Demand-Driven Register Allocation

TODD A. PROEBSTING
University of Arizona
and
CHARLES N. FISCHER
University of Wisconsin

A new global register allocation technique, *demand-driven register allocation*, is described. Demand-driven register allocation quantifies the costs and benefits of allocating variables to registers over live ranges so that high-quality allocations can be made. Local allocation is done first, and then global allocation is done iteratively beginning in the the most deeply nested loops. Because local allocation precedes global allocation, demand-driven allocation does not interfere with the use of well-known, high-quality local register allocation and instruction-scheduling techniques.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*code generation;optimization*

General Terms: Algorithms, Languages, Performance, Theory

Additional Key Words and Phrases: Optimizing compiler

1. INTRODUCTION

The dominant paradigm in modern global register allocation is graph coloring [Briggs et al. 1989; Chaitin et al. 1981; Chow and Hennessy 1990; Larus and Hilfinger 1986]. Unfortunately, graph coloring does not really address the issue of register allocation, but rather the related issue of register assignment. That is, graph coloring tells us how to assign registers so that simultaneously live values are not assigned the same register. The harder problem—which values to put in *some* register—is not directly addressed. Hence there is always a spilling heuristic that reduces register demand until coloring (register assignment) can succeed.

A famous political maxim states that "All politics is local," and we believe that much the same is true for register allocation. Ultimately, when an operand is actually used it must be in a register, and once a value is in a register, it is easy to reuse within a basic block. Simple, fast, and nearly optimal local register allocators

This work was partially supported by NSF Grants CCR-9122267, CCR-9415932, CCR-9502397, ARPA Grant DABT63-95-C-0075, and IBM Corporation.
A preliminary version of this article was presented at the 1992 SIGPLAN Conference on Programming Language Design and Implementation.
Authors' addresses: T. A. Proebsting, Department of Computer Science, University of Arizona, Tucson, AZ 85721; C. N. Fischer, Computer Sciences Department, University of Wisconsin, 1210 W. Dayton Street, Madison, WI 53706.

are known [Hsu et al. 1989]. Once local register needs are met, the effects of global allocation can be estimated [Beatty 1974; Morris 1991]. In particular, a good global allocation improves upon good local allocation by eliminating unnecessary loads at the entrance to a basic block and by eliminating unnecessary stores at the exit from a basic block. An initial load of `v` is unnecessary if all predecessors exit with `v` in some register, and a terminal store of `w` is unnecessary if `w` is dead or if all succeeding loads of `w` can be replaced with a reference to `w`'s register.

Of course we do not initially know which values will ultimately be allocated to registers, so we estimate each value's chances of ultimate register residence. At the basic block level, we know values loaded locally into a register and not overwritten have a 100% chance of exiting in a register. The likelihood of other values residing in a register on exit depends on their likelihood of residing in a register upon entrance, the number of unused registers in a block, and the pattern of local register usage. The likelihood that a value will be in a register upon entrance to a block depends on the likelihood it will exit in a register from all predecessor blocks.

During global register allocation, we will model the competition between register candidates with estimates of the likelihood of register residence based on the demand for registers. Each instruction that requires a target (destination) register must acquire a register from somewhere—either from a pool of free registers, or, if necessary, from a register candidate. If the instruction must take a register from one of possibly many register candidates, we will assume that it can choose to spill any of competing candidates with equal probability. For instance, if the instruction must take a register from one of four candidates that might be in registers, each of those candidates has a 75% chance of "surviving" that instruction. That 75% is a measure of the register demand among those candidates at that instruction. By combining estimates of register residence at all instructions where a register candidate is live in a program, we can measure the total competition that candidate faced.

This article describes *demand-driven register allocation*. Our algorithm estimates the demand for registers among register candidates. The algorithm makes allocation decisions driven by estimates of the demand for registers and estimates of the benefits of register residence.

Once initial estimates of register allocation are made, these estimates are weighted by the net benefit gained by allocating a given value to a register, and the most promising candidate is allocated to a register. Estimates are recomputed, and again the most promising candidate is allocated to a register. This continues until all registers are allocated. The resulting technique is simple and yet identifies those values that can readily and profitably reside in a register.

2. GRAPH-COLORING ALLOCATORS

The basic graph-coloring technique involves creating a register interference graph and then pruning nodes from that graph that can be trivially colored (assigned a physical register) [Briggs et al. 1989; Callahan and Koblenz 1991; Chaitin 1982; Chaitin et al. 1981; Chow and Hennessy 1990; Larus and Hilfinger 1986]. The nodes of the graph represent the *live ranges* of the different register candidates (variables and temporaries). The live range of a candidate is the set of all program points where that candidate is live—as computed by data-flow analysis. Figure 1 gives

an example inner loop of a procedure, and Figure 2 gives the interference graph for the variables and virtual registers.[1] Notice that the graph abstracts away all control-flow information about *how* the different live ranges interfere with each other.

Given enough registers to "color" all register candidate values, this technique works well. However, once the interference graph is reduced to a graph for which the node-pruning heuristic blocks, the allocator must act so that register assignment may continue. Various graph-coloring techniques differ precisely in what they do when pruning blocks.

When the pruning heuristic blocks, Chaitin's techniques [Chaitin 1982; Chaitin et al. 1981] take the simplest approach. A node (register candidate) is picked based on a cost measure, removed from the graph, and assigned permanently to a memory location. All subsequent references to that value must be from memory.

Priority-based coloring [Chow and Hennessy 1990; Larus and Hilfinger 1986] also builds an interference graph and attempts to color it by pruning nodes. If this pruning blocks, a heuristic is employed to split large, costly live ranges into smaller ranges in an attempt to produce a graph that can be further pruned. Heuristics are used to split live ranges to minimize the costs of spilling and reloading the values across the boundaries to the new, smaller live ranges. This is repeated until all the register candidates are assigned registers.

Callahan and Koblenz [1991] allocate registers globally by doing graph-coloring "hierarchically." They treat the program as a hierarchy of nested "tiles." Tiles may be basic blocks, conditionals, or loops. They assign registers using graph-pruning techniques, but start by assigning registers in innermost tiles and progressively assigning registers in enclosing tiles. This technique succeeds in isolating some local register needs from global register usage so that variables referenced within deeply nested loops will be assigned a register before a variable that is not referenced within the loop. If, however, within a tile, the graph-pruning algorithm fails to find a coloring, their technique resorts to the graph-coloring spill techniques outlined in Briggs et al. [1989]. Therefore, while succeeding in biasing register allocation within a loop to variables used within that loop, spill decisions must still be made via ad hoc heuristic methods.

It is difficult for graph-coloring techniques to do local (basic block level) register allocation as well as established local allocation algorithms [Fischer and Leblanc 1988; Freiburghouse 1974; Hsu et al. 1989]. Unlike graph-coloring algorithms, local allocation techniques are able to exploit information about the simple sequential nature of register usage in the block to minimize local spill code. This information is lost when register allocation is cast as a graph-coloring problem.

3. DEMAND-DRIVEN REGISTER ALLOCATION

Our technique, *demand-driven register allocation*, utilizes graph-coloring techniques to assign registers, but not to allocate them. Demand-driven register allocation uses estimates to measure register demand among global register candidates so that good allocation decisions are made. Allocation is done prior to assignment based on

[1]Only virtual registers `v6` and `v3` are given in the interference graph because the other virtual registers are subsumed by a particular variable.

