load in block (C) is inserted because it is an entry to the live range. This requires a store be inserted in block (A). The store in block (B) is because of its definition of X. The store in block (D) is for the two definitions in X that reach the live range exit.
pass we will actually insert the loads and stores in the correct block and split edges as necessary. That process is discussed in Section 3.6.2. Figure 3.9 shows the effect of moving loads and stores at the live range boundary.

To move the loads, we check to see if there is at least one predecessor block that belongs to our live range. This check ensures that there is some benefit to moving the load since no load will be executed on the path that comes from the predecessor block in the live range. If there is no predecessor block that belongs to the live range the load is inserted at the top of the block. Movement of stores is handled similarly, except that we can ignore completely any store that is only internal to the live range. Algorithm 6 shows the pseudo-code for these operations.
Figure 3.9: Example load and store movement at live-range boundaries. The load in block (C) is moved up the edge towards (E), and a new block is created to split that edge. This allows the internal store in block (A) to be removed. The store in block (B) is moved down to the successor block (F). The store in block (D) is not moved because it has only one successor.
Algorithm 6: InsertLoadsAndStores

**Input**: \( lr \) - live range needing loads and stores at its boundaries

// move loads and stores at entry and exit points onto the edges

**foreach** live unit \( u \) \( \in \) \( lr \), call it unit **do**

// move load

**if** unit.need_load **then**

**if** unit.block has at least one predecessor in the live range **then**

**foreach** predecessor of unit.block, call it pred **do**

**if** pred \( \notin \) \( lr \) **then**

insert a load for \( lr \) on the edge from pred

**end**

**end**

**else**

insert a load for \( lr \) at the top of unit.block

**end**

// move store

**if** unit.need_store and not unit.internal_store **then**

**if** unit.block has at least one successor in the live range **then**

**foreach** successor of unit.block, call it succ **do**

**if** succ \( \notin \) \( lr \) and \( lr \) \( \in \) \( \text{LiveIn}(\text{succ}) \) **then**

insert a store for \( lr \) on the edge to succ

**end**

**end**

**else**

insert a store for \( lr \) at the bottom of unit.block

**end**

**end**

3.4 Building live ranges

Initial live ranges are found using Algorithm 8. The method shown here differs from the description given by Chow and Hennessy. In this thesis, live ranges are found using the analysis provided by putting the code into SSA form [30]. Chow finds initial live ranges by using data-flow analysis to compute reaching and live sets for each program value. Using SSA to compute live ranges allows for a more precise
definition of the initial live ranges. As shown in figure 3.2, Chow and Hennessy will find live ranges that contain sections which do not need to be kept in the same register. They find live ranges by taking all of the blocks in which a variable is both live and reaching. This does not take def-use chains into account, and the initial live ranges may not be a contiguous set of blocks. Using SSA to compute the live ranges builds a live range made up of a contiguous set of blocks.

Chow and Hennessy claim that separating the initial live ranges into their constituent connected components would require a "potentially costly separation process". Furthermore the splitting algorithm would presumably perform the necessary splitting at the point it is needed. They note that these live ranges could lead to suboptimal allocation, but argue that the benefits of quicker allocation due to fewer live ranges and a splitting algorithm that handles the separation process makes this formulation of live ranges acceptable. Part of their motivation comes from the fact that they were writing in a time when memory and processor resources for compilation were much more constrained compared to current machines. It is interesting to note that although memory and processor time is relatively abundant today, CPU registers are still a highly contested resource, making register allocation an important ongoing problem.

In this thesis we use SSA to compute live ranges because it produces more precise live ranges. Many modern transformations use SSA for analysis, and it can be computed quickly in practice. For example, Briggs uses SSA to find live ranges in his register allocator. Since one of the goals of this thesis is a comparison with that allocator, the use of SSA is needed for a fair comparison. In short, there are no good arguments to prevent the use of SSA to find initial live ranges and several good reasons why we believe it is the right choice.
3.4.1 Data-flow analysis

We must analyze the input code to find the live ranges in a program. This analysis should provide information about how values are related to each other and the points at which they need to be kept in a register. This information is provided by the following analysis:

**SSA** - the pruned version of SSA should be used to avoid inserting phi-nodes for variables that are not live.

**Liveness** - live sets should be built for variables in the SSA namespace. Liveness can be solved once in the original name space and then mapped to the SSA name space. It is necessary to have both LiveIn and LiveOut sets.

**Reachable** - graph reachability information for nodes in the flow graph is needed for inserting loads and stores at live range boundaries.

**Local Variables** - variables that are referenced in only one block are local. These may be removed from the interference graph and handled separately as described in Section 3.4.3.

3.4.2 Building initial live ranges

Live ranges are built in three steps:

1. partition SSA names into sets corresponding to live ranges
2. find the set of blocks that make up each live range
3. build the interference graph

Step one is done by running a union-find algorithm over all of the phi-nodes in the program. Initially, each SSA name is placed in its own set. At each phi-node the set for the name defined by the phi-node is unioned with all the sets for each variable flowing into the phi-node. The pseudo-code for this procedure is shown in Algorithm 7.
The result of this step is that all definitions that can reach a use are put into the same set. These groups of definitions and uses correspond exactly to the live ranges. Using SSA to find the live ranges automatically builds live ranges to contain only the definitions and uses that correspond to the same value. Names that happen to share the same virtual register, but correspond to different values, will be placed in different live ranges because of the SSA construction. Once the partition has been found, a mapping is recorded from SSA name to live-range id. This id gives each live range a unique identifier that will be used throughout the register-allocation algorithm. The identifier also serves as an identifier for the live-range family and is stored in the orig_lrid field of the live-range structure in figure 3.6

Algorithm 7: FindLiveRangeNames

| Result: live_range_name_map- mapping from SSA name to live range id |
| for each ssa name, call it v do |
| MakeSet(v) |
| end |
| for φ ∈ phinodes do |
| for each incoming parameter of φ, call it v do |
| phiset ← FindSet(φ) |
| vset ← FindSet(v) |
| UnionSets(phiset, vset) |
| end |
| end |
| for each ssa name, call it v do |
| live_range_name_map [v] ← FindSet(v).id |
| end |
| return live_range_name_map |

Once the mapping from SSA names to live ranges is known, we find the blocks that make up each live range. The live range includes any block containing a definition or a use of an SSA name that belongs to the live range. Phi-nodes can be ignored here since they do not constitute “real” definitions – they will not be spilled. Any use of a name defined by the phi-node will cause the block to be included in the live range.
Additionally, any block in which a name belonging to the live range is live on exit should be included in the live range. A live-out variable indicates an upwards exposed use, which means we should keep the variable in a register, if possible, until that use is reached. After the complete set of live ranges in a block is known, the interference graph is built by adding an interference between all live ranges spanning the block. The pseudo-code for building live ranges and interferences is shown in Algorithm 8.
Algorithm 8: BuildInitialLiveRanges

**Result:** live_ranges - set of all the initial live ranges

// find live ranges and initialize the live range structures
live_range_name_map ← FindLiveRangeNames()

**foreach** live range id in live_range_name_map, call it lrid do
| live_ranges [lrid] ← a new LiveRange structure
**end**

**foreach** basic block, call it block do
| lrs ← ∅

// find live ranges for this block
**foreach** instruction in the block, call it inst do
| for use ∈ inst.uses do
| | lr ← live_ranges [live_range_name_map [use]]
| | lrs ← lrs U {lr}
| **end**
| for def ∈ inst.defs do
| | lr ← live_ranges [live_range_name_map [def]]
| | lrs ← lrs U {lr}
| **end**
| for v ∈ block.LiveOut do
| | lr ← live_ranges [live_range_name_map [v]]
| | lrs ← lrs U {lr}
| **end**
**end**

// build interference graph for this block
**for** lr ∈ lrs do
| lr.units ← lr.units U {LiveUnit(block)}
| lr.blocks ← lr.blocks U {block}
| for lr' ∈ (lrs - {lr}) do
| | lr.interferences ← lr.interferences U {lr'}
| **end**
| **end**

**foreach** lr ∈ live_ranges do
| if lr is local then
| | DeleteFromInterferenceGraph(lr)
| else
| | MarkLoadsAndStores(lr)
| **end**
**end**
3.4.3 Local live ranges

Some variables are defined and used in only one block. A local live range is a live range for such a variable. Chow and Hennessy do not allocate these live ranges using the global priority-based algorithm because they assume that a later code generation phase will do the allocation. We use a flag to specify whether or not local live ranges should be allocated by the global allocator. By default, we do not globally allocate local live ranges. If local live ranges are allocated separately, they are removed from the interference graph. The local allocation phase described in Section 3.6.1 handles allocation for these local live ranges.

3.4.4 Marking loads and stores

Once the extent of a live range is known, each live unit in the live range must be annotated with whether it needs a load at the top of the block and/or a store at the bottom of the block. These annotations are used when computing the priority of the live range and also to know where to insert loads and stores for the live range when it is assigned a register. Initially, the live ranges should not need any loads or stores at their boundaries, as they are completely self contained; all blocks on a path from a definition to a use in the live range will be included in the live range. After a live range is split, part of the path from a definition to a use may belong to a new live range that was created by splitting the original live range containing that definition and use. In other words, a live range may no longer consist of contiguous basic blocks. In this case, loads and stores may be required at a live-range boundary.

The flags for loads and stores are computed as follows. First, the flags for all the loads needed in a live range must be computed, because the flags for stores depend on whether or not a load is needed. If the first occurrence of a variable in a live unit is a definition, then a load will not be needed because the value is destroyed before
it is read. Otherwise, a load is needed in that block only if it is an entry to the live range. A block is an entry point of a live range if it contains a predecessor in the CFG which is not part of the live range.

If the live range contains definitions, then a store may be needed at exit points of the live range. A block is an exit point of the live range if it has a successor in the CFG that is not part of the live range. It is possible that a live range contains no definitions if it has previously been split from another live range. To determine where a store is needed, we first find all of the blocks containing a definition for the live range. We look at all of the blocks in the live range that are reached by any of these definitions, using the results from our graph-reachability analysis computed earlier. If any block reached by a definition is an exit point of the live range and the live range is live out of that block, then a store is needed. Also, if a successor of the block needs a load then a store is needed so that the load accesses the current value. This second case is called an internal store. As long as we perform code motion of loads and stores as described in Section 3.3, these stores can be safely removed. We mark them as internal stores to distinguish them for Algorithm 6. The detailed procedure for marking loads and stores is given by Algorithm 9.
Algorithm 9: MarkLoadsAndStores

Input : lr - the live for which loads and stores should be marked
Result: the live units are annotated with whether they need loads and stores

// mark loads
foreach live unit ∈ lr, call it unit do
  if not unitstartswithdef and unit is an entry point to lr then
    unit.need_load ← true
  end
end

// mark stores
defblocks ← {unit.block ∈ lr.units : unit.defs > 0}
if |defblocks| > 0 then
  may-need-store ← lr.blocks ∩ (∪_{b ∈ defblocks} Blocks-Reachable-From(b))
  foreach block ∈ may-need-store do
    unit ← lr.unitmap[block]
    if block is an exit point of lr and lr ∈ LiveOut(block) then
      unit.need_store ← true
      unit.is_internal_store ← false
    else
      if a successor of block needs a load then
        lr.unitmap[block].need_store ← true
      end
    end
  end
end

3.5 Live-range splitting

Live-range splitting is one of the most important activities in the priority-based allocator. Splitting live ranges into pieces when they would otherwise have to be wholly spilled allows otherwise uncolorable live ranges to be broken up into two live ranges that may be given different colors. The goal of splitting is to take a live range for which there is no available color and find a section of the live range that can be assigned a color. The split live range should be as large as possible, with the
constraint that there is a still color available. We want large live ranges so that we
do not end up with too many small live ranges.

3.5.1 The splitting algorithm

The splitting algorithm is invoked when a live range has no color that can be
assigned for its entire length, but there is still a section for which a valid color exists.
The algorithm starts by finding a point in the live range to begin splitting out. This
point is chosen according to the following preferences, as originally described by Chow
and Hennessy:

1. a block that is an entry point to the live range and contains a definition
2. a block that contains a definition
3. a block that is an entry point to the live range and contains a use
4. a block that contains a use

Once the starting point for the new live range is chosen, the algorithm attempts
to transfer as many blocks as possible to this new live range. The control-flow graph
is explored in a breadth-first manner starting at the chosen block. When a node is
expanded, each successor block is checked to see if it is part of the live range that is
being split and can be transferred to the new live range. The block can be transferred
if adding it to the new live range does not cause its forbidden set to become full. If
the block is added to the new live range, then it is also added to the exploration
queue so that its successors will be examined for inclusion in the live range. The
entire algorithm is show in detail in Algorithm 10.
Algorithm 10: SplitLiveRange

| Input   | lr - the live range to be split |
| Output  | (lr, lr') - the remaining and new live ranges split from the original input |
| Data    | used_colors - a mapping from blocks to a set of colors that have been assigned to live ranges extending over the block |

initialize lr' from lr
lu ← choose a splitting point in lr
TransferLiveUnit(lr, lr', lu)
queue.push(lu.block)

while queue ≠ ∅ do
    block ← queue.pop()
    for succ ∈ block.successors do
        if succ ∈ lr.blocks and (lr'.forbidden ∪ used.colors[succ]) ≠ all colors then
            TransferLiveUnit(lr, lr', lr.unitmap[succ])
            queue.push(succ)
        end
    end
end
UpdateAfterSplit(lr, lr')
return (lr, lr')

3.5.2 Transferring live units between live ranges

When a block is transferred from one live range to another, it is necessary to update several data structures associated with the live ranges. The details of these updates are shown in Algorithm 11. When a block is moved, the receiving live range needs to add the live unit to its set of units as well as add the block the live unit is wrapping to its set of blocks. The giving live range needs to remove the live unit and block from its sets. In addition, the forbidden set is updated for the receiving live range. This is necessary because the forbidden set is used to determine what blocks can be added to the new live range. This set needs to be maintained incrementally to ensure the proper termination of the splitting algorithm.
3.5.3 Updating live ranges after a split

After the extents of the new live range have been established, several of the live range’s data structures must be updated. These updates are required for both the new live range and the split live range. The details of these updates are shown in Algorithm 12.

