Register Allocation over the Program Dependence Graph *

Cindy Norris
norris@udel.edu

Lori. L. Pollock
pollock@udel.edu

Department of Computer and Information Sciences
University of Delaware
Newark, DE, 19716
(302) 831-1953

Abstract

This paper describes **RAP**, *a* **R***egister* **A***llocator that allocates registers over the* **P***rogram Dependence Graph (PDG) representation of a program in a hierarchical manner. The PDG program representation has been used successfully for scalar optimizations, the detection and improvement of parallelism for vector machines, multiple processor machines, and machines that exhibit instruction level parallelism, as well as debugging, the integration of different versions of a program, and translation of imperative programs for data flow machines. By basing register allocation on the PDG, the register allocation phase may be more easily integrated and intertwined with other optimization analyses and transformations. In addition, the advantages of a hierarchical approach to global register allocation can be attained without constructing an additional structure used solely for register allocation. Our experimental results have shown that on average, code allocated registers via RAP executed 2.7% faster than code allocated registers via a standard global register allocator.*

1 Introduction

The Program Dependence Graph (PDG) representation of a program has been used successfully as the basis for various scalar optimizations [16, 25, 23] as well as for detecting and improving parallelization for vector machines [28, 5], multiple processor machines [30, 3], and architectures that exhibit instruction level parallelism [19, 7, 2]. Variations of the PDG have also been used for debugging and integrating different versions of a program via program slicing [29, 1, 18, 22, 21, 26], and to enable translation of imperative programs for data flow machines and demand driven graph reducers [4]. This paper describes a register allocator based on the PDG and presents experimental evidence that shows how it usually outperforms a traditional global graph coloring register allocator.

Our overall research interests have been directed toward the problem of the sometimes conflicting goals of various tasks of optimizing and parallelizing compilers. In particular, we have been investigating ways of providing cooperation between a register allocator and an instruction scheduler [24]. Our original motivation for building a register allocator based on the PDG was to have a common program representation for both the register allocator and global instruction scheduler, as a first step towards integrating these two phases. The PDG provides a natural representation for scheduling across basic block boundaries, and thus several global instruction scheduling methods have been expressed as transformations over the PDG [6, 19, 2], for instance, region scheduling [19] and software pipelining [2]. However, we could not find any indication in the literature of a register allocation technique that was based on the PDG.

The widespread use of the PDG representation can be attributed to the fact that it represents both data and control dependences in a single program representation and it expresses only the essential partial ordering of statements and predicates in a program that must be followed in order to preserve the semantics of the original program. However, actual improvements in allocation over traditional global graph coloring register allocation schemes can be obtained by exploiting the hierarchical nature of the PDG. The hierarchical organization of the regions of a program that have the same set of control dependences allows the register allocator to process regions of the program separately, and more importantly, in a manner based on the program structure. When it is determined that a variable needs to be spilled within a region, it may be possible to spill

*This work was partially supported by NSF under grant CCR-9300212.

SIGPLAN 94-6/94 Orlando, Florida USA
© 1994 ACM 0-89791-662-x/94/0006..$3.50

the variable only locally, without spilling it throughout the program. For example, a variable may be assigned to register R1 in one region, register R2 in another region, and spilled in another region. Like other approaches [12, 20], a space savings can be obtained by performing register allocation for code segments separately, which causes smaller interference graphs to be constructed than one interference graph for the whole program. The PDG depicts the hierarchical control regions of a program making it easier to place spill code in the less frequently executed regions of the program than standard global register allocation techniques.

This paper describes *RAP*, a *R*egister *A*llocator that allocates registers over the *P*DG representation of a program in a hierarchical manner. The register allocation of each region is performed in a bottom up pass over the PDG, using an enhanced version of Chaitin's graph coloring algorithm for each region [14, 9]. On a top down pass, RAP attempts to move loads and stores out of loops. A third pass over the PDG removes unnecessary loads and stores created by the hierarchical register allocation process. Our experimental results have demonstrated that on average, code allocated registers via RAP executed 2.7% faster than code allocated registers via a standard global register allocator.

The quality of the allocation by RAP is due primarily to the initial focus on smaller segments (regions) of the code, and then building upon these allocations to obtain allocations for larger segments (parent regions) of the code. Several other researchers have presented register allocation methods directed specifically toward improving the overall allocation by considering both local register needs and global register usage in making register allocation and assignment decisions. Proebsting and Fischer [27] developed a probabilistic approach in which local allocation is followed by probabilistic global allocation performed iteratively from inner to outer loops. Although the register allocator performed very well on the Stanford Benchmarks, compile time is significantly increased by this method, primarily due to the frequent recomputation of probabilities.

A hierarchical approach to global register allocation was presented by Callahan and Koblenz [12] in which they build a tree of "tiles" which covers the basic blocks of the control flow graph. In a bottom up pass, a graph coloring allocation of each tile is performed, summarizing the allocation of subtiles within the interference graph of the parent tile. A top down pass then binds virtual registers to physical registers and inserts spill code. Although the Callahan/Koblenz allocation method and RAP both take a hierarchical approach, the two research efforts differ in several ways. Most importantly, RAP performs register allocation over the PDG, rather than creating an additional tile structure that is used only for register allocation. By using a common compiler representation, RAP can be more easily implemented in existing compilers and possibly intertwined with other compile time optimization analyses and transformations.

The underlying target architectures of the allocators are different, as RAP is designed to support a load/store architecture, while the Callahan/Koblenz allocation method presumes that operands can be accessed from memory. Thus, RAP is more suitable for RISC and pipelined architectures than the direct memory access of the Callahan/Koblenz model. A register allocator for a load/store architecture needs to be able to allocate a register for every variable, even if it is only for a short time while the variable is being accessed. Spill code insertion consists of inserting loads immediately before variable uses and stores immediately after variable definitions. In contrast, a register allocator for an architecture where variables can be accessed from memory can avoid allocating a register to a variable for a segment of code, but needs to insert loads and stores on the boundaries of code segments that differ in their allocation for that variable. The major differences in the two register allocation algorithms are in the approach to allocating within code segments, the time at which register allocation and assignment actually take place, and spill cost calculations. No experimental results have been reported on the performance of the Callahan/Koblenz allocator.

