What's Wrong with Graph Coloring?

David Koes and Seth Copen Goldstein
Computer Science Department
Carnegie Mellon University
{dkoes,seth}@cs.cmu.edu

Graph coloring is the de facto standard technique for register allocation within a compiler. In this paper we examine the intuition that a better coloring algorithm results in better register allocation. By replacing the coloring phase of the `gcc` compiler's register allocator with an optimal coloring algorithm, we demonstrate both the importance of extending the graph coloring model to better express the costs of allocation decisions and the unsuitability of a pure graph coloring model of register allocation.

1 Introduction

Register allocation is one of the most important optimizations a compiler performs and is becoming increasingly important as the gap between processor speed and memory access time widens. The textbook [2, 16, 1] approach for performing register allocation begins by building an interference graph of the program. If variables interfere, they cannot be assigned to the same register. Thus, if there are k registers, register allocators attempt to solve the NP-complete problem of finding a k-coloring of a graph. If not all the variables can be colored with a register assignment, some variables are spilled to memory and the process is repeated.

An initial intuition one might have is that the quality of the register allocation found by a graph coloring register allocator would be primarily dictated by performance of the coloring algorithm.

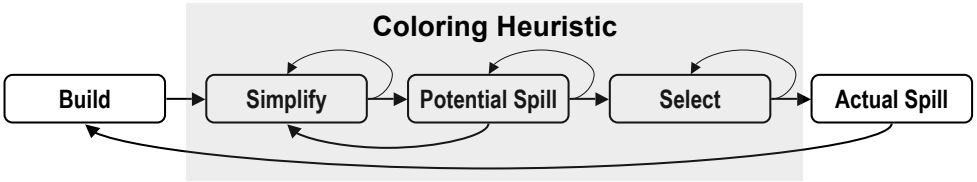

Figure 1: The flow of a traditional graph coloring algorithm.

We debunk this intuition by comparing a simple heuristic coloring algorithm to an optimal algorithm. Not only does the heuristic usually find as good a coloring as the optimal algorithm, but we show that the quality of the register allocation is determined by factors other than the quality of coloring, such as how spill decisions are made and extensions to the simple graph model which more accurately model the costs of allocation. We deconstruct the performance of a graph coloring register allocator by examining the effect on performance as these extensions to the simple graph model are added to an optimal coloring pass.

We describe the standard algorithm for graph coloring register allocation in Section 2 and our optimal coloring algorithm in Section 3. Our evaluation procedure is described in Section 4 with results given in Section 5. We conclude with some discussion in Section 6.

2 Graph Coloring

2.1 Algorithm

The traditional optimistic graph coloring algorithm[6, 8, 7] consists of five main phases as shown in Figure 1:

Build An interference graph is constructed using the results of data flow analysis. A node in the graph represents a variable. An edge connects two nodes if the variables represented by the nodes interfere and cannot be allocated to the same register. Restrictions on what registers a variable may be allocated to can be implemented by adding precolored nodes to the graph.

Simplify A heuristic is used to help color the graph. Any node with degree less than k, where k is the number of available registers, is removed from the graph and placed on a stack. This is repeated until all nodes are removed, in which case we skip to the Select phase, or no nodes can be simplified.

Potential Spill If only nodes with degree greater than k are left, we mark a node as a potential spill node, remove it from the graph, and optimistically push it onto the stack. We repeat this process until there exist nodes in the graph with degree less than k, at which point we return to the Simplify phase.

Select In this phase all of the nodes have been removed from the graph. We now pop the nodes off the stack. If the node was not marked as a potential spill node then there must be a color we can assign this node that does not conflict with any colors already assigned to this node's neighbors. If it is a potential spill node, then it still may be possible to assign it a color; if it is not possible to color the potential spill node, we mark it as an actual spill and leave it uncolored.

Actual Spill If any nodes are marked as actual spills, we generate spill code which loads and stores the variable represented by the node into new, short lived, temporary variables everywhere

the variable is used and defined. Because new variables are created, it is necessary to rebuild the interference graph.

