Journal of Instruction-Level Parallelism 7 (2005) 1-31 Submitted 10/04; published 1/05

A Faster Optimal Register Allocator

Changqing Fu CQFU@TRANSMETA.COM
Transmeta Corporation
3990 Freedom Circle
Santa Clara, CA 95054 USA

Kent Wilken WILKEN@ECE.UCDAVIS.EDU
Dept. of Electrical and Computer Engineering
University of California, Davis
Davis, CA 95616 USA

David Goodwin GOODWIN@TENSILCA.COM
Tensilica Corporation
3255-6 Scott Blvd.
Santa Clara, CA 95054 USA

Abstract

Researchers have previously formulated register allocation as an integer linear program (IP) and solved the IP optimally using a general-purpose integer programming solver. Compared with the traditional graph-coloring approach, an IP-based optimal register allocator can significantly increase program performance and reduce code size because the IP formulation captures details of the register allocation problem that are abstracted away by a graph coloring allocator. However, the time to solve a register-allocation integer program can be significant, which limits the applicability of optimal register allocation. This paper presents a simplified IP formulation for optimal register allocation that can be solved much faster. The proposed approach uses several global and local analysis techniques to identify locations in a program's control-flow graph where spill and deallocation decisions are unnecessary for an optimal allocation. The corresponding unnecessary decision variables are removed from the integer program, resulting in an integer program that is much easier to solve. The proposed faster optimal register allocator is built into the GNU C Compiler and is evaluated experimentally using the SPEC CPU2000 and SPEC CPU92 integer benchmark suites targeted to a RISC processor. Compared with the basic IP formulation, the simplified formulation reduces the number of register-allocation IP decision variables by a factor of 4 and allows hard register allocation problems to be solved 200 times faster. The faster optimal register allocator optimally allocates 98% of the functions in the SPEC CPU2000 integer benchmarks within a 1024 second per-function time limit and reduces dynamic instruction count by an average of 10.6%.

1. Introduction

Register allocation is one of the most important compiler optimization phases. Effective register allocators can improve a program's performance [1], reduce a program's code-size [2] and lower a program's power consumption [3]. Basically, the register allocation problem is to assign a target processor's limited *real registers* to the unlimited *symbolic registers* used in the compiler's 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 [4] proves that global register allocation is NP-complete, and Farach and Liberatore [5] show that local register allocation is NP-complete. Various heuristics have been proposed for local [6, 7, 8, 9], and global [10, 11, 1, 12, 13, 14] register allocation.

This paper focuses on the harder global register allocation problem. The traditional global allocator is based on the graph-coloring heuristic developed by Chaitin et al. [1]. Various graph-coloring improvements have been proposed more recently [15, 16, 17, 18, 11, 12, 19, 13, 20]. These improvements rely on different heuristics to determine the placement of the spill code and the order in which symbolic registers are 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 may produce allocation overhead which significantly degrades program performance. Issues about register coalescing, procedure call conventions, and special register requirements for irregular register architecture make it more difficult to produce good allocations.

Goodwin and Wilken [21] proposed modeling register allocation as an integer linear programming (IP) problem and solving it optimally with an IP solver. They show that although optimal register allocation is NP-complete, practically it is solvable in most cases. The *Optimal Register Allocator* (ORA) proposed in [21] considers a uniform register architecture and integrates into one framework the register allocation issues of coalescing, live range splitting, rematerialization, register pairs, and caller-saved/callee-saved registers. Kong and Wilken [22] present a framework based on ORA for irregular architectures, like the x86 architecture. Naik and Palsberg [2] use integer programming to model code-size aware register allocation for embedded microprocessors. [21] shows that ORA can significantly reduce register allocation overhead compared with the graph-coloring allocator. 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 size of the solution space and the time for finding an optimal solution.

Appel and George [23] 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. Each subproblem is solved using integer programming. Although this IP-based allocator is faster, the decomposition results in an allocation which may not be optimal.

Scholz and Eckstein [24] presents an approach which models register allocation as a *partitioned boolean quadratic optimization problem* (PBQP) for an irregular DSP architecture. The model is incomplete. While register coalescing is integrated in the model, live range splitting and rematerialization issues are not considered, and spilling is handled using Chatin's spilling-everywhere approach. [24] also sacrifices allocation optimality because of the heuristic reduction rules that are used when solving the PBQP problem via dynamic programming.

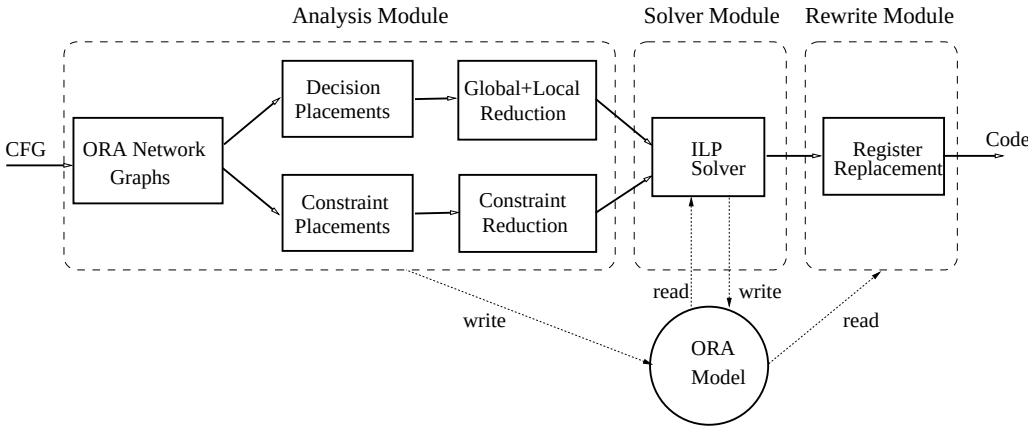

Figure 1: Faster ORA allocator framework.

This paper presents a faster optimal global register allocator based on new techniques for reducing IP model complexity. The new techniques identify and eliminate certain redundancies from the IP model while preserving the allocation's optimality. Our allocator builds on the *basic ORA model* proposed in [21]. We analyze the placement of spill decisions and register deallocation decisions produced in the program control-flow 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 efficiently 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 program size, $n^{1.2}$. The faster ORA allocator's local and global reduction techniques produce an average 4X reduction in the number of the decision variables and an average 2X reduction in the number of constraints. Optimal allocation is much faster than the basic ORA allocator, 200X faster for hard register allocation problems. For the 8492 functions in the SPEC CPU2000 and SPEC CPU92 integer benchmark suites, 98% of the functions are solved optimally.

The rest of this paper is organized as follows. Section 2 gives an overview of the faster ORA allocator. Section 3 describes the basic ORA model and the IP formulation. Section 4 shows the existence of redundant decisions in the basic ORA model. Section 5 and 6 present global and local reduction techniques for load, store and deallocation decisions. Section 7 presents various constraint reduction techniques. Section 8 shows the experimental results and Section 9 summarizes the paper's contributions.

2. 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 function's CFG to construct a set of network graphs, and determines the locations in the network graphs where various allocation decisions should be made and various allocation constraints should be enforced. The network graphs, the

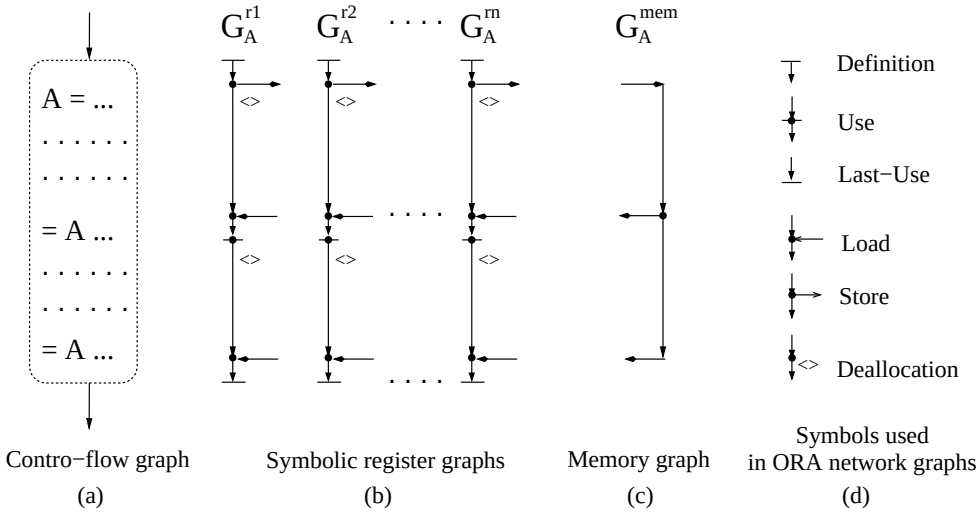

Figure 2: Control-flow graph and ORA network graphs.

allocation decisions, and the constraints comprise the basic ORA model. Local and global reduction techniques are then used to identify and eliminate unnecessary decisions and constraints to produce the simplified, faster ORA model.

