An Analysis of Graph Coloring Register Allocation

David Koes Seth Copen Goldstein

March 2006

CMU-CS-06-111

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Graph coloring is the de facto standard technique for register allocation within a compiler. In this paper we examine the importance of the quality of the coloring algorithm and various extensions of the basic graph coloring technique by replacing the coloring phase of the GNU compiler's register allocator with an optimal coloring algorithm. We then extend this optimal algorithm to incorporate various extensions such as coalescing and preferential register assignment. We find that using an optimal coloring algorithm has surprisingly little benefit and empirically demonstrate the benefit of the various extensions.

This research was sponsored in part by the National Science Foundation under grant CCR-0205523 and in part by the Defense Advanced Research Project Agency (DARPA) under contracts N000140110659 01PR07586-00 and MDA972-01-3-0005. Any opinions, findings and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect those of the sponsors.

Keywords: compilers, 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, 17, 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 the performance of the coloring algorithm. However, experienced practitioners know that, especially for architectures with register usage constraints, the coloring algorithm has only a minor impact on the quality of the register allocation.

Our goal is to verify and explicate the knowledge practitioners have gained from experience with our experimental methodology. We replace the existing heuristic coloring algorithm in `gcc` 3.4.4 with an optimal coloring algorithm. We find that not only does the heuristic coloring algorithm usually find an optimal coloring, but the optimal coloring algorithm performs poorly as it lacks extensions to the pure graph coloring model specific to register allocation. We document the effect extensions such as spill cost heuristics, move coalescing, and register assignment preferences have on code quality by incrementally adding them to our optimal solver. We find, for example, that adding move coalescing has a relatively small effect compared to heuristically improving spill decisions or preferentially allocating certain registers.

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. Each 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 which 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 none of the remaining nodes can be simplified. If all nodes have been pushed on the stack we skip to the Select phase.

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 in the hope that we might be able to assign it a color. 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 assign colors to nodes in the order we pop them off the stack. If the node was not marked as a potential spill node then there

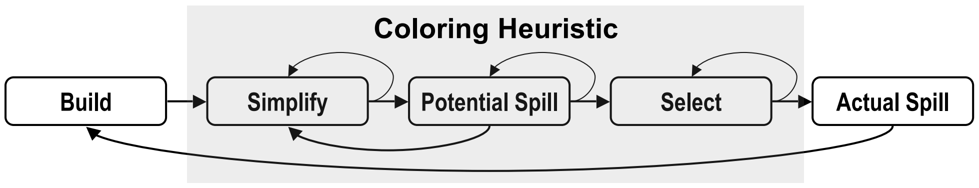

Figure 1: The flow of a traditional graph coloring algorithm.

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 [14, 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 i.

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 is 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) 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 order to investigate the effect of the quality of the coloring algorithm we replace 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.

3.1 Algorithm

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.

Adding a cost to uncolored nodes is simple to incorporate in 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 the objective function:

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

where the coefficient c_n is the cost of leaving the node uncolored. We simply set c_n to 1.0 to minimize the total number of nodes spilled.

3.2 Improvements

There are a number of register allocation specific extensions to the basic graph coloring model we can add to our optimal solver so that the resulting coloring will produce a superior register allocation.

Spill Cost We modify the objective function so that c_n, the coefficient of n_{spill}, is the same as the 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). This may result in more generated spills, but they will be less costly.

Coalescing 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.

Ordered Assignment When selecting colors for a node in the Select stage the heuristic coloring algorithm assigns registers to variables in a certain order. For example, it might assign caller-save registers before callee-save registers in order to avoid having to generate save and restore code for these registers. The optimal coloring algorithm can mimic this behavior by assigning a small cost to a less desirable assignment. As long as this cost is sufficiently small relative to the minimum spill cost this modification will only change the assignment of variables to registers; it will not change the spill decisions.