Note that the Simplify, Potential Spill, and Select phases together form a heuristic for graph coloring. If this heuristic is successful, there will be no actual spills. Otherwise, the graph is modified so that it is easier to color by spilling variables and the entire process is repeated.

2.2 Improvements

A number of improvements to the basic graph coloring algorithm have been proposed. Four common improvements are:

Web Building [13, 8] Instead of a node in the interference graph representing all the live ranges of a variable, a node can just represent the connected live ranges of a variable (called webs). For example, if a variable i is used as a loop iteration variable in several independent loops, then each loop represents an unconnected live range. Each web can then be allocated to a different register, even though they represent the same variable.

Coalescing [11, 8, 7] If the live ranges of two variables are joined by a move instruction and the variables are allocated to the same register it may be possible to coalesce (eliminate) the move instruction. Coalescing is implemented by adding move edges to the interference graph. If two nodes are connected by a move edge, they should be assigned the same color. Move edges can be removed to prevent unnecessary spilling.

Spill Heuristic [5] A heuristic is used when determining what node to mark in the Potential Spill stage. An ideal node to mark is one with a low spill cost (requiring only a small number of

dynamic loads and stores to spill) but one whose absence will make the interference graph easier to color and therefore reduce the number of future potential spill nodes.

Improved Spilling [4, 7, 9] If a variable is spilled, loads and stores to memory may not be needed at every read and write of the variable. It may be cheaper to rematerialize the value of the variable (if it is a constant, for example). Alternatively, the live range of the variable can be partially spilled. In this case, the variable is only spilled to memory in regions of high interference.

3 Optimal Coloring

In this paper we investigate the relationship between the quality of the coloring and the resulting register allocation by replacing the coloring heuristic of a traditional allocator with an optimal coloring algorithm. Our optimal coloring algorithm transforms the graph coloring problem into an integer linear program (ILP) that we solve using a commercial optimizer.

Given a graph with N nodes and K colors, we create an ILP with $N * K$ binary variables, n_k, which are constrained to be one if and only if node n is assigned color k and zero otherwise. Every node n has a sufficiency condition:

$$\sum_{k=1}^{K} n_k = 1$$

which states that a node must be assigned exactly one color. In addition, every edge (n, m) imposes

a coloring constraint for every color k:

$$n_k + m_k \leq 1$$

which states that nodes connected by an edge cannot both be assigned the same color.

Although this ILP formulation exactly describes the graph coloring problem, it is not flexible enough to be used inside of a register allocator since interference graphs are not always K-colorable. Instead, we assign a cost to leaving a specific node uncolored. The optimal coloring minimizes this cost. We consider two different optimality metrics for graphs that are not K-colorable:

Number of Spilled Variables In this case the optimal coloring is the coloring which leaves the minimum number of nodes uncolored.

Spill Cost In this case each node is assigned a spill cost and the optimal coloring is the coloring which minimizes the total spill cost of all the uncolored nodes. The spill cost is the same spill cost used by `gcc`'s allocator. It is the sum of the costs of the loads and stores needed to spill the variable weighted by the expected frequency of each memory operation (that is, spills inside loops cost more).

Both notions of optimality are simple to add to our ILP model by introducing an additional binary variable for each node, n_{spill}, which is one if and only if node n should be left uncolored. This variable is incorporated into the sufficiency constraints, but not the coloring constraints. In

order to minimize the cost of spilling, we introduce an objective function:

$$\min \sum_{n=1}^{N} c_n n_{spill}$$

where the coefficient c_n is one if we are minimizing the number of variables that are spilled and the value of the spill cost if we are minimizing the total spill cost.

We can also model coalescing using our ILP. For every move edge e with endpoints n and m in the interference graph we introduce a binary variable e_k which is one if and only if the nodes connected by the edge are both assigned k. Then for every color k we add the constraint:

$$e_k \leq n_k \quad e_k \leq m_k$$

so that e_k can only be one if both n_k and m_k are one.

In addition, we add these variables to the objective function with some small negative coefficient, c_e. As long as the sum of these coefficients is less than the cost of the cheapest spill, coalescing will never result in more spills.

Our optimal coloring algorithm is substituted for the heuristic coloring phase of a traditional allocator. The nodes in the graph that are colored with the $spill$ color are spilled, the interference graph is rebuilt, and the process repeats until a coloring is found.

