

# A COMBINED ALGORITHM FOR GRAPH-COLORING IN REGISTER ALLOCATION

M. ALLEN, GIRIDHAR KUMARAN, AND TONG LIU

ABSTRACT. Our research involves improved algorithms for graph-coloring, in the context of register allocation. We extend the usual algorithm, first proposed by Chaitin, adding two further routines, one a form of semi-randomized greedy allocation of colors, and the second using local search with random restarts, a method developed in the context of logical satisfiability problems. For typical register-set sizes, the extended algorithm can color graphs using significantly less time and spilling significantly smaller numbers of nodes to memory than does the Chaitin algorithm; these advantages become less pronounced, as register-set size decreases, and Chaitin can offer some small measure of better performance for very small sizes. Our algorithm has some interesting additional features. On the one hand, it can be extended to the general graph-coloring problem, outside of the particular application considered. In addition, it makes available adjustable parameters for producing more- or less-optimal executables during compiler runs.

## 1. INTRODUCTION TO THE PROBLEM

One of the most common methods for performing register allocation during the compilation of code is by way of graph-coloring algorithms. Following on the instruction-scheduling portion of the process, the compiler attempts to allocate registers to the compiler analyses sections of the code in order to create a series of *interference graphs*. As described by Appel and Ginsburg [1], “Each node in the interference graph represents a temporary value; each edge  $(t_1, t_2)$  indicates a pair of temporaries that cannot be assigned to the same register.” So, a set of temporaries with their interferences corresponds to a particular graph with edges between those nodes that interfere with one another.

A permissible register allocation scheme, then, is one that never assigns two interfering nodes of the graph to the same register. This amounts to a version of the *k-coloring problem* for graphs, as follows. If our target machine has  $k$  registers,  $\{r_1, r_2, \dots, r_k\}$ , then a permissible register allocation scheme for some set of temporaries is identical to a *k-coloring* of the interference graph such that no neighbouring nodes—that is, nodes sharing an edge—are of the same color. If such a coloring *cannot* be found, then the compiler is unable to assign all the temporaries to registers while still respecting interferences. So, to ensure that the section of code being compiled runs properly, it is necessary that some of its temporaries be *spilled* to main memory, to be swapped in and out at runtime using loads and stores. Due to the higher latency of memory access as opposed to register use, spills represent a performance penalty taken by the compiled program. Too, since the insertion of a spill means that extra code must be inserted to deal with the loads and stores—a situation that can create new temporaries and new interferences in the overall graph—any spill can increase the

---

*Date:* May 3, 2005.

*Key words and phrases.* Compilers, register allocation, graph-coloring, GSAT.

M. Allen’s research has been supported by NSF Grant IIS-9907331. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of the NSF..

amount and difficulty of the work that the compiler has to perform. The goal, then, is to reduce the number of spills as much as possible.

## 2. GOALS AND METHODOLOGY

Our work attempts to improve upon the standard method of solving graph-coloring problems in compilers. Our main goals are two-fold:

- (1) Generate a *time-constrained* graph-coloring algorithm—one that works in a reasonable amount of time, relative to existing methods.
- (2) Improve upon the standard with respect to *spill cost*: i.e., an algorithm that generates less spills, and so less memory accesses in general.

We now describe the combination of methods used in our approach.

**2.1. The Chaitin algorithm.** The standard algorithm for coloring graphs during register allocation was first put forth by Chaitin, et al [4, 5], who first formulated register allocation as a graph-coloring problem, and put forth an algorithm originally proposed by Kempe [6]. Despite its relative simplicity, this algorithm remains the most commonly-used graph-coloring method for use in modern compiler technology:

---

For any graph  $G$  and number of colors  $k$ :

- (1) Remove from  $G$  some vertex  $v$  with number of neighbours *strictly less than*  $k$ , and push  $v$  onto stack.
  - (2) Remove all edges for vertex  $v$  from  $G$ , and repeat step (1) until either:
    - (a) All vertices are removed, and  $G$  is empty; or
    - (b) There exists no vertex  $v$  with less than  $k$  neighbours.
  - (3) If condition (2a) holds, then color  $G$ , by popping vertices from the stack one by one, reassembling the full graph as we go, and assigning to each vertex  $v$  as it is popped some color  $c$  such that no current neighbour of  $v$  is already colored  $c$ .
  - (4) Else, if (2b) holds, select some vertex  $v$  to be spilled, remove  $v$ , and begin again.
- 

The resulting algorithm is sound, but incomplete—it is easy to find even very small  $k$ -colorable graphs for which the method is guaranteed to fail in finding a  $k$ -coloring. (A simple example is the graph shown in Figure 1, which is 2-colorable, but for which the Chaitin algorithm fails, since every node has more than one neighbour.) However, given efficient representations of graphs,<sup>1</sup> the Chaitin algorithm has the advantage that it can run in time linear in the number of vertices, for  $k$ -colorable graphs; it is thus hoped that the number of problem instances for which the algorithm generates unnecessary spills is relatively small, and that the cumulative cost of spills remains low.

