A Faster Optimal Register Allocator

Changqing Fu and Kent Wilken
Department of Electrical and Computer Engineering
University of California, Davis
Davis, CA 95616, U.S.A.
{*cqfu,wilken*}*@ece.ucdavis.edu*

Abstract

Abstract – *Recently researchers have proposed modeling register allocation as an integer linear programming (IP) problem and solving it optimally for general purpose processors [17, 20] and for dedicated embedded systems [23]. Compared with traditional graph-coloring approaches, the IP-based allocators can improve a program's performance. However, the solution times are much slower.*

This paper presents an IP-based optimal register allocator which is much faster than previous work. We present several local and global reduction techniques to identify locations in a program's control-flow graph where spill decisions and register deallocation decisions are unnecessary for optimal register allocation. We propose a hierarchical reduction approach to efficiently remove the corresponding redundant decisions and constraints from the IP model. This allocator is built into the Gnu C Compiler and is evaluated experimentally using the SPEC92INT benchmarks. The results show that the improved IP model is much simpler. The number of constraints produced is almost linear with the function size. The optimal allocation time is much faster, with a speedup factor of about 150 for hard allocation problems.

1. Introduction

Register allocation is one of the most important compiler optimization phases. Effective register allocators can improve a programís performance [9], reduce a programís code-size [23] and lower a programís power consumption [28]. Basically, the register allocation problem is to assign a target processorís limited *real registers* to the unlimited *symbolic registers* used in the compilerís representation of the program. The objective is to minimize *register allocation overhead*, the performance overhead of the *spill code* inserted when no real register is available for a symbolic register.

The register allocation problem can be categorized into local allocation and global allocation. Local register allocation assigns real registers to symbolic registers for each *basic block*, a sequence of straight-line code. Global register allocation assigns real registers to symbolic registers throughout a procedure or a function. Sethi [27] proves that global register allocation is NP-complete, and Farach and Liberatore [13] show that local register allocation is NP-complete. Various heuristics have been proposed for local [1, 14, 15, 21], and global [5, 8, 9, 10, 18, 26] register allocations.

This paper focuses on the harder global register allocation problem. The traditional global allocator is based on a graph-coloring heuristic developed by Chaitin et al. [9]. Various graph-coloring improvements have been proposed more recently [3, 4, 6, 7, 8, 10, 11, 18, 25]. These improvements rely on different heuristics to determine the placement of the spill code and the order of the symbolic registers selected for spilling.

Practically, graph-coloring allocators have worked well in modern compilers. However, the quality of the solution produced cannot be guaranteed. Sometimes graph-coloring allocators produce register overhead which signiÝcantly degrades program performance [17]. Issues about register coalescing, procedure call conventions, special register requirements for irregular register architectures and for dedicated embedded systems make it more difÝcult to produce good allocations.

[16, 17, 20, 23] recently proposed modeling register allocation as an integer linear programming (IP) problem and solve it optimally with an IP solver. They show that although optimal register allocation has been considered to be computationally intractable [22], practically it is solvable in most cases. Goodwin and Wilken [17] proposed an *Optimal Register Allocator* (ORA) for a uniform register architecture. Kong and Wilken [20] present a framework based on ORA for irregular architectures, like the x86 architecture. Naik and Palsberg in [23] use IP to model code-size aware register allocation for embedded microproces-

Proceedings of the 35 th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)
1072-4451/02 $17.00 © 2002 IEEE

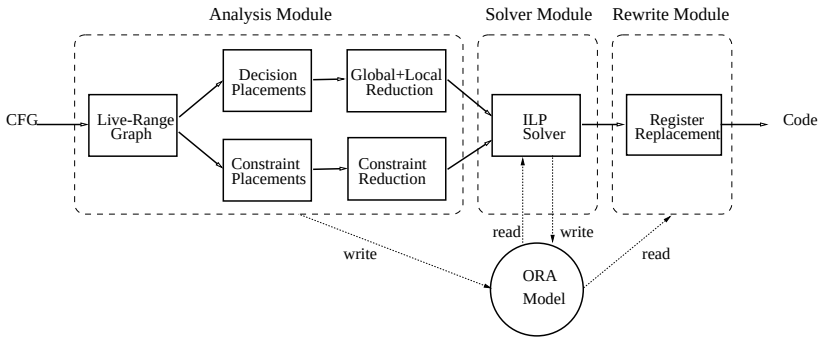


Figure 1. Faster ORA allocator framework.

sors. [17] shows that ORA can signiÝcantly reduce register allocation overhead compared with the graph-coloring allocator. [23] shows that an IP-based allocator can generate code as compact as carefully hand-crafted code. However, the IP allocator solution times are slow.

A primary reason for the slow solution times is the high complexity of the IP model. Many redundancies exist in the proposed IP models. These redundancies can dramatically increase the number of solutions and the time for Ýnding an optimal solution.

Appel and George [2] propose a faster IP-based allocator. Their approach reduces the IP model complexity by decomposing the register allocation problem into two subproblems: spill code placement and register assignment, and solves each subproblem using IP. Although the IP-based allocator is faster, the decomposition results in an allocation which may not be optimal.

This paper presents a faster optimal global register allocator based on new techniques for reducing IP model complexity. The new techniques identify and reduce redundancies from the IP model while preserving the allocationís optimality. Our allocator builds on the *basic ORA model* proposed in [17]. We analyze the placement of spill decisions and register deallocation decisions produced in the program control-Đow graph (CFG), and analyze the dependency between a potential register allocation action (load) and a register deallocation action. We explore several reduction techniques to identify locations where spill decisions and deallocation decisions are unnecessary for an optimal register allocation. A region-based hierarchical reduction approach is used to efÝciently remove the corresponding redundant decisions and constraints from the basic ORA model.

Our results show that the improved IP model is much simpler. The number of constraints generated in the integer programs grows almost linearly with the program size $(O(n^{1.1}))$. Optimal allocation is much faster than the basic ORA. For 2202 functions from the SPEC92 integer benchmarks, the new allocator solves each function within a time limit 7 seconds, while the basic ORA requires a time limit 1024 seconds, making the new allocator 150X faster for the hard problems from these functions.

