Register Allocation Across Procedure and Module Boundaries

Vatsa Santhanam
Daryl Odnert

Hewlett-Packard Company
California Language Laboratory
19447 Pruneridge Avenue
Cupertino, California 95014

Abstract

This paper describes a method for compiling programs using interprocedural register allocation. A strategy for handling programs built from multiple modules is presented, as well as algorithms for *global variable promotion* and register *spill code motion*. These algorithms attempt to address some of the shortcomings of previous interprocedural register allocation strategies. Results are given for an implementation on a single register file RISC-based architecture.

1. Introduction

Priority-based graph coloring techniques for register allocation have become common in many optimizing compilers designed for recent computer architectures. This intraprocedural form of register allocation has proven to be effective for computers with a plurality of general purpose registers [Chow 84]. However, in the absence of interprocedural information, the following situations occur, which leave room for further performance improvement:

- Local variables in different procedures can be assigned to the same register. As a result, procedures must execute spill code to save and restore registers whose contents need to be preserved across procedure calls.
- Global variables can be referenced out of different registers in different procedures. Registers holding values for global variables need to be stored to memory and loaded back into registers at procedure calls and returns.

To solve this problem, interprocedural register allocation techniques have been developed. Implementations of these techniques for most programming languages are complicated by the need to support programs built from multiple modules (compilation units). For example, if a global variable is maintained in a single register across procedures of one module, there must be a method to communicate this information to procedures in other modules.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

©1990 ACM 0-89791-364-7/90/0006/0028 $1.50

Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation. White Plains, New York, June 20-22, 1990.

1.1. Related Work

Previous attempts at interprocedural register allocation generally follow one of two approaches. One approach is to relocate memory references at link-time, as was first suggested in [Wall 86]. In this technique, the compiler performs traditional register allocation on each procedure and generates register relocation information for the linker. Interprocedural register allocation is implemented in the linker. Registers are assigned to frequently referenced global variables and local variables that are not concurrently active in the call graph. To maintain a variable in a register across procedures, the linker simply needs to follow the relocation actions prescribed by the compiler.

This link-time technique was found to be effective for a processor with a large register file. Most of the performance benefit was attributed to eliminating memory references to global variables. In this approach, registers used for global variables are dedicated to that purpose throughout the program. A dedicated register is consequently unavailable for other uses in regions of the program that do not access the global variable.

The second approach to interprocedural register allocation attempts to minimize register spill code by propagating register usage information through the program call graph as each procedure is compiled. Examples of this technique are described in [Steen 87], [Chow 88], and [Mulder 89]. In the approach taken in [Chow 88], an intermediate code representation is written to a file by the first phase of the compiler. Then, instead of linking object code, the user links the required intermediate code files into one large intermediate file. The intermediate code linker completes the code generation and optimization process.

Beginning at the leaf nodes, procedures are compiled in a bottom-up ordering of the call graph so that register usage information can be propagated upwards. By examining the register usage information at each call site, the register allocator can avoid assigning registers already used in the called routines, effectively minimizing spill code.

This technique has yielded generally positive results, although exceptions have been noted. With a limited number of available machine registers, the technique has been less effective when applied to programs whose execution is concentrated in the upper regions of the call graph.

1.2. A Different Approach

The approach presented in this paper differs from previous interprocedural register allocation efforts in three main areas. First, to maintain global variables in registers, instead of dedicating a register to a global variable throughout the program call graph, limited regions of the call graph are identified where a global variable is assigned to a register. This allows the same register to be used for different global variables in distinct regions of the program, resulting in a larger number of global variables being maintained in registers. Moreover, registers can be used for other purposes in regions where they are not reserved for a global variable.

Second, to minimize register spill, instead of simply propagating register usage information to call graph nodes in depth-first (bottom-up) order, restricted sets of call graph nodes are identified over which interprocedural register allocation is performed. These collections of nodes are arranged to correspond to regions of the call graph that are procedure call intensive. The objective is to move register spill code out of frequently called routines in such regions. For some applications, minimizing register save-restore overhead in call intensive regions can improve run-time performance more than an approach that always favors the lower regions of the call graph.

Third, a two-pass compiler organization is used that differs from previous strategies for interprocedural register allocation. The key component of this compilation environment is a tool called the *program analyzer*. The program analyzer facilitates interprocedural register allocation without requiring procedures to be compiled in any particular order. The program analyzer is somewhat similar to the *Program Compiler* component of the R^n compiler system described in [Cooper 86]. Although the Program Compiler is used to facilitate interprocedural optimization, it has not been applied directly to the register allocation problem.

The register allocation algorithms described in this paper have been implemented in a prototype compiler and program analyzer targeted for the Hewlett-Packard Precision RISC Architecture (PA-RISC) [Mahon 86] [Lee 89]. The method will naturally adapt, however, to any processor architecture with a large register file. A linkage convention which allows each procedure a set of callee-saves and a set of caller-saves registers is assumed.[1]

PA-RISC is a 32-bit load-store architecture in which most instructions execute in a single clock cycle. Of the 32 general-purpose registers defined by the architecture, 16 registers are designated as callee-saves by software convention. Data collected over a variety of optimized applications suggested that several of the callee-saves registers remain unused in the most common paths of execution. Making more efficient use of these registers and reducing procedure call overhead were the primary motivations behind our work.

This paper is divided into seven major sections. Section 2 introduces the compilation process used. Section 3 describes the functionality of the compiler's first phase. Section 4 describes how interprocedural register usage is determined by the program analyzer tool. Section 5 discusses the role of the compiler's second phase. Benchmark measurements and analysis are presented in Section 6. Section 7 describes limitations of our approach and ideas for future work.

1. The contents of a *callee-saves* register must be saved and restored by any procedure that modifies it, and are thus preserved across calls. A *caller-saves* register can be modified by procedures without preserving its contents. Hence, a caller-saves register must be saved and restored around a call if its contents are subsequently used.

2. The Compilation Process

Figure 1 illustrates the two-pass compilation system used to perform interprocedural register allocation.