**2.2. A modified satisfiability algorithm.** Most research into altering the algorithms used in register allocation attempts to preserve the basic Chaitin algorithm, while minimizing the cost of spills generated by that algorithm. For instance, a body of work [8, 9, 10, 11, 12] has concentrated upon producing new heuristics for use in the spill phase of the code, trying to make better, or more sophisticated, choices as to which nodes to spill, and when. Others attempt to combine spilling

---

<sup>1</sup>For discussion, see Cooper, Harvey, and Torczon [7].

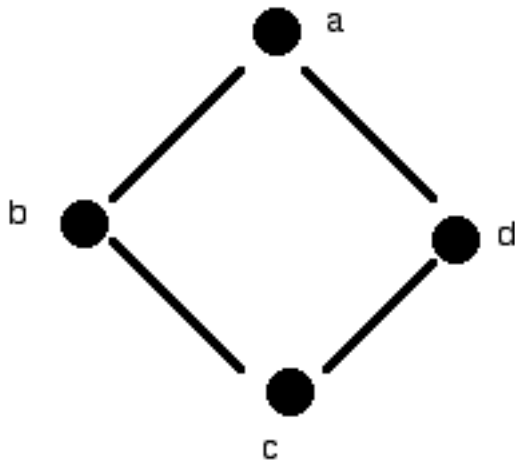


FIGURE 1. A 2-colorable graph for which the Chaitin algorithm is guaranteed to fail.

with other methods, such as live range splitting [13], or on trying to optimize the placement of inserted spill code [14]. While all of these possibilities have merit, we consider altering the actual graph coloring algorithm itself.

We base our work in the first place on the randomized local search algorithm GSAT, first developed by Selman, Levesque and Mitchell [15]. For any boolean logical expression  $\varphi$  in CNF form, GSAT begins by randomly assigning truth-values to variables, and then performs a hill-climbing search, altering one variable's value so as to maximize the number of clauses of  $\varphi$  made true. If, after a fixed number of iterations of the search process, no satisfying assignment has been found, the algorithm restarts with a new random initial truth-assignment, for a fixed number of restarts. It is easy to see that GSAT is another instance of a sound but incomplete algorithm. While any satisfying assignment it finds for a logical expression is correct, GSAT may terminate without having found any such assignment, even where such a one exists.

While the authors of the original GSAT study briefly consider the possibility of using this algorithm for graph-coloring problems, the method they describe is generally infeasible. In their discussion, a graph-coloring problem is converted to a logical expression that both describes the structure of the overall graph, and the constraints on the coloring in question. However, this sort of explicit method can generate very large logical expressions: a graph problem with 125 vertices and 18 colors results, under their encoding, in an expression consisting of 89,288 clauses with 2,250 variables, for which GSAT took 5 hours to run. Such an encoding of problems is thus far too cumbersome to be of any real use in compilers, which can be dealing with tens of thousands of graphs, some of size in the thousands of vertices, all in the compilation of a single program. To avoid this problem, we modified the GSAT procedure as follows.

---

For any graph  $G$  and number of colors  $k$ :

- (1) For a fixed number of restarts (**MAX1**), or until a permissible coloring is found, do the following:
    - (a) Randomly partition the vertices of  $G$  into  $k$  separate partitions, representing the  $k$  available colors.
    - (b) For a fixed number of iterations (**MAX2**), or until a permissible coloring is found, do the following:
      - (i) Select that vertex  $v$  such that recoloring  $v$  *most reduces* the number of conflicts in the current coloring (i.e., the number of neighbouring nodes of the same color).
      - (ii) Recolor vertex  $v$  by assigning it to the next available partition; that is, if  $v$  is currently in partition  $i$ , move it to partition  $i + 1$  (modulo the number of partitions).
    - (c) If after **MAX2** iterations, we have still not found a permissible coloring, then *restart*, going back to step (1).
  - (2) If after **MAX1** restarts, we have still not found a permissible coloring, then select some vertex  $v$  to be *spilled* from the graph, remove  $v$ , and begin again.
- 