We begin with a background section which describes the problem of register allocation and gives a brief description of the Program Dependence Graph intermediate representation. Section 3 describes each of the phases of RAP. Our experimental study and results are presented in Section 4.

2 Background

2.1 Register Allocation

The goal of register allocation is to map variables in an intermediate language to physical registers in order to minimize the number of accesses to memory during program execution. Most recent research casts register allocation as a graph coloring problem [11, 15, 14, 12, 9, 8, 13]. These techniques involve building an *interference* graph in which nodes of the graph represent variables or virtual registers, and edges between two nodes indicate that the corresponding variables cannot be mapped to the same physical register because their values will coexist during program execution. The register allocation is determined by coloring the nodes of the graph with no more than k colors where k is the number of physical registers. Two nodes cannot be assigned the same color if they are connected by an edge.

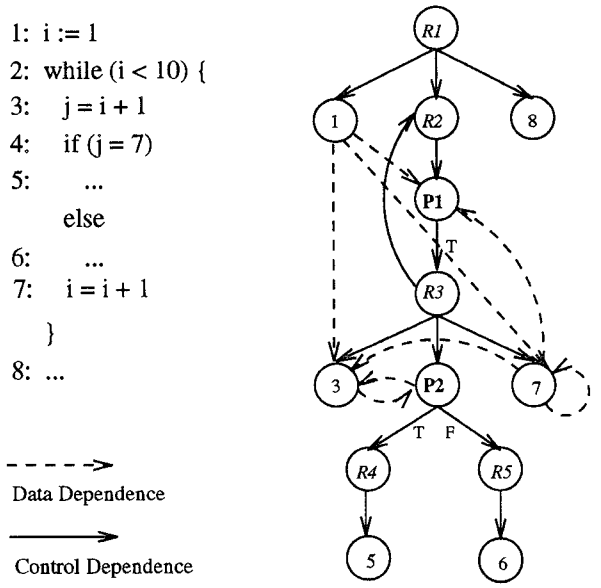

Figure 1: Program Dependence Graph

If a coloring cannot be found, a variable is spilled to memory. How the spilling is performed is dependent upon the target architecture and the actual global register allocation technique. A RISC architecture requires that operands be accessed from memory and thus a load is inserted before each use of the spilled variable and a store after each definition. When the architecture allows operands to be accessed out of memory, spilling can consist of changing each reference to the variable to be a reference to memory.

2.2 Program Dependence Graphs

The Program Dependence Graph for a program is a directed graph that represents the relevant control and data dependences between statements in the program. The nodes of the graph are statements and predicate expressions that occur in the program. An edge represents either a control dependence or a data dependence among program components. A data dependence edge from x to y denotes the fact that executing y before x could alter the program's semantics because x and y may reference the same memory location with at least one of them writing to that location. A control dependence edge from p to n denotes the fact that predicate p immediately controls the execution of the statement n. The source of a control dependence edge is either the entry node of the program or a predicate. A control dependence edge from a predicate node is labeled with either *true* or *false*, indicating the value of the predicate under which the statement at the sink of the edge will be executed.

Special nodes called region nodes are inserted into the graph to summarize the set of control conditions for a node and to group all of the nodes that are executed under the same control conditions together as successors of the same region node. For a given node n, each subset of its control dependences that is common with control dependences of another node, say m, is factored out, and a region node R is created to represent this control dependence subset. The common control dependence edges to n and m are replaced by edges $R \rightarrow n$ and $R \rightarrow m$. Each region node represents a set of control conditions, and after region nodes are inserted, each predicate node has at most one *true* outgoing edge and one *false* outgoing edge. Thus, region node insertion creates a hierarchical organization of the PDG.

Figure 1 shows a program segment and its PDG representation. The data dependence due to the definition of i in instruction 1 and the use of i in instruction 3 is represented by a directed edge from node 1 to node 3. The self dependence due to the increment of scalar variable i in instruction 7 is represented by the cyclic edge on node 7. The WHILE loop condition and IF statement are represented by predicate nodes P1 and P2, respectively. The region node R1 represents the control conditions on program entry, region node R2 represents the conditions on entering the loop or looping back to possibly execute another iteration of the loop, R3 represents the conditions under which the body of the loop is executed, R4 represents the THEN branch, and R5 represents the ELSE branch.

3 RAP

The input to the RAP register allocator consists of the PDG with attached low-level intermediate code statements which were generated assuming an infinite number of available registers. Definitions and uses in the intermediate code are references to *virtual registers*. The goal of RAP is to map the unlimited number of virtual registers onto the limited number of physical registers. RAP consists of three phases. The first phase of the allocator computes the register allocation for each region in a bottom up pass over the PDG. The second phase moves loads and stores out of loops where possible. The final phase is a local optimization that eliminates loads and stores in basic blocks when possible. The following subsections explain each of these phases.

3.1 Bottom Up Allocation

Instead of building the interference graph for the entire procedure at once as is done by most global register allocators, RAP begins by building an interference graph for *leaf* regions. During a bottom up pass of the PDG, it builds the interference graphs of each region, with

```
procedure rap(V, G_V){
// Input: region node V
// Output: interference graph, G_V, for region node V

  R = set of m subregions of V
  spill = true
  while (spill){
    add_region_conflicts(V, G_V)
    add_subregion_conflicts(V, G_V)
    calc_spill_costs(V, G_V)
    color_stack = simplify(G_V)
    spill_list = color(G_V, color_stack)
    if (spill_list is empty){
      combine(G_V)
      spill = false
      for i = 1 to m{
        if (loop_node(R_i) == false)
          delete(G_R_i)
      }
    } else
      insert_spill_code(V, spill_list)
  }
}
```

Figure 2: RAP on a region

the interference graphs of subregions incorporated into the parent region's graph. The interference graph for the entry region of the PDG has nodes to represent every virtual register referenced in the PDG and the register assignment is done at this level.

