Register Allocation by Graph Coloring: A Review

Frank Mueller
Department of Computer Science
Florida State University
Tallahassee, Florida 32306-4019
phone: (904) 644-3441, *e-mail: mueller@cs.fsu.edu*

Fall 1992, 1st revision 3/26/93, 2nd revision 8/31/94

1 Introduction

The problem of register allocation for code generation can be described as follows: Given a set of registers, what is the most efficient mapping of registers to program variables in terms of execution time of the program. First, we define some terms to describe the problem more formally.

A variable is *defined* at a point in a program when a value is assigned to it. A variable is *used* at a point in a program when its value is referenced in an expression. A variable is said to be *live* at a point if it has been defined earlier and will be used later. The *live range* of a variable is the execution range (a set of adjacent vertices of the control-flow graph of the program) between a definition and uses of the variable. Two variables are *simultaneously live* if and only if both are live for the vertex v of the control-flow graph.

A proper *register allocation* is a mapping from the live ranges of the variables of a program into the available registers such that no register is assigned to variables that are live simultaneously.[1]

The *interference graph* of a program consists of a vertex for each variable and an edge between any two vertices if and only if the two variables associated with the vertices can be live simultaneously.

The problem of register allocation can then be described as the problem of finding a proper *vertex coloring* for the interference graph with no more than k colors, where k is the number of available registers.[2]

Unfortunately, it has been shown that there exists a corresponding program for any possible interference graph. In particular, the interference graph may have a chromatic number greater than k. If this is the case, the mapping from variables into registers becomes a partial mapping.

Two solutions have been proposed to deal with unmapped variables. Let $u1$ and $u2$ be such variables with a conflicting live range and let there be an available register. Then either $u1$ is mapped into main memory and will be accessed exclusively from there while $u2$ is mapped into a register or the live range of both variables has to be modified by introducing spill code to alternately store $u1$ and $u2$ in main memory when required. The former approach cannot be taken on load-store architectures (RISC) and may not result

[1] Chaitin points out that live ranges of variables that always contain the same value can be coalesced to one live range, so that they are later allocated to the same register. This avoids register copying by coalescing [7]. While coalescing is neglected in the definition of proper register allocation for now, it will be addressed in the algorithm for register coloring later on.

[2] A proper vertex coloring is defined such that no two adjacent vertices share the same color [9].

in the best possible performance if u is frequently used since access to main memory is considerably slower than access to registers. In the latter approach, $u1$ is spilled and $u2$ is loaded at definitions or uses of $u2$, and vice versa. The new live ranges of $u1$ and $u2$ will therefore alternate around the original conflicting live ranges.[3]

2 NP-Completeness

Consider the general problem of graph colorability:

Problem 1 (Graph k-Colorability) *Let $G(V, E)$ be a graph with a set of vertices E and a set of edges E. Furthermore, let there be a set of k colors. Is G k-colorable,* i.e. *is there a function $f : V(G) \rightarrow \{1, ..., k\}$ such that $f(u) \neq f(v)$ for $(u, v) \epsilon E(G)$?*

It has been shown by Karp [15] that this problem is NP-complete, *i.e.* solvable in polynomial time by a nondeterministic one-tape Turing machine.

Theorem 1 (Karp) *The problem of graph k-colorability is NP-complete.*

The proof reduces the NP-complete problem of *satisfiability*[4] at first to *satisfiability with at most three literals per clause*, then it reduces it further to the graph coloring problem as stated above.

Since only exponential-time algorithms are known to solve NP-complete problems (unless the conjecture $P = NP$ can be proven), greedy algorithms are commonly used to produce suboptimal solutions (also see [12]).

3 Register Coloring

The principle behind register allocation by graph coloring is based on decomposing the graph according to the following observation:

Theorem 2 (Chaitin [7]) *Let G be a graph and $v \epsilon V(G)$ such that $deg(v) < k$, where $deg(v)$ denotes the number of edges of vertex v. A graph G is k-colorable if and only if $G - v$ is k-colorable.*

This result can be used to decompose a graph by repeatedly deleting vertices with degree less than k until either the graph is empty or only vertices with degree greater than or equal to k are left. In the latter case, the graph cannot be colored. But for an interference graph, by modifying the live range corresponding to one of the remaining vertices, a new coloring attempt can be made. The register coloring algorithm suggested by Chaitin works as follows:

1. Perform variable live-analysis, *i.e.* construct the interference graph.
2. For any two vertices of interference graph, coalesce the vertices corresponding to the live ranges of variables if the variables contain the same value during these live ranges.
3. For all vertices of the interference graph with degree less than k, delete such vertices (and their edges) and push them onto a stack.