In effect, this algorithm treats the graph-coloring problem as a *multi-valued* logical satisfiability problem. Rather than having two truth-values for an individual variable, we have up to  $k$  values through which we cycle. Each vertex of the graph can be considered a single variable, and we treat the original adjacency matrix of the graph as our logical expression: a clause of the expression is an edge between two vertices, and it is satisfied, in this context, iff those two vertices are assigned distinct values. This provides a much smaller encoding of the coloring problem than that proposed in the original work on GSAT, especially since it does not require any direct translation to a logical expression, but instead treats the adjacency matrix of the graph itself as equivalent to same. Furthermore, this method provides something similar to another local search method used in graph-coloring investigations, namely Tabu Search [16, 17], without all of the necessary overhead, and with some important differences. Whereas Tabu search retains a “taboo list” of recently visited configurations in order to avoid local maxima in the search procedure, the modified GSAT method, by virtue of its ordering or color partitions, will not revisit just visited states, since it only considers recolorings of individual nodes that *increase* the color assigned to some vertex  $v$ . Too, we do not need to introduce pre-defined restrictions on the number of next neighbours considered in the search procedure, as is usual in Tabu search. Instead, this number is automatically restricted at each iteration of local search, since we generally consider only those recolorings that increase the color of individual nodes by at most one color (but see the discussion of complications with this feature in Section 2.3, below). Lastly, we exploit the randomness in initial partitioning at each restart, allowing us, in theory, to visit more diverse parts of a large search space without running our algorithm for extremely long periods of time.

**2.3. A combined algorithm.** Preliminary testing indicated three main deficits with the modified GSAT algorithm:

- (1) If we did not restrict the number of nodes that we had to consider for a graph  $G$ , then the algorithm ran much too slow to be of much use in practice.
- (2) The random restarts tended to create a large number of conflicts right off the bat; the search method spent a lot of time trying to remove these unnecessary conflicts before it could get to work on the real problem.
- (3) Unless we increased our values **MAX1** and **MAX2** to relatively high numbers, the performance of the search technique was below that attained by the Chaitin algorithm. Unfortunately, very high values made performance even slower, particularly for those graphs for which some spills had to be generated.

In order to solve these problems, we made four major adjustments to our method:

- (1) **Pruning the graph:** Rather than perform search over the entire graph, it is sufficient to do so for just those vertices left over after the Chaitin algorithm removes all those it can. If a permissible coloring of the remainder can be found, those already pushed onto the stack can be recolored as before. Thus, we use the fast Chaitin routine as a front-end to our main algorithm, reducing the problem. If the Chaitin algorithm can reduce the graph completely, then no more need be done, and our resulting algorithm works on all graphs for which Chaitin finds a solution without spills.
- (2) **Semi-greedy partitioning:** Because purely random partitions tend to start the search procedure in unnecessarily difficult parts of the search space, we altered the initial partitioning procedure in order to reduce initial conflicts. Our partitions are thus generated by randomly choosing nodes from those remaining after pruning, and then assigning each to the first available partition for which there are no conflicts. If, for some vertex, there are no such conflict-free partitions—since every partition already contains one of its neighbours—it is randomly assigned to one of them as before. (It is worth noting that the addition of this method alone completely solves some graphs that the Chaitin method cannot, generating no spills at all.)
- (3) **Inversely proportional MAX values:** Since we want our algorithm to run in a time-effective manner overall, it was necessary to minimize run time of the search over very large sets of hard-to-color vertices. We therefore set the MAX parameters *inversely proportional* to the size of this set, weighting them further so that they decreased in proportion to the number of overall conflicts present in the set. Thus, for a large set with many existing conflicts, the search algorithm is *pessimistic*, stopping earlier in cases where a resolution seemed less likely.
- (4) **Avoiding hopeless cases:** We also reduced overall iterations by introducing a feature that immediately ceased search if it encountered a partition for which increasing the color of any node by one (1) only *increased* the number of overall conflicts. In such a case, we then restarted search, considering the case where we increased color of nodes by two (2), instead, and so on for all  $(k - 1)$  possible increments in the color of an individual node. If all such increments produced only increases in conflict, the search algorithm immediately restarted and re-partitioned. The number of checks performed in such a case only ends up being at most  $((k - 1) \times n)$ , for  $n$  the number of nodes, rather than all possible recolorings  $(k^n)$ .