The solver module constructs a 0-1 integer program [25] from the faster ORA model with an objective function for minimizing register allocation overhead. The integer program is passed to an IP solver. The IP solver determines a value of either 0 or 1 for each decision so that the constraints are satisfied and the objective function is minimized.

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

3. Background

This section gives a brief description of the basic ORA model and the IP formulation.

3.1 Basic ORA Model

ORA models register allocation as a set of network graphs: symbolic-register graphs and memory graphs. A *symbolic register graph* represents the allocation state of a symbolic register to a real register over the lifetime of the symbolic register. A *memory graph* expresses the allocation state of a symbolic register to memory over the lifetime of the symbolic register. For each symbolic register, a symbolic register graph is constructed for each real register, and one memory graph is constructed.

An ORA network graph is a directed subgraph derived from a CFG. A symbolic register graph starts at a symbolic register's definition(s) and ends at the symbolic register's last-use(s). A memory graph starts at the symbolic register's first store(s) and ends at the symbolic register's last-load(s). For example, Figure 2(a) shows a CFG where symbolic register A is defined, used and last-used in a basic block. Figure 2(b) shows the derived

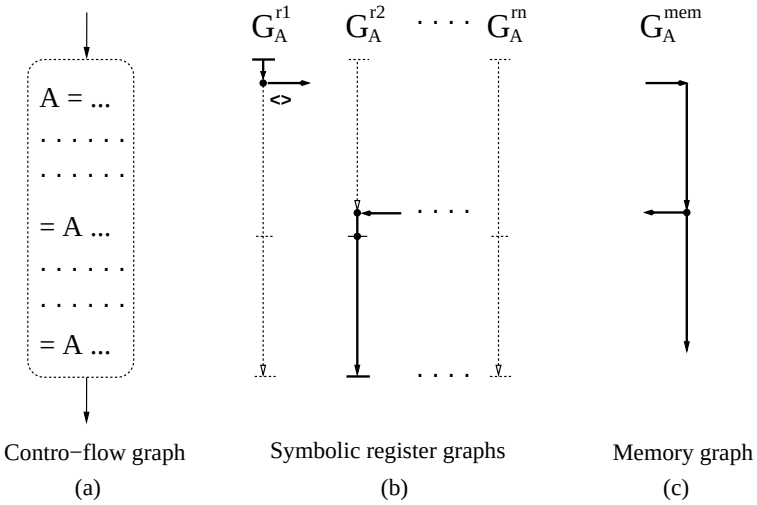

Figure 3: ORA network graphs and register allocation solution.

symbolic register graphs for A for real register $r1$ through rn, and Figure 2(c) shows the derived memory graph for A.

The edges in the ORA network graphs represent various allocation decisions and various deallocation decisions. An *allocation* decision represents whether or not the value of a symbolic register should be allocated to a real register or saved into memory. A *definition* decision, a *load* decision, or a *rematerialization* decision each represents a register allocation decision, while a *store* decision represents a memory allocation decision. A *deallocation* decision represents whether or not the value of a symbolic register should be deallocated from a real register. A *coalescing* decision represents whether the values of two symbolic registers used in a copy instruction should be allocated to a same real register. This paper focuses load, store and deallocation decisions, and the related constraints.

The vertices of an ORA network graph represent the locations where the decisions are to be made by an allocator. In order to preserve the optimality of the register allocation problem in the ORA model, ORA chooses a sufficient set of locations on the network graphs to place allocation decisions and deallocation decisions. It is proved theoretically that at least one optimal allocation solution can be derived from the set of decision locations [21].

Sufficient decision placements:
For optimal register allocation, it is sufficient to place
load decisions before every symbolic register use and at every ORA network graph merge edge,
store decisions after every symbolic register definition and at every ORA network graph diverge edge,
deallocation decisions after every definition, after every use and at every ORA symbolic-register graph diverge edge.

Figure 2(b) and Figure 2(c) show a sufficient set of placements of the load, store and deallocation decisions for symbolic register A on $A's$ symbolic register graphs and memory graph. The symbols used in the network graphs are explained in Figure 2(d).

A register allocation can easily be expressed through the network graphs. Figure 3 shows the network graphs and the highlighted graph edges to express an allocation where

A is first defined in $r1$, $r1$ is stored into memory after the definition, and then deallocated from $r1$. Before the use of A, the value of A is loaded back into $r2$ till the last-use of A.

An *optimal allocation solution*, which has minimal allocation overhead for improving a program's performance, can be derived from the network graphs by selecting a set of graph edges whose costs are minimal. The *cost* of a network graph edge is the allocation overhead of an allocation decision or an deallocation decision which is associated to the network graph edge. For a load decision, a store decision, and a rematerialization decision, the cost is equal to the estimated dynamic instruction execution count of the basic block where the graph edge resides (e.g. from program profiling). The cost of a deallocation decision is zero, and the cost of a coalescing decision is the negative of the number of dynamic cycles of the copy instruction.

3.2 IP Formulation

ORA formulates the network model as an integer-programming problem with the objective of

$$\text{min} \quad \sum_s \sum_r c^r_{sp} x^r_{sp}$$

$$\text{s.t.} \quad \sum_r x^r_{sp} \geq 1 \qquad (1) \quad \text{must-allocate}$$

$$\sum_s x^r_{sp} \leq 1 \qquad (2) \quad \text{single-symbolic}$$

$$\sum_r x^r_{sp} + x^{mem}_{sp} \geq 1 \qquad (3) \quad \text{liveness}$$

$$x^r_{sp-1} \geq x^r_{sp} \qquad (4) \quad \text{deallocation}$$

$$x^r_{sp} = x^r_{sp'} \qquad (5) \quad \text{merge}$$

$$x^r_{sp} = 0, 1$$

Where x^r_{sp} is a decision variable, which is associated with a symbolic register graph edge at location p for symbolic register s for real register r. x^{mem}_{sp} is a decision variable, which is associated with a memory graph edge. x^r_{sp} and $x^r_{sp'}$ represent allocation states on each side of a merge vertex on an ORA network graph. (1) to (5) are IP constraints to ensure a valid allocation is produced for the set of decision variables.

- *must-allocate* constraints: ensure a symbolic register must be allocated to a real register at each definition and each use.
- *single-symbolic* constraints: ensure a real register can be allocated to at most one symbolic register at any point of the program.
- *liveness* constraints: ensure the liveness of a symbolic register exists either in real registers or in memory or in both at any point of the lifetime of the symbolic register.

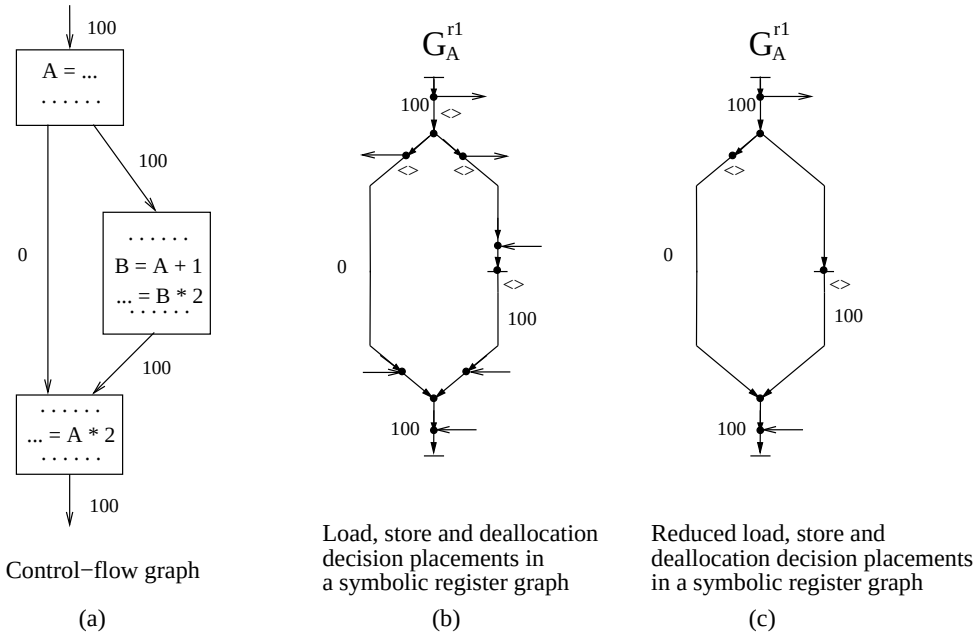

Figure 4: Placement reduction of load, store and deallocation decisions in a symbolic register graph.

- *deallocation* constraints: allow a real register to be deallocated from a symbolic register at the deallocation locations.
- *merge* constraints: ensure the allocation states of a symbolic register on all incoming graph edges at a merge vertex, where the symbolic register is alive, must be equal.

4. Redundancy

The various allocation decisions, deallocation decisions, and constraints used in the ORA model are sufficient, 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, Figure 4(a) shows a CFG, and Figure 4(b) shows the placements of load, store, and deallocation decisions in the symbolic register 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.