A *region* refers to a region node in the PDG and all of its control dependence successors. The *parent region* refers to only the topmost region node of the region. A *subregion* of the parent region refers to a subregion node and all of its control dependence successors. For example, in Figure 1, *R1* is the parent region of the region consisting of *R1*, *R2* , *R3*, *R4*, and *R5*. The subregion of this region consists of *R2*, *R3*, *R4*, and *R5*. Also, *R3* is the parent region of the region consisting of *R3*, *R4*, and *R5*. The subregions of this region are *R4* and *R5*. A virtual register is *local* to a region if all references to that virtual register can be found in intermediate code within the region; otherwise, the virtual register is *global* to that region.

Figure 2 presents the RAP allocation procedure executed on a region of the PDG. The *add_region_conflicts* routine adds nodes and edges to the interference graph for the region by looking at statements in the parent region only. The next routine, *add_subregion_conflicts*, incorporates the interference graphs of the subregions of the region. The *calc_spill_cost* routine attaches a spill cost to each node in the resulting interference graph. The *simplify* routine removes nodes from the interference graph and pushes them onto a stack for coloring and *color* pops nodes from the stack and assigns each node a color different from the color assigned to each of its neighbors. The nodes that cannot be colored are returned by the *color* routine in a list of nodes to be spilled. If the list of nodes to be spilled is empty, then a new interference graph is created for the region by combining those nodes which have been assigned the same color into a single node. The interference graphs of subregions, except those which represent a loop, can be deleted. The interference graph of the region is saved for incorporation into the interference graph of its parent region. If any nodes are marked for spilling, spill code is inserted and the interference graph is rebuilt. The following subsections describe each of these routines in more detail.

3.1.1 Building the Interference Graph

The interference graph for a region is built in two steps, *add_region_conflicts* and *add_subregion_conflicts*. The *add_region_conflicts* procedure builds the part of the interference graph for the parent region interferences, without concern for the subregions. This is similar to standard global register allocation techniques [14], except that RAP adds an interference between any two virtual registers that are live on entrance to the parent region and referenced (either used or defined) within the region. These interferences need not be added in the standard technique for a given basic block because they will eventually be added as all basic blocks are examined. A virtual register that is not referenced within the parent region will not be added to the interference graph by *add_region_conflicts* even though the virtual register may actually interfere with each node in the interference graph. Omitting these unreferenced virtual registers insures that referenced virtual registers are given priority when coloring. An unreferenced virtual register, although possibly live within the parent region, will be one of the first candidates for spilling.

Figure 3 contains a section of code and its corresponding PDG. The statements *S1*... *S5* contain references to virtual registers *a*, *b*, *c*, *d*, and *e*. Figure 3 (c) contains the interference graph built for the parent of the region consisting of *R1*, *R2*, and *R3*. The interference graph consists of a node for each virtual register referenced in the parent region. An edge between two nodes indicates that they may not be assigned the same register; for example, *a* and *c* cannot be assigned the same register because they are simultaneously live. Note that the interference graph does not contain a node for virtual register *d* although *d* interferes with each node in the parent interference graph.

The *add_subregion_conflicts* procedure, shown in Figure 4, incorporates the interferences of the subregions into the parent region's interference graph. The *combine* step of each of the subregions' allocation results in a final interference graph for each subregion that contains no more than k nodes where k is the number of physical registers. Thus, each node in the inter-

