# Chow and Hennessy vs. Chaitin-Briggs Register Allocation: Using Adaptive Compilation to Fairly Compare Algorithms

Keith D. Cooper[1], Timothy J. Harvey[1], and David M. Peixotto[1]

Rice University {keith,harv,dmp}@rice.edu

**Abstract.** When we set out to compare the performance of two algorithms against each other, the quality of the comparison is wholly dependent on the quality of the two implementations. If an algorithm is particularly difficult to implement correctly, if it is poorly understood or documented, or if its performance is highly variable based on any number of idiosyncratic settings, a comparison against another algorithm should normally be suspect.

In our case, we wanted to compare a Chaitin-Briggs graph-coloring register allocator against a Chow and Hennessy priority-based allocator, but it seemed impossible to get a fair comparison. The Chaitin-Briggs allocator is relatively well understood. It has been studied extensively. It has few parameters to guide its behavior.

In contrast, Chow's allocator, although often cited as a viable competitor to the graph-coloring allocator, is none of these. It has been implemented very few times. Its presence in the literature is more historic than analytic. Worst of all, it not only includes a number of implementation decisions that individually affect the quality of code, but these decisions interact unpredictably.

Adaptivity suggests a solution to this problem. By allowing a search technique to optimize an algorithm's performance, we can make definitive statements comparing two algorithms.

In this paper, we present an implementation of a priority-based register allocator as described by Chow and Hennessy. We show how to implement the allocator in an adaptive framework, and we show that the code produced consistently achieves the performance of a Chaitin-Briggs graph-coloring register allocator.

The Chaitin-Briggs algorithm has received so much attention in the literature that we felt it would be fair to use a non-adaptive implementation as a baseline to compare against.

Our results show that the two algorithms give similar performance, but only when we use an adaptive search algorithm for the Chow allocator. We believe that this is the first time these two algorithms have been fairly tested side-by-side, and we argue that adaptivity is an effective tool for experimental computer science.

## 1 Introduction

From the first compiler by Backus *et al.* [1] to compilers for modern-day multi-core microarchitectures, there has been one overarching engineering goal: use the compiler to efficiently manage the hardware resources to maximize performance. Regardless of the architecture, managing the register set has been one of the primary concerns of resource management. Even early architectures that had zero-wait-state memory and memory-to-memory operations used registers for calculations such as array indexing, and these were critical for overall performance. Later, memory-to-memory operations were mostly phased out – the only way to perform a calculation on a value was to first put that value in a register.

Into this architectural phase shift came Chaitin and his colleagues at IBM who developed the first global register-allocation algorithm for RISC machines. They abstracted the problem to that of graph coloring, where two nodes in a graph cannot be colored the same if there is an edge between them. For program values, the analogy to graph coloring is that if two values need to be *alive* at the same time they cannot share the same register. Values are alive from the point of their initial definition to their last use. We call each distinct value a *live range*. Chaitin's algorithm walks the code, building a graph where the nodes are values and edges connect nodes if the values *interfere* – because they are alive at the same time. With the graph in hand, the coloring algorithm uses a simple heuristic to order the nodes for which to choose colors. When a node cannot get a color – that is, a machine register – because the colors being used by the node's neighbors comprise the complete set of possible colors, that value is *spilled*, meaning it is stored to memory and only loaded into a register for short periods of time around each definition and use. Of course, this still requires a register, so some shuffling has to occur, and Chaitin's algorithm iterates until no more spill code is added, at which time it has a legal coloring of the graph, and, thus, an assignment of machine registers to program values.

Chaitin's algorithm is probably the best known, most thoroughly studied, and most widely used algorithm for register allocation – in truth, any new algorithm introduced today would probably have to compare itself against an implementation of Chaitin's algorithm before the compiler community will even consider it. That said, the algorithm is not without weaknesses: it uses a "spill everywhere" strategy for live ranges and is pessimistic in its attempts to color the interference graph. Many people have published improvements to the algorithm, including Briggs *et al.* [2], Callahan and Koblenz [3], Bergner [4], and Cooper and Simpson [5]. We used the Chaitin-Briggs version of the algorithm in our comparison.

