Minimum Cost Interprocedural Register Allocation

Steven M. Kurlander   Charles N. Fischer*
University of Wisconsin—Madison†

Abstract

Past register allocators have applied heuristics to allocate registers at the local, global, and interprocedural levels. This paper presents a polynomial time interprocedural register allocator that models the cost of allocating registers to procedures and spilling registers across calls. To find the minimum cost allocation, our allocator maps solutions from a dual network flow problem that can be solved in polynomial time. Experiments show that our interprocedural register allocator can yield significant improvements in execution time.

1 Introduction

Effectively using registers can significantly decrease the execution time of a program. Common policy in current compilers using only intraprocedural register allocation is to spill at call sites registers that might be used by both the caller and callee[CHKW86].

The goal of *inter*procedural register allocation is to minimize execution time given the register requirements of individual procedures in a program. Based on these requirements, an interprocedural register allocator selects which registers are available to each procedure and, correspondingly, around which calls registers are spilled. An interprocedural allocator aims to spill registers across infrequent calls (or not at all).

This paper presents both a *save-free* interprocedural register allocator (which never spills registers across calls), and an interprocedural register allocator that spills registers as necessary across calls. Our save-free allocator models the cost of allocating registers to procedures and finds a minimum cost allocation. A profile is used to estimate the benefit of allocating different levels of registers to each procedure.

Our interprocedural register allocator that spills registers across calls minimizes the cost of allocating registers to procedures as well as spill cost. The cost of spilling a register across a call is a function of the call's frequency. Register spilling allows registers to be reassigned along a path in the call graph when profitable.

To generate a save-free interprocedural register allocation of a call graph, we use Cameron's algorithm for finding a maximum weight k-antichain in a partially ordered set [Cam85]. To find a maximum weight k-antichain, Cameron maps solutions from a dual minimum cost flow problem[1]. A dual minimum cost flow problem can be transformed into a minimum cost flow problem and solved in polynomial time. In Section 4, we generalize our allocation model to allow for register spilling across calls. To find these allocations, we map solutions from a more general dual minimum cost flow problem.

Our approach can be used with conventional compilers that translate one procedure at a time. Each procedure may be translated using any of the well-known, high-quality, intraprocedural register allocators[BCKT89][CK91][PF92]. Then using profile information our minimum cost interprocedural register allocator determines how many registers each procedure will be given and where spills will be placed. A minimum cost interprocedural register allocation may not allocate registers to all locals in a procedure. For each of these procedures, an intraprocedural register allocator will generate a revised allocation using the procedure's interprocedurally allocated registers and the temporary registers available to each procedure.

The algorithm we describe can be part of a more general interprocedural register allocator. Such an interprocedural allocator can select global candidates to be allocated registers across procedure calls. In addition, an interprocedural allocator need not follow a predefined parameter passing convention. This allows the allocator to pass additional parameters in registers as well as to choose which registers to use (the registers selected need not always be caller-save).

2 Related Work

Past interprocedural register allocators have relied on heuristics. Wall [Wal86] observes that two procedures that are not simultaneously active can share the same registers for their locals. With this in mind, Wall groups locals that can be

*This work was supported by NSF grants CCR-9122267 and CCR-9505922.

†Authors' address: Dept. of Computer Sciences, 1210 W. Dayton St., Madison, WI 53706. Email: {smk,fischer}@cs.wisc.edu

[1] Cameron refers to the dual minimum cost flow problem as a dual transportation system of linear inequalities. When we transform this dual problem into the primal problem, the network flow graph is not bipartite and, thus, the primal problem is not a transportation problem. However, the primal problem is a minimum cost flow problem. We, therefore, refer to the dual as a dual minimum cost flow problem.

assigned a common register. The locals of a procedure are always placed in different groups than those of its descendants and ancestors in a call graph. In addition, each interprocedurally shared global is placed in a singleton group, as globals are allocated registers throughout the entire program. Groups are then allocated registers based on the total frequency in which their members are referenced. Walls' allocator may not find the best allocation with respect to his model, since he allows locals infrequently referenced to be grouped together with locals frequently referenced.

Steenkiste and Hennessy [SH89] design an interprocedural register allocator for LISP programs. Their approach allocates registers to locals in a bottom-up fashion over the call graph. Since they find that LISP programs tend to spend their time in the leaf procedures of a call graph, their method first allocates registers in the leaves. While registers are available, a procedure is assigned registers that are not already assigned to its descendants in the call graph. When the registers are exhausted, they switch to an intraprocedural allocation. This approach may introduce register spilling around calls in frequently executed procedures near the top of a call graph. The approach we propose avoids register spilling across frequently executed calls.

Santhanam and Odnert [SO90] perform interprocedural register allocation over *clusters* of frequently executed procedures. Their heuristic aims to move spill code to the root node of a cluster. The approach we propose examines the *entire* call graph to generate a minimum cost allocation spilling registers as inexpensively as possible.

3 Save-free Interprocedural Register Allocation

In this section, we describe a save-free interprocedural register allocator that determines the number of registers to allocate to the locals of each procedure for acyclic call graphs (cycles in call graphs normally force saves across recursive calls). Our solution is based on Cameron's algorithm for finding a maximum weight k-antichain in a partially ordered set [Cam85]. In Section 4, we generalize our allocation model to compute a minimum cost allocation that may include register spilling across calls in (possibly cyclic) call graphs.

For each procedure, we assume an intraprocedural register allocator has already grouped locals that can be assigned the same register. We refer to each group as a *register candidate*. An interprocedural register allocator selects which candidates are allocated registers. Each procedure has a few temporary registers available. Locals assigned these registers do not require interprocedurally allocated registers. These locals are correctly allocated at register allocation time.

Initially, we assume that a register candidates is live across all calls in a procedure. However, in Section 6 we distinguish between candidates not live across calls and candidates live across one or more calls.

Our interprocedural register allocator may give fewer registers to a procedure than the number of candidates it has. An intraprocedural register allocator can produce a valid allocation when given fewer registers. The intraprocedural register allocator will spill values internally as necessary.

We assume there is a positive benefit associated with allocating a register to a register candidate. As more candidates are allocated registers the benefit of the allocation increases (and, equivalently, the cost associated with the allocation decreases). In our interprocedural register allocation, a benefit estimates the decrease in loads and stores from allocating a register to a candidate. Given k registers, our save-free interprocedural register allocator selects an allocation in which the benefits of register allocated candidates sum to a maximum (across all procedures). That is, registers are given to procedures that benefit the most.

3.1 Defining a partial ordering on the candidates of a call graph

Let $G = (P, E)$ be an acyclic call graph, where P is a set of procedures and E is a set of call edges. We represent the calls from procedure $P_v \in P$ to $P_w \in P$ as a single edge in the call graph. Let S be the set of register candidates in P. For procedure $P_v \in P$, let $C(P_v)$ be the set of register candidates in P_v.

We define the following partial order ($\sqsubseteq$) on candidates in an acyclic call graph such that there is an ordering between two candidates if and only if they cannot be assigned the same register in a save-free interprocedural register allocation:

1. In the partial order, assume the relation between candidates in a procedure is an arbitrary chain; that is, there is an ordering between every two candidates in the same procedure.
2. Let $c_v \in C(P_v)$, $c_w \in C(P_w)$, and $P_v \neq P_w$. If there is a path from procedure P_v to P_w, then $c_w \sqsubset c_v$.

For $c_i, c_j \in S$, $c_i \sqsubset c_j$ is defined as $c_i \sqsubseteq c_j$ and $c_i \neq c_j$. Given (1) and (2), there is an ordering only between two candidates of the same procedure or between candidates in separate procedures connected along a path in the acyclic call graph. Thus, either $c_v \sqsubset c_w$ or $c_w \sqsubset c_v$ for candidates $c_w, c_v \in S$ if and only if c_w and c_v cannot be assigned the same register in the call graph.

