APPROACHES TO INTERPROCEDURAL REGISTER ALLOCATION

Steven M. Kurlander

Technical Report #1294

January 1996

APPROACHES TO INTERPROCEDURAL REGISTER ALLOCATION

By

Steven M. Kurlander

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

1996

Abstract

The goal of interprocedural register allocation is to minimize execution time by selecting the registers to assign to each procedure and the registers to spill across calls. To generate an allocation, an interprocedural register allocator may consider the frequency of procedure calls and the register needs of each procedure.

Past interprocedural register allocators have applied heuristics to generate an interprocedural register allocation. This thesis presents models of interprocedural register allocation and algorithms to find a minimum cost allocation. We model both the benefit of allocating registers to procedures and the cost of register loads and stores across calls.

This thesis presents two approaches for finding an interprocedural register allocation of locals. The first approach models the benefit of allocating registers to procedures. To find an allocation, the interprocedural register allocator recursively simplifies a call graph. We add a heuristic that spills registers across calls to assign additional registers to procedures.

Our next approach solves a network flow problem to find an interprocedural register allocation of locals. With this approach we model both the benefit of allocating registers to procedures and the cost of spilling registers across calls. This interprocedural register allocator finds minimum cost allocations in polynomial time. Also, this allocator is fast in practice and can generate significant improvements in execution-time.

We extend our network flow approach to interprocedural register allocation to allocate registers to globals. Two models are presented for interprocedural register allocation of globals. The first model assumes globals are allocated registers across all procedures in the call graph, and the second divides the call graph into regions, in which globals can be spilled from registers within each region.

Acknowledgements

I am grateful to my advisor Charles Fischer. Through his more than four years of tutelage, I have learned the skills necessary for research.

Harish Patil's perpetual optimism has been very much appreciated through the ups and downs of graduate school. His insightful remarks improved my work. I enjoyed lively discussions with T. N. Vijaykumar. My thesis benefited from his extensive knowledge of Computer Architecture.

When beginning my research, Lorenz Huelsbergen and Todd Proebsting displayed the effort needed to be a successful graduate student.

I would like to thank Robert Meyer and Armand Zakarian for answering my questions on network flow optimization problems. Armand Zakarian wrote the network flow problem solver used in this thesis.

Questions and comments by Satish Chandra, Susan Horwitz, Jim Larus, and Thomas Reps improved the content of my work.

Contents

Chapter 1

Introduction

An *intra*procedural register allocator performs register allocation individually on each procedure. Common policy in current compilers using only intraprocedural register allocation is to spill a register at a call site if the register might be used by both the caller and callee. To avoid spilling a register around a call, an intraprocedural register allocator may choose not to allocate a register across a call.

An *inter*procedural register allocator, given the register needs of each procedure and the frequency of each call, can improve upon an intraprocedural register allocation. For example, an interprocedural register allocation can allocate different registers to a caller and callee to try to avoid register spilling across the call. Also, given an estimate of the procedure call frequencies, an interprocedural register allocator can try to spill registers across infrequent calls.

To estimate the register needs of each procedure, we generate an intraprocedural register allocation. An intraprocedural register allocator may assign registers to live ranges, whose interference relation can be non-transitive[Cha82]. The interference relation for interprocedural register allocation is transitive. When finding an interprocedural register allocation with no register spilling across calls, if procedure P calls Q, then we assign different registers to P and Q. If procedure Q calls R, then we assign different registers to P, Q, and R.

1.1 Contributions

This thesis presents two approaches for finding a minimum cost interprocedural register allocation. In both approaches we model the benefit associated with allocating varying numbers of registers to each procedure. In the second approach we also model the cost of spilling registers across calls.

1.1.1 A DAG-Based Approach to Interprocedural Register Allocation

We refer to the first technique as a *DAG-based* approach to finding an interprocedural register allocation. Computing a minimum cost allocation involves recursively simplifying a DAG, while searching the solution space. This approach, which is discussed in Chapter 3, models only a save-free allocation (no register spilling along the call edges). By not spilling registers around calls, some locals that are frequently referenced may not be allocated registers. We choose to add a simple heuristic as a postpass to introduce register spilling.

Though this algorithm is interesting, the worst-case time complexity of our DAG-based approach is exponential. We are able to find interprocedural register allocations for only a few of the SPEC92 benchmarks.

1.1.2 A Network Flow Approach to Interprocedural Register Allocation

Our next approach, which we discuss in Chapter 4, computes an interprocedural register allocation using network flows. This approach is more general than our DAG-based approach, as it allows us to model more aspects of interprocedural register allocation and can be solved in polynomial time. First, we present a dual network flow problem that finds a minimum cost save-free interprocedural register allocation. Next, we extend our model of interprocedural register allocation to include register spilling, and revise our network flow formulation.

Our last addition to the network flow-based approach involves modeling register allocation of globals. Wall[Wal86] proposes allocating registers to globals throughout the entire call graph. Wall's model is simple to add to our existing solution for interprocedural register allocation of locals with spilling. Santhanam and Odnert[SO90] propose allocating registers

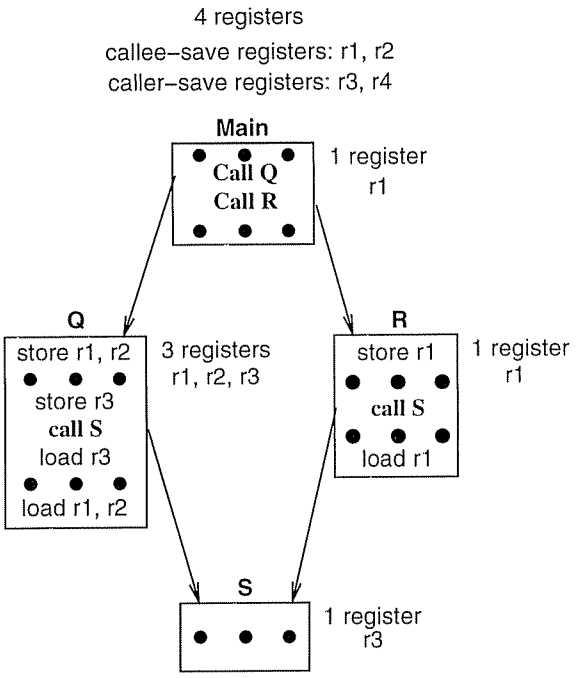


Figure 1.1: Call graph with an intraprocedural register allocation with 4 available registers.

to globals over an entire subgraph of the call graph. We further refine their approach by allowing register-allocated globals to be spilled within a subgraph. This allows sharing of registers between locals and globals in a subgraph.

1.2 A Comparison of Intraprocedural and Interprocedural Register Allocation

By convention some compilers, using *intra*procedural register allocation, designate certain registers to be saved by either the caller or callee in a procedure call [CHKW86]. Often callee-save registers contain variables that are live across a procedure call. Callee-save registers are saved only once on entrance to a procedure and reloaded upon return from the procedure, instead of spilling the register around each call. Caller-save registers are spilled if they are live across calls. These registers usually contain values not live across calls to avoid a store before a call and a reload after the call.

Figure 1.1 presents a call graph in which there are four available registers. Registers

$r1$ and $r2$ are callee-save registers and $r3$ and $r4$ are caller-save registers. The registers referenced by a procedure are shown to the procedure's right. In the call from procedure *Main* to procedure Q, *Main*, the caller, does not have any caller-save registers live across the call. However, Q, the callee, references two callee-save registers, $r1$ and $r2$ and, therefore, saves these registers on entrance to Q and reloads them upon returning from Q. In the call from procedure Q to procedure S, caller-save register $r3$ is live across the call to S. This register is saved and restored around the call. If Q called S multiple times, then a calling convention in which $r3$ is a callee-save register would result in a better register allocation. In this case $r3$ would be saved on entrance to Q and reloaded upon returning from Q. Thus, a compiler restricted by a fixed spilling convention may find it has too few caller or callee-save registers.

In the call from Q to S, S is the callee and references $r3$, which is a caller-save register. Thus, $r3$ is not spilled on entry to S. In the call from procedure *Main* to procedure R, R, the callee, references callee-save register $r1$. Thus, R saves $r1$ on entry to R and restores $r1$ upon return.

Figure 1.2 shows an interprocedural register allocation on the call graph of Figure 1.1. Next to each procedure is a table showing the benefit of allocating each additional register to a procedure. For example, procedure Q can productively use three registers. The benefit of allocating the first register is 35. The additional benefit of allocating a second register is only 30. Allocating a third register to Q adds a benefit of 25. A benefit can be an estimate on the number of dynamic loads and stores removed by allocating an additional register to a procedure. To estimate the register needs of each procedure, an intraprocedural register allocation can be performed on each procedure before an interprocedural register allocation.

The number next to each call edge is the frequency of the call. We model register spilling along the call edges. The cost of spilling a register around a call is a function of the call's frequency.

Assume each procedure is given its desired number of registers. The registers assigned to each procedure are shown below the benefits. With an intraprocedural register allocation, there can be a path through the program's execution where some registers are never

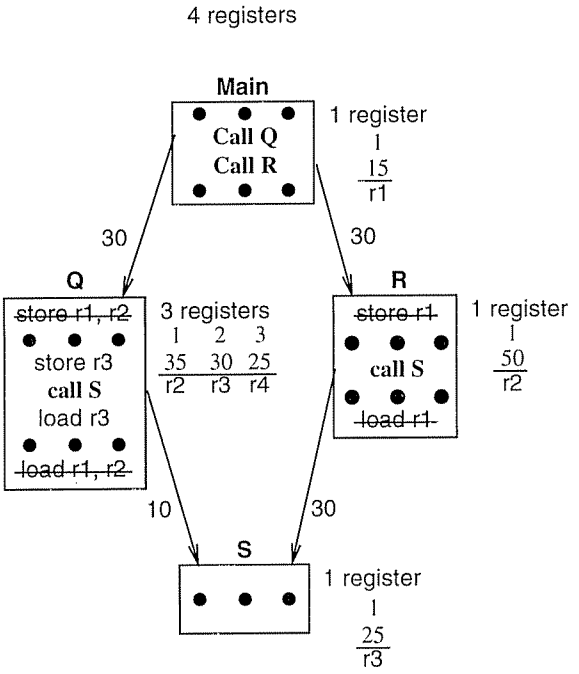


Figure 1.2: Call graph with an interprocedural register allocation with four available registers. The additional benefit for allocating each register to a procedure is shown. The number next to a call edge represents the frequency of the call.

referenced, while others are referenced and spilled multiple times. With an interprocedural register allocation, in some cases we can avoid these spills by allocating one of the unreferenced registers.

In an interprocedural register allocation, a register must be spilled only if there is a call path between two procedures and both procedures are assigned that register. By assigning register $r1$ in procedure *Main* and registers $r2$, $r3$, and $r4$ in Q, as shown in Figure 1.2, we can avoid the save and restore of $r1$ and $r2$ present in the intraprocedural register allocation of Q. Similarly, by assigning $r2$ to procedure R, the save and restore of $r1$ in R can be avoided.

If there is no call path between two procedures, these procedures can be assigned the same register. For example, both Q and R are assigned register $r2$.

An interprocedural register allocator can consider call frequency to determine where to spill a register. For example, five registers are needed along the path from *Main* to Q to S in the call graph, but there are only four available registers. An interprocedural register allocator can choose to spill a register along the call edge from Q to S, which has a frequency of 10, instead of the call from *Main* to Q, which has a frequency of 30.

Assume the cost of spilling a register around a call is twice the frequency of the call (one for a store and one for a load). In this case, an interprocedural register allocator could choose to avoid spilling the register around the call from Q to S, and not allocate a register to procedure *Main*. The benefit of this allocation is an improvement of 5 over the previous allocation, since the cost of the save/restore is 20 and the benefit of allocating a register to *Main* is only 15. We assume each procedure may always access a few temporary registers. The variable in *Main* that would have been assigned the extra register could be assigned a temporary register instead.

1.3 Context for Interprocedural Register Allocation

To perform interprocedural register allocation, our approach requires a measure of the register needs of each procedure and the cost of spilling registers across calls. The register needs

of a procedure can be estimated by the number of register references from an intraprocedural register allocation. A profile can be used to compute the number of dynamic register references in a procedure and the frequency of each call. We assume the cost of spilling a register across a call is twice the frequency of the call (one for a load and one for a store).

Given the information of the register usage and call frequency, we generate an interprocedural register allocation. In our approach, we recompile the code with this additional information. However, one could also perform interprocedural register allocation at link time [Wal86].

Since library routines are pre-compiled, we assume a fixed calling convention across calls to these routines. In this thesis, we examine the benefit of including library routines in the interprocedural register allocation. We also assume a fixed calling convention across indirect calls, since we do not know the frequency in which the caller invokes the callee.

1.4 Thesis Overview

In Chapter 2 we present architectural and software approaches to interprocedural register allocation. Chapter 3 presents our DAG-based approach to solving the interprocedural register allocation problem. With this approach we recursively simplify the call graph. Our second approach, which is discussed in Chapter 4, finds an interprocedural register allocation by solving a dual network flow problem. In Chapter 5, we prove the correctness of our network flow approach. To solve the dual network flow problem, we transform it into a primal network flow problem. Chapter 6 discusses this transformation. We extend our network flow-based interprocedural register allocator to include register allocation of globals in Chapter 7.

Chapter 2

Related Work

In this chapter, we present previous work related to interprocedural register allocation. First, we present architectural solutions and then we present software solutions.

2.1 Architectural Support to Avoid Saving and Restoring Registers Across Calls

Architectural solutions have been proposed to reduce or eliminate saving and restoring of registers across procedures calls [Moo85] [BDM87] [Hen84]. However, there are drawbacks to each of these approaches.

The architecture of the Symbolics 3600 is a modern implementation of a stack-based architecture[Moo85]. The instruction set includes 0-address instructions, in which an instruction pops all of its operands from the top of the stack, and 1-address instructions, in which one operand can be accessed from within the stack, but the other operands are accessed from the top of the stack. The few virtual-memory pages corresponding to the top of the stack are kept in a stack buffer, a fast-access memory. If at a procedure call, there is insufficient space for the next stack frame, pages containing older frames are spilled to memory, allowing the next frame to be added. Since operands are referenced off of the stack, interprocedural register allocation is not applicable.

A register-based machine has advantages over this stack-based approach. For example,

the operands in a stack-based machine can be less easily accessible than in a register-based machine. If an instruction in a stack-based machine references two operands, neither of which are at the top of the stack, an instruction is needed to move an operand to the top of the stack. A register-based machine can be more flexible, as an instruction can usually access its operands from among several registers.

A motivation in the design of the CRISP architecture [BDM87] [DM82] was to avoid register saves and restores around procedure calls. CRISP uses a stack cache, which is a set of registers containing the top elements of the stack, which can be accessed efficiently. The stack cache is addressable like memory. This means that aliased variables that cannot be kept in registers can be kept in the stack cache.

Instruction operands of the form "stack pointer plus offset" are computed before the instruction is stored in the instruction cache, saving a pipeline stage. One drawback of this approach is that a context switch causes the stack cache to be flushed to memory. A large stack cache can lead to a longer context switch.

Register windows [Hen84] [HP90] [Pat85] is an approach that tries to avoid saving registers across procedure calls. At each procedure invocation, the callee receives a bank (window) of registers. To allow for parameter passing, the caller's window can overlap with the callee's window. Register windows can be implemented as a circular buffer to give the perception of an unlimited number of register banks. Window overflow occurs when the buffer is full at a procedure call. The registers in a window are spilled to make room for the caller's registers. A drawback of implementing register windows is the additional cost and complexity. In addition, procedures need different numbers of registers. Wall found that a fixed-size register window can be inappropriate [Wal88].

2.2 Software Support for Interprocedural Allocation

In this section we examine software solutions to interprocedural register allocation. This research is motivated by Wall's seminal work on global register allocation at link time [Wal86].

2.2.1 Wall's Approach

Link time allocation allows for information to be used for interprocedural register allocation that otherwise is unavailable under separate compilation. First, the frequently referenced variables can be allocated registers, since the frequency of variable references in all procedures can be estimated at link time. Second, variables that are simultaneously live can be determined from the call graph (which can be constructed when all procedures are available). Assume procedure A calls B. Locals in A that are live across the call to B must have their registers saved if their registers are to be assigned to the locals in B.

To support register allocation at link time, the compiler annotates the object code in each module. For each procedure, the compiler generates a list of locals, the procedure call frequency, and frequency of variable references. To estimate the frequency a variable is referenced, the number of times in which a variable is referenced in a procedure is multiplied times the estimated frequency in which the procedure is called. To compute the estimated frequency in which a procedure is called, Wall adds the number of calls of the procedure, weighting calls in loops by a factor of 10.

Once an estimate on the number of references for each variable is known, Wall assigns each variable to a group, such that all the variables in a group can be assigned the same register. Each global variable is allocated to a singleton group. A frequency is computed for each group based on the frequencies of its elements. The groups with the largest frequencies are assigned registers.

Wall found substantial improvements in the execution of six benchmarks, as shown in Figure 2.1. Each improvement is over an *intra*procedural register allocation. Wall's machine had 64 registers of which 52 were available to his allocator. The first column (*register allocation*) refers to interprocedural register allocation with compile-time estimates. A procedure can allocate at most one local variable to each group, which is comprised of variables that will be assigned the same register. The second column (*with coloring*) refers to allowing a group entry to include multiple variables with non-intersecting live ranges from a single procedure—these candidates in a procedure can be assigned the same register. Graph coloring is used to select candidates with non-intersecting live ranges. The third

Benchmarks	*register allocation*	*with coloring*	*with profile*	*with both*
Livermore	18%	18%	19%	19%
Whetstone	10%	10%	10%	10%
Linpack	13%	13%	13%	13%
Stanford	25%	25%	27%	28%
Simulator	12%	14%	15%	16%
Verifier	10%	15%	16%	19%

Figure 2.1: Percentage improvement in speed with 52 registers al located to variables

column (*with profile*) refers to global register allocation using estimates from a profile of the program. The last column (*with both*) refers to using profiling and graph coloring.

Wall allows register spilling only across indirect and recursive calls. Our model of interprocedural register allocation with spilling subsumes Wall's model, as we allow register spills across all calls. Walls' allocator may not find the best allocation with respect to his model, since he allows infrequently referenced locals to be grouped together with frequently referenced locals. Our interprocedural register allocator with spilling finds allocations of minimum cost.

Wall [Wal88] compares the performance of register windows to link time allocation. For the register window approach, global variables were not kept in registers. A motivation for register windows is to avoid the interprocedural analysis needed to keep globals in registers. At link time the non-aliased global variables are known and can safely be allocated registers. Profile-based link time allocation performed better than register windows by approximately 5.4%, assuming that register windows does not increase the cycle time of a processor. The link time allocator uses procedure-call frequencies as well as frequencies of variable references. Register-windows can benefit from knowledge of the latter, as it uses the profile only to determine the most frequently referenced locals. Wall found the main advantage of the link time scheme is its ability to keep globals in registers.

2.2.2 Steenkiste and Hennessy's Approach

Steenkiste and Hennessy [SH89] designed an interprocedural register allocator for LISP programs. Their algorithm allocates registers to locals in a bottom-up fashion over the call

graph. They found that LISP programs spend a large amount of time in procedures near the bottom of the call graph (leaf procedures). Since these procedures are executed most often, they are allocated registers first. Also, since they observed that the call graph is wider near the bottom, a local near the bottom of the call graph may tend to have fewer conflicts with other variables and, therefore, be easier to keep in a register than one near the top. When possible a procedure is allocated registers not used by descendants of the procedure in the call graph. Otherwise, registers are spilled across a call if they are used by both the caller and a descendant of the caller in the call graph.

Unlike Wall's algorithm, Steenkiste's algorithm allows locals that are not live during a procedure call to be assigned registers used by procedures invoked during the call. Steenkiste finds that his interprocedural register allocation algorithm shows a 10% reduction in the number of executed instructions. Reasons given for a worse performance than Wall's algorithm include the fact that Wall had more registers available, allocated global variables to registers, and his test programs were not as recursive. Recursion makes interprocedural register allocation less effective because the registers used in a recursive procedure must be spilled to avoid being overwritten in the next invocation of the procedure. In fact, in Steenkiste's work, most of the remaining stack accesses after the register allocation were due to recursion rather than due to lack of registers.

Steenkiste and Hennessy's approach may spill registers across frequently executed routines in the internal part of the call graph. The approach to interprocedural register proposed in this thesis considers call frequencies. We are able to avoid spilling registers across frequently executed calls.

2.2.3 Chow's Approach

Chow [Cho88] presents a one-pass algorithm for interprocedural register allocation. Similar to Steenkiste's algorithm, Chow's algorithm uses a bottom-up traversal over the call graph. However, Chow's algorithm deals with incomplete procedure information, which can result from separate compilation.

Chow divides registers into two sets, callee-save registers and caller-save registers. When allocating registers to a procedures whose caller (parent in the call graph) has been processed

or is unknown, Chow saves all callee-saved registers used by the current procedure and its descendants in the call graph. This allows the procedure's caller to use the callee-save registers without conflict. The caller in this case assumes that caller-saved registers are in use. Thus, it is better to use the caller-save registers before the callee-save registers near the bottom of the call graph, which is a common optimization for leaf procedures.

Chow's results, which are not as good as those of Steenkiste, range from an improvement of 0% to 12% over *intra*procedural register allocation. Chow found that some of the save/restore overhead is minimized in intraprocedural register allocation because the allocator has the choice of which variables to put in caller and callee-save registers. He also felt 20 general-purpose registers is insufficient to allocate registers to variables to avoid spilling. His algorithm does not use profile information, which can be helpful in choosing where to spill registers.

2.2.4 Santhanam and Odnert's Approach

Santhanam and Odnert's approach to interprocedural register allocation [SO90] keeps global variables in registers throughout regions of the call graph. This improves upon Wall's approach in which once a global variable is assigned a register, the global variable keeps the register throughout the entire program. Santhanam and Odnert divide the references of each global variable into *webs*. A web is a minimal subgraph of the call graph such that neither an ancestor nor a descendant of a node in the subgraph references the global. The procedures covered by the web reference the global variable from a common register, thus avoiding loads and stores of the global within each procedure.

In this thesis, we generalize their notion of webs by allowing a register assigned to a global to be spilled within a web. The register may then be allocated to a local.

They also do interprocedural register allocation in *clusters* of frequently called procedures. The procedures in a cluster are dominated by a single procedure within the cluster, called the root node of the cluster. We refer to the root node of the current cluster as C. A cluster, however, can include the root nodes of other clusters, but no other nodes in those clusters. We refer to other cluster root nodes that are in the current cluster as O. A non-root node, therefore, is in the cluster that immediately dominates it.

The algorithm tries to place spill code for registers only at the entrance to a cluster (C), if the procedures represented by the nodes in the cluster are called more frequently than C. The algorithm visits each cluster using a bottom-up ordering, trying to allocate registers that have not been spilled by clusters O. This can allow spill code from cluster roots O in the current cluster to be moved to C, if the register is not referenced along paths between C and O (this idea is similar to moving code in a loop to the loop header). They add spill code at the root of the cluster for registers used in the current cluster.

Santhanam and Odnert find that spill code motion using clusters is not as effective as using webs to allocate registers to globals. Using a profiler they found that the combined benefit of spill code motion and allocating registers to globals is between 2% to 9% over interprocedural register allocation in their compiler.

The approach we propose in this thesis examines the *entire* call graph to generate a minimum cost allocation that spills registers as inexpensively as possible.

2.2.5 Conclusions

As shown by Wall and suggested by Chow, a profile of a program can improve the allocation of an interprocedural register allocator. Comparing the impact of a real and estimated profile will be investigated in this thesis.

Wall's allocator spills registers only in the case of recursion and indirect calls. Since Steenkiste and Hennessy perform a bottom-up allocation of the call graph, their approach may introduce register spilling around frequently executed calls near the top of a call graph. Only Santhanam and Odnert considers call-frequency when choosing where to spill registers. However, they limit spill code motion to cluster roots. In this thesis, we present an allocator that examines the entire call graph to select the placement of spill code.

Work by Wall and Santhanam and Odnert reveals the importance of allocating registers to globals. In Wall's approach, globals allocated registers remain in those registers throughout the entire execution. Santhanam and Odnert allow globals to be allocated registers in procedures where they are not referenced, which can sometimes be profitable, as this would avoid reloading the value of the global into a register. However they do not consider the potential benefit of spilling a register-allocated global and allocating that register to a local.

This refinement will be investigated in this thesis.

Chapter 3

A DAG-Based Approach

In this chapter, we present an algorithm for finding a save-free minimum cost interprocedural register allocation of local candidates. A save-free allocation does not spill registers across calls. First, we present our model for interprocedural register allocation. Then, we propose an algorithm for finding a save-free minimum cost interprocedural register allocation for call trees, which are call graphs whose structure is a tree. Next, we use the algorithm for call trees to find a save-free interprocedural register allocation for call graphs with shared procedures, which are procedures with more than one parent, only at the leaves. We then generalize our approach to allow for arbitrary acyclic call graphs. Recursive procedures, which represent cycles in the call graph, can be handled by replacing strongly connected components with a single node[SH89]. We also incorporate a heuristic for register spilling around calls to increase the number of candidates that are allocated registers.

3.1 Modeling Interprocedural Register Allocation

In the call graph each procedure P_i is represented by a node. Assume P_i has n children. The children of P_i are represented as $P_{i_1}, \ldots, P_{i_n}$. Calls between procedure P_i and P_{i_j} are represented as an edge between these nodes. The edge has a frequency $Freq[P_i, P_{i_j}]$, representing the number of times P_i calls P_{i_j} in the execution. $Freq[P_i]$ is the total number of times P_i is called over the entire execution.

The algorithms discussed in this chapter assume a "cost" of executing instructions in

a procedure at varying register levels. We choose to represent this cost as the number of memory references, since the number of memory references varies directly with the number of available registers. $Mem[P_i, r]$ is the expected number of dynamic memory references of procedure P_i given r registers to the procedure.

Function values for Mem have two interesting properties. The first is that $Mem[P_i, r] \geq Mem[P_i, r+1]$. This means that the cost of a procedure's execution does not increase as more registers are available to the procedure. We also assume that $Mem[P_i, r-1] - Mem[P_i, r] \geq Mem[P_i, r] - Mem[P_i, r+1]$; that is, the benefit of allocating registers to a procedure is monotonically non-increasing. This means that allocating the r^{th} register does not result in a bigger benefit than allocating the $r-1^{st}$ register (any reasonable register allocator will use each additional register to its best advantage).

The total cost of allocating r registers to procedure P_i and all of its descendants in the call graph is represented as $Cost[P_i, r]$. In our case, $Cost[P_i, r]$ represents the total number of memory references, distributing r registers among P_i and all its descendants.

3.2 Minimum Cost Allocation for Call Trees

This section presents an algorithm for finding a minimum cost allocation of procedures' local variables in a call tree. Since we are dealing with call trees each callee has only one caller. For each procedure in a call tree an equal number of registers will be allocated to the subtrees rooted at each of its children, since the procedures in one of its children's subtrees can never be simultaneously active with the procedures in another. Given R registers, no more than R registers can be allocated along any path from the root to a leaf in a call tree.

3.2.1 Measuring Cost

Given a call tree with a frequency along each edge as determined by function $Freq$, a maximum number of available registers R, and a function Mem for each procedure at each register level, a $Cost$ function will be defined and computed for the subtree rooted at each procedure for each register level. Based on this function a distribution of registers, n_p and n_c, will be computed for each procedure and its subtrees rooted at the children, respectively.

Apportioning r registers among P_k and its descendants, and assuming P_k has n children, we represent $Cost$ as:

$$Cost[P_k, r] = \underbrace{MIN}_{n_p+n_c=r;\ n_p,n_c\geq 0}$$

$$Freq[P_k] * Mem[P_k, n_p] + \sum_{i=1}^{n} Cost[P_{k_i}, n_c].$$

The cost of P_k when allocated n_p registers ($Mem[P_k, n_p]$) is multiplied by the frequency in which P_k is invoked ($Freq[P_k]$) to give the cost of allocating n_p registers to P_k for the entire execution. We add this cost to the total cost of allocating n_c registers to the subtrees rooted at each child P_{k_i} of P_k.

This function defines the least cost of distributing r registers to the subtree rooted at P_k by considering all possible partitions of r registers to P_k and the subtrees rooted at its children. Inductively, we have computed a minimum cost allocation at each register level for subtrees rooted at P_{k_i}. If necessary, by trying all partitions of r registers between P_k and the subtrees rooted at its children, we can determine the minimum cost allocation of r registers for the subtree rooted at P_k.

3.2.2 Generating an allocation

An allocation can be computed using dynamic programming. The algorithm maintains the least cost distribution of r registers, $0 \leq r \leq R$, between each procedure and its subtrees rooted at the children. We assign the value of n_p to a table entry indexed by $\{P_k, r\}$ when $Cost[P_k, r]$ is computed. This entry tells us how to distribute r registers between procedure P_k and its subtrees. Once the table entries have been determined at all register levels between 0 and R for all procedures, it is easy to find the number of registers to assign to each procedure. Given the number of registers available for a subtree whose root is P_k, we can determine, using the table, the number of registers to give to P_k and its subtrees. We can recursively walk down the call tree applying this information to each subtree. The recursion begins by using the table entry indexed by $\{P_0, R\}$, where P_0 is the root of the call tree, to determine the number of registers to give to P_0 and its subtrees.

3.2.3 Complexity

This algorithm has a complexity of $O(P)$, where P is the number of procedures. A naive implementation has a a constant term of R^2, where R is the number of registers, since there are at most $R+1$ way of distributing r registers, where $n_p+n_c = r$ and $0 \leq r \leq R$. However, because of the monotonicity property we can compute the distribution of r registers given the distribution of $r-1$ registers. We give the additional register to either the parent or its children, depending on which results in a greater benefit. Thus, the constant of R^2 can be reduced to R.

3.2.4 Example

Figure 3.1 gives an example call tree. Assume there are only two registers available. The values of function Mem for each procedure at each register level are shown inside the nodes. Procedures A, C, and F have only one register candidate—there is no extra benefit for allocating a second register; that is, for each of these procedures $Mem[1] = Mem[2]$—whereas all other procedures have two register candidates. The frequencies, which are all 1, are shown along the edges of the call tree. The values of the cost function for each procedure at each register level is shown in Figure 3.2. Next to each entry $Cost[P_k, j]$, we display n_p p and n_c c, which are the distribution of the j registers—the parent is given n_p registers and the subtrees rooted at the children are given n_c registers.