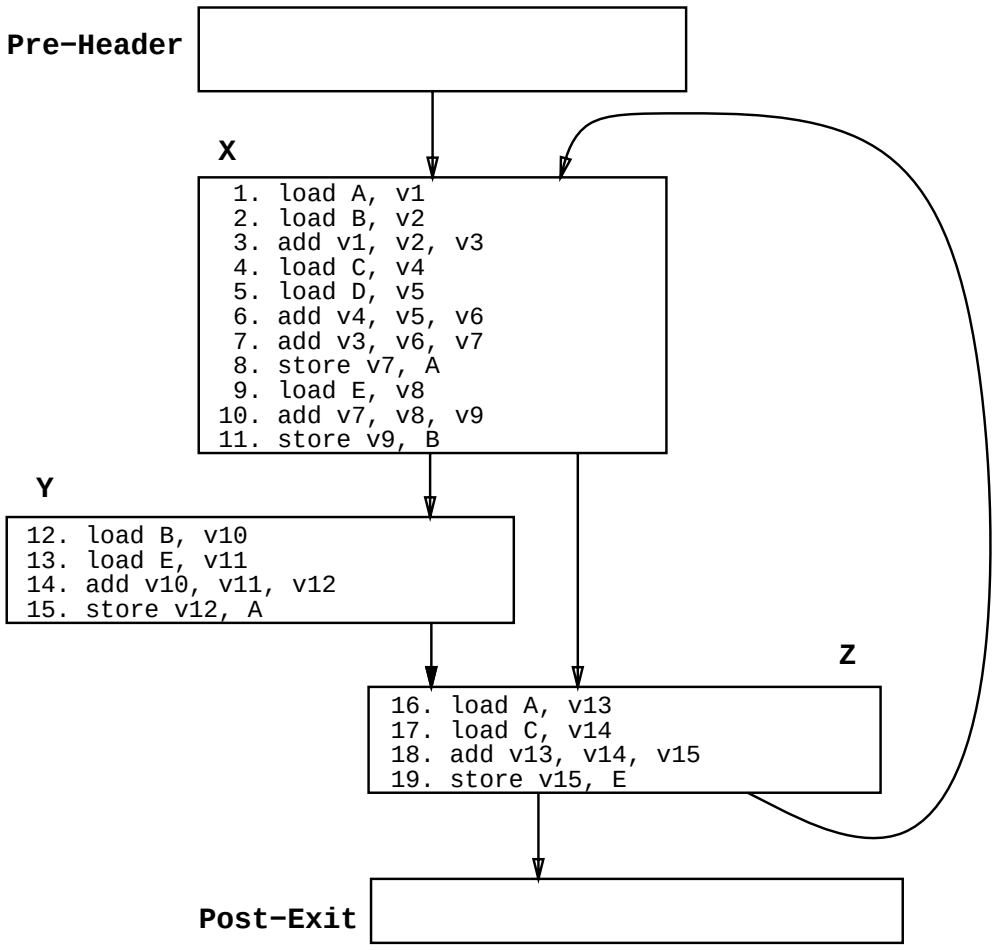
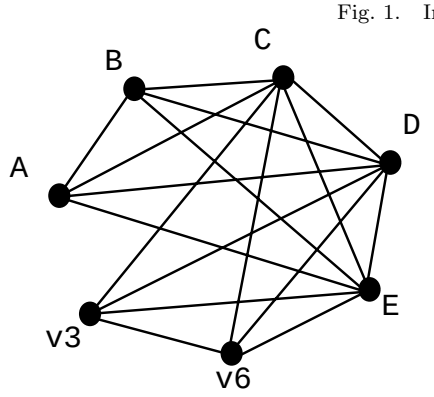


Fig. 1. Inner loop.




```
repeat
      A := (A+B)+(C+D)
      B := A+E
      if () then
           A := B+E
      endif
      E := A+C
until ()
```

Fig. 2. Interference graph and source code (without conditional expressions).

quantified measures of the costs and benefits of having particular values in registers. Separating allocation from assignment allows our algorithm to concentrate on the important problem of determining which values will profitably be held in registers at different points in the program.

Demand-driven global register allocation follows local allocation. Global allocation proceeds from inner loops to outer loops so that values used within a loop are routinely allocated a register for that loop.

3.1 Local Register Allocation

Most local register allocators share the basic principle of deciding what value should stay in a register (or when a spill is necessary) by checking the closeness of the next use of values already in registers [Fischer and Leblanc 1988; Freiburghouse 1974; Hsu et al. 1989]. If a value in a register has only distant next uses then it will be spilled before a value to be used sooner. The intuitive explanation for such a heuristic is simple: the farther a potential use of a value is from a program point, the less likely the value is to remain in a register all the way to that use, thereby decreasing our estimate of the likelihood the value will retain register residence. The likelihood of a value being able to stay register-resident can be viewed as roughly inversely proportional to the distance that value would have to stay register-resident. Therefore, heuristics choose to spill the value that seems likely to lose its register anyway. Conversely, one can view retaining a register allocation over a long distance as more likely to result in spills of other register values—another motivation for spilling those values with only distant uses.

To avoid the NP-Complete problems of optimal local register allocation, heuristics use a simple summary of the local circumstances to drive register allocation and spill decisions. We believe that a formal characterization of this summary information as numeric estimates has been ignored because, in straight-line code, distance is always directly related to the likelihood of either being spilled or causing other values to be spilled. That is, from any given point in a basic block, a distant use has a greater chance of being spilled than a closer use.

3.2 Global Register Allocation

The use of estimates of register residence to drive global register allocation has not been previously studied. Estimates present the foundation for a global register allocator that combines the advantages of excellent local allocation with global allocation. Demand-driven register allocation avoids the problems of live-range splitting that plague graph-coloring techniques [Briggs et al. 1989; Chaitin et al. 1981; Chow and Hennessy 1990] by implicitly (and automatically) splitting ranges where the likelihood and benefit of residing in a register are low.

The likelihood that a register value will continue to exist in that register at a more distant point in straight-line code can be seen as inversely proportional to the distance to that point. This is not exact, but is a good estimate.

Global estimates require the ability to handle control flow. Loops and conditionals complicate matters because a value in a register may originate in many different locations and reach many different uses—there is not necessarily a unique next use or a single defining point. Simple data-flow analysis can determine all of the definitions that reach a particular use. We must estimate the likelihood that a value will

remain in a register over all paths reaching a particular use from all the reaching definitions.

A simple estimate can be derived by extending our local, basic block heuristic—count the total number of intervening instructions on all of the paths from definitions to the use in question.

Unfortunately, in practice this yields a relatively poor estimate because not all intervening instructions equally affect the chance that a register will need to be spilled. In addition, global register values only need to be spilled when there are too few registers for both local needs and live global values. If along some path there is a surplus of registers, all variables needing registers along that path can be allocated registers with a 100% chance. Likewise, if along some path from a definition to a use the local allocator requires *all* of the registers, it would be impossible for the global value to be in a register along that path, and hence the chance of the value being in a register at the use would be 0.

3.3 Calculating Local Register Estimates

Our *global* estimate that a variable will reside in a register at a given use (load) is computed from the Δ-*estimates* for that variable in the blocks that reach the load. Δ-estimates are computed locally (for each live variable) based on local allocations and live-variable analysis. Δ-estimates are computed on a per-instruction basis and indicate our estimate that a variable's value will continue to reside in a register after that instruction's register needs are resolved *if* the value had been in a register up to that instruction.

The algorithm maintains a configuration state at each point in the basic block. The configuration is a 4-tuple, (*allocated*, *candidates*, *unallocated*, *possibly*). The values (variables and temporaries) that are allocated to registers make up the set *allocated*. The variables (register candidates) that are competing for registers at a given point in the program are in the set *candidates*. The maximum number of register candidates that could be allocated registers is *possibly*. The count of registers known to contain no useful value is *unallocated*. The total number of allocatable registers (a constant) is *REGISTERS*.