Load decision and the deallocation decisions placed between the definition of A and the use of A in the graph $G_A^{r_1}$ in Figure 4(b) are redundant decisions because in an optimal allocation, once A is allocated to a real register at the definition instruction, A will remain

in this real register until the use of A. The observation leads to an interval-based local reduction technique which will be described in Section 6.

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 definition 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 eliminated, 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. These observations lead to several global reduction techniques which are presented in Section 5.

Figure 4(c) shows the simplified graph $G_A^{r_1}$ for the example used in Figure 4(b).

5. 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 significantly simplify the model's complexity.

5.1 Decision Placement

The placement of the deallocation decisions at the diverge edges and the load decisions at the merge edges in a symbolic register graph can have a hierarchical structure as shown in Figure 5. 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 [26], which starts at a diverge vertex and ends at a merge vertex of the symbolic register graph. Each diverge-merge region is categorized as: a *diamond region*, a *loop region* or an *unstructured region*.

A diamond region is a two-way diverge (entry) and two-way merge (exit) SESE region; each diverge edge immediately dominates [6] only one merge edge, and the dominated merge edge immediately postdominates [6] 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* [6] connects the entry to the exit. All other SESE regions are unstructured regions. In Figure 5, 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* [6] if the unstructured regions are multi-way diverge or multi-way merge region. Figure 6 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.*

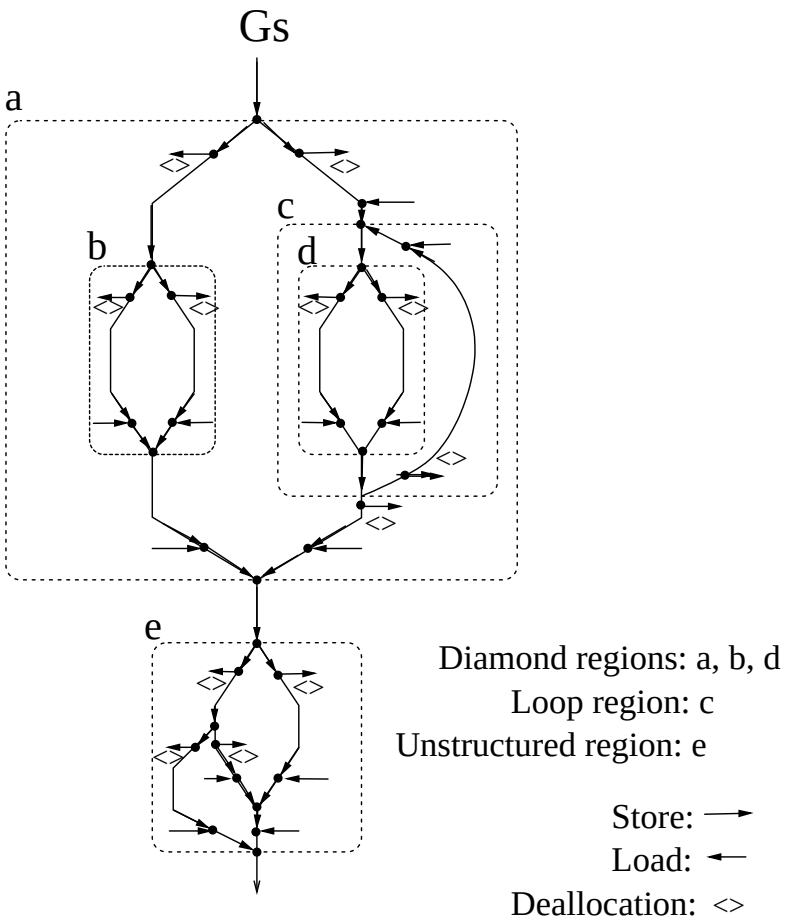

Figure 5: Decision placement hierarchy.

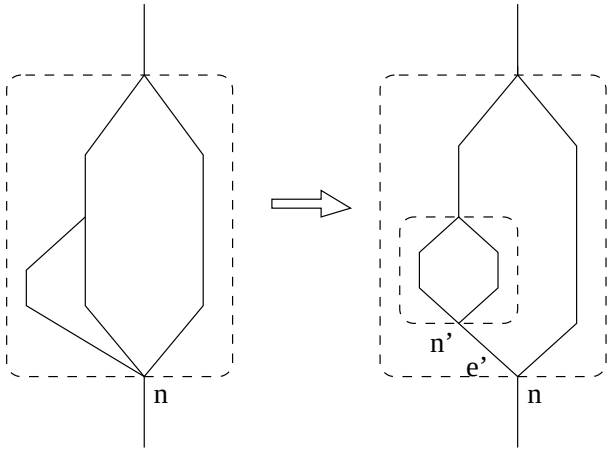

Figure 6: Node splitting for an unstructured region.

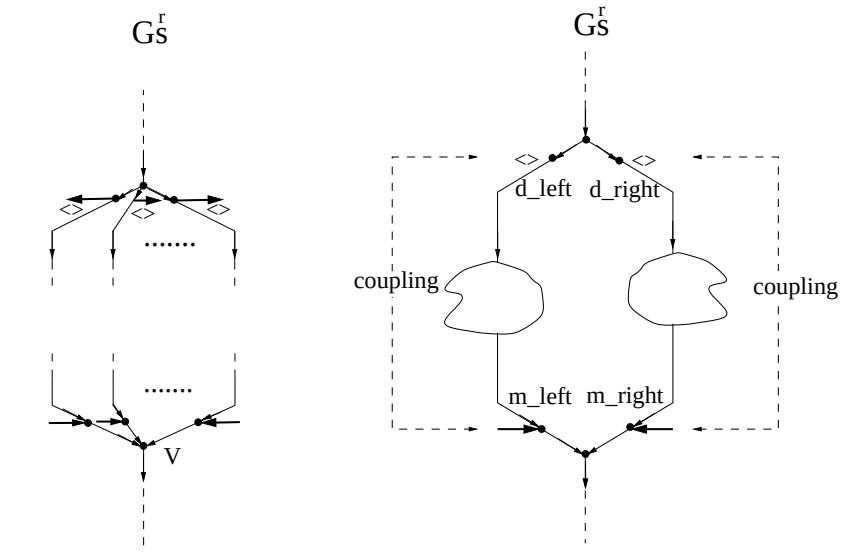

Figure 7: Decision decoupling and coupling.

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-first-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. □

5.2 Diamond Region Reductions

Diamond regions are common constructs in symbolic register graphs. 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 nullification, and asymmetric decision elimination.

5.2.1 VOID REGION COUPLING

void region A symbolic register graph region is *void* for symbolic register S if S is not redefined 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 Two decisions are *paired* if one is located on the diverge edge starting from the region entry, and the other is located on the merge edge ending at the region exit, such that the two edges are on the same side of the diamond.

Theorem 5.1 (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 a real register r on a diverge edge in a diamond region but 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 but loads S on the merge edge. S can not be in r at the merge vertex or at the diverge vertex of the region, respectively. 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.

5.2.2 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 8(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 the 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 defined 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 5.2 (Symmetric decision selection) *For optimal register allocation, if the number of 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.*

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 defined 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 defined or used inside the region. It is

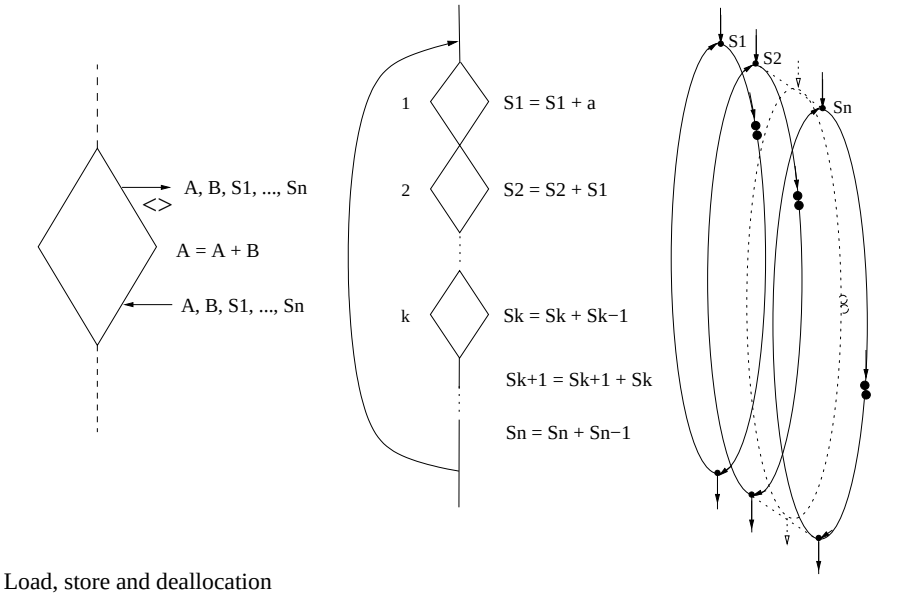

Figure 8: Symmetric solutions in diamond regions.