[3] Historically, Chaitin [7] talks about *spilling* virtual registers to reduce the interfence between variable live ranges, and Chow [10] talks about *splitting* live ranges of variables residing in memory by introducing loads and stores. We try to unify the different views. We talk about *splitting* live ranges by introducing *spill* code.

[4] Let U be a set of variables and C be a collection of clauses over U. Is there a satisfying truth assignment for C?

4. If the resulting graph is non-empty, split the live ranges of some variables by introducing spill code. Return to step 1.

5. For all vertices on the stack, pop one vertex at a time, add it back into the graph (rebuilding the original edges), and color it such that none of the adjacent vertices (up to this point) have the same color.

4 Clique Separators

To reduce the complexity of Chaitin's algorithm, it has been suggested to decompose the interference graph using clique separators before any coloring attempt is made. A *clique separator* is a complete subgraph whose removal disconnects the graph. The following result, which has been rephrased in terms of graph coloring, is the basis for the decomposition step.

Theorem 3 (Tarjan [17]) *Let G be a graph, C be a clique separator for G, and $G_1 ... G_n$ be the components of $G - C$. Then G is k-colorable if and only if the subgraphs $G_i + C$ $(1 \leq i \leq n)$ are k-colorable.*

Naively, a register coloring algorithm using clique separators would repeatedly split the interference graph into smaller graphs $G_i + C$ until no more clique separators can be found. It would then apply the above algorithm and possibly introduce spill code if any of the decomposed graphs still were not k-colorable. But due to the decomposition using clique separators, the subgraphs are more likely to be k-colorable such that less spill code may be introduced. Furthermore, the analysis can be constrained to the subgraphs and should therefore result in time and space savings.

Practically, choosing all clique separators makes register allocation too expensive. Thus, the choice is restricted to the set of clique separators such that any variable live range cannot occur more than c times within these separators. An algorithm for finding such a set for $c = 2$ is given in [14, 13].

5 Vertex Ordering by Degree

Bernstein *et al.* suggested a vertex ordering for the deletion in step 3 of Chaitin's algorithm [2]. They proposed to always delete a maximum degree vertex. This strategy may succeed in coloring a graph or subgraph for some cases where an arbitrary choice of a vertex for deletion, as proposed originally, might have failed. Consequently, costly spill code can be avoided in some cases. The modifications to Chaitin's algorithm are:

4. For all vertices in the interference graph with degree less than k, choose one vertex with maximum degree at a time, delete this vertex (and its edges), and push it onto a stack.

6 Delaying Spill Decisions

Briggs *et al.* [3] made a suggestion to improve on the choice of vertices by allowing the deletion of vertices with degree greater than k, which delays spill decisions. The modifications to steps of Chaitin's algorithm are:

4. If the resulting graph is non-empty, choose a vertex for potential spilling and remove it from the graph. Push it onto the stack. Return to step 3.

5. For all vertices on the stack, pop one vertex at a time, add it back into the graph (rebuilding the original edges), and color it such that non of the adjacent vertices (up to this point) have the same color. If a vertex cannot be colored, mark it.

6. For all marked vertices, split the live range of the corresponding variable and introduce spill code. Return to step 1.

Delaying the spill decision allows vertices with more than k neighbors to be colored if some of the neighbors have the same color and there is still a color not assigned to any neighbor. For example, consider the cycle graph [9] with four vertices C_4 and $k = 2$ colors (aka. diamond graph). Chaitin's algorithm cannot find vertices with degree less than k and therefore introduces spill code. If the coloring of vertices was attempted with the above algorithm, the first vertex is pushed in step 4, and all others in step 3 as before. When the vertices are popped in step 5, a 2-coloring of C_4 is found and no spill code has to be introduced. This method is guaranteed to find the same set or a subset of live ranges found by Chaitin's algorithm.

The reported savings for this method over Chaitin's approach were a reduction of spilled registers by 51% and a reduction of the spill cost by 22% while the asymptotic bounds for the algorithm remain unchanged. The heuristic function and the size of a unit are the same as in Chaitin's approach.