The rest of this paper is organized as follows. Section 2 gives an overview of optimal register allocation and the faster ORA allocator. Section 3 introduces ORA live-range graphs. Section 4 and 5 present global and local reduction techniques for load, store and deallocation decisions. Section 6 presents various constraint reduction techniques. Section 7 shows the experimental results and Section 8 summarizes the paperís contributions.

2. Faster optimal register allocator overview

The faster ORA allocator consists of three components: the analysis module, the solver module and the rewrite module, as illustrated in Figure 1.

The analysis module analyzes a functionís CFG to construct a set of symbolic register live-range graphs, and determines the locations in the live-range graphs where various allocation decisions should be made and various allocation constraints should be enforced. These decisions and constraints are a sufÝcient set for an optimal allocation [17]. Local and global reduction techniques are then applied to identify and eliminate unnecessary allocation decisions. An objective function for minimizing register allocation overhead is produced . The objective function and the remaining decisions and constraints comprise the ORA model.

Each decision in the ORA model represents a register allocation action. For example, a *define* decision represents whether or not a symbolic register should be deÝned into a speciÝc real register, and a *deallocation* decision represents whether or not the speciÝc real register should be deallocated from the symbolic register. Each decision is a binary decision. A 1(0) indicates a *taken (not taken)* register allocation action.

The constraints ensure a valid register allocation is produced for the set of decisions. For example, a *single-symbolic constraint* [17] is used to ensure a real register can be allocated to at most one symbolic register at any point of the program.

The objective function consists of a set of *costed* decisions to ensure a valid solution is optimal. The cost of a

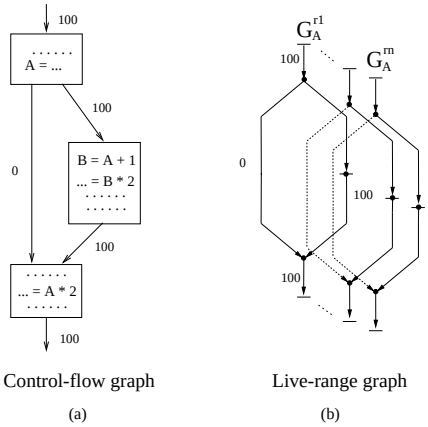


Figure 2. Control-flow graph and live-range graph.

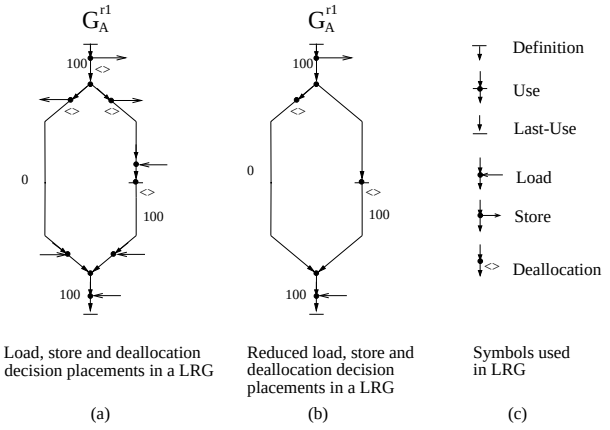


Figure 3. Placements of load, store and deallocation decisions in a LRG.

decision is equal to the estimated dynamic instruction execution count which will occur if the corresponding action is taken.

The solver module constructs a 0-1 integer program [24] from the ORA model, and passes it to an IP solver. The solver determines a value of either 0 or 1 for each decision so the constraints are satisÝed and the objective function is minimized.

The rewrite module rewrites the intermediate instructions based on the solver solution, with each symbolic register replaced by the assigned real register, and with spill code inserted at the prescribed locations.

3. ORA Live-Range Graphs

An ORA *Live-Range Graph* (LRG) is a directed subgraph derived from a CFG. A LRG starts at a symbolic registerís deÝnition(s) and ends at the symbolic registerís last-use(s). For each symbolic register, a LRG is constructed for each real register. Figure 2(a) shows a CFG, and Figure 2(b) shows the LRGs for symbolic register A for each real register from r_1 to r_n. The number labeled at each edge of the graphs is the estimated execution count of the basic block where the edge resides (e.g. from program proÝling).

LRGs are used to mark the placement of various register allocation decisions, such as load, store and deallocation decisions. In the basic ORA model, various allocation decisions are placed at locations which are sufÝcient for optimal register allocation [17]:

- *Load* decisions, which represent whether or not the value of a symbolic register should be loaded from memory, are placed before every symbolic register use and placed at every LRG merge edge.
- *Store* decisions, which represent whether or not the value of a symbolic register should be saved into memory, are placed after every symbolic register deÝnition and at every LRG diverge edge.
- *Deallocation* decisions, which represent whether or not the value of a symbolic register should be deallocated from a real register, are placed after every definition, after every use and at every diverge edge.

Figure 3(a) shows the placements of load, store and deallocation decisions in the live-range graph of symbolic register A for real register r_1 in the basic ORA model. There are 4 load decisions, 3 store decisions and 4 deallocation decisions. The symbols used in the LRG are given in Figure 3(c).

The various allocation decisions used in the basic ORA model are sufÝcient, but not necessary for optimal register allocation. Some of the decisions are redundant and can be removed. This is because these decisions may represent register allocation actions that are not required for an optimal register allocation. Typically, a register allocation problem will have many different optimal allocations, each resulting in the same register allocation overhead. To ensure optimality, the decisions necessary to produce at least one of the optimal allocations must be retained. However, many of the decisions for other optimal allocations can often be eliminated.

For example, the load decision and the deallocation decisions placed between the deÝnition of A and the use of A in the graph $G_A^{r_1}$ in Figure 3(a) are redundant decisions because in an optimal allocation, once A is allocated to a real register at the deÝnition instruction, A will remain in this real register until the use of A. From this observation, we derive an interval-based local reduction technique which will be introduced in Section 5.