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* [25] iterations for the IP solver to find an optimal integer solution. Figure 8(b) shows an example of a control-flow graph which illustrates the huge reduction in symmetric solutions the technique can produce. Figure 8(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}.

5.2.3 JUMP-EDGE NULLIFICATION

jump edge A *jump edge* is both a merge edge and a diverge edge in a symbolic register graph.

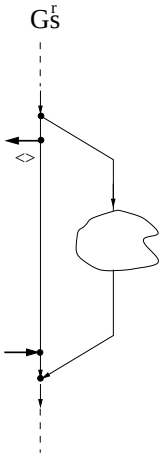

Figure 9: Jump-edge nullification.

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 5.3 (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 5.1, 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 definition 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.

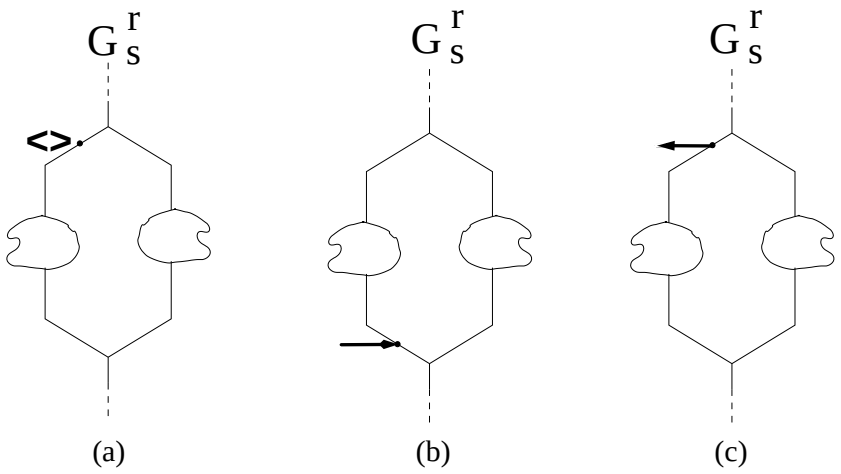

Figure 10: Asymmetric decisions.

c) Suppose S is loaded into real register r on the jump edge. Without a definition 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. □

5.2.4 ASYMMETRIC DECISION ELIMINATION

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

Figure 10(a)-10(c) show the asymmetric deallocation, asymmetric load and asymmetric store decisions. Under certain conditions, an asymmetric decision is unnecessary for optimal register allocation.

Theorem 5.4 (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. □

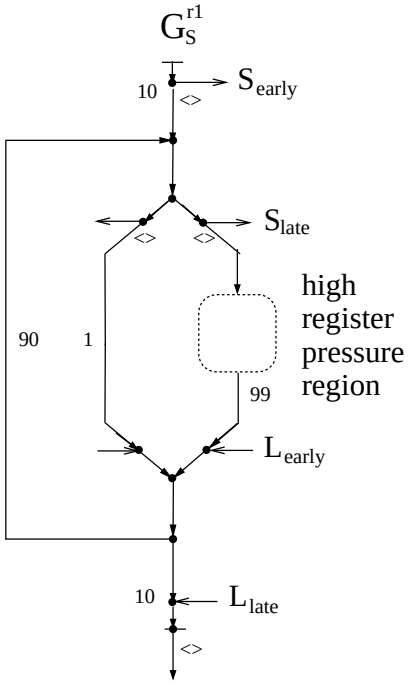

Figure 11: Cost-guided load/store reductions.

5.3 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.

5.4 Cost-Guided Reduction

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

Theorem 5.5 (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 first 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 11 shows a portion of a control-flow 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.

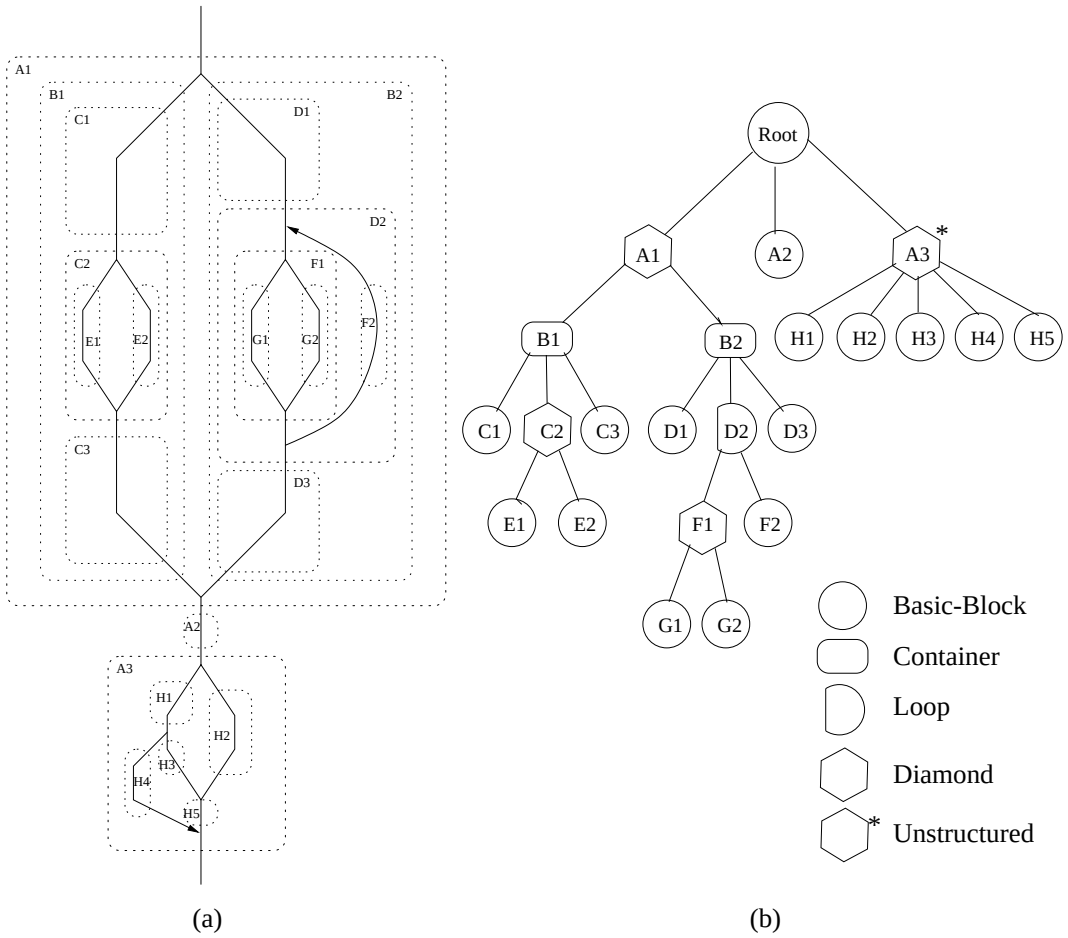

Figure 12: SESE region hierarchy and program structure tree.

Symmetrically, a load placed at the L_{early} as shown in the Figure 11 is redundant based on the following theorem.

Theorem 5.6 (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) the late location postdominates the early location, 2) no use, last-use and definition of the symbolic register S is reachable from a load at the earlier location to a load at the later location, 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 5.5. □

5.5 Building the Hierarchy

The hierarchical reduction approach starts by building the *program structure tree* (PST)[26], a hierarchical representation of program structure based on *single-entry single-exit* (SESE) regions of the control flow graph. Figure 12(a) shows a program control-flow graph with

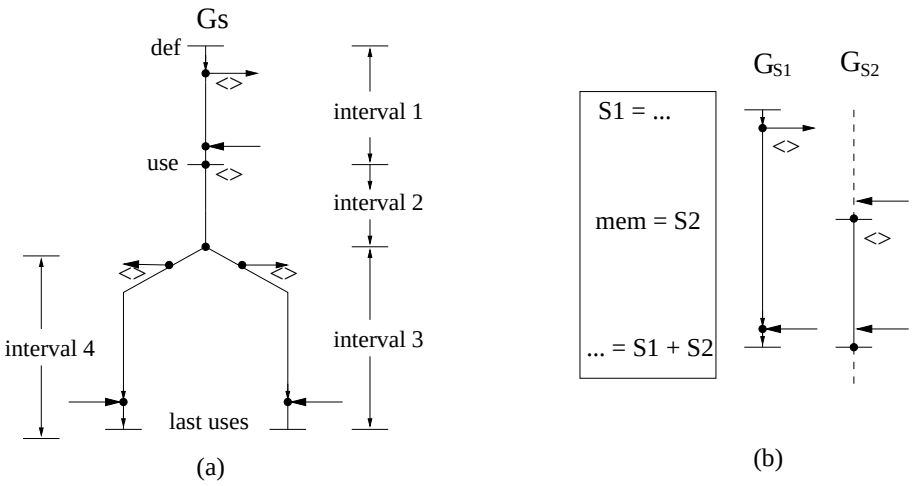

Figure 13: (a) Def-use intervals and decision placements, (b) interval-based reduction.

its SESE regions marked. Figure 12(b) is the region-based PST. There are five 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 first during the DFS traversal of the tree in post-order. Data flow analysis is accomplished for basic block regions by scanning each instruction. The data flow 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.

