Graph Coloring vs. Optimal Register Allocation for Optimizing Compilers*

Ulrich Hirnschrott, Andreas Krall, and Bernhard Scholz

Institut für Computersprachen, Technische Universität Wien
Argentinierstraße 8, A-1040 Wien, Austria
{uli,andi,scholz}@complang.tuwien.ac.at

Abstract. Optimizing compilers play an important role for the efficient execution of programs written in high level programming languages. Current microprocessors impose the problem that the gap between processor cycle time and memory latency increases. In order to fully exploit the potential of processors, nearly optimal register allocation is of paramount importance. In the predominance of the x86 architecture and in the increased usage of high-level programming languages for embedded systems peculiarities and irregularities in their register sets have to be handled. These irregularities makes the task of register allocation for optimizing compilers more difficult than for regular architectures and register files. In this article we show how optimistic graph coloring register allocation can be extended to handle these irregularities. Additionally we present an exponential algorithm which in most cases can compute an optimal solution for register allocation and copy elimination. These algorithms are evaluated on a virtual processor architecture modeling two and three operand architectures with different register file sizes. The evaluation demonstrates that the heuristic graph coloring register allocator comes close to the optimal solution for large register files, but performs badly on small register files. For small register files the optimal algorithm is fast enough to replace a heuristic algorithm.

1 Introduction and Motivation

Nowadays high level programming languages dominate the software development for all kinds of applications on a broad range of processor architectures. For the efficient execution of programs optimizing compilers are required whereby one of the most important components in the compiler is register allocation. Register allocation maps the program variables to CPU registers. The objective of an allocator is to assign all variables to CPU registers. However, often the number of registers is not sufficient and some variables have to be stored in slow memory (also known as *spilling*).

Traditionally, register allocation is solved by employing graph coloring that is a well-known NP-complete problem. Heuristics are used to find good register allocation in nearly linear time [4,3].

* This research is partially supported by the Christian Doppler Forschungsgesellschaft as part of the Project "Compilation Techniques for Embedded Processors"

L. Böszörményi and P. Schojer (Eds.): JMLC 2003, LNCS 2789, pp. 202–213, 2003.

For regular architectures with large register sets graph coloring register allocation is well explored. In general the heuristic algorithms produce very good results and register allocations come close to the optimal solution [6]. With small and irregular register sets register allocation by graph coloring has to be adapted.

A prominent example of an irregular architecture with a small number of registers is the x86 which has overlapping registers and special register usage requirements. Another example is the Motorola 68k architecture which has separate register files for data and addresses. Both architectures have two-operand instructions whereas more modern RISC architectures provide three-operands in the instruction set. Intel's IA-64 processors feature VLIW (Very-Large Instruction Word), predicated execution, and rotating register files. Furthermore, embedded processors and digital signal processors (DSP) have non-orthogonal instruction sets and impose additional constraints for register allocation.

2 Processor Models

2.1 General Description

Our processor is a 5 way variable length VLIW load/store architecture. It supports some commonly met extensions for the DSP domain, like multiply accumulate instructions, various addressing modes for loads and stores, fixed point arithmetic, predicated execution, SIMD instructions, etc. The processor's register file consists of two distinct register banks, one bank for data registers, the other one for address registers. Each data register is 40 bit wide, but can also be used as a 32 bit register, or as two registers of 16 bit width ("shared registers", "overlapping registers", "register pairs"). Address registers are 16 bit wide.

The register file layout described above causes several idiosyncrasies in the instruction set. Not all of the instructions may use both registers types. E.g., a multiplication may not use address registers, or load and stores can get their addresses only from address registers (hence the name). Only simple integer arithmetic is possible with address registers. Some instructions require their operands to be assigned to adjacent register pairs (SIMD).

2.2 Model Parameters

The first parameter is the number of registers per bank. We modeled 4, 8, and 16 registers per bank, respectively. This covers many of todays popular processors in the embedded domain. See Fig. 1 for the register model.