In contrast, the priority-based allocator of Chow and Hennessy has received relatively little attention. Although the paper on the implementation is heavily cited, it turns out that those citations merely mention that it is a competing algorithm; with the exception of Larus and Hilfinger [6], we were unable to find any description of an implementation, and certainly no comparison with any other register-allocation algorithm. The source code for the original allocator was freely available, and people could use it for a reference if they wanted to implement the allocator. We wanted to compare the two algorithms, and thought that Chow and Hennessy's priority-based allocator seemed a good candidate for adaptation as its implementation is underspecified and its behavior can be controlled through a variety of parameters.

Chow's algorithm is similar to Chaitin's graph-coloring allocation algorithm, but it varies in four significant ways:

1. Chow's machine model includes memory-to-memory operations; as a result, Chow's model is somewhat pessimistic: Chaitin optimistically assumes that everything resides in a register until he proves that can't be true, while Chow assumes that everything starts out in memory, and *promotes* values into registers.
2. Chow's live ranges are imprecise; they are defined in terms of basic blocks, rather than instructions. While Chaitin recognizes an interference between two values that are both live at an instruction, two values interfere in Chow's algorithm if they are both live in the same block – if a value has its last use before the definition of a different value in the same block, Chow will record that as an interference.
3. Chow's algorithm uses a different spill-cost metric than Chaitin' algorithm; instead of computing the cost of spilling a value, Chow computes the *priority*, or benefit, of promoting a value from memory into a register. He assigns colors to live ranges in descending order of priority, while Chaitin attempts to assign colors to as many live ranges as possible.
4. Chow's algorithm performs spilling by first trying to split a live range into smaller pieces, some or all of which may get a register (although it was unable to find a single register for the entire live range, it may be able to keep different parts of the live range in different registers). Chaitin uses a "spill everywhere" strategy that spills the entire live range when it can not be assigned a color.

The details of the implementation include a number of interacting settings, such as adjusting the length of the basic block (which changes the precision of the live ranges – an innovation from Larus and Hilfinger [6]), adjusting the number of registers reserved for local allocation, adjusting the coloring heuristic, *etc.* These adjustments are historically the reason that a comparison between the two algorithms has never before been done: popular wisdom has always been that only an expert could find the best set of parameters for Chow's algorithm, making a fair comparison impossible. Further, the dominance of the RISC machine model has meant that Chow's algorithm needs adjusting before it can be compared to Chaitin's algorithm. Finally, twenty-five years of research aimed at improving data-flow analysis and optimization – with static single assignment (SSA) form being the most significant development – has been integrated into Chaitin's algorithm but not into Chow's.

In this paper, we show how to parametrize Chow's algorithm to provide a large enough search space to allow the adapter to find good settings and produce efficient code. We then use the adapted version to compare against the Chaitin algorithm. We implemented several versions of the priority-based allocator to perform our experiments: a classic version that follows the description by Chow and Hennessy, an engineered version that adds some basic improvements and uses a fixed set of parameters designed by our local "expert", and an adapted version that uses adaptive compilation to tune the algorithm's parameters. The result of comparing Chow's algorithm to Chaitin's were surprising. Although the community has long considered the two algorithms to be similar in terms of the quality of code they produced, we were unable to consistently get as good results from the priority-based allocator as from Chaitin's allocator. As we will show, it was only by adapting Chow's allocator for each routine individually that we could rival the quality of code from Chaitin's allocator.

In the next section, we will show the details of setting up the adaptive search for Chow's original algorithm.

## 2 Adaptive Compilation Framework

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 [7]. We follow his lead and use a random-restart impatient-hillclimber as the search mechanism in our adaptive compiler.

Our work varies from Grosul's, because we found it necessary to change the level of granularity from the program level to the procedure level. On Grosul's problem, a single parameter setting sufficed for all of the functions in a program, but when we implemented our search at that level of granularity, we found that most of the results showed very little improvement over not using the adapter.