Splitting creates a new live range and removes some blocks from an existing live range. Since the interference graph is built in respect to live ranges and basic blocks, these changes require an update to the interference graph. The interferences that can be changed by splitting are only those interferences involving the original live range. Each live range that interfered with the live range before it was split is examined. If it interferes with the new live range, then an interference is added. If it does not interfere with the new live range, then no update is necessary since the interference did not previously exist. If the original live range no longer interferes with some live ranges (because they no longer have any overlapping basic blocks) then those
interferences are removed.

The blocks in a live range that need a load or a store are dependent on the actual boundaries of the live range. These boundaries have changed for the split live range, and therefore we need to re-examine where the live-ranges's loads and stores must be placed. The new live range also needs to compute these locations since they have never been marked. These values must be known because they are used in computing the priority of the live range.

Next, the forbidden set for the live range that was split needs to be updated. Blocks were removed from this live range which could potentially reduce the number of forbidden colors. It is not possible to incrementally compute this set when a block is removed from a live range because more than one block could forbid a color, so it is not possible to know that a color can be removed from the forbidden set by examining a single block. The forbidden set for the new live range is built incrementally during the splitting process and does not need to be recomputed here.

Finally, priorities for the live ranges need to be recomputed. In the algorithm presented in this thesis, we simply mark that the priority needs to be computed and let the main loop handle the computation if needed. This is done both as an efficiency measure and to simplify the code. The priority for the new live range only needs to be computed if it is going to be put on the constrained list. The priority for the live range that was split should only be computed if it is not going to be split again. In order to postpone those tests we lazily recompute the priority after a split.
Algorithm 12: UpdateAfterSplit

Input: origlr - the live range that was split
Input: newlr - the new live range carved out of the split live range
Result: the live ranges are updated with correct interferences and other information

// update interferences
updates ← origlr.interferences
foreach fearlr ∈ updates do
    if (newlr.blocks ∩ fearlr.blocks) ≠ ∅ then
        newlr.interferences ← newlr.interferences ∪ {fearlr}
        fearlr.interferences ← fearlr.interferences ∪ {newlr}
    end
    if (origlr.blocks ∩ fearlr.blocks) = ∅ then
        origlr.interferences ← origlr.interferences - {fearlr}
        fearlr.interferences ← fearlr.interferences - {origlr}
    end
end
MarkLoadsAndStores(newlr)
MarkLoadsAndStores(origlr)

// rebuild forbidden set
origlr.forbidden ← ∅
foreach block ∈ origlr.blocks do
    origlr.forbidden ← origlr.forbidden ∪ used_colors[block]
end
mark that newlr.priority needs to be recomputed
mark that origlr.priority needs to be recomputed

3.5.4 Updating the constrained lists

The constrained and unconstrained lists must be updated after a live range has been split. Splitting may increase the number of interferences for any live range that interferes with both the original and new live range. The original live range may become unconstrained and the new live range must be put on the correct list. In short, we need to handle three cases:
1. Any live range that now interferes with both the new live range and the split live range may now be constrained.

2. The new live range needs to be put in the correct list depending on whether or not it is constrained.

3. The live range that was split may no longer be constrained.

The checks for these items are straightforward. The detailed pseudo-code is shown in Algorithm 13.

<table>
<thead>
<tr>
<th>Algorithm 13: UpdateConstrainedLists</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: lrnew - new live range created from a split</td>
</tr>
<tr>
<td><strong>Input</strong>: lrorig - original live range that was split from</td>
</tr>
<tr>
<td><strong>Input</strong>: constrained- set of constrained live ranges</td>
</tr>
<tr>
<td><strong>Input</strong>: unconstrained- set of unconstrained live ranges</td>
</tr>
<tr>
<td><strong>Result</strong>: constrained, unconstrained contain the correct live ranges</td>
</tr>
</tbody>
</table>

```plaintext
for fearlr ∈ (lrnew.interferences ∩ lrorig.interferences) do
    if |fearlr.interferences| ≥ k then
        constrained ← constrained U {fearlr}
        unconstrained ← unconstrained − {fearlr}
    end
end
if |lrnew.interferences| ≥ k then
    constrained ← constrained U {lrnew}
else
    unconstrained ← unconstrained U {lrnew}
end
if |lrorig.interferences| < k then
    constrained ← constrained − {lrorig}
    unconstrained ← unconstrained U {lrorig}
end
```

3.5.5 Zero-occurrence live ranges

Splitting can leave the original live range with no blocks containing a reference to the variable. This situation results in a zero-occurrence live range. These live ranges will be given a priority of zero and so do not degrade the quality of the allocation.
results. They do, however, take time and space to process, so it is a good idea to remove them from the interference graph as soon as possible. We can insert a check in Algorithm 2 to see if the split live range becomes a zero-occurrence live range after splitting. The live range can then be immediately deleted from the interference graph. Live range trimming, discussed in Section 4.2, can reduce the number of zero-occurrence live ranges generated by the splitting algorithm.

3.6 Rewriting the code

Once registers have been assigned to live ranges, the code must be rewritten to reflect the results of the allocation. Spill code must be inserted for any live ranges that have not been allocated a register. Chow and Hennessy leave the details of this step unspecified because they assume the presence of a code generator that will rewrite the code and insert any needed spill code. In this section, we describe our method for rewriting and spill code insertion.

3.6.1 Local allocation and spill-code insertion

We perform local allocation and spill code insertion in one walk over the code after the priority-based allocator has run to completion. For each block, we examine all of the variables in each instruction. Using the live_range_name_map, we translate the SSA variable name to the original live range corresponding to that name. The original live range may have been split, so we use the blockmap field in the live range to find the actual live range that contains this block. Once we have a handle to the real live range, we can check its color field to see if it has been assigned a color. If it has a color, then we translate the color into an actual machine register.

Live ranges that have not been assigned a color and local live ranges are handled
uniformly by the local allocator. If an instruction contains a reference to a live range that has no color, then a temporary register is found to hold the live range. For a use, the live range is loaded from memory into the temporary register. We keep track of which live ranges are currently in temporary registers as we walk through the block so that subsequent references to the same live range can simply access the temporary register. If no temporary register is available, we evict some live range from its current temporary and use that register. The choice of which live range to evict follows the method of Best and evicts the live range that is referenced the farthest distance in the future [7]. This distance is calculated in terms of the current block, or a series of blocks if they were previously split to reduce their size as described in Section 3.7. When we evict a live range from a temporary or we reach the end of a basic block, we may need to insert a store for the live range. If the temporary contains a live range that was defined and is live, then a store is generated.

3.6.2 Instantiating loads and stores at live range boundaries

After local allocation and spill code insertion, one final step is needed to complete the code rewriting phase. Loads and stores at live range boundaries that were moved onto edges must be inserted into the actual code. At this point, we also split edges as needed. We examine all loads and stores that have been moved onto an edge. For each store, we check to see if the successor block has multiple predecessors, and, if so, we split that edge with a new block so that the store is only executed on the path leading from the live range exit. Similarly for loads, we check to see if the predecessor block has multiple successors, and, if so, we split the edge with a new basic block. Once the edges have been split, we can insert the loads and stores. Loads are inserted at the bottom of the predecessor blocks and stores are inserted at the top of the successor blocks. These loads and stores should access the machine registers assigned to the
live ranges since the remaining code has already been rewritten.

3.7 Limiting basic-block size

The size of a basic block has a direct impact on the number of edges in the interference graph. Live ranges are built from a set of basic blocks, so reducing block size potentially removes an interference. We allow an input parameter to specify the maximum number of instructions a basic block may contain. We split any block that exceeds the maximum length before allocating registers. Before splitting the blocks, we first identify local live ranges.

We remove local live ranges from the interference graph and leave them for the local allocation phase. If local values are not computed before splitting basic blocks, then some previously local values will now span basic blocks and thus increase the work of the global allocator. We choose to compute local values before splitting basic blocks because we have perfect information about the lifetime of these values, and this allows for a higher quality local allocation. Also, we have an option to do a global allocation of local values so we can always leverage the power of the global allocator if desired.

3.8 Differences between the priority-based and Chaitin-Briggs allocators

The Chow and Hennessy approach differs from the previous graph-coloring register-allocation algorithms described by Chaitin and Briggs. There are three main differences between the algorithm given in this thesis and the Chaitin-Briggs approach. First, the priority-based algorithm uses coarse live ranges defined over basic blocks, while Chaitin-Briggs uses live ranges defined over instructions. This difference is not
absolute, because by limiting basic blocks to be a single instruction, the priority-based algorithm builds the same interference graph as the Chaitin-Briggs algorithm. The second difference is that Chaitin-Briggs builds and colors the interference graph multiple times, iterating every time spill code is inserted. The priority-based algorithm only builds and colors the graph once, which forces the priority-based allocator to a priori reserve registers for spill code. The final difference is the order in which nodes in the interference graph are colored. Chaitin-Briggs attempts to assign a color to all nodes, spilling when this is not possible. The priority-based approach assigns colors in priority order. The emphasis is on allocating registers to the most important live ranges even if that causes less important live ranges to be spilled.
Chapter 4

Engineered priority-based allocator

4.1 Introduction

While implementing the priority-based allocator described in the previous section, we found several areas in which the allocator could be more aggressive in the pursuit of good performance. Some of these improvements, such as rematerialization, have been done in other allocators, while others are specific to our implementation. Each of these additions will cause the allocator to make different choices during the allocation. These choices may lead to better allocations, although this is not guaranteed. Problems arise with the way that these choices interact with the rest of the allocation. The effects of these additions on allocation quality are discussed in chapter 6.

In this chapter, we discuss five additions to the priority-based allocator: live range trimming, optimistic simplification, rematerialization, load/store pair removal, and smart loading of copies. Optimistic simplification, rematerialization, and smart loading of copies have been discussed in the context of a Chaitin-Briggs allocator. Live-range trimming and load/store pair removal came about based on our observation of the behavior of the priority-based allocator. We also discuss how to limit the number of splits performed by the allocator and describe an enhancement to the local-allocation phase. Each of these changes is discussed in detail below.
4.2 Live-range trimming

The idea behind live range trimming is to prune useless blocks from the live range after splitting. Recall that the splitting algorithm works by choosing a starting point and then adding blocks in a breadth first walk over the successors in the control-flow graph. When a block is reached that can not be added to the new live range, the search is stopped along that path. Figure 4.1 shows the results of running the splitting algorithm on a live range. On the left, the image shows a live range represented by the nodes it spans in the control-flow graph. It contains a definition of x in block A and uses of x in blocks D, H and L. The right side of the image shows the two new live ranges formed by splitting the original live range. The split started at node A and was stopped at nodes H and I.

The point to notice in this image is that the blocks E, F, and G in the split live-range (gray) and I, J, and K in the original live-range (white) serve no purpose. The last reference of x in the first live range is in block D. Keeping E, F, and G in the live range ties up a register in those blocks even though it will never be referenced. If we remove those blocks from the live range, we free up the register for another live range to use. A similar situation exists with the I, J, and K blocks of the second live range. There is no reference to the live range in those blocks, so assigning a register to the live range only prevents another live range from using that register. These blocks should be removed to reduce the register pressure in those regions. Figure 4.1 shows the shape of the live ranges after trimming.

Live-range trimming is done by a process similar to mark and sweep garbage collection. The blocks that contain an actual use or a definition, called the critical blocks, are marked as initially useful and put on a worklist. Each element of the worklist is processed by checking its predecessors in the control-flow graph. If the
Figure 4.1: Live range (left), after splitting (right) and trimming (bottom)
predecessor is in the live range, then it is marked and added to the worklist. Once all blocks in the worklist have been processed, it is safe to remove any live unit that has not been marked. The live range from which the new live range is split also needs to run the trimming algorithm working over the successors of the critical blocks. This will remove any useless blocks from the top of the live range. It is not necessary for the new live range, because it starts with a definition or a use block, and all blocks in the live range are reachable from this starting point due to the way the splitting algorithm works. Algorithm 14 gives the code for the trimming process in the downward direction.