The second parameter is the number of operands per instruction. The quasi–standard mode is three operand mode, meaning that instructions can address up to 3 different registers (two source registers and one destination register in case of a binary operation). Two operand mode only allows to address two registers per instruction. Binary operations therefore require one of the source operands being the same as the destination operand. This can easily be achieved by inserting additional register move instructions prior to binary operations.

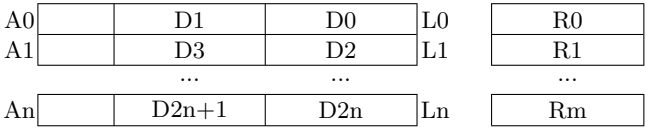

Fig. 1. Register file for model 8R/xA

3 Extension to Graph Coloring

Chaitin [4] was the first who used a graph coloring approach for register allocation. Many of the ideas and phases of graph coloring register allocation introduced by him are used in current register allocators. It is assumed that an unlimited number of symbolic registers is available to store local variables and large constants. During register allocation these symbolic registers are mapped to a finite number of machine registers. If there are not sufficient machine registers some of the symbolic registers are stored in the memory (spilled registers). Liveness analysis determines the live ranges of symbolic registers. Symbolic registers which are live at the same time cannot be assigned to the same machine register. These assignment constraints are stored in an interference graph. The nodes of this graph represent the symbolic registers. An edge connects two nodes, if these two symbolic registers are live at the same time. After liveness has been computed live ranges are combined when they are connected by a copy instruction (coalescing). Graph coloring register allocation colors the interference graph with machine registers spilling some symbolic registers to memory. The register allocator has to respect constraints like certain register requirements or pairing of registers.

3.1 Irregular Processors

Graph coloring register allocation becomes complicated when processors have nonorthogonal register sets or restrictions. For example paired registers form registers of twice the size. There are instructions which can only use a certain register or a subset of registers. Two operand instructions require that the destination register of an instruction is the same as one of the source registers.

Briggs [2] solved the problem of paired registers using a multigraph. A paired register has two edges to another register and a single register has a single edge to another single register. Such a scheme models paired registers without changing the other parts of the algorithm or the heuristics. The colorability of a graph can be determined by the degree of a node as before. Smith and Holloway suggested using a weighted interference graph and we followed their approach [12].

When using a weighted interference graph the nodes are augmented with a weight which corresponds to the occupied register resources. The edges represent the program constraints whereas the weights represent the architectural constraints. In a weighted graph minor changes to the algorithm are necessary. The colorability of a node cannot be determined by the degree anymore. Since

we are using optimistic coloring where spill decisions are determined in the select phase exact computation of the colorability is not necessary. Because we are already using inaccurate heuristics, we should use exact computations when possible.

We are weighting short registers with the weight 1 and long registers or accumulators with weight 2. The colorability equation has to take the weight of the node and of its neighbors into account. Equation 1 gives the details of the computation.

$$d_n = \sum_{j \in adj(n)} \lceil \frac{w_j}{w_n} \rceil * w_n \tag{1}$$

d_n is the virtual degree of the weighted node n. If d_n is smaller than N, it is guaranteed that the node can be colored (N is the number of CPU registers multiplied with the applied weight for n's register type).

The assignment of certain kinds of values maybe restricted to a predefined set of registers. This is achieved by precoloring these registers to the required machine registers [5]. To ensure that coloring a graph with precolored registers is possible we add move instructions between the symbolic register and the machine register. If possible these move instructions are eliminated during coalescing. A typical example are argument registers. At the entrance of a function the incoming argument registers are copied to symbolic registers and before a function invocation symbolic registers are copied to the outgoing arguments registers. Similarly before the end of a function the symbolic register is copied to the return value register. Machine instructions like a multiply-accumulate with a fixed accumulator register also get this operand precolored and move instructions are added to ensure colorability.