4 Evaluation

We evaluate the effect of using various optimal coloring algorithms by substituting them for the `ra-colorize` function of the graph coloring based allocator of `gcc` version 3.4.3. The graph coloring allocator of `gcc` is enabled with `-fnew-ra` and implements all the improvements discussed in Section 2. The optimal coloring algorithms use CPLEX 9.0 [12] to solve the ILPs.

We consider two different metrics for evaluating the quality of a register allocation: execution speed and code size. For both metrics we use the SPECint200 and SPECfp2000 (omitting the Fortran 90 benchmarks) benchmark suites [20] with the reference input sets for evaluation. When targeting execution speed we compile with `-O3 -funroll-loops` and when targeting code size we compile with `-Os`.

We evaluate the allocators using the x86 architecture; this architecture, with its limited register file, will likely see the biggest impact from the performance of the register allocator. We execute the various benchmarks natively on a 2.8 Ghz Pentium IV with 1 GB of RAM running RedHat Linux 9.0

5 Results

The efficacy of the optimal coloring allocators at eliminating spills is shown in Figure 2. Of the functions of the SPEC benchmark suite, 52.57% can be fully allocated to registers. The heuristic coloring algorithm fails and performs unnecessary spilling in only 6 functions (.13% of the total), indicating that the heuristic is sufficient for determining the colorability of most typical interference

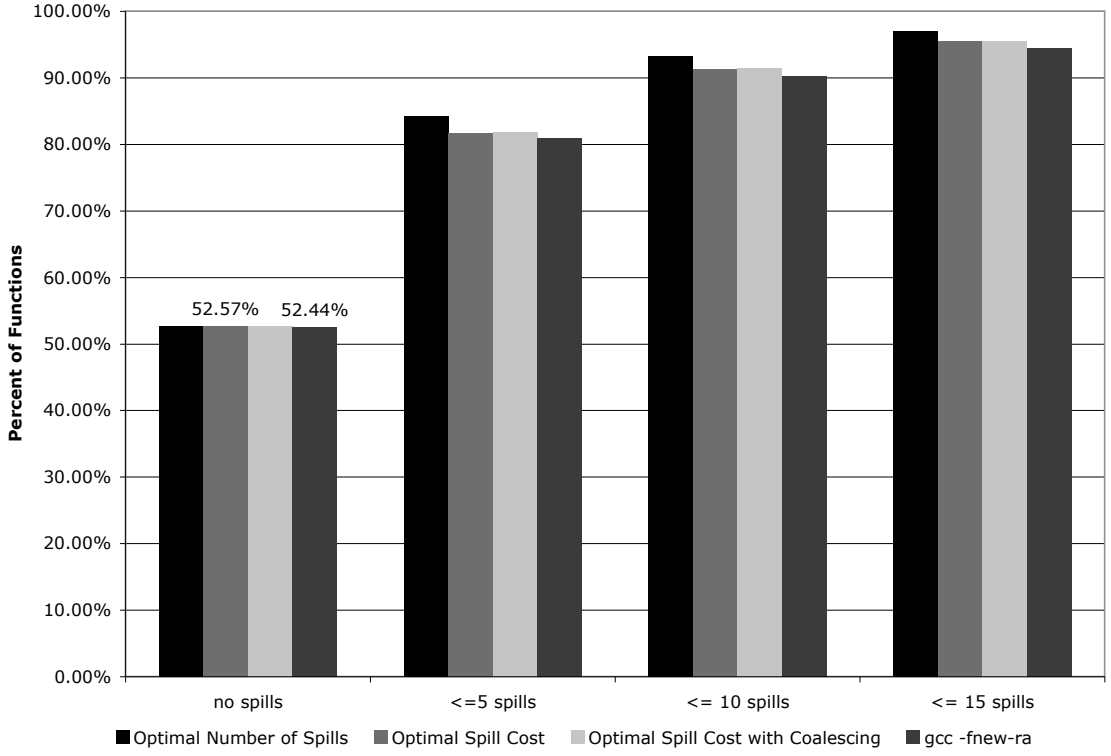

Figure 2: The percent of functions in the SPEC benchmark suite that spill at most a given number of variables. The benchmarks were compiled with `-O3 -funroll-loops`.