**Algorithm 14: TrimLiveRange**

<table>
<thead>
<tr>
<th>Input</th>
<th>lr - the live range to be trimmed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result</td>
<td>the live range is trimmed of blocks that are not needed</td>
</tr>
</tbody>
</table>

worklist ← ∅  
// find starting useful units (critical blocks)
foreach unit in lr.units do
    if unit.uses > 0 or unit.defs > 0 then
        mark unit useful
        worklist ← worklist ∪ {unit}
    end
end

// mark predecessors useful
while worklist ≠ ∅ do
    choose and remove unit ∈ worklist
    foreach pred of the unit in the cfg do
        if pred is part of lr and not already marked then
            mark pred useful
            worklist ← worklist ∪ {pred}
        end
    end
end

// remove useless units
any unit not marked useful can be removed from the live range
4.3 Rematerialization

Some values can be recomputed at any point during the program and therefore do not need to be kept in memory when spilled. The compiler should recognize when it is cheaper to recompute a value than to spill it. This improvement is called rematerialization. Briggs et al. showed how rematerialization can be implemented in the context of a graph-coloring register allocator [11]. The idea is to find the values that can always be recomputed. In practice, the rematerializable values are constants, and they are recomputed using a load immediate of the constant. These values will be cheaper to spill because they do not need to be stored in memory. Any reference to the value can be replaced with a reference to a load immediate of the value. In this section we briefly review Briggs-style rematerialization and show how it can be implemented in the priority-based allocator.

There are two major steps to implementing rematerialization: finding rematerializable values and splitting live ranges into rematerializable sections. The first step is necessary because we have to know which values can be recomputed instead of stored. The second is necessary because we need to be able to spill the parts of a live range that are rematerializable before we spill the parts that require heavyweight spilling. The easiest way to do this is to split the live range into separate sections: one that is rematerializable and one that is not. Figure 4.2 shows how Briggs handles rematerializable live ranges. The left side shows a piece of code and the corresponding live range. The right side shows the live range split into the rematerializable and non-rematerializable sections. The live ranges have been split by inserting a copy between the two sections. Some of these splits may not be productive if the live range never requires spilling. Briggs attempts to minimize the impact of splitting though two mechanisms: conservative coalescing and biased coloring. Biased coloring tries to
assign both parts of the live range the same color, making the copy useless. Conservative coalescing attempts to coalesce the copy if it can guarantee that the resulting live range will get a color.

We implemented rematerialization in the priority-based allocator using ideas from the Briggs implementation. The main difference between the implementations is that we do not insert a copy to split the live ranges. We wanted to leverage the mechanism for splitting that is built into the allocator to help with rematerialization. Inserting a copy would cause the separate parts of the live range to interfere, since they are both referenced in the same block. We use the idea of splitting live ranges from other parts of the algorithm to handle splitting for rematerialization. This fits naturally into the priority-based allocator's structures for live ranges and does not cause the difficulties of inserting the copy. However, avoiding the copy does present its own difficulties that are discussed below.

4.3.1 Finding rematerializable values

As with Briggs's formulation, values which are rematerializable are found using a variant of the Sparse Simple Constant propagation algorithm proposed by Wegman and Zadeck [54]. The details are presented in the rematerialization paper by Briggs et al. [11], and we follow that implementation exactly. The general idea is that we have a lattice with the elements \{T, \bot, C\}, and each SSA name is tagged with a value in the lattice. Constants are initially tagged with \(C_i\), where \(C_i\) is the value of the constant. Names that are defined by copies are given the value \(T\) because their tags will be defined from the source of the copy. All other names are tagged with \(\bot\). The algorithm propagates the tags in a worklist fashion, processing each value by looking at all of its uses and lowering the lattice element when appropriate. A copy will be lowered to the same tag as its source. Phi nodes are processed by
Figure 4.2: Example of Briggs-style rematerialization. Part (a) shows the original code containing two definitions of x and three uses. Part (b) shows the rematerializable part of the live range split into a separate live range x'. Part (c) shows the resulting code if x' is spilled.
Figure 4.3: Example of rematerialization in the priority-based allocator. Part (a) shows the original code containing two definitions of x and three uses. Part (b) shows the rematerializable part of the live range split into a separate live range x'. Part (c) shows the resulting code if x' is spilled. We rely on dead code elimination to remove the definition and store of T. The store of T can not be eagerly deleted because if x is spilled, it must load the value from memory. This is explained in detail on page 73.
looking at each name flowing into the phi node. If all names have the same tag, then
the name defined by the phi node is lowered to that tag; otherwise it is lowered to ⊥.
Any time a value has its tag lowered, it is added to the worklist. The end result will
tag all SSA names that are defined by constant values. These tags are used to split
out the parts of the live range that are rematerializable.

4.3.2 Splitting live ranges into rematerializable sections

Once the names corresponding to rematerializable values have been found, it is
time to build the live ranges. We differ from the approach described by Briggs et al. to
take advantage of the splitting mechanism already present in the priority-based algo­
rithm. Our idea is to build the live ranges as normal, but keep track of which parts
of the live range can be rematerialized. After completely building the live ranges,
we make a separate pass to split out the rematerializable parts of the live ranges
using the same techniques needed for splitting live ranges during the running of the
allocation algorithm. Building the live ranges in the presence of rematerialization is
a two step process:

1. Find live ranges by running the union-find algorithm described in Algorithm 7.
   At the same time, keep track of the live ranges that would result from not
   unioning the result of a phi node with one of its parameters if they have different
tags. Any time a parameter to the phi node is not unioned with the result,
   record that as a split.

2. Process all of the splits recorded in the previous step. The splits are processed
   by partitioning the blocks of a live range into separate live ranges based on the
   information recorded in the previous step.
The process for step one is shown in Algorithm 15. The algorithm is nearly the same as the previous algorithm for finding live-range names. The only support added for rematerialization is to keep a separate collection of sets for rematerialization used as input to union-find. One extra check is needed before unioning rematerializable sets. We see if the parameter to the phi node has the same tag as the name defined by the phi node. If the tags are the same then we union as normal. The live ranges can be joined because they are either both constants, or both non-constants. If the tags are not then same, then we do not union the live ranges because we have found a point where a constant value is joining with a non-constant value. We record the phi nodes that have different tags flowing into them to remember that their live ranges should be split later. Once the splits have been found, we use the live_range_name_map to build the live ranges as normal. The live ranges in the remat_splits set are then split according to the rematerialization tags found in the first step.
Algorithm 15: FindRematLiveRangeNames

Input: tags - mapping from SSA name to rematerialization tag
Result: live_range_name_map - mapping from SSA name to live range id
Result: remat_splits - SSA names whose corresponding live ranges can be split into rematerializable sections
Result: remat_sets - collection of sets with one set per live range and each SSA name belonging to one set. SSA names with different rematerialization tags will be in different sets.

for each SSA name, call it v do
    sets ← MakeSet(v)
    remat_sets ← MakeSet(v)
end

for φ ∈ phinodes do
    for each incoming parameter of φ, call it v do
        // build standard live ranges
        phiset ← FindSet(sets, φ)
        vset ← FindSet(sets, v)
        UnionSets(phiset, vset)
        // build remat live ranges
        remat_phiset ← FindSet(remat_sets, φ)
        remat_vset ← FindSet(remat_sets, v)
        if tags[φ] = tags[v] then
            UnionSets(remat_phiset, remat_vset)
        else
            remat_splits ← remat_splits ∪ φ
        end
    end
end

for each SSA name, call it v do
    live_range_name_map [v] ← FindSet(v).id
end

Live-range splitting considers each split in the remat_splits set. The live units of the original live range are partitioned according to the live ranges represented by remat_sets. The SSA name used in the live unit will fall into one of the sets in remat_sets, and this set represents the new live range. After the live units have been partitioned into sets corresponding to their new live ranges, the new live ranges
are created and the live units transferred to those live ranges. Any new live ranges that are rematerializable are marked so they can use that information when spilling and computing priorities. The interference graph needs to be updated because the original live range has been split into multiple pieces. The algorithm for splitting rematerializable live ranges is shown in Algorithm 16. At this point, we also record that the new live ranges have all been split from a single live range. This is useful when we need to assign colors to the live ranges. We can check to see if any color has been assigned to a live range we were split from, and select that color for the current live range. This is the same idea as biased coloring discussed by Briggs et al. to reduce the impact of unproductive splits [11].
Algorithm 16: SplitRematLiveRangeNames

**Input**: tags - mapping from SSA name to rematerialization tag

**Input**: remat_splits - SSA names whose corresponding live ranges can be split into rematerializable sections

**Input**: remat_sets - collection of sets with one set per live range and each SSA name belonging to one set. SSA names with different rematerialization tags will be in different sets.

// convert split SSA names to live ranges
splits ← ∅
foreach SSA name in remat_splits, call it φ do
    lrid ← live_range_name_map[φ]
    splits ← splits ∪ live_ranges[lrid]
end
foreach lr ∈ splits do
    // split live units according to remat_sets
    unit_map ← ∅ // map from setid to list of units
    foreach lu ∈ lr.units do
        setid = FindSet(remat_sets, lu.ssa_name)
        unit_map[setid].append(lu)
    end
    // create live ranges according to unit_map
    new_lrs ← ∅
    foreach (setid, unit_list) ∈ unit_map do
        initialize lr' from lr
        if tags[unit_list.head.ssa_name] = C then
            lr'.rematerializable ← true
        end
        foreach lu ∈ unit_list do
            TransferLiveUnit(lr, lr', lu)
        end
        new_lrs ← new_lrs ∪ lr'
    end
    // rebuild interferences for new live ranges
    foreach fearlr ∈ lr.interferences do
        foreach lrnew ∈ new_lrs do
            if fearlr and lrnew interfere then
                add interference between lrnew and fearlr
            end
            // original lr has no live units thus no interferences
            remove interference between lr and fearlr
        end
    end
end
4.3.3 Spilling in the presence of rematerialization

The last issue to discuss when implementing rematerialization is how to handle spilling of rematerializable values. Given the choice of spilling two live ranges, we would prefer to spill the live range that is rematerializable because we expect recomputing the value will be cheaper than loading it from memory. We make this preference explicit by altering the priority function. We change the cost of loads and stores in the priority function to be two for non-rematerializable live ranges and one for those that can be rematerialized. The lower cost of loads and stores causes those live ranges that are rematerializable to tend to have lower priority so that they will be allocated later in the algorithm and therefore are more likely to be spilled.

A small change is needed to the mechanics of spilling for rematerializable live ranges. When we are rewriting the code as described in Section 3.6 and come upon a load request for a live range, we check to see if the live range is rematerializable and, if so, use a load immediate instruction to rematerialize the value. Any stores of a rematerializable live range are left in place, for a reason we discuss on page 73.

We also make a small change to how we process the loads and stores moved onto edges as described in Section 3.6.2. When looking at a load that has been moved onto an edge, we check to see if the predecessor block was originally part of the same live range and was split off because it was rematerializable. If so, then we can use a load immediate instead of a heavyweight load to load the value along that edge. We found it useful for each original live range to keep a mapping from the blocks it started with to the live range currently owning the block (the blockmap field in Figure 3.6). This mapping can be used to check if the block is currently owned by a rematerializable live range.

As mentioned above, if a rematerializable live range is spilled, then we must keep the original definition of the live range along with the store inserted after the definition
because it is spilled. This is different than Briggs-style rematerialization which can freely delete this definition and the corresponding store. The reason we must leave the definition and store is that we may later attempt to generate a load for the live range and must have the value in memory for correctness. Consider the situation in Figure 4.4. In Part a, the original live range \( x \) has been split into its rematerializable sections \( x \) and \( x' \). As the allocation proceeds, \( x \) is split twice more into live ranges \( x_2 \) and \( x_3 \) as shown in Part b. Now assume that \( x_2 \) and \( x_3 \) are assigned two different registers. The final generated code is shown in Part c. Here we see why the store of the rematerializable live range is necessary: the load of \( x_3 \) in the final block expects the value to be in memory. The live range \( x_2 \) will not know to store the value because it contains no definitions. If we were to delete the definition and store of the rematerializable live-range, then the later load would be incorrect. When we are spilling a rematerializable live-range, we must retain the definition. We run a post pass of dead code elimination that will remove the definition and store if they are found to be useless.

The difficulties of implementing rematerialization in the priority-based allocator stem from limited knowledge of how a future live range will be split and the coarse interference graph that makes inserting copies clumsy because the live ranges will conflict with each other. Our method avoids the problems with coarse interference by using a splitting mechanism more natural for the algorithm. The issue with future splits is dealt with by a separate post pass that looks for useless stores. Any algorithm for rematerialization in a priority-based allocator will have to deal with these two issues.
Figure 4.4: This example shows why we must leave the definition and store of a rematerializable live range that is spilled. Part (a) shows a live range split into its rematerializable parts. In (b) we have further split x such that x and x2 can be given different colors. In Part (c) we see the resulting code that is generated. The store in the rematerializable section is necessary because of the load for the use of the live range x3.
4.4 Aggressive code motion of loads and stores

In Section 3.6.2 we discussed Chow and Hennessy’s method for moving loads and stores at basic-block boundaries to minimize their runtime cost. The rule they use for moving loads and stores is that a load can be moved as long as there is at least one predecessor that is part of the live range, with a similar restriction on successor blocks for the stores. The reason this restriction exists is to limit the negative effect of code motion, which may introduce extra basic blocks, and thus extra control flow operations, if it has to split an edge. However, we noticed that we were missing opportunities to improve the code by following this restriction. The specific opportunity we saw was to replace a store at the end of a basic block followed by a load at the successor block for the same live range. This situation occurs when a live range is split and its two parts assigned different registers. To take advantage of this opportunity, we use aggressive code motion, which always moves loads and stores onto an edge and then attempts to change a load/store pair into a register-to-register copy. Figure 4.5 shows a full example of aggressive code motion.