Some values have to stay in a subset of the machine registers. Examples are symbolic registers which are live across a function call or symbolic registers which are operands of machine instructions which only accept a subset of the register set. Additionally to the symbolic registers the interference graph contains all machine registers. If only a subset of the registers is allowed, interference edges are added to the complement set of the allowed machine registers. Symbolic registers which are live across a function call are in conflict with all machine registers except callee saved registers. As a further optimization these registers are saved in caller saved registers instead of memory when caller saved registers are available and the cost of spilling to memory is higher.

3.2 Coalescing

If source and destination of a move instruction do not interfere, the move can be eliminated and its operands can be combined into one live range. The interferences of the new live range are the union of those of the live ranges being combined. Coalescing of live ranges can prevent coalescing of other live ranges. Therefore coalescing order is important. Move instructions with higher execution frequencies should be eliminated first.

Aggressive coalescing combines any of non interfering copy related live ranges. This reckless strategy may turn a k–colorable graph into a not k–colorable graph and thus is unsafe. Briggs suggests in [1] only to combine live ranges that result in a live range that has fewer than k neighbors of degree k or higher. George and Appel suggest in [7] to combine a and b only if for every neighbor t of a, either t already interferes with b or the degree of t is less than k. Additionally, they propose interleaving coalescing and graph simplification. Both strategies are safe, but the iterative approach of George and Appel achieves better coloring results.

If one of the copy related live ranges is precolored, non interference is not a sufficient prerequisite to do the combination. Suppose a live range a that is precolored to R and a copy related live range b. a and b can only be coalesced if neither of b's neighbors is precolored to R. Further constraints on coalescing arise from the irregular architecture. It is only possible to combine live ranges of equal types. Interbank moves (between data and address bank) cannot be coalesced at all. Combining live ranges of shorts can be problematic, too. Any of them can be part of a paired register. They can only be combined, if there are no such constraints, or if they have equal constraints. If only one is constrained, the constraint must be propagated to the resulting live range.

Two operand mode requires to copy one source operand of a binary operation to the destination operand. In case of commutative operations, choosing a source operand that does not interfere with the destination should be favored. The inserted move will later be eliminated.

3.3 Complete Algorithm

The complete algorithm executes the single phases linearly and does an iteration if spilling is necessary. The single phases are liveness analysis with interference graph construction, coalescing, interference graph construction and coloring. When adding spill instructions care has to be taken to avoid endless looping because of spilling live ranges which occur due to loading spilled registers.

4 Optimal Register Allocation

Optimal register allocation delivers the most accurate register allocation. By its nature register allocation is a very hard problem to solve and an optimal solution requires exponential run-time for its computation. Therefore, production compilers sacrifice optimality in order to obtain a fast register allocation. Heuristics are applied which give sub-optimal answers.

However, an optimal register allocation scheme is relevant for assessing heuristics how good they work in practice. When traditional graph-coloring heuristics [3], which have an excellent performance for RISC architectures with a reasonable number of registers, are extended for architectural peculiarities, the quality of the register allocation might suffer and the comparison with the optimal solution is essential.

Instead of using an *Integer Linear Programming(ILP)* [8,9,6] approach for obtaining an optimal solution, we employed a new algorithm [11] that is based

on *Partitioned Boolean Quadratic Programming(PBQP)*. The PBQP approach is a unified approach for register allocation that can model a wide range of peculiarities and supersedes traditional extended graph-coloring approaches [3, 12]. In addition coalescing is an integral part of the register assignment which is necessary for achieving good allocations.

The PBQP approach uses cost functions for register assignment decisions. The cost functions have to fulfill two tasks: (1) cost functions that express mathematically the model of cost of the architecture, and (2) cost functions that describe interference constraints, coalescing benefits, and constraints which stem from the CPU architecture. Basically, we have two classes of cost functions. One class of cost functions model the costs and constraints involved for one symbolic register, and the second class of cost functions model the costs and constraints of two dependent symbolic registers. For our processor we only need a sub set of the cost functions that were introduced in [11] since its architecture is fairly orthogonal.