The configuration will always maintain three important invariants. It must be the case that the sum of the number of allocated registers, possibly allocated registers, and known available registers is equal to the total number of registers available:

$$|allocated| + possibly + unallocated \equiv REGISTERS.$$

The size of each group is bounded below by 0 and above by the total number of registers:

$$0 \leq |allocated|, possibly, unallocated \leq REGISTERS.$$

And, it cannot be the case that there are more candidate values possibly in registers than there are candidates:

$$possibly \leq |candidates|.$$

The Δ-estimates are calculated after local register allocation by iterating (in order) over the instructions in each basic block. Figure 3 gives the algorithm for computing the Δ-estimates for instructions within a basic block.

```
1 procedure ComputeDeltas(BasicBlock)
    //Initialize Configuration.
2   allocated ← { variables allocated registers entering the block }
                        // Before global allocation begins, allocated will be empty here.
3   candidates ← { live variables entering BasicBlock } − allocated
4   possibly ← Min(REGISTERS– |allocated|, |candidates|)
5   unallocated ← REGISTERS– (|allocated| + possibly)
6   ∀ insn ∈ BasicBlock do                          // Iterate over instructions in order.
7     ∀ f ∈ registers freed by insn do                  // Last use of a register frees it
8       Let v be the value held in f.
9       allocated ← allocated − { v }
10      if v is a live variable then
11        candidates ← candidates ∪ { v }
12        possibly ← possibly + 1
13      else
14        unallocated ← unallocated + 1
15      end if
16    end ∀
17    if insn allocates r, a register then            // First use of a register allocates it
18      Let v be the value held in r.
19      allocated ← allocated ∪ { v }
20      if v holds the value of variable (ie., not a temporary) then
21        candidates ← candidates − { v }
22      end if
23      if possibly > |candidates| then
          // Only occurs when possibly = |candidates| prior
          // to satisfying preceding conditional.
24        Δ ← 1.0
25        possibly ← possibly − 1
26      else if unallocated > 0 then
27        Δ ← 1.0                  // Allocating an empty register cannot kill anything.
28        unallocated ← unallocated – 1
29      else
30        Δ ← (possibly–1)/possibly
31        possibly ← possibly – 1
32      end if
33    else
34      Δ ← 1.0                          // Allocating no register cannot kill anything.
35    end if
36    ∀ v ∈ candidates do
37      ΔTable[insn][v] ← Δ
38    end ∀
39  end ∀
40end procedure
```

Fig. 3. Procedure to compute Δ-estimates.

Given the local allocation, each instruction will possibly free and allocate registers. Freeing a register will change the configuration by deleting a member of the *allocated* set. If the register held the value of a variable that is still live (determined by global data-flow analysis), then that variable is added to *candidates*, and the *possibly* count is incremented. If the variable is dead, then the *unallocated* count is incremented.

An instruction that allocates a register must acquire a register from either *unallocated* or *possibly*. When *possibly* = |*candidates*|, and the allocated register holds a variable from *candidates*, all the Δ-estimates will be 1, and the register used will come from the set counted by *possibly*. This follows from the fact that when *possibly* = |*candidates*|, there are exactly enough registers available to hold the elements of *candidates*.

Otherwise, if no unallocated registers exist, one of the registers counted by *possibly* must be taken. Remember that *possibly* counts the number of variables in *candidates* that might be allocated registers. Since we may be taking a register from such a variable, we must compute our estimate that a particular variable's register will be taken. Given *possibly* registers to choose from, *if* a given variable were in a register, our estimate that the allocator would *not* choose that variable's register is

$$\frac{possibly - 1}{possibly}.$$

That is, since registers have not yet been allocated, we assume each value in *possibly* has an equal chance of being spilled. Therefore, (*possibly* − 1)/*possibly* is the Δ-estimate that a live variable (a member of *candidates*) will keep its register past this instruction. If there are unallocated registers, such a register is used first, the *unallocated* count is decremented, and the Δ-estimate of all live variables at this instruction is 1. If an instruction does not allocate any registers, it cannot cause any live variables to lose a register, so the Δ-estimate for all such values is 1.

3.4 Example Computation of Local Estimates

The example in Figure 4 illustrates how local allocation, liveness analysis, and estimates interact for potential register variables. (It is based on the program whose flow graph is shown in Figures 1 and 6.) The example assumes that there are three registers available to be allocated among the five variables and the intrablock temporary values. After the first instruction, local allocation requires one register, so one of the three registers must be allocated to `A` at this point, and only two of the remaining three registers will keep their values on entry. Therefore, the others (`B`–`E`) have a 67% chance of retaining a register. (We assume that if a value is live on entry, then it *may* be in a register, and some value must give up its register when the local allocator needs an extra register. For simplicity, in the algorithm as implemented, we do not take "*distance to next use*" into account when estimating spill probabilities at a given instruction. Distance *does* affect overall estimates of register residence for a complete live range.) A "1*" indicates that the variable was either loaded or calculated into a register at this instruction and therefore must exist in a register. A boldface "**1**" indicates that the value has been allocated a register through that instruction. (At this point all allocations are local, but later

Block X

Instr. No.	Instruction	Local Needs	Δ-Estimates A	B	C	D	E	Comments
1	load A, v1	1	1*	2/3	2/3	2/3	2/3	
2	load B, v2	2	**1**	1*	1/2	1/2	1/2	
3	add v1, v2, v3	1	0	0	1	1	1	`A, B` are dead
4	load C, v4	2	-	-	1*	1	1	
5	load D, v5	3	-	-	**1**	1*	0	All registers in use
6	add v4, v5, v6	2	-	-	1/2	1/2	1/2	Reuse v4 or v5
7	add v3, v6, v7	1	1*	-	1	1	1	Reuse v3's register
8	store v7, A	1	**1**	-	1	1	1	
9	load E, v8	2	**1**	-	1	1	1*	Reuse v6's register
10	add v7, v8, v9	1	2/3	1*	2/3	2/3	2/3	
11	store v9, B	0	1	**1**	1	1	1	
Estimate that Value Remains in Register			2/3	**1**	1/3	1/3	2/3	

Fig. 4. Δ-estimates (3 registers available for locals and globals).

the global allocation algorithm will also allocate registers into, out of, and through basic blocks for different register candidates.)

Instruction 2 requires another register, so one of the remaining variables must be spilled—but not `A` because the local allocator keeps it in a register for a local use (instruction 3). Because instruction 2 requires one of the two remaining registers to be spilled, each candidate has a 50% chance of retaining a register. Instruction 3 does not lower our estimates for any live variable because it frees the registers holding both `A` and `B`, and both those values are now dead. Therefore, either of those two registers can be used to hold the result of the addition without having to spill any other register candidates. It is impossible (chance = 0) for `E` to be register-resident at instruction 5 because the instruction sequence 1–5 requires all three registers.

The bottom row of the table in Figure 4 indicates our estimates that a variable will be in a register on exit. These values were calculated by multiplying together all the Δ-estimates of the variable from the last point it was certain to be in a register to the end of the block. Thus our estimate of register residence depends on the number of instructions a register-held value must span and the demand for registers at each instruction.

It is also possible to calculate a *conditional estimate* for variables that are not referenced within a basic block. If such a variable is in a register on entry to a block, the product of the Δ-estimates for the entire block is our estimate that it will be in a register on exit.

Figure 5 gives two example basic blocks with such "pass-through" values. In the top example (**Block Y**), the exit estimates for `A`, `B`, and `E` are absolute (because the variables are referenced within the block). The estimates for `C` and `D` are conditional—they are our estimates that the value will still be in a register on exit *if* it was in a register on entry.

The computation of the Δ-estimates at instruction 14 (Figures 5 and 6) demonstrates the powerful interaction of local allocation and global data-flow analysis. Global data-flow analysis indicates that the value of `E` is dead after instruction 14, and, therefore, the local allocator may reallocate its register to hold the value of

Block Y

Instr. No.	Instruction	Local Needs	Δ-Estimates A	B	C	D	E	Comments
12	load B, v10	1	2/3	1*	2/3	2/3	2/3	
13	load E, v11	2	1/2	**1**	1/2	1/2	1*	
14	add v10, v11, v12	1	1*	**1**	1	1	0	E is dead (global info)
15	store v12, A	0	**1**	**1**	1	1	-	
Estimate that Value Remains in Register			**1**	**1**	1/3	1/3	0	

Block Z

Instr. No.	Instruction	Local Needs	Δ-Estimates A	B	C	D	E	Comments
16	load A, v13	1	1*	2/3	2/3	2/3	2/3	
17	load C, v14	2	**1**	1/2	1*	1/2	1/2	
18	add v13, v14, v15	1	2/3	2/3	2/3	2/3	1*	
19	store v15, E	0	1	1	1	1	**1**	
Estimate that Value Remains in Register			2/3	2/9	2/3	2/9	**1**	

Fig. 5. Δ-estimates (3 registers available for locals and globals).

the computation (`A`). Because no additional registers were needed by the local allocator, the Δ-estimates of `B`, `C`, and `D` are 1, reflecting no demand for registers (at that instruction).