In Figure 1(a), procedure P_w has two candidates, p and q, and procedure P_x has two candidates t and v. A partial order on the candidates in the call graph appears in (b). We assume the ordering between p and q is $q \sqsubset p$ and the ordering between t and v is $v \sqsubset t$. Since P_v calls P_w, $q \sqsubset m$ and $p \sqsubset m$, and since P_v calls P_x, $v \sqsubset m$ and $t \sqsubset m$.

Throughout Section 3, we assume ($\sqsubseteq$) refers to the partial order defined by (1) and (2).

3.2 Interference Graphs

Let T be a set on which there is some partial order. Define a comparability digraph $D(T)$ as having an edge from u to v when u is less than v in the partial order [Cam85]. If S is the set of candidates of a call graph G and the partial order is ($\sqsubseteq$), then $D(S)$ is the interference graph for a save-free interprocedural register allocation of G. If there is an edge between c_u and c_v in $D(S)$, then either c_u and c_v are candidates in the same procedure, or c_u and c_v are candidates in procedures along a path in the call graph. Candidates c_u and c_v cannot be assigned the same register.

Since a partial order defines the interference relation between candidates in a save-free interprocedural register allocation, the interference graph is transitive. The interference graph for *intra*procedural register allocation, however, can

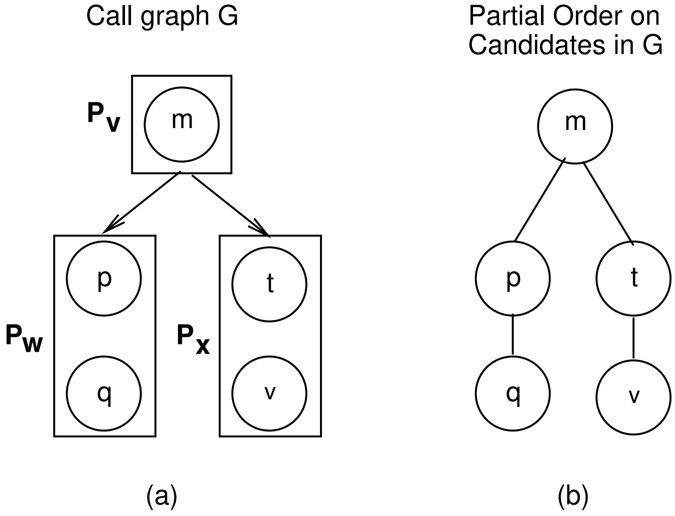

Figure 1: An example call graph, allowing for multiple candidates in a procedure, and a partial order on the candidates of the call graph.

be non-transitive[Cha82]. In an intraprocedural register allocation, two live ranges that interfere are assigned different registers. Assume live ranges l_a and l_b interfere and live ranges l_b and l_c interfere. Live range l_a does not necessarily interfere with l_c. In a save-free interprocedural register allocation, if procedure P_v calls P_w and P_w calls P_x, then execution will normally return to P_v. To avoid overwriting the registers live across a call, candidates in P_v, P_w, and P_x are all assigned different registers.

Figure 2 displays a call graph G, the partial order ($\sqsubseteq$) on the set of candidates, S, of G, and the comparability digraph $D(S)$. The number below a candidate is the benefit of allocating a register to that candidate. Since $q \sqsubset p$ and $p \sqsubset m$, there is an edge in $D(S)$ between q and p, p and m, and q and m. These three candidates can never be assigned the same register. Candidate t can be assigned the same register as p or q, as there is no edge joining either t and p or t and q.

3.3 Antichains

We call a set of nodes in a digraph independent if none of the nodes in the set are joined by an edge. Let S be a set and assume some partial order on S. An *antichain* in S is an independent set of nodes in $D(S)$. For example, in Figure 2(c), $\{p,t\}$, $\{p,v\}$, and $\{q,v\}$ are antichains, as the candidates in each set are not joined by an edge in the comparability digraph. A k-*antichain* is the union of at most k antichains [Cam85]. Both $\{p,t,q,v\}$ and $\{p,t,v\}$ are 2-antichains.

Let S be the set of candidates of a call graph G and assume partial order ($\sqsubseteq$) on S. $D(S)$ represents an interference graph for a save-free interprocedural register allocation of G and, thus, the candidates of an antichain in $D(S)$ can be assigned the same register. A k-antichain in $D(S)$ is a set of candidates that can be allocated using at most k registers in G.

Assume each register candidate $c_j \in S$ has a positive integer weighting, w_j. Let ($\sqsubseteq$) be a partial order on S. If w_j is the benefit of allocating a register to candidate c_j, then a maximum weight k-antichain in S corresponds to a k-register save-free interprocedural register allocation whose elements sum to the maximum benefit; that is, a save-free minimum cost interprocedural register allocation using at most k registers.

In Figure 2(c), assume $k = 2$ antichains. Among the possible allocations of candidates to antichains, the choice with the greatest benefit allocates candidate m to an antichain (A_2), and candidates q and t to an antichain (A_1). Each antichain maps to an arbitrary, but different register. In Figure 2(a), m is assigned register r_2, and q and t are assigned register r_1. Since candidates p and v are not allocated registers, an intraprocedural register allocator will spill registers as necessary in procedures P_w and P_x to generate a valid register allocation.

3.4 Finding a maximum weight k-antichain sequence

A k-antichain can be partitioned into a k-antichain sequence. A k-antichain sequence $A = (A_1, \ldots, A_k)$, where $A_i \subseteq S$, and if $c_i \in A_p$, $c_j \in A_q$, and $c_i \sqsubset c_j$, then $p < q$ [Cam85]. Each A_i, $1 \leq i \leq k$, corresponds to an antichain—if $c_i \sqsubset c_j$, then c_i and c_j cannot be members of the same antichain. Given the 2-antichain $\{q,t,m\}$, and the partial order $t \sqsubset m$ and $q \sqsubset m$, then antichain $A_1 = \{q,t\}$ and $A_2 = \{m\}$, as shown in Figure 2(c).

Given partial order ($\sqsubseteq$) on a set of candidates in a call graph G, we can view each antichain A_i, as an abstract register R_i. An abstract register is an equivalence class of candidates that can be assigned the same register. Each abstract register maps to a different hardware register.

Let $c_v \in C(P_v)$, $c_w \in C(P_w)$, and assume $c_w \sqsubset c_v$—there is a call path from P_v to P_w or $P_v = P_w$. Assume we assign registers to candidates in a topological ordering over the partial order—if $c_w \sqsubset c_v$, then we visit c_v before c_w. If we assign register R_q to c_v, then c_w can only be assigned register R_p, such that $0 < p < q$. This register ordering models the sequence in which antichains are assigned to candidates.