In Table 1 the cost functions for our processor architecture are listed. Spilling is modeled by a cost function for one symbolic register. The parameter c gives the costs for spilling and parameter a determines the allocation of symbolic register $\mathtt{s}$. A symbolic register is either spilled (i.e. a is equal to sp) or a register is assigned to it (i.e. $a \in \{R_0, R_1, \dots\}$). Depending on this decision different costs are involved. The architecture features four register classes which can be modeled by $c_{\mathtt{s}}(a)$. Registers which are disabled for a symbolic register have ∞ costs and therefore are excluded for register assignments. An interference of two symbolic registers $\mathtt{s}_1$ and $\mathtt{s}_2$ is given by $i_{\mathtt{s}_1\mathtt{s}_2}(a_1, a_2)$. Either both allocations have different register assignments ($a_1 \neq a_2$) or one of the registers is spilled ($a_1 = sp$). Again, infinite costs will be raised if for both symbolic register the same CPU register is allocated. The shared register interference constraint is given in equation for $d_{\mathtt{s}_1\,\mathtt{s}_2}(a_1, a_2)$. This constraint is necessary since two short registers share the same memory of one long register in the architecture. Coalescing costs of two symbolic registers are given by $p^{(b)}_{\mathtt{s}_1\,\mathtt{s}_2}(a_1, a_2)$. If both register assignments of $\mathtt{s}_1$ and $\mathtt{s}_2$ are identical we obtain a coalescing benefit expressed as a negative number $-b$.

The cost functions are used for constructing cost matrices and cost vectors for the PBQP problem. The NP-hard PBQP problem is given as follows:

$$\min f = \left[\sum_{1 \le i < j \le n} \boldsymbol{x}_i \cdot C_{ij} \cdot \boldsymbol{x}_j{}^T\right] + \left[\sum_{1 \le i \le n} \boldsymbol{c}_i \cdot \boldsymbol{x}_i{}^T\right]$$
$$\text{subject to: } \forall i \in 1 \dots n : \boldsymbol{x}_i \cdot \mathbf{1}^T = 1$$

The cost matrices C_{ij} are determined by the cost functions $F_{\mathtt{s}_i,\mathtt{s}_j}$ of symbolic registers $\mathtt{s}_i$ and $\mathtt{s}_j$ as follows:

$$\forall a_k, a_l \in A : C_{ij}(k,l) = \sum_{f_{\mathtt{s}_i\,\mathtt{s}_j} \in F_{\mathtt{s}_i\,\mathtt{s}_j}} f_{\mathtt{s}_i\,\mathtt{s}_j}(a_k, a_l)$$

Table 1. Cost functions for our processor

Spilling:

$$s^{(c)}_{\mathbf{s}}(a) = \begin{cases} c, & \text{if } a = sp \\ 0, & \text{otherwise} \end{cases}$$

Class Constraint:

$$c_{\mathbf{s}}(a) = \begin{cases} 0, & \text{if } a \in \mathit{class}(\mathbf{s}) \cup \{sp\}, \\ \infty, & \text{otherwise} \end{cases}$$

Interference:

$$i_{\mathbf{s}_1\mathbf{s}_2}(a_1, a_2) = \begin{cases} 0, & \text{if } a_1 \neq a_2 \vee a_1 = sp \\ \infty, & \text{otherwise} \end{cases}$$

Shared Register(Interference)

$$d_{\mathbf{s}_1\,\mathbf{s}_2}(a_1, a_2) = \begin{cases} \infty, & \text{if } a_1 \in \mathit{shared}(a_2) \\ 0, & \text{otherwise} \end{cases}$$