3.5 Calculating Global Register Estimates

Given the Δ-estimates for each variable at each instruction, it is a simple matter to estimate the likelihood that a variable will be available in a register at a particular load. Data-flow analysis isolates all the reaching *register-definitions* of the variable. A register-definition is any point in the program that the variable is known to be register-resident due to previous allocation decisions. The **Pre-Header** (of a block or subprogram) is also considered to be a register-definition of *every* variable, since loads can be added there as needed. A register-definition, d, reaches a load, ℓ, if there exists a path from d to ℓ that does not redefine d's register. The set of all program points along all definition-free paths of a register from its definitions to a load is the *register-live-range* of the load. To remove the load of a variable, the variable must be allocated a register over its entire register-live-range.

The Δ-estimate of a particular instruction is our estimate that a variable's value will continue to be in a register after that instruction executes if it were in a register before the instruction. Therefore, our estimate that none of the instructions in the register-live-range will kill the value is simply the product of all their Δ-estimates.

The computation of global estimates is done over the entire register-live-range rather than on a per-path basis because a load cannot be removed unless the variable is allocated a register along *all* paths from (possibly many) reaching register-definitions to the load. The register-live-range is precisely the set of all such program points.

The inner loop in Figure 6 has been annotated with global estimates of register residence at each load assuming there are three registers available. (Blocks **X**, **Y**, and **Z** are those given in Figures 4 and 5.) For instance, our estimate of `C`

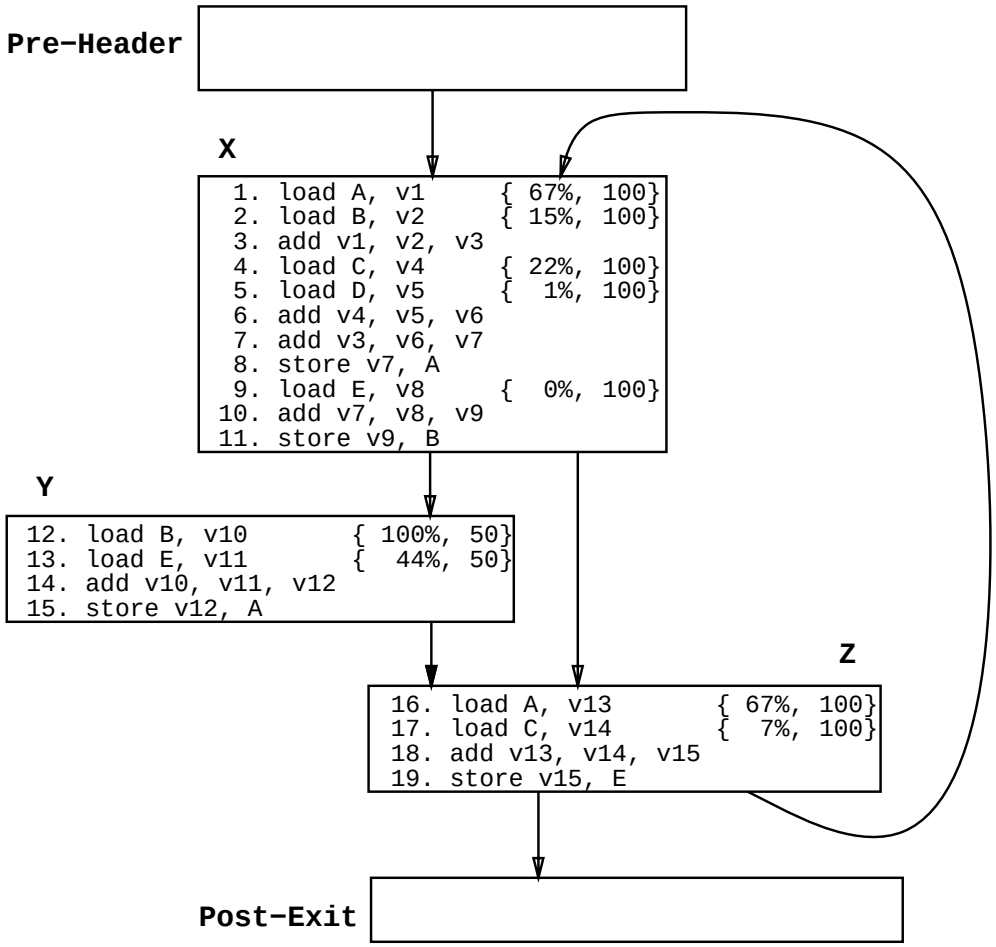


Fig. 6. Sample inner loop with global estimates (assuming 3 registers). The loads are annotated with {*estimate, benefit*} pairs.

being in a register at instruction 4 is equal to the product of the Δ-estimates for instructions 18, 19, 1, 2, and 3. (There are two register-definitions that reach the load at instruction 4: one in the **Pre-Header** and one at instruction 17.) Our estimate is, therefore,

$$2/3 \times 1 \times 2/3 \times 1/2 \times 1 = 2/9 \approx 22\%$$

Similarly, to remove the load of `A` at instruction 16, `A` must be register-resident from both of its reaching register-definitions at instructions 10 and 15. (Instruction 10 is a register-definition because `A` is in `v7`, an operand of instruction 10's addition.) Therefore, the register-live-range of `A`'s load at instruction 16 is instructions 10, 11, and 15, with Δ-estimates of 2/3, 1, and 1, respectively. Our estimate of `A` being register-resident at instruction 16 is, therefore, 67% (the product of the Δ-estimates).

C is more likely to be register-resident at instruction 4 than B is at instruction 2 even though it is later in the basic block. This can be explained intuitively by observing that B must be register-resident through more of the loop (instructions 13–19, 1) than C (instructions 18–19, 1–3).

The extremely low estimate (1%) for register residence at instruction 5 is a consequence of the fact that, for D to be available in a register at that location, it must be in a register for the entire loop.

4. ESTIMATES GUIDE GLOBAL REGISTER ALLOCATION

Our global register allocation algorithm uses local estimates combined with a measure of benefit (as defined below) to determine which variable uses should be globally allocated registers. Uses with a large benefit and a high estimate of register residence are given the greatest register allocation priority. We combine the measure of benefit and register residence by multiplying them together to get a *figure of merit*. Candidates with a higher figure of merit will be given priority. Our figure of merit reflects the benefit of allocating a register to a value, biased by our estimate of how likely it is a value can be allocated a register.

The benefit of allocating a register to a use (and hence to all the paths reaching that use) is determined by estimating how often the use will be executed and hence how many cycles may be saved if the value is accessed from a register rather than from memory. This can be determined heuristically from loop and conditional nesting levels or empirically through profiling information from previous executions of the program [Ball and Larus 1992].

Once a particular use has been allocated a register, there is no need to do a load of the variable at that use, thus saving time and space. Allocating a register causes the estimates for other interblock variables to change. This follows from the observation that if a register becomes allocated at some point in the program, it's estimate must be 1 at that point. To make room for this fixed value, our estimates for any competing candidates must change.

If, for example, B and E in block **Z** become allocated, then after the load of A at instruction 16 all three of the registers will be in use. Therefore our estimate that C could be in a register at instruction 17 must drop to 0. Because E is dead at instruction 17, its register will be allocated to C when C is loaded.

Because estimates and allocations interact, global allocation is done iteratively. A greedy algorithm finds the best candidate for register allocation based on their relative figures of merit (*estimate* × *benefit*). Then the estimates are recalculated, and another register is allocated. This process is repeated until there are no remaining uses with estimates greater than 0.