Since the frequency along each incoming edge to a leaf is 1, the values of the cost functions for the leaves, shown in the uppermost table of Figure 3.2, correspond to the values of function Mem for each node. In the lower table, we have the cost functions and register distribution for the other nodes. For example, $Cost[B, 1]$, the cost of allocating 1 register to the subtree rooted at B, allocates the 1 register to B, because of the greater benefit, $Freq[B]*(Mem[B, 0] - Mem[B, 1])$ $(= 10)$ $> Cost[D, 0] - Cost[D, 1]$ $(= 4)$. $Cost[B, 2]$, however, allocates the second register to D, since $Freq[B] * (Mem[B, 1] - Mem[B, 2]) = 2$ and $Cost[D, 0] - Cost[D, 1] = 4$. $Cost[C, 1]$ allocates its first register to E and F, since $(Cost[E, 0] - Cost[E, 1]) + (Cost[F, 0] - Cost[F, 1])$ $(= 14)$ $> Freq[C] * (Mem[C, 0] - Mem[C, 1])$ $(= 8)$. C is allocated the second register. The final allocation assigns one

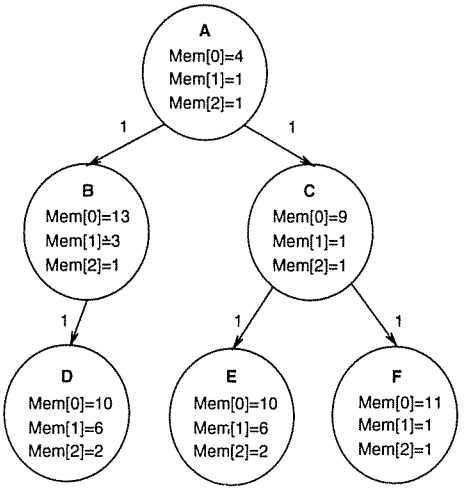


Figure 3.1: Example call tree to illustrate register allocation of locals.

Cost Tables	Alloc	Cost Tables	Alloc	Cost Tables	Alloc
Cost[D,0] = 10	0p, 0c	Cost[E,0] = 10	0p, 0c	Cost[F,0] = 11	0p, 0c
Cost[D,1] = 6	1p, 0c	Cost[E,1] = 6	1p, 0c	Cost[F,1] = 1	1p, 0c
Cost[D,2] = 2	2p, 0c	Cost[E,2] = 2	2p, 0c	Cost[F,2] = 1	1p, 0c

Cost Tables	Alloc	Cost Tables	Alloc
Cost[B,0] = 13 + 10 = 23	0p, 0c	Cost[C,0] = 9 + (10 + 11) = 30	0p, 0c
Cost[B,1] = 3 + 10 = 13	1p, 0c	Cost[C,1] = 9 + (6 + 1) = 16	0p, 1c
Cost[B,2] = 3 + 6 = 9	1p, 1c	Cost[C,2] = 1 + (6 + 1) = 8	1p, 1c
Cost[A,0] = 4 + (23 + 30) = 57	0p, 0c		
Cost[A,1] = 4 + (13 + 16) = 33	0p, 1c		
Cost[A,2] = 4 + (9 + 8) = 21	0p, 2c		

Figure 3.2: Example call tree and cost tables for each procedure.

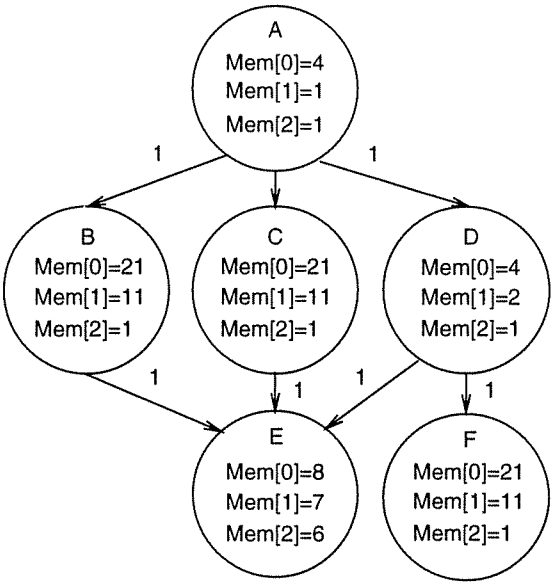


Figure 3.3: Children of a node can be allocated a different number of registers in a minimum cost register allocation of a DAG.

register to B, C, D, E, and F.

3.3 Minimum Cost Interprocedural Register Allocation for DAGs

Given R registers, an allocation for a DAG allows no more than R registers to be allocated along any path from the root to a leaf. Unlike in the call tree case, the number of registers allocated to each child of a procedure can differ in a minimum cost allocation of a DAG. For example, in Figure 3.3, the benefit of allocating two registers to B, C, or F is 20, since for each of these procedures, $1 * (Mem[0] - Mem[2]) = 20$ (each of these procedures is called once). However, the benefit of allocating two registers to E is $3 * (Mem[0] - Mem[2]) = 6$ (E is called three times). Assuming two registers are available, a minimum cost allocation of the DAG in Figure 3.3 assigns two registers to nodes B, C, and F, and 0 registers to the remaining nodes. D's child E is allocated 0 registers, but child F is allocated two registers. If neither B nor C called E, then this call graph would be a tree, and E, like its sibling F, would be allocated two registers.

To find an minimum cost solution for a DAG, the DAG is first simplified by collapsing subtrees into a singleton node using the register allocation algorithm described in Section 3.2.2. The *Cost* functions in the root nodes of these subtrees become *Mem* functions for the singleton nodes. After collapsing subtrees, the DAG can have shared nodes (more than one incident edge) at the leaves and possibly at the internal nodes. We call a shared leaf node a "level-1" node, and a shared node internal to the DAG a "level-i" node, if there are at most i shared nodes (including itself) along a path from it to a leaf.

Assume that after collapsing subtrees, a DAG's shared nodes are only at level-1 (shared nodes appear only as leaves of the DAG). We can split these shared nodes such that each parent has an edge incident on its own copy of the node. The DAG is now a tree. Assigning, in turn, each possible combination of register values to these copies, such that the copies of a shared node are allocated the same number of registers, we can find a solution using the allocation algorithm for trees.

To ensure that each copy of a shared node is allocated the same number of registers, r, $0 \leq r \leq R$, we can assign to each copy a cost of infinity below register level r, and a cost of b at register level r and above, where b is the cost of allocating r registers in the shared node. Each copy will allocate at least r registers, as there is an infinite decrease in cost between allocating $r-1$ and r registers. It is possible that more than r registers will be allocated to each copy if the registers beyond r cannot be profitably used elsewhere (there is no benefit, decrease in cost, in allocating more than r registers to each copy). However, a pass over the DAG after the allocation can limit the number of registers allocated to r.

If a DAG has i shared nodes at level-1, and there are R registers available, we would check no more than $O((R+1)^i)$ combinations of register assignments. The allocation with the minimal cost is the best solution.

3.4 Shortening the Search Space

Rather than checking all possible register values for each shared node (a potentially explosive process), we can shorten the search space using a branch and bound technique.

An important optimization involves tightening the bounds on the maximum and minimum number of registers that a shared node can have in a minimum cost solution for a DAG. Before computing the maximum and minimum bounds on the number of registers available to a shared node at level-1, we first precompute for each procedure P_k the expression

$$Mem'[P_k, j] = \sum_{i=1}^{n} Freq[P_i, P_k] * Mem[P_k, j]$$

for $0 \leq j \leq R$, where P_i is a parent of P_k, $Freq[P_i, P_k]$ is the frequency along the edge between P_i and P_k, and n is the number of P_k's parents. $Mem'[P_k, j]$ is the cost of executing P_k throughout the entire execution, assuming j registers are allocated in P_k. This has the effect of folding the call frequency of P_k into its cost function Mem'.

A call graph may contain transitive edges. A transitive edge connects two nodes, which are connected by a path that does not include this edge. Transitive edges affect the call frequency of a node, but do not further constrain the solution by affecting which registers can be shared among procedures. Since the call frequencies have been folded into the cost of allocating registers for each node, we can simply remove transitive edges from the call graph.

3.4.1 Finding the Maximum Bound

To find the maximum bound for a shared node at level-1, we split the shared node, such that each parent has its own copy (see Figure 3.4). We let each new leaf, D_1 and D_2 in Figure 3.4, have the same Mem' function as D in the original DAG. Using the register allocation algorithm for trees (using Mem' instead of $Freq*Mem$), we compute the number of registers allocated to D_1 and D_2 (they need not be equal). Without loss of generality, assume D_1 is allocated less than or equal to the number of registers allocated to D_2. Let r be the number of registers allocated to D_1. We now show that r is a maximum bound on the number of registers allocated to D in a minimum cost allocation.

We can add an edge between C and D_1, as shown in Figure 3.5(a). Since D_2 is allocated at least as many registers as D_1, there are no more than R registers allocated along the path A–C–D_1. Because the edge does not affect the number of registers allocated to each

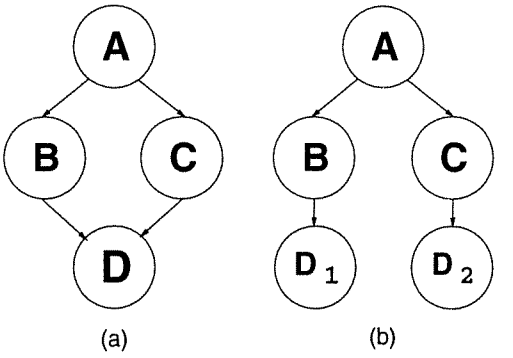


Figure 3.4: Splitting a shared leaf node.

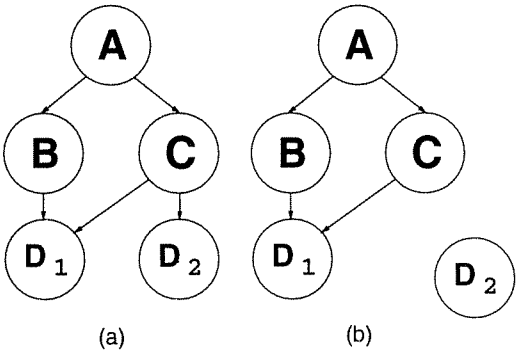


Figure 3.5: Adding an edge between nodes C and D_1, and decreasing the number of registers allocated to D_2 produces a DAG in which D_1 has at least as many registers as D of the original DAG.

node, a minimum cost allocation of the tree before the edge is added is still a minimum cost allocation after the edge is added.

Now we decrease the benefit of allocating registers to D_2. Registers previously available to D_2 may now be allocated to D_2's ancestors. Since D_1 was allocated less than or equal to the number of registers allocated to D_2, additional registers allocated to D_2's ancestors may lead to fewer registers allocated to D_1 (the number of registers allocated along each path from the root to a leaf cannot exceed R). The total number of registers allocated to D_1, however, will not exceed its original value of r. Decreasing the benefit of allocating registers to D_2 has the effect of removing D_2 from the DAG (Figure 3.5(b)). The DAG in Figure 3.5(b) now has the same structure as the DAG in Figure 3.4(a). The total number of registers allocated to D_1 in Figure 3.4(a), r, is a maximum bound on the number of registers allocated to the shared node D.

3.4.2 Finding the Minimum Bound

To compute the minimum bound for a shared node at level-1, we compute Mem' and split the shared node, such that each parent has its own copy, as was done for computing the maximum bound (Figure 3.4). However, the Mem' function for each copy is the function Mem' of D divided by p, where p is the number of parents of the shared node D. In our example, p is 2. Next, we find a solution for the tree in Figure 3.4(b). Without loss of generality, assume D_1 has less than or equal to the number of registers available to D_2. Let the current number of registers allocated to D_1 be r. We now show that r is a minimum bound on the number of registers allocated to D in a minimum cost allocation.

As in the case for finding the maximum solution, we add an edge between C and D_1 (see Figure 3.6(a)). In our next step, we add an edge between B and D_2 (see Figure 3.6(b)). The path A–B–D_2 may allocate more than R registers, since D_2 is allocated at least as many registers as D_1. If we remove registers from A or B, then D_1 can be allocated more than r registers, bringing the number of registers allocated to D_1 up to D_2's level. If we remove registers from D_2, the number of registers available to D_2 will not drop below D_1's count; D_1 and D_2 have the same constraints with the other nodes, they will have the same number of registers. We can combine D_1 and D_2 into a single node transforming the DAG

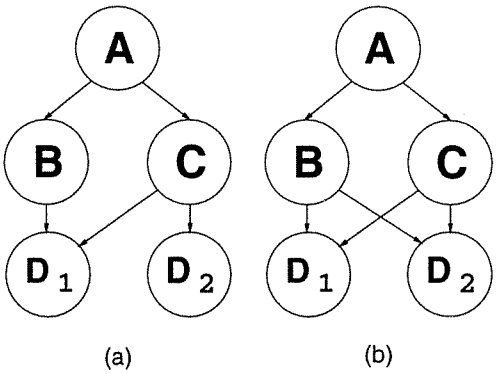


Figure 3.6: Adding constraints between nodes C and D_1, and B and D_2.

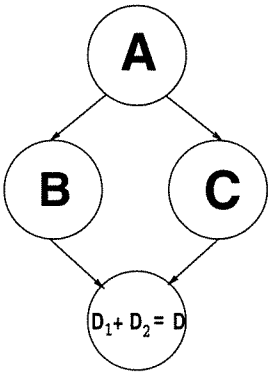


Figure 3.7: We can combine nodes D_1 and D_2 to transform the DAG into the original DAG.

of Figure 3.6(b) into the original DAG of Figure 3.7. The sum of the Mem' functions of D_1 and D_2 equals that of D. Since the Mem' function of D is greater than that of D_1 and the number of registers now allocated to D_1 is greater than or equal to r, r is a minimum bound on the number of registers allocated to D.

3.5 Level-n Shared Nodes

We can recursively apply the register allocation algorithm for DAGs with level-1 shared nodes to generate an allocation for DAGs with level-n shared nodes. Assume, for example, that we want to find an allocation for the DAG with the level-2 shared node (D) in Figure 3.8(a). We apply our algorithm for level-1 shared nodes to split the shared node F at level-1 into F_1 and F_2 (Figure 3.8(b)) to find minimum and maximum bounds on the number of registers to be allocated to F. Treating the level-2 shared node D as the root of

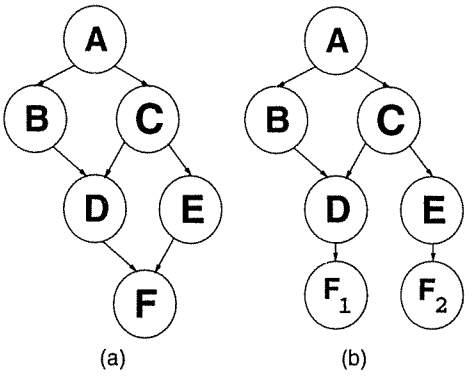


Figure 3.8: To find an allocation for the level-1 shared nodes we first split the level-1 shared nodes.

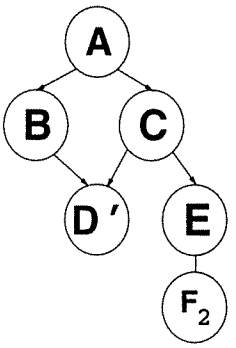


Figure 3.9: We collapse the subtree rooted at D into a new node D'. We then find the allocation for the new DAG, in which D' is a level-1 shared node.

a subtree, our allocator computes a cost function and allocation for the subtree rooted at D. We collapse the subtree rooted at D to form a new node D' (Figure 3.9).

If we can determine the minimum cost allocation for D', we can determine the allocation for each node within the corresponding subtree. D' can be treated as a level-1 shared node. We now have a new level-1 DAG for which we know how to generate its register allocation and, thus, we can determine the number of registers allocated to D'. Once we know the allocation for D', we can compute the allocation at the split nodes F_1 and F_2.

Computing the cost of allocating registers within the minimum and maximum bounds is identical to the process outlined above.

3.6 Complexity

The complexity of our algorithm is exponential in the worst case. As the height of the call graph increases by one, we may double the amount of work needed to find an allocation. This leads to an exponential worse case time complexity.

3.7 Register Spilling

By using a save-free allocation we have avoided adding spill code around calls. Procedures, however, may need more registers than are allocated by a save-free register allocation. Our approach for adding register spilling allows procedures that can use additional registers to spill registers used by an ancestor or descendant in the call graph. Registers allocated save-free to an ancestor of a procedure p can be saved and restored on entrance and exit to p (treated as a callee-save register), and save-free registers used by a descendant of p can be saved and restored around calls in p (treated as a caller-save register). We choose to spill registers in which the least cost is incurred and only if it is profitable—the cost of the added spill to a procedure is less than the benefit of an additional register.

To choose between allocating a caller-save or callee-save register, we provide our algorithm with a function $Live[P_k, r]$, which equals one if the r^{th} register is live across a call in P_k, and zero otherwise. We compute the cost of spilling a callee-save register in procedure P_k as $2 * Freq[P_k]$, where the value 2 represents the store and load added to spill the register upon entering and exiting the routine, and $Freq[P_k]$ is the frequency in which P_k is called. Letting P_{k_i} be one of the n children of P_k, we compute the cost of a caller-save register r as $2 * Live[P_{k_i}, r] * \sum_{i=1}^{n} Freq[P_k, P_{k_i}]$. The benefit of an additional register for procedure P_k over the current register level r is simply $Mem[P_k, r] - Mem[P_k, r+1]$.

3.8 Implementation

We implemented our algorithm using gcc [Sta93] on a DECstation 5000/125 with a MIPS R3000 processor. To generate a call graph and cost functions for performing interprocedural register allocation, we modified gcc to write information for each procedure on register

speedup over gcc -O2		
benchmark	*save-free*	*with spilling*
compress	0.1%	0.5%
decompress	-0.4%	1.2%
ear	-0.4%	0.3%
eqntott	-0.2%	0.1%
ora	0.3%	1.0%

Figure 3.10: Speedups of interprocedural register allocations with and without register spilling over gcc at optimization level 2.

usage. In addition, Gcc was modified to assume that all live ranges are not spilled (split) because of calls, rather than using the usual caller-save/callee-save convention of the MIPS architecture. This gives our interprocedural register allocator a better estimate of the benefit of allocating save-free registers, which are not spilled across calls.

We weight register references in loops by 10^d, where d is the loop's nesting level. To compute function $Mem[P_k, r]$, the number of memory references for procedure P_k given r registers, we compute the total number of references of the r most referenced registers, as determined by gcc, and subtract this from the total number of references for the procedure. To compute $Live[P_k, r]$, we simply check whether the r^{th} most referenced register is live across a call. The interprocedural register allocator also reads a dynamic profile of the program. The allocator uses the profile to compute the frequency along the edges as well as to improve upon our estimation of function Mem. If procedure P_k's execution represents $p\%$ of the total number of cycles t, we let $Mem[P_k, 0] * Freq[P_k] = p\% * t$, where $Freq[P_k]$ is the number of times P_k is called.

The interprocedural register allocator generates information on the registers that are save-free and spilled. Gcc then reads this information to produce the corresponding interprocedural allocation. Our interprocedural register allocator could be included as part of a link time allocation as, for example, implemented by Wall [Wal86].

Figure 3.10 gives speedups from a few SPEC92 benchmarks[[Sp] of our save-free allocation and our save-free allocation followed by register spilling. As we have not performed

benchmark	*time spent computing allocation as a percentage of total compilation time*
compress	13.8%
decompress	12.0%
ear	12.2%
eqntott	22.1%
ora	29.5%

Figure 3.11: Times for computing a minimum cost allocation with register spilling and allocation of globals as a percentage of the total compilation time.

interprocedural register allocation on library routines, each allocation (including gcc's) spills caller-save registers around library calls.

Adding register spilling to our approach generates only a small improvement over gcc's intraprocedural register allocation. A weakness of our approach is that we may have an insufficient number of callee-save registers or caller-save registers. We may have too few caller-save registers in a leaf routine, where we can avoid spilling a caller-save register. For a routine near the top of the call graph, spilling caller-save registers can be much more expensive then spilling a callee-save register. A caller-save register will be saved and restored around each call, whereas a callee-save register is saved and restored only upon calling and returning from the routine.

Our results indicate that a save-free allocation can be competitive with a gcc allocation. In addition, using our approach for spilling registers around calls, only a small benefit is achieved from register spilling.

Figure 3.11 presents the time finding an interprocedural register allocation as a percentage of the total compilation time without interprocedural register allocation. Our interprocedural register allocation represents a significant percentage of the total compilation time.

As mentioned in Section 3.4, our algorithm removes transitive edges to simplify a call graph. After removing transitive edges, the call graph for each of these benchmarks has a maximum shared level, the largest number of shared nodes along a path from the root of the call graph to a leaf, of three or less. On benchmarks with a shared level greater than

three, our algorithm required too much space on our machine.

In the next chapter, we present an interprocedural register allocator that applies a network flow approach. Applying network flows yields a much faster interprocedural register allocator that can efficiently find allocations for large call graphs.

3.9 Conclusions

This chapter presents a DAG-based minimum cost interprocedural register allocation algorithm. This algorithm has been extended to perform register spilling around calls. We found that for a sample of the SPEC92 benchmarks, a save-free approach is comparable with gcc's intraprocedural allocation, which may spill registers around calls. Unfortunately, due to the space required by this algorithm, we are able to run it only on a subset of the SPEC92 benchmarks.

Chapter 4

A Network Flow Approach

This chapter 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 utilize 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 4.2, we generalize our allocation model to allow for register spilling across calls.

Our approach can be used with conventional compilers that translate one procedure

[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.

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.

4.1 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 4.2, 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 4.4 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.

4.1.1 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 set of 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 4.1(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

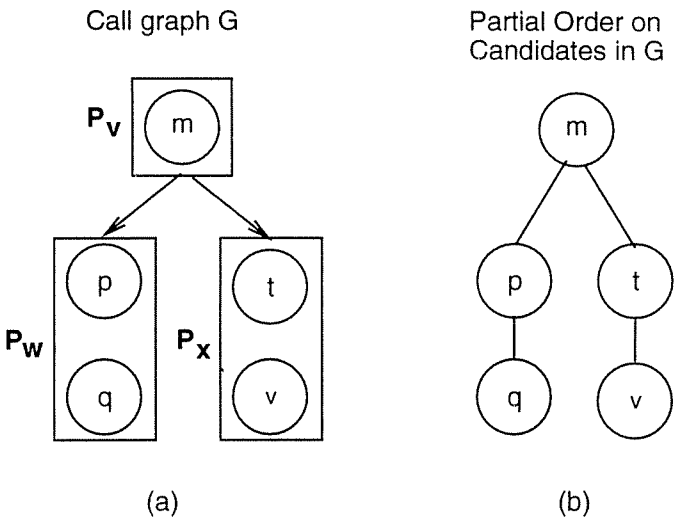


Figure 4.1: An example call graph, allowing for multiple candidates in a procedure, and a partial order on the candidates of the call graph.

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

4.1.2 Interference Graph

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 be non-transitive[Cha82]. In an intraprocedural register allocation, two live ranges that interfere are assigned different

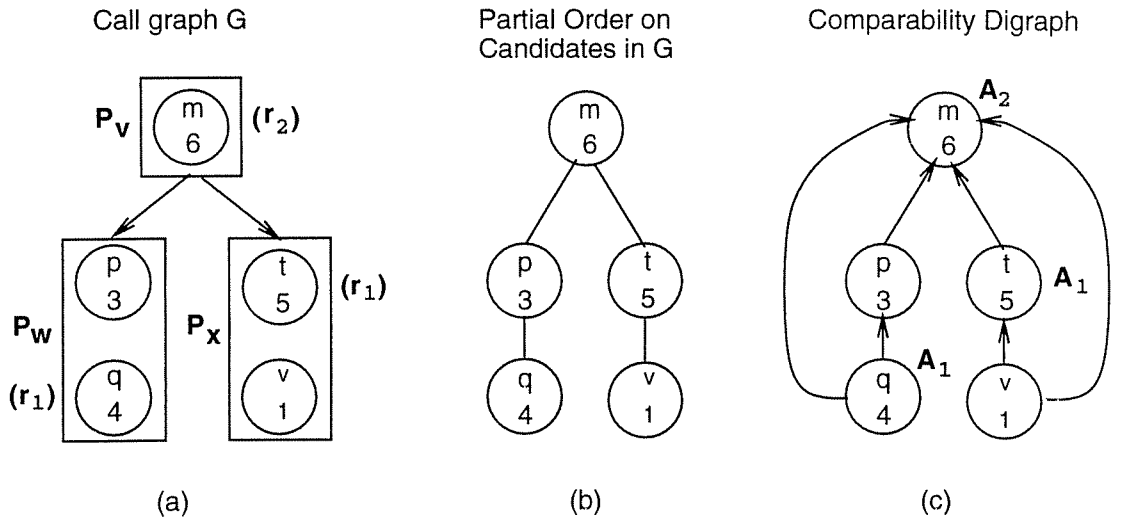


Figure 4.2: A call graph G, a partial order on the candidates in G, and the comparability digraph of the candidates in G.

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 4.2 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.

4.1.3 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 $D(S)$. In Figure 4.2(c), $\{p,t\}$, $\{p,v\}$, $\{q,v\}$, and $\{q,t\}$ are examples of 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 4.2(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 4.2(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.

4.1.4 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 4.2(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.

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 j such that $c_j \in S$

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 objective coefficient (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 4.3(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

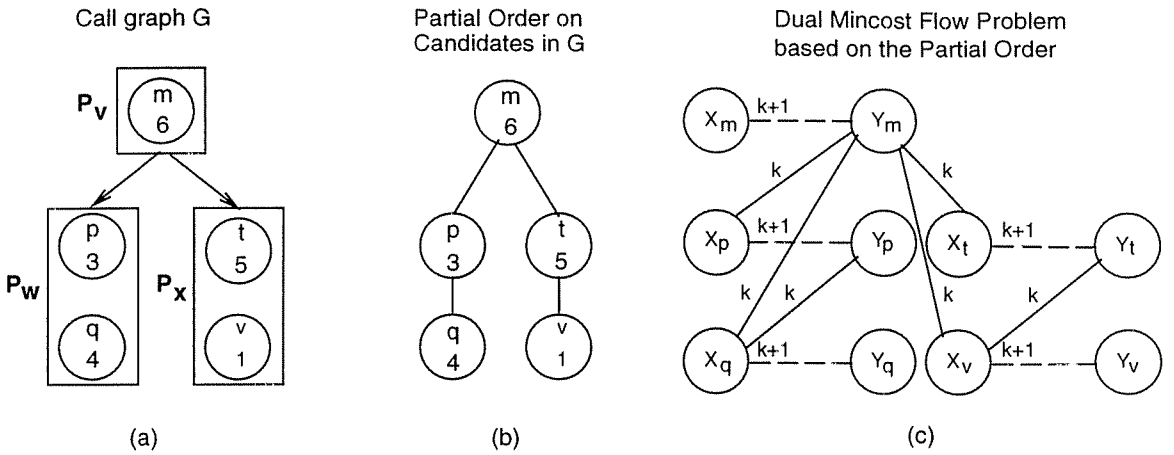


Figure 4.3: Example call graph G, graph of a partial order on the candidates in G, and the dual minimum cost flow problem with respect to the partial order. Each node in the dual minimum cost flow problem represents an integer-valued dual variable, and each edge represents a constraint between two dual variables. The constant above an edge is the integer upper bound in the corresponding constraint.

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, $0 < m < h$.

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

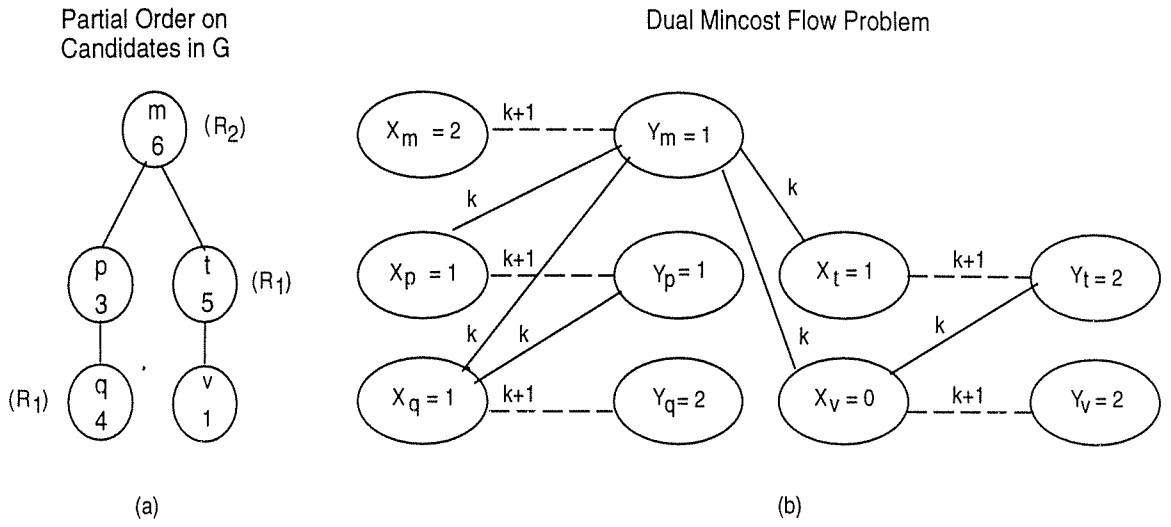


Figure 4.4: Partial order on candidates in G, and graph of dual minimum cost flow problem for G.

$A(z)$ 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.

4.1.5 Example

Figure 4.4(a) shows the partial order on the candidates of call graph G of Figure 4.3. Candidates along a path must be assigned to distinct abstract registers. Let R_p and R_q be 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 4.4(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 4.4(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.

4.2 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 4.1.1. We refer to this partial order as ($\sqsubseteq$) throughout Section 4.2.

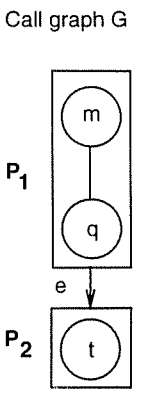


Figure 4.5: Example call graph G. A partial order exists only among candidates in each procedure.

Since partial order ($\sqsubseteq$) only relates candidates in the same procedure, there is an ordering between q and m in Figure 4.5. 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 $(free\_out_1, \ldots, free\_out_{|E|})$.

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).

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 $q < p$.