Coalescing:

$$p^{(b)}_{\mathbf{s}_1\,\mathbf{s}_2}(a_1, a_2) = \begin{cases} -b, & \text{if } a_1 = a_2 \wedge a_1 \neq sp \\ 0, & \text{otherwise} \end{cases}$$

The cost vectors $\boldsymbol{c}_i$ are determined by the cost function $f_{\mathbf{s}_i}$ of symbolic register $\mathbf{s}_i$.

$$\forall a_k \in A : \boldsymbol{c}_i(k) = \sum_{f_{\mathbf{s}_i} \in F_{\mathbf{s}_i}} f_{\mathbf{s}_i}(a_k)$$

The PBQP problem can be solved by dynamic programming as proposed in [11]. In each step of the algorithm a vector $\boldsymbol{x}_i$ is eliminated until the objective function f becomes trivial, i.e. the first part of the sum $\sum_{1 \leq i < j \leq n} \boldsymbol{x}_i \cdot C_{ij} \cdot \boldsymbol{x}_j{}^T$ vanishes. Then, the solution of remaining vectors in the objective function is determined. Reduced vectors can be computed by reconstructing the original objective function. Unfortunately, not all reductions can be applied in polynomial time. Therefore, a recursively enumeration is necessary for obtaining the optimal solution. Basically, we have three reduction: reduction RI for nodes of only one cost matrix, reduction RII for nodes of two cost matrices, and reduction RN for nodes with arbitrary number of cost matrices. Reductions RI and RII can be solved in polynomial time — reduction RN needs exponential time.

The RN reduction for the optimal solution is given in Fig. 2. The first loop enumerates all possible solutions of vector x. For a given solution the costs are determined. If it is smaller than the current minimum the solutions of the

```
1:  procedure ReduceN(x)
2:  begin
3:        min := ∞;
4:     for i:=1 to |c_x| do
5:        h := c_x(i);
6:        for all y ∈ adj(x) do
7:           c_y := c_y + C_x y(i, :);
9:        end
10:       remove x;
11:       for all scc ∈ G do
12:          solve scc;
13:          h := h+cost(scc);
14:       endfor
15:       if h < min then
16:          save solutions
17:       endif
18:       reconstruct node x
19:    endfor
20:    restore min. solution
21: end
```

Fig. 2. RN reduction

remaining vectors are saved. The reduction of the vector can split the PBQP graph in several independent sub-graphs (scc). The performance of the algorithm can be substantially improved by solving the independent sub-graphs on their own. For reducing the number of RN nodes it is a good heuristic to select the vector with the highest number of cost matrices.

5 Results

The intention of the experiments is to compare the implemented register allocation methods on a broad range of architectural models on a wide variety of programs. The complete evaluation is contained in a longer version of this article and available at `www.complang.tuwien.ac.at/papers/HiKrSch2003.ps`. We measured the obtained results both in terms of spilling costs and coalescing benefits, as well as solve times for coloring the interference graph.

Our experiments covered a total of 210 functions from typical DSP applications.

Since the optimal method has a worst case of $O(k^n)$ (with k ... number of registers , n ... number of nodes), we implemented a timeout. Each function is given 30 minutes to be solved. If no solution is found, the record is deleted from all the result sets, and only the remaining subsets of the qualitative records can be compared. Figure 3 shows the number of solved functions per model. The minimum is at model *16R/2O*, where 148 functions were solved (i.e. all