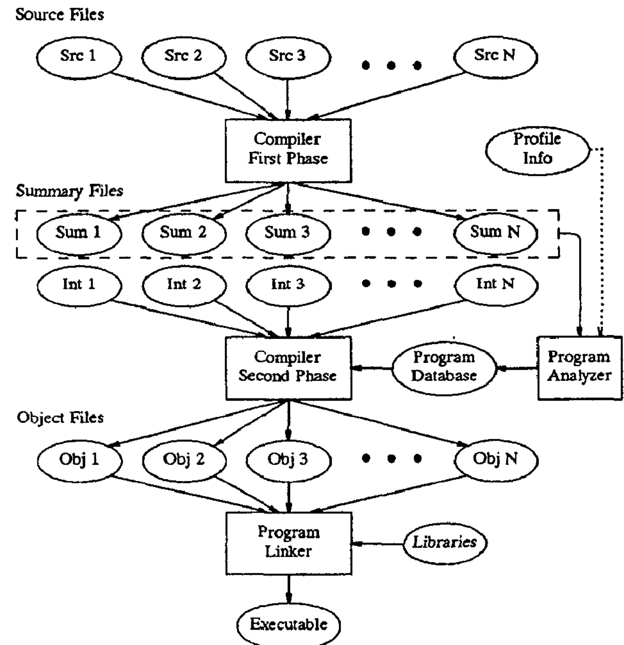

Figure 1

The compiler first phase reads each source file containing high-level program language text one at a time. After verifying syntactic and semantic correctness, an intermediate representation of the source is generated and saved in a file. Additionally, for each procedure, the compiler first phase collects a record of local information required to construct the program call graph and make interprocedural register allocation decisions. Each record of local information is stored in a *summary file* for the module.

The compiler first phase must be run on each source file that is to be included in the program. After all the summary files have been generated, the program analyzer is run. The program analyzer examines the record for each procedure and constructs the program call graph. It then computes register allocation directives for interprocedural register allocation. The program analyzer does not modify any code. Instead, a summary of the relevant directives for each procedure is placed in a *program database*.

By default, the register allocation decisions made by the program analyzer are guided by compile-time heuristics. The program analyzer can optionally use dynamic profile data to improve the accuracy of these heuristics.

After the program database is generated, the compiler second phase must be run on each intermediate file. This phase translates the intermediate code into machine code and is responsible for traditional global optimizations. As each procedure is optimized, the program database is consulted to obtain the register allocation directives computed by the program analyzer. These directives are used by the register allocation component of the compiler second phase.

The compiler second phase generates one object file corresponding to each intermediate file. The object files are then bound together by the linker with the appropriate run-time libraries to produce an executable program.

3. The Compiler First Phase

The primary responsibility of the compiler first phase is the traditional syntactic and semantic analysis of the source text. It then generates an intermediate representation of the source that is saved in a file and processed later by the compiler second phase. When compiling for interprocedural register allocation, the compiler first phase also writes out a record for each procedure in the source to a summary file. Each record contains the following information:

- The list of global variables accessed in that procedure, along with a value representing the frequency of local accesses to that variable. Flags are also set for each global variable to indicate, among other things, whether aliased references are possible.
- A list of all procedures called, and a value representing the frequency of local calls to each procedure.
- A list of procedures whose addresses have been computed for possible use in indirect calls and a flag indicating if indirect calls are made by this procedure.
- An estimate of the number of callee-saves registers needed for the procedure.

4. The Program Analyzer

The program analyzer is run after all the summary files for an application have been generated by the compiler first phase. The program analyzer first reads in all the summary files to construct a call graph for the program. The global variable promotion algorithm is then run, followed by the spill code motion algorithm.

4.1. Global Variable Promotion

Global variable promotion converts memory references of global variables into register references. In effect, the global variable is promoted from being a memory object to a register object - hence the term *global variable promotion*. Many optimizers are able to promote global variables to registers locally within a procedure. Such locally-promoted global variables are still accessed from memory across procedures. Before procedure calls and at the exit point, the optimizer must insert instructions to store the register containing the promoted global variable back to memory. Similarly, at the entry point and just after procedure returns, the optimizer must insert instructions to load the promoted global variable from memory to the register.

The program analyzer facilitates *interprocedural* global variable promotion. The objective is to arrange for a global variable to be accessed out of the same register in separate procedures, thus obviating intermediate transfers to and from memory. In the link-time technique described in [Wall 86], a register is dedicated for each global variable to be interprocedurally promoted throughout the program.

However, dedicating a unique register for each promoted global variable can be inefficient in routines that do not access those variables. To improve register use, we have applied the graph coloring technique commonly used in intraprocedural register allocation, described in [Chaitin 82], to live ranges of global variables computed over the program call graph.

4.1.1. Webs

The nodes of the call graph which reference a particular global variable can be grouped into disjoint live ranges, or *webs*, for that variable. A web for a global variable is a minimal subgraph of the program call graph such that the global variable is neither referenced in an ancestor node nor a descendent node of the subgraph. Partitioning the procedures that access a global variable into webs allows each web to be considered for register promotion individually.

If a web for a global variable is selected for promotion, a (callee-saves) register is dedicated to that global only in the procedures belonging to the web. The same register can be assigned to another global variable web or used for local values at other nodes of the call graph.

Within web nodes, the value of the promoted global variable is preserved across calls to nodes outside the web since it is maintained in a callee-saves register. Furthermore, by construction, it is guaranteed that such external nodes do not reference that global variable.

At the root nodes of the web sub-graph, called *web entry nodes*, code is inserted at the entry point by the compiler second phase to load the global variable to the register. Likewise, code is inserted at the exit point of web entry nodes to store the global variable back to memory.

Note that with this scheme, statically initialized global variables do not require special treatment. The initial value for a promoted global variable is simply loaded into the appropriate callee-saves register when the web entry node is invoked.

Another attractive feature of this scheme is that it is well suited to global variable usage found in structured programs. References to global variables tend to occur in localized sets of procedures (that are logically dependent on that global variable), rather than being scattered all over the program call graph. Webs can encompass such regions of the call graph and help limit dedicated register usage.

4.1.2. Identifying Webs

The program analyzer identifies webs by first determining the global variables that are eligible for promotion. To be eligible for promotion, a global variable must satisfy several criteria. For instance, it must be a variable small enough to fit in a single register, and it must not have been aliased to another variable.

A simple interprocedural data flow analysis over the call graph on the eligible global variables is then performed. The following sets are defined for this purpose:

L_REF[P] the local reference set for each procedure P. An eligible variable appears in this set if that variable is accessed within procedure P.

P_REF[P] the parent reference set for each procedure P. An eligible variable appears in this set if that variable is accessed in some procedure along a call chain from a start node to P.[2]

C_REF[P] the child reference set for each procedure P. An eligible variable appears in this set if that variable is accessed in some procedure along a call chain starting at P.