Preferential Assignment In addition to assigning registers in a particular order, some variables may prefer to be allocated to specific registers, regardless of the regular order of assignment. For example, the results of the x86 multiply and divide instructions must be allocated to the `eax` and `edx` registers. A variable that represents the result of such an instruction would then prefer to be allocated to one of these registers. The optimal allocator models preferences by assigning a small cost to registers which are valid, but not preferred.

4 Evaluation

We evaluate the effect of using the optimal coloring algorithm with its various extensions by substituting it for the `ra-colorize` function of the graph coloring based allocator of `gcc` version 3.4.4. 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 [13] to solve the ILPs.

We use the metric of code size when evaluating the quality of a register allocation. By using the code size metric we can accurately evaluate the code quality of the entire program, not just the most frequently executed portions. We use the SPECint200 and SPECfp2000 benchmark suites [21] with the reference input sets for evaluation. We omit benchmarks that `gcc` was unable to compile or generate correct code for (176.gcc, 186.crafty, 255.vortex, and the Fortran 90 benchmarks 178.galgel, 187.facerec, 189.lucas, and 191.fma3d). Unless otherwise stated, we compile with `-Os`.

We evaluate the allocators using the x86 architecture; this architecture, with its limited register file and register usage constraints, will likely see the biggest impact from the performance of the register allocator. Our test machine a 2.8 Ghz Pentium 4 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 graphs. As expected, the optimal coloring allocator that minimizes the number of spilled variables, but does not seek to minimize the cost of the spills, outperforms the other allocators in this comparison.