7 Heuristics

Step 4 in Chaitin's algorithm and step 6 in Briggs' algorithm both split the live ranges of some variables by introducing spill code. But the choice of a live range (*i.e.* the vertex) to be spilled may result in different costs in terms of execution time of a program. Intuitively, one would want to spill a variable that is used very infrequently such that expensive memory loads can be minimized. Several heuristics for making a "good" choice of a vertex to be spilled have been proposed, which result in near-optimal solutions. Such heuristics are typically expressed in terms of the savings if a live range was allocated a register. The live range with the smallest savings is chosen for spilling.

Some techniques are restricted to the scope of instructions, others to basic blocks. A *basic block* is a sequence of straight-line code such that only the first instruction may be the destination of a jump and only the last instruction may be a jump instruction.

7.1 Chaitin *et. al.*

The heuristic used by Chaitin *et al.* [8, 7] for a live range is stated as follows:

$$save(v) = \sum_{i=1}^{N}(defs_i + uses_i) * freq_i$$

where v is a vertex of the interference graph, N is the number of units within the live range corresponding to v, $defs$ and $uses$ are the number of definitions and uses in a unit, and $freq$ is the estimated execution frequency of a unit. The execution frequency is typically calculated as

$$freq_i = 10^{nest_i}$$

where *nest* is the loop nesting depth of the unit.

In their approach, a unit corresponds to a machine instruction. The graph coloring algorithm is applied to both global register allocation and local allocation (within basic blocks) at the same time. This approach for the PL.8 compiler is revised later on by Bernstein *et al.* [2].

7.2 Chow and Hennessy

Chow and Hennessy [10, 11] restricted their analysis to the unit of a basic block. Their refined cost function is:

$$save(v) = \frac{1}{N}\sum_{i=1}^{N}(loadsave * uses_i + storesave * defs_i - movecost * n_i) * freq_i$$

where $loadsave$ is the savings due to a reference, $storesave$ is the savings due to a definition of a variable in a register instead of memory, $movecost$ is the cost of moving a value between a register and memory, and $n_i \epsilon \{0, 1, 2\}$ is the number of loads and stores required to keep the value in the register within a basic block, which can be at most one load at the beginning and one store at the end of a block. But a load can be avoided if the value is in the register within all predecessor blocks, and a store can be avoided if the live range extends into all successors of the current block. Furthermore, a load/store may not always be necessary at the beginning/end of a live range.

The normalization by N reflects that a register allocated over a long live range consumes more register resources and should therefore be discouraged. The savings for a vertex may be zero or negative using this heuristic indicating that register allocation does not result in any savings at all and should be avoided for such a live range.

The graph coloring algorithm is applied to global register allocation after local allocation (within basic blocks) has taken place. Global allocation for a live range is therefore restricted to the registers not already used for local allocation within all basic blocks of the live range.

7.3 Gupta *et. al.*

The heuristic used by Gupta *et al.* [14] is similar to Chow's except that the savings are not normalized. It uses basic blocks as units as well and can be stated as follows:

$$save(v) = \sum_{i=1}^{N}(loadsave * uses_i + storesave * defs_i - movecost * n_i) * freq_i$$

No distinction between local and global allocation is made, *i.e.* all registers are allocated by graph coloring. The use of clique separators allows such an approach since it reduces the analysis to much smaller subgraphs of the interference graph.

7.4 Best-of-Three

Bernstein *et al.* [2] used three heuristics and chose the best one of them to decide on spills. All three functions $save_1, save_2, save_3$ are based on $save_0$, a simplified version of Chaitin's heuristic.

$$save_0(v) = \frac{\sum_{i=1}^{N} freq_i}{deg(v)}$$

$$save_1(v) = \frac{save_0(v)}{deg(v)} \quad save_2(v) = \frac{save_0(v)}{\sum_{i=1}^{N} lives_i * 5^{nest_i}} \quad save_3(v) = \frac{save_2(v)}{deg(v)}$$

where $lives$ is the number of live variables at the unit (instruction). $save_1$ increases the likelihood of spilling a register (vertex) with high degree to increase the probability that the graph becomes colorable afterwards. $save_2$ and $save_3$ are used to spill registers that have a long live range first. Again, local allocation precedes global allocation by coloring.

7.5 Tile Trees: Hierarchical Graph Coloring

Callahan and Koblenz [6] suggest to decompose the graph to be colored by using a tree of tiles. Tiles correspond to a part of a live range of a variable (basic blocks, conditionals, or loops) and are either properly nested or pairwise disjoint. They are constructed using interval analysis on the control-flow graph [1]. Register coloring is performed on tiles inside-out, if tiles are nested, or independently, if tiles are disjoint. Tile graphs are typically much smaller than interference graphs so that fewer vertices have to be analyzed during coloring.