2. Every node without a predecessor is treated as a start node.

In the program analyzer, the L_REF sets are easily initialized using the information kept in the summary files. The P_REF and C_REF sets are initially empty. The following dataflow equations are used to iteratively propagate the local information through the call graph:

$$P_REF[P] = \bigcup_{\{ i \mid i \text{ is a predecessor of } P \}} (P_REF[i] \cup L_REF[i])$$

$$C_REF[P] = \bigcup_{\{ i \mid i \text{ is a successor of } P \}} (C_REF[i] \cup L_REF[i])$$

Note that these equations are correct only if we limit the eligible global variables to those that have not been aliased to any other variable. For faster convergence, the C_REF sets are propagated in depth-first (bottom-up) order while the P_REF sets are propagated in breadth-first (top-down) order.

Webs for a global variable are identified by looking for potential web entry nodes. The candidate web entry nodes for a global variable will have that variable in their L_REF sets, but not in their P_REF sets. Beginning from a web entry node, the web is recursively expanded to include all successor nodes which have the associated variable in either the L_REF or C_REF set for that node. The algorithm is shown in Figure 2.

For correctness, the algorithm in Figure 2 ensures that all immediate predecessors of web entry nodes are external to the web. Entry nodes would otherwise replace the contents of the dedicated register with a stale value when invoked from within a promoted web. The algorithm also ensures that nodes internal to the web do not have any external predecessors. Otherwise, when the internal node is invoked bypassing the web entry node, the dedicated register could be referenced without being properly loaded.

Enlarging webs to overcome these restrictions implies dedicating a register over larger regions of the call graph. We will later describe an extension involving stub routines to alleviate this problem.

The algorithm in Figure 2 nodes of some recursive call chains that reference eligible global variables will not be incorporated into webs. This is because a global variable can be included in the P_REF set for all nodes in the cycle, but not in the L_REF set of any nodes in the entry paths into that cycle. A simple solution is to include all the nodes in each such cycle into a separate web and enlarge that web for correctness using the algorithm shown in Figure 2.

```
Compute_Webs()
begin
for each eligible global variable G do
  for each procedure P such that G ∈ L_REF[P] do
    if ( G ∉ P_REF[P] ) then
      begin
      allocate a new web W for G;
      W→nodes := Ø ;
      temp := { P } ;
      repeat
        for each node Q in temp do
          Expand_Web( W, Q, G );
        S := { Z ∈ W→nodes |
                (Z has a predecessor in W→nodes) and
                (Z has a predecessor not in W→nodes) };

        if ( S ≠ Ø ) then
          temp := ∪ predecessors of nodes in S
                    that are not in W→nodes;
      until ( S = Ø );
      if ( W has any nodes in common with
           any other web X for G ) then
        begin
        W→nodes := X→nodes ∪ W→nodes;
        delete web X;
        end;
      end;
end;

Expand_Web( W, Q, G )
begin
W→nodes := W→nodes ∪ Q;
for each successor S of Q do
  if ((G ∈ C_REF[S]) or (G ∈ L_REF[S]))
     and (S ∉ W→nodes) then
    Expand_Webs( W, S, G );
end;
```

Figure 2

Using a predetermined subset of the callee-saves registers, as many web nodes of the interference graph are colored as possible. A web that is not colored results in the corresponding global variable not being promoted across procedures. The global variables of uncolored webs can still be promoted to registers intraprocedurally by the compiler second phase.

For each procedure belonging to a colored web, the promotion of the global variable to a specific register is recorded the program database. During the compiler second phase, all memory references to the corresponding global variable are converted into register references in the procedures belonging to the web. The register assigned to a web is unavailable for other purposes in any of the web procedures. The compiler second phase is also responsible for inserting register spill and initialization code at web entry procedures as described earlier.

4.1.3. Promoting Webs

Once webs have been identified, a web interference graph is constructed. Arcs connecting web nodes in this graph represent interferences. Two webs are said to interfere if they share a common call graph node. Clearly, interfering webs can not be promoted to the same register. The nodes of the web interference graph are sorted into a priority order for coloring (i.e. register promotion). The priority value of a web is computed using a heuristic function. This function factors in the the estimated number of dynamic global variable references within each web and the estimated calls to web entry nodes (for which loads and stores would need to be inserted at the entry and exit points).

4.1.4. An Example

The call graph shown in Figure 3 illustrates how this method allows a register to be shared among multiple webs. The nodes of the graph, labeled **A** to **H**, represent user procedures, and the arcs between these nodes represent procedure calls. In this example, there are three global variables named **g1**, **g2**, and **g3**. An explicit access to one of these variables is indicated by the presence of that variable in an L_REF set, from which the P_REF and C_REF sets are derived (Table 1).

Procedure	L_REF	C_REF	P_REF
A	g3	g1 g2 g3	Ø
B	g1 g3	g1 g2	g3
C	g2 g3	g2	g3
D	g1	Ø	g1 g3
E	g1 g2	Ø	g1 g3
F	g2	Ø	g2 g3
G	g2	Ø	g2 g3
H	Ø	Ø	g2 g3

Table 1

The webs for the three global variables are summarized in Table 2 and are outlined with dotted lines in Figure 3. As mentioned earlier, web coloring ensures that webs that share common procedures are assigned different registers. For this example, all four webs can be colored using just two callee-saves registers. Different webs for the same variable may be assigned different registers (e.g. Web 4 and Web 2 for global variable **g2**).

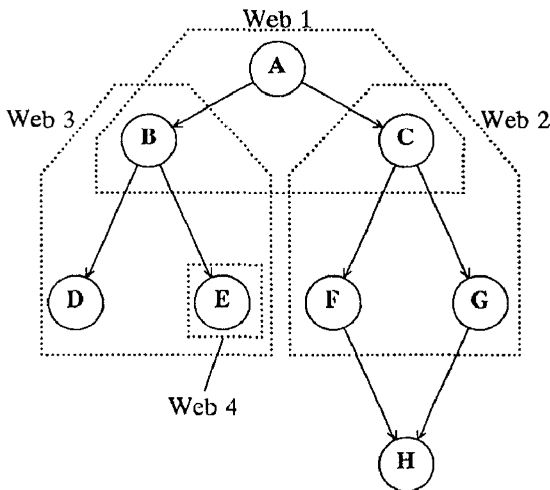

Figure 3

Web	Variable Name	Nodes In Web	Interfering Webs	Promote To Register
1	g3	A B C	2 3	r1
2	g2	C F G	1	r2
3	g1	B D E	1 4	r2
4	g2	E	3	r1

Table 2

Procedure **B** is an entry procedure of Web 3. The compiler second phase will insert code at the entry point of procedure **B** to load the value of variable **g1** from memory into the designated register. At the exit point of procedure **B**, code will be added to store the contents of that register back to the memory location associated with variable **g1**.