In addition, the decision to store A on the right diverge edge in the graph is redundant because this store decision is dominated by a equal cost store decision placed after the deÝnition of A. Similarly, a decision to load A on the right merge edge in the graph is redundant because this load decision is post-dominated by a equal cost load decision placed before the last-use of A. After this load decision is elim-

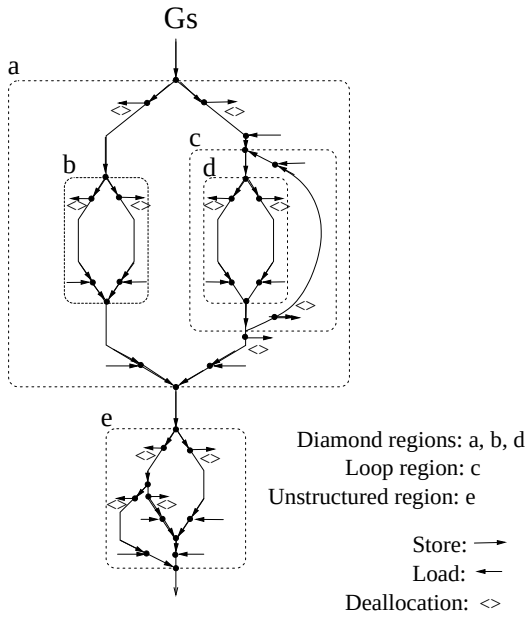


Figure 4. Decision placement hierarchy.

inated, the decision to load A on the left merge edge becomes redundant because for any valid allocation, the allocation of A at both incoming edges of the merge vertex has to be the same. Finally, the decision to store A on the left diverge edge in the graph is redundant because no load will use the stored value. From these observations, we derive several global reduction techniques which are presented in Section 4.

Figure 3(b) shows the simpliÝed graph $G_A^{r_1}$ for the example used in Figure 3(a).

4. Global reduction

Global reduction eliminates unnecessary load, store and deallocation decisions placed at the diverge and merge edges in the live-range graphs. These decisions account for more than 80% of the total decisions generated by the basic ORA model. Reducing these decisions can signiÝcantly simplify the modelís complexity.

4.1. Decision placement

The placement of the deallocation decisions at the diverge edges and the load decisions at the merge edges in a LRG can have a hierarchical structure as shown in Figure 4. Analysis of this hierarchical structure can reduce unnecessary decisions from the basic ORA model. A *diverge-merge region* is a single-entry single-exit (SESE) region [19], which starts at a diverge vertex and ends at a merge vertex of the LRG. Each diverge-merge region is categorized as: a *diamond region*, a *loop region* or an *unstructured region*.

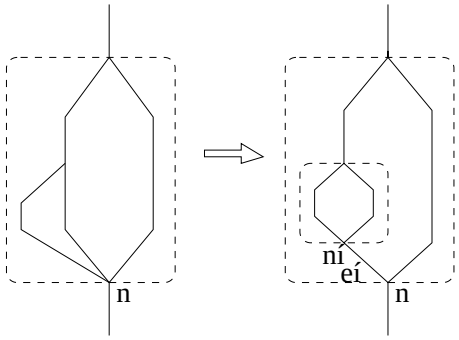


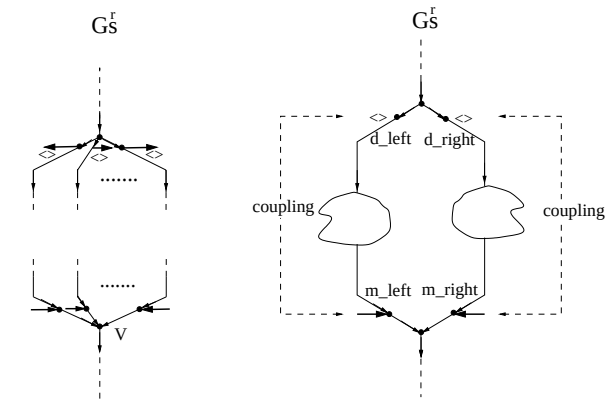
Figure 5. Node splitting for an unstructured region.

A diamond region is a two-way diverge (entry) and two-way merge (exit) SESE region; each diverge edge immediately dominates [1] only one merge edge, and the dominated merge edge immediately postdominates [1] the diverge edge. A loop region is a SESE region where the entry of the region is a merge vertex, the exit of the region is a diverge vertex, and a *backedge* [1] connects the entry to the exit. All other SESE regions are unstructured regions. In Figure 4, regions a, b and d are diamond regions, region c is a loop region and region e is an unstructured region.

Some unstructured regions can be transformed into diamond regions by using *node splitting* [1] if the unstructured regions are multi-way diverge or multi-way merge region. Figure 5 shows a three-way merge unstructured region has been transformed into two diamond regions by introducing a new node n' and a new edge e'.

Lemma 1 (Load, store and deallocation decoupling) *For an optimal register allocation, it is unnecessary to load symbolic register S to real register r at all incoming edges of a merge vertex, or to store S or to deallocate S from r at all outgoing edges of a diverge vertex.*

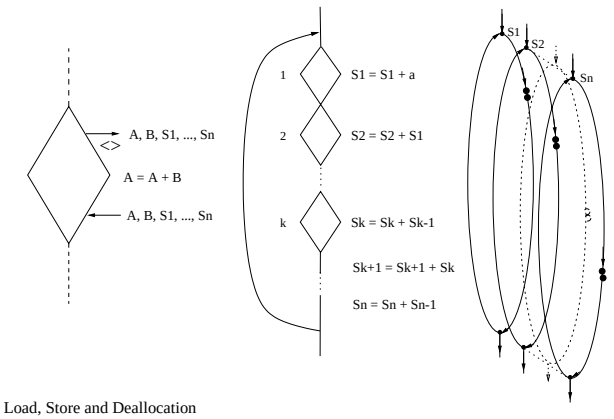
Proof: By construction. Assume an optimal solution O includes load of S at each incoming edge of merge vertex V. Another optimal solution O' without a load of S at any incoming edge of V can be constructed as follows: Remove the loads of S from each incoming edge of V. Perform depth-Ýrst-search (DFS) traversal of the live-range graph of S forward from V. Place a load of S preceding a use of S, and place a load of S at a merge edge of the graph. Because the set of the inserted load locations immediately postdominates the removed load locations, the removed load locations immediately dominates the set of the inserted load locations, the constructed solution is cost equivalent to the optimal solution. Furthermore, because no use is between the two set of load locations, the constructed solution is a valid solution. Similar arguments hold for stores and deallocations at all outgoing edges of a diverge vertex. □



Load, store and deallocation decoupling (a)

Load and deallocation coupling (b)

Figure 6. Decision decoupling and coupling.