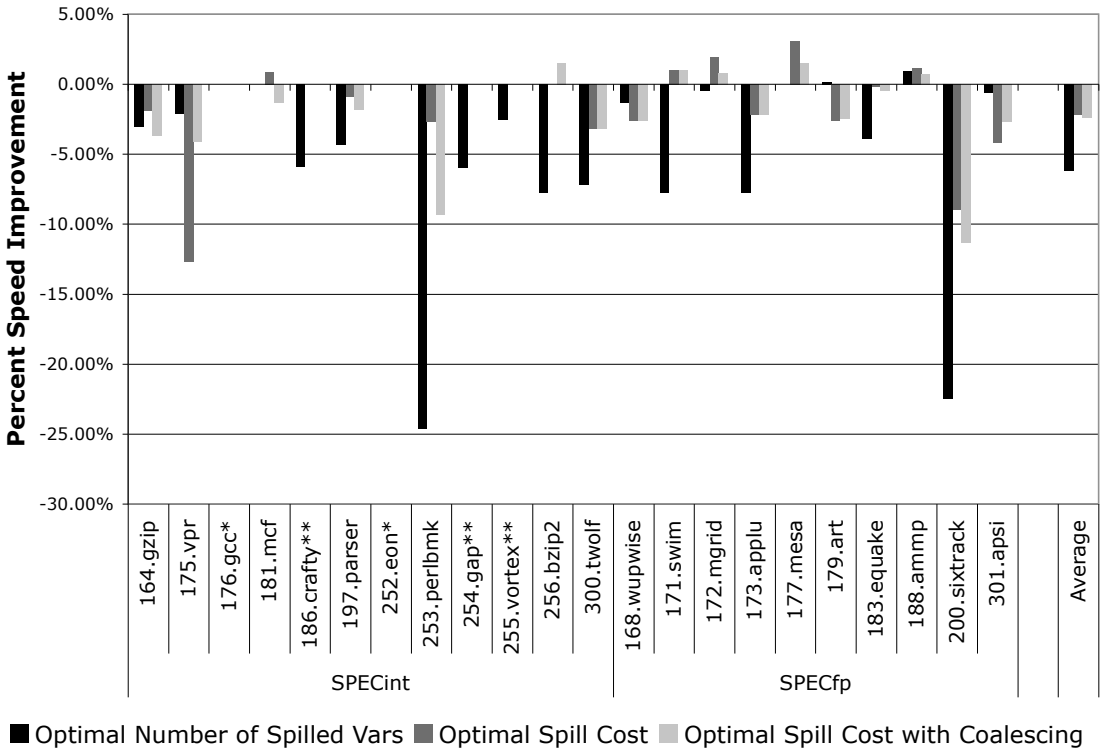

Figure 3: Performance improvement relative to code produced using the `gcc` graph allocator. The average improvement is the geometric mean of the speedups expressed as percentage improvement. The average performance improvement when optimizing the number of spilled variables, the cost of spilling, and the cost of spilling with coalescing is -6.2%, -2.2%, and -2.4% respectively.

*Results are omitted because the baseline compiler failed to produce a correct executable