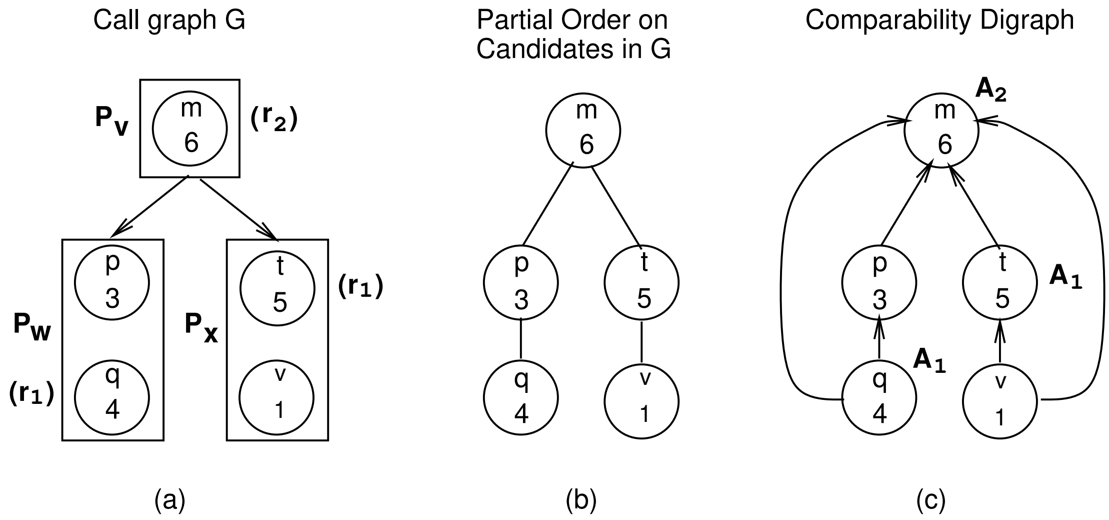

Figure 2: A call graph G, a partial order on the candidates in G, and the comparability digraph of the candidates in G.

To find a maximum weight k-antichain sequence in a partially ordered set, we solve the following dual minimum cost flow problem[Cam85].

Dual Variables
x_j, y_j for $c_j \in S$
Constraints
A.1 for $c_j \in S$, $0 \leq x_j$, $y_j \leq k$.
A.2 if $c_i, c_j \in S$ and $c_i \sqsubset c_j$, then $x_i + y_j \leq k$.
A.3 for $c_j \in S$, $x_j + y_j \leq k + 1$.
Objective Function
A.4 Maximize $\sum_{c_j \in S} w_j * (x_j + y_j)$.

For each candidate in $c_j \in S$, there is a pair of integer dual variables, x_j and y_j, and an integer weight $w_j > 0$ in the dual minimum cost flow problem. A solution to this dual minimum cost flow problem maximizes the objective function A.4, given the constraints A.1–A.3 on the dual variables. Whereas a minimum cost flow problem minimizes an objective function, a dual minimum cost flow problem maximizes an objective function. If a candidate c_j is allocated to an antichain, the variable x_j will specify the antichain that c_j is assigned. The variable y_j constrains the value of x_i for $c_i \sqsubset c_j$ to prevent both candidates c_i and c_j from being assigned to the same antichain.

Figure 3(a) shows a call graph G. Let S be the set of candidates in G. A representation of the partial order on S appears in (b). Based on this partial order, a representation of the dual minimum cost flow problem appears in (c). Each node represents a dual variable. Solid edges represent constraint A.2. Dashed edges represent constraint A.3.

Intuitively, a correspondence exists between a maximum weight k-antichain sequence and assignments to the dual variables of the dual minimum cost flow problem. In solutions to the dual minimum cost flow problem, one can prove that for $c_j \in S$, $x_j + y_j = k + 1$ or $x_j + y_j = k$ [Cam85]. If $x_j + y_j = k + 1$, then we map c_j to the antichain whose number in the sequence equals the value of x_j. Otherwise, if $x_j + y_j = k$, c_j is not mapped to an antichain.

Assume that (a) $x_j + y_j = k + 1$, and let $c_i \sqsubset c_j$. By constraint A.2, (b) $x_i + y_j \leq k$. Equations (a) and (b) imply that $x_i < x_j$. Assume $x_j = h$. We map c_j to antichain A_h. All candidates $c_i \sqsubset c_j$ can only map to antichains A_m, $0 < m < h$.

Let $x_j + y_j = k$ for $c_j \in S$. If $c_i \sqsubset c_j$, then by constraint A.2, $x_i + y_j \leq k$. Thus, $x_i \leq x_j$. Assume $x_j = h$. Thus, we do not map c_j to antichain A_h, as c_i may be mapped to A_h. Candidates c_i and c_j can never share the same antichain.

There exists a 1-1 and onto mapping (a bijection) from maximum weight k-antichain sequences to solutions of the dual minimum cost flow problem above. A solution to the dual minimum cost flow problem is represented by a sequence of tuples $z = ((x_1, y_1), \ldots, (x_{|S|}, y_{|S|}))$.

Let $Q^*(k, S)$ be the maximum weight k-antichain sequences in S, and let $P^*(k, S)$ be the solutions to the dual minimum cost flow problem. For $A \in Q^*(k, S)$ there is a bijection $z(A)$ onto $z \in P^*(k, S)$, and for $z \in P^*(k, S)$, there is an inverse function $A(z)$[Cam85]. $A(z)$ maps $z \in P^*(k, S)$ onto a maximum weight k-antichain sequence $(A_1, \ldots, A_k)$. Mapping $A(z)$ is defined as $(A_1(z), \ldots, A_k(z))$. For $1 \leq p \leq k$, $A_p(z)$, which maps candidates to antichain A_p, is defined as

$$A_p(z) = \{c_i \mid x_i = p;\ x_i + y_i = k + 1\}.$$

If the dual variables x_i and y_i sum to $k + 1$, then candidate c_i is mapped to the antichain whose number in the sequence equals the value of x_i.

Assume $z \in P^*(k, S)$ and $A(z) = A \in Q^*(k, S)$. The objective function A.4 maximizes $\sum_{c_j \in S} w_j * (x_j + y_j)$. If $x_j + y_j = k + 1$, then c_j is mapped to an antichain; otherwise, $x_j + y_j = k$ and c_j is not mapped to an antichain. Thus, $x_j + y_j - k = 1$ if c_j is mapped to an antichain; otherwise, $x_j + y_j - k = 0$. The value of the objective function for solution z, therefore, differs from the weight of maximum weight k-antichain sequence A by a constant.

3.5 Example

Figure 4(a) shows the partial order on the candidates of call graph G of Figure 3. Candidates along a path must be assigned to distinct abstract registers. Let R_p and R_q be

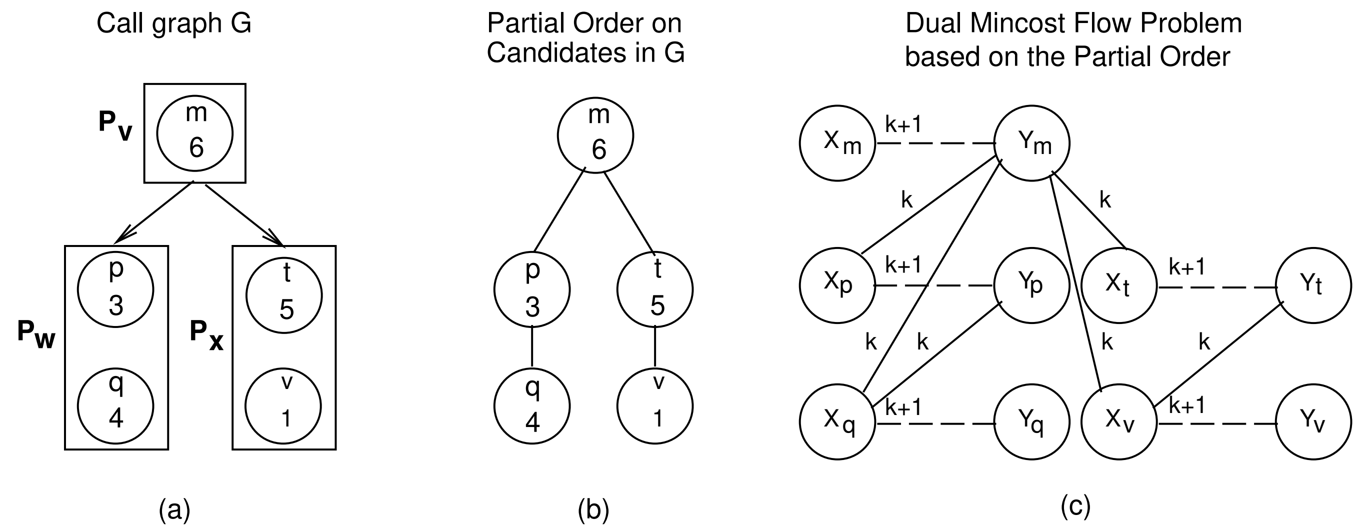

Figure 3: Example call graph G, graph of a partial order on the candidates in G, and the dual minimum cost flow problem with respect to the partial order. Each node in the dual minimum cost flow problem represents an integer-valued dual variable, and each edge represents a constraint between two dual variables. The constant above an edge is the integer upper bound in the corresponding constraint.