4.2. Spill Code Motion

RISC architectures that do not employ a register windowing scheme tradeoff hardware implementation cost for register spill overhead at procedure calls [Wall 88]. Minimizing this overhead in software is the primary motivation for compile-time spill code minimization. In the technique presented here, callee-saves register save and restore code is elevated in the call graph so that descendant nodes may use them for free, without the cost of spilling them - hence the term *spill code motion*.

In previous attempts to minimize spill code at procedure calls, procedures are compiled in bottom-up order, favoring routines close to the bottom of the call graph.[3] Even in relatively small graphs, it is likely that the "free" registers will be quickly exhausted. To minimize spill code at different levels of the call graph, we have developed a technique which first identifies regions of the call graph called *clusters*.

4.2.1. Clusters

Informally, clusters correspond to areas of the call graph where spill code motion is expected to be effective. The standard register linkage convention is adhered to at the boundary of a cluster, but not internally. More precisely, a *cluster* is defined as a set of nodes in the program call graph with the following properties:

[1] There exists some node **R**, called the root of the cluster, which dominates all other nodes within the cluster. (Node **D** *dominates* node **N** if and only if every path from each start node to **N** includes **D**.)

Note that this does not imply that all nodes dominated by the root node are in the cluster.

[2] For every node **P** in the cluster except **R**, all immediate predecessors of **P** are also in the cluster.

[3] A non-root node is included only in the cluster of the immediately dominating root node. That is, a node dominated by more than one root node is only included in the cluster associated with the nearest root.

The root nodes of clusters will save and restore callee-saves registers so that other nodes in the cluster can use them without incurring this expense. Cluster root nodes are identified by considering the estimated call frequencies along the edges of the call graph. If the internal cluster nodes are called more frequently than the root node, fewer instructions and fewer memory references will be required to execute the program.

Consider the example in Figure 4. Assume **R** is a cluster root node while **S** and **T** are members of that cluster. Normally, any callee-saves registers used in S must be saved at the entry point of S and restored at the exit point. The program analyzer can arrange for those callee-saves registers to be spilled at the entry point of the cluster root, **R**. S may then

3. Bottom-up register allocation is well suited to programs that spend most of their execution time in the lowest regions of the call graph. A top-down algorithm would favor procedures near the top of the call graph.

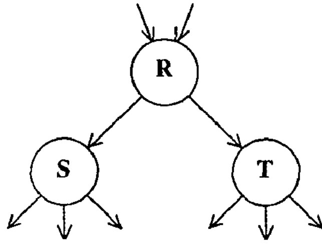

Figure 4

use the registers without spilling them as long as **R** does not use the same registers to hold values across a call to **S**. This results in a performance gain, assuming **S** is called more frequently than **R**. The spill code can be reduced further if sibling nodes within the cluster use the same registers. For example, **R** could spill a single set of registers that could be used by both **S** and **T**.

Note that the definition of a cluster allows leaf nodes of a cluster to be root nodes of other clusters. This facilitates spill code motion upwards in the call graph across clusters. For example, node **R** in Figure 4 could also belong to a cluster rooted higher up in the call graph. Spill code that is moved out of nodes **S** and **T** into node **R** can be moved further up to the root node of the cluster containing node **R**.

4.2.2. Identifying Clusters

Clusters are found in a depth-first traversal of the program call graph. Cluster root nodes are identified using a heuristic that compares incoming call counts with call counts to potential cluster member nodes. Successors nodes that meet the criteria for inclusion in the cluster, are recursively added to the cluster. The cluster identification algorithm is outlined in Figure 5.

At the end of this algorithm, for each cluster root node R, Cluster_Nodes[R] specifies the remaining nodes that belong to that cluster. Note that a cluster root node can itself appear in Cluster_Nodes of a higher level cluster root.

The function *Postpone_Visit* is used in the cluster identification algorithm to defer consideration of a node until all its predecessors have been visited. The exception to this rule is when the unvisited predecessors of a node are part of a recursive call chain involving that node.

For correctness, the algorithm outlined in Figure 5 is designed to disallow recursive call cycles within clusters.[4] Consider a recursive routine which uses callee-saves registers. Such a routine would expect that the values in these registers would remain safe across the recursive call. If we eliminate the spill code at a recursive routine's entry and exit points, however, we will destroy the values that were live across that recursive call.

4. Note, however, that it does not limit clusters from being identified within cycles in the call graph. The example shown later in Figure 7 illustrates this.

```
Find_Clusters( G )
begin
for each node P in the call graph G do
  begin
  mark P as not visited;
  Cluster_Nodes[P] := Ø;
  end;
for each start node S in the call graph do
  Examine_Node( S, NULL );
end;

Examine_Node( P, ClusterRoot )
begin
if Postpone_Visit(P) then
  return;
mark P as visited;
if ( (ClusterRoot ≠ NULL) and
     (ClusterRoot dominates P) and
     (all predecessors of P ∈ Cluster_Nodes[ClusterRoot]) )
  then
     add node P to Cluster_Nodes[P];
if P is a cluster root then
  NearestClusterRoot := P;
else
  NearestClusterRoot := ClusterRoot;
for each successor S of P do
  if S has not been visited then
     Examine_Node( S, NearestClusterRoot );
end;

boolean Postpone_Visit( P )
begin
for each predecessor Q of P do
  if ( Q has not been visited and
       Q is not a descendant node of P )
  then
     return TRUE;
return FALSE;
end
```

Figure 5

4.2.3. Register Usage Sets

To facilitate spill code motion, the program analyzer identifies how each general register may be used within each procedure of a cluster. This is accomplished by placing each register into one of four sets.

FREE[P] registers in this set need not be saved on entry and restored on exit if they are used in procedure P, and may hold live values across calls.

CALLER[P] registers in this set need not be saved on entry and restored on exit if they are used in procedure P, but may not hold live values across calls.

CALLEE[P] registers in this set must be saved on entry and restored on exit if they are used in procedure P, but may hold live values across calls.

MSPILL[P] registers in this set must be saved on entry and restored on exit if they are used in procedure P and they may not hold live values across calls.