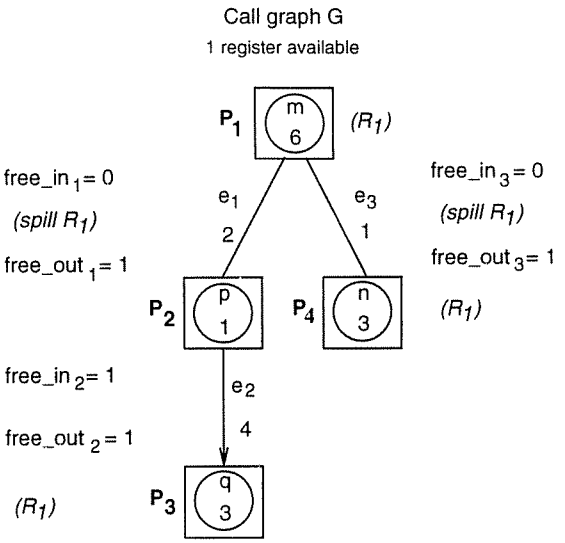


Figure 4.6: Interprocedural register allocation of call graph G.

Each candidate $c_j \in C(P)$ has a positive integer objective coefficient, w_j, which represents the benefit of allocating a register to the candidate. Each call edge $e_j \in E$ has a positive integer objective coefficient, s_j, which represents the cost of spilling a register across e_j, 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 4.6 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

Dual Variables

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

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

assigned to n.

4.2.1 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 4.7.

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 4.7. Let $Q^*(k, G)$ be solutions to interprocedural register allocation with spilling. In chapter 5, 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 4.7 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 4.1, 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 4.1, 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 $\sum_{e_j \in E} s_j * k$.

4.2.2 Example

Figure 4.8(a) shows call graph G of Figure 4.6 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 4.8(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 4.8(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.

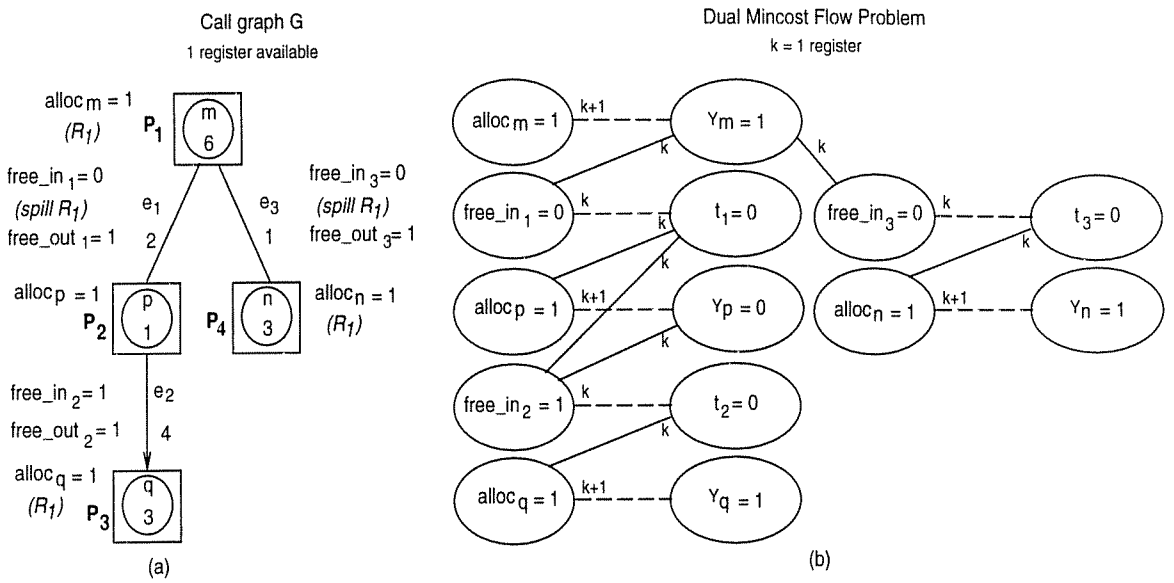


Figure 4.8: Example call graph G and graph representation of the dual minimum cost flow problem for G.

Assume $k = 1$ in Figure 4.8(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).

4.3 Complexity

For p candidates and edges in a call graph, the number of dual variables in the dual minimum cost flow problem of Section 4.2 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 $O(p^3 \log p)$.

4.4 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 4.9, 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 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.

4.5 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

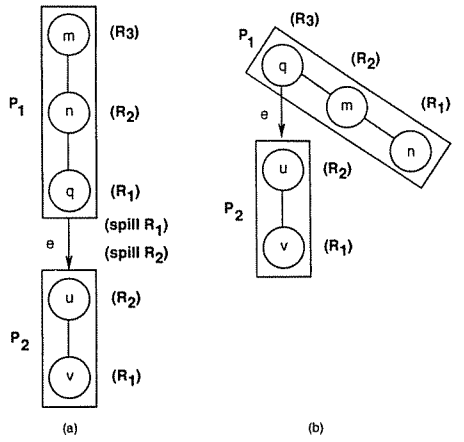


Figure 4.9: By distinguishing between candidates live and not live across calls, fewer registers are spilled.

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 4.7 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.

4.6 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 4.7. 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.

4.7 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 benchmarks *nasa7* and *swm256*, 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 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' benefit.

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.

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 is j. Register candidates 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, candidates are always allocated a register.

Column two of Figure 4.10 shows the execution-time improvement from adding our interprocedural register allocator to gcc. The benchmarks are compiled at optimization level O2 with loop-unrolling enabled. We assume that library routines have been pre-compiled using a caller-save/callee-save convention for spilling registers around calls. Our interprocedural register allocator finds 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, 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, in which our allocator avoids spilling registers.

On benchmark *su2cor* our allocator performs worse than simply running gcc's intraprocedural register allocator. A drawback of our algorithm is that a candidate live across calls may not be allocated a register because of the cost of a register spill, but there still may be a benefit of allocating that candidate a register that is not live across calls and does not require a register spill. We can add to our allocator a postpass that can allocate additional registers not live across calls. Let call edge $e_j \in OUT(P_v)$. If $free\_in_j = r$, then there are r registers that are free on exit from procedure P_v. We can treat these r registers in P_v as caller-save registers and allocate them to P_v. The improvement of our interprocedural register allocator with this optimization over gcc is shown in column three (*Minimum Cost+*)

Performance Improvements				
benchmark	*Min Cost*	*Min Cost+*	*Steenkiste and Hennessy*	*Min Cost+ w/Static Prof*
compress	1%	2% (+1%)	0%	0%
doduc	5%	5% (+0%)	4%	1%
ear	0%	0% (+0%)	0%	0%
eqntott	0%	0% (+0%)	0%	0%
espresso	9%	9% (+0%)	7%	7%
fpppp	4%	4% (+0%)	3%	2%
gcc	8%	8% (+0%)	1%	3%
hydro2d	0%	0% (+0%)	0%	0%
mdljdp2	1%	1% (+0%)	1%	0%
mdljsp2	0%	0% (+0%)	0%	0%
nasa7	0%	1% (+1%)	0%	0%
ora	1%	1% (+0%)	1%	1%
sc	11%	11% (+0%)	7%	-2%
spice	3%	3% (+0%)	2%	3%
su2cor	-1%	0% (+1%)	-1%	-1%
swm256	0%	0% (+0%)	0%	0%
xlisp	11%	11% (+0%)	-3%	5%

Figure 4.10: Column two shows the execution-time improvement from adding our minimum cost interprocedural register allocator with spills to gcc. Column three shows the improvement of our interprocedural register allocator with a postpass optimization, which allocates additional registers to each procedure. Column four shows the performance improvement from adding Steenkiste and Hennessy's bottom-up interprocedural register allocator to gcc. The last column shows the execution-time improvement from adding our minimum cost interprocedural register allocator with static profiling and the postpass optimization to gcc.

in Figure 4.10. The numbers in parentheses are the performance increase over our interprocedural register allocator without this optimization. Overall, this optimization yields only a slight improvement.

Column four shows the results from adding Steenkiste and Hennessy's bottom-up interprocedural register allocator[SH89] to gcc. Running Steenkiste and Hennessy's bottom-up register allocator on *xlisp* 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.

The final column shows the performance of our minimum cost interprocedural register allocator with the postpass and static profiling. We use the *call+1-loop(10)* estimate proposed by Wall[Wal91]. For a procedure invocation an instruction contributes 10^d to the execution count, in which d is the instruction's loop-nesting level within the procedure. Each procedure is called one plus the static number of calls to the procedure. Among the static profile estimates tested by Wall, he found this one to be among the best.

On benchmarks *doduc* and *sc*, our interprocedural register allocator with static profiling performs worse than Steenkiste and Hennessy's allocator. Steenkiste and Hennessy's bottom-up allocator may spill registers at points higher in the call graph than our allocator with static profiling. This effect leads to a better performance by Steenkiste and Hennessy's allocator on these benchmarks, but results in a better performance by our allocator with static profiling on *xlisp*.

With a static profile, our allocator's overall performance is competitive with Santhanam and Odnert's allocator. Using the static profiling heuristics of Ball and Larus [BL93], we would expect better results.

Steenkiste and Hennessy's allocator is faster (runs in linear-time in the size of the input). Given only static profile information, their allocator seems preferable.

Figure 4.11 shows the improvement over gcc from profiling and measuring a benchmark's performance on the standard input. The numbers in parentheses represent the increase in performance with respect to training on a short execution-time input and measuring performance on the standard input. Interestingly, the largest improvements are on benchmarks

Profiling the Standard Input	
benchmark	*improvement*
compress	1% (+0%)
doduc	5% (+0%)
ear	0% (+0%)
eqntott	0% (+0%)
espresso	11% (+2%)
fpppp	4% (+0%)
gcc	10% (+2%)
hydro2d	0% (+0%)
mdljdp2	1% (+0%)
mdljsp2	1% (+1%)
ora	1% (+0%)
sc	11% (+0%)
spice	3% (+0%)
su2cor	-1% (+0%)
xlisp	11% (+0%)

Figure 4.11: Improvement over gcc from using the same input to both profile and measure a benchmark's performance.

espresso and *gcc*. Both of these benchmarks have multiple input files. These benchmarks are trained on a dynamic profile that is a combination of profiles from their different input files. For other benchmarks, profiling the short execution-time input is a good predictor of the standard input.

Figure 4.12 shows the performance improvement over gcc from running our interprocedural register allocator on both the library and user routines. The numbers in parentheses represent the increase in performance with respect to running our interprocedural register allocator only on user routines. The C source code was available for many of the library routines. Since some system calls overwrite the caller-save registers, we still follow the caller-save/callee-save convention around system calls. Also, we follow the caller-save/callee-save convention around library routines written in assembly code. Overall, we found only a small improvement from interprocedural register allocation of library routines. Since many library calls lead to system calls, in many cases registers live across calls to library routines will still be spilled.

Since the user code in benchmark *ora* does not reference all the general-purpose registers, our interprocedural register allocator avoids some register spilling when allocating registers

Interprocedural Register Allocation of Library and User Routines	
benchmark	*improvement*
compress	1% (+0%)
doduc	5% (+0%)
ear	1% (+1%)
eqntott	0% (+0%)
espresso	9% (+0%)
fpppp	5% (+1%)
gcc	10% (+2%)
hydro2d	1% (+1%)
mdljdp2	1% (+0%)
mdljsp2	0% (+0%)
nasa7	0% (+0%)
ora	2% (+1%)
sc	11% (+0%)
spice	3% (+0%)
su2cor	-1% (+0%)
swm256	0% (+0%)
xlisp	11% (+0%)

Figure 4.12: Performance improvement over gcc from interprocedural register allocation on library routines and user code.

to library routines. In benchmarks *gcc* and *fpppp*, register spills in frequently called library routines are moved to less frequently called routines. Interprocedural register allocation of library routines also allows *yylex* in benchmark *gcc* to avoid spills across some library calls.

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 4.13 shows the time 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. We assume library routines are pre-compiled. The number of procedures in each benchmark appears in column two. Columns three and four 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.

Figure 4.14 shows results from running our interprocedural register allocator on both user and library routines. The number of procedures, the number of candidates, and the time spent running the network simplex method as a percentage of the total compilation time are shown. The numbers in parentheses represent the percentage increase from not including library routines in the interprocedural register allocation. The absence of a percentage increase in the columns under *candidates* indicates there were no candidates allocated registers when library routines were pre-compiled. The absence of a percentage increase in the columns *% of compilation time* indicates that the percentage was less than 0.1% when library routines were pre-compiled.

Including library routines in the interprocedural register allocation represents a significant increase in the number of routines for the smaller benchmarks. These smaller benchmarks also show a large increase in the number of candidates and the time running

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%
ear	107	88	220	0.2%	0.7%
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%
hydro2d	44	15	175	< 0.1%	0.2%
mdljdp2	55	35	138	0.1%	0.5%
mdljsp2	52	43	165	0.2%	0.6%
nasa7	23	23	168	< 0.1%	< 0.1%
ora	2	10	7	0.1%	0.2%
sc	154	18	344	0.2%	1.2%
spice	142	158	626	0.1%	0.2%
su2cor	41	41	239	< 0.1%	0.4%
swm256	9	14	114	0.1%	0.3%
xlisp	357	5	507	1.2%	2.3%

Figure 4.13: 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 inter-procedural register allocation. The number of procedures and the number of integer and floating-point candidates are also shown.

benchmark	*procedures*	*candidates*		*% of compilation time*	
		floating-point	*integer*	*floating-point*	*integer*
compress	39 (+143%)	1	102 (+229%)	0.1%	0.7% (+133%)
doduc	163 (+288%)	180 (+6%)	570 (+108%)	0.2%	0.8% (+300%)
ear	195 (+82%)	135 (+53%)	349 (+59%)	0.4% (+100%)	0.6% (-14%)
eqntott	84 (+35%)	1	242 (+36%)	0.1% (+0%)	0.9% (+13%)
espresso	400 (+11%)	2 (+100%)	1,713 (+7%)	0.5% (-17%)	2.6% (-10%)
fpppp	149 (+1046%)	49 (+36%)	354 (+580%)	0.1%	0.2%
gcc	1,512 (+4%)	5 (+25%)	3,351 (+5%)	0.5% (-29%)	5.0% (+16%)
hydro2d	189 (+329%)	28 (+87%)	498 (+185%)	0.2%	0.9% (+350%)
mdljdp2	222 (+303%)	48 (+37%)	484 (+151%)	0.3% (+200%)	0.7% (+40%)
mdljsp2	220 (+323%)	56 (+30%)	511 (+209%)	0.3% (+50%)	0.8% (+33%)
nasa7	184 (+700%)	37 (+61%)	514 (+205%)	0.2%	0.8%
ora	126 (+6200%)	15 (+50%)	303 (+4228%)	0.2% (+100%)	1.0% (+500%)
sc	240 (+56%)	34 (+89%)	517 (+50%)	0.3% (+50%)	1.4% (+17%)
spice	295 (+107%)	207 (+30%)	968 (+55%)	0.2% (+100%)	0.3% (+50%)
su2cor	216 (+426%)	61 (+49%)	598 (+150%)	0.2%	0.6% (+50%)
swm256	136 (+1411%)	21 (+50%)	361 (+217%)	0.2% (+100%)	0.7% (+133%)
xlisp	399 (+12%)	19 (+280%)	583 (+15%)	0.9% (-25%)	1.2% (-48%)

Figure 4.14: The time spent 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. Interprocedural register allocation is performed on both user and library routines.

the network simplex method. Despite these large increases, the time running the network simplex method is still small. In fact, by compiling library routines, the time solving the network flow problem as a percentage of the total compilation time can decrease, since compiling library routines adds to the total compilation time.

4.8 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.

We found our allocator to be fast in practice and yield significant run-time improvements for some benchmarks. Our interprocedural register allocator performs much better with a dynamic profile. In addition, profiling benchmarks on input leading to short execution times is sufficient for generating good results. With a static profile, our allocator yields improvements that are competitive with Steenkiste and Hennessy's interprocedural register allocator. Performing interprocedural register allocation on library routines generates only a small improvement in execution-time.

Chapter 5

Proof of Correctness

In this chapter we prove that there is a bijection from solutions of the problem of interprocedural register allocation of locals with spilling (I.1 – I.7) in Section 4.2 to solutions of the dual minimum cost flow problem (D.1 – D.9) in Section 4.2.1. To find an interprocedural register allocation we can solve the dual minimum cost flow problem and apply the inverse function. Our proof is structured similar to Cameron's for proving a bijection from maximum weight k-antichain sequences to a network flow problem [Cam85].

For convenience we reproduce the interprocedural register allocation problem (I.1 – I.7) in Figure 5.1 and the corresponding dual minimum cost flow problem (D.1 – D.9) in Figure 5.2.

Given k registers and a call graph G, let $Q(k, G)$ represent the set of valid interprocedural register allocations following constraints *I.1 - I.6*, which excludes the maximization function found in *I.7*. Also, let $P(k, G)$ represent solutions to *D.1 - D.8*, which excludes the objective function in *D.9*. Both $P(k, G)$ and $Q(k, G)$ contains solutions that are not optimal with respect to their maximization functions.

For $z \in P(k, G)$, function $I(z)$ shown in Figure 5.3 (same mapping as in Section 4.2.1) maps solutions to $I \in Q(k, G)$. For $I \in Q(k, G)$, function $z(I)$ shown in Figure 5.4 maps solutions to $z \in P(k, G)$.

In this paper, we first prove that if $z \in P(k, G)$, then $I(z) \in Q(k, G)$ and if $I \in Q(k, G)$ then $z(I) \in P(k, G)$. Next, we restrict $z \in P(k, G)$ and $I \in Q(k, G)$ to maximal solutions

Interprocedural Register Allocation with Spilling

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

Figure 5.1: Defining an interprocedural register allocation of locals with spilling

Dual Minimum Cost Flow Problem

Dual Variables

$(x_i,\ y_i$ for i such that $c_i \in C(P)), (r_j,\ t_j$ for j such that $e_j\ inE)$

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 5.2: Dual minimum cost flow problem whose solutions map to interprocedural register allocations with spilling.

Iz.1 $I(z) = (R(z),\ free\_in(z),\ free\_out(z))$.

Iz.2 for $1 \leq h \leq k$, $R_h(z) = \{c_j \mid x_j + y_j = k+1, x_j = h, c_j \in C(P)\}$;
$R(z) = (R_1(z), \ldots, R_k(z))$.

Iz.3 for $e_j \in E$, $free\_in_j(z) = r_j$; $free\_in(z) = (free\_in_1(z), \ldots, free\_in_{|E|}(z))$.

Iz.4 for $e_j \in E$, $free\_out_j(z) = k - t_j$; $free\_out(z) = (free\_out_1(z), \ldots, free\_out_{|E|}(z))$.

Figure 5.3: Maps solutions from $P(k, G)$ to $Q(k, G)$.

zI.1 $\forall e_j \in E, r_j = free\_in_j, t_j = k - free\_out_j$

zI.2 $\forall c_j \in R_h, x_j = h, y_j = k + 1 - h$

zI.3 $\forall c_j \notin \bigcup_{i=1}^{k} R_i$, if $c_j \sqsubset c_k, c_k \in R_h$, for some h, $1 \leq h \leq k$, then

zI.3a x_j is one less than the smallest value of m such that $c_j \sqsubset c_i$ and $c_i \in R_m$.
$y_j = k - x_j$.

otherwise assume $c_j \in C(P_v)$,

zI.3b for all edges $e_i \in IN(P_v), x_j$ equals the smallest value of $free\_out_i$.
$y_j = k - x_j$.

zI.3c if P_v has no incident edge, then $x_j = k$ and $y_j = (k - x_j) = 0$.

Figure 5.4: Mapping from $Q(k, G)$ to $P(k, G)$.

in $P(k,G)$ and $Q(k,G)$, respectively. A solution in $P(k,G)$ is maximal if x_i and y_i for $c_i \in C(P)$ and r_j and t_j for $e_j \in E$ cannot be increased. We denote the set of maximal solutions in $P(k,G)$ as $\hat{P}(k,G)$. A solution in $Q(k,G)$ is maximal if no additional candidates can be allocated registers and the number of registers spilled along any call edge cannot be decreased. We denote maximal solutions in $Q(k,G)$ as $\hat{Q}(k,G)$. For both $\hat{P}(k,G)$ and $\hat{Q}(k,G)$ maximal solutions are not necessarily optimal with respect to their maximization functions *D.9* and *I.7*, respectively. We prove that $I(z)$ maps $z \in \hat{P}(k,G)$ to $I \in \hat{Q}(k,G)$, and $z(I)$ is its inverse.

Let maximal weighted solutions of $P(k,G)$ be denoted $P^*(k,G)$. Solutions to $P^*(k,G)$ maximize the objective function *D.9*, $\sum_{c_j \in C(P)} w_j * (x_j + y_j) + \sum_{e_j \in E} c_j * (r_j + t_j)$. Since w_j and c_j are positive values, $P^*(k,G) \subseteq \hat{P}(k,G)$. Let maximal weighted solutions (optimal solutions) of $Q(k,G)$ be denoted $Q^*(k,G)$. These solutions maximize *I.7*, $\sum_{c_j \in \bigcup_{i=1}^{k} R_i} w_j - \sum_{e_j \in E} s_j * (free\_out_j - free\_in_j)$. Since w_j is positive and the value of $(free\_out_j - free\_in_j) * s_j$ for $e_j \in E$ cannot be decreased in a maximal solution, $Q^*(k,G) \subseteq \hat{Q}(k,G)$. We prove that $I(z)$ maps solutions from $P^*(k,G)$ to $Q^*(k,G)$ and $z(I)$ is its inverse.

Mappings $I(z)$ and $z(I)$ can be computed in linear time with respect to the size of the call graph G.

5.1 Mapping Solutions between $P(k,G)$ and $Q(k,G)$

Theorem 1 *For any* $z \in P(k,G)$, $I(z) \in Q(k,G)$.

Prove that conditions *I.1* - *I.6* hold for $I(z) \in Q(k,G)$.

Prove (*I.1*) *for all* $e_j \in E$, $free\_in_j \leq free\_out_j$.

Proof: Let $e_j \in E$. By (*D.8*), (1) $r_j + t_j \leq k$. By (*Iz.3*) and (*Iz.4*), (2) $free\_in_j = r_j$ and $free\_out_j = k - t_j$, which is equivalent to (3) $t_j = k - free\_out_j$. Substituting r_j and t_j in (1) by r_j in (2) and t_j in (3), yields $free\_in_j + k - free\_out_j \leq k$ and, thus, $free\_in_j \leq free\_out_j$. □

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

Proof: By (*D.4*), for $e_j \in E$, $0 \leq r_j, t_j \leq k$. By (*Iz.3*), $free\_in_j = r_j$, so $0 \leq free\_in_j \leq k$.

Multiplying the constraint $0 \leq t_j \leq k$, by -1 yields $-k \leq -t_j \leq 0$. Adding k yields $0 \leq k - t_j \leq k$. By $(Iz.4)$, $free\_out_j = k - t_j$ and, thus, $0 \leq free\_out_j \leq k$. □

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

Proof: By assumption $e_j \in IN(P_v)$, $c_i \in C(P_v)$, and $R(z)$ assigns c_i to R_p. By $(D.5)$, $x_i + t_j \leq k$, which is equivalent to (1) $x_i \leq k - t_j$. By $(Iz.2)$ and $(Iz.4)$, (2) $p = x_i$ and (3) $free\_out_j = k - t_j$. By substituting (2) and (3) into (1), $p \leq free\_out_j$. □

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

Proof: Assume $e_j \in OUT(P_v)$, $c_i \in C(P_v)$, and $R(z)$ assigns c_i to R_p. By $(D.6)$, (1) $r_j + y_i \leq k$ and by $(Iz.2)$, $x_i + y_i = k + 1$, which is equivalent to (2) $y_i = k + 1 - x_i$. Substituting (2) in (1) yields $r_j + k + 1 - x_i \leq k$, which is equivalent to $r_j + 1 \leq x_i$ and, thus, $r_j < x_i$. By $(Iz.3)$, $free\_in_j = r_j$ and by $(Iz.2)$, $p = x_i$; therefore, $free\_in_j < p$. □

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

Proof: Let $e_i \in OUT(P_v)$ and $e_j \in IN(P_v)$. Then by $(D.7)$ $r_i + t_j \leq k$, which is equivalent to (1) $r_i \leq k - t_j$. By $(Iz.3)$ and $(Iz.4)$, (2) $free\_in_i = r_i$ and (3) $free\_out_j = k - t_j$. Substituting (2) and (3) into (1) yields $free\_in_i \leq free\_out_j$. □

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

Proof: Let $c_i \sqsubset c_j$, and assume $R(z)$ assigns c_i to R_p and c_j to R_q. Since c_j is assigned to R_q, by $(Iz.2)$, $x_j + y_j = k + 1$ and, therefore, (1) $y_j = k + 1 - x_j$. Since $c_i \sqsubset c_j$, by $(D.2)$ (2) $x_i + y_j \leq k$. Replacing y_j in (2) by y_j in (1) yields $x_i + k + 1 - x_j \leq k$, which is equivalent to $x_i + 1 \leq x_j$ and, thus, $x_i < x_j$. By $(Iz.2)$, $p = x_i$, $q = x_j$ and, therefore, $p < q$. □

Theorem 2 *For any* $I \in Q(k, G)$, $z(I) \in P(k, G)$.

Prove that conditions *D.1 - D.8* hold for $z(I) \in Q(k, G)$.

Prove $(D.1)$ *for* $c_j \in C(P)$, $0 \leq x_j, y_j \leq k$.

Proof: Let $c_j \in C(P)$. If $c_j \in R_p$, for any p, $1 \leq p \leq k$, then by $(zI.2)$, $x_j = p$ and $y_j = k + 1 - p$. Because $1 \leq p \leq k$, then $1 \leq x_j, y_j \leq k$. If $c_j \notin \bigcup_{i=1}^{k} R_i$, then by $(zI.3)$, $0 \leq x_j \leq k$. Since $y_j = k - x_j$, $0 \leq y_j \leq k$. □

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

Proof: Let $c_i \sqsubset c_j$.

(a) Assume $c_j \in R_h$, $1 \leq h \leq k$. By $(zI.2)$, $x_j = h$. If $c_i \in R_p$, $1 \leq p \leq k$, then by $(zI.2)$, $x_i = p$. Since $c_i \sqsubset c_j$, by $(I.6)$ $p < h$ and, thus, (1) $x_i < x_j$. By $(zI.2)$, (2) $x_j + y_j = k + 1$. Equations (1) and (2) imply $x_i + y_j \leq k$. If $c_i \notin \bigcup_{v=1}^{k} R_v$, then $x_i < x_j$ by $(zI.3a)$. Again, $x_i + y_j \leq k$.

(b) Otherwise, assume $c_j \notin \bigcup_{v=1}^{k} R_v$. Now, $x_j + y_j \leq k$ by $(zI.3)$. If we can show $x_i \leq x_j$, then $x_i + y_j \leq k$. Assume $x_j = h$, $0 \leq h \leq k$, and $c_j \in C(P_v)$.

(b.1) Suppose there exists an element $c_m \in R_q$, $1 \leq q \leq k$, such that $c_j \sqsubset c_m$. As $x_j = h$, by $(zI.3a)$ there must exist $c_n \in R_{h+1}$ such that $c_j \sqsubset c_n$. By assumption $c_i \sqsubset c_j$. If $c_i \in R_p$, then by $(I.6)$, $p < h + 1$. By $(zI.2)$, $x_i(= p) < x_n(= h + 1)$. Thus, $x_i \leq x_j(= h)$. If $c_i \notin \bigcup_{v=1}^{k} R_v$, then by $(zI.3a)$, $x_i < x_n(= h + 1)$. Thus, $x_i \leq x_j$.

(b.2) Suppose there does not exist an element $c_m \in R_q$ such that $c_j \sqsubset c_m$. Assume $c_i, c_j \in C(P_v)$.

(b.2a) If $IN(P_v)$ is not empty, then by $(zI.3b)$, for $e_g \in IN(P_v)$ such that $free\_out_g$ is minimum, (1) $x_j = free\_out_g$.

(b.2a1) If there is no $c_n \in \bigcup_{v=1}^{k} R_v$ such that $c_i \sqsubseteq c_n \sqsubset c_j$ (c_i can be c_n), then by $(zI.3b)$, $x_i = x_j$ and, thus, $x_i \leq x_j$.

(b.2a2) Otherwise, let $c_i \sqsubseteq c_n \sqsubset c_j$ and let $c_n \in R_p$, $1 \leq p \leq k$. By $(I.3)$, (2) $p \leq free\_out_g$ for every $e_g \in IN(P_v)$. Equations (1) and (2) imply that (3) $p \leq x_j$.

(b.2a2a) If c_i is c_n then by $(zI.2)$ $x_i = p$ and then by (3), $x_i(= p) \leq x_j$.

(b.2a2b) Otherwise, $c_i \sqsubset c_n$. If element $c_i \in R_q$, $1 \leq q \leq k$, then by $(I.6)$, $q < p$. By $(zI.2)$, $x_i = q$. Since $q < p$ and by (3), $p \leq x_j$, then $x_i < x_j$. Otherwise, $c_i \notin \bigcup_{v=1}^{k} R_v$. By $(zI.3a)$, $x_i < p$ and by (3), $x_i < x_j$.

(b.2b) Otherwise, $IN(P_v)$ is empty. By $(zI.3b)$ $x_j = k$. Since $0 \leq x_i \leq k$, $x_i \leq x_j$. □

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

Proof: Assume $c_j \in R_p$, $1 \leq p \leq k$. Then by $(zI.2)$, $x_j = p$ and $y_j = k + 1 - p$. So, $x_j + y_j \leq k + 1$. Assume $c_j \notin \bigcup_{v=1}^{k} R_v$. By $(zI.3)$, $y_j = k - x_j$. Thus, $x_j + y_j \leq k$. □

Prove $(D.4)$ *for* $e_j \in E$, $0 \leq r_j, t_j \leq k$.

Proof: For $e_j \in E$, $0 \leq free\_in_j, free\_out_j \leq k$. By $(zI.1)$, $r_j = free\_in_j$ and, thus, $0 \leq r_j \leq k$. Also, by $(zI.1)$, $free\_out_j = k - t_j$. So $0 \leq k - t_j \leq k$. Subtracting k, we have $-k \leq -t_j \leq 0$. Multiplying by -1 yields $k \geq t_j \geq 0$. □

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

Proof:

(a) Assume $c_i \in R_q$, $1 \leq q \leq k$, and $c_i \in C(P_v)$. Let $e_j \in IN(P_v)$. By $(I.3)$, (1) $q \leq free\_out_j$. By $(zI.2)$, (2) $x_i = q$, and by $(zI.1)$, $t_j = k - free\_out_j$, which is equivalent to (3) $free\_out_j = k - t_j$. Substituting (2) and (3) in (1) yields, $x_i \leq k - t_j$ and, thus, $x_i + t_j \leq k$.

(b) Assume $c_i \notin \bigcup_{v=1}^{k} R_v$.