Load, Store and Deallocation Decisions in a Diamond Region (a)

Control Flow Graph (b)

Live Range Graphs (c)

Figure 7. Symmetric solutions in diamond regions.

4.2. Diamond region reductions

Diamond regions are common constructs in LRGs. This section presents four reduction techniques which can eliminate unnecessary load, store, and deallocation decisions in diamond regions: Void region coupling, symmetric decision selection, jump-edge nulliÝcation, and asymmetric decision elimination.

4.2.1. Void region coupling

void region A LRG region is *void* for symbolic register S if S is not redeÝned or used in the region.

coupled decision Two decisions are *coupled* if the taken action of one decision is associated with the taken action of another decision.

paired decision A decision placed on a diverge edge in a diamond region is *paired* with the decision placed on the same side merge edge in the diamond region if the diverge edge starts at the entry of the diamond region, the merge edge ends at the exit of the diamond region.

Theorem 4.1 (Void region coupling) *For optimal register allocation, if a diamond region is void for symbolic register S, then the paired deallocation and load decisions of S are coupled in the region.*

Proof: By contradiction. Assume an optimal solution deallocates S from real register r on a diverge edge in a diamond region and does not load S to r on the same side merge edge in the region, or assume an optimal solution does not deallocate S from r on the diverge edge and loads S on the merge edge. S can not be in r at the merge vertex or at the diverge vertex of the region. This requires S to be deallocated from r for both diverge edges or to be loaded into r for both merge edges. But these are contradictions because from Lemma 1 it is never necessary to deallocate r on both diverge edges or to load r on both merge edges. □

By coupling the paired deallocation and load decisions, one decision replaces the two decisions in the basic ORA model.

4.2.2. Symmetric decision selection

symmetric Two decisions x and y are *symmetric decisions* if for any solution T with a taken action of x and a not taken action of y, there always exists an equivalent cost solution T' with a not taken action of x and a taken action of y. The solutions T and T' are *symmetric solutions.*

Figure 7(a) shows a void diamond region for symbolic registers S_1 to S_n. In the basic ORA model, coupled deallocation/load decisions for S_1 to S_n are placed at the diverge edge and at the merge edge of the region for each of R real registers available for allocation. The total number of the coupled deallocation/load decisions is $n \times R$.

However, many of these decisions are symmetric decisions because an optimal solution requires at most 2 real registers for symbolic registers deÝned and used inside the region. The coupled deallocation/load decisions for any set of 2 out of R real registers are symmetric decisions. These symmetric decisions produce the total number of $\binom{R}{2}$ symmetric solutions for this region.

Theorem 4.2 (Symmetric decision selection) *For optimal register allocation problem, if the number of the symbolic registers which are void to a diamond region is greater than the number of real registers available for allocation, and the region is colorable with k colors, then it is sufficient to arbitrarily select k real registers as the allocation set for the coupled deallocation/load decisions of the symbolic registers which are void to the region.*

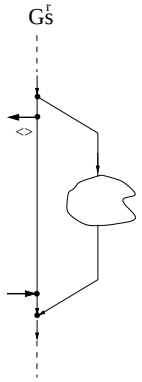


Figure 8. Jump-edge nullification.

Proof: By contradiction. Suppose $k+1$ symbolic registers which are void to the region are deallocated from and loaded into $k+1$ real registers at the diverge edge and the merge edge of the region. Because k real registers will be enough for allocation for all symbolic registers deÝned or used in the region without spills, at least one of the $k+1$ real registers will not be used by any of those symbolic registers deÝned or used inside the region. It is unnecessary to deallocate the real register from a symbolic register at the diverge edge of the region and load back into the same symbolic register at the merge edge of the region. □

By applying the proposed reduction technique, the number of symmetric decisions has been reduced from $n \times R$ to $n \times 2$, and the number of symmetric solutions has been reduced by a factor of $\binom{R}{2}$. When $n = 40$ and $R = 32$, the number of the reduced symmetric decisions is 1200, and the reduction of the symmetric solutions is about a factor of 500.

In practice, the number of symmetric decisions produced are very large for some cases. The large number of symmetric decisions can produce a large number of symmetric solutions which causes a large number of *branch-and-bound* [24] iterations for the IP solver to Ýnd an optimal integer solution. Figure 7(b) shows an example of a control-Ðow graph which illustrates the huge reduction in symmetric solutions the technique can produce. Figure 7(c) shows the live range graphs of the symbolic registers. The number of the symmetric decisions produced before and after applying the reduction technique are $Q_{before} = n \times R \times k$, and $Q_{after} = n \times 2 \times k$, where k is the number of diamond regions in the graph. The total number of symmetric solutions have been reduced by a factor of $P = \binom{R}{2}^k$. For $n = 40$, $R = 32$, and $k = 10$, the number of symmetric decisions have been reduced from 12800 to 800, and the number of symmetric solutions have been reduced by a factor of 10^{11}.

4.2.3. Jump-edge nullification

jump edge A *jump edge* is both a merge edge and a diverge edge in a LRG.

merge constraint The *merge constraint* enforces that the allocation of a symbolic register on any of the incoming edges of a merge vertex must be same for each real register.

A jump edge is void for any symbolic register live through it. ORA allows symbolic registers to be deallocated and spilled on a jump edge to satisfy the merge constraint at the end vertex of the jump edge. Under certain conditions, the deallocation and spill decisions on a jump edge are unnecessary for optimal register allocation.

Theorem 4.3 (Jump-edge nullification) *For an optimal register allocation, when a symbolic register S is live through a diamond region, and a jump edge connects the entry of the region to the exit:*

a) the deallocation decision on the jump edge for S is unnecessary if S is void in the region,

b) the store decision on the jump edge for S is unnecessary if S is not defined in the region,

c) the load decision on the jump edge for S is unnecessary if S is not defined in the region and there is no load of S before a use of S in the region.