Recomputing Δ-estimates requires that *ComputeDeltas*() (Figure 3) be reexecuted for each block in which global allocations have taken place. These global allocations change the set of registers that are allocated upon entry (*allocated*), and they change which registers are freed by various instructions (e.g., allocated registers are no longer freed).

4.1 Improving Demand-Driven Register Allocation

Demand-driven register allocation carefully determines which candidates for registers show the greatest promise to benefit the program. Additional techniques

(1) Do local allocation for all basic blocks in the procedure.
(2) For all loops, l, in the procedure, from innermost to outermost do the following:
  (a) Attempt to remove loads in l:
    i. Compute Δ-estimates for all instructions in the loop (with *ComputeDeltas*()).
    ii. For each load of a register candidate in the loop, estimate the chance that the value will reach the load in a register. This is the product of all the Δ-estimates for that candidate in the load's register-live-range.
    iii. If no loads have an estimate greater than 0, then quit processing loads.
    iv. Otherwise, remove the load with the greatest figure of merit, and allocate the candidate a register across its entire register-live-range. (This may require putting a load in the loop's preheader.)
  (b) Attempt to remove stores in l:
    i. Compute Δ-estimates for all instructions in the loop (with *ComputeDeltas*()).
    ii. For each store of a register candidate in the loop, estimate the chance that the stored value will reach *all* remaining reachable loads in l, and those postexits in which the value is live in a register. This is the product of all the Δ-estimates for that candidate along all paths to these loads and postexits.
    iii. If no stores have an estimate greater than 0, then quit processing stores.
    iv. Otherwise, remove the store with the greatest figure of merit, and allocate the candidate a register along all paths to the postexits. This requires putting stores in the loop's postexits.
(3) Remove loads that are not in any loop. (Use techniques described in (2a) above.)
(4) Remove stores that are not in any loop. (Use techniques described in (2b) above.)
(5) Assign registers to all candidates allocated registers.

Fig. 7. Outline of global register allocation.

complete the register allocation process. Our algorithm allocates registers *inside-out*, from the most deeply nested regions to the outermost, thereby emphasizing allocation in the code most likely to be executed often. The algorithm assumes that all variables initially reside in memory until they are allocated a register. After local allocation, the algorithm locates loads that can be removed by allocating a register along all paths from reaching definitions to the chosen load. After loads are removed, it is possible that some stores may be removed—those whose values are no longer referenced from memory.[2] In addition, some loads and stores may be removed from a loop by placing them outside the loop in a preheader or postexit to increase loop speed. Local allocation, followed by load and store removal, provides a mechanism for global allocation that avoids the difficulties of splitting live ranges or isolating spill candidates. The global allocation algorithm is outlined in Figure 7.

4.2 Example

The example in Figure 6 illustrates a slightly simplified version of our algorithm (again, assuming 3 registers). Loads will be examined for removal in order of benefit and estimate of register residence. The second number of each tuple is the *benefit* of removing the load—a value of 100 indicates the load will be executed on all of the 100 iterations of the loop whereas a value of 50 represents an estimate that that branch of the conditional will only execute half as often. (Our estimate that a value resides in a register from the **Pre-Header** is 100% because that can be made

[2]Care must be taken not to remove the final store of a global variable in a procedure or a store of a global immediately prior to a call.

so by adding a load of the value there.)

Because instruction 12 has a 100% chance of finding B in a register, its consideration could be deferred indefinitely—it is certain to be in a register! This 100% chance is easily deduced by recognizing that block **Y** has only **X** as a predecessor and that instruction 10 (in **X**) left B in a register. (This is information that an interference graph could not readily provide.) Delaying allocation (and the subsequent removal of the load) would, however, contribute spurious Δ-estimates to the calculation of global estimates for other uses; we therefore remove it immediately. The (possibly) inaccurate Δ-estimates occur because the load allocates a register for its result, and that may lower estimates for competing register candidates. All loads with 100% chance of register allocation are removed immediately, regardless of their benefit. Only this situation of a 100% chance (*certainty*) is treated as a special case; it is done to ensure accurate Δ-estimates.

The loads of A at instructions 1 and 16 will be removed next because they have the highest estimates (67%) of those loads with a benefit of 100 (and hence the highest "merit" value of 67). In order to remove the load at instruction 1, it is necessary to ensure that A's value will be in a register on the incoming control-flow arc from the **Pre-Header** by putting a load of A there.

After registers have been allocated in order to remove these three loads (instructions 1, 12, 16), the estimates for the remaining loads are recalculated. The new Δ-estimates are given in Figure 8, and the new global estimates (calculated using these new Δ's) are given in Figure 9.

The Δ-estimates for block **Y** of Figure 8 have undergone a dramatic change after the allocation of registers for A and B. Because A is allocated a register upon entry, and because A is dead at instruction 13, the register allocator can reuse that register for the load of E. Therefore, the Δ-estimates for C and D are 1 throughout the block. The summary "*Estimate that Value Remains in Register*" seems to indicate that 4(!) values have a 100% chance of being in a register at the end of the block. This result can be explained by recognizing that the estimates for C and D are conditional—the value of C (or D) is in a register on exit *only if* it is in a register on entry. Since at most one of the two could have been in a register upon entry, we have not erroneously calculated that four values could fit in three registers.

Now, four remaining loads have the highest "figure of merit" (25) for removal: three loads with benefit of 100 and an estimate of 25% (instruction 2(B), instruction 4(C), and instruction 17(C)), and one load with benefit of 50 and an estimate of 50% (instruction 13(E)). We will arbitrarily choose to allocate a register to C at instruction 4—this too requires an initial load of C in the **Pre-Header**. After this allocation, the remaining estimates change again—only two of the remaining load instructions have estimates greater than 0: instructions 13 and 17. Fortunately both of these loads can be removed.

It is useful to examine how estimates could establish that D could not be in a register at instruction 5 after removing the load of C at instruction 4. It might appear that only two of the three registers are allocated at the entrance to block **X** (for registers A and C). However, prior to loading D it is necessary to load B without destroying A or C, and this uses all three registers. Similarly, B cannot be in a register at instruction 2 because all three registers must be in use at the end of block **Z** holding the values of A, C, and E thus preventing the value of B from being

Block X

Instr. No.	Instruction	Local Needs	Δ-Estimates A	B	C	D	E	Comments
1	-	0	**1**	-	-	-	-	*Removed*
2	load B, v2	1	**1**	1*	1/2	1/2	1/2	
3	add v100, v2, v3	1	0	**1**	1	1	1	`A` is dead
4	load C, v4	2	-	0	1*	1	1	`B` is dead
5	load D, v5	3	-	-	**1**	1*	0	
6	add v4, v5, v6	2	-	-	1/2	1/2	-	
7	add v3, v6, v100	1	1*	-	1	1	-	
8	store v100, A	1	**1**	-	1	1	-	
9	load E, v8	2	**1**	-	1	1	1*	
10	add v100, v8, v200	1	**1**	1*	1/2	1/2	1/2	
11	store v200, B	0	**1**	**1**	1	1	1	
Estimate that Value Remains in Register			**1**	**1**	1/4	1/4	1/2	

Block Y

Instr. No.	Instruction	Local Needs	Δ-Estimates A	B	C	D	E	Comments
12	-	0	**1**	**1**	-	-	-	*Removed*
13	load E, v11	1	0	**1**	1	1	1*	`A` is dead
14	add v200, v11, v100	0	1*	**1**	1	1	0	`E` is dead
15	store v100, A	0	**1**	**1**	1	1	-	
Estimate that Value Remains in Register			**1**	**1**	1	1	0	

Block Z

Instr. No.	Instruction	Local Needs	Δ-Estimates A	B	C	D	E	Comments
16	-	0	**1**	-	-	-	-	*Removed*
17	load C, v14	1	**1**	1/2	1*	1/2	1/2	
18	add v100, v14, v15	1	**1**	1/2	1/2	1/2	1*	
19	store v15, E	0	**1**	1	1	1	**1**	
Estimate that Value Remains in Register			**1**	1/4	1/2	1/4	**1**	

Fig. 8. Δ-estimates after 2 allocations for `A` (`v100`), and 1 allocation for `B` (`v200`).

held in a register over that control-flow path.