**Some results are omitted due to errors in compilation.

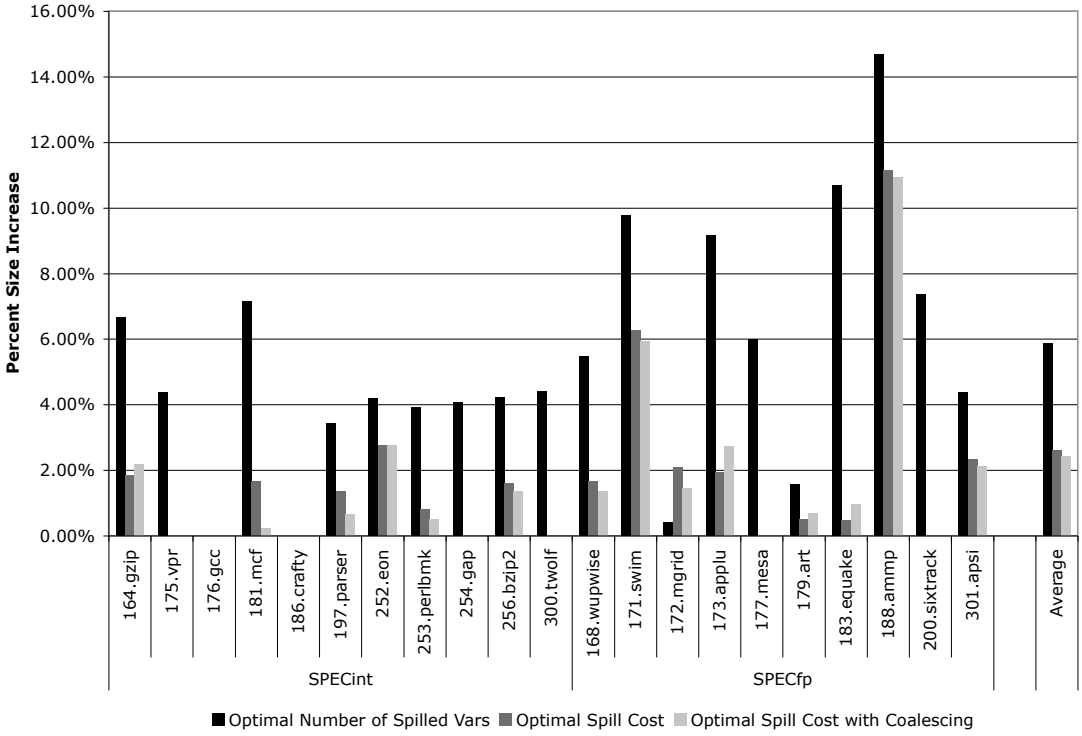

Figure 4: Code size increase (as determined by measuring the size of the `.text` section) relative to code produced using the `gcc` graph allocator. The average code size increase when optimizing the number of spilled variables, the cost of spilling, and the cost of spilling with coalescing is 5.9%, 2.6%, and 2.4% respectively.

graphs. As expected, the optimal coloring allocator that minimizes the number of spilled variables outperforms the other allocators in this comparison. However, as we will see, reducing the number of spilled variables does not necessarily result in a good allocation. As seen in Figure 2, the remaining optimal coloring allocators, which minimize estimated spill cost, are slightly better than the heuristic allocator at reducing the number of spills.

Although an optimal coloring minimizes the number of spills, this does not directly correspond to a better register allocation, either when optimizing for speed (Figure 3) or code size (Figure 4). These results clearly indicate the importance of incorporating spill cost information into the graph coloring algorithm. Interestingly, the optimal coloring algorithms, even when optimizing for the same spill cost as the heuristic algorithm and incorporating coalescing, do not, in most cases, perform better than the heuristic algorithm. The success of the heuristic algorithm is underscored by the increase in code size seen with the optimal coloring algorithms since the code size metric better reflects the compiler's ability to optimize the whole program as opposed to just the most frequently executed portions.

6 Discussion

The coloring heuristic used by the allocator correctly determines the colorability of typical interference graphs 99.9% of the time. There is clearly no benefit in incorporating a more sophisticated coloring algorithm into the allocator. In fact, if the coloring algorithm is not modified to incorporate the spill costs of uncolored nodes, the result is a decidedly poor allocation.

Elements other than coloring, such as spill cost estimation, clearly play an important roll in

determining the quality of the register allocation. It is not clear that the existing heuristics for determining what and where to spill have the same efficacy as the coloring heuristics. Although the optimal spill cost coloring algorithms sometimes perform better than the heuristics, the heuristic solution often significantly outperforms the supposedly optimal solution. This is a good indication that there are elements of the register allocation problem that the graph coloring model does not incorporate.