4.4.1 Replacing store/load pairs

The key problem to solve when using aggressive code motion is finding and replacing a store followed by a load of a live range. Our idea is to first examine all of the loads and stores moved onto a single edge and look for pairs that operate on the same live range, then find an order that will permit all the copies to be executed without prematurely overwriting a register. It may be that a correct order cannot be found for all copies, in which case we revert to using stores and loads. Finally, the copies must be inserted into the correct position. When using aggressive code motion, we always split the edge containing the loads and stores with a new basic block. This
Figure 4.5: This example shows the effects of aggressive code motion. Part (a) is the original code with five live ranges. Part (b) shows the code after colors are assigned and x has been split. Part (c) is the rewritten code and Part (d) shows the code after aggressive code motion. The load for x2 has been changed to a copy. The store must remain so it can be loaded by x3.

simplifies the code insertion since we can always safely insert stores at the top of the block and loads at the bottom of the block. The copies are inserted so that they are sandwiched between the stores at the top and the loads at the bottom.

We use Algorithm 17 to determine the correct order for inserting copies. The difficulty arises when a register written to by one copy is also read in another copy. In this case we must take care that the copy that reads the register is executed before the copy that writes to the register. That is, we must ensure that we generate that code so that there is an anti-dependence between the two copies. Three special cases need consideration:

1. Direct cyclic-dependence

\[ R1 \Rightarrow R2 \]
\[ R2 \Rightarrow R1 \]
2. Indirect dependence

\[ R_1 \Rightarrow R_2 \]
\[ R_3 \Rightarrow R_1 \]

3. Indirect cyclic-dependence

\[ R_1 \Rightarrow R_2 \]
\[ R_2 \Rightarrow R_3 \]
\[ R_3 \Rightarrow R_1 \]

In the case of a direct cyclic-dependence we revert to using stores and loads. Alternatively, we could use a swap instruction to swap the values, or a series of xor instructions to swap the values. With an indirect cyclic-dependence we also revert to using loads and stores. In this case we would need a multi-register swap instruction (or a temporary register). With an indirect dependence, we attempt to order the copies so that the copy that reads the register comes before the copy that writes to the register.

The algorithm works by examining all the copies that must be inserted and building an ordered list of copies. For each copy, it attempts to insert it as late as possible into the list. At each copy, it examines all the copies already inserted into the ordered list. If it finds a direct cyclic-dependence then it reports failure immediately. Otherwise, it checks to see if it needs to be inserted after the copy, and if so ensures that it has not already been inserted. If it finds a copy that it must proceed, then it inserts itself into the sorted list at that point. If it reaches the end without being inserted, then it can be appended to the list, because there are no dependencies with any copy already in the list.

We can verify that this algorithm correctly handles the three problem cases described above. Case (1) is tested for explicitly and we return failure if it is found. Case (2) will be handled correctly, since regardless of which copy is inserted into the
list first, the other copy will be put either before or after it according to the final
two tests in the inner loop. Case (3) will also be handled correctly, since any of the
two copies can be inserted into the result list, but insertion of the third copy will fail. If we have already inserted the copies (R2, R3) and (R1, R2), then when we attempt to insert (R3, R1), we will try to put it before (R2, R3). But, when we encounter (R1, R2), the second check in the inner loop will notice that it should be inserted after this copy, but that it has already been inserted. Thus, in all three cases, the OrderCopies algorithm will perform correctly. Once the copy-order list has been constructed, we can simply insert the copies between the stores and the loads according to that list.
Algorithm 17: OrderCopies

Input: copies - list of copies (r1,r2) where r1 is copied into r2
Output: ordered_copies - list of copies in the correct order
Output: true if the copies were ordered successfully, false otherwise

ordered_copies ← []
foreach (r1, r2) ∈ copies do
    // insert into the correct position in ordered_copies
    inserted ← false
    foreach (R1, R2) ∈ ordered_copies do
        // 1. check for direct cyclic-dependence
        if r1 = R2 and r2 = R1 then
            return false
        end
        // 2. see if (r1,r2) must be inserted after (R1,R2)
        if R1 = r2 then
            if inserted = true then
                return false
            end
        end
        // 3. see if (r1,r2) must be inserted before (R1,R2)
        if R2 = r1 then
            if inserted = false then
                insert into ordered_copies directly before (R1,R2)
                inserted ← true
            end
        end
    end
    // if not inserted yet, then insert now
    if inserted = false then
        append (r1, r2) onto ordered_copies
    end
end
return true

4.4.2 Cleanup

To increase the profitability of aggressive code motion, we must perform some cleanup after register allocation. There are two areas where cleanup is helpful: dead-
store elimination and basic-block elimination. Dead stores result from the restriction that when we convert a store/load pair into a copy, we must leave the store intact. The load can safely be deleted, but the store may be necessary for correctness as shown in Figure 4.5. Rather than try to determine whether the store is still necessary each time we convert a load/store pair into a copy, we run a post pass to clean up useless stores. Rematerialization will also benefit from this pass as discussed in Section 4.3.

Since edges are always split with a basic block when using aggressive code motion, we clean up useless blocks when possible. In particular, we can eliminate any block that has a single predecessor and a single successor by moving that block’s instructions to its successor. This lowers the number of jumps executed in the program and reduces the size of the generated code. The algorithm used for this transformation is called “Clean”, and is documented by Cooper and Torczon [29].

4.5 Optimistic simplification

There are some interference graphs that a Chaitin-Briggs allocator will successfully color that a priority-based allocator will fail to color. Consider the interference graph in Figure 4.6. The graph shows six live ranges, where each live range except the end points have two interferences. This interference graph can be easily colored with two colors as shown in Part (b) of the image. A Chaitin-Briggs allocator will always find a coloring for this graph. However, the priority-based allocator may fail to color this graph. The key difference is the order in which the nodes are colored. A Chaitin-Briggs allocator performs a simplification step in which it removes each node whose number of neighbors is less than the number of colors, since these nodes can always be given a color no matter what colors are assigned to its neighbors. These nodes are placed on a stack and colored in reverse order of removal once all nodes have been
removed from the graph. In the case of Figure 4.6, the allocator would have removed all of the nodes, because after each node is removed, it reduces the degree of all its neighbors; removing A allows us to remove B, and so on until all nodes are removed from the graph. They are then put back in the graph and assigned an available color, which is guaranteed to exist.

\[ \text{Figure 4.6: An interference graph with six live ranges. Part (a) shows the original interference graph. Part (b) shows a coloring of the graph that will be found by a Chaitin-Briggs allocator. Part (c) shows how a priority-based allocator can fail to color the graph, when colors are assigned in the order E,B,C there is no color left for D.} \]

In contrast to this approach, a priority-based allocator does not remove these unconstrained nodes from the graph. The nodes will be separated into two sets so that the constrained nodes will be assigned colors before any unconstrained nodes. If the priorities are such that the live ranges are assigned colors in the order E,B,C, then we can get the situation in Part (c) of Figure 4.6, where there is no color left for D.
D, and it must be spilled. This is surprising, because we expect to easily color such a simple graph. Our observation of this case led us to develop a form of Chaitin-Briggs style simplification that works well in the context of a priority-based allocator, so that we would have a better chance of coloring more interference graphs.

Our approach is to throw away the unconstrained list and instead remove nodes from the graph when they would have been placed on the unconstrained list. When a node is removed from the graph it may make some of its neighbors unconstrained, so they too will be removed from the graph. As the nodes are removed from the graph, they are placed onto a coloring stack. After all constrained live ranges have been assigned a color, we work through the nodes in the coloring stack, assigning colors when possible – it may be the case that during the course of the algorithm, an unconstrained node becomes constrained after a live range has been split. Instead of attempting to pull the node out of the coloring stack and insert it back into the graph, we make the optimistic assumption that any node which is unconstrained will never become constrained in the future. This greatly simplifies the implementation. However, if a node truly is constrained and there is no color available when we pop it from the coloring stack, we must spill the node. With these changes, the priority-based allocator will successfully color the graph in Figure 4.6, since it will simplify all the nodes and then color directly from the stack.

Our implementation differs a little bit from the description above in that we do not actually remove the nodes from the graph. Instead, we have a field in the live range which keeps track of the number of simplified neighbors. When a node is “removed” from the graph through simplification, we increment the count of simplified neighbors in each neighbor of the simplified node. Then, when deciding if a live range is constrained we subtract the number of simplified neighbors from the total neighbor count. It is useful to keep the edges in the interference graph so that a color can be
easily assigned when the live range is popped from the coloring stack. Also, the live range will continue to have its interferences updated as its neighbors are split, which will give more accurate information when it comes time to assign a color. When live ranges are split and they must rebuild the interference lists, they should also update the count of simplified neighbors to reflect the changed interference graph.

Since the unconstrained list is only updated in a few places, it is easy to change those updates to handle optimistic simplification. When a value was previously added to the unconstrained list, it can now call a simplification routine to “remove” it from the graph as described above. Additionally, because of the optimistic assumption, no live ranges would ever be removed from the unconstrained list, so the UpdateConstrainedLists routine can ignore any checks to remove live ranges from the unconstrained list. Finally, the initial distribution of live ranges to constrained and unconstrained lists can ignore any inserts to the unconstrained lists and instead call the simplification routine.

4.6 Smart loading into copies

A simple improvement that is quite useful and easy to implement is to load directly into the destination of a copy when the source has been spilled. This idea was suggested by Chaitin in the description of his allocator [18]. The idea is illustrated with a simple example shown in Figure 4.7. The figure shows the code generated if the live range x has been spilled and y assigned a register. We can eliminate the copy by loading directly into the register for y. This is always profitable since we are eliminating an instruction that would otherwise be executed. This idea is simple and easy to implement; we include it in the thesis for completeness.
Figure 4.7: Smart loading of copies. On the left is the original code before allocation. The middle shows the code after allocation, assuming x is spilled and y is given a register R1. On the right is the desired code, which eliminates the copy from a temporary register to an allocated register.

### 4.7 Limiting the number of splits

As mentioned in Section 3.5, splitting a live range can leave one of the live ranges with no occurrence of the variable in any blocks of the live range. These live ranges are obviously useless and are not considered for coloring. The problem is that we still have to do the work in order to split the live range. This can be a problem when the live range contains many empty blocks and each time it splits, it is partitioned into two live ranges one which contains one empty block and the other which contains all the remaining blocks. In the worst case, we may end up splitting the live range $N - 1$ times, where $N$ is the number of basic blocks in the original live range. Figure 4.8 demonstrates this case.

We would like to reduce the number of live ranges created by splitting so that we reduce the overall number of splits. Live-range trimming, introduced in Section 4.2, has the added benefit of eliminating zero-occurrence live ranges resulting from splits. Trimming will remove excess blocks in a live range that do not add benefit to the live range. The blocks removed are exactly the types of blocks that produce zero-occurrence live ranges. Trimming these blocks gets rid of zero-occurrence live ranges and reduces the overall number of splits. Figure 4.9 shows how trimming gets rid of the zero-occurrence live ranges produced in the example of Figure 4.8.

While the example in the figure may seem contrived, we have seen such a degenerate case appear during a run of our adaptive compiler on the fpppp benchmark.
Figure 4.8: Example showing how splitting can create many zero-occurrence live ranges. In this example there is only one register for allocation. The register is assigned to the live ranges in the order d, c, b, a. Each time a register is allocated, the live range x must be split, for a total of five splits resulting in four zero-occurrence live ranges.

The combination of parameter settings caused the allocator to perform 134,403 splits, producing 130,305 zero-occurrence live ranges and taking almost 57 minutes to complete. Adding live range trimming reduced the number of splits to 4,107 with no zero-occurrence live ranges, and reduced the allocation time to just over a minute.

We also experimented with limiting the number of splits the allocator performed by capping it with an absolute upper bound. Once the bound is reached no more splitting is done, and any live range that would be split is instead spilled. The idea is that splitting done early in the allocation process is more likely to produce higher priority live ranges that can be colored, while the later splits are for lower priority live-ranges and may not produce profitable live ranges after splitting. One could also limit the number of splits to be some constant factor of the initial number of live ranges. Limiting splits is done simply to speed the allocation, and may produce inferior results. Our experience suggests that limiting the number of splits to 1,000
4.8 Enhanced register promotion

As described in Section 3.6.1, we perform local allocation and spill code insertion after completing the global priority-based allocation. When a live range is spilled, it must be loaded into a temporary register before each use and stored back to memory after each definition. As we walk through the blocks during spill code insertion, we keep track of which live ranges are in a temporary register. If we reach the end of a block that has a single successor, we do not need to clear out our knowledge of
which live ranges are in a temporary register, because control must flow through the
current block to get to the successor block. It is possible that both blocks previously
were part of a single live range, and have now been split into two different live ranges.
Additionally, the live range may be allocated a register in the successor block, but not
the predecessor block. In this case, the live range will have a load at the top of the
successor block. If we reach the end of the predecessor block with the live range in a
temporary register, then we would like to delete the load at the top of the successor
block and insert a copy from the temporary register to the assigned register in the
successor block. We call this transformation “enhanced register promotion”.

The transformation is implemented during the spill-code insertion phase of the
algorithm. When we reach the end of a block that has a single successor, we check to
see if any live range stored in a temporary register is allocated in the successor block.
We insert a copy onto the edge between the predecessor block and the successor block
and delete the load on that edge. These copies are then handled by the same code
that deals with loads and stores that have been moved onto an edge as described in
Section 3.6.2.
Chapter 5

Adaptive priority-based allocator

This chapter describes how to tune the priority-based allocator using adaptive compilation. *Adaptive compilation* is a technique for optimizing code using repeated compilation, performance feedback, and search. In this chapter, we describe a generic framework for an adaptive compiler, and show how the priority-based allocator can be tuned using this framework.