6. 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.

def-use interval A def-use interval is a segment of the live-range graph of a symbolic register S that starts at a definition 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 13(a) shows the four def-use intervals of symbolic register S.

Theorem 6.1 *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 13(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.

7. Constraint Reduction

Constraint reductions are based upon decision reductions. In the basic ORA model, a sufficient set of constraints are generated to ensure a valid allocation is produced for the set of decisions [21]. After redundant decisions have been eliminated, some constraints may not exist, or some constraints may be automatically satisfied by other constraints. Those constraints can be eliminated from the ORA model.

> **Sufficient constraint placements:**
> For optimal register allocation, it is sufficient to construct
> *deallocation constraints* after every definition, after every use and at every ORA symbolic register graph diverge edge,
> *must-allocate constraints* after each definition and before each use of a symbolic register,
> *single-symbolic constraints* after each definition and after each load for a real register,
> *liveness constraints* after each deallocation decision.

Deallocation constraints A deallocation constraint is constructed as follows to allow a real register to be deallocated from a symbolic register at the deallocation decision location:

$$x^r_{sp-1} \geq x^r_{sp} \tag{1}$$

Where x^r_{sp-1} represents the allocation state of real register r to symbolic register s before the deallocation constraint location p, and x^r_{sp} represents the allocation state after p.

The deallocation decision variable is implicitly expressed in the constraint. When x^r_{sp-1} is 1, and x^r_{sp} is 0, the deallocation action is taken. When the deallocation decision is eliminated, the decision variables x^r_{sp-1} and x^r_{sp} are reduced to one variable. The deallocation constraint is no long necessary.

Must-allocate constraint A must-allocate constraint is constructed as follows to ensure a symbolic register must be allocated to a real register at each definition and each use:

$$\sum_r x^r_{sp} \geq 1 \tag{2}$$

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 first constraint. The second must-allocate constraint can be automatically satisfied by the first must-allocate constraint.

Single-symbolic constraint A single-symbolic constraint is constructed as follows to ensure a real register can be allocated to at most one symbolic register:

$$\sum_{s} x_{sp}^{r} \leq 1 \tag{3}$$

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

Liveness constraint A liveness constraint is constructed as follows to ensure the liveness of a symbolic register always exists during the lifetime of the symbolic register:

$$\sum_{r} x_{sp}^{r} + x_{sp}^{mem} \geq 1 \tag{4}$$

Where x_{sp}^{mem} represents the allocation state of symbolic register s to memory at the liveness constraint location p.

For optimal register allocation, if no deallocation decision exists between two adjacent liveness constraints for a symbolic register, then the second liveness constraint is redundant. The rationale of the elimination is the same as for the must-allocate constraint reduction.

8. Experimental Results

This section experimentally evaluates the faster ORA allocator in terms of allocation time, integer program size and the amount of spill code produced. The results for the faster ORA allocator are compared with the results for the basic ORA allocator published in [21]. The comparison shows that the faster ORA allocator solves hard register allocation problems 200 times faster than the basic ORA allocator. The faster ORA allocator achieves optimal allocation for 98% of the functions in the SPEC CPU2000 integer benchmark suite and reduces dynamic instruction count by an average of 10.6%.

8.1 Methodology

The faster ORA allocator, the basic ORA allocator and a Chaitin-style graph-coloring allocator are all built into the GNU C Compiler (GCC) version 2.5.7 and are evaluated experimentally using the SPEC CPU2000 [27] and SPEC CPU92 [28] integer benchmarks. Compilations are targeted to the HP PA-RISC architecture using the 24 real registers that

GCC has available for allocation for the PA-RISC architecture[1]. Integer programs are solved using CPLEX 9.0 [29], a commercial integer program solver, running on a 1.5GHz Itanium 2 processor with 4Gbytes of main memory.

The faster ORA allocator is evaluated first using the 2399 functions from the SPEC CPU92 integer benchmark suite. The use of SPEC CPU92 allows a direct comparison of allocation time and integer program size with the results for the basic ORA allocator published in [21]. The faster ORA allocator is next evaluated for allocation time and integer program size using 6093 functions from the SPEC CPU200 integer benchmark suite. SPEC CPU2000 is then used to compare the spill code produced by the faster ORA allocator with the spill code produced by a Chaitin-style graph-coloring allocator. The graph-coloring allocator is based on the optimistic allocator proposed by Briggs, Cooper and Torzcon [17]. Our implementation of the optimistic allocator includes rematerialization and coalescing to reduce spill code overhead. An execution profile is used to determine spill costs for the two ORA allocators and for the graph coloring allocator.

8.2 Allocation Time

The faster ORA allocator is first evaluated for allocation time using the 2399 functions from the SPEC CPU92 integer benchmarks. SPEC CPU92 is compiled using the faster ORA allocator with allocation time limits varying by powers of 2 from 1 to 1024 seconds. The integer programs are solved using CPLEX 9.0 running on the Itanium 2. The number of functions that are optimally allocated at each time limit is shown in the first set of bars in Figure 14, labeled `Faster ORA`. The measured allocation time includes the time for all aspects of optimal register allocation, including program analysis time, integer-program solution time and the time to rewrite the intermediate representation. At a time limit of one second, 2170 out of 2399 (90%) of the functions are optimally allocated by Faster ORA. At a time limit of 1024 seconds 2374 (99%) of the functions are optimally allocated. Prior to the results in [21] optimal register allocation was considered an intractable problem. The Faster ORA results in Figure 14 show that optimal register allocation is tractable: nearly all functions can be optimally allocated in a time scale of seconds or minutes.

The second set of bars in Figure 14 labeled `ORA 2004` shows the number of functions allocated optimally using the basic ORA allocator with the integer programs solved by CPLEX 9.0 running in the Itanium 2. These data allow the faster ORA allocator to be compared with the basic ORA allocator to determine the speedup in register allocation time due to the improved integer-program model. The number of functions solved by the ORA 2004 allocator at a time limit of 1024 seconds (2304 functions) is used as a reference for the allocation-time performance of the ORA 2004 allocator. From Figure 14 using interpolation the faster ORA allocator is seen to solve the same number of functions in a time limit of about 5 seconds. This means the hardest problems solved by the ORA 2004 allocator, problems solving at close to the 1024 second time limit, are now solving 200 times faster using Faster ORA. Figure 14 also shows that Faster ORA can optimally allocate about 75%

1. of the 32 registers in the PA-RISC architecture, the compiler reserves seven registers for special purposes (stack pointer, frame pointer, global data pointer, interrupt handling, etc.), and one register is hardwired to 0.

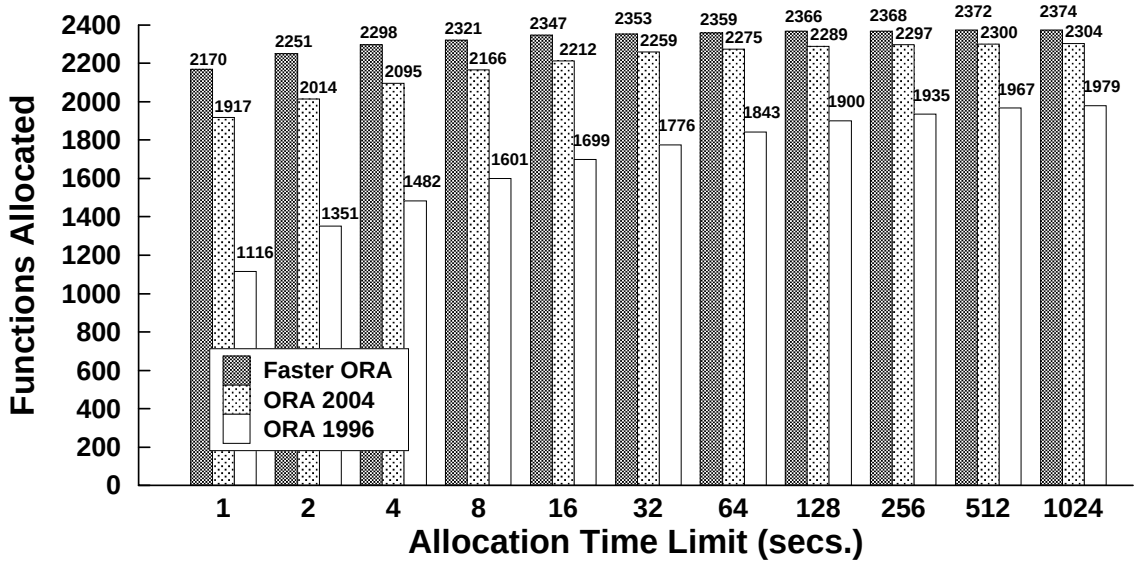

Figure 14: SPEC CPU92 Integer Benchmark functions allocated by Faster ORA, allocated by the basic ORA allocator using the current processor and solver, and allocated by the basic ORA allocator using the older processor and solver in [21].

of the functions that cannot be optimally allocated by ORA 2004 at a time limit of 1024 seconds.