Our final algorithm, hereafter referred to as the **combined algorithm** (or “Combo”) thus involved combination of three diverse methods within one routine:

---

For any graph  $G$  and number of colors  $k$ :

- (1) Use the Chaitin method to prune nodes with less than  $k$  neighbours from  $G$ , pushing each to a stack as before.
  - (2) If, after Step (1), we have emptied the graph and pushed all nodes to the stack, then color in reverse order as before for the Chaitin algorithm.
  - (3) Else if, after Step (1), some vertices remain in the graph, perform the modified GSAT routine on this remainder, using:
    - (a) Semi-greedy partitioning.
    - (b) **MAX** values inversely proportional to the size of the remaining set.
  - (4) If, any time during Step (3), we find a permissible partitioning of the remaining vertices, then halt, color those vertices according to that partitioning, and then color those vertices on the stack in reverse order as for the Chaitin algorithm, constrained by the coloring already introduced.
  - (5) Else if, after Step (3), we have still not found a permissible coloring, then select some vertex  $v$  to be *spilled* from the graph, remove  $v$ , and begin again.
- 

### 3. TESTING AND RESULTS

Initial tests of our algorithm against the Chaitin method were hampered by a decision to write each as a stand-alone unit in Java. However, because of the relatively low performance of the Java Virtual Machine, this provided no consistent measure of program speed, and the code was subsequently translated to C++, in which context the significant speed-up allowed more testing, and constant results.

Graphs were presented to the algorithms in the form of text files containing descriptions of their adjacency matrices. As data, we used a set of interference graphs made available by Lal George and Andrew Appel (at [www.cs.princeton.edu/appel/graphdata](http://www.cs.princeton.edu/appel/graphdata)) for use as a general testbed for graph-coloring and register allocation experiments. The testbed contains 27,921 graphs, each generated by the SMLNJ compiler running on its own source code, and targeted to a machine with between 21 and 29 available temporary registers at each step. The graphs range in size between 21 and approximately 6,000 nodes, and represent a range of challenges for coloring algorithms, even with the relatively large number of registers/colors available. Because these graphs do not come with actual spill costs for individual vertices, and because the Chaitin heuristic for spilling bases its decision in part upon this estimate of the “costliness” of individual spills, we simulated spill costs, assigning each node some value at random, following a generally bell-shaped distribution, such that nodes with the very lowest or very highest spill costs made up 5 percent of the nodes overall, and 40 percent of the nodes had spill costs squarely in the middle of the range (see Figure 2). Preliminary results indicated that the relative performance of the two methods was generally unaffected by changes in the particular distribution, or overall distribution pattern, of these costs, and so this distribution was held fixed throughout all our tests.

Following tests over the full testbed, averaging spills generated—a measure that only ever varies, and slightly, for the combined algorithm, given its semi-random elements—we compared results for

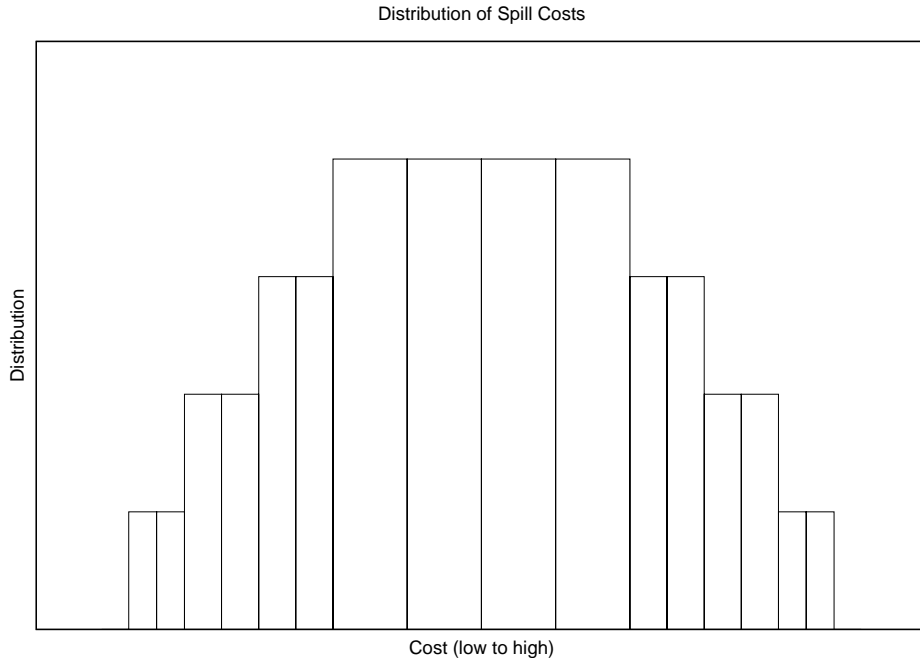


FIGURE 2. Distribution of spill costs in the coloring testbed.

each algorithm when the number of available registers was reduced, first to 16, then to 12, and finally to 8.<sup>2</sup> Due to the increased time taken for each run of each algorithm for these restricted register-set examples, because of extra time spent in spilling and restarting, it was infeasible to test them as many times, but results were relatively constant over multiple runs. Testing was carried out on a Dell Precision 330, running an Intel Pentium4 1.7GHz processor, with 512MB RDRAM.