5.1 Adaptive compilation framework

Our adaptive framework consists of three components: a search module, a static compiler, and a results database. The basic work flow is given in Algorithm 18, and displayed graphically in Figure 5.1. The goal of adaptive compilation is to find a set of input parameters that minimizes some objective function. In this thesis, the objective function is the dynamic instruction count for an execution of the program. Each file in the input program is associated with a combination of parameters that control some aspect of compilation. In this thesis, the parameters control the behavior of the priority-based allocator. We hold the rest of the compiler’s parameters constant. To minimize the objective function, we search for the best combination of parameters for each file.

The search begins by associating a random combination of parameters with each input file to avoid biasing the search with preconceived notions of good parameter
settings. The main loop compiles each file using its associated set of parameters, and then generates an executable program from the resulting object files. The program is run and performance data is collected for each source file. The performance data is added to the results database which contains a record associating the file and parameters with the collected performance data. The database is used to avoid repeated compilations with the same parameters.

After recording the performance data, a new search state is generated based on the current state, the current results, and previous results stored in the database. The details of generating the next state are discussed in Section 5.3. The main loop continues until we use up the allotted time for adaptive compilation. Time can be limited by wall-clock time, number of evaluations, or any desired limit. The search state maintains the best combination of parameters found for each file and returns them at the end of the search. The remainder of this chapter describes how we integrated the priority-based allocator into the adaptive framework.
Algorithm 18: Adaptive compilation

```
Input : files - set of files to adaptively compile
Output: the best found parameters and corresponding performance measure for each file

Data: db - database containing performance results of previous executions
Data: search-state - structure encapsulating the current state of the search

// initialize search state
search-state <- new search state with random parameters for each file

// run the measure-record-search cycle
repeat
    foreach (file, parameters) ∈ search-state.need-eval do
        | COMPILE(file, parameters)
    end
    link object files into an executable
    results <- run the executable and measure performance
    foreach (file, parameters, perf-data) ∈ results do
        | DB-INSERT(db, file, parameters, perf-data)
    end
    search-state <- GENERATE-NEXT-STATE(search-state, results, db)
until execution budget is reached

// collect best results
(best-data, best-params) <- BEST-RESULTS(search-state)
return (best-data, best-params)
```

5.2 Priority-based allocator search space

The previous section discussed our general framework for adaptive compilation. To use the framework with the priority-based allocator, we must define both the search space for the allocator and how the search moves around that space.

A point in the search space for the priority-based allocator is a setting of parameters that control its behavior. The entire search space consists of every possible combination of settings for all the parameters. We have exposed a set of parameters in the allocator that can be configured with command line arguments. When the file
is compiled, the allocator is passed command line flags corresponding to the combination of parameters currently under examination. The neighborhood structure describes how the points in the search space are connected, which dictates the order that parameter combinations are tried, since the search can only move to a point that is in its neighborhood. The neighborhood structure is defined in Section 5.2.3. We discuss our choice of parameters exposed in the priority-based allocator below.

5.2.1 Allocation parameters

We expose as many choices as possible to the adaptive compiler without unnecessarily increasing the size of the search space. We designed the allocator so that various aspects of its behavior can be controlled by command line parameters. We used three guiding principles when choosing the aspects of the allocator to expose as parameters. First, we looked for places in the allocator where we choose among several competing values, and there is no obvious best choice. Basic-block size and coloring heuristic are
examples of this type of parameter. The second case we looked for was choices that could sometimes hurt the allocation and sometimes help the allocation, where the exact outcome is hard to predict. Aggressive code motion of loads and stores is an example of this type of parameter. Whether it hurts or helps depends on the number of opportunities to turn store/load pairs into copies. The third case of parameters is for behaviors in the allocator that interact with each other. For example, global allocation of local live-ranges and the number of local registers should be not be set in isolation from each other. The setting of one parameter will influence the profitable values of the second parameter.

Once we have identified the parameters to include, the adaptive compiler finds a good combination of parameters. We included as many parameters as we could identify, subject to the three principles stated above. The precise parameters chosen are discussed below and summarized in Table 5.1

**basic-block size** The size of a basic block changes the number of interferences as described in Section 3.7. Reducing the block size increases the accuracy of the live ranges but also changes the priority for a live range, since it is normalized by the number of blocks it spans. It also affects placement of the spill code that is inserted at basic-block boundaries when a live range is split.

**reserved local registers** We can reserve a number of registers for local live-ranges and spilled global live ranges as discussed in Section 3.4.3. Reserving local registers reduces the number of registers used for global allocation.

**aggressive motion of loads and stores** We try moving all loads and stores onto edges in an attempt to substitute a copy for a store in a predecessor block that is followed by a load in the current block. This transformation may increase the number of control-flow operations because of the need to split edges.
rematerialization  Rematerialization attempts to recompute values rather than spill them to memory. Live ranges are split into rematerializable sections before allocation. Splitting live ranges may have a negative effect if the splitting does not help and they cannot be joined back together.

live-range trimming  Live-range trimming removes useless blocks from a live range after it has been split. Splitting a live range may remove an opportunity for aggressive code motion to change a load/store pair into a copy by trimming the block containing the memory operation.

coloring heuristics  We use four different coloring heuristics.

1. Choose the first available color.

2. Find a neighbor in the interference graph with the largest number of forbidden colors. Choose a color that is already in the forbidden set of that live range. The goal is to keep the most constrained live-range colorable for as long as possible.

3. Choose the color that is in the forbidden set of the most neighbors in the interference graph. The goal is to minimize the number of live ranges further constrained by the current assignment.

4. Choose the color from a live range that was created by splitting the current live-range. The goal is to undo the detrimental effects of early splitting in rematerialization. This is similar to biased coloring used in the Chaitin-Briggs allocator [11].

splitting heuristic  We use two different heuristics when deciding which blocks to include in the new live-range when splitting.
1. Standard splitting as originally published by Chow and Hennessy, and described in section 3.5.

2. Include blocks in the new live-range by searching both up and down the control-flow graph. The published algorithm only looks down the control-flow graph at the successors of the starting block.

**global allocation of local live-ranges** We have a choice between removing local live-ranges from the interference graph and using local allocation, or leaving them in the graph and treating all live ranges as if they were global.

**optimistic simplification** Optimistic simplification permanently removes live ranges from the interference graph if they have fewer than \( k \) neighbors, where \( k \) is the number of colors. The standard method leaves the live ranges in the graph in case they later become constrained due to splitting.

**allocate all unconstrained live-ranges** It is possible that some unconstrained live-ranges may have negative priority. This can happen if a large live-range is split into a smaller unconstrained live range, but requires more loads and stores at its boundaries than the number of uses and definitions in the live range. However, not allocating these live ranges may remove an opportunity for inserting a copy with aggressive code motion.

**enhanced register promotion** Live ranges that are spilled are handled by the local allocator so that repeated references in the same basic block do not generate multiple loads. Enhanced register promotion tries to eliminate a load at a live-range entry when the live range is in a temporary register at the predecessor block by changing the load into a copy. It requires aggressive code motion to be effective, and this may degrade performance. In addition, the opportunity for
eliminating a load at a live-range entry depends on the basic-block size, which dictates the live-range boundaries.

**loop-depth weight** The priority function weights each use, definition, and memory operation in a live range with the loop nesting depth of that occurrence. If the reference occurs at loop depth $d$, then it is weighted by $10^d$. This parameter uses multiple bases for the loop-depth weight. We consider weights of $1^d, 5^d, 10^d$, and $20^d$.

**prefer spilling clean local values** This parameter affects spilling decisions in the local allocator. The standard local allocator spills the live range whose next use is farthest in the future. This flag tells the local allocator to also consider whether the live range has been written to when making spilling decisions. If the live range has not been written to, then it does not need a store when it is evicted from a temporary register.

**priority function** We make available five priority functions, all of which are modifications of the basic priority function described in Section 3.1.2.

1. Use the standard priority function described by Chow and Hennessy.
2. Use the standard priority function, but do not normalize based on live range size. This function does not punish large live-ranges.
3. Take the log of the priority before normalizing by the live range length. This was suggested by a similar priority function used in GCC [46].
4. Square the size of the live range before normalizing. This makes larger live-ranges less desirable.
5. Combination of 3 and 4.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>basic block maximum size</td>
<td>integer</td>
<td>11</td>
</tr>
<tr>
<td>l</td>
<td>number of registers reserved for local allocation</td>
<td>integer</td>
<td>variable</td>
</tr>
<tr>
<td>e</td>
<td>use aggressive motion of loads and stores</td>
<td>boolean</td>
<td>2</td>
</tr>
<tr>
<td>z</td>
<td>use rematerialization</td>
<td>boolean</td>
<td>2</td>
</tr>
<tr>
<td>t</td>
<td>use live-range trimming</td>
<td>boolean</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>coloring heuristic</td>
<td>integer</td>
<td>4</td>
</tr>
<tr>
<td>s</td>
<td>live-range splitting heuristic</td>
<td>integer</td>
<td>2</td>
</tr>
<tr>
<td>g</td>
<td>use global allocation for local live ranges</td>
<td>boolean</td>
<td>2</td>
</tr>
<tr>
<td>o</td>
<td>use optimistic simplification</td>
<td>boolean</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>allocate all unconstrained live ranges</td>
<td>boolean</td>
<td>2</td>
</tr>
<tr>
<td>n</td>
<td>use register promotion</td>
<td>boolean</td>
<td>2</td>
</tr>
<tr>
<td>d</td>
<td>loop depth weight</td>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>k</td>
<td>prefer spilling clean local values</td>
<td>boolean</td>
<td>2</td>
</tr>
<tr>
<td>x</td>
<td>priority function</td>
<td>integer</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 5.1: Priority-based allocation parameters used in the adaptive compiler. The type of parameter indicates how it is specified on the command line to the allocator. The settings column lists the number of different settings for the parameter. The number of settings for local registers is explained in Section 5.2.2.

### 5.2.2 Search space size

The size of the search space is the total number of distinct parameter combinations. The number of combinations depends on the number of registers, because the number of settings for local registers decreases as the total number of registers decreases. The number of reserved local registers is the only parameter that varies with the number of registers. We are using partitioned register sets in our machine model so that integer values and floating point values use different register files. We can reserve a different number of locals for each class, which increases the number of parameter settings for local registers.

We always require at least four integer and six floating point registers for allocation. The four integer registers use two for holding spilled values, one to hold the frame pointer, and one for global allocation. The register for global allocation is
needed due to a limitation in the implementation of our set data structure which does not allow for zero sized sets. We have to reserve more floating-point registers because a double value occupies two registers. Four of the reserved floating-point registers are used to hold spilled values, and the other two are used for global allocation. When reserving local floating-point registers, we always reserve in blocks of size two so that they can hold a double value. With these restrictions, we can reserve up to an additional \( k - 4 \) integer registers and \( k - 6 \) floating point registers.

The total number of settings for the local registers parameter is the number of settings for local integer registers times the number of settings for local floating point registers, since we allow a different number to be reserved for each class. The total size of the search space is calculated by multiplying the number of settings available for each parameter. Table 5.2 lists the number of settings for local registers and the total search space size when there are 32, 24, 16, and 8 total registers. We also list the number of years required to exhaustively search the entire space for the best solution. This “back of the envelope” calculation gives an intuitive sense of the size of the space. We calculated this number assuming that one loop of the adaptive compiler takes a minute to complete, which is a reasonable assumption for small to medium sized benchmarks.

<table>
<thead>
<tr>
<th>Registers</th>
<th>Total size</th>
<th>Local settings</th>
<th>Neighbors</th>
<th>Years for exhaustive search</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>182,927,360</td>
<td>406</td>
<td>434</td>
<td>348</td>
</tr>
<tr>
<td>24</td>
<td>94,617,600</td>
<td>210</td>
<td>238</td>
<td>180</td>
</tr>
<tr>
<td>16</td>
<td>35,143,680</td>
<td>78</td>
<td>106</td>
<td>66</td>
</tr>
<tr>
<td>8</td>
<td>4,505,600</td>
<td>10</td>
<td>38</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 5.2: Search space size for various number of registers. The local settings column lists the number of settings for the local variables parameter, calculated as \(((k-4)+1)\times\left(\frac{k-6}{2}+1\right)\). The neighbors column lists the number of neighbors for each point in the space. The years for exhaustive search is calculated with the premise that it takes a minute to complete one iteration of the compile/run/record in the adaptive compiler.
5.2.3 Neighborhood structure

The previous sections described the elements in the search space as a combination of parameters for the priority-based allocator. The search algorithm needs to know how it can move around in the search space. The *neighborhood* describes which points in the search space are connected.

We define the neighbor of a point to be all points that can be reached by changing a single setting in one parameter. Neighbors can be generated by first selecting a parameter and then choosing a new setting. If we imagine a combination of parameters is represented as a string, where each setting of a parameter is a unique character, then the neighbors of a point will be all the strings that are a Hamming distance of one from the string representation of that point. The number of neighbors is calculated by summing the number of choices for each parameter that are different from the current setting. Table 5.2 lists the number of neighbors when allocating with 32, 24, 16, and 8 registers.