Proof: By contradiction. a) Suppose S is deallocated from real register r on the jump edge. Based on Theorem 4.1, S must be loaded into r on the jump edge as well. Because no symbolic register will reuse r on the jump edge, it is redundant to deallocate and reload S on the jump edge.
b) Suppose S is stored into memory on the jump edge from real register r. Without a deÝnition of S in the region, S must be either stored into memory from r on both diverge edges, or loaded into r or loaded into another real register t on the jump edge. For case 1, based on Lemma 1, it is never necessary to store S on both diverge edges. For case 2, it is redundant because no symbolic register will reuse r. For case 3, the merge constraint will enforce a load of S to t on both merge edges. Based on Lemma 1, it is never necessary to load S on both merge edges.
c) Suppose S is loaded into real register r on the jump edge. Without a deÝnition of S and without a load of S before any use of S in the region, the merge constraint will enforce S to be loaded on both merge edges. Base on Lemma 1, the loads of S on both merge edges are never necessary for optimal register allocation. □

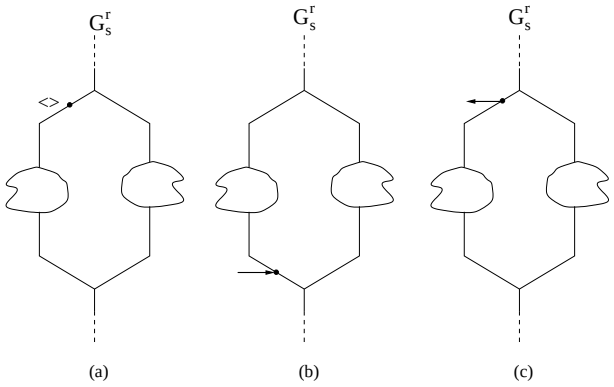


Figure 9. Asymmetric decisions.

4.2.4. Asymmetric decision elimination

asymmetric decision A register allocation decision is *asymmetric* if the decision is placed on one, not both diverge edges starting at the entry of a diamond region or placed on one, not both merge edges ending at the exit of the diamond region.

Figure 9(a)-9(c) show the asymmetric deallocation, asymmetric load and asymmetric store decisions. Under certain conditions, an asymmetric decision is unnecessary for optimal register allocation.

Theorem 4.4 (Asymmetry elimination) *For an optimal register allocation, when symbolic register S is live through a diamond region:*

a) an asymmetric deallocation decision of S is unnecessary if there is no load of S along at least one path from the deallocation decision location to the exit vertex of the region and no deallocation along at least one path of the other side of the region.

b) an asymmetric load decision of S is unnecessary if there is no deallocation of S along at least one path from the entry vertex of the region to the load decision location and no load along at least one path of the other side of the region.

c) an asymmetric store decision of S is unnecessary if there is no load of S along any path from the store decision location to the exit vertex of the region.

Proof: Based on Lemma 1 and the merge constraint. □

4.3. Loop and unstructured regions

The asymmetry reduction technique presented previously can be applied to loop regions and unstructured regions, and the jump-edge reduction technique can be applied to the jump edge in a loop region.

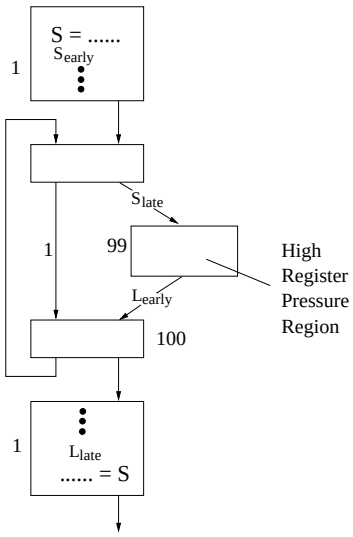


Figure 10. Cost-guided load/store reductions.

4.4. Cost-guided reduction

Previously presented reduction techniques identify redundant allocation decisions by taking advantage of a programís structure. Cost-guided reduction is to identify redundant load and store decisions by considering the allocation overhead of each decision.

Theorem 4.5 (Dominated store) *Given two store decisions allowed at an early location and a late location in a control-flow graph for symbolic register S, the late store decision is redundant for optimal register allocation if 1) the early location dominates the late location, and 2) the execution count of the basic block at the early location is less than or equal to the execution count of the basic block at the late location.*

Proof: The Ýrst condition ensures the correctness when replacing the late store with the early store. The second condition ensures the allocation overhead to place a store at the early location is less than or equal to the overhead to place a store at the late location. □

Figure 4.5 shows a portion of a control-Ðow graph labeled with execution counts, that contains a region of high register pressure. Symbolic register S is allowed to store at either S_{early} or S_{late} in the basic ORA model. However, optimal register allocation will never store S at S_{late} because the allocation overhead to store S at S_{early} is cheaper.

Symmetrically, a load placed at the L_{early} as shown in the Figure 4.5 is redundant based on the following theorem.

Theorem 4.6 (Postdominated load) *Given two load decisions allowed at a early location and a late location in a control-flow graph for symbolic register S, the early load decision is redundant for optimal register allocation if 1)*

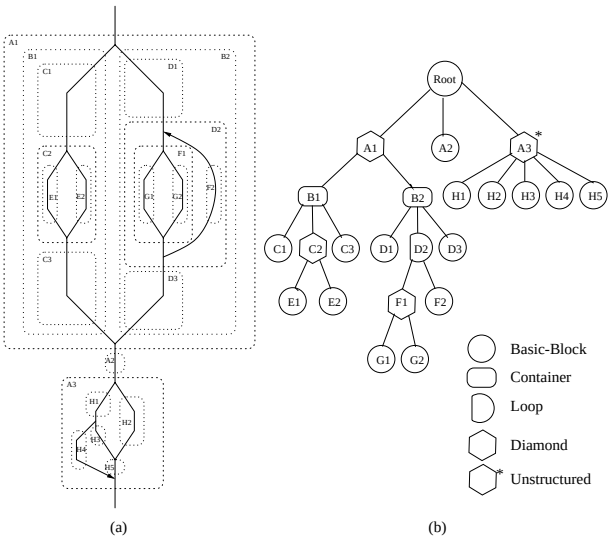


Figure 11. SESE region hieararchy and program structure tree.

the late location postdominates the early location, 2) a load at the later location must result in the symbolic register S being in a real register at every use and last-use that a load at the earlier location would, 3) the execution count of the basic block at the late location is less than or equal to the execution count of the basic block at the early location.