(b.1) If there exists a $c_m \in R_q$ such that $c_i \sqsubset c_m$, then by $(I.3)$ $q \leq free\_out_j$ for $e_j \in IN(P_v)$. Since $c_i \sqsubset c_m$, by $(zI.3a)$, $x_i < q$ and, therefore, (1) $x_i < free\_out_j$. By $(zI.1)$, (2) $free\_out_j = k - t_j$. Replacing $free\_out_j$ in (1) by $free\_out_j$ in (2) yields $x_i < k - t_j$ and, thus, $x_i + t_j < k$.

(b.2) Assume $c_i \in C(P_v)$. If there does not exist a $c_m \in \bigcup_{v=1}^{k} R_v$ such that $c_i \sqsubset c_m$, then by $(zI.3b)$, $x_i = free\_out_g$ for $e_g \in IN(P_v)$ such that $free\_out_g$ is a minimum. Therefore, for $e_j \in IN(P_v)$, (1) $x_i \leq free\_out_j$. By $(zI.1)$, (2) $free\_out_j = k - t_j$. Replacing $free\_out_j$ in (1) by $free\_out_j$ in (2) yields, $x_i \leq k - t_j$ and, thus, $x_i + t_j \leq k$. □

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

Proof:

(a) Assume $c_j \in R_q$ and $c_j \in C(P_v)$. Let $e_i \in OUT(P_v)$. By $(I.4)$, $q > free\_in_i$, and by $(zI.2)$, $x_j = q$. Thus, $x_j > free\_in_i$. By $(zI.1)$, $r_i = free\_in_i$ and, therefore, (1) $x_j > r_i$. By $(zI.2)$, (2) $x_j + y_j = k + 1$. Solving for x_j in (2) and substituting this equation for x_j in (1) yields $k + 1 - y_j > r_i$ and, thus, $k \geq r_i + y_j$.

(b) Assume $c_j \notin \bigcup_{v=1}^{k} R_v$ and $x(I)$ assigns $x_j = h$, $0 \leq h \leq k$. By $(zI.3)$, $y_j = k - h$.

(b.1) Assume there exists $c_n \in R_p$, $1 \leq p \leq k$, such that $c_j \sqsubset c_n$. By $(zI.3a)$, there must exist $c_m \in R_{h+1}$, as $x_j = h$. By $(I.4)$, (1) $h + 1 > free\_in_i$. Since $y_j = k - h$, $h = k - y_j$ and, thus, (2) $h + 1 = k - y_j + 1$. Substituting (2) into (1) yields $k - y_j + 1 > free\_in_i$. By $(zI.1)$, $r_i = free\_in_i$. Thus, $k - y_j + 1 > r_i$, which is equivalent to $k + 1 > r_i + y_j$ and, thus, $k \geq r_i + y_j$.

(b.2) Assume there does not exist $c_n \in R_p$ such that $c_j \sqsubset c_n$. Let $c_j \in C(P_v)$.

(b.2a) If $IN(P_v)$ is not empty, then by $(zI.3b)$, (1) $x_j = free\_out_g$ for $e_g \in IN(P_v)$ such that $free\_out_g$ is minimum. By $(I.1)$, $free\_out_g \geq free\_in_i$. Since by $(zI.1)$, $r_i = free\_in_i$, then (2) $free\_out_g \geq r_i$. Equations (1) and (2) imply (3) $x_j \geq r_i$. Since by assumption $x_j = h$, and $y_j = k - h$, (4) $x_j = k - y_j$. Equations (3) and (4) imply $k - y_j \geq r_i$. Thus, $k \geq r_i + y_j$.

(b.2b) If $IN(P_v)$ is empty, then $x_j = k$, and $y_j = 0$. Since $0 \leq free\_in_i \leq k$, then $free\_in_i + y_j \leq k$. By $(zI.1)$, $r_i = free\_in_i$. Thus, $r_i + y_j \leq k$. □

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

Proof: Assume $e_j \in IN(P_v)$ and $e_i \in OUT(P_v)$. By $(I.5)$, (1) $free\_in_i \leq free\_out_j$. By $(zI.1)$, (2) $r_i = free\_in_i$ and (3) $k - t_j = free\_out_j$. Substituting (2) and (3) in (1), yields $r_i \leq k - t_j$ and, thus, $r_i + t_j \leq k$. □

Prove *(D.8) for* $e_j \in E$, $r_j + t_j \leq k$.

Proof: Let $e_j \in E$. By $(I.1)$, (1) $free\_in_j \leq free\_out_j$. By $(zI.1)$, (2) $free\_in_j = r_j$ and (3) $free\_out_j = k - t_j$. Substituting (2) and (3) in (1) yields $r_j \leq k - t_j$ and, thus, $r_j + t_j \leq k$. □

5.2 Maximal Solutions in $P(k, G)$ and $Q(k, G)$

A solution in $P(k, G)$ is maximal if for $c_i \in C(P)$ and $e_j \in E$, neither x_i, y_i, r_j nor t_j can be increased. Let $\hat{P}(k, G)$ be the set of maximal solutions of $P(k, G)$. $\hat{P}(k, G) \subseteq P(k, G)$. Some solutions in $\hat{P}(k, G)$ may not be optimal, as they do not necessarily maximize the objective function $D.7$.

Let $\hat{Q}(k,G)$ represent the set of maximal allocations in $Q(k,G)$. In $\hat{Q}(k,G)$, additional candidates cannot be allocated registers given both the current set of candidates already allocated registers and the registers spilled along the edges in the call graph. Also, in a maximal allocation, fewer registers cannot be spilled along an edge given both the registers spilled along other edges in the call graph and the candidates allocated registers in each procedure. Some solutions in $\hat{Q}(k,G)$ may not be maximum weighted allocations.

More formally, a solution I in $Q(k,G)$ is maximal if:

- $\forall c_j \notin \bigcup_{i=1}^{k} R_i, c_j$ cannot be allocated to any $R_h, 1 \leq h \leq k$.
- $free\_out_i$ cannot be decreased and $free\_in_i$ cannot be increased for any edge $e_i \in E$.

We now prove a few properties of $\hat{P}(k,G)$ before proving that if $z \in \hat{P}(k,G)$ then $I(z)$ is a bijection onto $Q(k,G)$ and $z(I)$, for $I \in \hat{Q}(k,G)$, is its inverse.

Theorem 3 *Let $z \in \hat{P}(k,G)$. Let $c_i \in C(P)$. Then $x_i + y_i = k$ or $x_i + y_i = k+1$.*

Proof: Let $c_i \in C(P_v)$. Assume (1) $x_i + y_i < k$. Since z is maximal, x_i and y_i cannot be increased. Therefore, there exist constraints on x_i and y_i. For $c_m \sqsubset c_i$, $x_m + y_i = k$ or for edge $e_g \in OUT(P_v)$, $r_g + y_i = k$. Let q be the value of x_m or r_g constraining y_i. Hence, (2) $q + y_i = k$. There is also a dual variable s for an edge $e_f \in IN(P_v)$ or candidate c_n, $c_i \sqsubset c_n$, that constraints x_i, such that (3) $x_i + s = k$. We also have the constraint (4) $q + s \leq k$.

Solving for y_i in (2) and replacing y_i in (1) yields (5) $x_i + k - q < k$. Solving for x_i in (3) and replacing x_i in (5) yields, $k - s + k - q < k$ and, thus, $k < s + q$. This leads to a contradiction, since by (4), $q + s \leq k$. □

Theorem 4 *Let $z \in \hat{P}(k,G)$, $c_i, c_j \in C(P)$. If $c_i \sqsubset c_j$, then $x_i \leq x_j$.*

Proof: Since z is maximal, by Theorem 3 either $x_j + y_j = k$, which is equivalent to (1) $y_j = k - x_j$, or $x_j + y_j = k+1$, which is equivalent to (2) $y_j = k + 1 - x_j$. Since $c_i \sqsubset c_j$, by *(D.2)*, (3) $x_i + y_j \leq k$. Substituting (1) into (3) yields $x_i + k - x_j \leq k$ and, thus, $x_i \leq x_j$. Substituting (2) into (3) yields $x_i + k + 1 - x_j \leq k$, which is equivalent to $x_i + 1 \leq x_j$ and, thus, $x_i < x_j$. □

Corollary 1 *Let $z \in \hat{P}(k, G)$, $c_i, c_j \in C(P)$. If $c_i \sqsubset c_j$ and $c_j \in R_v(z)$ for some $1 \leq v \leq k$, then $x_i < x_j$.*

Proof: If $c_j \in \bigcup_{v=1}^{k} R_v(z)$, then $x_j + y_j = k + 1$. From the proof of Theorem 4, $x_i < x_j$. □

Theorem 5 *Assume $c_i \in C(P_v)$ and $c_i \notin R_h(z)$ for some $1 \leq h \leq k$. Let $x_i = h$. If $z \in \hat{P}(k, G)$, then*

1. *if there exists a $c_m \in R_v(z)$ from some $1 \leq v \leq k$, $c_i \sqsubset c_m$, then there exists a $c_q \in R_{h+1}(z)$, $c_i \sqsubset c_q$.*
2. *Otherwise, assume there does not exist a $c_m \in R_v(z)$, $c_i \sqsubset c_m$. If $IN(P_v)$ is not empty, then $x_i = k - t_m$ for some $e_m \in IN(P_v)$.*
3. *Otherwise, if there does not exist an $e_j \in IN(P_v)$, $x_i = k$.*

Proof of (1)

Let $x_i = h$. By Theorem 4, for $c_u \sqsubset c_i$, $x_u \leq x_i$, and for $c_i \sqsubset c_u$, $x_i \leq x_u$. By assumption, $c_m \in R_v(z)$ and $c_i \sqsubset c_m$. By Corollary 1, $x_i < x_m$. Let c_j be the candidate such that $x_u \leq h$ for all $c_u \sqsubseteq c_j$ and $x_u > h$ for all $c_j \sqsubset c_u$. Since $x_u \leq h$ for all $c_u \sqsubseteq c_j$, then $x_j = x_i = h$.

(1a) Assume for all c_u such that $c_j \sqsubset c_u$, we have (1) $x_u \geq h + 2$. By Theorem 3, $x_u + y_u = k$ or $k + 1$. If $x_u + y_u = k$, then $x_u = k - y_u$, and by (1), $k - y_u \geq h + 2$ and, thus, $k - h - 2 \geq y_u$. If $x_u + y_u = k + 1$, then $k - h - 1 \geq y_u$. In both cases, $x_j (= h) + y_u < k$. Thus, z is not maximal, as x_j can be increased by 1 by D.2.

(1b) Otherwise, assume for each c_u, such that $c_j \sqsubset c_u$ and $x_u = h + 1$ that $c_u \notin R_{h+1}(z)$. Since $x_u + y_u = k$, $y_u = k - h - 1$. Again, x_j can be increased. Therefore, there must be an element c_v, $c_i \sqsubseteq c_j \sqsubset c_v$ such that $c_v \in R_{h+1}(z)$.

Proof of (2)

Let $x_i = x_j = h$. Let c_j be the candidate such that for all $x_u \leq h$, $c_u \sqsubseteq c_j$. Since $c_m \notin \bigcup_{v=1}^{k} R_v(z)$ for all $c_j \sqsubset c_m$, then from the proof of (1), there is no candidate greater than c_j in the partial order. The only constraint on x_j is t_g for $e_g \in IN(P_v)$.

By assumption, $IN(P_v)$ is not empty. Since $x_j = h$ and z is maximal, there must be an edge $e_g \in IN(P_v)$ such that $t_g = k - h$ to constrain the value of x_j. Thus, $x_i = x_j = k - t_g$.

Proof of (3)

As in the proof of (2) above, let c_j be the candidate such that for all $x_u \leq h$, $c_u \sqsubseteq c_j$. Again there can be no c_m such that $c_j \sqsubset c_m$. If $x_j = h$, $h < k$, then $y_j > 0$, as $x_j + y_j = k$. If there does not exist an incident edge on P_v, then x_j can be increased. Since z is maximal, $x_j = k$ and $y_j = (k - x_j) = 0$. □

Theorem 6 *Assume $c_i \in C(P_v)$, $x_i = h$, $c_i \notin R_h(z)$, and $0 \leq h \leq k$. If $z \in \hat{P}(k, G)$, then*

1. *if there exists a candidate $c_j \in C(P_v)$, such that $c_j \sqsubset c_i$ and $c_j \in \bigcup_{v=1}^{k} R_v(z)$, then there is a candidate $c_m \sqsubset c_i$ such that $c_m \in R_h(z)$*

2. *Otherwise, assume there is no candidate $c_j \sqsubset c_i$ such that $c_j \in \bigcup_{v=1}^{k} R_v(z)$. If $OUT(P_v)$ is not empty, then $r_g = x_i$ for some $e_g \in OUT(P_v)$.*

3. *Otherwise, if there does not exist an $e_g \in OUT(P_v)$, then $x_i = 0$.*

Proof of (1)

Let $x_i = h$, and $c_m \sqsubset c_i$. Let c_j be the candidate such that for all $x_u < h$, $c_u \sqsubset c_j$ and $x_u \geq h$ for all $c_j \sqsubseteq c_u$. Thus, $x_j = h$, and $c_j \sqsubseteq c_i$.

If there exists a candidate $c_m \in R_h(z)$, then (1) is true. Otherwise, assume there does not exist $c_m \in R_h(z)$. Then since $x_j = h$, by *(Iz.2)*, $c_j \notin R_h(z)$. As z is maximal, $y_j = y_i = k - x_j (= h)$. If for all c_u such that $c_u \sqsubset c_j$, we have $x_u < h$, then y_j can be increased since $y_j = k - h$ and $x_u + y_j \leq k$ by D.2. But this is impossible since z is maximal.

Proof of (2)

Let $x_i = h$. As in the proof of (1), let c_j be the candidate such that for all $x_u \geq h$, $c_j \sqsubseteq c_u$. Since by assumption there does not exist a candidate $c_m \in \bigcup_{v=1}^{k} R_v(z)$, such that $c_m \sqsubset c_i$, then from the proof of (1), there cannot be a candidate c_u such that $x_u < h$ and, thus, there is no candidate c_u such that $c_u \sqsubset c_j$. By assumption $OUT(P_v)$ is not empty. Since $c_j \notin \bigcup_{v=1}^{k} R_v(z)$, there must be a candidate $e_g \in OUT(P_v)$ such that r_g constrains the value of y_j to $k - h$. Thus $r_g = x_j = x_i = h$.

Proof of (3)

As in the proof of (2), let c_j be the candidate such that for all $x_u \geq h$, $c_j \sqsubseteq c_u$. For $x_j = h$, $h > 0$, then $y_j < k$. If there is no outgoing edge in P_v, then the value of y_j can be increased to k. But, then z is not maximal. Hence $x_i = x_j = 0$. □

Theorem 7 *If* $I \in \hat{Q}(k, G)$, $z(I) \in \hat{P}(k, G)$.

Proof: Let $z' = z(I)$ for $I \in \hat{Q}(k, G)$. Assume candidate c_i is assigned to $R_h, 1 \leq h \leq k$, in I. In z', $x_i = h$ and $y_i = k + 1 - h$. For candidates not assigned a register in I, $x_i + y_i = k$ in z'. $I(z')$, in turn, assigns c_i to R_h if $x_i = h$ and $y_i = k + 1 - h$. In addition, in z' $r_j = free\_in_j$ and $t_j = k - free\_out_j$ for $e_j \in E$. $I(z')$ maps r_j to $free\_in_j$ and $k - t_j$ to $free\_out_j$. Thus, I is equivalent to $I(z')$.

Assume $z' = z(I)$ is not maximal in $\hat{P}(k, G)$. Then we can increase $x_i + y_i$ for $c_i \in C(P)$, or either r_j or t_j for $e_j \in E$. If we can increase $x_i + y_i$ then its value increases to $k + 1$. However, then I would not be maximal, since c_i would be allocated a register in $I(z')$ and, therefore, can be allocated a register in I. Similarly, since $r_j = free\_in_j$, if the value of r_j can be increased then $free\_in_j$ in I can also be increased. If t_j can be increased then $free\_out_j$ can also be decreased since $free\_out_j = k - t_j$. In either case, the values of $free\_in_j$ and $free\_out_j$ are then not part of a maximal solution in $I(z')$ and, therefore, I.

Theorem 8 *If* $z \in \hat{P}(k, G)$, *then* $I(z) \in \hat{Q}(k, G)$.

Proof:

Let $z \in \hat{P}(k, G)$. For $I(z) \in Q(k, G)$, we prove the following: (1) if $c_i \notin \bigcup_{v=1}^{k} R_v(z)$, then c_i cannot be added to $R_h(z)$, $1 \leq h \leq k$; (2) the value of $free\_in_i$ cannot be increased for $e_i \in E$; and (3) the value of $free\_out_i$ cannot be decreased for $e_i \in E$.

(1) Let $c_i \in C(P_v)$ and $c_i \notin \bigcup_{v=1}^{k} R_v$. Assume $x_i = h$, $0 \leq h \leq k$.

(1a) We first prove that c_i cannot be assigned to $R_p, p > h$. By Theorem 5, if there exists an element $c_m \in \bigcup_{v=1}^{k} R_v(z)$, such that $c_i \sqsubset c_m$, then there is an element c_n, $c_i \sqsubset c_n$ such that $c_n \in R_{h+1}(z)$. By *(I.6)*, $p < h + 1$, so c_i cannot be assigned to $R_p, p > h$. Otherwise, if $IN(P_v)$ is non-empty, then by Theorem 5, there exists an

edge $e_g \in IN(P_v)$ such that $x_i = (k - t_g) = h$. By $(zI.1)$, $free\_out_g = (k - t_g) = h$. If c_i is added to $R_p, p > h$, then $p > free\_out_g$, but by $(I.3)$, $p \leq free\_out_g$. Otherwise, there is no incident edge on P_v. Again, by Theorem 5, $x_i = h = k$. Since there is no $R_p, p > k$, then c_i cannot be assigned $R_p, p > h$.

(1b) We now prove that c_i cannot be added to $R_p, p \leq h$. If there exists a candidate $c_n \in \bigcup_{v=1}^{k} R_v(z)$, such that $c_n \sqsubset c_i$, then by Theorem 6, there is a candidate c_j, $c_j \sqsubset c_i$, such that $c_j \in R_h(z)$. By $(I.6)$, if $c_i \in R_p$, then $p > h$, which leads to a contradiction. Otherwise, if there is no candidate $c_n \in \bigcup_{v=1}^{k} R_v(z)$, $c_n \sqsubset c_i$, then by Theorem 6, if there exists an $e_g \in OUT(P_v)$, then $r_g = x_i = h$. By $(I.4)$, if $c_i \in R_p(z)$, $e_g \in OUT(P_v)$, then $p > free\_in_g$, By Iz.3, $free\_in_g = r_g = h$ and, therefore, c_i cannot be added to register R_p, $p \leq h$. If there is no outgoing edge from P_v, then by Theorem 6, $x_i = h = 0$. Again, c_i cannot be added to a group R_p, such that $p \leq h$.

(2) We now show that for $e_i \in E$, $free\_in_i$ is maximal if z is maximal. Let $r_i = h$, $e_i \in OUT(P_v)$. $I(z)$ maps the value of r_i to $free\_in_i$ and $k - t_i$ to $free\_out_i$. There are three constraints on r_i in z: (a) $r_i \leq k - t_i$; (b) if $e_g \in IN(P_v)$, then $r_i \leq k - t_g$; and, (c) if $c_j \in C(P_v)$, then $r_i \leq k - y_j$. Since z is maximal, r_i must be equal to one or more of these upper bounds. $I(z)$ maps these constraints to the three constraints on $free\_in_i$ in $Q(k, G)$.

Case(a) Assume $r_i = k - t_i$. By $I(z)$, $free\_in_i = r_i$ and $free\_out_i = k - t_i$. Thus, $free\_in_i = free\_out_i$. By $(I.1)$, $free\_out_i$ is an upper bound on $free\_in_i$ and, thus, $free\_in_i$ is maximal.

Case(b) Assume $r_i = k - t_g$ for $e_g \in IN(P_v)$. Then by $(Iz.3)$ and $(Iz.4)$, $free\_in_i = free\_out_j$. By $(I.5)$, $free\_out_j$ is an upper bound on $free\_in_i$ and, thus, $free\_in_i$ is maximal.

Case(c) Assume $r_i(= h) + y_j = k$ for $c_j \in C(P_v)$, $e_i \in OUT(P_v)$.

(1) Assume $c_j \in \bigcup_{v=1}^{k} R_v(z)$. Then (1) $x_j + y_j = k + 1$. Since $r_i(= h) = k - y_j$, then (2) $y_j = k - h$. Equations (1) and (2) imply $x_j = h + 1$, and $c_j \in R_{h+1}(z)$.

Since $r_i = h$, then $free\_in_i = h$ by $(Iz.3)$. By $(I.4)$, $h+1 > free\_in_i$ and, thus, $free\_in_i$ is maximal.

(2) Otherwise, assume $c_j \notin \bigcup_{v=1}^{k} R_v(z)$. Let $x_j = v$. One of three cases hold.

(a) If there exists a candidate $c_m \in \bigcup_{s=1}^{k} R_s(z)$ such that $c_j \sqsubset c_m$, then by Theorem 5, there is a candidate $c_n \in R_{v+1}(z)$ and, thus, $x_n = v+1$. Since $c_n \in R_{v+1}(z)$, $y_n = k+1-x_n = k-v$. Since $x_j = v$ and c_i is not mapped to a register $(x_j + y_j = k)$, then $y_j = k-v$. Thus, $y_j = y_n$. Since there is already a constraint, $r_i + y_n \leq k$ (D.6), the constraint $r_i + y_j \leq k$ does not further limit the value of r_i.

(b) If there does not exist a candidate $c_m \in \bigcup_{s=1}^{k} R_s(z)$ such that $c_j \sqsubset c_m$, then if $IN(P_v)$ is not empty, by Theorem 5, $x_j = k - t_g$ for some $e_g \in IN(P_v)$. Then $y_j = k - x_j = t_g$. As there is already a constraint $r_i + t_g \leq k$ (D.7), the constraint $r_i + y_j \leq k$ does not further limit the value of r_i.

(c) If $IN(P_v)$ is empty, then $x_j = k$ and, since z is maximal and $c_j \notin \bigcup_{s=1}^{k} R_s(z)$, $y_j = 0$. The constraint $r_i + y_j \leq k$ does not further constrain the value of r_i. □

(3) We now show that for $e_i \in E$, $free\_out_i$ cannot be decreased, if z is maximal. Let $t_i = k - h$, $e_i \in IN(P_v)$. $I(z)$ maps the value of r_i to $free\_in_i$ and $k - t_i$ to $free\_out_i$. There are three constraints on t_i in z: (a) $t_i \leq k - r_i$; (b) if $e_j \in OUT(P_v)$, then $t_i \leq k - r_j$; and, (c) if $c_j \in C(P_v)$, then $t_i \leq k - x_j$. Since z is maximal, t_i must be equal to one or more of these upper bounds. $I(z)$ maps these constraints to the three constraints on $free\_out_i$ in $Q(k, G)$.

Case(a) Let $t_i = k - r_i$. Equivalently, $k - t_i = r_i$. Since z is maximal, t_i is maximal. Thus, $k - t_i$ is minimal. By $(Iz.3)$ and $(Iz.4)$, $free\_out_i = k - t_i$ and $free\_in_i = r_i$ and, thus, $free\_in_i = free\_out_i$. By $(I.1)$, $free\_in_i$ is a lower bound on $free\_out_i$ and, thus, $free\_out_i$ is minimal.

Case(b) Assume $r_j = k - t_i$ for $e_j \in OUT(P_v)$. Then by $(Iz.3)$ and $(Iz.4)$, $free\_in_j =$

$free\_out_i$. By $(I.5)$, $free\_in_j$ is a lower bound on $free\_out_j$ and, thus, $free\_out_j$ is minimal.

Case(c) Assume (c1) $x_j = k - t_i$ for $c_j \in C(P_v)$.

(1) Assume $c_j \in R_h(z)$, $1 \leq h \leq k$. Since $c_j \in R_h(z)$, (c2) $x_j = h$. By $(zI.3)$, (c3) $free\_out_i = k - t_i$. Substituting (c2) and (c3) into (c1) yields $h = free\_out_i$. By $(I.3)$, h is a lower bound on $free\_out_i$ and, therefore, $free\_out_i$ is minimal.

(2) Otherwise, assume $c_j \notin R_h(z)$, $1 \leq h \leq k$, $x_j = h$. One of the following three cases hold.

(a) Let $c_m \in \bigcup_{v=1}^{k} R_v(z)$ and $c_m \sqsubset c_j$. By Theorem 6, there exists a $c_n \in R_h(z)$ such that $c_n \sqsubset c_i$. By $(Iz.2)$, $x_n = h$. As there exists constraints $x_n(= h) + t_i \leq k$ and $x_j(= h) + t_i \leq k$ (D.5), x_j does not further constraint the value of t_i.

(b) Assume there does not exist a $c_m \in \bigcup_{v=1}^{k} R_v(z)$ such that $c_m \sqsubset c_j$. If $IN(P_v)$ is not empty, then by Theorem 6, $x_j = r_g$ for some $e_g \in IN(P_v)$. Since by $(D.7)$ there is already the constraint $r_g + t_i \leq k$, the constraint $x_j + t_i \leq k$ $(D.5)$, does not further limit the value of t_i.

(c) If $IN(P_v)$ is empty, then by Theorem 6, $x_j = 0$. Since the constraint between c_j and $e_i \in IN(P_v)$ is $x_j + t_i \leq k$, x_j does not limit the value of t_i. □

Theorem 9 *If $I \in \hat{Q}(k, G), z \in \hat{P}(k, G)$, and $I(z) = I$, then $z = z(I)$.*

Let $e_j \in E$. For r_j and t_j, $I(z)$ maps r_j to $free\_in_j$ and $k - t_j$ to $free\_out_j$. Mapping $z(I)$ simply inverts the mapping. Thus, $z = z(I)$.

Assume $c_i \in R_h(z)$, $1 \leq h \leq k$, $x_i = h$, and $y_i = k + 1 - h$, then $I(z) = I$ maps c_i to R_h. If $c_i \in R_h$, then $z(I)$ assigns $x_i = h$ and $y_i = k + 1 - h$. Thus, $z = z(I)$.

Otherwise, $c_i \notin R_h(z)$. Let $x_i = h$ and, therefore, $y_i = k - h$ since z is maximal.

(a) Assume $c_i \sqsubset c_m$ such that $c_m \in \bigcup_{v=1}^{k} R_v(z)$. By Theorem 5, there exists a c_j, $c_i \sqsubset c_j$, such that $I(z)$ maps c_j to R_{h+1} $(x_j = h+1)$. Therefore, by Corollary 1, there cannot

exist a $c_m \in R_h(z)$ such that $c_i \sqsubset c_m$, since by assumption $x_i = h$. Since $c_i \sqsubset c_j$ and $c_j \in R_{h+1}$, by $(zI.3a)$, $z(I)$ maps x_i to h and y_i to $k - h$.

(b) Otherwise, there does not exist a $c_i \sqsubset c_m$ such that $c_m \in \bigcup_{i=1}^{k} R_v(z)$. Let $c_i \in C(P_v)$.

(b.1) If $IN(P_v)$ is not empty, then by Theorem 5, (1) $x_i = k - t_g$ for some $e_g \in IN(P_v)$. As z is maximal, (2) $y_i = k - x_i = t_g$.

Since c_i has a constraint $x_i + t_p \leq k$ for every $e_p \in IN(P_v)$, and $x_i + t_g = k$, this implies $t_g \geq t_p$ for every $e_p \in IN(P_v)$. Thus, t_g is maximal. Since t_g is maximal, and $I(z)$ maps $k - t_g$ to $free\_out_g$ in I, then $free\_out_g$ is minimal for all edges in $IN(P_v)$. Since $free\_out_g$ is minimal, by $(zI.3b)$, x_i is assigned $free\_out_g = k - t_g$, and y_i is assigned t_g. Thus, x_i and y_i match their values in (1) and (2).

(b.2) Otherwise, assume P_v has no incident edges. In this case, by Theorem 5, $x_i = k$ and $y_i = 0$. By $(zI.3c)$, $z(I)$ assigns k to x_i and 0 to y_i. □

Theorem 10 *If $z \in \hat{P}(k, G), I \in \hat{Q}(k, G)$, and $z(I) = z$, then $I = I(z)$.*

Assume $I \neq I(z)$. Let $I' = I(z)$. By Theorem 10, $z(I') = z$. Since $z(I) = z$ by assumption, $z(I') = z = z(I)$. By mapping zI.1 - zI.3, if $I \neq I'$, then $z(I) \neq z(I')$. Hence, $I = I'$ and $I = I(z)$. □

Corollary 2 *For $I \in \hat{Q}(k, G)$, $z(I)$ is a bijection onto $\hat{P}(k, G)$ and $I(z)$ is its inverse.*

This follows from Theorems 9 and 10. □

5.3 Mapping solutions between $P^*(k, G)$ and $Q^*(k, G)$

Corollary 3 *Let $z \in P^*(k, G)$ and $I \in Q^*(k, G)$. $I(z)$ is a bijection onto $Q^*(k, G)$ and $z(I)$ is its inverse.*

Proof: Since $w_i > 0$ for $c_i \in C(P)$ and $s_j > 0$ for $e_j \in E$, $P^*(k, G) \subseteq \hat{P}(k, G)$. Since $w_j > 0$ and $-s_j * (free\_out_j - free\_in_j)$ cannot be decreased in a maximal solution, $Q^*(k, G) \subseteq \hat{Q}(k, G)$. For $z \in P^*(k, G)$, function $(D.9)$:

$$\sum_{c_i \in C(P)} w_i * (x_i + y_i) + \sum_{e_j \in E} s_j * (r_j + y_j) \tag{1}$$

is maximized. Since $\sum_{c_i \in C(P)} w_i * k$ and $\sum_{e_j \in E} c_j * k$ are constants, equation (2) below is maximal if and only if equation (1) is maximal.

$$\begin{aligned} & \sum_{c_i \in C(P)} w_i * (x_i + y_i) - \sum_{c_i \in C(P)} w_i * k \\ & + \sum_{e_j \in E} s_j * (r_j + t_j) - \sum_{e_j \in E} s_j * k \end{aligned} \tag{2}$$

$$\begin{aligned} = \; & \sum_{c_i \in C(P)} w_i * (x_i + y_i - k) \\ & + \sum_{e_j \in E} s_j * (r_j + t_j - k) \end{aligned} \tag{3}$$

$$= \sum_{c_i \in \bigcup_{v=1}^{k} R_v} w_i + \sum_{e_j \in E} s_j * (r_j + t_j - k) \tag{4}$$

$$= \sum_{c_i \in \bigcup_{v=1}^{k} R_v} w_i - \sum_{e_j \in E} s_j * (k - t_j - r_j) \tag{5}$$

$$= \sum_{c_i \in \bigcup_{v=1}^{k} R_v} w_i - \sum_{e_j \in E} s_j * (free\_out_j - free\_in_j) \tag{6}$$

Equation (4) follows from (3) since for $z \in \hat{P}(k, G)$, $x_i + y_i = k+1$ for $c_i \in \bigcup_{v=1}^{k} R_v$, and $x_i + y_i = k$ for $c_i \notin \bigcup_{v=1}^{k} R_v$. Equation (5) follows from (4), since $r_j + t_j - k$ is equivalent to $-(k - t_j - r_j)$. Equation (6) follows from (5) since $k - t_j$ equals $free\_out_j$ and r_j equals $free\_in_j$ for $e_j \in E$. Equation (6) is the maximization function for interprocedural register allocation with spilling (I.7).