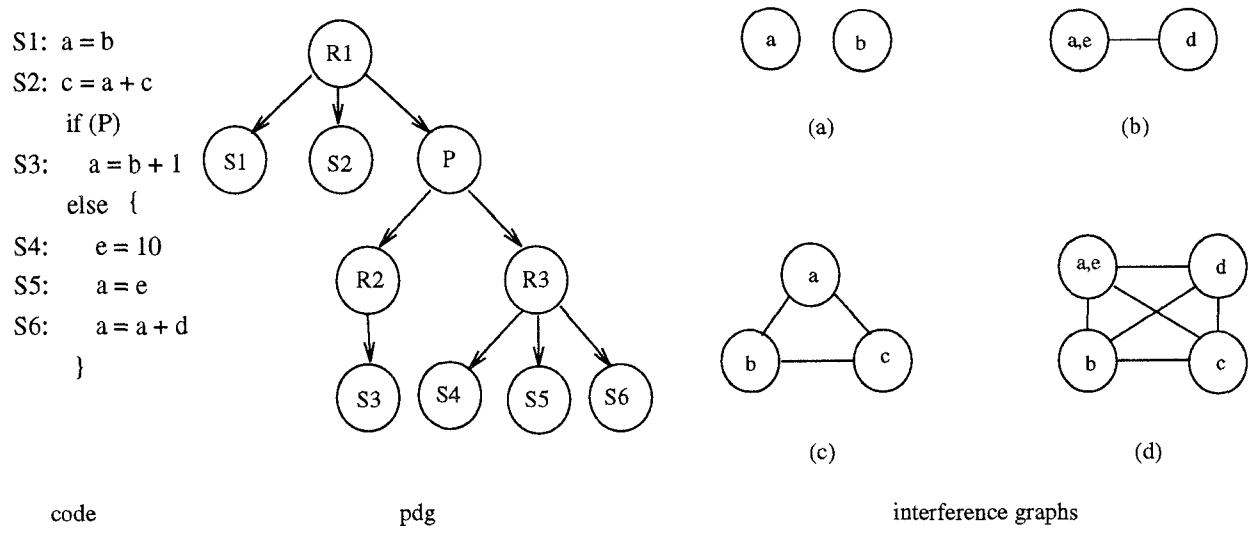

Figure 3: Building the interference graph for a region

```
procedure add_subregion_conflicts(V, G_V){
// Input: parent region node V,
// interference graph, G_V, of V
// Output: G_V modified to be interference
// graph of region

  R = set of m subregions of V
  Vars = set of Virtual Registers referenced in either V or
         some R_i
  Livein = set of Virtual Registers live on entrance to V
  for each v_k ∈ Vars where v_k ∋ G_V
    if v_k ∈ Livein
      add interference between v_k and all v ∈ G_V to G_V
  //
  for i = 1 to m
    for each v ∈ G_Ri {
      add v's interferences ∈ G_Ri to G_V
      for each v_k ∈ Vars where v_k ∋ G_Ri
        if v_k is live on entrance to R_i
          add interference between v_k and v to G_V
    }
}
```

Figure 4: Adding interferences to graph for region V

ference graph may represent several virtual registers which RAP has determined can be allocated to the same physical register in the subregion.

Each node and its conflicts in an interference graph of a subregion is added to the interference graph of the parent region, possibly by combining the subregion node with one of the parent's nodes if the nodes correspond to the same virtual register. All virtual registers referenced in the parent region and any subregions of this region will be present in the new interference graph. If a virtual register is live in the parent region, not referenced in the parent region, but referenced in one of the subregions, then a conflict must be added between the virtual register and every virtual register referenced in the parent region. [1] Similarly, if a virtual register is live, but not referenced in a subregion, but is referenced in the parent region or one of the other subregions, then a conflict must be added between that virtual register and all virtual registers referenced in the subregion. After all of these conflicts are added, the interference graph is complete for the parent region.

Figures 3 (a) and 3 (b) show the interference graphs for the subregions *R2* and *R3* of the region consisting of *R1*, *R2* and *R3*. Figure 3 (b), which contains the interference graph for *R3*, contains a single node for virtual registers *a* and *e* because the coloring routine colored these two virtual registers the same color. Note that *a* and *b* in Figure 3 (a), which contains the interference graph for *R2*, were not colored the same color because there are uses of both *a* and *b* outside of the subregion. Although *a* and *b* do not interfere in the subregion *R2*, they do interfere in the parent region and it would be incorrect to assign them the same color and thus the same register.

Figure 3 (d) contains the complete interference graph for the region consisting of *R1*, *R2*, and *R3*. The interference graph of the region is constructed by combining the interference graphs of the subregions and the interference graph of the parent parent. For example, the node for virtual registers *a* and *e* in Figure 3 (b) is combined with the node for virtual register *a* in 3 (a) and the node for virtual register *a* in the 3 (c). The edges in each interference graph are added to the in-

[1] The general PDG structure requires more analysis than this to determine whether each of those interferences is necessary. For our particular implementation of the PDG (described in the experimental section), this is sufficient.

terference graph for the whole region. In addition to this straightforward creation of the region interference graph, interference edges must be added corresponding to those virtual registers which are live in a region but not accessed in it. For example, an interference is added between d and each virtual register referenced in the parent region because d is live, but not accessed, in the parent region.

3.1.2 Calculating Spill Costs

Figure 5 gives an overview of the *calc_spill_costs* algorithm. Initially, the algorithm determines which virtual registers in the region have already been spilled in the region or are completely local to a subregion. Spilling these virtual registers will not help to make the graph colorable (i.e., will not eliminate any interferences) and thus the spill cost is set to be quite high to prevent them from being spilled in this region. For example, in Figure 3 (d), ignoring the combining of a with e, the virtual register e which is local to subregion *R3* interferes only with the virtual register d which is global to the parent region. Spilling e would create two new live ranges in the subregion by inserting a store after the definition of e and a load before the use of e. Both of these live ranges would still interfere with d. Eliminating the interference between d and e can only be achieved by spilling d.

If the virtual register has not been spilled and is not completely local to a subregion, the spill cost of a node is calculated by examining the code in the parent region. For each virtual register, the spill cost is initialized to be the number of definitions and uses of the virtual register in the parent region. This corresponds to placing a load before each use and a store after each definition in the intermediate code of the parent region.

In addition to spilling a virtual register in the parent region, spill code may need to be inserted in a subregion if the virtual register is live on entrance to or exit from a subregion. Although each node in a subregion's interference graph was colorable at that level, incorporating the node into the interference graph of the parent region may now cause it to be uncolorable. This is because the virtual register represented by the node may be referenced in more than one subregion or in both the subregion and the parent region. Since these nodes represent the same virtual register, they are combined in the parent's interference graph possibly increasing the number of conflicts. Thus, the next loop of *calc_spill_costs* determines whether a spill of a virtual register will require a load and/or a store to be inserted into a subregion and increments the spill cost accordingly.