Kulkarni *et al.* report a similar result using their set of optimizations: they have to allow the adapter to find different parameter settings for each function individually [8]. Our method has a slight improvement over Kulkarni's VISTA system: whereas their system isolates a single routine and finds the best compilation sequence before moving on to the next routine in the program, our system searches on each of a program's functions concurrently and in lockstep. At each iteration, we derive new parameter settings for each function and record separate performance data for each function upon execution. We can do this because our objective function uses the number of instructions executed by the program, rather than the time the program takes to execute. If we find a local minima for one of the functions, that function can start a new descent, while other functions in the program may still be progressing downwards through the space. We stop the search at a predetermined number of compilations, rather than, as Grosul does, a set number of restarts.

We exposed a set of parameters in the allocator that can be configured with command-line arguments. When a file is compiled, the allocator is passed command-line flags corresponding to the combination of parameters currently under examination. A set of specific parameter settings is defined as a *point* in the search space. The neighborhood structure describes how the points in the space are connected, and this further dictates the order that parameter combinations are tried, since the search can only move to a point that is in its neighborhood.

We define the neighbor of a point to be all points that can be reached by changing a single setting in a single parameter. Neighbors can be generated by first selecting a parameter and then choosing a new setting. If we imagine that 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. The number of neighbors and the total size of the search space varies with the number of registers used for allocation, because the domain of one of the parameters grows as we increase the size of the register set. With 32 registers, each point has 434 neighbors and the search space has 182,927,360 elements. With 8 registers, this decreases to 38 neighbors and 4,505,600 elements.

## 3 Finding Parameters for Adaptation

We wanted to 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 criteria 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, but the exact outcome is hard to predict. Aggressive code motion of loads and stores is an example of this type of parameter; it may hurt or help depending on the number of opportunities to turn load/store pairs into copies. The third case of parameters are 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 not be set in isolation from each other. If we set the number of local registers very high, it would not make sense to also allow local live ranges to be put in registers reserved for global allocation.

To give maximal freedom to the adaptive compiler, we included as many parameters as possible, subject to the three criteria stated above. The precise parameters chosen are discussed below.

**basic-block size** The size of a basic block changes the number of interferences. 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 to use for allocating local live ranges and spilled global live ranges. 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 turn a store in a predecessor block followed by a load in the current block into a copy. Moving all loads and stores 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. The splitting of live ranges may have a negative effect if they should not have been split and cannot be joined back together.

**live-range trimming** Live-range trimming removes useless blocks from a live range after it is split. Trimming 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 [9].

**splitting heuristic** We use two different heuristics when deciding which blocks to include in the new live range when splitting.

1. Split as originally described by Chow and Hennessy.
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 uniformly with the global priority-based allocator.

**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 which 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 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 that 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 by Chow and Hennessy.

1. Use the standard priority function.
2. Use the standard priority, 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 [10].
4. Square the size of the live range before normalizing. This makes larger live-ranges less desirable.
5. Combination of 3 and 4.

## 4 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 for an abstract RISC machine. The input program initially contains an arbitrary number of pseudo-registers, which 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. Popular folklore has long held that a low number of registers – say, 8 – favored Chow's algorithm and that a larger number of registers – on the order of 32 – would give an advantage to Chaitin's algorithm. Thus, our experiments were run with 8, 16, 24, and 32 machine registers with separate register files for integer and floating point values.

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 and instrumented with performance counters. 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. This is important because we use the performance measure to guide the adaptive compiler's search.

### 4.1 Setup

We use a variety for 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 [11–13]. These programs include a total of 121 distinct routines. Due to a limitation of our intermediate representation, not all files can be compiled for each register setting. The files that cannot be compiled with fewer registers are excluded in the results reported for that specific number of registers.

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 set of optimizations. After allocation, the code is passed through two optimizations for clean up: dead code elimination and empty basic-block removal. These passes are needed because several of the techniques used in the engineered version of the allocator expect a post-pass that removes useless code.

### 4.2 Results

We added the priority-based allocator to our adaptive compilation framework as described in Sect.2. The results in this section were gathered using two runs of the adaptive compiler for each benchmark. Based on previous experience, we use a patience of 20% for the hillclimber and limit each run to 1000 evaluations [7, 14] . Our success in finding good parameter combinations indicates these values also work well when searching for priority-based allocation parameters.