**3.1. Results.** Prior to the conversion to C++, the clocked testing run-times for each algorithm varied greatly, in large part due to variations in the scheduling of garbage collection in the JVM, and no meaningful comparison was possible. After code conversion, however, we achieved a somewhat surprising result: for reasonable register-set sizes, our algorithm actually outperforms the Chaitin method on both main counts, running both significantly faster, and with far fewer spills, than does the standard algorithm.

Initially, we expected that our algorithm would run somewhat slower than the Chaitin method, due to the additional overhead of the search routines. As will be seen, however, the combined algorithm systematically reduced the number of spills necessary for  $k$ -coloring the various graphs in the testbed, and as a result, even with its additional routines, there were graphs for which our algorithm actually performed *faster* than the Chaitin algorithm, due to its ability to settle on a coloring with fewer overall restarts. In one example, for instance, a graph with 116 nodes (graph 20 in the Appel/George testbed) was only 21-colored by the Chaitin algorithm after spilling 12

<sup>2</sup>As Appel has pointed out (see the source-page for the data), such comparisons are interesting even in the context of large register-set machines, since we may find that the use of registers for purposes other than the storage of program temporaries might speed up overall system performance. Thus, there can be advantages to algorithms that can generate the same number of spills, or less, while using fewer registers.

# of Registers	Spills ( <i>Chaitin</i> )	Spills ( <i>Combo</i> )	Reduction
21/29	2,872	1,398	51.3%
16	6,978	5,071	27.3%
12	14,886	13,073	12.2%
8	41,917	$\leq 41,917$	(n/a)

TABLE 1. Comparison of spills generated by each algorithm over various testbeds.

# of Registers	Seconds ( <i>Chaitin</i> )	Seconds ( <i>Combo</i> )	Reduction
21/29	1,602	687	57.1%
16	4,653	3,078	33.8%
12	13,163	14,674	11.5%
8	14,609	$> 72,800$	(n/a)

TABLE 2. Comparison of time used by each algorithm over various testbeds.

nodes, given our assignment of spill costs; the combined algorithm, on the other hand, was able to eliminate all spills, finding a 21-coloring without spilling any nodes. Thus, for an instance like this one, the Chaitin algorithm had to go through 12 restarts of its main procedure, whereas the combined algorithm was able to color the graph in one pass, thus saving a substantial amount of effort. Added up over the large number of cases in which spills were partially or completely eliminated, this translated finally into notable time savings over the initial testbed.

Over the original set of graphs, using between 21 and 29 registers, the combined algorithm achieved a reduction of spills, relative to the Chaitin method, from 2,872 (or one memory access every 10 interference graphs, on average) to 1,399 (or one memory access every 20 interference graphs, on average) for the combined algorithm: a reduction of 51.3% in the number of spills overall. In addition, the runtime for the combined algorithm averaged 687 seconds over the entire testbed, as compared to the Chaitin method’s 1,602 seconds, a reduction of 57.1%. As we reduced the registers available, however, this margin of improvement narrowed, and eventually disappeared. While the combined algorithm consistently out-performed the Chaitin algorithm with respect to number of spills, reduction in the number of available registers corresponds to a reduction in the relative benefit of performing the combined method. In addition, the reduction of available registers to 12 or below coincides with an eventual increase in the time taken by the combined algorithm, over the time taken by the Chaitin method. Tables 1 and 2, as well as Figures 3 and 4, summarize the differences in spills generated, and time spent, as between the two algorithms. Reductions are measured relative to the Chaitin method. A negative value indicates that the combined method performed less well.<sup>3</sup>

<sup>3</sup>Testing was conducted averaging run-times and spills over a number of runs, over the entire Appel testbed of 27,921 graphs. (Note that the number of spills for the Chaitin algorithm is an exact figure, constant over all runs.) The tests were run as follows:

- (1) **21/29 registers:** Combined: 100 runs; Chaitin: 20 runs.
- (2) **16 registers:** Combined: 20 runs; Chaitin: 20 runs.
- (3) **12 registers:** Combined: 20 runs; Chaitin: 8 runs.
- (4) **8 registers:** Combined: 3 runs; Chaitin: 1 run (not completed, due to time constraints).