In the last two loops of *calc_spill_costs*, the degree of each node is incremented appropriately, and the spill cost of each node is divided by its degree. The degree

```
procedure calc_spill_costs(V, G_V){
// Input: parent region node V,
// interference graph G_V
// Output: spill cost of each node in G_V

  N_{G_V} = set of nodes of G_V
  R = set of m subregions of V
  S = set of n statements in region V
  Livein_{R_i} = set of virtual registers live on entrance to
                 region R_i and used in region R_i
  Liveout_{R_i} = set of virtual registers live on exit from
                  region R_i and defined in region R_i
  degree(node) = number of interferences of node
  //avoid spilling those local to subregion or
  //already spilled
  for each node ∈ N_{G_V} do{
    if ∃R_i such that ∀ v ∈ node, v is local to R_i
      spill_cost(node) = 999999
    else if some v ∈ node is spilled in V
      spill_cost(node) = 999999
    else
      spill_cost(node) = 0
  }
  //initialize spill cost to #uses + #defs in region
  for i = 1 to n do
    for each v referenced in S_i {
      node = node ∈ G_V representing v
      spill_cost(node) ++
    }
  //increment for load/store on boundaries
  for i = 1 to m do
    for each node ∈ N_{G_V} do{
      if ∃ v ∈ node such that v ∈ Livein_{R_i}
        spill_costs(node) ++
      if ∃ v ∈ node such that v ∈ Liveout_{R_i}
        spill_costs(node) ++
    }
  //calculate degrees
  for each node_i ∈ G_V
    for each node_j ∈ G_V, i ≠ j
      if (∃ no edge between node_i and node_j)
          and (∃v_i ∈ node_i and ∃v_j ∈ node_j
          such that v_i and v_j are both global to V) {
          degree(node_i)++
          degree(node_j)++
        }
  for each node ∈ N_{G_V} do
    spill_cost(node) = spill_cost(node) / degree(node)
}
```

Figure 5: Calculating spill costs

of each node in the interference graph is equal to the number of neighbors of that node. When assigning the node a color, the node cannot be assigned the same color as any of its neighbors. In addition, if the virtual register represented by the node is global to a region, it cannot be assigned the same color as another global virtual register. Although the two global virtual registers may not interfere in the parent region, they could possibly interfere in another region thus prohibiting them from being allocated to the same physical register. For this reason, the degrees of the two nodes are each incremented by one if both are global to this region, but do not interfere in the region.

3.1.3 Coloring the Interference Graph

The interference graph is simplified by removing each node in the interference graph and pushing it onto a stack for coloring. The *simplify* phase repeatedly removes a node with degree less than the number of physical registers, k, or if none are available, a node with the least spill cost, and pushes it onto the stack. The nodes of the interference graph are colored by popping each node off of the stack and assigning it a color different from the color assigned to each of its neighbors. If a node corresponds to a global virtual register, then this virtual register cannot be colored the same color as any other global virtual register, although it may be colored the same color as a local virtual register. Because of the order in which the nodes are pushed onto the stack, the nodes with the most expensive spill cost are colored first. If a node cannot be colored, it is added to a list of nodes to be spilled.

If a node cannot be found with degree less than k and instead a node with least spill cost has to be removed, then Chaitin's original technique [14] marks this node to be spilled and not pushed on the stack. Instead, the technique used in RAP, which delays the identification of nodes to be spilled, is an enhancement to Chaitin's original coloring technique proposed by Briggs, Cooper, Kennedy and Torczon [9]. By deferring the spilling until the nodes are popped from the stack, it may be possible to color a node which Chaitin's method would have spilled for two reasons. First, a neighbor of the node may have been spilled thus possibly leaving a color available for this node or second, two neighbors of the node may have been assigned the same color. The set of nodes spilled by this method is a subset of the nodes spilled by Chaitin's method.

3.1.4 Inserting Spill Code

The *insert_spill_code* step removes each node from the list of nodes to be spilled and first adds spill code to the intermediate code of the parent region and then to the intermediate code of each subregion. For each spilled virtual register, a load is added before each of its uses and a store is added after each of its definitions in the intermediate code of the parent region. The virtual register is then renamed in the parent region. Next, the spill code is inserted into each subregion. If the virtual register is live on entrance to the subregion, a load is inserted before the first use in the subregion. In addition, a store is inserted after each definition of the virtual register which has a corresponding use outside of the subregion. The virtual register is then renamed, making it completely local to the subregion.

This only inserts spill code within the parent region and its subregions, but it may also be necessary to insert some spill code outside of the region. For example, suppose a load is placed before the use of a spilled virtual register and the definition corresponding to that use is outside the region. A store must be placed after this definition; otherwise, the load will not retrieve the correct value. Similarly, if a store is placed after a definition which has a corresponding use outside of the region, then a load must be placed before this use. The insertion of spill code outside of the region is done recursively so that each stored definition has a load placed prior to the corresponding use and each loaded use has a store placed after the corresponding definition. This technique does not place loads and stores after every use and definition of the virtual register since (1) the virtual register could correspond to more than one live range and (2) a load is only placed before the first use in a subregion, and a store is only placed after definitions which have uses outside of the subregion. This can cause spill code to be inserted into a region in which the virtual register was not actually spilled by the allocation; we attempt to move some of this spill code to less frequently executed segments (or eliminate it entirely) in a later phase.

3.1.5 Combining

After the interference graph for the parent region has been colored, the same color nodes of the interference graph are combined and this interference graph is saved for incorporation into the interference graph of its parent region. Although there may be many virtual registers referenced in the region, the final interference graph contains at most k nodes, where k is the number of physical registers. The interference graphs of each subregion may now be deleted unless the subregion represents the top region node of a loop. The interference graph of a loop region is saved for spill code movement.

3.2 Spill Code Movement

After the allocation phase, RAP attempts to move loads and stores outside of loops which were possibly inserted there because the virtual register was spilled

```
(1)   ldm  r2, 20          (4)   stm  20, r2
      ...no redef of r2...       ...no redef of r2...
      ldm  r2, 20                stm  20, r2

(2)   ldm  r2, 20          (5)   stm  20, r2
      ...no redef of r2...       ...no redef of r2...
      ldm  r3, 20                mv   r3, r2
                                 ...no redef of r3...
(3)   ldm  r2, 20                stm  20, r3
      ...no redef of r2...
      stm 20, r2
```

Figure 6: Elimination of unnecessary loads and stores

in another region. This separate phase to move the spill code out of loops is simpler than preventing the insertion of the spill code inside the loop during the initial allocation phase. The loads and stores of a virtual register may be removed from a loop if the virtual register was not combined with another virtual register in the region. This information is available in the interference graph for the loop region.