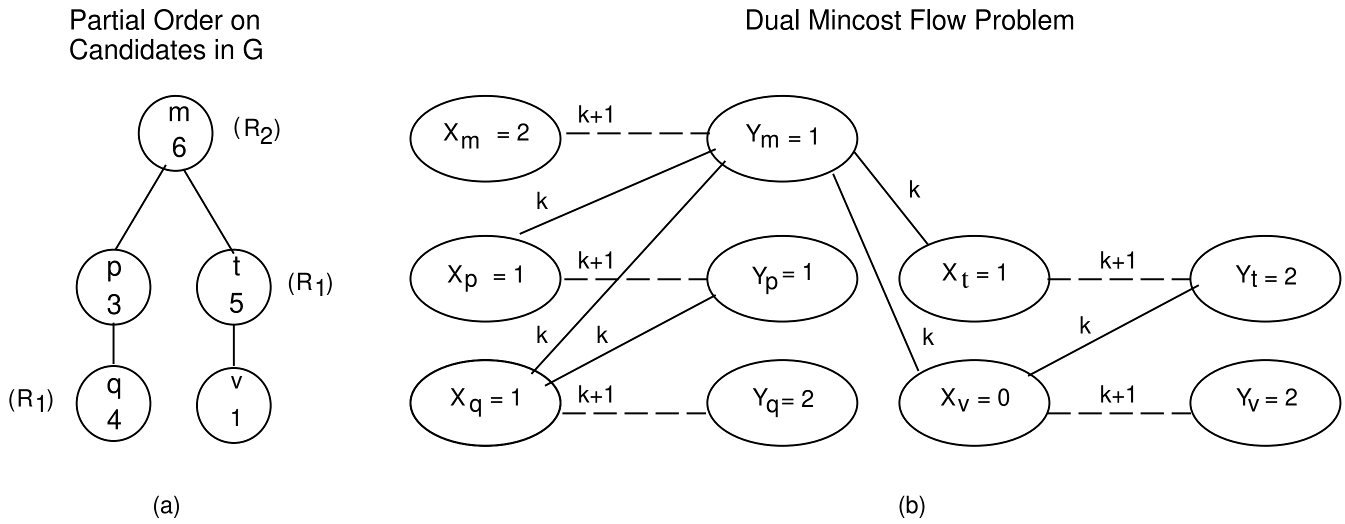

Figure 4: Partial order on candidates in G, and graph of dual minimum cost flow problem for G.

abstract registers. If $c_i \sqsubset c_j$, $c_i \in R_p$, and $c_j \in R_q$, then $p < q$.

A register allocation and assignment using two registers appears in Figure 4(a). Candidate m is assigned register R_2, and q and t are assigned register R_1. This register allocation has the maximum benefit.

Figure 4(b) shows the the graph of the dual minimum cost flow problem based on the partial order (a) for $k = 2$ registers. The solution in (b) can be mapped to the solution in (a). Since $x_m + y_m = k + 1$, and $x_m = 2$, candidate m maps to register R_2. As $x_q + y_q = k+1$ and $x_q = 1$, q maps to R_1. Similarly, candidate t maps to R_1, since $x_t + y_t = k + 1$ and $x_t = 1$. Since $v \sqsubset t$, $x_v = 0$. As $x_v + y_v = k$, candidate v is not mapped to a register.

4 Interprocedural Register Allocation with Spilling

In this section, we consider an interprocedural register allocation that allows for register spilling across calls. The call graph can now be cyclic (save-free allocations are generally not possible for cyclic call graphs). As in the save-free approach, we assume a benefit associated with allocating registers to procedures, but now we also assume a cost associated with spilling registers across calls. The cost of spilling a register is two (for a load and a store) times the frequency of the calls represented by the edge. To find an allocation with maximum benefit, we again map solutions from a dual minimum cost flow problem.

Let call graph $G = (P, E)$, where P is a set of procedures and E is a set of call edges. For $P_v \in P$, let $C(P_v)$ represent the set of local register candidates in P_v, and let $C(P)$ represent the set of local candidates in all procedures in the call graph. For a procedure P_v, let $IN(P_v)$ be the set of call edges incident on P_v, and let $OUT(P_v)$ be the set of outgoing call edges from P_v.

In the save-free approach, if there is an ordering between two candidates, then they cannot be assigned the same register. However, since registers are now spilled as necessary around calls, if $c_v \in C(P_v)$, $c_w \in C(P_w)$ and P_v calls P_w, then c_w may be assigned the same register as c_v. We, therefore, now assume a partial order that only relates candidates in the same procedure, as these candidates can never be assigned the same register. The ordering among the candidates in a procedure is a chain, as in (1) of Section 3.1. We refer to this partial order as ($\sqsubseteq$) throughout Section 4.

Since partial order ($\sqsubseteq$) only relates candidates in the same procedure, there is an ordering between q and m in Figure 5. Let $q \sqsubset m$. We represent this ordering by an undirected edge between q and m. For $t \in C(P_2)$, $t \sqsubseteq t$.

Let an abstract register R_h, $1 \leq h \leq k$, be a set composed of candidates assigned that register. Each abstract register is mapped to a hardware register after interprocedural register allocation. Let R be the sequence $(R_1, \ldots, R_k)$.

To model spills along the edges of a call graph, two integer variables are introduced for each edge. For $e_j \in E$, the variable $free\_in_j$ represents the number of unallocated registers on entrance to edge e_j, and the variable $free\_out_j$ represents the number of unallocated registers on exit from edge e_j. The number of registers spilled along edge e_j is, therefore, $free\_out_j - free\_in_j$. Let $free\_in$ be the sequence $(free\_in_1, \ldots, free\_in_{|E|})$ and $free\_out$ be the sequence $(free\_out_1, \ldots, free\_out_{|E|})$.

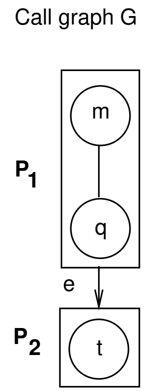

Figure 5: Example call graph G. A partial order exists only among candidates in each procedure.

Assume k registers are available for an interprocedural register allocation. Allowing for register spilling along the call edges, an interprocedural register allocation I for a call graph G is represented by $I = (R,\ free\_in,\ free\_out)$, and has the following constraints and maximization function:

Constraints

I.1 for $e_j \in E$, $free\_in_j \leq free\_out_j$.

I.2 for $e_j \in E$, $0 \leq free\_in_j,\ free\_out_j \leq k$.

I.3 if $c_i \in R_p$, $c_i \in C(P_v)$, and $e_j \in IN(P_v)$, then $p \leq free\_out_j$.

I.4 if $c_i \in R_p$, $c_i \in C(P_v)$, and $e_j \in OUT(P_v)$, then $p > free\_in_j$.