The performance of an optimal register allocator is the product of the register-allocation integer-program model, processor performance and solver performance. Readers of this journal will be well aware of the rapid increase in processor performance that has occurred over many decades. However readers will be generally unaware that integer-program solver performance has also been increasing rapidly over many decades [30]. In the next paragraph we estimate the register allocation speedup that improved solver technology has provided since the initial ORA publication [21].

The third set of bars in Figure 14 labeled `ORA 1996` shows the number of functions allocated optimally at each time limit by the basic ORA allocator as reported in [21]. These data were produced using a processor and solver circa 1996. The ORA 1996 data can be used to estimate the speedup in register allocation time due to advances in processor and solver technology over the past eight years. The number of functions solved by the ORA 1996 allocator at a time limit of 1024 seconds (1979 functions) is used as a reference for the allocation-time performance of the ORA 1996 allocator. From Figure 14 using interpolation the ORA 2004 allocator is seen to solve the same number of functions in a time limit of about 1.6 seconds. Thus the hardest problems solved by the ORA 1996 allocator are solved 600 times faster by the ORA 2004 allocator. From published SPEC benchmark data the floating point performance of the newer processor is about 75 times that of the older processor, which suggests that the newer solver contributes a speedup of about 8x.

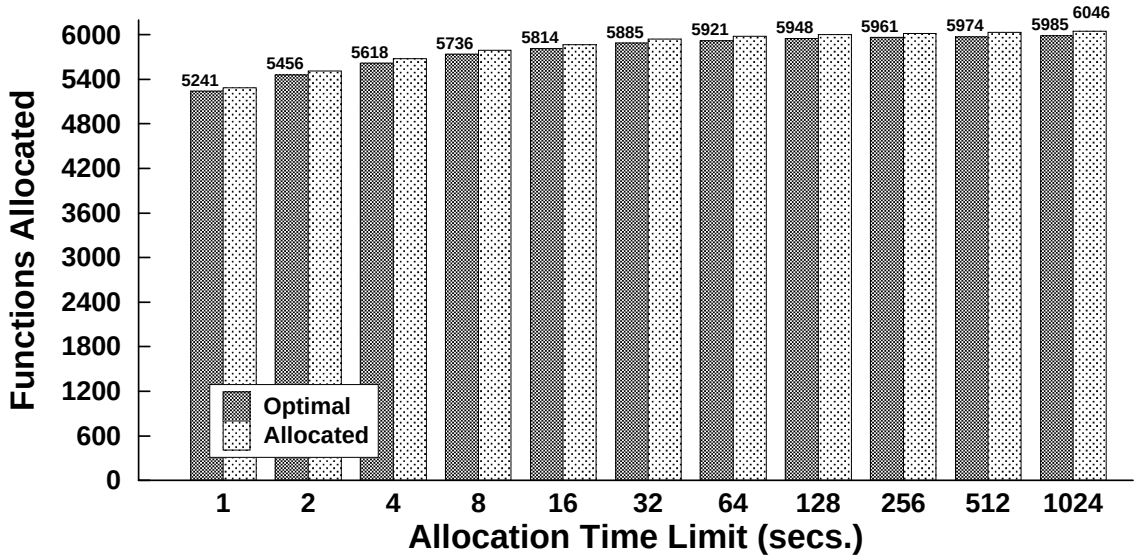

Figure 15: SPEC CPU2000 Integer Benchmark functions allocated and allocated optimally by Faster ORA at various time limits, out of a total of 6093 functions.

Comparing the performance of Faster ORA with ORA 1996, Faster ORA is faster by (200 x 600) or about 100,000x, with about 100x coming from the improved integer-program model proposed here, 10x coming from a faster solver, and 100x coming from a faster processor. As each of these three technologies improves, optimal register allocation performance benefits from a multiplicative effect, so that future increases in optimal register allocation performance will also likely be prodigious, and optimal register allocation will become increasingly attractive.

Allocation time is next evaluated for Faster ORA using 6093 functions from the SPEC CPU2000 integer benchmarks. SPEC CPU2000 is compiled using the faster ORA allocator with allocation time limits varying from 1 to 1024 seconds. The results from this compilation are shown in Figure 15. The first set of bars labeled `Optimal` shows the number of functions that are allocated optimally at each time limit. The second set of bars labeled `Allocated` shows the total number of functions that are allocated by the Faster ORA allocator. `Allocated` includes functions that are allocated optimally and functions for which the integer-program solver found an allocation and either could not prove the allocation was optimal or had not found an optimal allocation when time expired. Bottom-up examination of the solver's behavior shows that for functions that are allocated but not optimal, the allocation is nearly always optimal (just not yet proven optimal) and if not optimal the allocation is usually very close to optimal.

At a time limit of one second 5241 out of 6093 (86%) of the functions are optimally allocated by Faster ORA. At a time limit of 1024 seconds, 5985 out of 6093 (98%) of the functions are optimally allocated by Faster ORA. These results are very similar to the results for SPEC CPU92 (90% at one second and 99% at 1024 seconds). Optimal

allocation percentages are a bit higher for SPEC CPU92 largely because the suite includes the application `xlisp` which consists of a large number of very small functions, 100% of which are allocated optimally in an average of 0.3 seconds each.

For the 108 functions that are not allocated optimally at 1024 seconds, 61 are allocated, bringing the total functions allocated either optimally or near optimally to 99%.

Table 1 shows the number of functions allocated (column `Solved`) and optimally allocated (column `Optimal`) at a time limit of 1024 seconds for each application in the SPEC CPU2000 integer benchmark suite. The figure shows there is at least one function in each application that cannot be optimally allocated. Column `Relaxed` shows the number of functions for which the solver has found a non-integer solution to the integer program (has solved a *relaxation* of the problem). Solving the relaxed problem is an important first step toward finding an integer solution (finding an allocation). For the functions in this experiment that are allocated, the time to an allocation is always less than 6 times the time to solving the relaxed problem. Thus the 35 functions that are `Relaxed` but not `Solved` could be considered close to being allocated, and would likely be allocated by increasing the time limit by 6x.

SPECint2000	Functions				
Benchmark	Attempted	Relax.	Solved	Optimal	Optimal/Attempted
gzip	108	108	108	107	0.99
vpr	270	270	270	269	0.99
gcc	2227	2215	2201	2182	0.98
mcf	26	26	26	25	0.96
crafty	65	65	65	63	0.97
parser	323	323	322	321	0.99
perlbmk	1061	1061	1054	1051	0.99
gap	851	851	847	827	0.97
vortex	922	922	921	920	0.99
bzip2	71	71	71	67	0.94
twolf	169	169	161	153	0.91
Total	6093	6081	6046	5985	0.98

Table 1: Number of functions solved by the Faster ORA allocator with a solver time limit of 1024 seconds for SPEC CPU2000 integer benchmarks.

Figure 16 shows Faster ORA's allocation time vs. function size for the 5985 SPEC CPU2000 functions that are allocated optimally. As shown in the figure, the solution time complexity is bounded by n^3. In contrast the solution time complexity for the basic ORA allocator is bounded by n^4 [21]. As a point of reference, a Chaitin-style graph-coloring allocator exhibits $nlog(n)$ run-time complexity [10].

8.3 Integer Program Size

This subsection presents data on the size of the integer programs produced by the faster ORA allocator and shows how the size compares with that produced by the basic ORA allo-

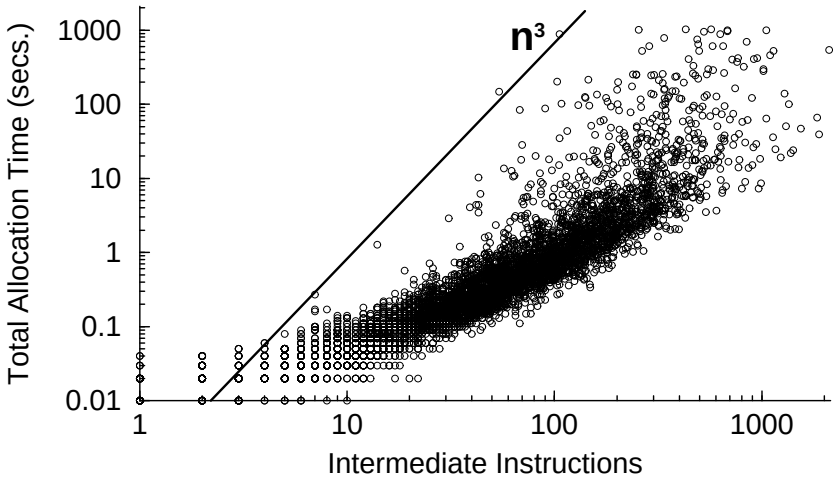

Figure 16: Faster ORA's allocation time vs. function size for the SPEC CPU2000 integer benchmarks.

cator. Table 2 shows IP decision-variable and constraint counts for the faster ORA allocator and the basic ORA allocator for each application in the SPEC CPU92 integer benchmark suite. Faster ORA's local and the global reduction techniques produce an average 4x reduction in the number of the decision variables and an average 2x reduction in the number of constraints, with the reduction being relatively uniform across all the applications.