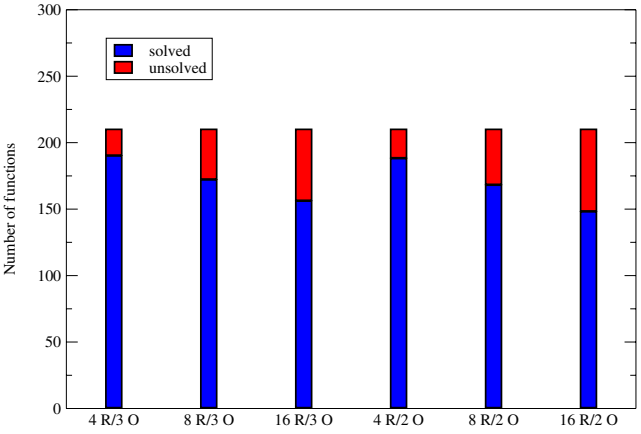

Fig. 3. Functions with optimal solution

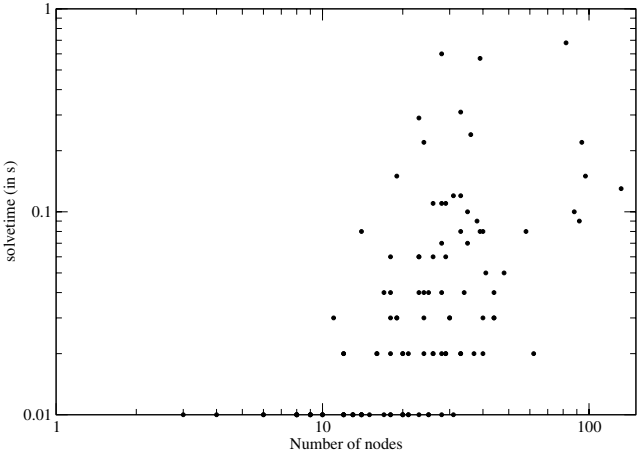

Fig. 4. Solve times for *4R/3O*

qualitative data refer to this subset of functions). The solve times for model *4R/3O* are presented in Fig. 4.

Key points of interest are spilling costs and coalescing benefits. We evaluated spilling costs and also counted the number of spilled live ranges. Optimal register allocation performs better in both terms. The gainings over graph coloring are best when the number of registers is small. The optimal allocator also achieves better coalescing benefits than graph coloring with aggressive coalescing. Be aware that the baseline of the coalescing benefits is not zero. It is known that aggressive coalescing may overly constrain live ranges, so that these cannot be colored and thus do not contribute to the coalescing benefit. See Figs. 5 and 6 for an overview.