We compare the 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. The main results are shown in Fig.1. This figure compares the results of the adapted version of the allocator with the classic and engineered versions. For the classic and engineered versions of the priority-based algorithm, 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 result contrasts with using a fixed set of parameters in the engineered version, where performance approaches the Chaitin-Briggs allocator when using 32 or 24 registers but deteriorates with fewer registers.

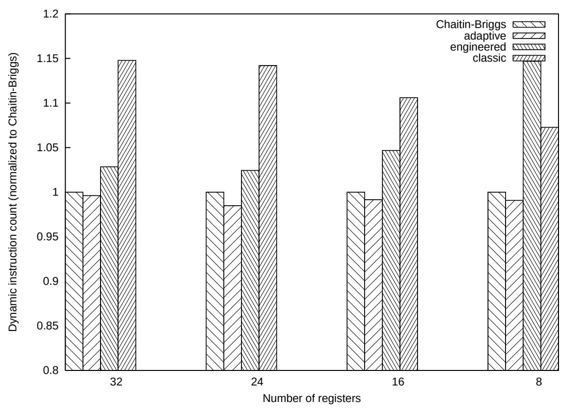

**Fig. 1.** Performance of the priority-based allocator after tuning with the adaptive compiler.

## 4.3 The Impact of Adaptivity on This Experiment

What evidence do we have that adaptivity was an efficacious way to approach this experiment?

As shown in Fig.1, the engineered version of the priority-based allocator does progressively poorer as the number of registers decreases. This degradation is largely due to the engineered version including local live ranges in the global allocation, rather than leaving them for a separate local allocation pass. It has more of a negative effect with fewer registers, because the allocator tends to spend its few global registers to allocate local values, spilling possibly more important global live ranges. The local live ranges get assigned a higher priority because they are only live in one block. The parameter settings were designed with 32 registers in mind, and they perform well in this case. Adapting the code gives us a dynamic algorithm against which to compare and helps keep small errors from muddying our results.

Another question to answer when considering an adaptive compilation scheme is: does the adapter find a single, better set of parameters for the compiler, or does it find idiosyncratic sets of parameters for likewise idiosyncratic input codes? If the former, we need only run the adapter offline on a representative sample of inputs. On the other hand, if we see that the adaptive compiler finds significantly different sets of parameters for each input we give it, then this argues strongly that an adaptive scheme is required to get the best results from the algorithm.

As we show in Table 1, no single parameter setting dominated. The table 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 more precise live-ranges allow for a better allocation. Also, the most frequently selected coloring heuristic is to choose the color that maximizes the number of neighbors for which it is already forbidden. This was our hand-picked heuristic 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 significant 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 it size, or that some combination of parameters enables the alternate priority function to work better. 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 1.** Frequency of various parameter settings (column S) 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 Sect.2.

| S | freq (per num regs) | | | | |
|---|---|---|---|---|---|
| | 32 | 24 | 16 | 8 | Tot(%) |
| *block size* | | | | | |
| 0 | 11 | 13 | 14 | 6 | 44(11%) |
| **1** | **31** | **28** | **26** | **16** | 101(**24%**) |
| 2 | 16 | 15 | 18 | 13 | 62(15%) |
| 3 | 8 | 8 | 7 | 8 | 31(7%) |
| 4 | 9 | 7 | 8 | 7 | 31(7%) |
| 5 | 10 | 8 | 8 | 4 | 30(7%) |
| 6 | 8 | 6 | 1 | 5 | 20(5%) |
| 7 | 4 | 6 | 2 | 4 | 16(4%) |
| 8 | 4 | 2 | 4 | 3 | 13(3%) |
| 9 | 8 | 14 | 10 | 6 | 38(9%) |
| 15 | 9 | 9 | 8 | 2 | 28(7%) |
| *allocate all unconstrained* | | | | | |
| T | 47 | **61** | 28 | 24 | 160(39%) |
| **F** | **71** | 55 | **78** | **50** | 254(**61%**) |