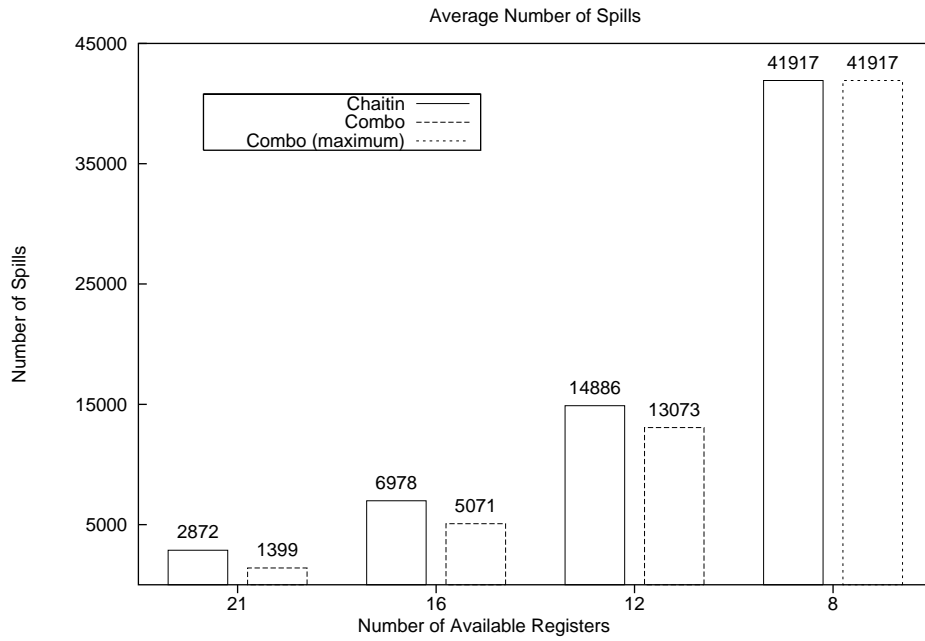


FIGURE 3. Comparison of spills generated by each algorithm over various testbeds.

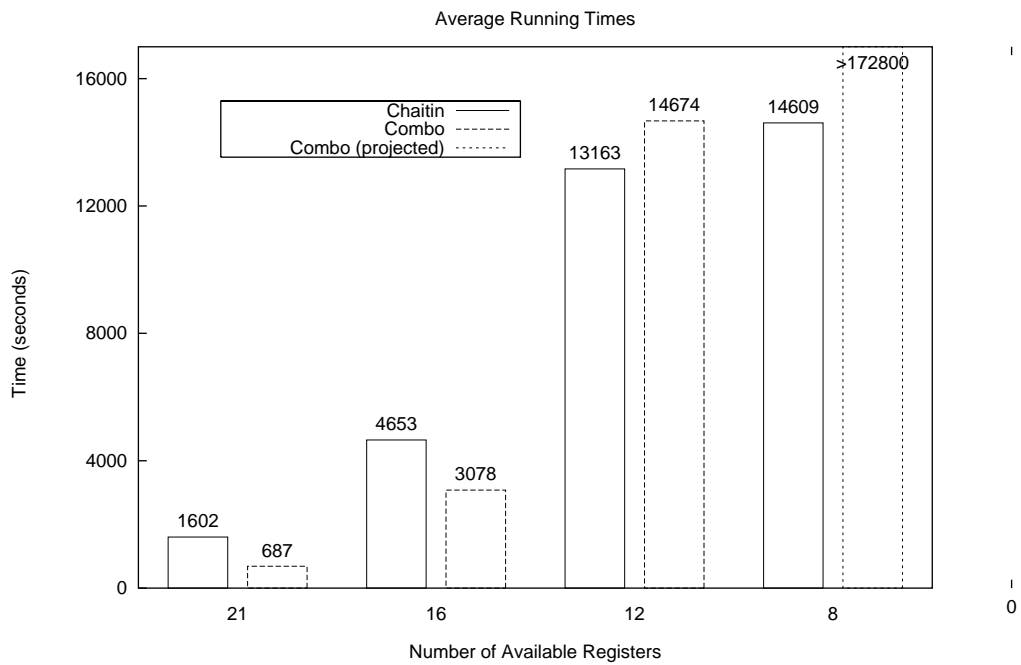


FIGURE 4. Comparison of time used by each algorithm over various testbeds.