By Corollary 2, $I(z)$ is a bijection from maximal solutions in $z \in \hat{P}(k, G)$ to $\hat{Q}(k, G)$, and $z(I)$ is its inverse. For each $z \in \hat{P}(k, G)$, we can compute a weight using $D.9$ which differs from the weight of $I \in I(z)$ $(I.7)$ by only a constant that is independent of z, as shown above. Thus, $I(z)$ restricted to $z \in P^*(k, G)$ is a bijection onto $I \in Q^*(k, G)$, and $z(I)$ is its inverse function. □

Chapter 6

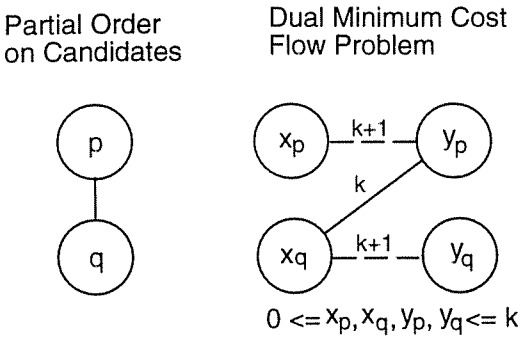
Mapping to Minimum Cost Flow Problem

In this section we present a transformation of the dual minimum cost flow problem for finding a save-free allocation from Section 4.1 to a minimum cost flow problem. For ease of reference, the dual minimum cost flow problem is shown in Figure 6.1. The transformation from the dual minimum cost flow problem for finding an interprocedural register allocation with spills is a simple extension and will be discussed later.

In a minimum cost flow problem, let N be a set of nodes and A be a set of arcs, which are pairs of nodes from N. Let f_{ij} represent a flow from node $i \in N$ to node $j \in N$ along arc $(i, j) \in A$, and let a_{ij} be the cost of the flow f_{ij} [Ber91]. The minimum cost flow problem

Dual Variables
x_j, y_j for j such that $c_j \in S$
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 $\sum_{c_j \in S} w_j * (x_j + y_j)$.

Figure 6.1: Dual minimum cost flow problem for finding a save-free interprocedural register allocation.



Dual Variables
x_p, y_p, x_q, y_q

Constraints
1) $0 \leq x_p, y_p, x_q, y_q \leq k$
2) $x_p + y_p \leq k + 1$
3) $x_q + y_q \leq k + 1$
4) $x_q + y_p \leq k$

Objective Function
maximize $w_p * x_p + w_q * x_q +$
$w_p * y_p + w_q * y_q$.

Figure 6.2: Partial order and dual minimum cost flow problem based on partial order.

minimizes $\sum_{(i,j)\in A} a_{ij} f_{ij}$ subject to

$$\sum_{\{j|(i,j)\in A\}} f_{ij} - \sum_{\{j|(j,i)\in A\}} f_{ji} = g_i, \forall i \in N.$$

The flow on arcs incident on node i minus the flow on outgoing arcs from node i is a constant, g_i. The value g_i is called the *divergence* of node i.

Figure 6.2 presents a partial order on the candidates of a call graph, in which $q \sqsubset p$, a graph of the dual minimum cost flow problem for the partial order, in which there are two dual variables for each candidate, and a formal definition of the dual problem. The relationship between a minimum cost flow problem and a dual minimum cost flow problem can be represented by a table. In Table 6.1 the dual variables of the dual minimum cost flow problem shown in Figure 6.2 appear in column 1. Each dual variable corresponds to a node in the minimum cost flow problem. The objective coefficients of the dual variables are shown in the last column. The objective function is the dot product of the entries in the first and last column. The remaining columns show the constraints on the dual variables. For example, column 2 represents the constraint $x_p + y_p \leq k + 1$ (there is a 1 in rows x_p and y_p; empty blocks represent the value 0).

The minimum cost flow problem's primal variables, which represent the flow along each arc in the minimum cost flow problem, appear in row 1. For example, $f_{x_p y_p}$ in row 2 represents the flow from node x_p to y_p. There is a primal variable for each constraint in the

1	2	3	4	5	6	7	8	9	10	11	12	13
	$f_{x_p y_p}$	$f_{x_q y_p}$	$f_{x_q y_q}$	$f_{x_p ?_1}$	$f_{x_p ?_2}$	$f_{x_q ?_3}$	$f_{x_q ?_4}$	$f_{y_p ?_5}$	$f_{y_p ?_6}$	$f_{y_q ?_7}$	$f_{y_q ?_8}$	
x_p	1			1	-1							$= w_p$
x_q		1	1			1	-1					$= w_q$
y_p	1	1						1	-1			$= w_p$
y_q			1							1	-1	$= w_q$
	$\leq k+1$	$\leq k$	$\leq k+1$	$\leq k$	≤ 0	$\leq k$	≤ 0	$\leq k$	≤ 0	$\leq k$	≤ 0	

Table 6.1: First step in finding a minimum cost flow problem.

dual minimum cost flow problem. For example, $f_{x_p y_p}$, in column 2, is the primal variable for the constraint $x_p + y_p \leq k+1$. Variable $f_{x_p ?_1}$, in column 5, represents the constraint $x_p \leq k$, and $f_{x_p ?_2}$, in column 6, represents the constraint $x_p \geq 0$. Since flows exist between pairs of nodes and there is only one node represented in columns $f_{x_p ?_1}$ and $f_{x_p ?_2}$, we represent the second node as "$?_1$" and "$?_2$", respectively. Through a series of transformations on the table, we will have two entries in each column.

In the minimum cost flow problem, the entry under each flow f in the last row of the table represents the cost of the flow along f. The function we are trying to minimize in the minimum cost flow problem is the dot product of the first and last row. The remaining rows represent the flow divergence for each node. For example, $f_{x_p y_p} + f_{x_p ?_1} + f_{x_p ?_2} = w_p$; that is w_p is the flow divergence of node x_p.

The equations given in Table 6.1 do not conform to a minimum cost flow problem. Two nodes are needed to represent flow along an arc. In columns 5 through 12, there is a single non-zero value, representing a single node. Each column should have a single 1 and a single -1, since each flow adds to the divergence of one node and subtracts from the divergence of another. For a node, a -1 in a row represents an incident edge (incoming flow) on that node and a positive 1 represents an outgoing edge (outgoing flow) from that node.

We transform the equations represented by Table 6.1 into an equivalent set of equations that describes the minimum cost flow problem given above [Zak95]. We perform three transformations. As a flow adds to the divergence of one node and subtracts from the divergence of another, we want each column to have a single 1 and a single -1. We can multiply rows y_p and y_q by -1. Columns 2 through 4 now have a 1 and a -1, since arcs only join nodes x_p and x_q with nodes y_p and y_q. To generate an additional 1 in columns 5

1	2	3	4	5	6	7	8	9	10	11	12	13
	$f_{x_p y_p}$	$f_{x_q y_p}$	$f_{x_q y_q}$	$f_{x_p D}$	$f_{D x_p}$	$f_{x_q D}$	$f_{D x_q}$	$f_{y_p D}$	$f_{D y_p}$	$f_{y_q D}$	$f_{D y_q}$	
x_p	1			1	-1							$= w_p$
x_q		1	1			1	-1					$= w_q$
y_p	-1	-1						-1	1			$= -w_p$
y_q			-1							-1	1	$= -w_q$
D				1	-1	1	-1	-1	1	-1	1	=0
	$\leq k+1$	$\leq k$	$\leq k+1$	$\leq k$	≤ 0	$\leq k$	≤ 0	$\leq k$	≤ 0	$\leq k$	≤ 0	

Table 6.2: Next step in finding a minimum cost flow problem.

1	2	3	4	5	6	7	8	9	10	11	12	13
	$f_{x_p y_p}$	$f_{x_q y_p}$	$f_{x_q y_q}$	$f_{x_p D}$	$f_{D x_p}$	$f_{x_q D}$	$f_{D x_q}$	$f_{y_p D}$	$f_{D y_p}$	$f_{y_q D}$	$f_{D y_q}$	
x_p	1			1	-1							$= w_p$
x_q		1	1			1	-1					$= w_q$
y_p	-1	-1						-1	1			$= -w_p$
y_q			-1							-1	1	$= -w_q$
D				-1	1	-1	1	1	-1	1	-1	=0
	$\leq k+1$	$\leq k$	$\leq k+1$	$\leq k$	≤ 0	$\leq k$	≤ 0	$\leq k$	≤ 0	$\leq k$	≤ 0	

Table 6.3: Next step in finding a minimum cost flow problem.

through 12, we add a row D whose entry in column i, $i > 1$, is the sum of columns i in rows 2 through 5. D represents an additional node in the minimum cost flow problem. Since row D is a linear combination of other rows in the table, adding row D does not change the solution of the minimum cost flow problem. Table 6.2 shows the current table.

The final transformation is to multiply the entries in row D by -1. There is now a 1 and -1 in each column. Table 6.3 gives the final solution.

We can read the the minimum cost flow problem from Table 6.3. For example, for node x_p (row 2), there are two outgoing arcs from x_p, one that is incident on y_p and the other that is incident on D, and there is one incoming arc from node D. The divergence of node x_p is w_p. The divergence of node y_p is $-w_p$. For $f_{x_p y_p}$ (column 2), the cost of the flow is $k+1$ (column 2, last row).

As we know the dual minimum cost flow problem has an optimal solution provided it is feasible and bounded, which is true; hence, so does the minimum cost flow problem [Ber91]. In fact, the objective functions for each problem are equal.

For the dual minimum cost flow problem for interprocedural register allocations with

spilling, we can treat the dual variables for the edges as we treat the dual variables for the candidates above.

Chapter 7

Allocating Registers to Globals

Global register candidates include global variables, procedure addresses, and global array and record addresses [Wal88]. Allocating registers to global candidates is less straightforward than allocating registers to local ones. Let procedures P and Q reference global variable g (see Figure 7.1). Assume an *intra*procedural register allocator assigns g a register in P and Q. As P modifies the value of g, P saves the value of g before the call. When we return to P, the value of g will be reloaded, assuming it will referenced again in P. We refer to the store of g preceding the call and the load upon return as a store/load of g. The value of g is similarly loaded in Q. If Q modifies g, it will save its value so P can reload the correct value of g. We refer to this load and store of g across the call to Q as a load/store of g. If we can somehow allocate global variable g a common register along that path between P and Q in the call graph, we can avoid these extra loads and stores.

Whether to allocate a register to g along a call path is not a simple decision. There may be local candidates in the procedures along the call path. As a result of allocating g a register along the path, these local candidates may not be allocated registers. Also, there may be additional register spilling of local candidates if fewer registers are available.

In the next section, we show that finding an assignment of registers to globals is NP-Complete. In the following sections, we present two algorithms that avoid loads and stores of selected globals by keeping them in registers across procedures. We assume that aliased globals are not allocated registers.

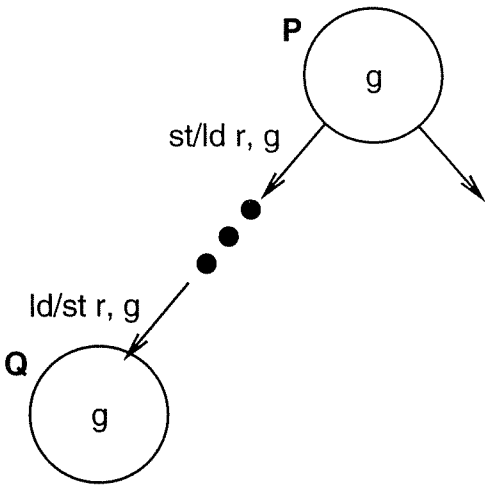


Figure 7.1: A global can be allocated a register along a path in the call graph.

7.1 NP-Complete

We prove that the decision problem, "find an assignment of registers to globals using at most R registers," is NP-Complete. We will reduce Graph k-Colorability to the problem of Interprocedural Register Assignment of Globals (IRG). Let $G_c = (V_c, E_c)$ be an undirected graph. The graph is k-colorable if each node can be assigned one of k colors such that no edge joins two nodes assigned the same color [GJ79].

In the IRG problem, let call graph $G = (P, E)$, in which P is the set of procedures and E is the set of call edges. If procedure $P_v \in P$, references global g, we refer to the instance of g in P_v as g_v. Let *Glob* be the set of instances of globals referenced in each procedure $P_v \in P$, and let $C(P)$ be the set of locals referenced in each procedure $P_v \in P$. Let R be a sequence of registers. We assume instances of the same global must be assigned the same register.

We map solutions from the k-Colorability problem to the IRG problem, in which there are k registers available. For each solution to the colorability problem, we add a *main* routine to the IRG problem. Each node $i \in N_c$ in the graph colorability problem maps in IRG to a unique procedure $p(i)$ that references an instance of a unique global $g(i)$. For $i \in N_c$, procedure $p(i)$ is called only by routine *main*. Edge $e \in E_c$, which joins nodes i and j in the graph colorability problem, maps to a unique procedure $p(e)$ that references an instance of

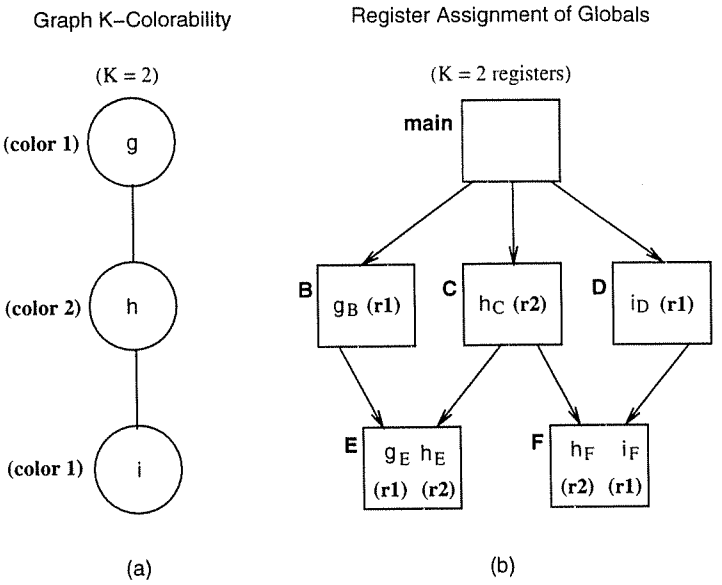


Figure 7.2: An example of a restricted call graph.

globals $g(i)$ and $g(j)$. Procedure $p(e)$ is called only by procedures $p(i)$ and $p(j)$. Figure 7.2 shows an undirected graph (a) in the graph colorability problem and the corresponding call graph (b). Node g in the undirected graph maps to procedure B, which references global g_B (g_B is an instance of global g) in the call graph; node h maps to procedure C, which references global h_C; and, node i maps to procedure D, which references global i_D. For the edge between g and h in (a) we create procedure E, which references global g_E and h_E and for the edge between h and i we create procedure F, which references globals h_F and i_F. The interference relation between globals $g(i)$ and $g(j)$ in procedure $p(e)$ for $e \in E_c$ represents the interference relation between $i \in N_c$ and $j \in N_c$ in the k-Colorability problem.

We refer to procedure $p(i)$ such that $i \in N_c$ as a *level1* procedure (B, C, and D are *level1* procedures in Figure 7.2(b)). Let $e \in E_c$ be an edge in the k-Colorability problem. Then procedure $p(e)$ is a *level2* procedure (procedures E and F in (b) are *level2* procedures). In the *level2* procedures, we assume the live ranges of the globals interfere and, therefore, these globals must be assigned different registers. We assume instances of each global must be assigned the same register. For example in (b), global g_E in E must be assigned the same register as g_B in B.

Theorem 11 *Interprocedural Register Assignment of Globals (IRG) is NP-Complete.*

Proof. 1) IRG $\in$ NP.

Assume there are R registers. Given a register assignment, we can check if all interfering globals are assigned different registers in polynomial time in the number of globals.

2) Graph k-Colorability reduces to IRG.

Each node i in Graph k-Colorability is assigned a *level1* procedure, $p(i)$, and is assigned an instance of global $g(i)$ referenced in procedure $p(i)$. Assume node i is assigned color m, $1 \leq m \leq k$. We assign instances of global $g(i)$ register r_m. If there is an edge e between node i and j then there is a *level2* procedure $p(e)$ called by $p(i)$ and $p(j)$. Both $g(i)$ and $g(j)$ are referenced in $p(e)$ and assigned different registers since i and j are assigned separate colors.

Assume we have a register assignment of globals. The register assigned to an instance of a global in a *level1* procedure is also assigned to another instance of that global in *level2* procedures called by the *level1* procedure. As each *level2* procedure represents an edge e between nodes i and j in the colorability problem, and globals $g(i)$ and $g(j)$ are assigned different registers in IRG, then we can use their register numbers to assign different colors to nodes i and j. For example in Figure 7.2(b), globals g_E and h_E in E are assigned registers r_1 and r_2, respectively. In (a), we assign node g color 1 and node h color 2. In procedure F, we assign global h_F register r_2 and assign i_F register r_1. Therefore, we assign node i in (a) color 1. □

7.2 Wall's Model of Interprocedural Allocation of Globals

In the next two sections we present models for interprocedural register allocation of globals. An *intra*procedural register allocator generates initial loads and terminal stores of a global even if it allocates a register to the global throughout a procedure. To avoid these loads and stores, a global can be assigned the same register across procedures. An *inter*procedural register allocator can keep a global in a register without loads and stores at procedure boundaries.

Wall[Wal86] proposes allocating a register to a global throughout the entire execution

of a program. Wall assigns locals and globals to groups and assigns registers to the groups whose members are most frequently referenced. All the members of a group can be assigned the same register. Since a global is allocated a register throughout the entire program, at most one global can be assigned to a group.

7.2.1 Incorporating Wall's Approach

Wall's approach to register allocation of globals can be easily incorporated into our minimum cost interprocedural register allocator. Assume k registers are available for interprocedural register allocation. Let *Glob* be the set of the most frequently referenced globals in a program, such that $|Glob| \leq k$. We define a partial order whose structure is a chain on the candidates in *Glob*. Two candidates related in a partial order are assigned different registers. As before a partial order has a structure that is a chain on the local candidates in each procedure. Since the local candidates cannot be assigned the same register as the global candidates, we extend the partial order such that each local candidate is less than the global candidates[1]. For example, in Figure 7.3, there is a partial order whose structure is a chain relating the two global candidates, g_1 and g_2, with the local candidates of each procedure. In the partial order, local candidates m, q, and t are less than the global candidates.

Figure 7.4 presents a dual minimum cost flow problem that allocates registers to globals throughout the entire program, and only registers allocated to local candidates can be spilled. In the dual minimum cost flow problem, we add a pair of dual variables (x_i, y_i) for each global $g_i \in Glob$. If $x_i + y_i = k + 1$ for $g_i \in Glob$, then g_i is assigned register x_i; otherwise, g_i is not allocated a register. As is the case for local candidates, associated with each global g_i is an integer weight $w_i > 0$. Let $C(P)$ be the set of local candidates in the call graph and $C(P_v)$ be the set of local candidates in procedure P_v.

The dual minimum cost flow problem in Figure 7.4 is nearly identical to the dual minimum cost flow problem for interprocedural register allocation of locals with spilling. As mentioned above, each local candidate, $c_i \in C(P)$, is less than each global candidate, $g_i \in Glob$, in the partial order. Constraints D.1, D.2, D.3, and D.6, which applied only

[1] We choose not to treat each global candidate as a separate procedure that can call each local procedure, since registers allocated to global candidates are not spilled across calls in Wall's approach.

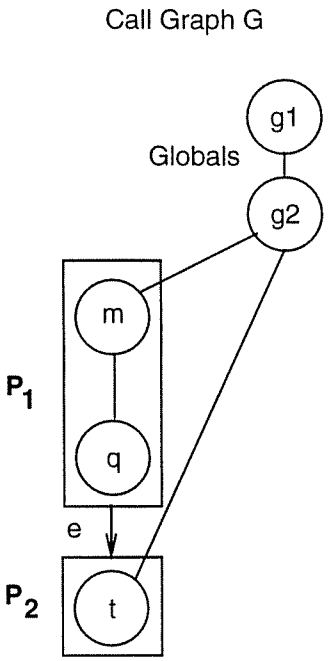


Figure 7.3: Example call graph, in which m, q, and t are local candidates and g_1 and g_2 are global candidates. A partial order relates globals g_1 and g_2. Each local candidate is less than the global candidates in the partial order.

to local candidates when computing an interprocedural register allocation of locals with spilling, now apply to global candidates as well. By D.1, the bounds on the dual variables for global candidates is the same as for local candidates. For $c_j \in Glob$, let $x_j + y_j = k + 1$. Constraints D.2 and D.3 imply that if c_j is assigned register x_j, than that register is not available for candidates less than c_j in the partial order. For example, let $c_i \sqsubset c_j$. Then by D.2, $x_i + y_j \leq k$. This implies that $x_i < x_j$. By constraint D.6, $r_i + y_j \leq k$ for $e_i \in OUT(P_v)$ and $c_j \in Glob \bigcup C(P_v)$. If $c_j \in Glob$ is allocated a register ($x_j + y_j = k + 1$), then r_i, the number of free registers on entrance to call edge e_i, is less than x_j, the register number assigned to c_j.

A register allocation and assignment of the call graph G in Figure 7.3 is shown in Figure 7.5(a). We assume three registers are available. Registers are allocated to global candidates g_1 and g_2 and local candidates m and t. A register is spilled along edge e_1. Figure 7.5(b) shows the graph of a dual minimum cost flow problem whose solution corresponds to the register allocation and assignment in (a). For candidate g_1, dual variables

Dual Variables

(x_i, y_i for i such that $c_i \in C(P) \bigcup Glob$), ($r_j$, t_j for j such that $e_j \in E$)

Constraints

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

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

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

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 $c_j \in C(P_v) \bigcup Glob$, $r_i + y_j \leq k$.

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

Figure 7.4: Dual minimum cost flow problem whose solutions are mapped to interprocedural register allocations of globals and locals. Only local candidates allocated registers are spilled across calls.

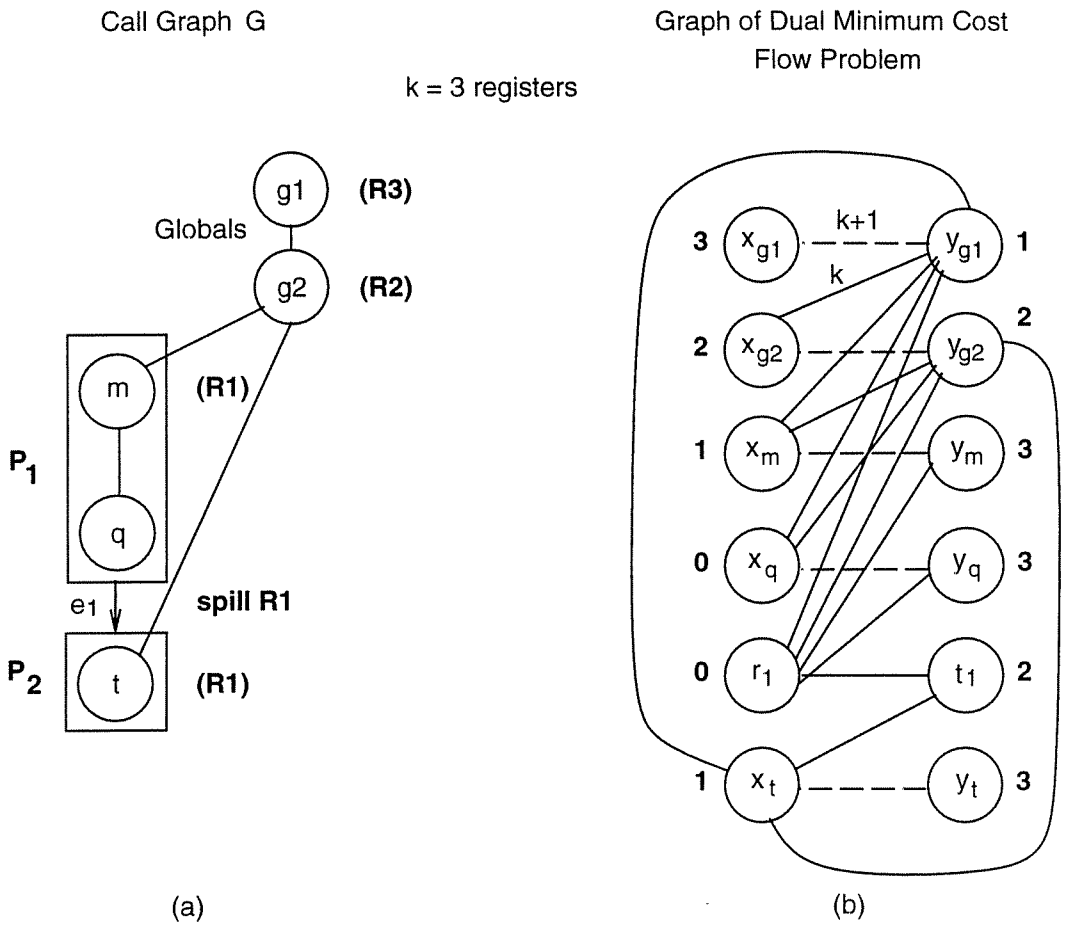


Figure 7.5: (a) shows an example call graph referencing two global candidates g_1 and g_2. A register allocation and assignment are also shown with the two global candidates assigned registers R_3 and R_2. (b) presents a dual minimum cost flow problem whose solution corresponds to the allocation and assignment in (a). Nodes represent dual variables and edges represent constraints on the dual variables. Two dual variables joined by a dashed edge may sum to at most $k+1$, and two dual variables joined by a solid edge may sum to at most k.

$x_{g_1}+y_{g_1} = k+1$ and $x_{g_1} = 3$. We, therefore, assign register R_3 to g_1. Since $x_{g_1}+y_{g_1} = k+1$, and $x_{g_2}+y_{g_1} = k$, $x_{g_2} < x_{g_1}$. Since dual variables $x_{g_2}(=2)+y_{g_2}(=2) = k+1$, g_2 is assigned register R_2. One register is available to candidates m and q. Candidate m is allocated a register, leaving 0 registers free on entry to edge e_1 ($r_1 = 0$). Since $k - r_1 - t_1 = 1$, one register is spilled on edge e_1. This register is allocated to t since $x_t(=1)+y_t(=3) = k+1$.

7.2.2 Implementation

We implemented the interprocedural register allocator discussed in the previous section on a DECstation 5000/125. SPEC92 benchmarks were optimized at level -O2. These benchmarks were not optimized with loop-unrolling since gcc does not unroll loops whose bounds reference register-allocated globals. Interprocedural register allocation was performed on both user code and library routines. To estimate the number of register and global references, we summed the number of global references and register references in each procedure. We scaled these numbers using profile information of the number of instructions executed in each procedure. We profiled input generating shorter execution times than the standard input for all benchmarks except *nasa7* and *swm256*, in which we have only one input file.

Figure 7.6 presents the performance improvement from allocating registers to global candidates. Column *without globals* refers to the benefit of interprocedural register allocation of locals with spilling over gcc. Column *with globals* is the benefit of interprocedural register allocation of locals and globals, in which only locals can be spilled across calls. The last two columns show the number of globals promoted to registers. Dashes indicates that no globals were allocated registers.

No globals were allocated registers in a few benchmarks. For these benchmarks, allocating registers to globals throughout the entire call graph represents too large of a granularity. On benchmark *hydro2d*, our allocator performs worse when allocating registers to globals. For this benchmark the allocator allocates registers to globals used in I/O library routines. These routines represent a much smaller percentage of the total compilation time for the standard input than for the profiler input. In benchmark *mdljdp2*, we are able to remove a significant number of loads and stores of globals in procedures *jloopu* and *jloopb*. These

Performance Improvement				
benchmark	*without globals*	*with globals*	*globals*	
			integer	floating-point
compress	1%	5%	7	0
doduc	3%	2%	1	0
ear	0%	0%	0	2
eqntott	0%	1%	1	0
espresso	—	—	—	—
fpppp	—	—	—	—
gcc	—	—	—	—
hydro2d	0%	-2%	4	0
mdljdp2	0%	8%	1	5
mdljsp2	1%	1%	2	3
nasa7	—	—	—	—
ora	1%	6%	5	5
sc	8%	8%	2	1
spice	—	—	—	—
su2cor	—	—	—	—
swm256	1%	2%	1	4
xlisp	11%	14%	5	0

Figure 7.6: Performance improvement of interprocedural register allocation with and without allocating registers to global candidates. Dashes indicate that no global candidates are allocated registers.

benchmark	*% of compilation time*	
	floating-point	integer
compress	0.1% (+0%)	1.3% (+86%)
doduc	0.3% (+50%)	1.3% (+63%)
ear	0.5% (+25%)	1.5% (+88%)
eqntott	0.1% (+0%)	2.3% (+155%)
espresso	0.6% (+20%)	4.7% (+81%)
fpppp	0.2% (+100%)	0.4% (+100%)
gcc	0.8% (+33%)	7.6% (+52%)
hydro2d	0.3% (+50%)	2.3% (+156%)
mdljdp2	0.7% (+133%)	1.8% (+125%)
mdljsp2	0.3% (+0%)	1.0% (+25%)
nasa7	0.2% (+0%)	1.0% (+25%)
ora	0.4% (+100%)	1.5% (+50%)
sc	0.4% (+33%)	2.2% (+57%)
spice	0.3% (+50%)	0.9% (+125%)
su2cor	0.2% (+0%)	0.7% (+17%)
swm256	0.5% (+150%)	1.3% (+86%)
xlisp	1.2% (+33%)	2.2% (+83%)

Figure 7.7: Time solving the network flow problem as a percentage of the total compilation time without interprocedural register allocation.

procedures represent 90% of the execution time. We are able to achieve additional improvement on benchmark *xlisp*. Global variables referencing the stack and free memory are allocated registers.