| S | freq (per num regs) | | | | |
|---|---|---|---|---|---|
| | 32 | 24 | 16 | 8 | Tot(%) |
| *color strategy* | | | | | |
| 1 | 22 | 23 | 26 | 19 | 90(22%) |
| 2 | 25 | 30 | 20 | **21** | 96(23%) |
| **3** | 32 | **40** | **32** | 17 | 121(**29%**) |
| 4 | **39** | 23 | 28 | 17 | 107(26%) |
| *split strategy* | | | | | |
| 1 | **61** | **70** | 44 | 32 | 207(50%) |
| 2 | 57 | 46 | **62** | **42** | 207(50%) |
| *live-range trimming* | | | | | |
| T | 52 | **59** | **58** | 37 | 206(50%) |
| F | **66** | 57 | 48 | 37 | 208(50%) |
| *enhanced promotion* | | | | | |
| **T** | **61** | **67** | **61** | **40** | 229(**55%**) |
| F | 57 | 49 | 45 | 34 | 185(45%) |

| S | freq (per num regs) | | | | |
|---|---|---|---|---|---|
| | 32 | 24 | 16 | 8 | Tot(%) |
| *priority function* | | | | | |
| 1 | 21 | 26 | 31 | 23 | 101(24%) |
| **2** | **36** | **44** | **34** | **27** | 141(**34%**) |
| 3 | 26 | 19 | 18 | 11 | 74(18%) |
| 4 | 18 | 18 | 14 | 8 | 58(14%) |
| 5 | 17 | 9 | 9 | 5 | 40(10%) |
| *rematerialization* | | | | | |
| T | 25 | 27 | 30 | 40 | 122(29%) |
| **F** | **93** | **89** | **76** | **34** | 292(**71%**) |
| *globally allocate locals* | | | | | |
| T | 51 | 37 | 28 | 22 | 138(33%) |
| **F** | **67** | **79** | **78** | **52** | 267(**67%**) |
| *aggressive code motion* | | | | | |
| **T** | **66** | 58 | 52 | **42** | 218(**53%**) |
| F | 52 | 58 | **54** | 32 | 196(47%) |

### 4.4 Search Efficiency

Previous work on adaptive compilation has focused on the efficiency of the search, the measure of how quickly the adaptive compiler finds a good solution [7, 8]. Although our work focuses on using adaptation in experimentation, our new search technique raises the question of efficiency because we change the level of granularity of the search from the whole program to individual files.

Figure 2 shows the search progress for the `applu` benchmark. Each line indicates the individual progress of the adaptive search for a particular function in the benchmark. The graph shows that all files are within 1% of the best result after 200 iterations. The files which have not yet reached the best value continue to approach it as the iterations go towards 1000. This 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. Compiling this data seems a necessary step in evaluating the effectiveness of using adaptation in experimental algorithm analysis.

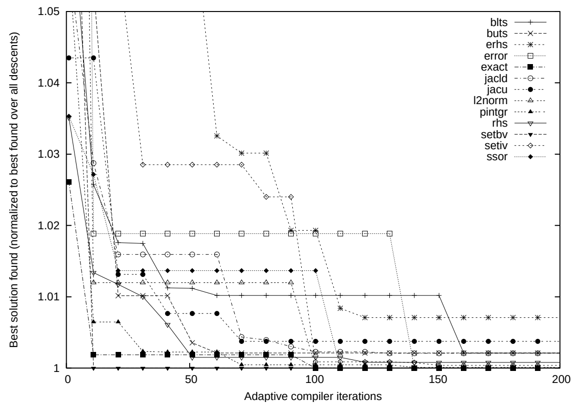

**Fig. 2.** Descent of the hillclimber during an adaptive compiler search of the `applu` benchmark.

## 5 Related Work

This work 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. This paper is one of few works that uses prior research in both areas to tackle the problem of register allocation.

### 5.1 Register Allocation

Register allocation has been an important optimization since the earliest compilers. Many techniques have been proposed for register allocation, but it was Chaitin *et al.* who described the first graph-coloring register allocator for modern RISC architectures [15]. They mapped the register allocation problem to finding a $k$ coloring of a graph, where $k$ is then number of machine registers. This abstraction has proven itself to be powerful, and much work has been done to improve the initial design.