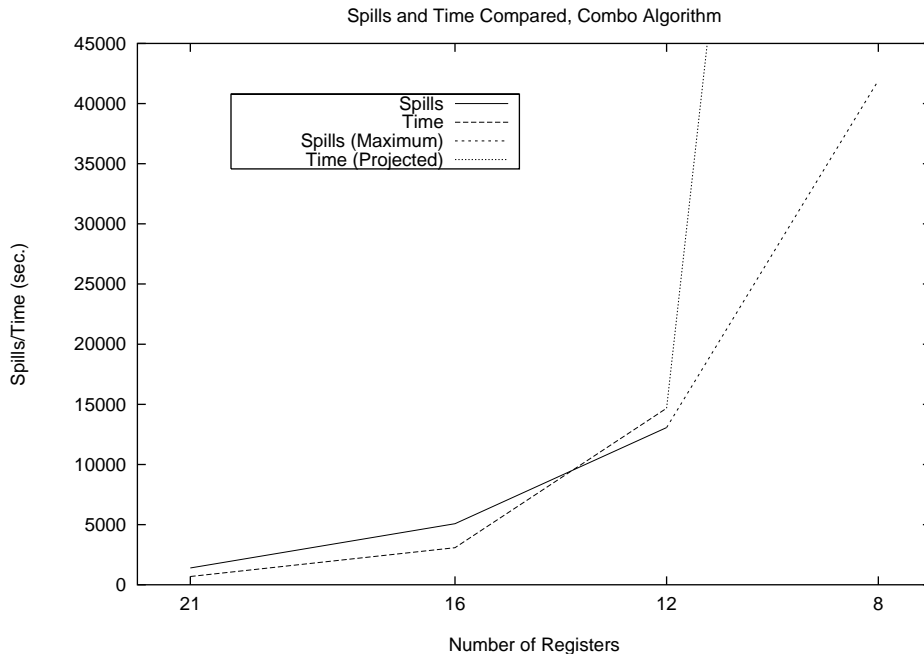


FIGURE 5. Comparison of time used and spills generated by the combined algorithm over various testbeds.

#### 4. CONCLUSIONS AND FUTURE WORK

Preliminary results for our combined algorithm have been favourable, but we would like to come to a better understanding of its full features and capabilities. Initial tests show that the addition of simple partitioning and local search routines to the standard register-allocation graph-coloring algorithm provides a significant boost in performance, running faster and reducing memory latency by over 50% each. This suggests that further work in this area would be of use in applications, as there are many other methods for graph-coloring that might be of use in this domain. The initial thought, that additional routines might only increase running time of the compiler without significantly improving memory access spilling, turns out not to be the case. Instead, quite the contrary is true: by reducing spills, overall processing time is in fact reduced, resulting in an improved algorithm in both main dimensions of performance. In addition, because of the potential for improved time performance, there is more room for further investigation into reducing spills further—since the combined algorithm can run in half the time of the Chaitin algorithm, further optimizations may be possible without exceeding the time bounds set by the standard method.

At the same time, further work can be done. Future testing would involve:

- (1) Isolating the two additions to the Chaitin algorithm (the initial greedy partitioning, and the local search routine) from one another, in order to see how much the performance increases observed can be attributed to each in turn.
- (2) More fine-grained testing to understand the trade-offs between spill minimization, processing time, and number of available registers. As described, the combined algorithm suffered a drop-off in its time performance relative to the proportional reduction of spills as the number of registers/colors was reduced. In addition, processing time grows somewhat faster

than additional spills (which never grows larger than the number generated by the Chaitin algorithm, by definition), suggesting that there exists a point at which it no longer becomes productive to consider the use of the combined algorithm (see Figure 5). More investigation is needed in order that something definite can be said about these trade-offs, and about the reasons why the two algorithms vary so in their performance with the register count.

- (3) Investigating the possibility of improved *heuristics* for use in the spill phase of our program. One possibility that comes readily to mind would be to use information about the various conflicts generated by individual nodes during the search phase of the algorithm; nodes for which every recoloring considered still yielded a large number of conflicts would make for more likely, and perhaps more useful, candidates for spilling.
- (4) Further testing of the algorithms over interference graphs generated by compilers for a number of different platforms and languages.
- (5) Adaptation of the method to a general graph-coloring problem, by letting our algorithm vary the number  $k$  of colors considered, repeating for higher  $k$  upon failure.

Clearly, this work merely scratches the surface of what is possible when more sophisticated graph-coloring methods are used in this application context. By continuing our testing over the other test instances, and by comparing results with a wider variety of coloring algorithms, we should also be able to say more about the possibilities our own particular algorithm makes available. In particular, it would be interesting to look at our performance relative to that of algorithms that combine a variety of methods similar to our own, including forms of greedy partitioning and search [18]. Of real interest would be the role of the randomization elements present in the GSAT-style search method, and the potential benefits to be had by way of the pruning methods introduced by the Chaitin algorithm; as far as we know, this means of both combining methods while working first to reduce the size of the problem has not been extensively investigated.