The spill code movement phase proceeds in a top down traversal of the PDG so that moving loads and stores outside of the entire loop nest is attempted before moving the loads and stores out of inner loops of that nest. Special spill nodes are created in the PDG to hold the moved spill code. A load of a virtual register must be inserted in the spill node immediately prior to the loop if the first reference in the loop is a use. Similarly, a store must be inserted in the spill node immediately following the exit of the loop if the loop contains a definition of the virtual register with a future use outside the loop.

3.3 Optimization

In the PDG used by RAP, there is a region node corresponding to each source code statement. This permits the interference graph at each of these levels to be quite small, but can also cause a basic block to be divided into several subregions. A virtual register which is referenced in these subregions and spilled in the parent region may cause the virtual register to be renamed in each of these subregions. If the renamed virtual registers are assigned to the same physical register, then unnecessary spill code is created in the basic block. The final phase of RAP performs a local optimization which eliminates these unnecessary loads and stores.

The last phase of RAP examines the spill code inserted into each basic block and removes spill code of the forms noted in Figure 6. In this figure, *ldm* is a load direct and *stm* is a store direct instruction. The register references refer to the assigned physical registers. In (1), the second *ldm* can be eliminated because register $r2$ still contains the value loaded from memory. In all of the examples except (2), the second memory reference can be eliminated. In (2), the *ldm* can be replaced by a copy statement which copies the contents of $r2$ into $r3$.

4 Experimental Study

The performance of RAP has been compared to the performance of a conventional global register allocator which we call GRA. GRA is basically an implementation of Chaitin's global register allocator with two exceptions: (1) The enhancement suggested by Briggs et. al. [9] has been incorporated. (2) No coalescing or rematerialization is done [14, 11]. These modifications were made to GRA in order to present a fair comparison with the current version of our prototype implementation of RAP. We have incorporated the Briggs enhancement into RAP, but RAP currently does not include coalescing in its traditional form or rematerialization.

The front end for RAP is the *pdgcc* compiler, developed at the University of Pittsburgh [17], which accepts C source code as input and outputs the corresponding PDG. RAP inputs the PDG representation of the C program and first generates and attaches low-level intermediate code to the corresponding region nodes. The intermediate code representation is iloc developed at Rice University for the development of optimizing compilers [10]. RAP performs register allocation over the PDG representation and generates code with a correct register assignment. Alternatively, RAP can simply output the unallocated iloc code which is then used as input to GRA for experimental comparison. An iloc interpreter is used to count the number of cycles required to execute the code. For this study, we assume that each instruction takes one cycle to execute.

Performance measurements of RAP and GRA have been taken for 13 of the Livermore Loops, the cLinpack routines, implementations of heapsort, hanoi, sieve and some of the Stanford routines. Table 1 presents the results of these experiments for register set sizes 3, 5, 7 and 9. The *tot* columns show the percentage decrease in the total executed cycles, calculated as $(cycles(GRA) - cycles(RAP))/cycles(GRA)$ where *cycles* is the number of cycles required to execute the code as determined by the iloc interpreter. The *ld* and *st* columns indicate what portion of the percentage is due to a change in the number of loads executed and stores executed, respectively. The remaining portion can be attributed to an increase or decrease in the number of copy statements executed. The −0.0 entries in the table indicate a very small negative percentage. The +0.0 entries indicate a very small positive percentage. An entry is blank in the table if the allocated code does not contain spill code.

Benchmark	Number of Registers											
	3			5			7			9		
	tot	ld	st	tot	ld	st	tot	ld	st	tot	ld	st
Livermore												
loop1	3.4	0.0	+0.0	0.0	-2.7	-1.1	4.0	0.0	0.0	4.1	0.0	0.0
loop2	1.1	0.0	-2.7	4.4	0.0	0.0	4.7	0.0	0.0	4.8		
loop3	0.0	-7.1	0.0	5.7	-2.9	-0.0	8.8			8.8		
loop4	3.9	1.3	0.0	-0.0	-1.5	-1.6	1.6	-3.3	-0.0	5.2		
loop6	11.2	6.3	1.0	5.4	-0.0	0.6	3.8	-1.3	-0.0	5.3		
loop7	3.8	0.0	+0.0	2.6	-0.0	-1.5	3.8	0.0	-0.7	3.9	0.0	-0.7
loop8	0.8	1.6	-2.0	3.2	-1.3	3.1	1.0	0.4	-1.0	3.8	0.5	1.6
loop9	5.7	2.1	-0.5	5.0	0.0	+0.0	5.1	0.0	+0.0	5.1	0.0	+0.0
loop10	3.2	-0.5	-0.5	5.0	-0.7	1.1	5.2			5.2		
loop11	-4.3	-6.4	-2.1	2.4	+0.0	-2.4	5.3			5.3		
loop12	4.3	2.2	-2.2	5.3	0.0	0.0	5.4			5.4		
loop13	5.5	1.2	-0.1	3.6	-1.5	0.5	4.3	+0.0	-0.6	6.6	1.2	-0.0
loop14	2.9	1.8	-1.4	-2.5	-2.1	-1.4	4.5	0.2	-0.0	2.3	-2.9	-0.0
Clinpack												
idamax	-13.2	-9.5	-9.0	-0.5	-5.5	-0.5	-8.0	-13.4	-1.1	6.6		
dscal	5.9	5.9	0.0	0.0	6.2	-6.2	7.7	7.7		0.0		
daxpy	3.7	3.7	0.0	7.3	7.3	0.0	0.0	7.6	-3.8	-6.0	6.1	-8.0
matgen	-0.1	-4.1	-0.1	-1.9	-4.9	-1.8	-1.9	-7.0	-0.2	-2.4	-8.0	-0.1
dgefa	9.5	3.1	5.0	1.7	-4.5	3.6	-3.8	-8.3	0.8	-3.2	-8.4	0.9
Heapsort												
hsort	-0.3	2.7	-3.8	4.3	0.5	0.4	4.5	-0.8	1.6	0.8	-4.0	-1.8
Hanoi												
main	-11.3	-6.7	-3.5	-10.4	-5.7	-5.7	8.3			8.3		
mov	8.5	5.7	2.8	1.2			1.2			1.2		
Nsieve												
seive	32.6	14.6	4.9	14.4	0.6	0.8	3.4	-0.4	-2.4	-5.2	-3.9	-2.6
Stanford												
initmatrix	-4.7	-7.1	-2.4	-3.0	-8.3	-0.1	6.1	+0.0	0.0	6.3		
innerproduct	-5.5	-8.2	-0.1	3.2	0.1	0.0	0.0	-3.6	-0.1	0.0	-3.8	-0.1
intmm	-19.2	-11.0	-8.3	-3.1	-6.4	-0.0	-3.3	-6.8	-0.0	3.6		
permute	14.1	9.1	3.2	2.6			2.6			2.6		
swap	9.1	9.1	0.0	11.1			11.1			11.1		
initialize	8.0	1.0	0.0	7.7			7.7			7.7		
perm	7.5	1.2	0.0	6.5			6.5			6.5		
fit	-20.9	-12.6	-8.3	-9.1	-13.4	-0.4	-0.6	-5.2	-0.3	5.1		
place	-9.4	-10.8	-3.1	3.9	-1.2	-0.1	-8.1	-11.8	-1.3	-2.9	-7.0	-1.4
trial	12.9	3.9	8.1	13.5	-1.9	-0.2	13.8	-2.5	-0.5	17.9		
remove	-13.8	-13.6	-4.6	4.5	-0.3	-0.1	-0.6	-5.4	-0.3	4.3	-0.3	-0.5
puzzle	5.3	-5.8	-0.0	-1.8	-6.3	-0.2	1.3	-0.1	-0.2	7.1	-0.0	-0.0
queens	12.3			12.3			12.3			12.3		
try	-14.4	-8.7	-5.2	-6.1	-5.2	-0.6	-25.2	-23.1	-4.4	-14.2	-14.8	-1.8
doit	3.5	0.1	0.0	1.2	-2.0	-0.1	3.8	0.3	0.0	4.0	0.2	0.0
Average	1.7			2.7			2.6			3.7		