The register sets for each procedure are saved in the program database. The register allocator in the compiler second phase must use each register according to the properties of the set to which it belongs.

Our algorithm adds the additional requirement that all registers in the MSPILL set at a cluster root node must be saved on entry and restored on exit, regardless of whether they are actually used inside that procedure. This will accomplish our goal of having the root node execute the spill code for the remaining nodes of the cluster. In fact, the MSPILL sets will contain registers only for cluster root nodes.

4.2.4. Computing Register Usage Sets

The MSPILL and FREE sets are first initialized to the empty set and CALLER and CALLEE are initialized to the standard caller-saves and callee-saves registers, respectively, for all procedures.

To compute the final register usage sets, each cluster root node is considered in a bottom-up ordering. As each cluster root is considered, we first find the set of registers that are in the MSPILL set of any cluster root node that is also a member of the current cluster.

This set is used to determine the order in which free registers will be selected in the current cluster. Registers not in the set will be selected first to increase the chances that we will be able to move registers from the MSPILL set at the child cluster root to the MSPILL set of the current cluster root. Moving registers from one MSPILL set to another effectively causes the spill code for one cluster's registers to be executed at a root node higher up in the call graph.

To assist in computing the register usage sets, for each cluster procedure P, the set AVAIL[P] is defined to be the subset of the standard callee-saves registers that are available for free use along calls out of P.

Recall that the compiler first phase estimated the number of callee-saves registers needed by each procedure and saved this information in the summary file. Using this estimate, we select a set of callee-saves registers for use in the cluster root procedure, R, and assign this set to CALLEE[R]. AVAIL[R] is then initialized to be the standard callee-saves registers that are not in CALLEE[R].

The set of registers that have been reserved at some node of the cluster for a promoted global variable will also be removed from AVAIL[R], conservatively preventing these registers from being used for other purposes at all nodes in the cluster.

The recursive procedure *Preallocate_Node*, shown in Figure 6, is then called on the root node. This procedure visits each node of the current cluster to preallocate the set of callee-saves registers that will be available in that node and arranges for the root node to spill that register, if possible.

When *Preallocate_Node* completes its pass over a cluster, all registers in the USED set are placed in the MSPILL[R] set. We then augment the CALLER set at each node of the cluster with the following algorithm:

```
for each node Q ∈ Cluster_Nodes[R] do
  if ( Q is not a cluster root ) then
    CALLER[Q] := CALLER[Q] ∪
                (AVAIL[Q] ∩ MSPILL[R]);
```

```
define USED : set of callee-saves registers;
  (* Registers in USED will be added to the
     MSPILL set of the current cluster root. *)

Preallocate_Node( N )
begin
mark N as visited;
AVAIL[N] :=
  ∩ AVAIL[P] over all immediate predecessors P of N;

if ( N is a cluster root ) then
  begin
  if ( N is not the current cluster root ) then
    begin
    USED := USED ∪ ( MSPILL[N] ∩ AVAIL[N] )
    MSPILL[N] := MSPILL[N] - AVAIL[N];
    USED := USED ∪ ( CALLEE[N] ∩ AVAIL[N] );
    FREE[N] := CALLEE[N] ∩ AVAIL[N];
    CALLEE[N] := CALLEE[N] - FREE[N];
    end
  end
else
  begin
  FREE[N] :=
    Get_Registers( Num_Regs_Needed( N ), AVAIL[N] );
  AVAIL[N] := AVAIL[N] - FREE[N];
  CALLEE[N] := CALLEE[N] - ( FREE[N] ∪ AVAIL[N] );
  USED := USED ∪ FREE[N];
  end;
for each immediate successor S of N do
  if ( S ∈ CurrentCluster and
    all immediate predecessors of S have been visited )
    then
      Preallocate_Node( S );
end;

Get_Registers( COUNT, REGS )
begin
select up to COUNT registers from the set REGS,
using the priority order determined for this cluster;
return the set of selected registers;
end;
```

Figure 6

This post-pass enables callee-saves registers spilled at the cluster root node to be used as caller-saves registers at intermediate nodes of certain paths within the cluster. In Figure 7, assume procedure **J** is a cluster root node while **K**, **L**, **M** are members of the cluster. For a register to be in FREE[M], it must have also been in AVAIL[K] and AVAIL[L]. Since **J** is the cluster root, all registers in FREE[M] will be in MSPILL[J]. From the definition of MSPILL, **J** must spill these registers. Hence, they can be safely used as caller-saves registers inside **K** and **L**.

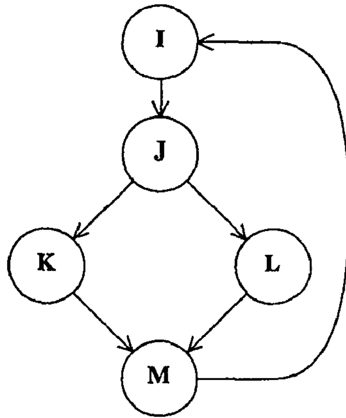

Figure 7

4.3. The Program Database

The program analyzer algorithms result in a set of register allocation directives for each procedure in the program. These directives are represented as program database entries for each procedure that contain the following information:

- A list of global variables promoted in that routine. For each promoted variable, the register reserved for that variable is specified along with a set of flags indicating conditions such as whether or not this procedure is a web entry node for the promoted variable.
- The contents of the FREE, MSPILL, CALLER, and CALLEE sets for the procedure.

Since these register directives are precomputed for each procedure by the program analyzer, the register allocator can be run on procedures in any order during the compiler second phase. Also, since the directives are stored in a single program database, the compiler second phase can be run on each source module independently.

5. The Compiler Second Phase

The compiler second phase reads in the intermediate representation generated by the first phase and performs low-level code generation and traditional global optimizations. For each procedure, the relevant program database entry is queried to implement interprocedural register allocation, as computed by the program analyzer.

The steps involved in implementing global variable promotion are outlined below.

- For each procedure belonging to a promoted web, memory references to the corresponding global variable are converted into register references.

  This can enable additional intraprocedural optimizations such as register copy elimination.

- For procedures corresponding to web entry nodes, the global variable is loaded into the callee-saves register at the entry point and stored back at the exit point. Care is taken to insert loads and stores to promoted static global variables correctly.

  Note that if none of the procedures belonging to a promoted web modify the global variable, a store instruction need not be inserted at the web entry procedures.

- Register save/restore code for the callee-saves register is suppressed at all procedures belonging to the promoted web except at web entry nodes.