Proof: Proof is similar to Theorem 4.5. □

4.5. Building the hierarch y

The hierarchical reduction approach starts by building the *program structure tree* (PST)[19], a hierarchical representation of program structure based on *single-entry single-exit* (SESE) regions of the control Đow graph. Figure 11(a) shows a program control-Đow graph with its SESE regions marked. Figure 11(b) is the region-based PST. There are Ýve types of SESE regions: basic-block regions, container regions, loop regions, diamond regions and unstructured regions. A *container region* is the maximal sequential set of SESE regions.

Once the PST is built, the basic block regions will be visited Ýrst during the DFS traversal of the tree in post-order. Data Đow analysis is accomplished for basic block regions by scanning each instruction. The data Đow information of the children regions are combined into the parent regions (containers). Structured and unstructured regions are analyzed separately by applying the previous discussed reduction techniques.

5. Local reduction

Local reduction examines symbolic registers used in adjacent instructions to identify unnecessary load and deallocation decisions generated in the basic ORA model. *def-use intervals* are introduced as the analysis scope.

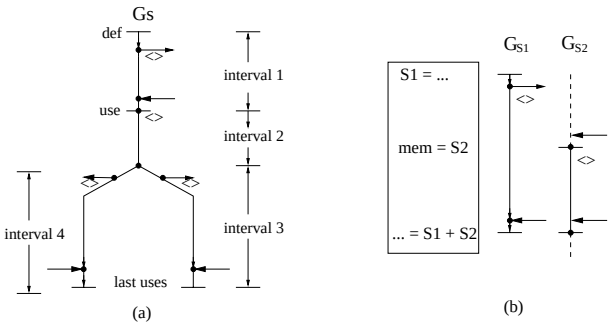


Figure 12. (a) Def-use intervals and decision placements, (b) interval-based reduction.

def-use interval A def-use interval is a segment of the live-range graph of a symbolic register S that starts at a deÝnition or a use of S, ends at an adjacent use or last-use of S, and contains no merge vertex.

When an interval crosses over a diverge vertex, the interval is segmented at the diverge vertex to form multiple intervals. Figure 12(a) shows the four def-use intervals of symbolic register S.

Theorem 5.1 *For an optimal register allocation, the deallocation and load decisions placed in an interval of symbolic register S are unnecessary if*

a) no symbolic register except S appears within the interval, or

b) any symbolic register appearing is used at the start or at the end of the interval.

Proof: By contradiction. Assume an optimal solution deallocates S inside the interval. a) Because no symbolic register except S will use the real register deallocated from S in the interval, it is unnecessary to deallocate the real register holding S, or load S back in the interval. b) The only symbolic registers which can use the real register deallocated from S are the symbolic registers which are used together with S at the start or at the end of the interval. For a valid allocation, those symbolic registers must have been allocated with different real registers at the start or at the end of the interval. □

Figure 12(b) shows a situation where the deallocation and load decisions for symbolic register S_1 and S_2 in their def-use intervals are redundant.

6. Constraint Reduction

Constraint reductions are based upon decision reductions. After redundant decisions have been eliminated,

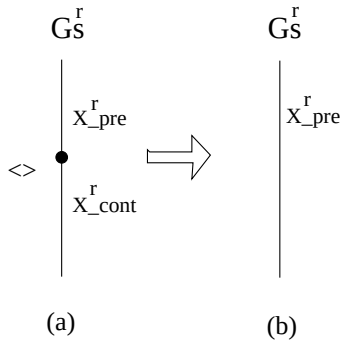


Figure 13. Deallocation constraint before (a) and after (b) reduction

some constraints may not exist, or some constraints may be automatically satisÝed by other constraints. This section will analyze redundant deallocation constraints, must-allocate constraints and single-symbolic constraints generated in the basic ORA model.

Deallocation Constraints A *deallocation constraint* is constructed at a deallocation decision location of symbolic register S to allow deallocation of S.

$$X^r_{pre} + X^r_{deallocate} = X^r_{cont} \quad (1)$$
$$r \in R : real\ register\ set$$

As shown in Figure 13, when the deallocation decision is eliminated, the decision variable $X_{deallocate}$ in the Equation 1 equals 0. The Ðow variables X_{pre} and X_{cont} are reduced to one variable as shown in Figure 13(b). The deallocation constraint is no long necessary.

Must-Allocate Constraint A *must-allocate constraint* is constructed after each deÝnition and before each use of a symbolic register in the ORA model [17] to ensure a symbolic register must be allocated to a real register at each deÝnition and each use. A must-allocate constraint is constructed as follows:

$$\sum_{r \in R} x^r_s \geq 1 \quad (2)$$
$$R : real\ register\ set$$

For optimal register allocation, if no deallocation decision exists between two adjacent must-allocate constraints for a symbolic register, then the second must-allocate constraint is redundant because only allocation decisions are inserted between the locations of the two constraints. The decision variable terms in the second constraint are a superset of the decision variable terms in the Ýrst constraint. The second must-allocate constraint can be automatically satisÝed by the Ýrst must-allocate constraint.

Single-symbolic Constraint A *single-symbolic constraint* is constructed after each deÝnition and after each load for a real register in the ORA model [17] to ensure a real register can be allocated to at most one symbolic register. A single-symbolic constraint is constructed as follows:

$$\sum_{s \in S} x^r_s \leq 1 \quad (3)$$
$$S : live\ symbolic\ register\ set$$

For optimal register allocation, if no deallocation constraint exists between two adjacent single-symbolic constraints for a real register, then the Ýrst single-symbolic constraint is redundant because only allocation decisions are inserted between the locations of the two constraints. The decision variable terms in the Ýrst constraint are a subset of the decision variable terms in the second constraint. The Ýrst single-symbolic constraint can be automatically satisÝed by the second single-symbolic constraint.

7. Experimental results

This section evaluates the faster optimal register allocator in terms of compile-time performance, model complexity and code quality.