After the loads are removed, simple data-flow analysis indicates that further improvements can be made by removing the stores of `A` (at instructions 8 and 15). If `A` is live on exit to the loop, a store of `A` must be added to the **Post-Exit**.

Note that removal of the "`load B, v10`" at instruction 12 was possible *without* allocating a register to `B` for its entire live range. `B` must be stored at instruction 11 so that it may be loaded at instruction 2 in a subsequent iteration, **but** our analysis indicated that the value stored at instruction 11 would be available at instruction 12 in a register. So the load could be removed.

After register allocation, `A` and `C` were effectively allocated registers for the inner region, and a load of `B` was removed. In total, five of eight loads and two of four stores were removed from the loop. Figure 10 gives the code as it would appear after allocation and assignment with the register contents after each instruction given in parenthesis.

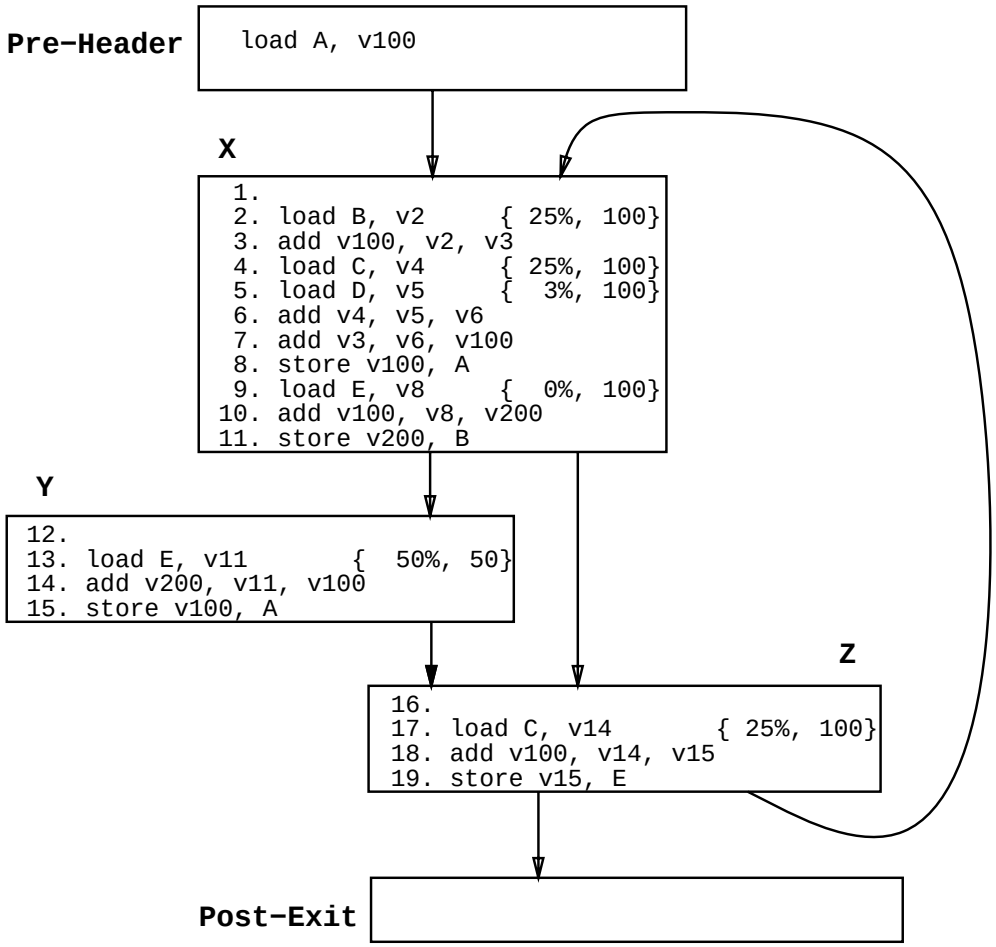


Fig. 9. Global estimates after 3 allocations.

By chance, our implementation actually gives a better allocation than the one described above by removing instructions 1, 2, 8, 11, 12, 13, 15, and 16 for an estimated cost of 50 fewer instruction. The better allocation was found because the implementation arbitrarily chose to remove instruction 2 whereas the previous (hand-calculated) example chose instruction 4. The implementation simply happened to break the tie between the candidates of merit 25 differently.

4.3 Estimates Improve Beatty's Algorithm

Our algorithm is an improvement to Beatty's register allocation scheme [Beatty 1974]. His algorithm does local allocation followed by global allocation through the removal of loads and stores to loop **Pre-Headers** and **Post-Exits**. Our algorithm differs from his in two important ways: ours uses estimates of register residence to provide better global allocation, and ours separates register assignment from register allocation. Beatty's algorithm uses only benefit analysis (estimates of execution

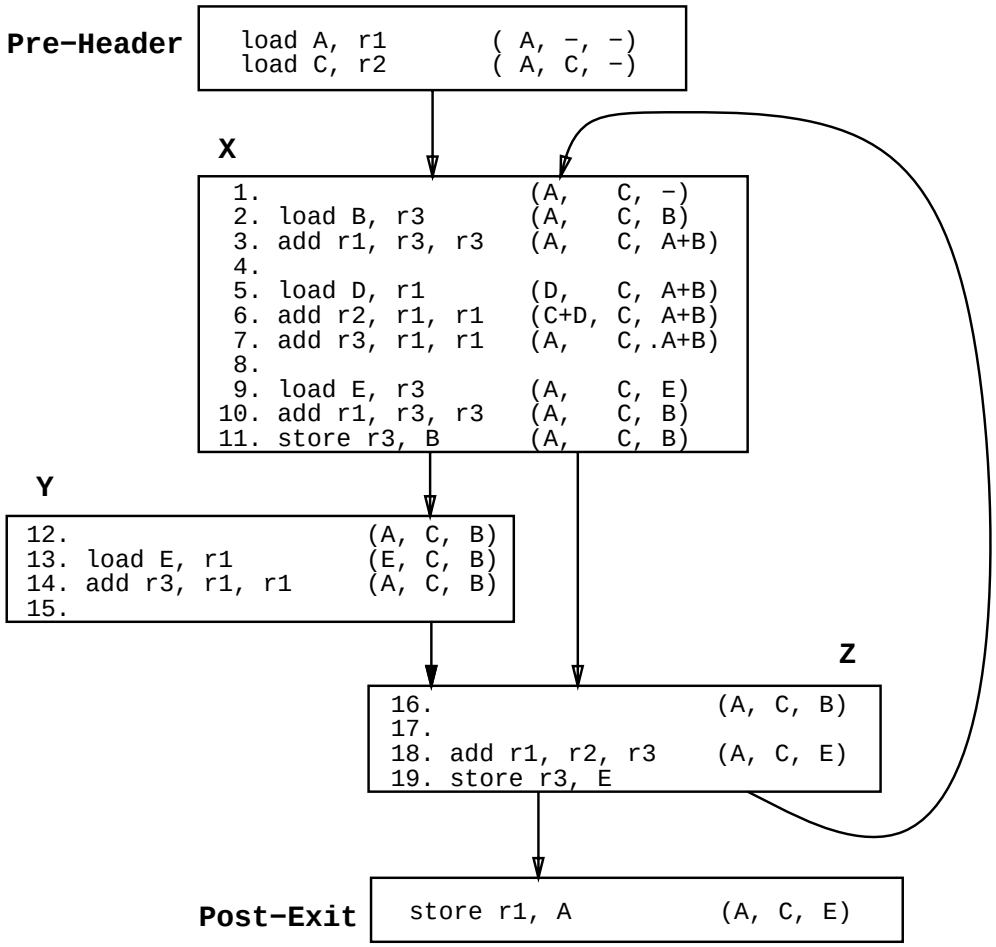


Fig. 10. Code after register allocation and assignment (register assignments in parentheses).

count) to determine which global entities are good candidates for register allocation. Absolutely no attempt is made to quantify the effects of allocating a register to a particular set of paths leading to a use. Estimates directly measure the *costs* of allocating a register in a particular code region; a low estimate indicates there is great demand for registers in the region; a high estimate indicates less demand. An estimate of 1 indicates absolutely no competition (very low demand) and hence a *free* allocation. Cost measures that incorporate register demand improve Beatty's algorithm by guiding it to make good allocation decisions by balancing benefit *and* cost estimates.