Table 1: Percentage decrease in cycles executed

For example, when 3 registers are available, the Livermore loop6 code allocated registers via RAP executes 11.2% faster than the same code allocated registers via GRA. 6.3% of the 11.2% is due to a reduction in the number of loads executed and 1.0% of the 11.2% is due to the reduction in the number of stores executed. The remaining 3.9% is due to the reduction in the number of copy statements executed. Although neither GRA nor RAP does coalescing, a copy statement in the unallocated iloc code can be eliminated when both operands of the copy are allocated the same register.

The last row in Table 1 indicates the average percentage decrease in the total number of cycles executed for each of the register set sizes. For register set sizes of 3, 5, 7, and 9 the average percent decrease is 1.7, 2.7, 2.6 and 3.7, respectively. The average percent decrease over all the experiments is 2.7. When the register set consists of 9 registers, the percentage decrease in the total number of cycles executed can be mostly attributed to a reduction in the number of copy statements since little spilling occurs in these routines with a register set of that size. For a register set size of 3, 25 of the 37 routines demonstrated a positive percentage decrease in the total number of cycles executed. For a register set size of 9, 30 of the 37 routines demonstrated a positive percentage decrease.

We found that although coalescing was not implemented, RAP was much better at eliminating copy statements than GRA. This was simply because each region in our PDG likely consists of only a few iloc statements. This is due to the *pdgcc* compiler which creates a region node for each C statement. The interference graph for such a region would consist of only a few nodes. The *first fit* coloring technique that is used colors a node with the first available color that it can find thus maximizing register reuse. The combination of a small interference graph and the first fit coloring technique would often cause the two nodes corresponding to the operands of the copy to be assigned the same color and thus the same register, effectively eliminating the copy statement. Implementing an explicit coalescing step in GRA and RAP is likely to increase the performance of each, but particularly, it should improve the performance of GRA.

Although good for coalescing, the small regions were not always good for adding spill code. If a virtual register is spilled within a region, then a store is added after each definition and a load is added before each use. In addition, a load is added before the first use of the definition in each subregion containing a use of that virtual register. Having a large number of regions can cause an excess amount of spill code to be inserted. For example in Figure 7, our pdg implementation would put statements *S2* and *S3* into the separate regions *R2* and *R3* as indicated, if *S2* and *S3* were generated from two different source code statements. If virtual register *a* is spilled when coloring the interference graph for the region with parent *R1*, then RAP inserts a load prior to the first use of *a* in subregions containing a use of the definition of *a* in *R1*. In this case, a load would be inserted prior to *S2* and prior to *S3*. If the two statements were in the same subregion, then only a load before *S2* would need to be inserted. For this reason, we believe that a PDG implementation in which the region contains more intermediate code statements than is possible in our PDG implementation would allow RAP to decrease the amount of spill code inserted.

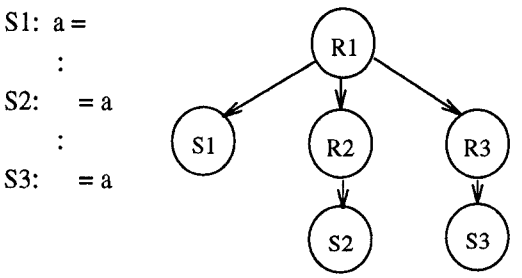

Figure 7: Effect of small regions on spilling

In Figure 7, if RAP chooses to spill *a* while coloring the interference graph for region *R1*, it will insert as much spill code as GRA would when choosing to spill that variable. However, if *R1* is the parent region node for a loop region, RAP may move the spill code for *a* out of the region. A single load for *a* may be placed prior to the entrance of *R1* and the two loads within the region can be eliminated. Thus, although RAP may add more spill code to subregions than is necessary, spill code cleanup may be able to eliminate some of that code.