7.1. Methodology

The faster optimal register allocator is compared experimentally against Goodwin and Wilkenís basic ORA allocator [17] in terms of both compile-time performance and model complexity. The faster allocator is compared against a Chaitin-style graph-coloring allocator in terms of code quality. The graph-coloring allocator is implemented based on the optimistic allocator by Briggs et al., [5]. Th optimistic allocator includes rematerialization and coalescing to reduce spill code overhead. Execution proÝle is used to determine spill costs for each allocator.

The faster ORA allocator, the basic ORA allocator and the graph-coloring allocator are all built into the GNU C compiler (GCC), version 2.5.7, targeted to the PA-RISC architecture with 24 real registers available for allocation. The SPEC92 integer benchmark suite, which consists of 2399 integer functions, is used for all experiments. This benchmark set is selected so that the results are directly comparable with those published in [17]. The integer programs are solved using a commercial integer programming solver, CPLEX 7.1 [12], running on a HP9000/785 workstation with a 360MHz PA-8000 processor and 1536MB of main memory.

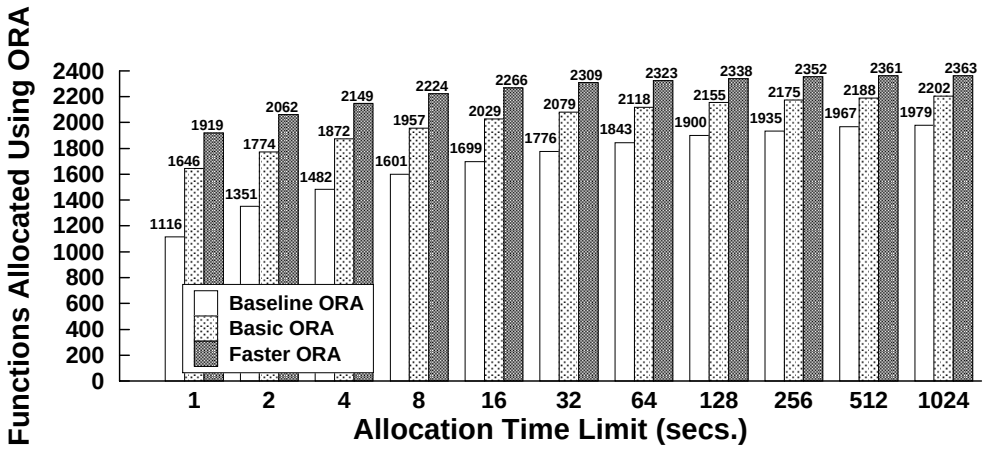


Figure 14. SPEC92 Integer benchmark functions allocated using the baseline ORA allocator, the basic ORA allocator and the faster ORA allocator.

7.2. Compile-time performance

Figure 14 shows the number of functions out of the 2399 functions in the SPEC92 integer benchmarks solved optimally by the basic ORA allocator and by the faster ORA allocator for time limits from 1 to 1024 seconds. The time refers to the allocation time used by each allocator for each function. The allocation time includes the analysis time, the integer-program solution time and the instruction rewrite time. Figure 14 also shows the results published in [17]. The baseline allocator is built on the same model as the basic ORA allocator, but the experiments use CPLEX 3.0 running on a HP9000/735 workstation with a 99MHz PA-7000 processor and 128MB of main memory.

The basic ORA allocator is compared against the baseline allocator to reÐect the speedup due to the faster machine and the improved CPLEX solver. The faster ORA allocator is compared against the basic ORA allocator to reÐect the speedup due to the improved model. The number of functions solved optimally in a time limit of 1024 seconds by each base allocator is chosen for each comparison to reÐect the ability of each allocator to solve *hard problems*, which require greater allocation time.

The baseline ORA allocator solves 1979 functions optimally within a time limit of 1024 seconds, while the basic ORA allocator solves 1979 functions optimally within a time limit of 10 seconds. This represents a 100X speedup for the hardest of these 1979 functions, which is due to the faster machine and the improved algorithm in the newer CPLEX solver. The basic ORA allocator solves 2202 functions optimally within a time limit of 1024 seconds, while the faster ORA allocator solves the same number of functions optimally within a time limit of 7 seconds. This represents a 150X speedup for the hardest of these 2202 functions, which is due to the improved model. Furthermore, the faster allocator solves 161 more functions optimally within a time limit of 1024 seconds, functions which are intractable using the basic ORA.

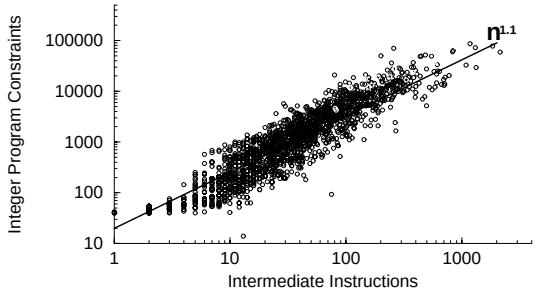


Figure 15. ORA integer program size vs. function size for the SPECT92 integer benchmarks.

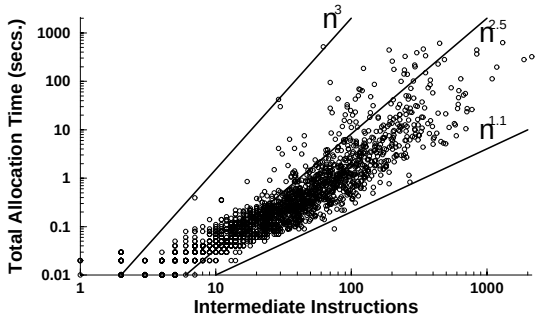


Figure 16. Faster ORA allocator's allocation time vs. function size for the SPECT92 integer benchmarks.

7.3. Model complexity

Figure 15 shows the almost linear relationship, $O(n^{1.1})$, between the number of constraints produced in the faster allocator model and function size. Compared to the $O(n^{1.3})$

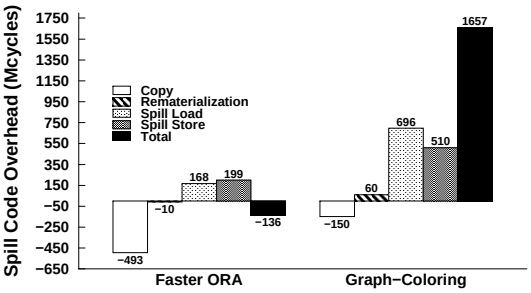


Figure 17. Spill code overhead components for the functions allocated by the ORA allocator, and by the optimistic graph-coloring allocator

relationship in the basic ORA model, the faster allocator model is much simpler.