5.3 Search algorithm

The adaptive compiler framework can be used with a variety of search algorithms by changing the `GENERATE-NEXT-STATE` function in algorithm 18. Grosul explored a variety of search algorithms for adaptive compilers, and found random-restart impatient-hillclimbers to be an effective method for finding good solutions [34]. We follow his lead and use a random-restart impatient-hillclimber as the search mechanism in our adaptive compiler.
5.3.1 Generic random-restart impatient-hillclimber

The basic structure for a random-restart impatient-hillclimber is given in Algorithm 19. The outer loop controls the number of random restarts for the hillclimber. For each search, a random initial state is chosen. Neighbors of the state are generated until either a better neighbor is found or the patience limit is reached. If a better neighbor is found, then the its value is recorded and the hillclimber moves to that state and resets its patience. The hillclimber is greedy because it moves to the first neighbor that is better than the current state. Once the current descent is finished, the globally best value is updated if the current value is better. The globally best value is returned after the restart limit is reached. Section 5.3.3 describes how the search algorithm in the adaptive compiler fits into the generic hillclimber framework.
Algorithm 19: Random-restart hillclimber

**Input**: patience-limit - number of neighbors to explore before giving up  
**Input**: num-restarts - number of random restarts  
**Output**: the best state and its corresponding value

for 1 to num-restarts do  
    state ← choose a random state  
    best-value ← Value(state)  
    repeat  
        neighbor ← Next-Neighbor(state)  
        // see if this neighbor has a better value  
        if Value(neighbor) < Value(state) then  
            best-value ← Value(neighbor)  
            state ← neighbor  
            patience ← patience-limit  
        else  
            patience ← patience - 1  
        end  
    until patience = 0 or best-value is good enough  
    // update the global best  
    if best-value < global-best-value then  
        global-best-value ← best-value  
        global-best-state ← state  
    end  
end  
return (global-best-value, global-best-state)

5.3.2 Search state

Before describing the details of the hillclimber search used in the adaptive compiler, it is useful to understand the underlying data structures. The search state is the most important of these because it captures all the data needed to continue the search. The adaptive compiler needs to maintain enough information about the state of each search to resume a search after evaluating allocation parameters by compiling and running the program.

The search-state structure shown in Figure 5.2 encapsulates the state of the
hillclimber search. The search state is organized as a two level structure; each file maintains its own separate search state. The first level is a convenient wrapper for the file-search-state structures, which hold the state for each individual file search. It also holds a list of (file, parameters) tuples, which are the next combination of parameters that should be evaluated for each file.

The second tier is the state of the search for each file. These searches are all run independently from each other and need to keep their own private state. The current position in the search space and its fitness are tracked, along with the number of restarts and number of neighbors examined since the previous downward move in the space (patience). Additionally, we track the best parameter string we have found on any descent and its corresponding fitness. We also use the file-search-state to store configuration settings, such as the patience limit. Section 5.3.3 describes how these structures are maintained during the search.

5.3.3 Random-restart impatient hillclimber in the adaptive compiler

The implementation of the random-restart impatient hillclimber is embedded into the generate-next-state function in the adaptive compiler. This function is called from the adaptive compiler when the next set of parameters are needed for evaluation. The search-state is a structure that encapsulates the state of the hillclimber search. The current state is passed to the generate-next-state function which performs the next step of the search and returns the updated state.

The implementation of generate-next-state is shown in algorithm 20. We begin by looping over all of the results obtained from the last performance evaluation. The individual file-search-state structures are updated by the apply-result function, which takes the performance evaluation and moves to the new point in the search space if it is better than the current location. We then collect the next set of
type search-state =
{
    states : file-search-state list;
    need-eval : (string * string) list;
}

type file-search-state =
{
    (* allocation parameters are represented as a command line string *)
    current : string;
    fitness : float;
    patience : int;
    restarts : int;
    best-fitness : float;
    best-params : string;
    file : string;
    (* configuration fields *)
    patience-limit : int;
}

Figure 5.2: Structures representing the current state of a search.

parameters that need to be evaluated. The NEXT-EVAL function handles the details of finding the next set of parameters. This function returns the updated file search state and the parameters to evaluate, which are saved in the search-state structure. After all of the results have been processed, the updated search state is returned.
Algorithm 20: GENERATE-NEXT-STATE

Input: search-state - current state of the search
Input: results - the performance results from the most recent evaluation
Input: db - database containing performance results from previous executions
Output: the next state

search-state.need-eval ← ∅

foreach (file, parameters, perf-data) ∈ results do
    file-result ← (file, parameters, perf-data)
    updated-state ← APPLY-RESULT(search-state.states[file], file-result, db)
    (updated-state', params-to-eval) ← NEXT-EVAL(updated-state, db)
    search-state.states[file] ← updated-state'
    search-state.need-eval ← search-state.need-eval ∪ {(file, params-to-eval)}
end

return search-state

Algorithm 21: APPLY-RESULTS

Input: state - a file-search-state
Input: file-results - the results from the last performance evaluation
Input: db - database containing performance results from previous executions
Output: the updated state for this file

(file, parameters, perf-data) ← file-results
if state.fitness has been recorded then
    if perf-data < state.fitness then
        state ← MOVE-TO-NEIGHBOR(state, parameters, perf-data)
    end
else
    // first evaluation in a new descent
    state.fitness ← perf-data
end

return state

The NEXT-EVAL function is shown in Algorithm 22, and is the heart of the hill-climber search. It works much like the generic hillclimber except that it uses a db to check for previous evaluations. The goal of the NEXT-EVAL function is to find an unevaluated neighbor of the current state. Each time a neighbor is generated the db is
queried to check for a past evaluation of that neighbor. If the neighbor is found in the database and its fitness is better than the current state, then we immediately move to that neighbor and search again for an unevaluated neighbor of that state using a recursive call to NEXT-EVAL. We continue to search for an unevaluated neighbor in this manner until we run out of patience and perform a random restart.

<table>
<thead>
<tr>
<th>Algorithm 22: NEXT-EVAL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong> : state - a file-search-state</td>
</tr>
<tr>
<td><strong>Input</strong> : db - database containing performance results from previous executions</td>
</tr>
<tr>
<td><strong>Output</strong>: the next state for this file</td>
</tr>
<tr>
<td>if state.patience &gt; 0 then</td>
</tr>
<tr>
<td>// pick a new state</td>
</tr>
<tr>
<td>neighbor &lt;- NEXT-NEIGHBOR(state)</td>
</tr>
<tr>
<td>state.patience &lt;- state.patience - 1</td>
</tr>
<tr>
<td>if neighbor is in db then</td>
</tr>
<tr>
<td>neighbor-fitness &lt;- DB-LOOKUP(db, state.file, neighbor)</td>
</tr>
<tr>
<td>if neighbor-fitness &lt; state.fitness then</td>
</tr>
<tr>
<td>state &lt;- MOVE-TO-NEIGHBOR(state, neighbor, neighbor-fitness)</td>
</tr>
<tr>
<td>return NEXT-EVAL(state, db)</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>return (neighbor, state)</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>// random restart</td>
</tr>
<tr>
<td>state.current &lt;- random set of parameters</td>
</tr>
<tr>
<td>state.fitness &lt;- None</td>
</tr>
<tr>
<td>state.patience &lt;- state.patience-limit</td>
</tr>
<tr>
<td>state.restarts &lt;- state.restarts + 1</td>
</tr>
<tr>
<td>if state.current is in the db then</td>
</tr>
<tr>
<td>state.fitness &lt;- DB-LOOKUP(db, state.file, state.current)</td>
</tr>
<tr>
<td>return NEXT-EVAL(state, db)</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>return (state.current, state)</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

Moving to a neighbor modifies some of the internal file search state variables.
The current fitness and parameter values are updated, and the patience is reset. We also need a check to see if the neighbor we are moving to is better than the best value found so far. If it is better, then we record the neighbor and its fitness as the best values in the file search state. The pseudo-code for these operations is shown in Algorithm 23.

\begin{algorithm}
\begin{algorithmic}
\State \textbf{Input:} state - a \texttt{file-search-state}
\State \textbf{Input:} neighbor - the neighbor we are moving to
\State \textbf{Input:} neighbor-fitness - the fitness for that neighbor
\State \textbf{Output:} the updated state for this file
\State state.current $\leftarrow$ neighbor
\State state.fitness $\leftarrow$ neighbor-fitness
\State state.patience $\leftarrow$ state.patience-limit
\If{neighbor-fitness $< state$-best-fitness} \State state.best-fitness $\leftarrow$ neighbor-fitness \State state.best-params $\leftarrow$ neighbor \EndIf
\Return state
\end{algorithmic}
\end{algorithm}
Chapter 6

Experimental results

We measured the allocation quality produced by the classic, engineered, and adapted versions of the priority-based allocator. Allocation quality was measured by counting the dynamic number of instructions executed after initial optimizations and register allocation. The priority-based register allocator was implemented in the ILOC compiler, a machine independent research compiler at Rice University. ILOC is a low-level intermediate language that resembles an abstract RISC machine. The input program initially contains an arbitrary number of pseudo-registers; they are mapped to a fixed number of machine registers by the allocator. The number and type of machine registers can be varied to allow experimentation with different machine configurations. Our experiments were run with separate register files for integer and floating point values. Each experiment was run with 8, 16, 24, and 32 machine registers in each register file. In addition, we assume that double values require two consecutive floating point registers.

Allocation performance is measured by counting the number of dynamic instructions. After allocation, the ILOC program is translated into C. The C code is compiled and executed; a global counter tracks the number of ILOC instructions executed. This number is used to compare the performance of the different allocators. The dynamic instruction count has an advantage over execution time as a performance measure because it is precise and exactly reproducible. Execution time is variable, subject to
clock accuracy and system load. These characteristics are important because we use the performance measure to guide the adaptive compiler's search.

We compare performance of the allocators on a function-by-function basis. The instruction counts for each allocator are normalized by the corresponding instruction count of the Chaitin-Briggs allocator. Aggregate numbers are calculated by taking the geometric mean of the performance ratio for each function.

We use a variety of FORTRAN programs as benchmarks. The programs include entries from SPEC 92, SPEC 95, and several kernels from a numerical analysis book by Forsyth, Malcolm, and Moler (FMM) [16, 17, 33]. These programs include a total of 121 distinct routines. A brief description and the number of files in each benchmark is given in Table 6.1. Due to a limitation of our intermediate representation, not all of files can be compiled for each register setting. The files that can not be compiled with fewer registers are excluded in the results reported for that specific number of registers.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Routines</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fmin</td>
<td>1</td>
<td>finds the minimum of a function</td>
</tr>
<tr>
<td>rkf45</td>
<td>3</td>
<td>Runge-Kutta-Fehlberg solver for ODE</td>
</tr>
<tr>
<td>seval</td>
<td>2</td>
<td>cubic spline function evaluator</td>
</tr>
<tr>
<td>solve</td>
<td>2</td>
<td>solver for systems of linear equations</td>
</tr>
<tr>
<td>svd</td>
<td>1</td>
<td>singular value decomposition of rectangular matrix</td>
</tr>
<tr>
<td>urand</td>
<td>1</td>
<td>random number generator</td>
</tr>
<tr>
<td>zeroin</td>
<td>1</td>
<td>finds a zero of a function</td>
</tr>
<tr>
<td>SPEC 92</td>
<td></td>
<td></td>
</tr>
<tr>
<td>doduc</td>
<td>38</td>
<td>Monte Carlo simulation of a thermo-hydraulic model</td>
</tr>
<tr>
<td>fpopp</td>
<td>10</td>
<td>calculates multi-electron integral derivatives</td>
</tr>
<tr>
<td>SPEC 95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>applu</td>
<td>14</td>
<td>solves partial differential equations</td>
</tr>
<tr>
<td>wave5X</td>
<td>45</td>
<td>solves Maxwell's equations</td>
</tr>
</tbody>
</table>

*Table 6.1: Benchmarks used in collecting experimental results*
Each routine in a benchmark was allocated separately; no inter-procedural information was used during allocation. Prior to allocation, each routine passes through a standard sequence of optimizations: dead-code elimination [38], value numbering [3], lazy code motion [39], constant propagation [54], copy coalescing [47], and another pass of dead-code elimination. After allocation, the code passes through two optimizations for clean up: dead-code elimination and empty basic block removal [29]. These passes are needed because several of the techniques described in Chapter 4 expect a post-pass that removes useless code. The specific cases where clean up is needed are discussed in Section 4.4.2. The following sections give results for the three versions of the priority-based allocator discussed in this thesis.

6.1 Classic

The classic priority-based allocator performs poorly in comparison with Chaitin-Briggs. The performance gap decreases from approximately 16% to 9% worse when moving from 32 to 8 registers. This result can be explained by the structure of the allocation problem; when there is only 8 registers, both allocators will assign a register to the most obviously profitable values and the remaining values will be spilled. The priority-based allocator is less likely to make a poor choice when only 8 registers are available. When 32 registers are available it has more opportunity to prematurely allocate a register and overly constrain the remaining live ranges when a judicious split of a live range could have simplified the later allocation. In contrast, Chaitin-Briggs attempts to pack all values into some register, and only spills when that becomes impossible. With more registers this packing strategy has a greater chance of success and therefore generates fewer spills. Figure 6.1 shows these results.