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 work [16]. Larus and Hilfinger describe their implementation of the priority-based allocator in the SPUR Lisp compiler [6]. They were the first to propose limiting the basic-block size to obtain a more precise interference graph. They 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 [17]. Sorkin makes some suggestions for changing the priority-based allocator [18], but implementing his changes would fundamentally change the allocator. Kim *et al.* show how to adapt the priority-based allocator for architectures with predicated execution [19]. 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 [20], and rematerialization [9]. 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 [4]. Cooper and Simpson give a method for reducing spill code using live-range splitting [5]. Callahan and Koblenz describe an enhancement to the Chaitin allocator that takes the structure of the program into account when making spilling decisions [3].

### 5.2 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 survey two uses of adaptive compilation: tuning a specific compiler optimization, and finding a good order for compiler optimizations (the phase-ordering problem).

Stephenson *et al.* use machine learning techniques to fine tune various compiler heuristics [21]. 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.

Cavazos, Moss, and O'Boyle apply machine learning to register allocation in the Jikes RVM [22]. 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.

Waterman [23] and Cavazos and O'Boyle [24] independently experimented with using adaptive compilation to improve procedure inlining. Both researchers report a reduction in execution time due to improved inlining decisions.

Other researchers use adaptive compilation to find a good order for compiler optimization passes. Kulkarni *et al.* describe VISTA, an interactive compilation system that uses search to find good sequences of compiler optimizations [25, 8]. 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 later extended their work to exhaustively explore the search space of compiler optimization passes [26, 27].

Cooper *et al.* describe their experience exploring the search space of compilation sequences [28, 29]. 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.

Agakov *et al.* use machine learning techniques to focus the search of an adaptive compiler [30]. 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.

## 6 Conclusion

In this paper, we used adaptivity to control the settings of Chow and Hennessy's priority-based register allocator so that we could compare that algorithm's performance against a Chaitin-Briggs register allocator. We believe that the comparison could not have been done without adaptivity because Chow's algorithm has many settings that significantly impact its performance, especially as the underlying architectural model changes. Our results showed significant differences between a naive implementation, an engineered version, and the adapted version. The quality of the code produced was heavily dependent on changing parameter settings for each individual function and for each architectural specification. This makes a strong argument that similarly adjustable algorithms should follow the model we present.

In the question of Chow and Hennessy vs. Chaitin-Briggs, the analysis strongly suggests that the priority-based allocator does not produce code as good as the Chaitin-Briggs allocator.

## References

1. Backus, J.W., Beeber, R.J., Best, S., Goldberg, R., Haibt, L.M., Herrick, H.L., Nelson, R., Sayre, D., Sheridan, P.B., Stern, H., Ziller, I., Hughes, R.A., Nutt, R.: The fortran automatic coding system. In: Western Joint Computer Conference. Volume 11. (1957)
2. Briggs, P., Cooper, K.D., Torczon, L.: Improvements to graph coloring register allocation. ACM Transactions on Programming Languages and Systems **16**(3) (1994) 428–455
3. Callahan, D., Koblenz, B.: Register allocation via hierarchical graph coloring. In: PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation, New York, NY, USA, ACM Press (1991) 192–203
4. Bergner, P., Dahl, P., Engebretsen, D., O'Keefe, M.T.: Spill code minimization via interference region spilling. In: SIGPLAN Conference on Programming Language Design and Implementation. (1997) 287–295
5. Cooper, K.D., Simpson, L.T.: Live range splitting in a graph coloring register allocator. In: CC '98: Proceedings of the 7th International Conference on Compiler Construction, London, UK, Springer-Verlag (1998) 174–187
6. Larus, J.R., Hilfinger, P.N.: Register allocation in the spur lisp compiler. In: SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler construction, New York, NY, USA, ACM Press (1986) 255–263
7. Grosul, A.: Adaptive Ordering of Code Transformations in an Optimizing Compiler. PhD thesis, Rice University (2005)
8. Kulkarni, P., Hines, S., Hiser, J., Whalley, D., Davidson, J., Jones, D.: Fast searches for effective optimization phase sequences. In: PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation, New York, NY, USA, ACM (2004) 171–182
9. Briggs, P., Cooper, K.D., Torczon, L.: Rematerialization. In: PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation, New York, NY, USA, ACM Press (1992) 311–321
10. Makarov, V.N.: Fighting register pressure in gcc. In: GCC Developers' Summit. (2004)
11. SPEC-CFP92. Standard Performance Evaluation Corporation. (1992)
12. SPEC-CFP95. Standard Performance Evaluation Corporation (1995)
13. Forsythe, G.E., Malcolm, M.A., Moler, C.B.: Computer Methods for Mathematical Computations. Prentice-Hall, Englewood Cliffs, New Jersey (1977)
14. Sandoval, J.A.: Tuning an adaptive-compilation search space with loop unrolling. Master's thesis, Rice University (2007)
15. Chaitin, G.J.: Register allocation & spilling via graph coloring. In: SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction, New York, NY, USA, ACM Press (1982) 98–101
16. Chow, F.C., Hennessy, J.L.: The priority-based coloring approach to register allocation. ACM Trans. Program. Lang. Syst. **12**(4) (1990) 501–536