SPECint92	Decisions			Constraints		
Benchmark	ORA	FORA	Reduction(%)	ORA	FORA	Reduction(%)
cc1	27075216	6855297	74.67	12883176	6460324	49.84
compress	229037	65316	71.48	117930	60990	48.26
eqntott	734783	195509	73.33	364888	182238	48.73
espresso	5910278	1467480	75.13	2821226	1348803	52.13
sc	2172658	586864	72.99	1128408	565526	49.82
xlisp	974359	243235	75.98	512704	212684	58.67
Total	37096331	9413701	74.64	17828332	8830565	50.48

Table 2: Number of decision variables and constraints produced by basic ORA allocator and by Faster ORA (FORA).

Figure 17 plots the constraint count vs. function size produced by Faster ORA for each of the 2399 SPEC CPU92 functions, and indicates whether a function is solved optimally, is allocated but not optimally, or is not allocated. The superimposed line shows the near-linear growth, $n^{1.2}$, in the number of constraints relative to function size. This constraint growth is slightly lower than the $n^{1.3}$ constraint growth seen for the same set of functions using the basic ORA allocator [21], due to Faster ORA's constraint reduction techniques. Figure 17 also yields information on the size of functions that are solved or not solved. The

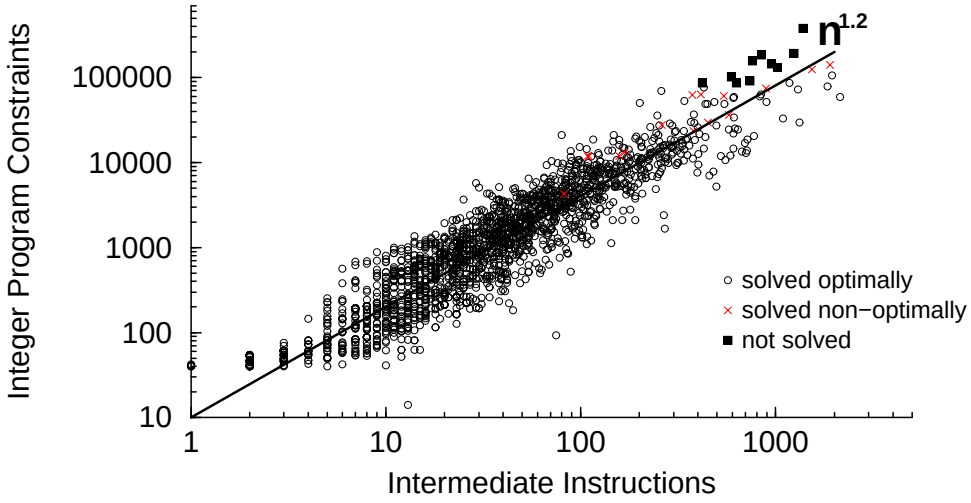

Figure 17: Faster ORA's integer program size vs. function size for the SPEC CPU92 integer benchmarks.

largest function solved optimally has about 2000 instructions and about 100,000 constraints, and the smallest function that is not optimally solved has about 90 instructions and 4000 constraints. The smallest function not allocated has about 400 instructions. Also, there is a relatively well-defined parting line at about 100,000 constraints between functions that are solved and not solved.

Figure 18 plots constraint count vs. function size produced by Faster ORA for each of the 6093 SPEC CPU2000 functions. The results are similar to those for SPEC CPU92: largest function solved optimally 2000 instructions, smallest function not solved optimally 80 instructions, smallest function not allocated 400 instructions, solved/not-solved parting line at about 100,000 constraints.

Figure 19 shows the *first solution time* vs. constraint size for Faster ORA for the 6093 SPEC CPU2000 functions. The first solution time is the amount time the solver takes to find its first allocation. After the first allocation is found the solver tries to find a better allocation or prove that the first allocation is optimal. In practice the first allocation is almost always optimal, and if not is very close to optimal. The superimposed trend line shows that the first solution time grows as $n^{2.5}$ in the number of constraints. As noted above, Figure 17 and Figure 18 show there is a relatively distinct parting line at about 100,000 constraints between the functions that are solved and not solved. These two figures also show that the largest functions have several hundred thousand constraints, with the largest SPEC CPU2000 function having about 700,000 constraints. The trend line in Figure 19 suggests that by increasing the time limit by a factor of $7^{2.5}$ (a factor of about 100) the solved/not-solved parting line would move up to the level of the largest function, and all functions might be allocated. While spending 100,000 seconds (about one day) allocating one function is not reasonable, future order-of-magnitude advances in modeling, solver and

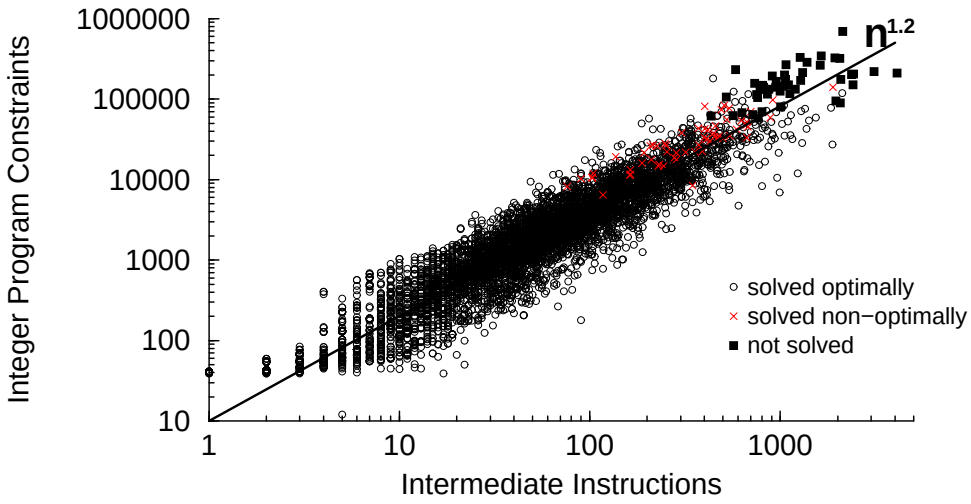

Figure 18: Faster ORA's integer program size vs. function size for the SPEC CPU2000 integer benchmarks.

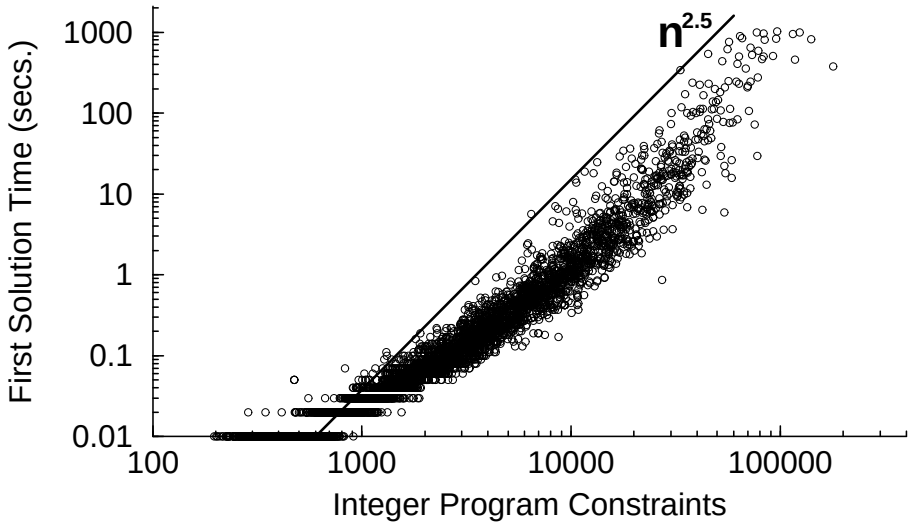

Figure 19: Faster ORA's integer program size vs. function size for the SPEC CPU2000 integer benchmarks.

processor technology may result in allocation times that are reasonable even for the very largest functions.

8.4 Register Allocation Quality

The quality of the register allocations produced by the Faster ORA allocator at a time limit of 1024 seconds is compared experimentally with that produced by the graph coloring allocator for the SPEC CPU2000 integer benchmark suite. Table 3 shows the total dynamic instruction count for each SPEC CPU2000 application using the graph coloring allocator and shows the dynamic instruction count reduction produced by Faster ORA. The reductions in dynamic instruction count are significant, ranging from 5% to 19% per allocation with an average of 10.6%.