We also experimented with limiting the number of instructions in a basic block.
Reducing the size of basic blocks increases the granularity of the interference graph. We expect to see that increasing the granularity improves the performance. Our results for the classic version show that performance is relatively constant regardless of the basic-block size. The main problem is the interaction between the local and global allocation phases. The classic allocator treats global and local values separately. Only the values that are live across a basic block are considered by the priority-based allocator; local values are allocated in a separate pass after global allocation. The classic allocator does not reserve enough registers for local allocation which results in a poor local allocation and leads to poor overall performance. We compute which values are local before limiting basic-block size; the decrease in block size does not change the number of local values. We can greatly improve the performance of the priority-based allocator by either allocating local values in the global allocator or increasing the number of registers for local allocation. These changes are discussed in the next section.

6.2 Engineered

To test the effects of various parameters, we ran the classic allocator and enabled one change at a time. These runs were repeated with basic-block sizes of 1, 5, and no limit. Table 6.2 lists the parameters used and how they change the allocation algorithm. The parameters in the top part of the table were used as single inputs to the allocator. We also selected a combination of parameters as input to the allocator. The combination is listed in the bottom of the table. It includes two parameters that were not used as single inputs because they both rely on the aggressive code motion parameter for profitability. This combination is referred to as the engineered version in the rest of the chapter.
Figure 6.1: Performance of the priority-based allocator with various limits on basic block size.

Our experiments found that when the parameters are applied in isolation, only two of them change the performance of the classic allocator: reserving more registers for local allocation and globally allocating local values. Applying the combination of parameters had an effect on the performance, but it also does global allocation of local values, and its improvement is largely due to this fact. In this section we examine the results of the parameter settings under the two extremes of limiting the basic-block size to one instruction, or having no size limit.

The results for unlimited basic-block size are shown in Figure 6.2. The results show that reserving two extra registers for local allocation further closes the gap between the priority-based and the Chaitin-Briggs allocator. The performance is about 5-
Table 6.2: Parameters used in the engineering version benchmarks. The top parameters were used as single inputs to the allocator. The bottom part shows a hand selected combination of parameters. The a and n parameters were not used as single inputs because they rely on other parameters to be profitable.

8% worse than Chaitin-Briggs, depending on the total number of registers used for allocation. These results show that register contention during local allocation causes a non-trivial degradation in performance in the classic allocator. Figure 6.3 shows the results when basic blocks are limited to one instruction. The improvement gain for reserving two extra registers for local allocation is roughly the same as when block size is unlimited. This result makes sense, because the coarseness of the interference graph does not effect local allocation. The number of local values remains the same, since they are determined before blocks are split. We would only expect the performance to increase if the global allocator does a better job, but we saw in Section 6.1 that the classic allocator does not improve much with smaller block size. In our experiments, reserving two registers for local allocation always improves the quality of allocation over the classic allocator.

In contrast to reserving more local registers, allocating all local values in the global allocator produces distinct results for the two different block limits. Unlimited block size results in a worse allocation than the classic allocator. Globally allocating local
values means there will be more nodes in the interference graph. The block size is not limited, so the graph will be contain many interferences. This coarseness is the cause of the poor allocation. It is even worse than the classic allocator because extra interferences cause the allocator to spill more values.

When block size is limited to only one instruction, the interference graph is no longer coarse and the global allocation method improves. With 32 and 24 registers, the allocator is only slightly worse than Chaitin-Briggs. When we move down to 8 registers, both the global allocation and combination perform worse than the classic allocator. The difficulty arises because the global allocation of locals introduces more choices, but there is a smaller margin of error because there are so few registers. Local values can have a high priority because they are only live in one block – the normalization by live-range length does not effect them. These factors combine to make it difficult for the allocator to choose the most profitable live ranges to keep in registers. Choosing the wrong live ranges results in worse performance.

The results so far have shown the difficulty in selecting a set of parameters that performs well in all cases. Global allocation of locals works, but only if you greatly reduce the size of a basic block. Our combination of parameters produces some good results, but it never does better than a simple global allocation of locals. Increasing the number of locals provides a fairly constant improvement, but it suggests that some codes may benefit from even more locals. In all these cases we still have not been able to equal or beat the Chaitin-Briggs allocator. To beat Chaitin-Briggs we need adaptive compilation.
6.3 Adapted

We added the priority-based allocator to our adaptive compilation framework as described in Chapter 5. The results in this section were gathered using two runs of the adaptive compiler for each benchmark and taking the best overall result. Based on previous experience, we use a patience of 20% for the hillclimber with a limit of 1000 evaluations for each run [34, 48]. Our success in finding good parameter combinations indicate these values also work well when searching for priority-based allocation parameters.
Figure 6.3: Performance of the priority-based allocator under various parameter settings with basic block size limited to one instruction. The settings correspond to the classic allocator (classic), reserving two extra registers for local allocation (local-2), globally allocating local live ranges (global), and the combination listed in table 6.2 (comb).

6.3.1 Performance results

The main results are shown graphically in Figure 6.4, and precise numbers are given in Table 6.3. This figure compares the results of the adapted version of the allocator with the classic and engineered versions. For the classic and engineered versions, basic blocks are limited to a single instruction. The results show that adaptive compilation was effective in closing the performance gap between the Chaitin-Briggs and priority-based allocators. The adapted version of the allocator performs slightly better than the Chaitin-Briggs allocator, regardless of the number of registers used for allocation. This contrasts with using a fixed set of parameters, which approaches
the Chaitin-Briggs allocator when using 32 or 16 registers, but deteriorates with fewer registers.

We also specifically examined the 20 largest routines that can be allocated with all four register settings (32, 24, 16 and 8 registers). We wanted to see if the results for large functions differed from the aggregate results reported in Table 6.3. The performance results for the 20 largest routines are listed in Table 6.4. The results for 24 registers is not shown to allow the data to fit in a single table, and because they are similar to the results for 32 registers. The data shows that the performance data for the large functions does not greatly differ from the aggregate results; the adapted version performs slightly better than Chaitin-Briggs, the engineered version does well except with 8 registers, and the classic version does poorly across the board.

<table>
<thead>
<tr>
<th>Version</th>
<th>32</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>geomean</td>
<td>stddev</td>
</tr>
<tr>
<td>AC</td>
<td>0.99</td>
<td>1.03</td>
</tr>
<tr>
<td>EC</td>
<td>1.01</td>
<td>1.03</td>
</tr>
<tr>
<td>CC</td>
<td>1.16</td>
<td>1.17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version</th>
<th>16</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>geomean</td>
<td>stddev</td>
</tr>
<tr>
<td>AC</td>
<td>0.99</td>
<td>1.03</td>
</tr>
<tr>
<td>EC</td>
<td>1.03</td>
<td>1.04</td>
</tr>
<tr>
<td>CC</td>
<td>1.13</td>
<td>1.13</td>
</tr>
</tbody>
</table>

Table 6.3: The table shows performance numbers for the adapted (AC), classic (CC), and engineered (EC) allocators for 32, 24, 16, and 8 registers. The geometric mean, geometric standard deviation (σ), and range for one standard deviation (×σ, /σ), are listed for each register setting.

6.3.2 Parameter frequency

Our results show that the fixed set of parameters used for the engineered version does not give the best performance. It is not too surprising that the adaptive com-
Figure 6.4: Performance of the priority-based allocator after tuning with the adaptive compiler.

The compiler was able to find combinations of parameters that improved performance, since the parameters control aspects of the allocator whose effectiveness depends on the opportunities available in the input code. We can examine the frequency of various parameters in the best combinations found by the adaptive compiler to understand how the effectiveness of a parameter varies with the input code.

Tables 6.5 and 6.6 lists the frequency with which various parameters were selected by the adaptive compiler. The observed frequencies produce both expected results and a few surprises. We see that the most frequently selected block size is one instruction, which confirms our intuition that a more precise interference graph allows for a better allocation. Also, the most frequently selected coloring heuristic is to choose the color
Table 6.4: Allocation quality for the 20 largest routines in the benchmark suite. The results have been normalized to Chaitin-Briggs. The table shows performance numbers for the adapted (AC), classic (CC), and engineered (EC) allocators for 32, 16, and 8 registers. The results for 24 registers are omitted to save space; they do not vary much from the results for 32 registers.

<table>
<thead>
<tr>
<th>Routine</th>
<th>32</th>
<th>16</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AC</td>
<td>CC</td>
<td>EC</td>
</tr>
<tr>
<td>rhs</td>
<td>1</td>
<td>1.27</td>
<td>1.01</td>
</tr>
<tr>
<td>jacld</td>
<td>0.91</td>
<td>1.37</td>
<td>0.97</td>
</tr>
<tr>
<td>jacu</td>
<td>0.95</td>
<td>1.36</td>
<td>1</td>
</tr>
<tr>
<td>erhs</td>
<td>1</td>
<td>1.14</td>
<td>1.01</td>
</tr>
<tr>
<td>pintgr</td>
<td>1</td>
<td>1.14</td>
<td>1</td>
</tr>
<tr>
<td>drepyi</td>
<td>0.98</td>
<td>1.08</td>
<td>0.98</td>
</tr>
<tr>
<td>paroi</td>
<td>0.99</td>
<td>1.06</td>
<td>1.04</td>
</tr>
<tr>
<td>pastem</td>
<td>1.01</td>
<td>1.02</td>
<td>1.03</td>
</tr>
<tr>
<td>bilan</td>
<td>1.01</td>
<td>1.22</td>
<td>1.02</td>
</tr>
<tr>
<td>heat</td>
<td>1.05</td>
<td>1.24</td>
<td>1.09</td>
</tr>
<tr>
<td>debico</td>
<td>1</td>
<td>1.38</td>
<td>1.03</td>
</tr>
<tr>
<td>repvid</td>
<td>1.03</td>
<td>1.05</td>
<td>1.08</td>
</tr>
<tr>
<td>iniset</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>fpppp</td>
<td>0.87</td>
<td>1.5</td>
<td>0.91</td>
</tr>
<tr>
<td>parmvr</td>
<td>0.99</td>
<td>1.39</td>
<td>1</td>
</tr>
<tr>
<td>getb</td>
<td>1</td>
<td>1.2</td>
<td>1</td>
</tr>
<tr>
<td>init</td>
<td>0.99</td>
<td>1.1</td>
<td>1</td>
</tr>
<tr>
<td>ffib</td>
<td>1</td>
<td>1.04</td>
<td>1</td>
</tr>
<tr>
<td>putb</td>
<td>1</td>
<td>1.2</td>
<td>1</td>
</tr>
<tr>
<td>energy</td>
<td>1</td>
<td>1.15</td>
<td>1</td>
</tr>
<tr>
<td>GEOMEAN</td>
<td>0.99</td>
<td>1.19</td>
<td>1.01</td>
</tr>
</tbody>
</table>

that maximizes the number of neighbors for which it is already forbidden. This heuristic was the same one we hand-picked for the engineered version. It is interesting to note that our hand selected block size and coloring heuristic were selected by the adaptive compiler less than 30% of the time. Even though they were the most frequently selected settings, the other values were chosen a non-trivial number of times.

The biggest surprise from the frequency numbers is the priority function. The
most frequently selected priority function ignores the size of the live range when computing priority. Our intuition was that the priority function described by Chow and Hennessy was quite good, but we found that an alternate priority function was selected more often by the adaptive compiler. This result can be interpreted in two ways: either it is more important to ensure that a live range with many uses and definitions is allocated a register regardless of its size, or that some combination of parameters enables the alternate priority function to work better than the original. These alternate explanations point to the difficulty in trying to statically understand the interaction of the allocation parameters and the need for an adaptive method to correctly set their values.

Table 6.6 shows the frequency for the Boolean parameters in the priority-based allocator. The table shows that rematerialization was rarely selected by the adaptive compiler – an indication that our approach to rematerialization was not often profitable. The process of rematerialization described in Section 4.3 has two potential disadvantages: it splits live ranges early in the allocation process, and it relies on another pass for clean up. Our results suggest that these disadvantages outweigh the benefits in most codes. It is worth noting that we weight all instructions equally in our comparison, and rematerialization may become more attractive if we weigh store and load instructions more than a load immediate. The other interesting result from this table is the frequency of live-range trimming, which occurs in only 50% of the top parameter strings. We thought that live-range trimming would have a large impact on performance, since it removes useless blocks and potentially allows other live ranges to be allocated in those blocks. It seems this benefit is countered by the reduction in opportunities for aggressive code motion to turn load/store pairs into copies. Aggressive code motion is selected a little more than 50% of the time, so it is reasonable that live-range trimming is selected a similar percentage, since disabling
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Frequency (per number of registers)</th>
<th>% of total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block size</td>
<td>32 24 16 8 total</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11 13 14 6 total</td>
<td>11%</td>
</tr>
<tr>
<td>1</td>
<td>31 28 26 16 total</td>
<td>24%</td>
</tr>
<tr>
<td>2</td>
<td>16 15 18 13 total</td>
<td>15%</td>
</tr>
<tr>
<td>3</td>
<td>8 7 8 8 total</td>
<td>7%</td>
</tr>
<tr>
<td>4</td>
<td>9 7 8 7 total</td>
<td>7%</td>
</tr>
<tr>
<td>5</td>
<td>10 8 8 4 total</td>
<td>7%</td>
</tr>
<tr>
<td>6</td>
<td>8 6 1 5 total</td>
<td>5%</td>
</tr>
<tr>
<td>7</td>
<td>4 6 2 4 total</td>
<td>4%</td>
</tr>
<tr>
<td>8</td>
<td>4 2 4 3 total</td>
<td>3%</td>
</tr>
<tr>
<td>9</td>
<td>8 14 10 6 total</td>
<td>9%</td>
</tr>
<tr>
<td>15</td>
<td>9 9 8 2 total</td>
<td>7%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>color strategy</th>
<th>32 24 16 8 total</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>22 23 26 19 total</td>
<td>22%</td>
</tr>
<tr>
<td>2</td>
<td>25 30 20 21 total</td>
<td>23%</td>
</tr>
<tr>
<td>3</td>
<td>32 40 32 17 total</td>
<td>29%</td>
</tr>
<tr>
<td>4</td>
<td>39 23 28 17 total</td>
<td>26%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>split strategy</th>
<th>32 24 16 8 total</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>61 70 44 32 total</td>
<td>50%</td>
</tr>
<tr>
<td>2</td>
<td>57 46 62 42 total</td>
<td>50%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>priority function</th>
<th>32 24 16 8 total</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21 26 31 23 total</td>
<td>24%</td>
</tr>
<tr>
<td>2</td>
<td>36 44 34 27 total</td>
<td>34%</td>
</tr>
<tr>
<td>3</td>
<td>36 34 27 11 total</td>
<td>18%</td>
</tr>
<tr>
<td>4</td>
<td>18 18 14 8 total</td>
<td>14%</td>
</tr>
<tr>
<td>5</td>
<td>17 9 9 5 total</td>
<td>1%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>loop weight</th>
<th>32 24 16 8 total</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>37 22 29 20 total</td>
<td>26%</td>
</tr>
<tr>
<td>5</td>
<td>25 30 24 13 total</td>
<td>22%</td>
</tr>
<tr>
<td>10</td>
<td>22 30 30 18 total</td>
<td>24%</td>
</tr>
<tr>
<td>20</td>
<td>34 34 23 23 total</td>
<td>28%</td>
</tr>
</tbody>
</table>