Our demand-driven register allocator works in three phases (in order): local register allocation, global register allocation, and register assignment. Local allocation is done by the simple, effective scheme of spilling (only when necessary) that value whose next use is most distant [Hsu et al. 1989]. Global allocation is done as outlined in the previous sections. Beatty's algorithm combines global allocation

and register assignment into one phase (much as graph-coloring algorithms do). Keeping allocation separate from assignment simplifies and improves the register allocation/assignments that can be found.

4.4 Register Assignment

Once registers have been allocated to live ranges, it is necessary to assign registers to them. The previous allocation phases guarantee that there will never be a point in the program that is overallocated, but so far no legal assignment has been found. All of the variables that have been allocated registers are assigned registers using graph-coloring techniques [Briggs et al. 1989; Chaitin 1982; Chow and Hennessy 1990]. An important difference between using graph coloring for allocation (as other algorithms do) and for assignment (as we do) is that failure to find a legal coloring (unlikely) does not necessitate spilling a value.

It is possible to have a program that is not overallocated for which we will not find a legal assignment. This occurs when the pattern of interferences between the register candidates is such that while there are never more than N candidates live at any point in the program, an N-coloring does not exist (or is not found) for the entire program. To create a legal assignment in such a situation, it would be necessary to either insert code to move register values to allow different register assignments for one or more candidates at different points in the program. Alternatively, we might duplicate a code fragment with a new set of register assignments to avoid conflicts.

Fortunately, situations with legal allocations, but with no legal assignments, arise infrequently. In the testing of this register allocation technique, this occurred only for the SPEC89 benchmark, `nasa`, when it was compiled with only six integer registers available.

5. IMPLEMENTATION RESULTS

A prototype demand-driven register allocator has been built as part of an experimental code generator for an ANSI C compiler ("`lcc`" [Fraser and Hanson 1991a; Fraser and Hanson 1991b]). The code generator produces MIPS R2000 assembler.

5.1 Stanford Benchmarks

We initially tested our allocator on the Stanford Benchmarks. Though these programs are rather small and simple, they did allow us to establish the viability of the technique, even for small register files. The table in Figure 11 summarizes the results of running the compiler on the Stanford benchmarks suite. Each program was run with three different register configurations for both integer and floating-point registers. For integer registers the configurations were 19 registers (9 caller-saved, 10 callee-saved), 12 registers (6, 6), and 6 registers (3, 3). For floating point, the configurations were 11 registers (5, 6), 8 registers (4, 4), and 4 registers (2, 2). The percentage of loads and stores removed in the table represent *dynamic* execution counts.

The results do not include the cost of saving and restoring callee-saved registers. This cost is negligible in all but the recursive procedures (e.g., towers.c and puzzle.c) and is modest and essentially unavoidable in those.

The Stanford Benchmarks that are not listed all had 100% of the possible loads removed at *all* register levels. It is evident that at all but the lowest register counts

Program	Registers (integer)	Loads Removed (Percent)	Stores Removed (Percent)
intmm.c	19	99.8	100.0
	12	99.8	100.0
	6	67.3	74.7
queens.c	19	98.4	100.0
	12	98.4	100.0
	6	73.0	87.7
quick.c	19	100.0	100.0
	12	100.0	100.0
	6	91.5	74.6
towers.c	19	97.3	100.0
	12	97.3	100.0
	6	96.0	94.7
fft.c	19	99.9	99.9
	12	97.7	98.9
	6	81.3	89.2
	(floating)		
fft.c	11	99.8	98.3
	8	99.8	98.3
	4	99.6	96.7

Fig. 11. Load and store removal results on selected Stanford Benchmarks.

essentially perfect allocation was obtained. Even at the lowest register levels, a very large fraction of removal loads and stores are eliminated.

5.2 SPEC Benchmarks

We also tested our allocator on the more demanding SPEC89 Benchmarks.[3] We used `f2c`, a Fortran-to-C conversion utility, to convert the Fortran benchmarks to C. Tables I–IV give the results for load and store removal.

For floating-point loads and stores, only `nasa`, `doduc`, `tomcatv`, and `fpppp` (all Fortran benchmarks) showed marked degradation with only four registers. The extremely low number of floating-point loads/stores done by `matrix300` (a floating-point matrix multiplication routine) resulted from the fact that `lcc` does not do any analysis to create register candidates from globally declared variables, and `matrix300` does not have very many local variables.

With six integer registers, the `nasa` benchmark failed to compile because the legal allocation found by our algorithm was not colorable via our coloring heuristic. It did compile with only five registers.

Only `nasa` and `matrix300` degraded significantly when integer registers were limited to the lowest level (5 and 6, respectively). For `nasa` this resulted from both local and global pressure. For `matrix300`, this is caused by inner loops that reference many scalar variables that hold index values or array addresses. With only six registers, some of those must be accessed from memory.

When compared with Chaitin-style graph coloring, demand-driven register allocation has some advantages and disadvantages. Tables I-IV indicate where each method of global allocation has advantages. Demand-driven allocation tends to do

[3]We excluded the GNU C compiler because it would not compile under ANSI C.

Table I. SPEC Benchmark Results: Float Loads Executed

		Load Removal (Execution Counts)			
Program	Registers (Float)	Total	Estimate Executed	Coloring Executed	Difference
008.espresso	11	4,484	0	0	0
	8	4,484	0	0	0
	4	4,484	0	0	0
013.spice2g6	11	331,605,625	0	0	0
	8	331,605,625	0	0	0
	4	331,605,625	23,392,753	25,090,107	-1,697,354
015.doduc	11	141,749,098	2,281,826	2,938,733	-656,907
	8	141,749,098	6,914,251	8,550,276	-1,636,025
	4	141,749,098	23,215,996	25,548,704	-2,332,708
020.nasa7	11	324,916,408	0	0	0
	8	324,916,408	0	0	0
	4	324,916,408	52,362,800	8,215,200	44,147,600
023.eqntott	11	0	0	0	0
	8	0	0	0	0
	4	0	0	0	0
030.matrix300	11	6	0	0	0
	8	6	0	0	0
	4	6	0	0	0
042.fpppp	11	25,824,867	0	0	0
	8	25,824,867	652,448	652,448	0
	4	25,824,867	3,921,032	4,908,774	-987,742
047.tomcatv	11	195,075,700	0	0	0
	8	195,075,700	0	0	0
	4	195,075,700	19,507,500	19,507,500	0

worse than Chaitin-style allocation at removing stores—this is seen most clearly in Table II. This can be explained by the fact that demand-driven allocation removes loads and stores independently, and stores may only be removed if all reaching loads are removed; Chaitin-style allocation allocates registers to live ranges and therefore removes stores and loads together. The results also indicate that demand-driven allocation often beats Chaitin-style allocation when very few registers are available. Unfortunately, the comparison does not yield a definitive winner.

5.3 Comments

`Lcc` does not do any global optimizations such as alias analysis or global common-subexpression elimination that would create many additional candidates for global register allocation and, therefore, test our techniques more strenuously. (`Lcc` only considers temporaries and scalar local variables and parameters as candidates for register allocation.) We have, therefore, chosen to create a similar register scarcity artificially by lowering the number of available registers. While not a perfect measure of how the algorithm would do in an optimizing compiler, the numbers are impressive nonetheless. Our results *do not* suggest that only a few registers are needed to remove all loads and stores. Rather, they suggest that our allocator is very effective even when the demand for registers outstrips their availability.