  Note that the dedicated callee-saves register will not be available for intraprocedural use in procedures belonging to the promoted web since it will not be contained in any of the register usage sets.

Implementing spill code motion is simpler. The CALLER set for the current procedure is examined to obtain caller-saves registers for local coloring. For callee-saves registers, the FREE set is checked before the CALLEE set. Spill code for registers in the MSPILL set is inserted at the entry point and exit point of procedures that are cluster root nodes. Spill code is also generated for callee-saves registers obtained from the CALLEE set.

6. Measurements

The interprocedural register allocation techniques described in this paper have been implemented in a prototype program analyzer consisting of about 6000 lines of C source code. The compiler first and second phases have been implemented by modifying the PA-RISC C compiler [Coutant 86] running under the HP-UX operating system.

In the prototype implementation, the compiler first phase was allowed to proceed through the normal code generation and optimization phases before generating summary files. This was done to obtain better heuristic information on usage counts for global variables, call frequencies and estimates for callee-saves register requirements. Usage counts and call frequencies were determined based on the location of each reference or call in the control flow hierarchy.

The compiler second phase can be designed to accept either an intermediate representation or the original programming language text. In our implementation, the compiler second phase uses the original program text file. Command-line options are used to specify which phase of the compiler is to be run. This helps minimize changes to existing compile scripts, at the expense of compiling each source file twice.

The C compiler and program analyzer were used to optimize the programs listed in Table 3, with both interprocedural global variable promotion and spill code motion. The benchmarked applications were linked with C run-time libraries which were not interprocedurally optimized. Also, interprocedural register allocation was restricted to callee-saves *general* registers.

Benchmark Programs

Name	Lines Of Code	Description
Dhrystone	380	Popular CPU benchmark
Fgrep	460	Text pattern matching tool
Othello	800	Game program
War	1500	Game program
CR Tool	2700	Prototype code repositioning tool
Proto C	6600	A fast C compiler, compiling itself
PA Opt	85000	PA optimizer, optimizing Othello

Table 3

6.1. Performance Improvement

Table 4 shows the percentage improvement over level two (global) optimization for the benchmarks with various options enabled in the program analyzer. Most of the measurements shown in Table 4 were obtained using a PA-RISC simulator. These simulations did not model a cache, so some of the benefits of interprocedural register allocation are not accounted for here. Obviously, the extent of this benefit will vary with differing cache parameters and placement algorithms.

Percentage Performance Improvement
Over Level 2 Optimization

Total cycles measured by a simulator, excluding cache miss penalties.
‡ Benchmarks measured with /bin/time on an HP 9000 Model 835

Benchmark	A	B	C	D	E	F
Dhrystone	0.8	0.8	3.4	3.4	5.5	3.4
Fgrep	0.0	0.0	8.8	8.4	8.6	8.8
Othello	0.1	0.0	4.8	4.8	4.7	4.9
War	1.2	1.2	3.7	3.7	3.7	3.7
CR Tool	0.0	0.0	2.2	1.5	0.8	2.3
Proto C‡	n/a	n/a	18.7	9.1	18.7	n/a
PA Opt‡	6.0	6.0	9.0	7.0	7.0	9.0

A = Spill motion only
B = Spill motion w/profile info
C = Spill motion & 6 reg coloring
D = Spill motion & greedy coloring
E = Spill motion & blanket promotion
F = Spill motion & 6 reg coloring w/profile info

Table 4

The first column of data gives the results for spill code motion alone. The results in the second and last columns were obtained by making profile information available to the program analyzer. The profile information was collected by running *gprof* [Graham 82] and provided the program analyzer with actual counts of run-time procedure calls.

The remaining columns combine the results of spill code motion with global variable promotion. For the third column, 6 callee-saves registers were made available for web coloring. For the fourth column, a "greedy" coloring algorithm was used which tries to color as many webs as possible without reserving any of the callee-saves registers required for any individual procedure.

"Blanket" promotion refers to the promotion of a set of user specified global variables across the entire program. This was implemented in the program analyzer to measure the advantages of a web coloring technique over dedicating a register for each key global variable over the whole program as in [Wall 86]. The 6 most frequently used global variables (as determined by analyzing the prioritized web list) were chosen for blanket promotion.

6.2. Analysis

Overall, Table 4 indicates that interprocedural register allocation is beneficial. The improvements for the *Proto C* benchmark are exceptionally high because this application was coded specifically to take advantage of global register variables.

Spill code motion typically provides a small reduction in instructions executed; global variable promotion has a larger impact. This is consistent with results reported in [Wall 86].

In our scheme, interprocedural global variable promotion has additional benefits beyond just reducing the number of dynamic memory references. In particular, the compiler second phase is able to remove instructions that set up the base register used in the deleted memory references to promoted globals. Certain register copies involving promoted globals are also eliminated.

Spill code motion as implemented in our prototype depends, among other things, on the size and shape of the clusters identified. The shorter and wider the cluster, the easier it is to move spill code to the root node. For the applications considered, the average cluster size ranged between 2 to 4 nodes. The small average cluster size is, in part, responsible for the marginal performance benefit observed. Note that the performance improvement observed for the PA Optimizer is probably due to external cache effects since it is not corroborated by the reduction in dynamic memory references (illustrated later in Table 5).

The figures in Table 5 are inconclusive with respect to the utility of procedure-level profile information in our algorithms. This is partly because the prototype program analyzer is able to use heuristic call counts effectively. The program analyzer normalizes the raw heuristic call counts obtained from the summary files over the entire program call graph, increasing the weights on recursive arcs and arcs to leaf nodes. Furthermore, procedure-level profile information is not fine-grained enough to make a large difference in global variable promotion.

As expected, web coloring has little or no advantage over blanket promotion on the small CPU benchmarks. In larger applications, which typically have a larger number of global variables, web coloring is advantageous.

In the case of the PA Optimizer, the 500 global variables eligible for register promotion were broken down into 1094 webs, of which 489 webs were considered for coloring. The remaining webs were discarded either because they were too sparse (low ratio of L_REF nodes to total nodes) or because the web contained just one node in which the corresponding global variable was accessed infrequently. Of the 489 webs, 280 were successfully colored using just 6 registers reserved for interprocedural use.

Greedy coloring did not do as well as 6 register coloring for some of the larger applications. With this strategy 309 webs out of 489 were colored in the PA Optimizer. However, it failed to color some of the more important webs that were colored with 6 register coloring.