Table 6.5: Frequency of multi-valued parameter settings occurring in the top combination found by the adaptive compiler. The most frequent value for each parameter and the best value for each register is displayed in bold face. The strategy and priority function settings correspond to the descriptions given in Section 6.2
Figure 6.5: Frequency of the number of registers used for local allocation in the top combination found by the adaptive compiler. Results are displayed separately for integer and float registers. Each graph displays results for 32, 24, 16, and 8 registers.
live-range trimming can create more opportunities for aggressive code motion. These results reinforce our original notion that the correct set of allocation parameters is highly dependent on the input code and difficult to specify statically.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Frequency (per number of registers)</th>
<th>% of total</th>
</tr>
</thead>
<tbody>
<tr>
<td>aggressive code motion</td>
<td>total</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>218</td>
<td>53%</td>
</tr>
<tr>
<td>false</td>
<td>196</td>
<td>47%</td>
</tr>
<tr>
<td>rematerialization</td>
<td>total</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>122</td>
<td>29%</td>
</tr>
<tr>
<td>false</td>
<td>292</td>
<td>71%</td>
</tr>
<tr>
<td>live-range trimming</td>
<td>total</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>206</td>
<td>50%</td>
</tr>
<tr>
<td>false</td>
<td>208</td>
<td>50%</td>
</tr>
<tr>
<td>globally allocate locals</td>
<td>total</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>138</td>
<td>33%</td>
</tr>
<tr>
<td>false</td>
<td>276</td>
<td>67%</td>
</tr>
<tr>
<td>optimistic simplification</td>
<td>total</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>207</td>
<td>50%</td>
</tr>
<tr>
<td>false</td>
<td>207</td>
<td>50%</td>
</tr>
<tr>
<td>allocate all unconstrained</td>
<td>total</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>160</td>
<td>39%</td>
</tr>
<tr>
<td>false</td>
<td>254</td>
<td>61%</td>
</tr>
<tr>
<td>enhanced promotion</td>
<td>total</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>229</td>
<td>55%</td>
</tr>
<tr>
<td>false</td>
<td>185</td>
<td>45%</td>
</tr>
<tr>
<td>spill clean</td>
<td>total</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>195</td>
<td>47%</td>
</tr>
<tr>
<td>false</td>
<td>219</td>
<td>53%</td>
</tr>
</tbody>
</table>

Table 6.6: Frequency of Boolean-valued parameter settings occurring in the top combination found by the adaptive compiler. The most frequent value for each parameter and the best value for each register is displayed in bold face. The parameters correspond to the descriptions given in Section 6.2.
6.3.3 Search efficiency

Previous work on adaptive compilation has focused on the efficiency of the search, which is how quickly the compiler finds a good solution [34, 53, 48]. This thesis uses adaptive compilation more as a tool to obtain a good allocation than a subject of study. We did not focus on the efficiency of search, except that it should find good answers in a reasonable amount of time. We chose the adaptive compiler’s iteration limit of 1000 to weight the search more towards finding good solutions than finding solutions quickly. Our experiments verified that 1000 iterations were enough to produce good solutions.

To address the efficiency issue, we discuss two example searches that are representative of the searches run by the adaptive compiler. Figure 6.6 shows the progress of the search during the compilation of the rkf45 benchmark. Each file has its own individual progress indicated by a unique line in the graph. We see that within 200 iterations, each file is within 2% of the best result found for that file. The search for the rkfs file is a bit of a straggler in this example. It does not approach the best result until nearly 1000 iterations, at which point it is only slightly worse than the best result found for that file across all of our experiments.

Figure 6.7 shows the search progress for the applu benchmark. The graph shows that all files are within 1% of the best result after 200 iterations. The files that have not yet reached the best value continue to approach it as the iterations go towards 1000. The long trail out to 1000 is not shown in the graph to give a better view of the first 200 iterations. This graph demonstrates the benefit of our search algorithm, where we can run a search for each file in parallel, but incur the same cost for compiling and running the program as if we only were searching for a single set of parameters across the entire benchmark. This graphic provides a nice visualization of the file searches progressing in parallel and shows that the adaptive compiler tends to find
good values early in the search.

Figure 6.6: Search descent for the rkf45 benchmark. The graph shows the progress of the search for each file in the benchmark.

Figure 6.7: Search descent for the applu benchmark. The graph shows the progress of the search for each file in the benchmark.
6.4 Allocation time

We measured the compile time of several versions of the priority-based allocator and compared them to the compile time of the Chaitin-Briggs allocator. To be fair, we must note that our Chaitin-Briggs allocator benefited from years of tuning to improve compilation speed. The priority-based allocator was written to produce the best possible code with little focus on allocation speed. The implementations have not had the same level of tuning, but it is still interesting to compare the allocation times and look for general trends.

Allocation times were collected for each function by compiling the function five times and taking the arithmetic mean of the allocation times. Table 6.7 displays the median and mean compilation times across all functions. We collected results for the Chaitin-Briggs allocator and two versions of the classic allocator — one with unlimited basic-block size (CC-0) and one with blocks limited to one instruction (CC-1) — along with the engineered (EC) and adapted versions (AC). Times for the adapted version were collected by allocating each function using the best parameter string found for that function.

The results show that the Chaitin-Briggs and CC-0 allocators are about equivalent, on average, in compilation times. The CC-1, EC, and AC versions of the priority-based allocator have a mean compilation time that is generally 1-2 orders of magnitude slower than the Chaitin-Briggs compiler. The discrepancy is caused by lengthy compile times for large functions, and can be seen by the high standard deviations for those allocators and the relatively close median compilation times. The CC-0 allocator does not suffer from these high compilation times because it removes all local live ranges from consideration and the global allocation phase ends quickly because of the coarse interference graph.
Figure 6.8 shows the allocation time plotted against function size. The graphs show the Chaitin-Briggs allocator is nearly always faster, even on small functions. Briggs compared the allocation times of Chow’s implementation of the classic priority-based allocator from the MIPS compiler and Briggs’s implementation of the Chaitin-Briggs allocator in his thesis [9]. He found that the priority-based allocator was faster on smaller functions, but the Chaitin-Briggs allocator was faster on larger functions. However, he compared allocators from two different compilers, so it was not a completely fair comparison. Our results do not have a clear case where the priority-based allocator is faster than Chaitin-Briggs.

<table>
<thead>
<tr>
<th></th>
<th>32</th>
<th></th>
<th></th>
<th>24</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>median</td>
<td>mean</td>
<td>stddev</td>
<td>median</td>
<td>mean</td>
<td>stddev</td>
</tr>
<tr>
<td>CB</td>
<td>0.0214</td>
<td>0.0950</td>
<td>0.2590</td>
<td>0.0237</td>
<td>0.0945</td>
<td>0.2905</td>
</tr>
<tr>
<td>CC-0</td>
<td>0.0403</td>
<td>0.0958</td>
<td>0.2018</td>
<td>0.0346</td>
<td>0.0850</td>
<td>0.1921</td>
</tr>
<tr>
<td>CC-1</td>
<td>0.0581</td>
<td>1.2801</td>
<td>6.0309</td>
<td>0.0538</td>
<td>1.1714</td>
<td>6.2158</td>
</tr>
<tr>
<td>EC</td>
<td>0.0952</td>
<td>1.9211</td>
<td>8.3174</td>
<td>0.0761</td>
<td>1.5865</td>
<td>7.5520</td>
</tr>
<tr>
<td>AC</td>
<td>0.0520</td>
<td>0.7212</td>
<td>4.4469</td>
<td>0.0432</td>
<td>1.1846</td>
<td>7.4092</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>16</th>
<th></th>
<th></th>
<th>8</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>median</td>
<td>mean</td>
<td>stddev</td>
<td>median</td>
<td>mean</td>
<td>stddev</td>
</tr>
<tr>
<td>CB</td>
<td>0.0250</td>
<td>0.1013</td>
<td>0.3083</td>
<td>0.0225</td>
<td>0.0812</td>
<td>0.2661</td>
</tr>
<tr>
<td>CC-0</td>
<td>0.0361</td>
<td>0.0950</td>
<td>0.2342</td>
<td>0.0314</td>
<td>0.0752</td>
<td>0.1911</td>
</tr>
<tr>
<td>CC-1</td>
<td>0.0531</td>
<td>1.3834</td>
<td>7.1593</td>
<td>0.0542</td>
<td>0.8208</td>
<td>5.2609</td>
</tr>
<tr>
<td>EC</td>
<td>0.0660</td>
<td>1.7414</td>
<td>8.0642</td>
<td>0.0576</td>
<td>1.2876</td>
<td>6.8937</td>
</tr>
<tr>
<td>AC</td>
<td>0.0415</td>
<td>0.7126</td>
<td>5.1378</td>
<td>0.0394</td>
<td>0.2289</td>
<td>0.8390</td>
</tr>
</tbody>
</table>

Table 6.7: Allocation time in seconds for various numbers of registers. The versions displayed are CB (Chaitin-Briggs), CC-0 (classic with no block limit), CC-1 (classic with blocks limited to one instruction), EC (engineered), and AC (adaptive).

6.5 Summary

In this section we presented our experimental results comparing the three versions of the priority-based allocator - classic, engineered, and adapted - with the Chaitin-
Briggs allocator. We showed that the classic allocator was not competitive with Chaitin-Briggs, giving results that were 16-9% worse depending on the number of registers used. We then showed that the engineered version can improve the allocation to within 1% of Chaitin-Briggs for 32 and 24 registers. We needed the full power of the adaptive compiler to tune the priority-based allocator to work with fewer registers. Using our adaptive compiler, we were able to consistently outperform the Chaitin-Briggs allocator, though only by a slim margin of 1-2%. We concluded by giving experimental results for frequency of parameter selection by the adaptive compiler, a brief discussion of the effectiveness of our search algorithm, and a comparison of
compilation times.
Chapter 7

Conclusion

In this thesis we provided a thorough study of the priority-based register-allocation algorithm. We gave a detailed description of the implementation and described some engineering enhancements. The allocator’s behavior was parametrized and added to an adaptive compiler. The adaptive compiler was used to search for parameter settings that give good allocations. The three versions of the priority-based allocator – classic, engineered, and adapted – were compared to a Chaitin-Briggs allocator. Our comparison revealed that, on average, the classic version is 16-9% worse than the Chaitin-Briggs allocator, the engineered version is nearly equal with Chaitin-Briggs except when using 8 registers, and the adapted version is about 1% better than Chaitin-Briggs.

This thesis provides five main contributions. First, we give a detailed description of the implementation of the priority-based allocator. A detailed description of the algorithm has not been previously published. Second, we describe engineering enhancements to the standard priority-based allocator. Third, we show how to parametrize the allocator so that it can be used in an adaptive compiler. Fourth, we give an adaptive compilation algorithm that simultaneously searches for different parameter settings over a set of files but incurs the same cost as searching for a single setting for the entire benchmark. Finally, we give the first fair comparison between the Chaitin-Briggs and priority-based algorithms. This comparison fills a hole in the
literature that has existed since the initial publication of the priority-based algorithm.

We should not conclude from these results that the priority-based algorithm is obsolete. Ultimately, the decision of which algorithm to implement is left to the compiler writer. Chow and Hennessy formulated their algorithm in the context of machines that included memory-to-memory operations. As architectures evolve, the priority-based approach may more naturally fit with some future architecture.

This thesis shows the benefits of adaptive compilation for priority-based allocation. We used adaptive compilation to make a fair comparison with the Chaitin-Briggs algorithm. Our results provide one piece of data on the relative performance characteristics of the two algorithms in the context of a RISC machine. We should not generalize to say that Chaitin-Briggs is better than the priority-based approach in all instances. Future work can examine how different architectures effect the relative performance of the algorithms. This thesis serves as a good starting point for anyone attempting to implement and improve on the priority-based register-allocation algorithm.
Bibliography