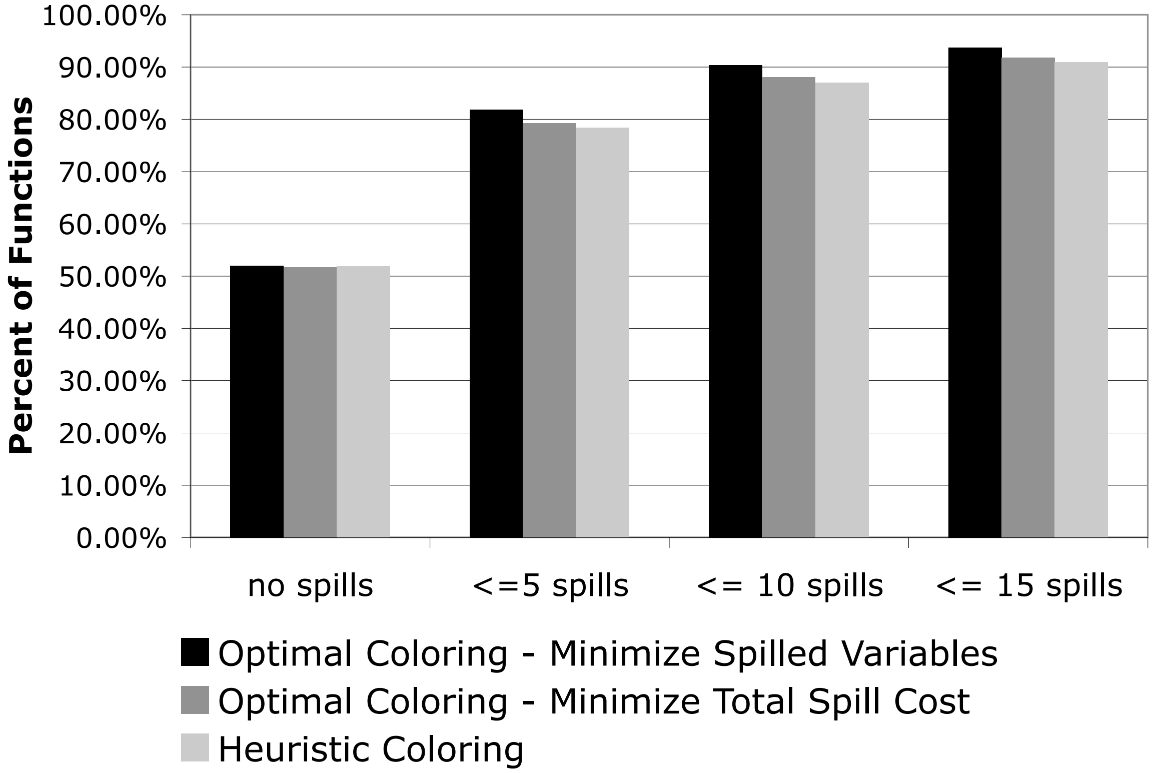

Figure 2: The percent of functions in the SPEC benchmark suite that spill at most a given number of variables. The benchmarks were compiled with `-O3 -funroll-loops`.

Although an optimal coloring minimizes the number of spills, this does not directly correspond to a better register allocation as indicated by Figure 3. The naive approach of just optimizing the number of variables spilled produces poor code relative to `gcc`'s heuristic based coloring allocator with an overall size improvement of -4.34%. Incorporating spill cost information improves the overall size to -2.54%. Somewhat surprisingly, the addition of move coalescing has a relatively small effect. Although it improves code size by as much as .95% on some benchmarks, overall coalescing only results in an additional .18% improvement.

Extending the optimal coloring algorithm to incorporate a register assignment ordering results in a surprising improvement of 1.37%. In addition, 5 of the 19 benchmarks compile to smaller or equal code than with `gcc`'s allocator. Although some of this gain comes from allocating caller-save registers before callee-save registers, the majority of the gain actually comes from a peculiarity of the x86 architecture. In the x86 architecture, several instructions, including the move instruction, use a smaller opcode if one of the operands is in `eax`. Since in the preferred order of allocation the first register to be allocated is `eax`, imposing an assignment ordering on the coloring algorithm results in many more variables allocated to this small-code inducing register. There is a similar effect with floating pointer variables and the `st(0)` register. Not to surprisingly, further biasing register assignments towards efficient register usage results in a further improvement of .42% with 10 of the 19 benchmarks compiling to smaller or equally sized code.

We do not include execution time results since, with the exception of the most naive optimal coloring algorithm, the changes in performance are mixed and minor. Using the most naive optimal coloring algorithm, four benchmakrs (254.gap, 171.swim, 173.applu, and 200.sixtrack) have a greater than 5% increase