SPEC CPU2000 Application	Applications's Total Dynamic Instructions	Dynamic Instruction Reduction Using Faster ORA
gzip	259531M	34785M (13.4%)
vpr	1676M	116M (6.9%)
gcc	54732M	6615M (12.1%)
mcf	51581M	3850M (7.5%)
crafty	207667M	1111M (5.3%)
parser	273045M	44339M (16.2%)
perlbmk	300337M	16222M (5.4%)
gap	48M	7M (14.6%)
vortex	186009M	35824M (19.2%)
bzip2	88956M	7021M (7.9%)
twolf	258357M	22223M (8.6%)

Table 3: Dynamic spill-code saved using the faster ORA allocator at a 1024 second time limit

Figure 20 shows the count of dynamic spill instructions for Faster ORA and for the graph coloring allocator aggregated across all of the SPEC CPU2000 applications. Counts are broken out by spill-code type: spill stores, spill loads, copies eliminated, and rematerialization. The copy component is negative because it reflects copies that are removed. Faster ORA reduces store spills by 1.7x, reduces load spills by 3.3x, reduces rematerial instructions by 5.8x, and increase eliminated copies by 2.2x. Overall, the graph coloring allocator causes a large net increase in the number of dynamic instructions in the code while the faster ORA allocator produces a small net decrease.

9. Conclusions

This paper presents a faster optimal register allocator that starts with the integer-program-based optimal register allocator (ORA) proposed in [21] and applies a set of reduction techniques that simplify the integer program, resulting in an integer program that solves much faster. Global and local analysis techniques are used to identify locations in the control flow graph where the spill and deallocate decisions used in the basic ORA integer-program model are not necessary for maintaining an optimal register allocation, and thus the corresponding decisions can be eliminated. Experimental results for the SPEC CPU2000 integer benchmarks show that the faster ORA allocator has four times fewer integer-program

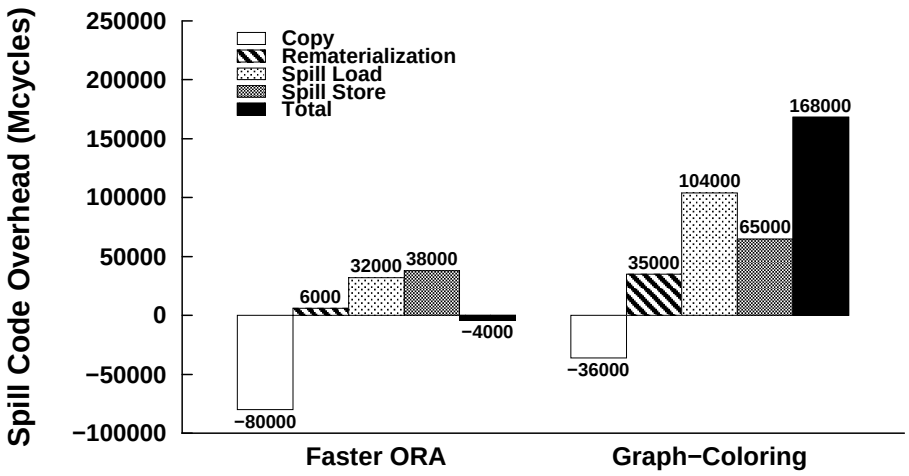

Figure 20: Dynamic spill code components for the SPEC CPU2000 functions allocated by the faster ORA allocator and by the optimistic graph-coloring allocator

decision variables, has two times fewer IP constraints, and can optimally solve hard register allocation problems 200 times faster compared with the basic ORA allocator. The faster ORA allocator is able to optimally allocate 98% of the functions in the SPEC CPU2000 integer benchmarks within a per-function time limit of 1024 seconds. Compared with a Chaitin-style graph coloring allocator, the faster ORA allocator reduces dynamic instruction count by an average of 10.6% for the SPEC CPU2000 integer benchmarks.

We believe the results presented in this paper are interesting and important for several reasons. The faster ORA allocator can be useful as a production allocator for applications where the best possible code quality is critical, such as an embedded-system application. Although the emphasis in this paper is on the performance component of code quality, a register allocation that is optimal for dynamic spill count also tends to have significantly less static spill code than what is produced by a graph coloring allocator. Code size is an important code-quality component for embedded systems. Reducing dynamic spill code can also significantly reduce power consumption because of the relatively high energy cost of accessing memory. Power consumption is another important embedded-system code-quality component.

Near-optimal IP-based allocators have been proposed that are derived from the basic ORA allocator, e.g., [21, 23]. These near-optimal formulations will also gain a significant speedup by using the IP reduction techniques proposed here. An IP allocator for an irregular architecture such as [22] will also experience a significant speedup.

An optimal register allocator is useful as a tool for studying register allocation problems to gain insights into register allocation. An optimal allocator can be used as a tool to understand how and why heuristic allocators produce suboptimal allocations, possibly leading to better heuristics. An optimal allocator can be used as a tool to identify those register allocation problems that are hard to solve when formulated as an integer program. Understanding the nature of these hard problems may lead to better integer program formulations

and to better insights into the register allocation problem. The faster ORA allocator is a sharper register-allocation tool for these purposes.

The proposed reduction techniques identify spill locations that are unnecessary for an optimal register allocation. Graph coloring allocators may be able to take advantage of these unnecessary spill locations to make better live-range splitting decisions.

The results in this paper show that order of magnitude speedup is possible for optimal register allocation through improved IP models and improved solvers, on top of processor performance advances. We believe that future order of magnitude speedup is likely. Figures 18 shows that for a function of a given code size, there is a consistent difference of more than a factor of 10 between the constraint count for the smallest IP of that code size and the constraint count for the largest IP of that code size. Gaining a good understanding of the reason for this large difference in integer program size will likely result in a much simpler and faster-solving integer program model.

Acknowledgements

This research is supported by the National Science Foundation under grant CCR-9711676. The authors are grateful to Timothy Kong of PathScale for his help to the experimental setup. The authors would like to thank the anonymous reviewers for their constructive comments that helped improved the final paper.

References

[1] G. Chatin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein, "Register allocation via coloring," *Computer Languages*, vol. 6, pp. 47–57, 1981.

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

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

[4] R. Sethi, "Complete register allocation problems," *SIAM Journal on Computing*, vol. 4, pp. 226–248, September 1975.

[5] 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.

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

[7] C. W. Fraser and D. R. Hanson, "Simple register spilling in a retargetable compiler," *Software-Practice and Experience*, vol. 22, pp. 85–99, January 1992.

[8] R. A. Freiburghouse, "Register allocation via usage counts," *CACM*, vol. 17, pp. 638–642, 1974.

[9] E. Lowry and C. Medlock, "Object code optimization," *CACM*, vol. 12, pp. 13–22, 1969.

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

[11] D. Callahan and B. Koblenz, "Register allocation via hierarchical graph coloring," in *Proceedings of Conference on Programming Language Design and Implementation*, pp. 192–203, June 1991.

[12] F. Chow and J. Hennessey, "The priority-based coloring approach to register allocation," *ACM Trans. on Programming Languages and Systems*, vol. 12, pp. 501–536, October 1990.

[13] 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.

[14] M. S. R. Gupta and T. Steele, "Register allocation via clique separators," *SIGPLAN conference on Programming language design and implementation*, vol. 25, pp. 264–274, 1989.

[15] 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*, vol. 24, pp. 258–263, June 1989.

[16] M. P. Bivens, "Incremental register reallocation," *Software Practice and Experience*, vol. 20, pp. 1015–1047, October 1990.

[17] P. Briggs, K. Cooper, and L. Torczon, "Improvements to graph coloring register allocation," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 428–455, May 1994.

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

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

[20] P. D. Peter Bergner and etc., "Spill code minimization via interference region spilling," in *SIGPLAN conference on Programming language design and implementation*, May 1997.

[21] D. Goodwin and K. Wilken, "Optimal and near-optimal global register allocation using 0-1 integer programming," *Software Practice and Experience*, vol. 26, pp. 929–965, August 1996.

[22] T. Kong and K. D. Wilken, "Precise register allocation for irregular architectures," in *International Symposium on Microarchitecture*, pp. 297–307, 1998.

[23] 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.

[24] B. Scholz and E. Eckstein, "Register allocation for irregular architectures," in *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pp. 139–148, ACM Press, 2002.

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

[26] R. Johnson, D. Pearson, and K. Pingali, "The program structure tree: Computing control regions in linear time," *ACM SIGPLAN Notices*, vol. 29, pp. 171–185, 1994.

[27] J. Henning, "Spec cpu2000: Measuring cpu performance in the new millennium," *Computer*, vol. 33, pp. 28–35, July 2000.

[28] K. Dixit, "New CPU benchmark suites from SPEC," in *Digest of Papers Compcon, Spring 1992*, pp. 305–310, IEEE, 1992.

[29] CPLEX Optimization, Inc., *Using the CPLEX Callable Library*. CPLEX: Incline Village, US (NV), 1994.

[30] G. Nemhauser, "The age of optimization: Solving large-scale real-world problems," *Operations Research*, vol. 42, pp. 5–13, Jan.-Feb. 1994.