I.5 if $e_j \in IN(P_v)$ and $e_i \in OUT(P_v)$, then $free\_out_j \geq free\_in_i$.

I.6 let $c_i, c_j \in C(P_v)$. if $c_i \in R_p$, $c_j \in R_q$, and $c_i \sqsubset c_j$, then $p < q$.

Maximization Function

I.7 maximize $\sum_{c_j \in \bigcup_{i=1}^{k} R_i} w_j - \sum_{e_j \in E} s_j * (free\_out_j - free\_in_j)$

Constraints I.1 - I.6 define how registers are spilled along the call edges and consumed within procedures. Constraint I.1 states that the number of free registers on exit from a call edge is greater than or equal to the number of free registers on entry to that edge (the difference is the number of registers spilled along the edge). Constraint I.2 bounds the number of free registers on entrance to and exit from an edge by the number of registers available for allocation. Constraint I.3 asserts that if candidate c_i is assigned to register R_p, then there must be at least p registers free on entry to the procedure from each incoming edge (c_i is assigned one of the free registers). Similarly, I.4 asserts that if c_i is assigned to register R_p, then there must be fewer than p registers free upon exit from the procedure along each outgoing call edge. By I.5, there cannot be more registers upon exiting a procedure than there are upon entering it (all saving is done on the edges).

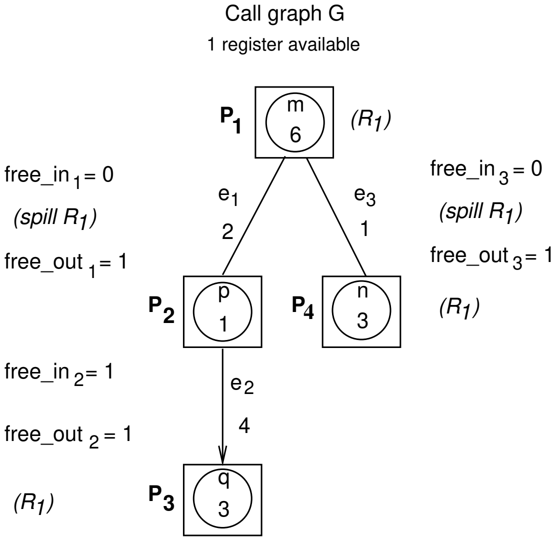

Figure 6: Interprocedural register allocation of call graph G.

Two candidates $c_i, c_j \in C(P_v)$ cannot be assigned the same register. By I.6, registers are assigned in a decreasing sequence within a procedure. If candidates c_i and c_j are both allocated registers in procedure P_v and $c_i \sqsubset c_j$, then the register assigned to c_i occurs before the register assigned to c_j in the sequence. As in the case for save-free interprocedural register allocation, if there is no register spilling, registers are assigned in a decreasing sequence across calls. Assume $c_v \in C(P_v)$, $c_w \in C(P_w)$, and P_v calls P_w (we model the call as edge e_j in the call graph). If c_v is assigned register R_p, then by constraint I.4, there are fewer than p registers free on entry to e_j ($free\_in_j < p$). Assume no registers are spilled around the call. Thus, $free\_out_j = free\_in_j$ in I.1. If c_w is assigned register R_q, then by constraint I.3, $q \leq free\_out_j$. Therefore, since $free\_out_j = free\_in_j$ and $free\_in_j < p$, then $q < p$.

Each candidate $c_j \in C(P)$ has a positive integer weight, w_j. Each call edge $e_j \in E$ has a positive integer cost, s_j, for spilling a register, and spills $free\_out_j - free\_in_j$ registers. Assume e_j is the call edge from P_v to P_w. Abstract register R_i, $i \leq free\_in_j$, is available on exit from P_v. By constraint I.3, register R_i, $i \leq free\_out_j$ is available to candidates in P_w. Our algorithm spills register R_i along edge e_j if $free\_in_j < i \leq free\_out_j$.

We want to find a k-register interprocedural register allocation that maximizes function I.7. Since spilling decreases the value of I.7, candidates are assigned a spilled register only if the sum of their weights is at least as large as the cost of spilling that register.

Figure 6 presents an interprocedural register allocation assuming one available register. The number below each candidate is the benefit of allocating that candidate a register. The number below a call edge is the cost of spilling a register on that edge. Candidate m is assigned register R_1. Since the benefit of allocating a register to p is less than the spill cost along edge e_1, p is not allocated a register. However, the benefit of allocating a register to q exceeds the spill cost along edge e_1 (but not the spill cost along edge e_2). Thus, register R_1 is spilled along edge e_1, and R_1 is assigned to q. Since the cost of spilling a register along edge e_3 is less than the benefit of allocating a register to n, R_1 is spilled along e_3 and assigned to n.

Dual Variables

$(x_i,\ y_i$ for $c_i \in C(P)), (r_j,\ t_j$ for $e_j \in E)$

Constraints

D.1 for $c_i \in C(P)$, $0 \leq x_i,\ y_i \leq k$.

D.2 if $c_i, c_j \in C(P_v)$ and $c_i \sqsubset c_j$, then $x_i + y_j \leq k$.

D.3 for $c_j \in C(P)$, $x_j + y_j \leq k + 1$.

D.4 for $e_j \in E$, $0 \leq r_j,\ t_j \leq k$.

D.5 for $c_i \in C(P_v)$ and $e_j \in IN(P_v)$, $x_i + t_j \leq k$.

D.6 for $e_i \in OUT(P_v)$ and $c_j \in C(P_v)$, $r_i + y_j \leq k$.

D.7 for $e_i \in OUT(P_v)$ and $e_j \in IN(P_v)$, $r_i + t_j \leq k$.

D.8 for $e_j \in E$, $r_j + t_j \leq k$.

Objective Function

D.9 Maximize $\sum_{c_j \in C(P)} w_j * (x_j + y_j) + \sum_{e_j \in E} s_j * (r_j + t_j)$.

Figure 7: Dual minimum cost flow problem whose solutions are mapped to an interprocedural register allocation with spilling.

4.1 Finding a Minimum Cost Allocation

To find a minimum cost interprocedural register allocation for a call graph, we solve the dual minimum cost flow problem of Figure 7.

In this dual minimum cost flow problem, there is a pair of integer dual variables $(x_i,\ y_i)$ for each candidate $c_i \in C(P)$ and a pair of integer dual variables $(r_j,\ t_j)$ for each edge $e_j \in E$. As before, for $c_i \in C(P)$, integer $w_i > 0$ represents the benefit of allocating a register to a candidate. For $e_j \in E$, integer $s_j > 0$ represents the cost of spilling a register on edge e_j in the call graph. As in the save-free approach if $x_i + y_i = k+1$, then candidate c_i will be assigned the register whose value is x_i. For $e_j \in E$, r_j represents the number of free registers on entry to edge e_j, and t_j represents the number of registers allocated on exit from e_j. For $e_j \in IN(P_c)$, t_j constrains the registers that can be allocated to candidates in procedure P_c (constraint D.5), the number of free registers on outgoing edges from P_c (constraint D.7), and the number of free registers on entry to e_j (constraint D.8). Also, candidates in P_c constrain the number of free registers on outgoing edges from P_c (constraint D.6).

We define xy to be the sequence of tuples

$$((x_1, y_1), \ldots, (x_{|C(P)|}, y_{|C(P)|})),$$

and rt to be the sequence of tuples

$$((r_1, t_1), \ldots, (r_{|E|}, t_{|E|})).$$

A solution to the dual minimum cost flow problem is represented by tuple $z = (xy, rt)$. For a call graph G on which we define partial order ($\sqsubseteq$), and given k registers, let $P^*(k,G)$ be solutions to the dual minimum cost flow problem of Figure 7. Let $Q^*(k,G)$ be solutions to interprocedural register allocation with spilling. In [Kur95], we prove that there exists a bijection $z(I)$, from $I \in Q^*(k,G)$ onto $z \in P^*(k,G)$, and an inverse function $I(z)$ for $z \in P^*(k,G)$. $I(z)$ is defined below.