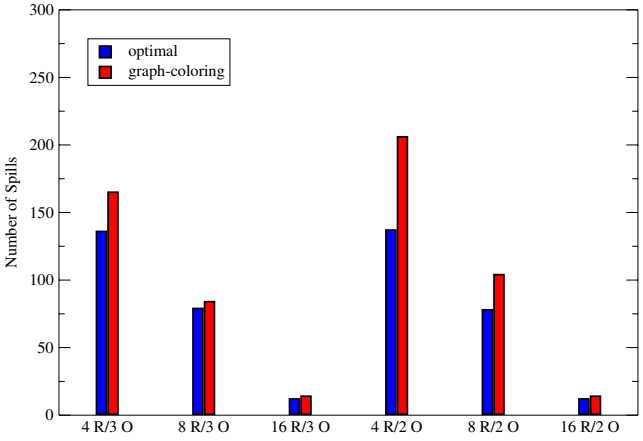

Fig. 5. Number of spills

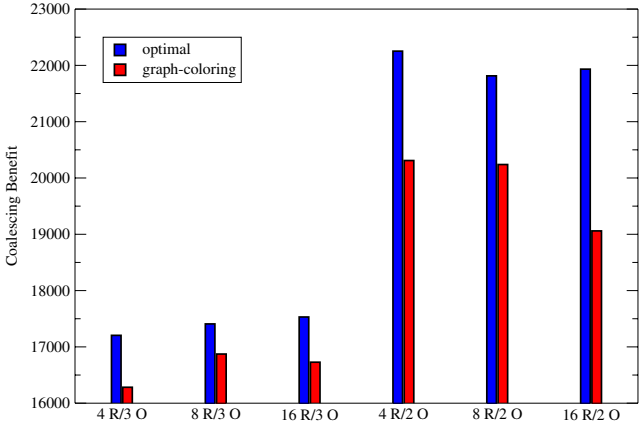

Fig. 6. Coalescing benefits

6 Related Work

Register allocation based on integer linear programming (ILP) was introduced by Goodwin and Wilken [8,6]. The approach maps the register allocation problem to an integer linear program which is solved by an NP-complete ILP-solver. The work was extended by Kong and Wilken [9] for irregular architectures. As an example they choose the IA-32 architecture and added additional features such as address mode selection. The approach can handle irregularities very nicely. However, the underlying algorithms have exponential solve time and the solvers are not able to solve bigger functions in reasonable time – they simple cut-off the solver.

Copy propagation is an important task for register allocators, which is achieved by assigning source and destination of a copy instruction the same registers. In the past several approaches as Chaitin's *aggressive coalescing*[4], Briggs' conservative coalescing [3], George and Appel's iterated coalescing[7], and Park and Moon's optimistic coalescing [10] were introduced. All have in common that they coalesce nodes of the interference graph in a separate pass. The node selection for coalescing and the strategy of uncolored and coalesced nodes differs depending on the approach.

7 Conclusions

In this work, we presented an extensive experimental evaluation of two different register allocation methods for irregular processor architectures. Our experiments show, that the graph coloring based algorithm causes more spilling costs than the algorithm with an optimal solution. Smaller register numbers result in an increase of this penalty. Further, aggressive coalescing does not result in higher coalescing benefits. It overly constrains some of the concerned live ranges and therefore forces additional spills.

Comparison purely by solve times makes graph coloring the winner. Most of the functions are colored in less time than we were able to measure, whereas PBQP is not even able to solve all the functions within a timeout of 30 minutes. This computation effort does not pay for architectures with a large register file, where graph coloring achieves near optimal results.

For architectures with a small register file, computation of the optimal solution runs acceptably fast, and the gainings over graph coloring are significant. In this case, it is practicable and feasible to chose the optimal method for most of the typical applications. A longer version of this article is available at `www.complang.tuwien.ac.at/papers/HiKrSch2003.ps`.

Acknowledgement. We like to thank the reviewers for their helpful suggestions.

References

1. K.D.C.P. Briggs, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. *SIGPLAN Notices*, 24(7):275–284, July 1989.
2. P. Briggs, K. D. Cooper, and L. Torczon. Coloring register pairs. *ACM Letters on Programming Languages and Systems, (LOPLAS)*, 1(1):3–13, Mar. 1992.
3. P. Briggs, K.D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428–455, May 1994.
4. G.J. Chaitin. Register allocation and spilling via graph coloring. *ACM SIGPLAN Notices*, 17(6):98–105, June 1982.
5. F.C. Chow and J.L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, Oct. 1990.

6. C. Fu and K.D. Wilken. A faster optimal register allocator. In *Proceedings of the 35st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-02)*, pages 245–256, Istanbul, Nov. 18–22 2002. IEEE Computer Society.
7. L. George and A.W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.
8. D.W. Goodwin and K.D. Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. *Software & Practice and Experience*, 26(8):929–965, Aug. 1996.
9. T. Kong and K.D. Wilken. Precise register allocation for irregular register architectures. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, pages 297–307, Los Alamitos, Nov. 30–Dec. 2 1998. IEEE Computer Society.
10. J. Park and S.-M. Moon. Optimistic register coalescing. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 196–204, Paris, France, Oct. 12–18, 1998. IEEE Computer Society Press.
11. B. Scholz and E. Eckstein. Register allocation for irregular register architectures. In *Proceedings of the International Conference of Languages, Compilers and Tools for Embedded Systems (LCTES'02) and SCOPES'02*, Berlin, June 2002. ACM.
12. M. D. Smith and G. Holloway. Graph-coloring register allocation for irregular architectures. Technical report, Harvard University, 2000.