Benchmark *doduc* allocates a register to global candidate *errno*, which appears frequently within library routines, but dynamically is infrequently referenced. Assigning a register to *errno* leads to a decrease in performance.

Figure 7.7 presents the time solving the network flow problem as a percentage of the total compilation time without interprocedural register allocation. The numbers in parentheses represents the increase in the percentage with respect to interprocedural register allocation of only locals. Though solving the network flow problem is expensive for *gcc*, solutions can be found quickly for the smaller benchmarks.

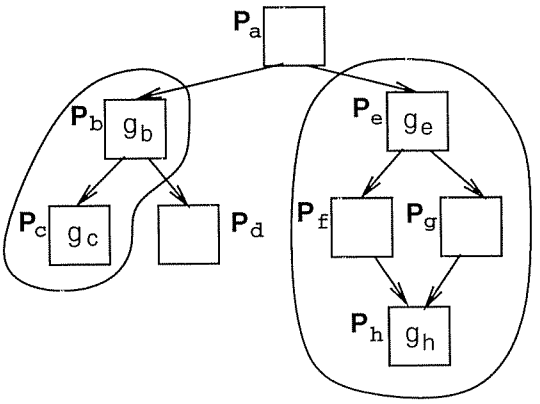


Figure 7.8: Call graph in which the circled nodes are webs of global g. We refer to an instance of global g in a procedure $P_v \in P$ as g_v.

7.3 A More Precise Model

In this section, we present a model that allocates registers to globals over a smaller section of the call graph than in Wall's approach. Our approach is based on Santhanam and Odnert's register allocation of globals [SO90].

7.3.1 Webs

For each global Santhanam and Odnert [SO90] partition the call graph into webs. A web for a global g is a minimal subgraph of a program call graph such that g is not referenced in a procedure that is either an ancestor or a descendant of the subgraph[2]. To models loads and stores of globals in procedures, a procedure that is part of a web has either all predecessors in the web or no predecessors in the web. There can be several disjoint webs that reference a particular global. A register assigned to a web is assigned to the corresponding global in all procedures of the web. In Figure 7.8 the circled procedures represent two webs for global g. Both of these webs can be assigned different registers. For a procedure P_v in a web, we refer to an instance of global g as g_v. Procedures that are ancestors and descendants of each web in Figure 7.8 do not reference global g.

Globals are only allocated callee-save registers to avoid spilling registers around library routines (which have been compiled using an intraprocedural register allocation). Define

[2]Webs generated by their algorithm are not necessarily minimal.

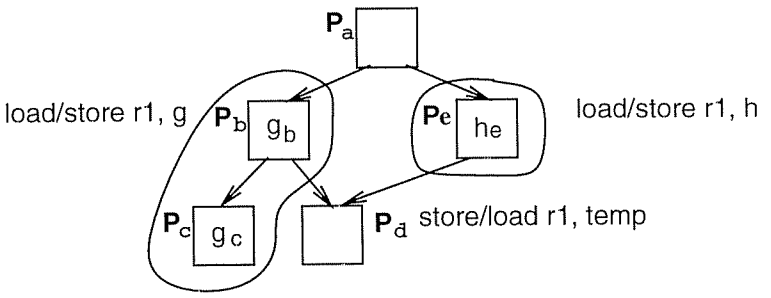


Figure 7.9: Two webs are shown. One web is for global g and the other is for global h. Instructions for loading the globals into registers and saving them back into registers upon return are shown. We assume that each web's global is assigned the same register $r1$, and $r1$ is also assigned to a local in procedure P_d.

a web entry procedure as a procedure having no predecessors in the web. At an entry procedure, a global is loaded from its home location into a callee-save register. When execution returns from a web entry procedure, the global is stored into its home location. When execution leaves a web through a non-entry procedure (as a result of a call), the global is spilled into a temporary when its register is needed for another register candidate. The global is spilled into a temporary instead of its home location, since outside of a web, a register may be allocated to one of several globals. The global will be reloaded into a register before execution reenters the web.

Figure 7.9 displays two webs, one web for global g and the other for global h. Since procedure P_d is in neither web, it does not reference globals g or h. At each web's entry procedures, its global is loaded into register $r1$ from the global's home location. Upon returning from an entry procedure, the global is stored back into its home location. Assume procedure P_d assigns $r1$ to a local candidate. P_d saves $r1$ on entry to P_d into a temporary, and reloads the value from the temporary upon returning from P_d. In an invocation of P_d, the temporary has the value of global g or h.

7.3.2 Refining Webs

A drawback with Santhanam and Odnert's approach is that a global is allocated a register throughout an entire web. Allowing the register to be spilled and allocated to a local within a web may lead to a better interprocedural register allocation.

As proved in Section 7.1, the problem of finding a mapping of registers to global candidates is NP-Complete. In our approach, we fix the register assignment and have the interprocedural register allocator select the procedures in which a global is allocated a register.

Two webs interfere (are not assigned the same register) if they overlap. We partition non-interfering webs into sets. Each register can be assigned to only one set of webs. Given a register assignment, our interprocedural register allocator will select the procedures within the webs where a global is register allocated. If we spill a global within a web, then the global is spilled into its home location.

The construction of our webs differ from that of Santhanam and Odnert. Santhanam and Odnert's approach allows either all callers of a procedure or no callers of a procedure to be in the same web as the callee. Our approach models register loads and stores of globals across calls, thereby allowing only some callers of a procedure to be in the same web as the callee. Our interprocedural register allocator generates webs that can never be larger than those in Santhanam and Odnert's algorithm.

7.3.3 Generating an Allocation with Global Candidates

Figure 7.10 outlines our approach to register allocation of local and global candidates. Let k be the number of available registers (line 1). First, our allocator forms a set of webs for each global. Next, a web interference graph is generated. Two webs interfere if they have a procedure in common. Interfering webs must be assigned different registers.

In lines 7 to 26 of Figure 7.10, we loop through the set of k registers. In iteration i, $1 \leq i \leq k$, there are $k - i + 1$ registers available for allocation. In each iteration, there is a single set of non-overlapping webs whose globals can be allocated a register (lines 8 to 12). Thus, in a given iteration there is at most one global candidate that can be allocated a register in each procedure. We also add a global candidate for procedures that are descendants of webs in the call graph, but are themselves not members of a web, since a global can remain in a register upon leaving a web through the leaves. Each global candidate can only be assigned abstract register R_1 (line 14). Local candidates can be assigned any abstract register.

```
procedure GenerateAllocation
1     Let k be the number of registers.
2     Let cand be the local candidates in all procedures.
3     Let web_set be the set of webs for each global.
4     Build web_interference graph from web_set.
5     i = 1.
6
7     while (i <= k) do
8           Remove a single set of non-overlapping webs from web_interference graph.
9           Let WebProc be the procedures in the set.
10          Let BelowWebProc be the procedures that are
11          descendants of WebProc in the call graph.
12          Add a global candidate to cand for procedures in WebProc and
13          BelowWebProc.
14          Perform interprocedural allocation on local and global candidates in cand.
            There are k - i + 1 registers available.
            Register R1 can be assigned to global and local candidates.
            All other registers can be assigned only to local candidates.
15
16          if a global is allocated a register then
17                Map R1 to a hardware register.
18                Remove all global candidates from cand and local candidates assigned R1.
19                Mark loads and stores of R1.
20                i ← i + 1.
21          else
22                Map abstract registers to hardware registers.
23                Mark loads and stores of registers.
24                break from loop.
25          end if
26    end while
end procedure
```

Figure 7.10: Outline of our algorithm to allocate registers to local and global candidates.

At the end of an iteration, if a global candidate is allocated a register, then we remove the local and global candidates assigned R_1 and save information of the loads and stores of R_1 along the call edges (lines 16 to lines 20). Each instance of R_1 in a loop iteration maps to a distinct hardware register. In the next iteration of the loop, there is one less available register. If a global is not allocated a register in the current iteration (lines 22 to 24), then this last allocation in conjunction with the previous allocations of R_1 represent the final allocation, and execution exits the loop.

7.3.4 Choosing a Set of Disjoint Webs for each Iteration

In iteration i, $1 \leq i \leq k$, we choose a set of disjoint webs that can be assigned the same register (lines 8 to 12 of Figure 7.10). We follow a similar approach as [SO90] for selecting a set of disjoint webs. Given a web interference graph, we select webs in an ordering based on a priority function.

Our approach has a different priority function than [SO90]. Santhanam and Odnert's priority function measures the benefit of allocating registers to globals, but subtracts the cost of loads and stores of these globals at the web entry nodes.

Our priority function includes additional costs. Register allocating a global in a procedure has a benefit of removing loads and stores, but if the procedure could allocate all $k-i+1$ registers to its locals, there is a cost of one less register available to the procedure's local candidates. We include this cost in our priority function. In addition, we include the cost of saving and restoring a register-allocated global around *setjmp* and saving a register-allocated global before a *longjmp*. Let the benefit minus the costs be t_s for web s. To favor small webs over large ones, we divide t_s by the number of procedures in the web. As we are interested in a single set of disjoint webs in each iteration, we traverse through the list of webs ordered by our priority function. We select a web if it does not interfere with other webs selected in the current iteration of our algorithm.

A register may remain allocated to a global upon (temporarily) exiting a web through the leaves. For each procedure P_v that is not in a web and is a descendant of a web procedure, we add a dummy global candidate. If a dummy global candidate is allocated a register, then a global in an ancestor of P_v in the call graph has its value in a register in P_v.

7.3.5 Modeling Interprocedural Register Allocation of Globals

In this section we define the constraints and maximization function for an interprocedural register allocation of globals. We assume R_1 can be shared between local and global candidates, while all other registers can be allocated only to local candidates. First, we describe our model for inserting loads and stores of globals along call edges.

Modeling Loads and Stores of Globals

We differentiate between three types of call edges for inserting loads and stores of globals: (1) calls between procedures of the same web (set $WebEdge$); (2) calls to a non-web procedure, which is also a descendant of a web procedure in the call graph (set $BelowWebEdge$); and, (3) calls to a web procedure such that the caller is not in the callee's web (set $WebEntry$). We identify the loads and stores of a global needed around a call based on the type of call edge and whether a global is allocated a register in the caller or callee.

1. Let edge $e_j \in WebEdge$ if the caller and callee of call e_j are in the same web. Loads and stores of a global along edge e_j reference the global's home location. The global's home location has the global's current value immediately before a load. Given these call edges, if a global is assigned register R_1 in the caller, but not in the callee, then a store/load of the global is needed around the call. If a global is assigned register R_1 in the callee, but not the caller, then a load/store is needed around the call.

2. Let edge $e_j \in BelowWebEdge$ if there is a path to e_j from a procedure in a web and e_j is not incident on a procedure in a web. The only loads and stores of a global along these edges is a store before a call and a load upon returning from the call. These loads and stores reference temporary locations. For example, in Figure 7.11, call edge e_j is incident on procedure P_f, which is not in a web. There is a path that leads to e_j from procedures P_b and P_e, both of which are in webs. Thus, $e_j \in BelowWebEdge$.

   In Figure 7.11, assume g and h are assigned the same register R_1. Based on the construction of a web, globals g and h are not referenced in procedures reachable from e_j, though they still may be in a register across e_j. If we choose to spill R_1 along

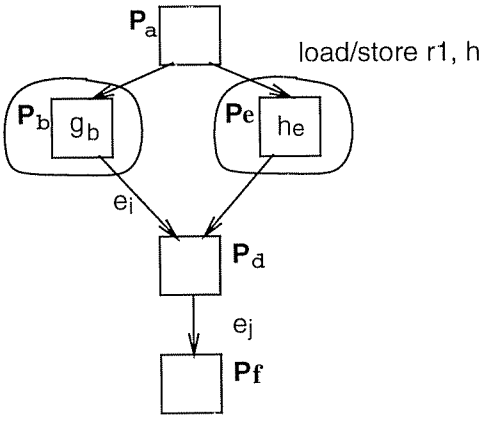


Figure 7.11: Assume g and h are assigned the same register. Either global g or h could be in that register across the call edge e_j.

e_j, then the register must be spilled into a temporary (because at e_j we do not know which global R_1 currently holds).

For a set of webs in which $e_i \in BelowWebEdge$, their globals can be spilled into a temporary along e_i, but their globals will not be loaded into a register along e_i. In Figure 7.11, assume global g is not allocated a register in procedure P_b, but global h is in a register in P_e and P_d. A load of global h is not needed along the call edge e_i from P_b to P_d. Along call edges incident on non-web procedures, there can only be a cost of storing a register-allocated global into a temporary before a call and loading from the temporary after the call.

3. Let set $WebEntry$ be the call edges in which the callee is in a web, but the caller is either in a different web or not in any web at all. If a different global in the callee is allocated a register, then a load/store of the global is added along the edge. This load and store reference the global's home location. If a global is allocated a register in the caller, then a store/load is added along the edge. This load and store reference a temporary location.

Indirect Calls

Indirect calls pose a problem since the caller and callee may be in the same web, may be in different webs, or the callee may not be in a web. As was done for interprocedural register

allocation of locals, we assume each indirectly called routine is called by a single dummy procedure. In addition, we assume that either all callers or no callers of an indirectly called procedure are in the same web as the callee. The call edge from the single dummy node will be a member of $WebEdge$ if all callers are in the same web as the callee, will be a member of $WebEntry$ if the callee is in a web different than its callers, and will be a member of $BelowWebEdge$ if the callee is not in a web, but is a descendant of a procedure in a web. Loads and stores along these edges are placed in the callee.

We want to avoid load/stores of globals along indirect calls, as the callee is unknown. Loading a global into a register before an indirect call and storing the register into the global's home location upon return can yield a run-time error if the callee does not allocate a register to that global. We assume that a global is either allocated a register across all calls or no calls to an indirect procedure. We, therefore, can model loads and stores as occurring in procedures instead of along call edges. We can fix the values of the dual variables to achieve this result. The dual minimum cost flow problem is discussed in Section 7.3.6.

Constraints and Maximization Function

As in interprocedural register allocation of locals with spilling, a partial order restricts which candidates can be assigned the same register. The partial order on the global and local candidates in a procedure is slightly different than the partial order on local candidates in our interprocedural register allocation of locals with spilling. We assume the global candidate and local candidates in a procedure form a partial order whose structure is a chain, in which the global candidate is the least-most element in the ordering. In Figure 7.12, procedures P_v and P_w are both in a web referencing global g. P_v references instance g_v and P_w references instance g_w of global g. Both g_v and g_w are the least-most elements in the ordering among candidates in a procedure.

An interprocedural register allocation of globals is represented by tuple

$$I = (R, free\_in, free\_out).$$

R is a sequence of abstract registers $(R_1, \ldots, R_k)$, where k is the number of registers. Entry

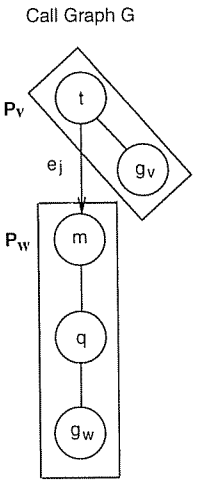


Figure 7.12: Call graph in which instances of global g are the least elements in the ordering among candidates in each procedure.

$free\_in$ is the sequence $(free\_in_1, \ldots, free\_in_{|E|})$, where $free\_in_j$, $1 \leq j \leq |E|$, is the number of unallocated registers on entry to edge $e_j \in E$. Entry $free\_out$ is the sequence $(free\_out_1, \ldots, free\_out_{|E|})$, where $free\_out_j, 1 \leq j \leq |E|$, is the number of unallocated registers on exit from edge $e_j \in E$.

For $e_j \in E$, $free\_out_j - free\_in_j$ is the number of register-allocated local candidates spilled across the call e_j. Let e_j be the call edge from procedure P_v to P_w. A register allocated to a global candidate in P_v is treated by $free\_in_j$ and $free\_out_j$ as not allocated across the call e_j. Hence, $free\_in_j$ and $free\_out_j$ consider the register as available to candidates in P_w. In Figure 7.12, global g_v does not constrain the number of free registers on entry to edge e_j. Both the global candidate and local candidates in P_w must compete for the same registers. If a global candidate is allocated a register in the caller and callee, then there are no register loads and stores of the global around the call e_j.

In each iteration of the interprocedural register allocation there is at most one global candidate that can be allocated a register in each procedure. Each procedure in a web has a single global candidate. A procedure that is not in a web, but is a descendant of a procedure in a web, has a single dummy global candidate. We allow an abstract register R_1 to be assigned to either a global candidate or local candidates in a procedure. All other abstract registers can only be assigned to local candidates.

Define the following sets:

- $WebProc$ is the set of procedures in a web.
- $BelowWebProc$ is the set of procedures not in a web and are reachable from a procedure in a web. Call edges in $BelowWebEdge$ are incident only on procedures in $BelowWebProc$.
- $WebGlobal(P_v) = \{g_v\}$ if procedure $P_v \in WebProc$ for global g; $WebGlobal(P_v) = \{dummy_v\}$ if procedure $P_v \in BelowWebProc$; otherwise, $WebGlobal(P_v)$ is the empty set.
- $WebGlobal(P) = \bigcup_{P_v \in P} WebGlobal(P_v)$ ($WebGlobal(P)$ includes dummy global candidates).

Set $C(P_v)$ is the set of local candidates in procedure P_v. The constraints on an interprocedural register allocation of locals and globals are given in Figures 7.13 and 7.14.

The list of constraints I.1 – I.6 in Figure 7.13 are identical to the list of constraints for interprocedural register of locals in Section 4.2. We add the constraints J.1 – J.3 in Figure 7.14 for the global candidates. Constraint J.1 implies that a global candidate can only be assigned to abstract register R_1. By constraint J.2, if both a local and global candidate are allocated registers in the same procedure, then the register number of the global candidate (R_1) must be less than the register number of the local candidate. In the partial ordering on candidates in a procedure, a global candidate is less than all the local candidates. By constraint J.3 there must be at least one register free on entry to procedure P_v for the global candidate in P_v to be assigned R_1.

Given the above constraints we want to maximize a function which represents the benefit of an allocation. The function uses the following terms:

- integer $s_j > 0$ is the cost at call edge e_j of a register store/load of a global or local. The store and load reference a temporary location.
- integer $hs_j > 0$ is the cost at call edge e_j of inserting a store/load or a load/store of a global. The load and store reference a global's home location.

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 $c_i \in C(P_v)$ and $c_j \in C(P_v)$. If $c_i \in R_p$, $c_j \in R_q$, and $c_i \sqsubset c_j$, then $p < q$.

Figure 7.13: Constraints between local candidates for interprocedural register allocation.

J.1 Let $P_v \in WebProc \bigcup BelowWebProc$. If $g_v \in WebGlobal(P_v)$ and $g_v \in R$, then $g_i \in R_1$.

J.2 Let $P_v \in WebProc \bigcup BelowWebProc$, $g_v \in WebGlobal(P_v)$ and $c_j \in C(P_v)$. if $g_v \in R_1$, $c_j \in R_q$, and $g_v \sqsubset c_j$, then $1 < q$.

J.3 Let $P_v \in WebProc \bigcup BelowWebProc$. If $g_v \in WebGlobal(P_v)$, $g_v \in R_1$, and $e_j \in IN(P_v)$, then $1 \leq free\_out_j$.

Figure 7.14: Constraints involving global candidates.

$$maximize\ (a)\ \sum_{c_i \in R} w_i - (b)\ \sum_{e_j \in E} s_j * (freeout_j - freein_j) - (c)\ \sum_{e_j \in GlobTempSt} s_j - (d)\ \sum_{e_j \in GlobHomeLdSt} hs_j.$$

Figure 7.15: Maximization function for interprocedural register allocation of globals.

- integer $w_i > 0$ for local candidate $c_i \in C(P)$ is the benefit of allocating a register to a local candidate. We assume a positive benefit of allocating a register to each candidate. This benefit is a function of the number of references to the candidate.

- integer $w_v \geq 0$ for global candidate $g_v \in WebGlobal(P_v)$ is the benefit of allocating a register to instance g_v in P_v.

- $GlobTempSt$ is the set of call edges in which there is a store/load of a register-allocated global, such that the load and store reference a temporary location. This set includes register saves and restores of globals around call edges in $BelowWebEdge$ and $WebEntry$.

- $GlobHomeLdSt$ is the set of call edges in $WebEdge$ and $WebEntry$ in which there is a register load/store or store/load of a global around a call, such that the load and store reference the global's home location.

The function that we maximize is shown in Figure 7.15. Expression (a) in Figure 7.15 refers to the benefit of register-allocated local and global candidates; (b) is the cost of spilling register-allocated local candidates around calls; (c) represents the cost of loads and stores of a global that reference a temporary location; and, (d) is the cost of loads and stores of a global that reference the global's home location.

Figure 7.16 presents a register allocation and assignment of the call graph in Figure 7.12. We assume two registers are available. Globals g_v and g_w are instances of global g. Procedures P_v and P_w are in the same web for global g. In procedure P_v, local candidate t and global candidate g_v are allocated registers. We assume global g is not allocated a register in the caller of P_v. Since g is allocated a register in P_v, we load the global into a register from its home location upon calling P_v and store the global's value upon return.

Since allocating a register to g_v in P_v does not decrease the value of $free\_in_j$, there is

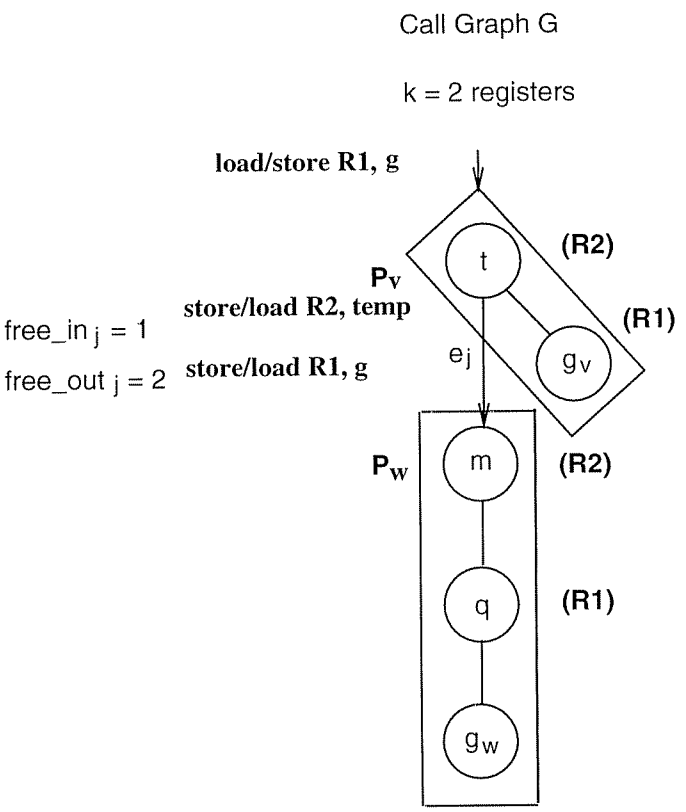


Figure 7.16: A register allocation and assignment of global and local candidates in a call graph.

no constraint between g_v and e_j. Hence, $free\_in_j = 1$ and R_1 is free on entry to P_w. As $free\_out_j = 2$ and $free\_out_j - free\_in_j = 1$, the register allocated to the local candidate t is spilled around call e_j. In procedure P_w, local candidates m and q are allocated registers. Since global g_w is not allocated a register in P_w, R_1, the register assigned to g_v, is spilled around the call e_j.

7.3.6 Dual Minimum Cost Flow Problem for Interprocedural Register Allocation of Globals

In this section we present a dual minimum cost flow problem that models the cost of loads and stores of globals around calls. A load and store of a global can occur along the call edges in sets $WebEdge$, $BelowWebEdge$ and $WebEntry$.

Modeling Spill Cost in $WebEdge$

Let procedure P_v and P_w be in the same web. Let edge e_j represent the call from procedure P_v to P_w. To model the cost of register loads and stores of globals around e_j, we add integer dual variables and constraints shown in Figure 7.17(a). Nodes represent dual variables and arcs represent constraints.

The value of each dual variable in (a) is either 0 or 1. The constraints represented by the arcs limit the sum of two dual variables to be at most 1. For edge e_j, Figure 7.17(b) shows the maximum values of dual variables a_{j1} and a_{j2} for each possible combination of values of dual variables b_{j1} and b_{j2}. Nodes a_{j1} and a_{j2} have the same constraints and, therefore, are equal in each case. As shown in (b), if $b_{j1} = b_{j2}$, then two dual variables equal 1; otherwise, only one dual variable equals 1. Assume that if a global is allocated a register in P_v, then $b_{j1} = 0$; otherwise, $b_{j1} = 1$. Similarly, if a global is allocated a register in P_w, then $b_{j2} = 0$; otherwise, $b_{j2} = 1$. Thus, if there are no register loads and stores of a global along call edge e_j, then $b_{j1} = b_{j2}$. For $e_j \in E$, define hs_j to be the cost of a register store/load or load/store, such that the load and store reference a global's home location. We let hs_j be the weight associated with each dual variable in (a). Then an allocation in which there are no register loads and stores of a global along e_j ($b_{j1} = b_{j2}$) has an additional benefit of hs_j, since two dual variables equal 1. Using the variables and constraints of a dual

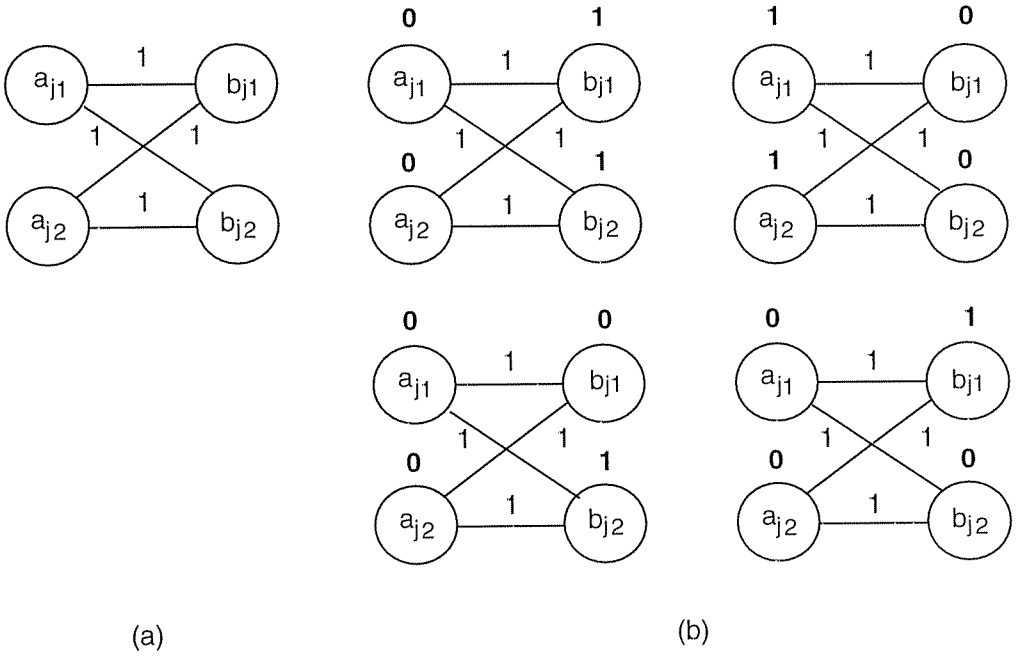


Figure 7.17: Dual variables and constraints to model loads and stores along call edges in set $WebEdge$. The maximum values of a_{j1} and a_{j2} are shown for each possible combination of values of b_{j1} and b_{j2}.

minimum cost flow problem, we have accurately modeled the additional cost of load/stores and store/loads of globals in a web.

Figure 7.18 shows the dual variables and constraints for a global g referenced in both P_v and P_w, such that P_v calls P_w along edge e_j. Variable pairs (d_v, f_v) are the dual variables representing g in P_v and (d_w, f_w) are the dual variables representing g in P_w. Dual variables $a_{j1}, b_{j1}, a_{j2}, b_{j2}$ model the cost of loads and stores for call edge e_j. Assume there are k registers available. We let $f_v = f_w = k$ and $0 \leq d_v, d_w \leq 1$. If $d_v + f_v = k + 1$, then global g in P_v is assigned register R_1. Otherwise $d_v + f_v = k$ and g is not allocated a register in P_v. In our example, we assume that both d_v and d_w are assigned the value 1. Since $f_v = k$, then $d_v = 1$ implies that g is assigned register R_1 in P_v. Since we also have a constraint that $d_v + b_{j1} = 1$, then $b_{j1} = 0$ implies g is allocated a register in P_v. Similarly, $b_{j2} = 0$ implies g is allocated a register in P_w. Let the weight of dual variables a_{j1}, a_{j2}, b_{j1}, b_{j2} be the cost of a register store/load or load/store. If $b_{j1} = b_{j2}$, then two of these dual variables equal one; otherwise; only one dual variable equals one. The decrease in benefit from a

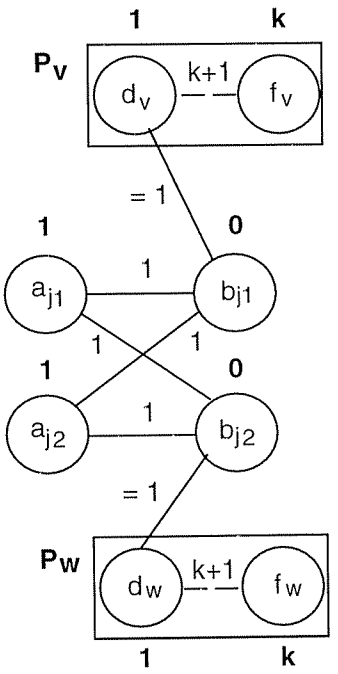


Figure 7.18: Variables d_v and f_v are the dual integer variables for global g in procedure P_v and d_w and f_w are the dual variables for global g in procedure P_w. The values of f_v and f_w are assigned k. The values of d_v and d_w are either 0 or 1.

register load and store of global g is accurately modeled.

Modeling Spill Cost in *BelowWebEdge*

A call edge e_j is in *BelowWebEdge* if there is a path from a procedure in a web leading to e_j and e_j is incident on a procedure not in any web. To model the cost of a store/load of a global into a temporary along e_j, we add the dual variables and constraints in Figure 7.19(a). The constraints are similar to those in Figure 7.17(a), but there is no constraint between a_{j2} and b_{j1}. Now only when $b_{j1} = 0$ and $b_{j2} = 1$ is there only one dual variable equal to 1. Since $b_{j1} = 0$, a global in the caller is allocated a register. Since $b_{j2} = 1$, a global in the callee is not allocated a register. This dual variable assignment represents a store/load of a register-allocated global along call e_j. By letting the weight of each dual variable be the cost of a spilling a global into a temporary, we are able to model the decrease in benefit from spilling a register-allocated global along a call edge in set *BelowWebEdge*.