Figure 16 shows the $O(n^{2.5})$ relationship between the total allocation time and function size. By comparison, the basic ORA model exhibits $O(n^3)$ run-time complexity [17], a Chaitin-style graph-coloring allocator exhibits $O(nLogn)$ run-time complexity [5] and a priority-based graph-coloring allocator exhibits $O(n^2)$ run-time complexity [5].

7.4. Code quality

Figure 17 shows the components of the dynamic spill code overhead for the faster allocator and the optimistic graph-coloring allocator for functions allocated optimally by the faster allocator with a 1024 second time limit. The copy component is negative because it represents copies that are eliminated. The rematerialization component is negative because it represents the dynamic overhead to reallocate the rematerializations is cheaper than the dynamic overhead to keep rematerializations at the original locations. Compared with the graph-coloring allocator, the faster allocator inserted 3.5 times fewer loads and stores, and removes 3.3 times more copies.

8 Summary

This paper presents a reduction approach to improve IP-based register allocator to solve the global register allocation problem optimally and efÝciently. EfÝciency is one of the major concerns for IP-based register allocation approaches to work well practically. Even though the IP allocators can signiÝcantly reduce allocation overhead compared with graph-coloring allocators, the slow solution time has impeded its application in commercial compilers. Theoretically, integer programming is NP-complete. However this paper shows that by understanding the characteristics of the register allocation problem from optimal point of view, one can Ýnd ways to simplify the model without losing its optimality. The reduction techniques presented in this paper are one example of using this methodology. It is shown that these techniques can successfully simplify the IP-based optimal register allocation model and speedup the solution time by a factor of 150 for hard problems. This paper also suggests that with the parallel progress of improved machines, improved solvers and improved models, optimally solving all practical register allocation problems may become possible in the future.

Understanding the characteristics of an optimal register allocator is not the only beneÝt for improving the efÝciency of the ORA allocator. These techniques may provide guidance to enhance graph-coloring allocators for better allocation quality as well. The identiÝcation of redundant or unnecessary deallocation decision locations in the control-Đow graph may be useful for graph-coloring allocators to make better live-range splitting decisions. Similarly, the identiÝcation of redundant spilling decision locations in the graph may be useful for graph-coloring allocators to make better live-range spilling decisions.

Acknowledgments

This research is supported by the National Science Foundation under grant CCR-9711676.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[2] A. W. Appel and L. George. Optimal spilling for cisc machines with few registers. In *SIGPLAN conference on Programming language design and implementation*, June 2001.

[3] D. Bernstein, M. C. Golumbic, Y. Mansour, R. Y. Pinter, D. Q. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compilers. In *SIGPLAN conference on Programming language design and implementation*, volume 24, pages 258ñ263, Portland, OR, 1989.

[4] M. P. Bivens. Incremental register reallocation. volume 20, pages 1015ñ1047, October 1990.

[5] P. Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, April 1992.

[6] P. Briggs, K. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428ñ455, May 1994.

[7] R. G. Burger, O. Waddell, and R. K. Dybvig. Register allocation using lazy saves, eager restores, and greedy shufÐing. In *SIGPLAN conference on Programming language design and implementation*, Feburary 1995.

[8] D. Callahan and B. Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of Conference on Programming Language Design and Implementation*, pages 192ñ203, June 1991.

[9] G. Chatin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages*, 6:47ñ57, 1981.

[10] F. Chow and J. Hennessey. The priority-based coloring approach to register allocation. *ACM Trans. on Programming Languages and Systems*, 12(4):501ñ536, October 1990.

[11] L. P. Cindy Norris, Lori. Register allocation over the program dependence graph. In *SIGPLAN conference on Programming language design and implementation*, Florida, Orlando, 1994.

[12] CPLEX Optimization, Inc. *Using the CPLEX Callable Library*. 1994.

[13] Farach and Liberatore. On local register allocation. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1998.

[14] C. W. Fraser and D. R. Hanson. Simple register spilling in a retargetable compiler. *Software-Practice and Experience*, 22(1):85ñ99, January 1992.

[15] R. A. Freiburghouse. Register allocation via usage counts. *CACM*, 17:638ñ642, 1974.

[16] D. Goodwin. *Optimal and Near-Optimal Global Register Allocation*. PhD thesis, University of California, Davis, December 1996.

[17] D. Goodwin and K. Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. *Software Practice and Experience*, 26(8):929ñ965, August 1996.

[18] T. G. Guei-Yuan Lueh and A.-R. Adl-Tabatabai. Global register allocation based on graph fusion. In *9th Workshop on Languages and Compilers for Parallel Computing*, August 1996.

[19] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. *ACM SIGPLAN Notices*, 29:171ñ185, 1994.

[20] T. Kong and K. D. Wilken. Precise register allocation for irregular architectures. In *International Symposium on Microarchitecture*, pages 297ñ307, 1998.

[21] E. Lowry and C. Medlock. Object code optimization. *CACM*, 12:13ñ22, 1969.

[22] M.R.Gary and D.Johnson. *computers and Intractability:A Guide to the to the Theory of NPCompleteness*. San Fransisco: W. H. Freeman, 1979.

[23] M. Naik and J. Palsberg. Compiling with code-size constraints. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES)*, June 2002.

[24] G. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, 1988.

[25] D. E. M. O. K. Peter Bergner, Peter Dahl. Spill code minimization via interference region spilling. In *SIGPLAN conference on Programming language design and implementation*, May 1997.

[26] M. S. R. Gupta and T. Steele. Register allocation via clique separators. *SIGPLAN conference on Programming language design and implementation*, 25:264ñ274, 1989.

[27] R. Sethi. Complete register allocation problems. *SIAM Journal on Computing*, 4(3):226ñ248, September 1975.

[28] Y. Zhang, X. Hu, and D. Chen. Low energy register allocation beyond basic blocks, 1999.