Finally, we note one more interesting feature of the use of a modified GSAT-type algorithm in compilers. It is often desired that a compiler be adjustable with respect to the amount of optimization it puts into code. During early stages of code writing and testing, it is often desirable that compilation proceed as quickly as possible, due to the frequency with which segments of code need to be recompiled. So long as a compiler produces working executable units, we can usually omit many available optimizations, leaving them until final compilation stages, at which point the time required for compilation can be allowed to run longer. It is common, then, for compilers to offer the option to turn optimization off or on as desired by the user, speeding up or slowing down the process depending on circumstance. Our modified algorithm offers a ready-made means of adjusting such a process with respect to register allocation, namely the adjustment of the parameters governing the number of repetitions performed by the local search phase. By reducing these parameters (to zero if desired), we can reduce the action of the local search process, at the cost of less effort spent to find spill-reducing colorings. In final compilation stages, on the other hand, we can increase these parameters significantly, and so increase the amount of time spent on attempts to reduce memory accesses, providing a greater measure of optimization to the resulting executable units. The method proposed thus offers practical benefits without always demanding more time in the bargain.

## REFERENCES

- [1] APPEL, ANDREW W. AND GINSBURG, MAIA. 1998. *Modern Compiler Implementation in C*. Cambridge University Press, Cambridge-New York-Melbourne.

- [2] KARP, RICHARD M. 1972. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, Eds. Plenum Press, New York-London, 85–104.
- [3] GAREY, M. R., JOHNSON, D. S., AND STOCKMEYER, LARRY. 1976. Some simplified *NP*-complete graph problems. *Theoretical Computer Science* **1**, 237–267.
- [4] CHAITIN, GREGORY J., AUSLANDER, MARC A., CHANDRA, ASHOK K., ET AL. 1981. Register allocation via coloring. *Computer Languages* **6**, 47–57.
- [5] CHAITIN, G. J. 1982. Register allocation & spilling via graph coloring. *SIGPLAN Notices* **17**, 98–105.
- [6] KEMPE, A. B. 1879. On the geographical problem of the four colors. *American Journal of Mathematics* **2**, 193–200.
- [7] COOPER, KEITH D., HARVEY, TIMOTHY J., AND TORCZON, LINDA. 1988. How to build an interference graph. *Software—Practice and Experience*, **1**, 1–22.
- [8] BERNSTEIN, DAVID, ET AL. 1989. Spill code minimization techniques for optimizing compilers. *SIGPLAN Notices*, **24**, 258–263.
- [9] BRIGGS, PRESTON, COOPER, KEITH D., KENNEDY, KEN, AND TORCZON, LINDA. 1989. Coloring heuristics for register allocation. *SIGPLAN Notices*, **24**, 275–284.
- [10] BRIGGS, PRESTON. 1992. *Register allocation via graph coloring*. Ph.D. thesis, Rice University.
- [11] BRIGGS, PRESTON, COOPER, KEITH D., AND TORCZON, LINDA. 1992. Coloring register pairs. *ACM Letters on Programming Languages and Systems (LOPLAS)*, **1**, 3–13.
- [12] BRIGGS, PRESTON, COOPER, KEITH D., AND TORCZON, LINDA. 1994. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **16**, 428–455.
- [13] COOPER, KEITH D., AND SIMPSON, L. TAYLOR. 1998. Live range splitting in a graph coloring register allocator. To appear in *Proceedings of the 1998 International Compiler Construction Conference*. Manuscript available at: <http://softlib.rice.edu/MSCP/publications.html>.
- [14] CALLAHAN, DAVID, AND KOBLLENZ, BRIAN. 1991. Register allocation via hierarchical graph coloring. *SIGPLAN Notices*, **26**, 192–203.
- [15] SELMAN, BART, LEVESQUE, HECTOR, AND MITCHELL, DAVID. 1992. A new method for solving hard satisfiability problems. *AAAI-92*, 440–446.
- [16] HERTZ, ALAIN, AND DE WERRA, DOMINIQUE. 1987. Using Tabu search techniques for graph coloring. *Computing*, **39**, 345–351.
- [17] HERTZ, ALAIN, TAILLARD, ERIC, AND DE WERRA, DOMINIQUE. 1995. A tutorial on Tabu search. *Proceedings of Giornate di Lavoro AIRO-95*, 13–24.
- [18] CULBERSON, JOSEPH C. 1992. *Iterated greedy graph coloring and the difficulty landscape*. Technical Report TR 92-07, University of Alberta, Canada.

(Allen and Kumaran) COMPUTER SCIENCE, UNIVERSITY OF MASSACHUSETTS, AMHERST, MA 01003  
E-mail address: [mwallen, giridhar]@cs.umass.edu

(Liu) ELECTRICAL AND COMPUTER ENGINEERING, UNIVERSITY OF MASSACHUSETTS, AMHERST, MA 01003  
E-mail address: tliu@ecs.umass.edu