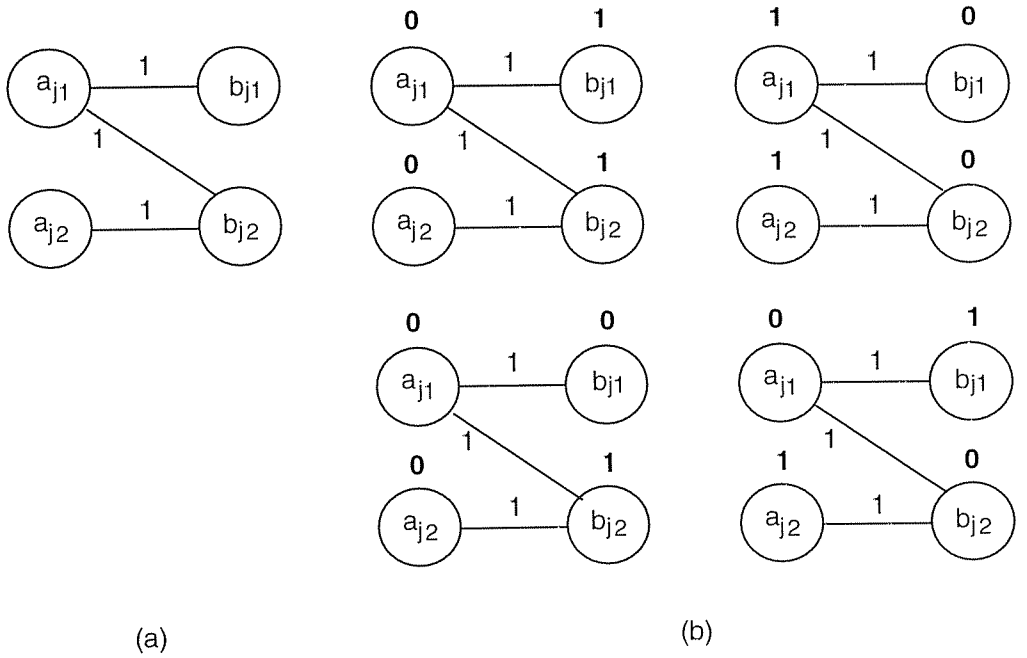


Figure 7.19: Dual variables and constraints to represent the cost of spilling register-allocated globals along call edges not incident on a web.

Modeling Spill Cost in *WebEntry*

Assume we are given a call edge e_j, such that the callee is in a web, and the caller is in either a different web than the callee or is not in a web.

If a global g is allocated a register in the caller, then there is a register store/load of global g into a temporary around the call. If a global h is allocated a register in the callee, then there is a register load/store of h around the call. We can model the cost of these loads and stores by assigning $a_{j1} = a_{j2} = 0$ in Figure 7.17(a). If $b_{j1} = 1$, then a global is not allocated a register in the caller; otherwise, $b_{j1} = 0$ and the global is allocated a register in the caller, in which case the global is saved and restored into a temporary around the call. Similarly, if $b_{j2} = 1$, then a global is not allocated a register in the callee; otherwise, $b_{j2} = 0$ and the global is loaded into a register from its home location in the callee and stored upon return. The weight of b_{j1} is the cost of a store/load of a global in the caller and the weight of b_{j2} is the cost of a load/store of a global in the callee. With these weights, we are able to model the decrease in benefit from the loads and stores along call edge $e_j \in WebEntry$.

Dual Variables

1. $(x_i,\ y_i)$ for i such that $c_i \in C(P)$,
2. $(r_j,\ t_j)$ for j such that $e_j \in E$,
3. $(a_{j1},\ b_{j1})$, $(a_{j2},\ b_{j2})$ for j such that $e_j \in WebEntry \bigcup WebEdge \bigcup BelowWebEdge$.
4. $(d_v,\ f_v)$ for v such that $P_v \in WebProc \bigcup BelowWebProc$

Figure 7.20: Dual variables of the dual minimum cost flow problem for allocating registers to globals.

Dual Minimum Cost Flow Problem

We present a dual minimum cost flow problem that generates an allocation allowing register R_1 to be shared between global and local candidates and the remaining registers to be allocated only to local candidates.

Figure 7.20 presents the dual variables of the dual minimum cost flow problem. As in the dual minimum cost flow problem for interprocedural register allocation without globals, there are a pair of dual variables (1) $(x_i,\ y_i)$ for each local candidate $c_i \in C(P)$ and a pair of dual variables (2) (r_j, t_j) for each edge $e_j \in E$. To model spilling of globals along call edges there are dual variables (3) $(a_{j1},\ b_{j1})$ and $(a_{j2},\ b_{j2})$ for each edge $e_j \in WebEntry \bigcup WebEdge \bigcup BelowWebEdge$. Since a global can be allocated a register in any procedure of a web or descendant of a web, we have a pair of dual variables (4) $(d_v,\ f_v)$ for $P_v \in WebProc \bigcup BelowWebProc$.

Figure 7.21 shows the constraints G on the dual variables for the global candidates. Figure 7.22 shows the constraints D on the local candidates. Constraints G.1 and G.2, which model the cost of load/stores and store/loads of globals, are as described in Section 7.3.6. If a global is allocated a register in the callee of an edge $e_j \in WebEntry$, then a register load/store of the global is added along the edge. If a global is allocated a register in the caller of an edge $e_j \in WebEntry$, then a save/restore is added along the edge. As discussed in Section 7.3.6, by assigning $a_{j1} = a_{j2} = 0$ for edge $e_j \in WebEntry$ (constraint G.3), if the global is allocated a register in the callee (or caller), there is a loss of a benefit equal to the cost of a load/store or (store/load).

Constraints

G.1 For $e_j \in WebEdge \bigcup WebEntry$,
$0 \leq a_{j1}, a_{j2}, b_{j1}, b_{j2} \leq 1,\ a_{j1} + b_{j1} \leq 1,\ a_{j2} + b_{j2} \leq 1,\ a_{j1} + b_{j2} \leq 1,\ a_{j2} + b_{j1} \leq 1$

G.2 For $e_j \in BelowWebEdge$,
$0 \leq a_{j1}, a_{j2}, b_{j1}, b_{j2} \leq 1,\ a_{j1} + b_{j1} \leq 1,\ a_{j2} + b_{j2} \leq 1,\ a_{j1} + b_{j2} \leq 1.$

G.3 For $e_j \in WebEntry$, $a_{j1} = a_{j2} = 0$

G.4 For $P_v \in WebProc \bigcup BelowWebProc$, $0 \leq d_v \leq 1, f_v = k$

G.5 For $P_v \in WebProc \bigcup BelowWebProc$, $g_v \in WebGlobal(P_v)$, and $c_i \in C(P_v)$, $d_v + y_i \leq k$

G.6 For $P_v \in WebProc \bigcup BelowWebProc$, $g_v \in WebGlobal(P_v)$, and $e_i \in IN(P_v)$, $d_v + t_i \leq k$

G.7 For $e_j \in IN(P_v), P_v \in WebProc \bigcup BelowWebProc$, and $g_v \in WebGlobal(P_v)$, $d_v + b_{j2} = 1$.

G.8 For $e_j \in OUT(P_v), P_v \in WebProc \bigcup BelowWebProc$, and $g_v \in WebGlobal(P_v)$, $d_v + b_{j1} = 1$.

Figure 7.21: The constraints on the global candidates.

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

Figure 7.22: The constraints on the local candidates.

In each procedure $P_v \in WebProc \bigcup BelowWebProc$, there is a pair of dual variables $(d_v,\ f_v)$ for a global that can be allocated a register in the procedure. Constraint G.4 specifies the bounds on these dual variables. If $d_v + f_v = k + 1$ $(d_v = 1)$, then the global is allocated a register; otherwise, $d_v + f_v = k$ $(d_v = 0)$ and the global is not allocated a register. Constraints G.5 and G.6 limit the value of d_v by the number of registers allocated to the candidates in P_v (G.5) and the number of registers allocated on entrance to P_v (G.6).

If a global is allocated a register in procedure P_v then dual variable $d_v = 1$; otherwise, $d_v = 0$. The dual variables in G.1 and G.2 model the cost of register loads and stores of globals around calls. Constraints G.7 and G.8 relate the dual variables for each global candidate with the dual variables in G.1 and G.2.

Constraints D.1 – D.8 in Figure 7.22 are identical to the constraints for interprocedural register allocation of local candidates with spilling.

Figure 7.23 shows the objective function of our dual minimum cost flow problem for allocating registers to globals and locals. The spill costs are positive, since spilling fewer registers increases the value of the objective function. Parts 1 and 2 are identical to the

Objective Function

1. Maximize $\sum_{c_j \in C(P)} w_j * (x_j + y_j)$
2. $+ \sum_{e_j \in E} s_j * (r_j + t_j)$
3. $+ \sum_{e_j \in WebEdge} hs_j * (a_{j1} + b_{j1} + a_{j2} + b_{j2})$
4. $+ \sum_{e_j \in WebEntry} s_j * (a_{j1} + b_{j1})$
5. $+ \sum_{e_j \in WebEntry} hs_j * (a_{j2} + b_{j2})$
6. $+ \sum_{e_j \in BelowWebEdge} s_j * (a_{j1} + b_{j1} + a_{j2} + b_{j2})$
7. $+ \sum_{P_v \in WebGlobal(P)} w_v * (d_v + f_v)$

Figure 7.23: Objective function of the dual minimum cost flow problem.

objective function for allocating registers to local candidates. The value s_j is the cost of saving a register into a temporary before call e_j and loading the value back into a register after the call. The value hs_j for call edge e_j is the cost of a load/store or store/load of a global that references the global's home location.

Parts 3 through 7 represents the decrease in benefit from loads and stores of register-allocated globals. Part 3 measures the cost of a store/load or load/store of a global along an edge in a web. Globals are spilled into their home location within a web. Part 4 measures the cost of a store of a global into a temporary before entering a web and a load upon returning from it. Part 5 measures the cost of a load of a global from its home location upon entering a web and a store upon returning from the web. Part 6 measures the cost of storing a global into a temporary before a call and reloading it into a register after the call. The call is not incident on a procedure in a web. Part 7 represents the benefit associated with allocating a register to a global in a procedure. If $d_v + f_v = k + 1$, then a global is allocated a register in P_v; otherwise, $d_v + f_v = k$ and a global is not allocated a register in P_v. Allocating a register to global g_v in procedure P_v increases the benefit by w_v, the benefit of the global. For a procedure in $BelowWebProc$, the benefit of allocating a register to a global is 0.

We define solutions to the dual minimum cost flow problem as follows. Let $web = WebProc \bigcup BelowWebProc$ be the set of procedures in which there is a global register

candidate. Let $w\_edges = WebEdge \bigcup BelowWebEdge \bigcup WebEntry$ be the set of edges in which there can be register loads and stores of globals. We partition the dual variables in Figure 7.20 as follows. Let xy be the sequence of tuples

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

let rt be the sequence of tuples

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

let df be the sequence of tuples

$$((d_1, f_1), \ldots, (d_{|web|}, f_{|web|}));$$

and, let ab be the sequence of tuple pairs

$$(((a_{11}, b_{11}), (a_{12}, b_{12})) \ldots, ((a_{|w\_edges|\ 1}, b_{|w\_edges|\ 1}), (a_{|w\_edges|\ 2}, b_{|w\_edges|\ 2}))).$$

A solution to the dual minimum cost flow problem is represented by the tuple

$$u = (xy, rt, df, ab).$$

7.3.7 Mapping between the dual minimum cost flow problem and interprocedural register allocation of globals

For a call graph G on which we define a partial order ($\sqsubseteq$), and given k registers, let u be an assignment to the dual variables of the dual minimum cost flow problem in Section 7.3.6, but which does not necessarily maximize the objective function in Figure 7.23. Represent this set of solutions as $S(k, G)$. Let H be a solution to the interprocedural register allocation of local and global candidates in Section 7.3.5, but which does not necessarily maximize the function in Figure 7.15. Represent this set of solutions as $T(k, G)$. We define the mapping $H(u)$ from $u \in S(k, G)$ to $H \in T(k, G)$ in Figure 7.24.

Hu.1 $H(u) = (R(u), free\_in(u), free\_out(u))$.

Hu.2 For $1 \leq h \leq k$, $R_h(u) = \{c_j | x_j + y_j = k+1, x_j = h, c_j \in C(P)\} \bigcup$
$\{g_v | d_v + f_v = k+1, d_v = h, g_v \in WebGlobal(P)\}$.
$R(u) = (R_1(u), \ldots, R_k(u))$.

Hu.3 For $e_j \in E$, $free\_in_j(u) = r_j$.
$free\_in = (free\_in_1(u), \ldots, free\_in_{|E|}(u))$.

Hu.4 For $e_j \in E$, $free\_out_j(u) = k - t_j$.
$free\_out = (free\_out_1(u), \ldots, free\_out_{|E|}(u))$.

Figure 7.24: Mapping from solutions in $S(k, G)$ to solutions in $T(k, G)$.

The mapping in Figure 7.24 is similar to the mapping for interprocedural register allocation of locals with spilling except that now global candidates are also allocated registers. As mentioned earlier, we can determine where the loads and stores of globals should appear based on which procedures allocate a register to the globals.

Figure 7.25 defines the mapping $u(H)$ from $H \in T(k, G)$ to $u \in S(k, G)$. Mappings uH.1, uH.2, and uH.3 define the values of the dual variables representing local register candidates and the dual variables representing the number of register-allocated locals spilled along each edge. uH.1 – uH.3 are identical to the mapping of solutions of interprocedural register allocation of locals with spilling to solutions of the dual minimum cost flow problem in Section 5.3. In uH.4, we assign values to the global candidates' dual variables. In uH.5, uH.6, and uH.7, we define the dual variables used to model the cost of register loads and stores of globals around calls.

We first prove that for $u \in S(k, G)$, $H(u) \in T(k, G)$ and for $H \in T(k, G)$, $u(H) \in S(k, G)$. Let $\hat{S}(k, G)$ represent maximal solutions in $S(k, G)$. Let $\hat{T}(k, G)$ represent maximal solutions in $T(k, G)$. We will prove that there is a bijection $u(H)$ from $H \in \hat{T}(k, G)$ to $u \in \hat{S}(k, G)$, and $H(u)$ is its inverse. Let $T^*(k, G)$ be maximum weighted solutions in $T(k, G)$, and let $S^*(k, G)$ be maximum weighted solutions in $S(k, G)$. We show that $T^*(k, G) \subseteq \hat{T}(k, G)$ and $S^*(k, G) \subseteq \hat{S}(k, G)$. Next, we prove that for $H \in T^*(k, G)$, $u(H)$ is a bijection to $S^*(k, G)$, and $H(u)$ is its inverse.

uH.1 For $e_j \in E$, $r_j = free\_in_j$, $t_j = k - free\_out_j$.

uH.2 For $c_j \in C(P)$, if $c_j \in R_h$, then $x_j = h$; $y_j = k + 1 - x_j$ $(x_j + y_j = k + 1)$.

uH.3 For $c_j \in C(P), c_j \notin R$,

uH.3a If $c_j \sqsubset c_k$, for some $c_k \in R_h$, then $x_j = m - 1$, where m is the smallest value such that $c_j \sqsubset c_l$ and $c_l \in R_m$. $y_j = k - x_j$.

otherwise, assume $c_j \in P_v$

uH.3b For all edges $e_j \in IN(P_v)$, x_j equals the smallest value of $free\_out_j$. $y_j = k - x_j$.

uH.3c If P_v has no incident edges, then $x_j = k$ and $y_j = k - x_j = 0$.

uH.4 For $P_v \in WebProc \bigcup BelowWebProc$, let $g_v \in WebGlob(P_v)$. if $g_v \in R_1$, then $d_v = 1$, $f_v = k$; else $(g_v \notin R)$ $d_v = 0$, $f_v = k$.

uH.5 For $P_v \in WebProc \bigcup BelowWebProc$,

uH.5a For $e_j \in IN(P_v)$, if $d_v = 1$ then $b_{j2} = 0$ else $b_{j2} = 1$.

uH.5b For $e_j \in OUT(P_v)$, if $d_v = 1$ then $b_{j1} = 0$ else $b_{j1} = 1$.

uH.6 For $e_j \in WebEdge$, $a_{j1} = a_{j2} = 1 - MAX(b_{j1}, b_{j2})$.

uH.7 For $e_j \in WebEntry$, $a_{j1} = a_{j2} = 0$.

uH.8 For $e_j \in BelowWebEdge$, $a_{j1} = 1 - MAX(b_{j1}, b_{j2})$, $a_{j2} = 1 - b_{j2}$.

Figure 7.25: Mapping from $T(k, G)$ to $S(k, G)$.

Mapping solutions between $S(k,G)$ and $T(k,G)$

Theorem 12 *For any $u \in S(k,G)$, $H(u) \in T(k,G)$*

Solutions in $S(k,G)$ are constrained by D.1 – D.8 and G.1 – G.8. Solutions in $T(k,G)$ are constrained by I.1 – I.6 and J.1 – J.3. In Section 5.1, we proved constraints I.1 – I.6 follow from constraints D.1 – D.8. We now prove that constraints J.1 – J.3 follow from constraints D.1 – D.8 and G.1 – G.8.

Prove *(J.1)* Assume $P_v \in WebProc \bigcup BelowWebProc$. If global $g_v \in WebGlobal(P_v)$ and $g_v \in R$, then $g_v \in R_1$.

Proof: Let $P_v \in WebProc \bigcup BelowWebProc$, $g_v \in WebGlobal(P_v)$, and $g_v \in R$. In $S(k,G)$, $0 \leq d_v \leq 1$ and $f_v = k$. By mapping Hu.2, if $R(u)$ maps g_v to a register, then $d_v = 1$ and g_v is mapped to R_1. □

Prove *(J.2)* Assume $P_v \in WebProc \bigcup BelowWebProc$. Let $g_v \in WebGlobal(P_v)$ and $c_j \in C(P_v)$. If $g_v \in R_1$, $c_j \in R_q$, and $g_v \sqsubset c_j$, then $1 < q$.

Proof: Let $P_v \in WebProc \bigcup BelowWebProc$, $g_v \in WebGlobal(P_v)$, and $c_j \in C(P_v)$. Assume $R(u)$ map g_v to R_1 and c_j to R_q. By Hu.2, (1) $d_v = 1$, $x_j = q$, and $x_j + y_j = k + 1$. Thus, (2) $y_j = k + 1 - q$. By G.5, (3) $d_v + y_j \leq k$. Substituting (1) and (2) into (3) yields $1 + k + 1 - q \leq k$, which implies $2 \leq q$ and, thus, $1 < q$. □

Prove *(J.3)* Assume $P_v \in WebProc \bigcup BelowWebProc$. If $g_v \in WebGlobal(P_v)$, $g_v \in R_1$, and $e_j \in IN(P_v)$, then $1 \leq free\_out_j$.

Proof: Let $P_v \in WebProc \bigcup BelowWebProc$, $g_v \in WebGlobal(P_v)$, $e_j \in IN(P_v)$, and $R(u)$ maps g_v to R_1. By Hu.2, (1) $d_v = 1$. By Hu.4, $k - t_j = free\_out_j$ and, thus, (2) $t_j = k - free\_out_j$. By G.6, (3) $d_v + t_j \leq k$. Substituting (1) and (2) into (3) yields $1 + k - free\_out_j \leq k$ and, thus, $1 \leq free\_out_j$. □

Theorem 13 *For any $H \in T(k,G)$, $u(H) \in S(k,G)$*

As mentioned above, solutions in $S(k,G)$ are constrained by D.1 – D.8 and G.1 – G.8, and solutions in $T(k,G)$ are constrained by I.1 – I.6 and J.1 – J.3. In Section 5.1, we proved constraints D.1 – D.8 follow from constraints I.1 – I.6. We now prove that constraints G.1 – G.8 follow from constraints I.1 – I.6 and J.1 – J.3.

Prove (*G.1*) For $e_j \in WebEdge \bigcup WebEntry$, $0 \leq a_{j1}, a_{j2}, b_{j1}, b_{j2} \leq 1$, $a_{j1} + b_{j1} \leq 1$, $a_{j2} + b_{j2} \leq 1$, $a_{j1} + b_{j2} \leq 1$, $a_{j2} + b_{j1} \leq 1$.

Proof: By uH.5, $0 \leq b_{j1}, b_{j2} \leq 1$. By uH.6 and uH.7, $0 \leq a_{j1}, a_{j2} \leq 1$. Also, by uH.6 and uH.7, $a_{j1} + b_{j1} \leq 1$, $a_{j2} + b_{j2} \leq 1$, $a_{j1} + b_{j2} \leq 1$, and $a_{j2} + b_{j1} \leq 1$. □

Prove (*G.2*) For $e_j \in BelowWebEdge$, $0 \leq a_{j1}, a_{j2}, b_{j1}, b_{j2} \leq 1$, $a_{j1} + b_{j1} \leq 1$, $a_{j2} + b_{j2} \leq 1$, $a_{j1} + b_{j2} \leq 1$.

Proof: By uH.5, $0 \leq b_{j1}, b_{j2} \leq 1$. By uH.8, $0 \leq a_{j1}, a_{j2} \leq 1$. Also by uH.8, $a_{j1} + b_{j1} \leq 1$, $a_{j2} + b_{j2} \leq 1$, and $a_{j1} + b_{j2} \leq 1$. □

Prove (*G.3*) For $e_j \in WebEntry$, $a_{j1} = a_{j2} = 0$.

Proof: True by uH.7. □

Prove (*G.4*) For $P_v \in WebProc \bigcup BelowWebProc$, $0 \leq d_v \leq 1, f_v = k$.

Proof: By uH.4, $0 \leq d_v \leq 1$, $f_v = k$. □

Prove (*G.5*) For $P_v \in WebProc \bigcup BelowWebProc$, $g_v \in WebGlobal(P_v)$, and $c_i \in C(P_v)$, $d_v + y_i \leq k$.

Proof: Assume $P_v \in WebProc \bigcup BelowWebProc$, $g_v \in WebGlobal(P_v)$, and $c_i \in C(P_v)$.

1. Assume $g_v \notin R$. Then by uH.4, $d_v = 0$. By uH.2 and uH.3, $0 \leq y_i \leq k$. Thus, $d_v + y_i \leq k$.

2. Otherwise, let $g_v \in R_1$, $1 \leq q \leq k$. By uH.4, $d_v = 1$.

   (a) Assume $c_i \in R_q$, $1 \leq q \leq k$. By J.2, (1) $d_v (= 1) < q$. By uH.2, $y_i = k+1-x_i (= q)$ and, thus, (2) $q = k + 1 - y_i$. Substituting (2) into (1) yields $d_v < k + 1 - y_i$ and, thus, $d_v + y_i \leq k$.

   (b) Assume $c_i \notin R_q$, $1 \leq q \leq k$. Only if $y_i = k$, does $d_v + y_i \not\leq k$. If $y_i = k$, then $x_i = 0$ by uH.3. Also by uH.3, either (A) there exists some $c_m \in C(P_v)$, $c_i \sqsubset c_m$, such that $c_m \in R_1$ (uH.3a), or (B) there exist an $e_j \in IN(P_v)$ such that $free\_out_j = 0$ (uH.3b).

      Case A: By J.2, since $g_v \in R$ and $c_m \in R_1$, we have $1 < 1$, which is false. Thus, $c_m \notin R_1$. Hence, $x_i \neq 0$, and $d_v + y_i \leq k$.

      Case B: By J.3, since $g_v \in R$, $1 \leq free\_out_j$, for all $e_j \in IN(P_v)$. Hence $free\_out_j \neq 0$. □

Prove (*G.6*) For $P_v \in WebProc \bigcup BelowWebProc$, $g_v \in WebGlobal(P_v)$, and $e_i \in IN(P_v)$, $d_v + t_i \leq k$.

Proof: We assume that $P_v \in WebProc \bigcup BelowWebProc$, $g_v \in WebGlobal(P_v)$, and $e_i \in IN(P_v)$.

1. If $g_v \notin R$, then by uH.4, $d_v = 0$. By I.2 and uH.1, $t_i \leq k$. Thus, $d_v + t_i \leq k$.

2. If $g_v \in R$, then by uH.4, $d_v = 1$. We must show that $t_i \leq k - 1$ and, thus, $1 \leq k - t_i$. Assume $1 > k - t_i$. By uH.1, $1 > free\_out_i$. But by J.3, $1 \leq free\_out_i$. □

Prove (*G.7*) For $P_v \in WebProc \bigcup BelowWebProc$, $g_v \in WebGlobal(P_v)$, and $e_j \in IN(P_v)$, $d_v + b_{j2} = 1$.

Proof: We assume that $P_v \in WebProc \bigcup BelowWebProc$, $g_v \in WebGlobal(P_v)$, and $e_j \in IN(P_v)$. By uH.4 and uH.5a, $d_v + b_{j2} = 1$. □

Prove (*G.8*) For $P_v \in WebProc \bigcup BelowWebProc$, and $g_v \in WebGlobal(P_v)$, and $e_j \in OUT(P_v)$, $d_v + b_{j1} = 1$.

Proof: We assume that $P_v \in WebProc \bigcup BelowWebProc$, $g_v \in WebGlobal(P_v)$, and $e_j \in OUT(P_v)$. By uH.4 and uH.5b, $d_v + b_{j1} = 1$. □

Maximal Solutions in $T(k, G)$ and $S(k, G)$

A solution in $S(k, G)$ is maximal if the values of the dual variables cannot be increased. We represent the maximal solutions in $S(k, G)$ as $\hat{S}(k, G)$. A maximal solution does not necessarily maximize the objective function in Figure 7.23.

We define $\hat{T}(k, G)$ to be the maximal solutions in $T(k, G)$. In $\hat{T}(k, G)$, additional local candidates cannot be allocated registers given the local and global candidates allocated registers and the register spills of local candidates across the call edges. In addition, a register spill of a local cannot be removed from a call edge given the local and global candidates allocated registers in each procedure and the other register spills of locals in the call graph.

Whether an allocation in $T(k, G)$ is maximal is independent of the global candidates that are allocated registers. Based on our model of interprocedural register allocation of globals in Section 7.3.5, we cannot assign a register to a global candidate without

changing the loads and stores of that global along the call edges. Each procedure $P_v \in WebProc \bigcup BelowWebProc$ may allocate its global candidate to a register. Each possible register allocation of global candidates, in which global candidates can only be assigned register R_1, occurs as part of an allocation in $\hat{T}(k, G)$.

Theorems 3, 4, 5, and 6 and Corollary 1 of Section 5.2 still hold in the presence of global candidates; that is, for $z \in \hat{S}(k, G)$. We list the theorems below.

Theorem 3 Let $z \in \hat{P}(k, G)$. Let $c_i \in C(P)$. Then $x_i + y_i = k$ or $x_i + y_i = k + 1$.

The proof of Theorem 3 can easily be updated for $z \in \hat{S}(k, G)$. In the proof, either dual variable x_j of candidate c_j, $c_j \sqsubset c_i$, or dual variable r_j for edge $e_j \in OUT(P_v)$ may constrain the value of y_i. In the presence of global candidates, a dual variable d_v of a global candidate g_v may also constrain the value of y_i. No other changes to the proof need to be made. □

Theorem 4 Let $z \in \hat{P}(k, G)$, $c_i, c_j \in C(P)$. If $c_i \sqsubset c_j$, then $x_i \leq x_j$.

This proof can easily be generalized to allow local candidate c_i to be replaced by a global candidate g_v. A global candidate is a least-most element in a partial order. In the proof, replace x_i and y_i with d_v and f_v. □

Corollary 1 Let $z \in \hat{P}(k, G)$, $c_i, c_j \in C(P)$. If $c_i \sqsubset c_j$ and $c_j \in R_v(z)$ for some $1 \leq v \leq k$, then $x_i < x_j$.

Given the change to Theorem 4, this corollary holds if c_i is replaced by a global candidate g_v. □

Theorem 5 Assume $c_i \in C(P_v)$ and $c_i \notin R_h(z)$ for some $1 \leq h \leq k$. Let $x_i = h$. If $z \in \hat{P}(k, G)$, then

1. if there exists a $c_m \in R_v(z)$ from some $1 \leq v \leq k$, $c_i \sqsubset c_m$, then there exists a $c_q \in R_{h+1}(z)$, $c_i \sqsubset c_q$.

2. Otherwise, assume there does not exist a $c_m \in R_v(z)$, $c_i \sqsubset c_m$. If $IN(P_v)$ is not empty, then $x_i = k - t_m$ for some $e_m \in IN(P_v)$.

3. Otherwise, if there does not exist an $e_j \in IN(P_v)$, $x_i = k$.

No changes need to be made to the proof of Theorem 5. □

Theorem 6 Assume $c_i \in C(P_v)$, $x_i = h$, $c_i \notin R_h(z)$, and $0 \leq h \leq k$. If $z \in \hat{P}(k, G)$, then

1. if there exists a candidate $c_j \in C(P_v)$, such that $c_j \sqsubset c_i$ and $c_j \in \bigcup_{v=1}^{k} R_v(z)$, then there is a candidate $c_m \sqsubset c_i$ such that $c_m \in R_h(z)$

2. Otherwise, assume there is no candidate $c_j \sqsubset c_i$ such that $c_j \in \bigcup_{v=1}^{k} R_v(z)$. If $OUT(P_v)$ is not empty, then $r_g = x_i$ for some $e_g \in OUT(P_v)$.

3. Otherwise, if there does not exist an $e_g \in OUT(P_v)$, then $x_i = 0$.

For $z \in \hat{S}(k, G)$, Theorem 6 part (1) also holds if c_j is a global candidate, in which case c_m may also be a global candidate. Parts (2) and (3) still hold. □

We now prove that for $H \in \hat{T}(k, G)$, $u(H)$ is a bijection onto $\hat{S}(k, G)$, and $H(u)$ is its inverse.