6.3. Memory Reference Reduction

Table 5 shows the percentage reduction in singleton memory references executed in a subset of the benchmarks. A singleton memory reference is roughly defined as an access of a simple variable (not an element of an array or structure) and whose size is one, two, or four bytes.

Table 5 indicates that global variable promotion is effective in eliminating many of the singleton memory references remaining after normal level two optimization. How much this impacts overall performance depends on the percentage of the total memory references that are eliminated. Note that interprocedural register allocation will not reduce the number of references to elements of arrays and other data structures.

Percent Reduction in Dynamic Singleton Memory References (Over Level 2 Optimization)

Benchmark	A	B	C	D	E	F
Dhrystone	14.0	14.0	25.6	25.6	41.9	25.6
Othello	0.0	-0.9	20.8	20.8	20.8	20.2
War	10.3	10.3	21.4	21.4	21.4	21.4
Fgrep	0.0	0.0	67.0	64.3	66.0	67.0
CR Tool	0.0	0.1	7.8	7.0	1.7	8.2
PA Opt	4.2	5.2	13.9	8.3	0.8	13.5

Table 5

7. Limitations and Extensions

7.1. Two-pass Compilation

The interprocedural register allocation scheme described in this paper improves register usage across module boundaries by using a two-pass approach. This two-pass approach has the following limitations:

- It requires management of additional files (summary files, intermediate files and program database files).
- There is additional compile-time overhead. Aside from the execution time of the program analyzer, the compiler second phase has to query the program database and either read in the intermediate files or re-process the source files.
- Source level changes need to be tracked carefully and can be very expensive.
- Interprocedural register allocation is not performed across calls to run-time libraries.

Most of the limitations associated with a two-pass approach can be circumvented by deferring interprocedural register allocation to link-time as described in [Wall 86]. The linker would need to perform the job of the program analyzer and implement interprocedural register allocation by re-writing each module appropriately. Module re-writing may be accompanied by certain local optimizations (e.g. peephole optimization, and instruction scheduling).

Alternatively, interprocedural register allocation could be performed in the context of a programming environment such as R^n [Cooper 86]. The module editor used to create source files could generate approximate summary information. The program compiler could compute interprocedural register allocation information. This information could be communicated to the module compiler, which would serve as the compiler second phase.

7.2. Partial Call Graphs

The methods described in this paper, can be applied to partial call graphs, where not all procedures and global variable references are exposed to the program analyzer. This situation might correspond to optimizing a set of run-time library sources. The program analyzer would be forced to make conservative assumptions about externally visible procedures and variables in this case. Alternatively, additional compiler directives could be supplied for the programmer to designate the non-exported procedures and variables.

Assuming that summary files for run-time libraries are not available, the program analyzer will rarely see the *whole* call graph. As long as the following assumptions hold true, partial call graphs pose no problems.

- Incoming calls are made only to the start nodes (nodes with no predecessors) of the partial call graph.
- Outgoing calls (e.g. to run-time library routines) return normally without invoking directly or indirectly a routine belonging to the partial call graph.
- Global variables eligible for interprocedural register promotion are not accessed outside the partial call graph.

There are cases involving the C run-time library routines when the last two assumptions may not hold.[5] When these assumptions are violated, the dataflow sets computed for global variable promotion will be invalid, potentially leading to incorrect optimizations. The program analyzer needs to be informed if any of the default assumptions are violated.

7.3. Indirect Calls

As mentioned earlier, the compiler first phase identifies procedures whose addresses have been computed as well as procedures that can make indirect calls. The program analyzer conservatively assumes that every procedure whose address has been computed can be a target of any indirect procedure call. This ensures that the dataflow set used in web construction are correct. It is still possible to promote webs and identify clusters that contain procedures that make indirect calls.

7.4. Statics

Global variables and routines declared to be *static* in C are private to individual modules. Since different modules may define identically named static global variables and routines, static identifiers need to be sufficiently qualified by the compiler first phase.

A more subtle complication is introduced by register promotion of static global variables. In particular, a web for a static global variable may need to be expanded to include an entry node defined in a different module. If such a web is promoted, the compiler second phase will be unable to correctly insert instructions at the entry routine to load and store a static global belonging to another module. The program analyzer could circumvent this problem by simply discarding such webs.

7.5. Heuristics

The program analyzer can use profile information instead of the heuristic call counts computed by the compiler first phase. Having accurate call counts can help cluster identification, but is of lesser value for global variable promotion. In prioritizing webs for coloring, we currently weight the heuristic reference counts for global variables within each web procedure by the call count for that procedure. A profiler that combines execution counts for each optimized basic block with the number of remaining global variable accesses in the basic block will be able to provide more accurate information.

If the user has previously optimized the application (e.g. for profiling), the number of callee-saves registers need for each procedure would be known. Otherwise, without performing code generation and optimization, it is difficult for the compiler first phase to accurately estimate register need for each procedure.

5. The *qsort()* routine is passed the address of a user (comparison) routine that can be invoked indirectly. Also, global variables such as *_ctype* can be referenced both in user code and run-time library routines.

7.6. Extensions

Interprocedural register allocation is applicable to languages other than C, and architectures other than the load-store RISC variety. Spill code motion may not be of use on an architecture that provides hardware support for creating a new register context at a procedure call. Interprocedural global variable promotion, however, may improve application performance on any machine providing storage locations with faster access time than main memory.

7.6.1. Global Variable Promotion

There are several opportunities for extending the global variable promotion strategy. One possibility is to split large but sparse global webs before coloring. A web with isolated variable references at two ends of a long call chain can be broken down into two smaller webs that more tightly encapsulate the procedures containing the references. If the higher level web is colored, the promoted global variable would need to be saved and restored around certain external calls.

Splitting webs in this manner effectively reduces the number of web interferences, allowing more nodes of the interference graph to be colored. Alternatively, independent webs of a global variable can be re-merged to allow sharing of entry nodes, at the expense of extra interferences.

Another possibility is to introduce stub routines to load and store promoted global variables instead of expanding webs that have entry nodes with both internal and external predecessors. All incoming calls to the web entry node from external predecessors would be re-directed through the stub routines while incoming calls from internal predecessor would continue to go directly to the web entry node. Stub routines can also be inserted between internal web nodes and external callers. Using stub routines avoids having to reserve callee-saves registers in additional web entry procedures. However, sharing such stub routines among different promoted global variables can become complicated.

By virtue of having done interprocedural dataflow analysis, the program analyzer can provide better than worst-case aliasing information for the optimization algorithms in the compiler second phase. The L_REF and C_REF sets for a called procedure can be examined to determine which of the global variables that were eligible for interprocedural promotion may actually be referenced.