For example, the coloring model minimizes the heuristic spill cost of the uncolored nodes. This does not directly translate into minimizing the cost of spilling in the final allocation because when spill code is inserted, the underlying interference graph is changed. In a pathologically bad example, as occurs in the frequently executed `try_swap` function of the `175.vpr` benchmark, at each iteration of the allocation algorithm the coloring pass chooses to spill a single, low cost variable which, when spilled, requires more variables be spilled. The end result is that the allocator takes many iterations and spills many variables when spilling a few higher cost variables would have resulted in a better allocation. Neither the optimal nor heuristic coloring algorithms model the effect of spill code generation. However, in this example simply biasing the coloring towards coalescing coincidentally removes this pathologically bad behavior. Furthermore, when choosing variables to spill, the heuristic coloring algorithm not only considers the the spill cost of the variable, but also prefers variables which conflict with many other variables as spilling these variables will likely make the resulting graph easier to color.

A graph coloring allocator explicitly models the interference element of the register allocation problem and is successful at solving this subproblem. However, graph coloring does not explicitly

model the additional elements of the allocation problem. Results from optimal register allocators that more precisely model the costs of register allocation [10, 15], but do not exhibit practical compile times, indicate there is a substantial gap between existing allocators and the theoretical optimal. Since this disparity is not due to the inability of the coloring algorithm, it most likely is the result of the failure of graph coloring allocators to explicitly model and optimize for additional elements of register allocation, such as spill code generation and placement.

Extensions to the graph coloring model that increase its expressiveness have been proposed [19] as well as other models of register allocation that are innately more expressive, such using integer linear programming [17, 3, 10, 15], partitioned boolean quadratic programming [18], and multi-commodity network flow [14]. Finding the right combination of model and solution technique to effectively close the gap between existing allocators and the theoretical optimal remains an open problem.

7 Conclusion

The title of this paper asks the somewhat provocative question, "What is wrong with graph coloring?" One answer is that there is nothing wrong with graph coloring register allocators. We have shown they do a great job of finding a good coloring of an interference graph even while minimizing spill costs. However, our investigation into the performance of register allocators has made it quite clear that there is more to getting a good allocation than coloring an interference graph. More expressive models than simple graph coloring, combined with natural and efficient solution techniques, are needed to fully solve the register allocation problem.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Princiles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Andrew W. Appel. *Modern Compiler Implementation in Java: Basic Techniques*. Cambridge University Press, 1997.

[3] Andrew W. Appel and Lal George. Optimal spilling for cisc machines with few registers. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 243–253. ACM Press, 2001.

[4] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew T. O'Keefe. Spill code minimization via interference region spilling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 287–295, 1997.

[5] D. Bernstein, M. Golumbic, y. Mansour, R. Pinter, D. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compliers. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 258–263. ACM Press, 1989.

[6] Preston Briggs. *Register allocation via graph coloring.* PhD thesis, Rice University, Houston, TX, USA, 1992.

[7] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, 1994.

[8] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–101. ACM Press, 1982.

[9] Keith D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. In *Proceedings of the 1998 International Compiler Construction Converence*, 1998.

[10] Changqing Fu, Kent Wilken, and David Goodwin. A faster optimal register allocator. *The Journal of Instruction-Level Parallelism*, 7:1–31, January 2005.

[11] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996.

[12] ILOG CPLEX. `http://www.ilog.com/products/cplex`.

[13] Mark S. Johnson and Terrence C. Miller. Effectiveness of a machine-level, global optimizer. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler contruction*, pages 99–108, New York, NY, USA, 1986. ACM Press.

[14] David Koes and Seth Copen Goldstein. A progressive register allocator for irregular architectures. In *CGO '05: Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)*, pages 269–280, Washington, DC, USA, 2005. IEEE Computer Society.

[15] Timothy Kong and Kent D. Wilken. Precise register allocation for irregular architectures. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 297–307. IEEE Computer Society Press, 1998.

[16] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[17] Mayur Naik and Jens Palsberg. Compiling with code-size constraints. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 120–129. ACM Press, 2002.

[18] Bernhard Scholz and Erik Eckstein. Register allocation for irregular architectures. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 139–148. ACM Press, 2002.

[19] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. *SIGPLAN Not.*, 39(6):277–288, 2004.

[20] Standard Performance Evaluation Corp. *SPEC CPU2000 Benchmark Suite*, 2000.