Some of the differences in the results of the allocators can be attributed simply to the difference in spill cost calculations. When RAP calculates the cost of spilling a node of the interference graph built for a particular region, it may be examining only a subset of the total uses and definitions of the variable. In contrast, GRA calculates the spill cost by counting each use and definition of a variable in the whole procedure. One of the inherent problems in this method of spill cost calculation is that spilling a node with the cheapest cost does not necessarily cause the interference graph to be colorable in the next iteration. This can be a problem in GRA and in RAP. Upon analyzing our results closely, we found that occasionally the spill cost caused RAP to make a smart decision resulting in less spilling than GRA and occasionally, vice-versa.

5 Conclusions

In this work, we set out with the goal of developing a global register allocator based on the PDG representation of a program without sacrificing the quality of the resulting register allocations. This paper described how we achieved this goal, and in fact, usually obtain improved allocations over traditional global graph coloring register allocators, by exploiting the region partitioning of the PDG.

Our experiments suggested some ways in which we could improve the performance of RAP. First, we found that at least some of the improvement of RAP over GRA can be attributed to the ability of RAP to eliminate copy statements without actually performing an explicit coalescing step. However, we also found routines in which GRA allocated code contained fewer copy statements than RAP allocated code. We expect that the performance of RAP will be improved by implementing coalescing, and we are interested in comparing the results when coalescing is performed by both RAP and GRA.

In addition, it is likely that the performance of RAP could be improved by increasing the number of iloc statements within a region. We are currently examining the possibility of obtaining some flexibility on the size of the region for which an interference graph is constructed. Finally, as Figure 7 suggests, better placement of spill code is desired. RAP currently attempts to move spill code out of loop regions, but moving spill code out of any subregion is also likely to reduce the amount of spill code executed. Unlike the hierarchical tile structure of Callahan and Koblenz [12], a region of a PDG may contain multiple exits making spill code placement more difficult. The tile structure allows placement of spill code in the single entrance and the single exit of the tile tree. Instead of attempting to identify the possibly multiple exits of a region, RAP inserts spill code within the region and attempts to move the spill code out of the region in a later phase.

Our current research also involves investigating possible collaborations between RAP, a region scheduler and a local instruction scheduler. We believe that RAP will aid in the phase integration desired for generating efficient final code because the register allocator now shares the same program representation with many other tasks of an optimizing or parallelizing compiler.

Acknowledgements

We would like to thank the University of Pittsburgh compiler group for providing us with the *pdgcc* compiler. Also, our thanks to the program committee for their helpful suggestions for the final paper.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 246–256, White Plains, NY, June 1990.

[2] V. H. Allan, J. Janardhan, R. M. Lee, and M. Srinivas. Enhanced region scheduling on a program dependence graph. In *Proceedings of the twenty-fifth International Symposium on Microarchitecture*, pages 72–80, Portland, OR, 1992.

[3] F. E. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5:617–640, 1988.

[4] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 257–271, White Plains, NY, June 1990.

[5] W. Baxter and H. R. Bauer, III. The program dependence graph and vectorization. In *Proceedings of the Sixteenth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, Austin, TX, 1989.

[6] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, June 1991.

[7] David Bernstein and Michael Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, CANADA, June 1991.

[8] Preston Briggs. *Register allocation via graph coloring.* PhD thesis, Rice University, April 1992.

[9] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, July 1989.

[10] Preston Briggs, Keith D. Cooper, and Linda Torczon. R^n programming environment newsletter #44. Department of Computer Science, Rice University, September 1987.

[11] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, June 1992.

[12] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 192–203, Toronto, CANADA, June 1991.

[13] G. J. Chaitin. Register allocation and spilling via graph coloring. In *SIGPLAN Symposium on Compiler Construction*, Boston, June 1982.

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

[15] Frederick Chow and John Hennessy. Register allocation by priority-based coloring. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, June 1984.

[16] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[17] Claude-Nicolas Fiechter. PDG C Compiler. University of Pittsburgh, 1992.

[18] J. H. Griffin and K. J. Ottenstein. PROBE: a dependence-based program browser. Technical Report LA-UR-89-1823, Los Alamos National Laboratory, Los Alamos, NM, November 1989.

[19] Rajiv Gupta and Mary Lou Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, 1990.

[20] Rajiv Gupta, Mary Lou Soffa, and Tim Steele. Register allocation via clique separators. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, Oregon, June 1989.

[21] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 234–245, White Plains, NY, June 1990.

[22] S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. In *Proceedings of the Fifteenth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 133–145, San Diego, CA, 1988.

[23] D. J. Kuck, R. H. Kuhn, B. Leasure, D. A. Padua, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 207–218, 1981.

[24] Cindy Norris and Lori L. Pollock. A scheduler-sensitive global register allocator. In *Supercomputing '93 Proceedings*, Portland, OR, November 1993.

[25] K. J. Ottenstein. An intermediate program form based on a cyclic data-dependence graph. Technical Report 81-1, Department of Computer Science, Michigan Tech. University, 1981.

[26] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of ACM SIGPLAN/SIGSOFT Symposium on Practical Software Development Environments*, pages 177–184, Pittsburgh, PA, April 1984.

[27] Todd A. Proebsting and Charles N. Fischer. Probabilistic register allocation. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 300–310, San Francisco, CA, June 1992.

[28] J. Warren. A hierarchical basis for reordering transformations. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 272–282, 1984.

[29] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[30] M. J. Wolfe. *Research Monographs in Parallel and Distributed Computing.* The MIT Press, 1989.