- $I(z) = (R(z),\ free\_in(z),\ free\_out(z))$.
- for $1 \leq h \leq k$,
  $R_h(z) = \{c_j \mid x_j + y_j = k+1, x_j = h, c_j \in C(P)\}$;
  $R(z) = (R_1(z), \ldots, R_k(z))$.
- for $e_j \in E$, $free\_in_j(z) = r_j$;
  $free\_in(z) = (free\_in_1(z), \ldots, free\_in_{|E|}(z))$.
- for $e_j \in E$, $free\_out_j(z) = k - t_j$;
  $free\_out(z) = (free\_out_1(z), \ldots, free\_out_{|E|}(z))$.

For $z \in P^*(k,G)$, function $I(z)$ maps z to an interprocedural register allocation defined as $(R,\ free\_in,\ free\_out)$. Functions $R(z)$, $free\_in(z)$, and $free\_out(z)$ map to sequences R, $free\_in$, and $free\_out$, respectively. $R_h(z)$ maps candidate c_j to register R_h if $x_j + y_j = k+1$ and $x_j = h$. For $e_j \in E$, $free\_in_j(z)$ maps the value of variable r_j to $free\_in_j$. For $e_j \in E$, $free\_out_j(z)$ maps $k - t_j$ to $free\_out_j$. The variable t_j, therefore, represents the number of unavailable registers on exit from edge e_j. The number of register spills along edge e_j is $free\_out_j - free\_in_j = k - t_j - r_j$. The number of registers spilled along an edge includes those not free on entry to the edge $(k - r_j)$ but made available on exit from the the edge $(k - r_j - t_j)$. Thus, $r_j + t_j$ is the number of registers not spilled along edge e_j.

Constraints D.1 – D.3 of Figure 7 are similar to constraints A.1 – A.3 of the dual minimum cost flow problem for a save-free allocation. As in the dual minimum cost flow problem of Section 3, if $z \in P^*(k,S)$, then for $c_j \in C(P)$, $x_j + y_j = k$ or $x_j + y_j = k+1$. Assume (a) $x_j + y_j = k+1$ and $c_i \sqsubset c_j$. If $c_i \sqsubset c_j$, then by constraint D.2, (b) $x_i + y_j \leq k$. Equations (a) and (b) imply $x_i < x_j$ and, thus, c_i and c_j can never be assigned the same register.

Constraint D.4 bounds the value of r_j and t_j for $e_j \in E$ by the number of available registers. By constraint D.5, for $e_j \in IN(P_v)$, $k - t_j$ bounds the number of registers available to candidates in $C(P_v)$. For $c_i \in C(P_v)$, assume $x_i + y_i = k+1$ and $x_i = p$. Candidate c_i is mapped to register R_p. By constraint D.5, $x_i \leq k - t_j$. Mapping $I(z)$ assigns $free\_out_j$ the value $k - t_j$. Thus, $p \leq free\_out_j$, which is constraint I.3 in our definition of an interprocedural register allocation.

By constraint D.6, the value of $k - y_i$ for $c_i \in C(P_v)$ bounds the value of r_j for $e_j \in OUT(P_v)$. Mapping $I(z)$ assigns $free\_in_j$ the value of r_j. Thus, $k - y_i$ bounds the number of free registers on entrance to e_j. Assume $x_i + y_i = k+1$ and $x_i = p$. Candidate c_i is mapped to register R_p. By D.6, $r_j + y_i \leq k$. Thus, $r_j < x_i$. Since $I(z)$ maps the value of r_j to $free\_in_j$ and $x_i = p$; therefore, $free\_in_j < p$, which is constraint I.4.

By constraint D.7, $k - t_j$ bounds r_i for $e_j \in IN(P_v)$ and $e_i \in OUT(P_v)$. By D.7, $r_i \leq k - t_j$. By mapping $I(z)$, $free\_in_i \leq free\_out_j$, which is constraint I.5.

By constraint D.8, for $e_i \in E$, $r_i + t_i \leq k$. As mentioned above, $r_i + t_i$ is the number of registers not spilled along e_i. As $r_i + t_i \leq k$, then $r_i \leq k - t_i$. Applying mapping $I(z)$, $free\_in_i \leq free\_out_i$, which is constraint I.1.

The objective function D.9 is

$$\sum_{c_j \in C(P)} w_j * (x_j + y_j) + \sum_{e_j \in E} s_j * (r_j + t_j),$$

and the maximization function I.7 is

$$\sum_{c_j \in \bigcup_{i=1}^{k} R_i} w_j - \sum_{e_j \in E} s_j * (free\_out_j - free\_in_j).$$

The value of the objective function for $z \in P^*(k,G)$ (D.9) and the value of the maximization function for $I(z) = I \in Q^*(k,G)$ (I.7) differ by a constant. As in Section 3, by subtracting the constant $\sum_{c_j \in C(P)} k * w_j$ from $\sum_{c_j \in C(P)} w_j * (x_j + y_j)$ in D.9 yields (a) $\sum_{c_j \in C(P)} w_j * (x_j + y_j - k)$. Since $x_j + y_j - k = 1$ if c_j is mapped to a register and, otherwise, $x_j + y_j - k = 0$, equation (a) is equal in value to $\sum_{c_j \in \bigcup_{i=1}^{k} R_i} w_j$ in I.7.

Moreover, for $e_j \in E$, (b) $r_j + t_j$ in D.9 is the number of registers not spilled along e_j, and (c) $-(free\_out_j - free\_in_j)$ in I.7 is the negative of the number of registers spilled along e_j. Thus, $r_j + t_j - k = -(free\_out_j - free\_in_j)$. As (b) and (c) differ by the constant k, $\sum_{e_j \in E} s_j * (r_j + t_j)$ in D.9 differs from $-\sum_{e_j \in E} s_j * (free\_out_j - free\_in_j)$ in I.7 by the constant $\sum_{e_j \in E} s_j * k$.

4.2 Example

Figure 8(a) shows call graph G of Figure 6 with the same interprocedural register allocation. Variable $alloc_i$ represents the register that c_i *may* be assigned. Only one register is available. Register R_1 is assigned to m, q, and n and spilled along edges e_1 and e_3.

Figure 8(b) displays the graph of the dual minimum cost flow problem for G. There is a pair of nodes for each candidate and call edge in G. For clarity, variable x_i is renamed $alloc_i$ for $c_i \in C(P)$. Since mapping $I(z)$ for $e_i \in E$ assigns $free\_in_i$ the value of r_i, we rename r_i in (b) as $free\_in_i$.

The dashed edges in Figure 8(b) represent constraints between pairs of nodes and solid edges represent constraints between nodes from separate pairs. The k or $k+1$ along an edge represents the bound in the corresponding constraint.

Assume $k = 1$ in Figure 8(b). Since $alloc_m + y_m = k+1$, and by constraint D.5, $free\_in_1 + y_m \leq k$, then $free\_in_1 < alloc_m$. Since the number of available registers decreases from 1 to 0, we assign candidate m to register R_1 ($alloc_m = 1$). As there are 0 free registers on entrance to e_1 ($free\_in_1 = 0$) and 0 unavailable registers on exit from e_1 ($t_1 = 0$), then the number of register spills along e_1 is $k - t_1 - free\_in_1 = 1$. Therefore, candidate p may be assigned register R_1, as $alloc_p = 1$. Since $alloc_p + y_p = k$, p is not allocated a register.