17. Chow, F.C.: Minimizing register usage penalty at procedure calls. In: PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation, New York, NY, USA, ACM Press (1988) 85–94
18. Sorkin, A.: Some comments on the priority-based coloring approach to register allocation. SIGPLAN Not. **31**(7) (1996) 25–29
19. Kim, H., Gopinath, K., Kathail, V.: Register allocation in hyper-block for epic processors. In D'Hollander, E.H., Joubert, J.R., Peters, F.J., Sips, H., eds.: Parallel Computing: Fundamentals & Applications, Proceedings of the International Conference ParCo'99, 17-20 August 1999, Delft, The Netherlands, Imperial College Press (2000) 550–557
20. Briggs, P., Cooper, K.D., Kennedy, K., Torczon, L.: Coloring heuristics for register allocation. In: PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation, New York, NY, USA, ACM (1989) 275–284
21. Stephenson, M., Amarasinghe, S., Martin, M., O'Reilly, U.M.: Meta optimization: improving compiler heuristics with machine learning. In: PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, New York, NY, USA, ACM Press (2003) 77–90
22. Cavazos, J., Moss, J.E.B., O'Boyle, M.F.P.: Hybrid optimizations: Which optimization algorithm to use? In: CC. (2006) 124–138
23. Waterman, T.: Adaptive Compilation and Inlining. PhD thesis, Rice University, Houston, TX, USA (2005)
24. Cavazos, J., O'Boyle, M.F.P.: Automatic tuning of inlining heuristics. In: SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing, Washington, DC, USA, IEEE Computer Society (2005) 14
25. Kulkarni, P., Zhao, W., Moon, H., Cho, K., Whalley, D., Davidson, J., Bailey, M., Paek, Y., Gallivan, K.: Finding effective optimization phase sequences. In: LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems, New York, NY, USA, ACM (2003) 12–23
26. Kulkarni, P.A., Whalley, D.B., Tyson, G.S., Davidson, J.W.: Exhaustive optimization phase order space exploration. In: CGO '06: Proceedings of the International Symposium on Code Generation and Optimization, Washington, DC, USA, IEEE Computer Society (2006) 306–318
27. Kulkarni, P.A., Whalley, D.B., Tyson, G.S., Davidson, J.W.: In search of near-optimal optimization phase orderings. In: LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems, New York, NY, USA, ACM (2006) 83–92
28. Almagor, L., Cooper, K.D., Grosul, A., Harvey, T.J., Reeves, S.W., Subramanian, D., Torczon, L., Waterman, T.: Finding effective compilation sequences. In: LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, New York, NY, USA, ACM (2004) 231–239
29. Cooper, K.D., Grosul, A., Harvey, T.J., Reeves, S., Subramanian, D., Torczon, L., Waterman, T.: Exploring the structure of the space of compilation sequences using randomized search algorithms. J. Supercomput. **36**(2) (2006) 135–151
30. Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M.F.P., Thomson, J., Toussaint, M., Williams, C.K.I.: Using machine learning to focus iterative optimization. In: CGO '06: Proceedings of the International Symposium on Code Generation and Optimization, Washington, DC, USA, IEEE Computer Society (2006) 295–305