The heuristics to select spill candidates are somewhat more complex:

$$
\begin{aligned}
save(v) &= local\_weight(v) + \textstyle\sum_s reg(s) - mem(s) \\
local\_weight(v) &= \textstyle\sum_b p_b * freq_b \\
reg(v) &= \begin{cases} min(transfer(v), save(v)) & \text{if allocated to a register} \\ 0 & \text{otherwise} \end{cases} \\
mem(v) &= \begin{cases} 0 & \text{if allocated to a register} \\ transfer(v) & \text{otherwise} \end{cases} \\
transfer(v) &= \textstyle\sum_e p_e * live(v,e) \\
live(v,e) &= \begin{cases} 1 & \text{if the variable corresponding to } v \text{ live along } e \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

where s are subtiles, b are basic blocks, e are entry or exit edges of a tile, p_i is the probability of a block b being executed, and p_e is the probability of following edge e. $local\_weight$ estimates the cost of allocating a register for the current tile while $transfer$ estimates the spill cost. A register will not be assigned if $save(v) < 0$ or $save(v) + transfer(v) < 0$, *i.e.* if register allocation does not result in any savings (optionally considering the spill cost).

8 Other Approaches

A recent effort by Proebsting and Fischer [16] returns to a register allocation scheme that does not employ graph coloring. Rather, a probabilistic approach to register allocation is taken. Local allocation is performed by calculating the probability p_i that a variable is allocated to a register for each instruction and for each live variable within a basic block, given k registers.

$$
\begin{aligned}
save_i &= p_i * freq_i \\
p_i &= \begin{cases} 1 & \text{if variable loaded at instruction } i \\ 1 & \text{if variable stored at instruction } i \\ 1 & \text{if variable calculated at instruction } i \\ 1 & \text{if } n \le k \\ k/n & \text{if } n > k \end{cases}
\end{aligned}
$$

where n is the number of live variables besides (at most) one variable that is defined, stored, or calculated at instruction i.

After local allocation, the savings heuristic for each unit (instruction) is used for global register allocation. The global allocation is performed inside-out with respect to loop nestings, followed by register assignment, which is done in reverse order.

Delaying register assignment until after allocation enables the consideration of different assignment combinations and their savings. Graph coloring, on the other hand, performs allocation and assignment at the same time. Nevertheless, the above heuristics could also be used to determine the savings for making spill decisions during graph coloring. The reported runtime behavior suggests though that the involved calculations might be too inefficient, such that the heuristics had to be simplified in order to make this approach competitive with other schemes.

Acknowledgements

I would like to thank Preston Briggs who pointed out several problems in this survey, which have been addressed in the first revision. In a personal communication, he suggested to use Chaitin's overall framework modified according to Briggs' delayed spill decisions, Bernstein's vertex ordering by degree, and Briggs' rematerialization [4, 5]. Furthermore, Peter E. Bergner pointed out that Chaitin's original approach did not separate local and global allocation. Since then, most researchers seem to prefer a local allocation phase that precedes global register allocation by coloring. Also, Ksheerabdhi Krishna pointed out several mistakes that have been corrected in the second revision.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] David Bernstein, Martin C. Golumbic, Yishay Mansour, and Ron Y. Pinter. Spill code minimization techniques for optimizing compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–263, June 1989.

[3] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 275–284, June 1989.

[4] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 311–321, June 1992.

[5] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, pages 428–455, May 1994.

[6] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 192–203, June 1991.

[7] Gregory J. Chaitin. Register allocation & spilling via graph coloring. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 98–105, 1982.

[8] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[9] G. Chartrand and L. Lesniak. *Graphs & Digraphs*. Wadsworth & Brooks, 2nd edition, 1986.

[10] Frederick Chow and John Hennessy. Register allocation by priority-based coloring. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 222–232, June 1984.

[11] Frederick Chow and John Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[12] Michael R. Garey and David S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[13] Rajiv Gupta, Mary L. Soffa, and Dennie Ombres. Efficient register allocation via coloring using clique separators. *ACM Transactions on Programming Languages and Systems*, pages 370–386, May 1994.

[14] Rajiv Gupta, Mary L. Soffa, and Tim Steele. Register allocation via clique separators. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 264–274, July 1989.

[15] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[16] Todd A. Proebsting and Charles N. Fischer. Probabilistic register allocation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 300–310, June 1992.

[17] Robert E. Tarjan. Decomposition by clique separators. *Discrete Mathematics*, 55:221–232, 1985.