7.6.2. Spill Code Motion

Improvements to the spill code motion algorithm are also possible. First, the heuristics used to decide whether a node is a candidate cluster root can be refined. In the prototype program analyzer, the incoming call counts to a potential root node are compared to the outgoing call counts to immediate successors that are dominated. We could also account for factors such as register need and call counts to dominated grandchild nodes.

Our program analyzer implementation performs spill code motion after global variable promotion. Before pre-allocating callee-saves registers within a cluster, the program analyzer determines if global variables have been promoted over any nodes of the cluster. Callee-saves registers dedicated to global variables over any portion of the cluster are taken out of the AVAIL set at the cluster root node and not considered for pre-allocation. This is more conservative then necessary. Instead, we can simply remove callee-saves registers reserved for global variables from the FREE and AVAIL sets at web nodes that are part of the cluster. This would allow such callee-saves registers to be pre-allocated along paths within the cluster where the global variable is not live.

Another possible improvement to the algorithm in Figure 6 would allow more free registers along some paths within clusters. This enhancement can be understood by considering the example shown earlier in Figure 7. Assume we wanted to pre-allocate one callee-saves register to each of **K** and **M** while two registers are needed at **L**. The current algorithm will result in the following FREE sets, where r1, r2, and r3 are some arbitrary callee-saves registers:

FREE[K] = { r1 } FREE[L] = { r1, r2 } FREE[M] = { r3 }

Since r2 will be included in MSPILL[J] and it is not used in **M**, it could be added to FREE[K]. **K** would then have an additional callee-saves register that it could use without executing spill code.

The spill code motion algorithm we have described has focussed on callee-saves register spill. The program analyzer could also pre-allocate caller-saves registers (based on information provided by the compiler first phase) in a bottom-up order, as described in [Chow 88]. The total caller-saves register usage for the call tree rooted at each procedure can be communicated to the compiler second phase. This would allow the compiler second phase to keep live values in caller-saves registers across calls that don't make use of those caller-saves registers. This strategy would be subject to the limitation of not being able to effectively exploit such register usage information for procedures found in recursive call chains and at indirect call sites.

8. Conclusions

A two-pass compilation system and program analyzer tool have been presented that facilitate interprocedural register allocation in programs built from multiple modules.

A graph coloring technique has been applied to webs of eligible global variables in the call graph. This method allows a single callee-saves register to be used for different promoted global variables in disjoint regions of the call graph. Web coloring results in a larger number of promoted variables and fewer executed memory references than previous methods which dedicate a register to a promoted variable over the entire program.

A clustering technique has been applied to nodes in the call graph to allow migration of callee-saves register spill code upwards in the call graph to infrequently executed procedures. Since clusters may be found at all levels of the call graph, this approach can result in a greater reduction in executed spill code than one which favors the procedures near the bottom of the call graph.

The combined results for the two algorithms have been positive. Generally, between 2 to 9 percent fewer machine cycles were executed, excluding the additional savings possible due to reduced cache misses. These results were obtained for a small collection of C programs for a RISC-based architecture. As expected, keeping global variables in registers across procedure calls was found to be of greater benefit than moving spill code out of frequently called procedures.

Additional large, representative benchmarks need to be measured to fully characterize the overall performance benefits. However, our results show, that these techniques can measurably improve the use of limited hardware resources.

Acknowledgements

We would like to thank the many people at Hewlett-Packard who gave us advice and guidance during the course of our work. Key contributions were made by Steve Saunders, Carol Thompson, Deborah Coutant, Karl Pettis, Bob Hansen, Ron Rasmussen, Marc Sabatella, Richard Holman, Cary Coutant, Jon Kelley, Sunil Jain, Tim Pasek, and Joseph Coha. A special note of thanks goes to Mark Laventhal who managed the project, as well as Larry Rosler, Jim Schultz, and Linda Lawson for their management support.

References

[Chaitin 82] G. J. Chaitin, "Register Allocation and Spilling via Graph Coloring", *Proceedings of the SIGPLAN '82 Symposium On Compiler Construction, SIGPLAN Notices*, Vol. 17, No. 6, June 1982, pages 98-105.

[Chow 84] Fred Chow and John Hennessy, "Register Allocation by Priority-based Coloring", *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices*, Vol. 19, No. 6, June 1984, pages 222-232.

[Chow 88] Fred C. Chow, "Minimizing Register Usage Penalty at Procedure Calls", *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, July 1988, pages 85-94.

[Cooper 86] Keith D. Cooper, Ken Kennedy, and Linda Torczon, "The Impact of Interprocedural Analysis and Optimization in the R^n Programming Environment", *ACM Transactions on Programming Languages and Systems*, October 1986, pages 491-523.

[Coutant 86] Deborah S. Coutant, Carol L. Hammond, and Jon W. Kelley, "Compilers for the New Generation of Hewlett-Packard Computers", *Hewlett-Packard Journal*, Vol. 37, no. 1, August 1986, pages 4-18.

[Graham 82] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. "gprof: a call graph execution profiler", *Proceedings of the SIGPLAN'82 Symposium on Compiler Construction*, June 1982, pages 120-126

[Lee 89] Ruby B. Lee, "Precision Architecture", *Computer;*, January 1989, pages 78-91.

[Mahon 86] M. Mahon, et al, "Hewlett-Packard Precision Architecture: The Processor", *Hewlett-Packard Journal*, Vol. 37, no. 8, August 1986, pages 4-21.

[Mulder 89] Hans Mulder, "Data Buffering: Run-Time Versus Compile Time Support", *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pages 144-151.

[Steen 87] P. A. Steenkiste, *LISP on a Reduced Instruction Set Processor: Characterization and Optimization*, PhD Thesis, Stanford University Computer Systems Laboratory, March 1987, Chapter 5.

[Wall 86] David W. Wall, "Global Register Allocation At Link Time", *Proceedings of the SIGPLAN '86 Symposium On Compiler Construction, SIGPLAN Notices*, Vol. 21, No. 7, July 1986, pages 264-275.

[Wall 88] David W. Wall, "Register Windows vs. Register Allocation", *Proceedings of the SIGPLAN '88 Conference On Programming Language Design and Implementation, SIGPLAN Notices*, Vol. 23, No. 7, July 1988, pages 67-78.

Hewlett-Packard Company has filed a patent application on some of the methods discussed in this paper.