Since p is not allocated a register, there is a register free on entry to e_2 ($free\_in_2 = 1$). By constraint D.8, $free\_in_2 + t_2 \leq k$. Thus, $t_2 = 0$—there are 0 unavailable registers out of e_2. Candidate q is allocated a register, as $alloc_q + y_q = k+1$.

A register is spilled along e_3, since $k - free\_in_3 - t_3 = 1$. This register is assigned to n. The register allocation and assignment of (b) correctly corresponds to the allocation and assignment described in (a).

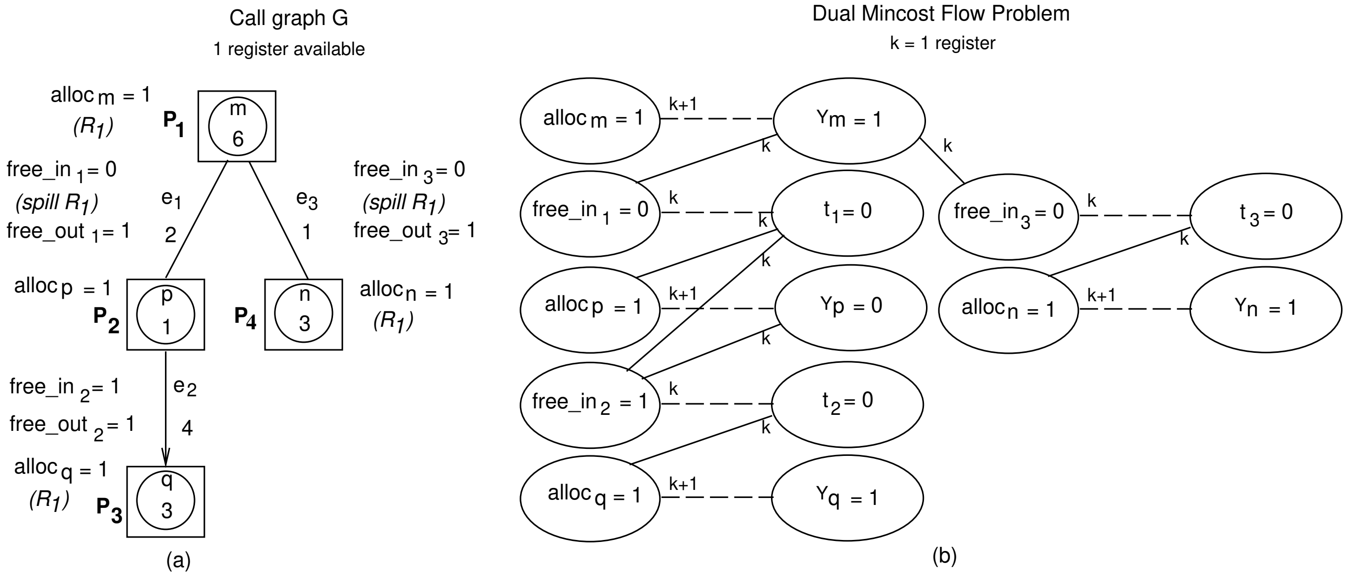

Figure 8: Example call graph G and graph representation of the dual minimum cost flow problem for G.

5 Complexity

For p candidates and edges in a call graph, the number of dual variables in the dual minimum cost flow problem of Section 4 is $O(p)$. However, the number of constraints between dual variables is $O(p^2)$, as a dual variable for a candidate or edge can have constraints with $O(p)$ other dual variables. Our dual minimum cost flow problem can be transformed into an unconstrained minimum cost flow problem, in which there are $O(p)$ nodes and $O(p^2)$ arcs.

Letting n be the number of nodes and m be the number of arcs, an unconstrained minimum cost flow problem can be solved in $O(n \log n(m+n \log n)$ [Orl93], which is independent of k, w_j, and s_j in our dual minimum cost flow problem. The complexity of solving our minimum cost flow problem is, therefore, $O((p \log p\ (p^2 + p \log p))$, which is $O(p^3 \log p)$.

6 Liveness

Before performing interprocedural register allocation, we can modify the call graph to avoid spilling registers assigned to candidates not live across any call. Our interprocedural register allocation model assumes that a candidate live across a call is live across all calls.

In a procedure, let L be the set of candidates that are live across a call, and let NL be the set of candidates not live across any call. In each procedure, we move the candidates in NL below the candidates L in the partial order. Constraints are not added between the candidates in NL and the outgoing edges of the procedure. All candidates in the procedure compete for registers as before, but as there are no constraints between the outgoing edges from the procedure and the candidates in NL, the registers assigned to these candidates are not spilled.

In Figure 9, we assume candidates m and n are not live across the call to P_2. In (b), m and n are moved below q in the partial order. By moving m and n below q, q is now assigned R_3, and m and n are assigned R_2 and R_1. Since

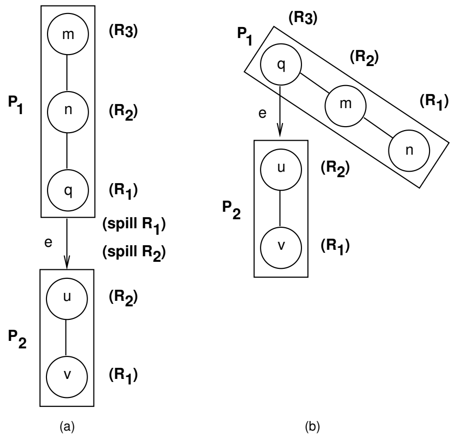

Figure 9: By distinguishing between candidates live and not live across calls, fewer registers are spilled.

we also remove the constraints between candidates m and n and edge e, we can assign u and v in P_2 the same registers as m and n, without spilling registers across the call.

7 Library Routines

We assume that library routines have been pre-compiled using a caller-save/callee-save convention for spilling registers across calls[CHKW86]. Any caller-save register live across a call to a library routine must be spilled across the call.

To allow for pre-compiled library routines, we create a pseudo library routine that allocates the abstract registers that we will map to the pre-defined caller-save registers. All procedures that call library routines have a call edge to this pseudo library routine. As all caller-save registers are allocated in this pseudo routine, a caller-save register live across a call to this routine will be spilled.

Assume there are n caller-save registers and k total registers. Since abstract registers are assigned in a decreasing sequence, we let abstract registers $R_1, \ldots, R_n$ map to the caller-save registers. Only if more than $k-n$ registers are live across the call to the pseudo library routine will a caller-save register be spilled. To ensure that the n candidates in the library routine are assigned abstract registers $R_1, \ldots, R_n$, we modify the dual minimum cost flow problem in Figure 7 such that $x_i = i; x_i + y_i = k+1$ for candidates c_i, $1 \leq i \leq n$, allocated in the pseudo library routine.

8 Indirect Calls

Indirect calls use the same caller-save/callee-save convention followed by library routines. When building a call graph, we assume that each procedure that can perform an indirect call can invoke any aliased procedure. The number of call edges representing indirect calls would, therefore, be the product of the number of routines that can make an indirect call and the number of aliased routines.

Since we assume a fixed calling convention it is not necessary to include these call edges. Instead, we add a call edge from a routine making an indirect call to the pseudo library routine. Caller-save registers allocated by the procedure making an indirect call must be spilled around the call. We remove the call edges incident on the aliased routines (for simplicity all indirect and non-indirect calls to aliased routines will use the fixed calling convention), and add one call edge e_j from a newly generated pseudo procedure to the aliased routine. We assign the number of caller-save registers, n, to dual variable r_j, the number of registers free on entry to edge e_j as defined by the dual minimum cost flow problem of Figure 7. Registers can be spilled along e_j (spilled on entry to the aliased routine), as the number of register spills, $k - r_j - t_j$, along e_j can be positive.