Theorem 14 *If $H \in \hat{T}(k, G)$, then $u(H) \in \hat{S}(k, G)$.*

Proof: Let $u' = u(H)$ for $H \in \hat{T}(k, G)$. Assume local candidate c_i is assigned to R_h, $1 \leq h \leq k$, in H. In u', $x_i = h$ and $y_i = k + 1 - h$. Assume global candidate g_v is assigned to register R_1 in H. Then $d_v = 1$ in u'. Mapping $H(u') \in T(k, G)$ assigns candidate c_i to register R_h and global candidate g_v to register R_1. For local candidates c_i not assigned a register in H, $x_i + y_i = k$ in u' and for global candidates g_v not assigned a register in H, $d_v = 0$ in u'. These candidates are not mapped to registers in $H(u')$. In addition, in u', $r_j = free\_in_j$ and $t_j = k - free\_out_j$ for $e_j \in E$. Mapping $H(u')$ assigns r_j to $free\_in_j$ and $k - t_j$ to $free\_out_j$. Thus, H is equivalent to $H(u')$.

If $u' = u(H)$ is not maximal, then we can increase the value of a dual variable to make the solution maximal. If we increase the value of dual variables x_i or y_i for candidate $c_i \in C(P)$ then the allocation of $H(u')$ will allocate a register to an additional local candidate. We cannot increase the value of dual variable d_v for global candidate $g_v \in WebGlobal(P_v)$, since

d_v is constrained by b_{j1} for $e_j \in IN(P_v)$ (G.7) and by b_{j2} for $e_j \in OUT(P_v)$ (G.8). Dual variable f_v always equals k (G.4). Increasing r_j and t_j for $e_j \in E$ in u' results in fewer registers spilled in $H(u')$. If we can increase dual variables in u', then $H(u')$ is maximal, and H is not maximal, and H and $H(u')$ are not equivalent, which is a contradiction. □

Theorem 15 *If* $u \in \hat{S}(k, G)$, *then* $H(u) \in \hat{T}(k, G)$.

Let $u \in \hat{S}(k, G)$. For $H(u) \in \hat{T}(k, G)$, we prove the following: (1) if $c_i \notin \bigcup_{v=1}^{k} R_v(u)$, then c_i cannot be added to $R_h(u)$, $1 \leq h \leq k$; (2) the value of $free\_in_i$ cannot be increased for $e_i \in E$; and, (3) the value of $free\_out_i$ cannot be decreased for $e_i \in E$.

(1) Let $c_i \in C(P_v)$ and $c_i \notin \bigcup_{v=1}^{k} R_v(u)$. Follow the same steps of (1) in Theorem 8, Section 5.2.

(2) We prove that for $e_i \in E$, the value of $free\_in_i$ cannot be increased in $H(u)$.

Again, we can prove (2) by following the same steps of (2) in Theorem 8, Section 5.2.

(3) We prove that for $e_i \in E$, $free\_out_i$ cannot be decreased, if u is maximal. Let $t_i = k - h$, $e_i \in IN(P_v)$. $H(u)$ maps the value of r_i to $free\_in_i$ and $k - t_i$ to $free\_out_i$. There are four constraints on t_i in u: (a) $t_i \leq k - r_i$; (b) if $e_j \in OUT(P_v)$, then $t_i \leq k - r_j$; (c) if $c_j \in C(P_v)$, then $t_i \leq k - x_j$; and, (d) if $g_v \in WebGlobal(P_v)$, then $d_v + t_i \leq k$. Since u is maximal, t_i must be equal to one or more of these constraints. $H(u)$ maps these constraints to the four constraints on $free\_out_i$ in $H \in T(k, G)$.

To prove 3(a-c), we can follow the proof of 3(a-c) in Theorem 8 of Section 5.2. We prove 3(d) below.

3(d) Assume that $d_v + t_i = k$. By G.4, $0 \leq d_v \leq 1$. If $d_v = 0$, then $t_i = k$. Since $0 \leq t_i \leq k$, d_v does not constrain t_i. If $d_v = 1$, then $t_i = k - 1$ and, thus, $k - t_i = 1$. By Hu.4, $free\_out_i = 1$. By J.3, $1 \leq free\_out_i$ and, thus, $free\_out_i$ cannot be decreased. □

Theorem 16 *If* $H \in \hat{T}(k, G)$, $u \in \hat{S}(k, G)$ *and* $H(u) = H$, *then* $u = u(H)$.

We can follow the same steps in the proof of Theorem 9 in Section 5.2 to show that the dual variables x_i and y_i for $c_i \in C(P)$ and r_j and t_j for $e_j \in E$ are preserved across the mappings. We still must show that the values of the remaining dual variables, a_{j1}, a_{j2}, b_{j1}, b_{j2} for $e_j \in WebEdge \bigcup BelowWebEdge \bigcup WebEntry$ and d_v, f_v for $P_v \in WebProc \bigcup BelowWebProc$ are also preserved.

Let $P_v \in WebProc \bigcup BelowWebProc$. Let $g_v \in WebGlobal(P_v)$. Assume that in $u \in \hat{S}(k,G)$, $d_v = 1$ and $f_v = k$. Then by Hu.2, $g_v \in R_1$ in H. By uH.4, $u(H)$ assigns $d_v = 1$ and $f_v = k$, which are their values in u. Assume $d_v = 0$ and $f_v = k$. Then by Hu.2, $g_v \notin R_1$ in H. By uH.4, $u(H)$ assigns $d_v = 0$ and $f_v = k$, which are their values in u.

Assume $e_j \in WebEdge$. Let e_j be the call edge from procedure P_v to P_w ($e_j \in OUT(P_v)$ and $e_j \in IN(P_w)$). Let g_v be the global candidate in P_v and g_w be the global candidate in P_w. Also, let d_v and f_v be the dual variables for g_v and d_w and f_w be the dual variables for g_w. By G.7, $d_w + b_{j2} = 1$ in u. In $u(H)$ d_w is assigned its original value as discussed above. By uH.5a, b_{j2} is assigned its original value in u. Similarly by G.8 and uH.5b, b_{j1} is assigned its original value in u. By G.1, a_{j1} and a_{j2} have the same constraints on b_{j1} and b_{j2} in u. The constraints $a_{j1} + b_{j1} \leq 1$ and $a_{j1} + b_{j2} \leq 1$ imply $a_{j1} = 1 - MAX(b_{j1}, b_{j2})$ in a maximal solution. Since the values of b_{j1} and b_{j2} are maintained across the mapping and by uH.6, so are a_{j1} and a_{j2}.

We can follow a similar argument to prove that the value of dual variables a_{j1}, a_{j2}, b_{j1}, and b_{j2} are maintained across mappings for edges $e_j \in WebEntry \bigcup BelowWebEdge$. □

Theorem 17 *If $u \in \hat{S}(k,G), H \in \hat{T}(k,G)$, and $u(H) = u$, then $H = H(u)$.*

Assume $H \neq H(u)$. Let $H' = H(u)$. By Theorem 16, $u(H') = u$. Since $u(H) = u$ by assumption, $u(H') = u = u(H)$. By mapping uH.1 - uH.8, if $H \neq H'$, then $u(H) \neq u(H')$. Hence, $H = H'$ and $H = H(u)$. □

Maximum Weight Solutions in $T(k,G)$ and $S(k,G)$

We now prove there exists a bijection between maximum weight solutions in $T(k,G)$ and $S(k,G)$.

Corollary 4 *For $H \in T^*(k,G)$, $u(H)$ is a bijection onto $S^*(k,G)$ and $H(u)$ is its inverse.*

Proof: For solutions in $\hat{S}(k,G)$ the values of the dual variables cannot be increased. The objective function is shown in (1) below. The value of dual variable d_v for global g_v is dependent on b_{j2} for $e_j \in IN(P_v)$, since $d_v + b_{j2} = 1$. Thus, even though w_v can equal 0 for d_v in a maximum weighted solution, the value of dual variable d_v cannot be decreased.

Since the weights of the dual variables other than the global candidates are always positive, the maximum weighted solution is also a maximal solution.

The maximization function for a solution to the interprocedural register allocation problem is shown in (6). A maximum weighted solution $T^*(k, G)$ is also a maximal solution. Since $w_i > 0$ for $c_i \in C(P)$, additional local candidates cannot be allocated registers in a maximum weighted solution. The expression $-s_j * (free\_out_j - free\_in_j)$ decreases for each register spill along edge e_j. Both a maximum weighted solution and a maximal solution will not spill registers unnecessarily along an edge. We defined a maximal solution to be independent of the global candidates allocated registers. The placement of the loads and stores of globals is dependent on the global candidates allocated registers. Hence, the placement of loads and stores of globals in parts (c) and (d) of (6) are also part of a maximal solution. Thus, $T^*(k, G) \subseteq \hat{T}(k, G)$.

$$
\begin{aligned}
&\text{(a)} && \textstyle\sum_{c_j \in C(P)} w_j * (x_j + y_j) \\
&\text{(b)} && + \textstyle\sum_{e_j \in E} s_j * (r_j + t_j) \\
&\text{(c)} && + \textstyle\sum_{e_j \in WebEdge} hs_j * (a_{j1} + b_{j1} + a_{j2} + b_{j2}) \\
&\text{(d)} && + \textstyle\sum_{e_j \in WebEntry} s_j * (a_{j1} + b_{j1}) \\
&\text{(e)} && + \textstyle\sum_{e_j \in WebEntry} hs_j * (a_{j2} + b_{j2}) \\
&\text{(f)} && + \textstyle\sum_{e_j \in BelowWebEdge} s_j * (a_{j1} + b_{j1} + a_{j2} + b_{j2}) \\
&\text{(g)} && + \textstyle\sum_{g_v \in WebGlobal(P)} w_v * (d_v + f_v)
\end{aligned}
\qquad (1)
$$

We now show that expression (1) differs from expression (6) by a constant through a series of transformations. First, to generate expression (2) from (1) we subtract constants from parts (a) – (g) of (1).

$$
\begin{aligned}
&\text{(a)} && \textstyle\sum_{c_j \in C(P)} w_j * (x_j + y_j) - \sum_{c_j \in C(P)} w_j * k \\
&\text{(b)} && + \textstyle\sum_{e_j \in E} s_j * (r_j + t_j) - \sum_{e_j \in E} s_j * k \\
&\text{(c)} && + \textstyle\sum_{e_j \in WebEdge} hs_j * (a_{j1} + b_{j1} + a_{j2} + b_{j2} - 1) - \sum_{e_j \in WebEdge} 2 * hs_j \\
&\text{(d)} && + \textstyle\sum_{e_j \in WebEntry} s_j * (a_{j1} + b_{j1}) - \sum_{e_j \in WebEntry} s_j \\
&\text{(e)} && + \textstyle\sum_{e_j \in WebEntry} hs_j * (a_{j2} + b_{j2}) - \sum_{e_j \in WebEntry} hs_j
\end{aligned}
$$

$$
\begin{aligned}
&\text{(f)} && + \textstyle\sum_{e_j \in BelowWebEdge} s_j * (a_{j1} + b_{j1} + a_{j2} + b_{j2}) - \sum_{e_j \in BelowWebEdge} 2 * s_j \\
&\text{(g)} && + \textstyle\sum_{g_v \in WebGlobal(P)} w_v * (d_v + f_v) - \sum_{g_v \in WebGlobal(P)} w_v * k
\end{aligned} \tag{2}
$$

To arrive at expression (3), we simplify each part.

$$
\begin{aligned}
&\text{(a)} && \textstyle\sum_{c_j \in C(P)} w_j * (x_j + y_j - k) \\
&\text{(b)} && + \textstyle\sum_{e_j \in E} s_j * (r_j + t_j - k) \\
&\text{(c)} && + \textstyle\sum_{e_j \in WebEdge} hs_j * (a_{j1} + b_{j1} + a_{j2} + b_{j2} - 2) \\
&\text{(d)} && + \textstyle\sum_{e_j \in WebEntry} s_j * (a_{j1} + b_{j1} - 1) \\
&\text{(e)} && + \textstyle\sum_{e_j \in WebEntry} hs_j * (a_{j2} + b_{j2} - 1) \\
&\text{(f)} && + + \textstyle\sum_{e_j \in BelowWebEdge} s_j * (a_{j1} + b_{j1} + a_{j2} + b_{j2} - 2) \\
&\text{(g)} && + \textstyle\sum_{g_v \in WebGlobal(P)} w_v * (d_v + f_v - k)
\end{aligned} \tag{3}
$$

Expression (4) follows from (3) since local candidates in part (a) and global candidates in part (g) are allocated registers if their dual variables sum to $k+1$; otherwise, their dual variables sum to k and these candidates are not allocated registers.

$$
\begin{aligned}
&\text{(a)} && \textstyle\sum_{c_j \in R} w_j \\
&\text{(b)} && + \textstyle\sum_{e_j \in E} s_j * (r_j + t_j - k) \\
&\text{(c)} && + \textstyle\sum_{e_j \in WebEdge} hs_j * (a_{j1} + b_{j1} + a_{j2} + b_{j2} - 2) \\
&\text{(d)} && + \textstyle\sum_{e_j \in WebEntry} s_j * (a_{j1} + b_{j1} - 1) \\
&\text{(e)} && + \textstyle\sum_{e_j \in WebEntry} hs_j * (a_{j2} + b_{j2} - 1) \\
&\text{(f)} && + \textstyle\sum_{e_j \in BelowWebEdge} s_j * (a_{j1} + b_{j1} + a_{j2} + b_{j2} - 2)
\end{aligned} \tag{4}
$$

In parts (b), (c), (d), (e), and (f) of (4), we multiply both the left hand side of the summation and right hand-side by -1 to form expression (5). By Hu.3 and Hu.4, $k - r_j - t_j$ equals $free\_out_j - free\_in_j$ in part (b). In part (c) $a_{j1} + b_{j1} + a_{j2} + b_{j2}$ equals 1 if there is a load and store of a global along e_j and 2 otherwise, as shown in Section 7.3.6. Thus, $-hs_j * (2 - (a_{j1} + b_{j1} + a_{j2} + b_{j2}))$ equals 0 if there is no load and store and $-hs_j$ otherwise.

Similarly, in part (f), if there is a load and store along an edge, then that edge contributes $-s_j$; otherwise, the edge contributes 0. In parts (d) and (e), an edge with a load and store contributes $-s_j$ and $-hs_j$, respectively.

$$
\begin{array}{ll}
\text{(a)} & \sum_{c_j \in R} w_j \\
\text{(b)} & -\sum_{e_j \in E} s_j * (free\_out_j - free\_in_j) \\
\text{(c)} & -\sum_{e_j \in WebEdge} hs_j * (2 - (a_{j1} + b_{j1} + a_{j2} + b_{j2})) \\
\text{(d)} & -\sum_{e_j \in WebEntry} s_j * (1 - (a_{j1} + b_{j1})) \\
\text{(e)} & -\sum_{e_j \in WebEntry} hs_j * (1 - (a_{j2} + b_{j2})) \\
\text{(f)} & -\sum_{e_j \in BelowWebEdge} s_j * (2 - (a_{j1} + b_{j1} + a_{j2} + b_{j2}))
\end{array} \tag{5}
$$

If there are loads and stores along an edge in (d) and (f) of (5), then these edges are members of set $GlobTempSt$ in the maximization function (6) of our interprocedural register allocation problem. If there are loads and stores along edges in (c) and (e) of (5), then these edges are members of the set $GlobHomeLdSt$ in our maximization function. Since there is a cost of $-s_j$ for loads and stores along edges in (d) and (f) of (5), and $-hs_j$ for loads and stores along edges in (c) and (e), the value of our maximization function (6) equals the value of expression (5).

$$
\begin{array}{ll}
\text{(a)} & \sum_{c_i \in R} w_i \\
\text{(b)} & -\sum_{e_j \in E} s_j * (free\_out_j - free\_in_j) \\
\text{(c)} & -\sum_{e_j \in GlobTempSt} s_j \\
\text{(d)} & -\sum_{e_j \in GlobHomeLdSt} hs_j .
\end{array} \tag{6}
$$

Thus, our objective function (1) differs from our maximization function (6) by a constant for maximal solutions in $S(k, G)$ and $T(k, G)$. Since by Theorems 16 and 17 we have a bijection from maximal solutions between both of these sets, the same mapping applies to maximum weight solutions in $S(k, G)$ and $T(k, G)$. Thus, for $H \in T^*(k, G)$, $u(H)$ is a bijection onto $S^*(k, G)$ and $H(u)$ is its inverse.

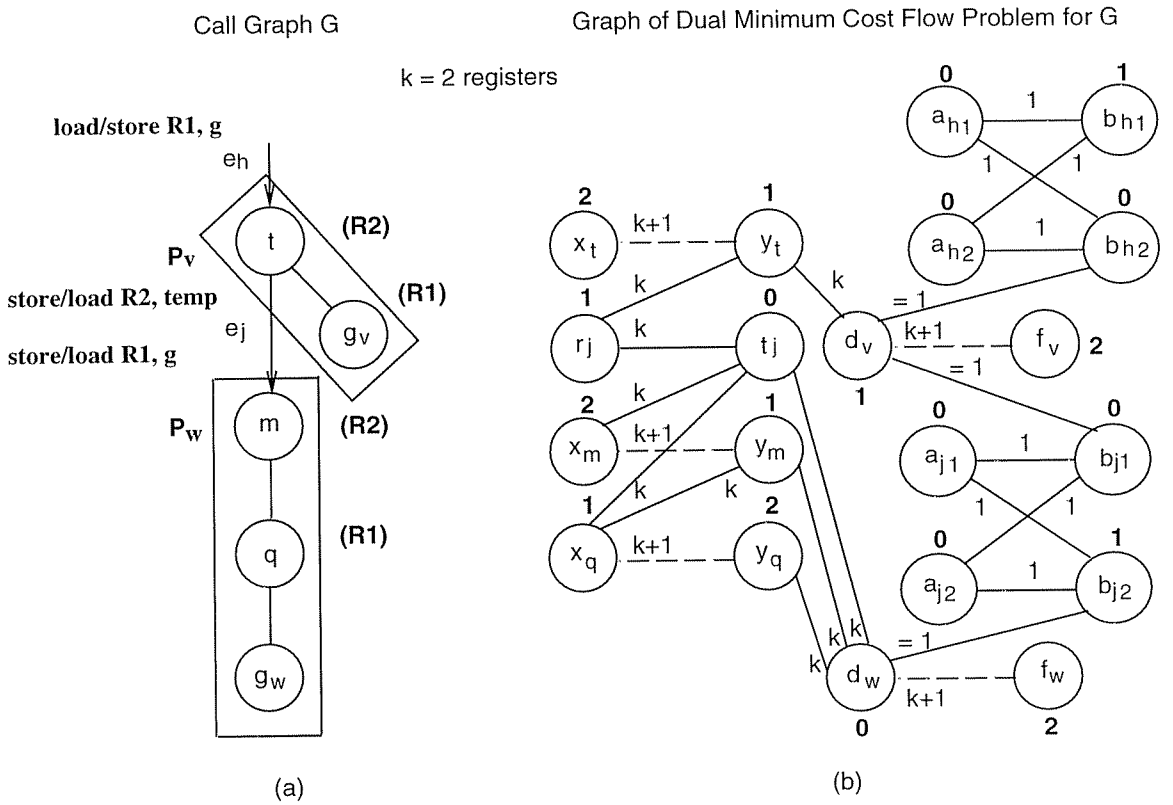


Figure 7.26: (a) shows a call graph with a register allocation and assignment to the candidates. Procedures P_v references an instance of global g (g_w) and P_w references an instance of global g (g_v). Candidates t, m, and q are local. A web for global g includes procedures P_v and P_w. (b) shows an assignment to the dual minimum cost flow problem that represents the register allocation and assignment in (a).

7.3.8 Example

Figure 7.26(a) presents a call graph G. There are two registers available for allocation. Procedures P_v and P_w are in the same web that reference global g. Let edge e_h represent the call to procedure P_v. Local candidate t and global candidate g are allocated registers in procedure P_v. We assume that the caller of P_v does not allocate a register to global g. We add a register *load/store* of global g along the call edge e_h. Local candidates m and q are allocated register in P_w. Register R_1, assigned to g in P_v, and register R_2, assigned to t in P_v, are spilled around the call to P_w.

Figure 7.26(b) shows the graph of a dual minimum cost flow problem based on G. The assignments to the dual variables in (b) corresponds to the allocation in (a). For $t \in C(P_v)$, $x_t(=2) + y_t(=1) = k+1$. Thus, we assign register R_2 to t. For $g_v \in Glob(P_v)$, $d_v(=1) + f_v(=2) = k+1$. Candidate g_v is assigned R_1. Along edge e_j, $r_j = 1$ and $t_j = 0$. Based on the mapping in Figure 7.24, $free\_in_j = 1$ (candidate g_v does not constrain the free registers available on entry to e_j) and $free\_out_j = 2$. As in interprocedural register allocation with spilling, we spill a register R_i along edge e_j if $free\_in_j < i \leq free\_out_j$. Thus, we spill register R_2 along e_j.

Dual variables $x_m + y_m = k+1$ and $x_q + y_q = k+1$, so candidates m and q are allocated registers in P_w. As there are no remaining registers, g is not allocated a register in P_w ($d_w + f_w = k$).

Now we examine the set of dual variables modeling the spill cost of globals along edge e_h—$(a_{h1}, b_{h1}, a_{h2}, b_{h2})$—and along edge e_j— $(a_{j1}, b_{j1}, a_{j2}, b_{j2})$. Each set has a single dual variable equal to 1, as $b_{h1} \neq b_{h2}$ and $b_{j1} \neq b_{j2}$. Thus, there are loads and stores along each edge. Since $b_{h1} \neq b_{h2}$ and g is allocated a register in P_v, a load/store of g is needed along edge e_h. Since $b_{j1} \neq b_{j2}$ and g is allocated a register in P_v, the register assigned to g is spilled along edge e_j.

7.3.9 Implementation

We implemented the interprocedural register allocator discussed in the previous section on a DECstation 5000/125. Interprocedural register allocation was performed on both user and

library routines. Dynamic profile information was provided to our interprocedural register allocator. We profiled input generating shorter execution times than the standard input for all benchmarks except *nasa7* and *swm256*, in which we have only one input file. We assume the cost of spilling a register into a temporary is two (one for a load and one for a store), but the cost of spilling a register-allocated global into its home location is four—an extra instruction is needed for both the load and store to form the 32-bit address of its home location [KH92].

Calls to *setjmp* and *longjmp* require special attention. After executing a *longjmp*, execution resumes after a previous *setjmp*. After a *setjmp*, we load the values of a web's register-allocated globals from their home locations. To be sure that we load the globals' current values, we store the globals' values into their home locations before executing the *setjmp* and *longjmp*.

We want to avoid executing a *setjmp* or *longjmp* in a procedure in $BelowWebProc$, since in general we do not know which global may be register-allocated in the procedure. To generate correct code, we extend webs downwards in the call graph to include occurrences of *setjmp* and *longjmp*.

Figure 7.27 presents the performance improvement of interprocedural register allocation of globals partitioned into webs. Numbers in parentheses represent the improvement over allocating registers to globals throughout the entire call graph (Section 7.2.2). Dashes indicate that no globals are allocated registers.

For smaller benchmarks, allocating registers to globals in webs can generate an allocation similar to allocating registers to globals throughout the entire call graph. Both approaches promote the same global candidates to registers in *compress*. In both allocations of *compress*, globals are register-allocated in the two most-frequently referenced procedures, which represent over 95% of the dynamic execution.

Benchmark *doduc* allocates registers to many global candidates, but most of these globals are referenced in infrequently called routines. An interprocedural register allocator that sets aside registers to be allocated only to globals would generate a poor allocation on *doduc*. Allocating registers to webs in benchmark *eqntott* does shows an improvement. In *eqntott*, a register is allocated to global *qsz*, which is referenced in routines *qsort* and *qst*.

Performance Improvement					
benchmark	*without globals*	*with globals*		*webs*	
				integer	floating-point
compress	1%	5%	(+0%)	7	0
doduc	3%	4%	(+2%)	14	10
ear	0%	0%	(+0%)	3	4
eqntott	0%	1%	(+0%)	4	0
espresso	7%	5%	(-2%)	5	0
fpppp	—	—	—	—	—
gcc	8%	8%	(+0%)	11	0
hydro2d	0%	0%	(+2%)	2	13
mdljdp2	0%	12%	(+4%)	7	8
mdljsp2	1%	2%	(+1%)	4	3
nasa7	0%	0%	(+0%)	6	0
ora	1%	6%	(+0%)	3	5
sc	8%	8%	(+0%)	1	1
spice	2%	1%	(-1%)	2	4
su2cor	-1%	-1%	(+0%)	3	2
swm256	1%	2%	(+1%)	2	4
xlisp	11%	13%	(-1%)	6	0

Figure 7.27: Performance improvement of interprocedural register allocation with and without allocating registers to globals in webs. Dashes indicate that no global candidates are allocated registers. Numbers in parentheses represent the improvement of interprocedural register allocation of globals in webs over allocating registers to globals throughout the entire call graph.

The fine-grained approach of allocating registers to globals in webs improves the performance of benchmark *mdljdp2* more than our other approach. Additional globals are allocated registers in frequently called routines in *mdljdp2*.

Allocating registers to globals in webs improves the performance of *xlisp* by only 2%; however, allocating registers to globals throughout the entire call graph improves performance by 3%. A reason for the performance difference is that global *xlstack* is allocated a register in the latter approach, but not the former. When evaluating which webs to include in an allocation, the fine-grained allocator divides the number of global references in a web by the number of procedures in the web. Since global *xlstack* is referenced in many routines, it has a relatively low benefit. Instead of allocating a register to global *xlstack*'s web, a register is allocated to webs with fewer procedures. The total number of global references in these webs does not equal the number of references to *xlstack*.

On benchmarks *espresso* and *spice* our interprocedural register allocator performs worse than an interprocedural register allocation without allocation of globals. For these benchmarks, we incorrectly estimate the number of global references. To estimate the number of global and register references in a procedure, we statically count the number of global and register references and scale these numbers by the dynamic instruction count of the procedure. Some globals appear frequently, but are infrequently referenced. The assignment of registers to locals is more uniform, since a register can be assigned to multiple locals in a procedure and, hence, can be better estimated by our approach.

Figure 7.28 presents the time solving the network flow problem for allocating registers to webs as a percentage of the total compilation time without interprocedural register allocation. As shown in Figure 7.28, our approach is slow, especially for integer benchmarks. The lengthy allocation time is due to both iterating through the set of registers and the variables and constraints for register allocation of globals.

7.4 Conclusions

In this chapter, we have presented two models of interprocedural register allocation of globals. The first model assumes registers are allocated to globals throughout the entire call

benchmark	*% of compilation time*	
	floating-point	integer
compress	0.2%	6.4%
doduc	3.1%	5.1%
ear	1.9%	1.7%
eqntott	0.1%	7.3%
espresso	3.5%	14.8%
fpppp	0.4%	1.5%
gcc	4.3%	87.0%
hydro2d	0.6%	15.1%
mdljdp2	2.7%	11.6%
mdljsp2	1.9%	12.2%
nasa7	1.2%	9.3%
ora	3.3%	1.7%
sc	1.7%	8.1%
spice	2.5%	5.9%
su2cor	2.2%	14.2%
swm256	3.0%	4.4%
xlisp	2.8%	20.3%

Figure 7.28: Time solving the network flow problem as a percentage of the total compilation time without interprocedural register allocation.

graph. This model is easily implemented as part of our network flow approach for finding a minimum cost interprocedural register allocation of locals with spilling. Allocating registers to global candidates over the entire call graph yields significant performance improvements on a couple of benchmarks. On other benchmarks, partitioning globals over the entire call graph is too large of a granularity to improve performance.

The second model of interprocedural register allocation allows a single register to be shared between a global candidate and local candidates and allows all other registers to be allocated only to local candidates. Running this allocator multiple times, mapping the shared register to different hardware registers, enables globals referenced in the same procedure to be allocated registers. This approach to interprocedural register allocation is added to our network flow-based interprocedural register allocator. Overall, this model yields better performance improvements than the first model, but is significantly slower. An intermediate approach that is more fine-grained than the first model, but can simultaneously allocate registers to globals referenced in the same procedure, is worthy of investigation.

Chapter 8

Future Work

This thesis raises interesting issues for further investigation. For example, we assume that local candidates are live across all calls or no calls. A more context-sensitive interprocedural register allocator could generate a better allocation. Currently, we use a number to represent the registers free in a procedure. If there are n registers free, then $R_1, \ldots, R_n$ are the free registers. Modeling registers as live across only some calls suggests that a single number is insufficient to represent the set of free registers. For example, five registers may be free in two procedures, but they may not be the same five registers.

A solution may involve having k copies of each candidate, where k is the number of registers. Constraints would prevent a candidate from being assigned to more than one register.

Finding a better interprocedural register allocation for local candidates may lead to a better solution for global candidates. Our model of interprocedural register allocation that allows spilling of register-allocated globals assumes at most one global candidate in each procedure. If we have multiple global candidates in a procedure, in which each global candidate can be assigned a pre-defined register, then an arbitrary set of registers may be allocated within a procedure.

Another area of future research involves parameters passing. There is a tradeoff in using registers to pass parameters. If a caller passes a parameter in a register to a callee then that register cannot hold values live across the call in the caller. However, passing values

in parameters can save instructions, since the caller may execute one less instruction by not saving the parameter's value on the stack and the callee may then avoid loading the parameter into a register from memory. A model that can accurately represent this cost would allow for a better allocation.

Bibliography

[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.

[BDM87] A. D. Berenbaum, D. R. Ditzel, and H. R. McLellan. Architectural innovations in the CRISP microprocessor. In *Spring 1987 COMPCON Digest of Papers*, pages 91–95, February 1987.

[Ber91] Dmitri P. Bertsekas. *Linear Network Optimization.* The MIT Press, 1991.

[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.

[BL93] Thomas Ball and James R. Larus. Branch prediction for free. In *Proceedings of SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 300–313, June 1993.

[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.

[Cho88] Fred C. Chow. Minimizing register usage penalty at procedure calls. In *Proceedings of SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 85–94, June 1988.

[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.

[DM82] David R. Ditzel and H. R. McLellan. Register allocation for free: The C machine stack cache. In *Proceedings of the SIGARCH/SIGPLAN Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 48–56, March 1982.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[Hen84] John L. Hennessy. VLSI processor architecture. *IEEE Transactions on Computers*, pages 1221–1246, December 1984.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 1990.

[KH92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture.* Prentice Hall, 1992.

[Moo85] David A. Moon. Architecture of the symbolics 3600. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 76–83, June 1985.

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

[Pat85] David A. Patterson. Reduced instruction set computers. *Communications of the ACM*, pages 8–21, January 1985.

[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.

[[Sp] Systems Performance Evaluation Cooperative, c/o Waterside Associates, Fremont, CA 1992.

[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.

[Wal88] David W. Wall. Register windows vs. register allocation. In *Proceedings of SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 67–78, July 1988.

[Wal91] David W. Wall. Predicting program behavior using real or estimated profiles. In *Proceedings of SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 59–70, June 1991.

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