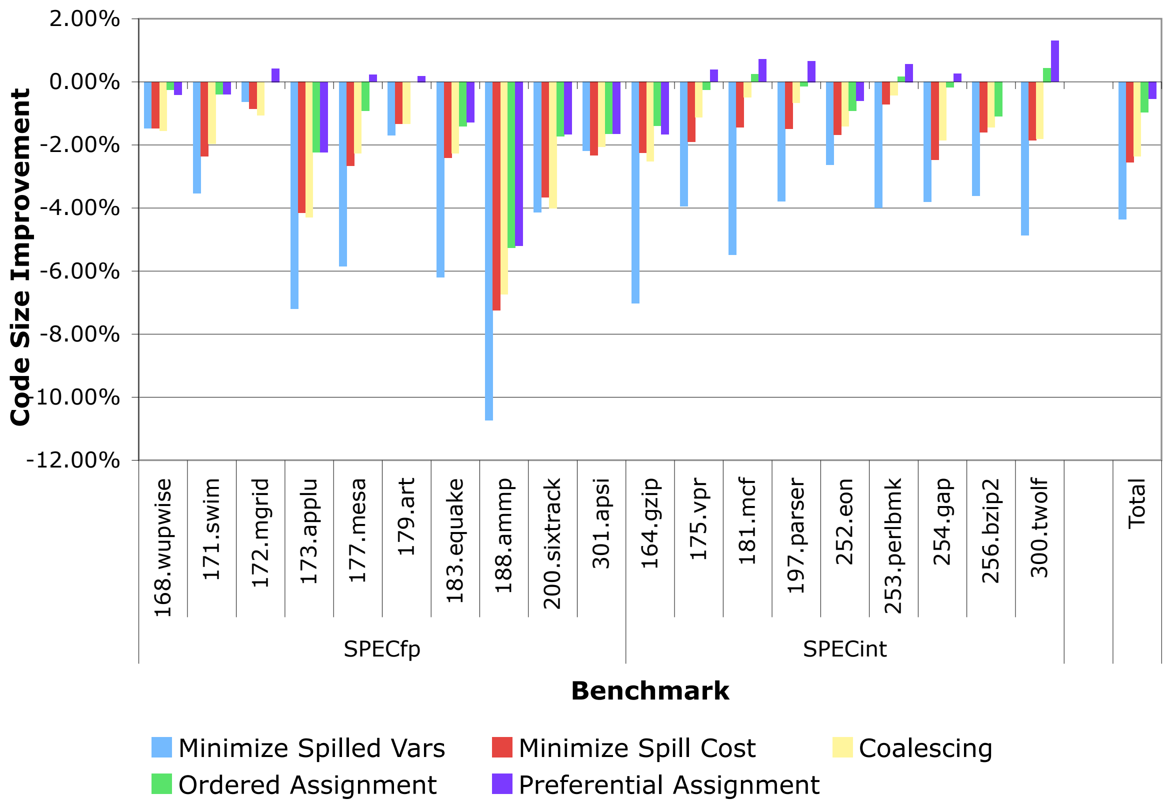

Figure 3: Code size improvement (as determined by measuring the size of the `.text` section) relative to code produced using the `gcc` graph allocator.

in execution time over the default heuristic allocator with 200.sixtrack showing an increase in execution time of 27%.

Although the optimal coloring algorithm generates better code for many of the benchmarks, it is clear from the remaining benchmarks that there are probably additional extensions to the model that the `gcc` allocator has implemented but the optimal coloring algorithm is not taking into account.

6 Discussion

The coloring heuristic used by the allocator correctly determines the colorability of typical interference graphs 99.9% of the time. There is little benefit in incorporating a more sophisticated coloring algorithm into the allocator. In fact, if the coloring algorithm is not modified to incorporate other aspects of register allocation, the result is a decidedly poor allocation.

Surprisingly, even when other aspects of register allocation are incorporated into the model, the optimal coloring algorithm does not always outperform the heuristic algorithm. Investigating the cases where the optimal coloring algorithm is outperformed reveals that the primary cause of the regressions is due to the strict hierarchy of register allocation extensions. For example, a variable will never be allocated to its preferred register class if doing so would result in more expensive spills. In some cases, the cost of not assigning a variable to its preferred register class is more expensive then the cost of the avoided spills. A more realistic cost model for register preferences would likely eliminate many of the regressions.

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. Instead, these elements are tacked onto the graph coloring problem by modifying the spill and register assignment heuristics. In contrast, consider `gcc`'s default allocator, which is a local/global, heuristic-driven, non-iterative allocator. The heuristics of this allocator are heavily tuned to optimize for the register usage constraints of the target architecture. Thus, even though this allocator does not have some of the more advanced features of a traditional graph allocator, the default allocator generally outperforms the graph allocator, producing code that is 2.13% smaller overall.

Furthermore, results from optimal register allocators that more precisely model the costs of register allocation [10, 16, 12], 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 is most likely a 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 [20] as well as other models of register allocation that are innately more expressive, such using integer linear programming [18, 3, 10, 16], partitioned boolean quadratic programming [19], and multi-commodity network flow [15]. 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

Our investigation into the performance of register allocators has shown that obtaining a good allocation requires more than coloring an interference graph. Using an optimal coloring algorithm we have shown the precise contribution of various extensions to the basic graph coloring model. This data indicates that 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] Ulrich Hirnschrott, Andreas Krall, and Bernhard Scholz. Graph coloring vs. optimal register allocation for optimizing compilers. In *JMLC*, pages 202–213, 2003.

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

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

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