9 Implementation

We generate code for a DECstation 5000/125, with MIPS R3000/R3010 processors. We assume that three general purpose integer registers, two general purpose floating-point registers, and the pre-defined parameter registers are work registers that are not allocated interprocedurally and hence are available to each routine.

We use profile information to compute the number of calls between each procedure and the number of instructions executed in each procedure. Profile information is gathered using qpt[BL92]. When profiling, benchmarks are run on input yielding short execution times, except for benchmark *nasa7*, in which we have only one input file. Since the profiled code is compiled using only an intraprocedural register allocator, some variables live across calls may not be allocated a register because of an insufficient number of callee-save registers. To accurately determine the number of references to registers that can be live across a call, we modified gcc[Sta93] to return the number of register references assuming the non-work registers are callee-save. We let the general-purpose registers that are non-work registers represent candidates in our interprocedural register allocation algorithm, and their number of register references scaled using profile information represents the candidates' weight.

After generating an interprocedural register allocation, the registers available to each procedure and the spills across each call are written to a file. Gcc reads this file to generate a register allocation. We assume that library routines have been pre-compiled using a caller-save/callee-save convention for spilling registers around calls.

Execution-time Improvement		
benchmark	*Minimum Cost*	*Steenkiste and Hennessy*
compress	1.4%	-0.2%
doduc	4.6%	4.2%
eqntott	0%	0%
espresso	8.7%	7.3%
fpppp	3.9%	3.0%
gcc	8.3%	1.1%
nasa7	0.2%	-0.1%
sc	10.7%	7.4%
spice	2.8%	1.7%
xlisp	11.2%	-3.2%

Figure 10: Execution-time improvement from adding our minimum cost interprocedural register allocator with spills and Steenkiste and Hennessy's bottom-up interprocedural register allocator to gcc.

In some benchmarks, a procedure that is not called when profiling with one input is called when using another. If a procedure is not called, we have no information on the frequency in which its candidates are referenced. We optimistically allocate registers to these procedures' candidates as follows. We increase all zero edge frequencies to one. Assume the total spill cost along incoming and outgoing edges of a procedure is j. Register candidates in a procedure called zero times are assigned a benefit of $1 + j$. Since the cost of spilling a register on entry to and exit from a procedure is less than the benefit of allocating a register to a candidate, these candidates are always allocated a register.

Figure 10 compares the execution-time improvement of adding our minimum cost interprocedural register allocator with spills with Steenkiste and Hennessy's bottom-up interprocedural register allocator[SH89] to gcc. The benchmarks are compiled at optimization level O2 with loop-unrolling enabled. Results from a sample of SPEC92 benchmarks are presented. Both interprocedural register allocators find a significant improvement on benchmark *doduc*, as this benchmark has procedures with many registers live across calls. An interprocedural register allocator can generate an allocation that spills fewer registers across calls than an intraprocedural register allocator. Benchmark *eqntott* shows no improvement for either allocator, as most of its execution is in a leaf procedure.

Benchmark *xlisp* shows a large improvement for our allocator as it has small, frequently called routines. However, running Steenkiste and Hennessy's bottom-up register allocator results in a worse allocation than an intraprocedural register allocation. Benchmark *xlisp* has many routines at the bottom of the call graph called less frequently than routines higher in the call graph. With a bottom-up allocation, registers are spilled across the more frequently executed calls. Steenkiste and Hennessy[SH89] note that a better interprocedural register allocation can be generated by adding register spills in infrequently executed procedures in the bottom of the call graph and then performing a bottom-up allocation assuming these routines are allocated zero registers.

benchmark	*procedures*	*candidates*		*% of compilation time*	
		floating-point	*integer*	*floating-point*	*integer*
compress	16	0	31	< 0.1%	0.3%
doduc	42	170	274	< 0.1%	0.2%
eqntott	62	0	178	0.1%	0.8%
espresso	361	1	1,604	0.6%	2.9%
fpppp	13	36	52	< 0.1%	< 0.1%
gcc	1,451	4	3,204	0.7%	4.3%
nasa7	23	23	168	< 0.1%	< 0.1%
sc	154	18	344	0.2%	1.2%
spice	142	158	626	0.1%	0.2%
xlisp	357	5	507	1.2%	2.3%

Figure 11: The time for solving the minimum cost flow problem for the floating-point and integer candidates as a percentage of the program's compilation time without interprocedural register allocation. The number of procedures and the number of integer and floating-point candidates are also shown.

To solve the dual minimum cost flow problem for interprocedural register allocation with spills, the problem is transformed into a minimum cost flow problem. Solutions to the minimum cost flow problem are found using the primal network simplex method[Zak95]. Though the primal network simplex method is exponential in the worst case, we found it faster in practice than a polynomial time dual network simplex algorithm available to us. Figure 11 shows the percentage of time spent running the network simplex method as a percentage of the total compilation time without interprocedural register allocation. For each benchmark, we solve two minimum cost flow problems, one with integer candidates and one with floating-point candidates. The number of procedures in each benchmark appears in column 2. Columns 3 and 4 show the number of available candidates for interprocedural register allocation. As mentioned earlier, work registers are not included as candidates. Interestingly, *espresso*, *gcc*, and *xlisp* have few floating-point candidates, but since they have a larger call graph than benchmarks, *doduc*, *fpppp*, and *spice*, all of which have more floating-point candidates, more time is spent finding a solution as a percentage of the total compilation time.

10 Conclusions

Past interprocedural register allocators have used heuristics to determine the registers to allocate to each procedure and to spill around each call. We have presented a polynomial time interprocedural register allocator that uses a model of cost to represent possible allocations. Our allocator finds a minimum cost allocation for allocating registers to each procedure and spilling registers around each call. This allocator is fast in practice and can yield significant run-time improvements.

11 Acknowledgements

We would like to thank Professor Robert Meyer and Armand Zakarian for answering our questions on network optimization problems. Armand Zakarian wrote the network solver used to test our interprocedural register allocator. Harish Patil and the anonymous referees provided helpful comments to improve the content of this paper.

References

[BCKT89] 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*, pages 275–284, July 1989.

[BL92] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 59–70, January 1992.

[Cam85] Kathie Cameron. Antichain sequences. *Order*, 2(3):249–255, 1985.

[Cha82] Gregory J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105, 1982.

[CHKW86] F. Chow, M. Himmelstein, E. Killian, and L. Weber. Engineering a RISC compiler system. In *Proceedings COMPCON*, pages 132–137, March 1986.

[CK91] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 192–203, June 1991.

[Kur95] Steven M. Kurlander. *Interprocedural Register Allocation.* PhD thesis, University of Wisconsin–Madison, 1995. In preparation.

[Orl93] James B. Orlin. A faster strongly polynomial minimum cost flow algorithm. *Operations Research*, 41(2):377–387, 1993.

[PF92] Todd A. Proebsting and Charles N. Fischer. Probabilistic register allocation. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, June 1992.

[SH89] Peter A. Steenkiste and John L. Hennessy. A simple interprocedural register allocation algorithm and its effectiveness for LISP. *Transactions on Programming Languages and Systems*, pages 1–30, January 1989.

[SO90] Vatsa Santhanam and Daryl Odnert. Register allocation across procedure and module boundaries. In *Proceedings of SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 28–39, June 1990.

[Sta93] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, October 1993.

[Wal86] David W. Wall. Global register allocation at link-time. In *Proceedings of SIGPLAN '86 Symposium on Compiler Construction*, pages 264–275, July 1986.

[Zak95] Armand Zakarian. Private communication. University of Wisconsin—Madison, February 1995.