Table II. SPEC Benchmark Results: Float Stores Executed

		Store Removal (Execution Counts)			
Program	Registers (Float)	Total	Estimate Executed	Coloring Executed	Difference
008.espresso	11	4,762	0	0	0
	8	4,762	0	0	0
	4	4,762	0	0	0
013.spice2g6	11	186983134	0	0	0
	8	186,983,134	0	0	0
	4	186,983,134	18,660,914	15,872,192	2,788,722
015.doduc	11	53,353,887	1,456,344	1,444,681	11,663
	8	53,353,887	3,820,863	3,756,941	63,922
	4	53,353,887	11,980,523	11,793,299	187,224
020.nasa7	11	156,163,274	0	0	0
	8	156,163,274	0	0	0
	4	156,163,274	52,362,800	5,476,800	46,886,000
023.eqntott	11	0	0	0	0
	8	0	0	0	0
	4	0	0	0	0
030.matrix300	11	4	0	0	0
	8	4	0	0	0
	4	4	0	0	0
042.fpppp	11	10,693,636	0	0	0
	8	10,693,636	326,224	326,224	0
	4	10,693,636	2,616,136	2,649,368	-33,232
047.tomcatv	11	91,035,400	0	0	0
	8	91,035,400	0	0	0
	4	91,035,400	19,507,500	19,507,500	0

6. COMPILER PERFORMANCE

The prototype of our register allocator slows `lcc`'s compilation rate from over 1000 lines/sec. to about 50 lines/sec. Our simple but naive implementation recomputes data-flow and Δ-estimates information for the current loop after each load is removed. Doing this recomputation incrementally would increase the allocation speed considerably.

Alternatively, a quicker, but less accurate, allocation heuristic could be built that would not recompute these values, but would instead use a static "snapshot" of the estimates. This would avoid recomputing the Δ-estimates after each load is removed. The static initial estimates would still serve as an accurate metric of the relative demand for registers faced by each register candidate.[4]

7. ALGORITHM EXTENSIONS

7.1 Manipulating Estimates

Estimates of register residence may be improved in many ways. Estimates of register residence change as the program is transformed (by removing loads and stores and

[4]The current algorithm terminates when all the global estimates are 0. If Δ-estimates are not recalculated, it would be necessary to terminate when it is determined that no more candidates can fit in the available registers.

Table III. SPEC Benchmark Results: Integer Loads Executed

		Load Removal (Execution Counts)			
Program	Registers (Integer)	Total	Estimate Executed	Coloring Executed	Difference
008.espresso	19	1,698,143,794	37,271	0	37,271
	12	1,698,143,794	546,602	16,557,393	-16,010,791
	6	1,698,143,794	170,616,116	248,673,176	-78,057,060
013.spice2g6	19	1,227,321,939	2,719,754	0	2,719,754
	12	1,227,321,939	2,719,754	0	2,719,754
	6	1,227,321,939	2,757,106	246,428	2,510,678
015.doduc	19	238,846,530	7,387,132	0	7,387,132
	12	238,846,530	7,729,344	1,180	7,728,164
	6	238,846,530	9,179,635	2,335,920	6,843,715
020.nasa7	19	8,761,160,769	10,906	0	10,906
	12	8,761,160,769	395,729,506	234,080,396	161,649,110
	5	8,761,160,769	2,406,081,073	2,431,955,202	-25,874,129
022.li	19	1,952,143,913	11,102,117	0	11,102,117
	12	1,952,143,913	11,102,117	0	11,102,117
	6	1,952,143,913	15,735,202	18,850,464	-3,115,262
023.eqntott	19	777044072	15386	0	15386
	12	777044072	15386	0	15386
	6	777044072	2577932	3519071	-941139
030.matrix300	19	2,178,363,191	1,454,477	0	1,454,477
	12	2,178,363,191	3,624,079	1,248	3,622,831
	6	2,178,363,191	438,523,874	653,788,257	-215,264,383
042.fpppp	19	21,166,919	2,737,461	3,270,471	-533,010
	12	21,166,919	2,813,390	3,327,639	-514,249
	6	21,166,919	2,975,768	3,539,760	-563,992
047.tomcatv	19	676,717,877	0	0	0
	12	676,717,877	0	0	0
	6	676,717,877	19,558,600	45,568,600	-26,010,000

allocating registers to candidates).

It is possible—and in some cases desirable—to artificially manipulate estimates by transforming the program. For instance, inserting a load of a particular value on a path to a second load of that value will almost certainly increase the chance of that value being in a register at the original load. In fact, the *closer* the inserted load is to the latter load, the more likely it will increase the estimate.

This observation can be used to tune the allocation process. Figure 12 gives an example of where *adding* a load may help get a better register allocation. If we assume that block **Q** is voracious in its use of registers, it may be that variable `A` has a very low chance ($\approx 0\%$) of being register-resident at its load at the beginning of block **R**. However, inserting a load of `A` at the end of block **Q** will raise the estimate of `A` at the beginning of **R** to 100%. With the inserted load in **Q**, the load in **R** can be removed. At the cost of inserting a load into one arm of a conditional, a load that must *always* be executed could be removed, resulting in superior code. This is a straightforward application of partial redundancy analysis [Morel and Renvoise 1979] in which computations redundant along *some* paths are made redundant along *all* paths by the insertion of code (in our case loads) on selected paths.

Table IV. SPEC Benchmark Results: Integer Stores Executed

		Store Removal (Execution Counts)			
Program	Registers (Integer)	Total	Estimate Executed	Coloring Executed	Difference
008.espresso	19	640,130,571	5,324	0	5,324
	12	640,130,571	946,989	451,245	495,744
	6	640,130,571	84,370,487	68,903,204	15,467,283
013.spice2g6	19	519,393,524	0	0	0
	12	519,393,524	0	0	0
	6	519,393,524	27,140	228,264	-201,124
015.doduc	19	80,001,256	231,000	0	231,000
	12	80,001,256	302,514	0	302,514
	6	80,001,256	1,276,037	662,992	613,045
020.nasa7	19	2,763,958,479	900	0	900
	12	2,763,958,479	285,402	599,004	-313,602
	5	2,763,958,479	738,723,438	424,535,032	314,188,406
022.li	19	9,75,099,481	6,320,506	0	6,320,506
	12	975,099,481	6,320,506	0	6,320,506
	6	975,099,481	30,613,911	18,850,728	11,763,183
023.eqntott	19	456,491,641	53,464	0	53,464
	12	456,491,641	53,464	0	53,464
	6	456,491,641	2,521,717	881,536	1,640,181
030.matrix300	19	440,684,228	24	0	24
	12	440,684,228	4,848	16	4,832
	6	440,684,228	1,459,258	2,896,858	-1,437,600
042.fpppp	19	6,859,283	223,798	203,760	20,038
	12	6,859,283	270,139	260,869	9,270
	6	6,859,283	485,421	368,179	117,242
047.tomcatv	19	266,895,460	0	0	0
	12	266,895,460	0	0	0
	6	266,895,460	13,030,600	19,533,100	-6,502,500

Our algorithm already insert loads into loop preheaders when necessary to give a better allocation within a loop. It does this by computing estimates as if the inserted loads were already there, and then as necessary, actually adding them. Inserting loads into less frequently executed arms of conditionals to bias allocation toward removing loads in busier parts of the code should further improve allocations.

It may be possible to isolate profitable opportunities for inserting loads by computing the estimates that a variable will reach a load in a register along individual paths. If a low-probability path traverses an infrequently executed block of code, that block may be a good location to insert a load. The infrequently executed load may greatly increase the estimate of the variable reaching a subsequent, more frequently executed load. Removing the subsequent load may now be possible, and the overall effect may be positive. We believe this would be a fruitful area of future research.

7.2 Allocation Interactions

Demand-driven register allocation greedily allocates registers by identifying the load with the greatest estimated figure of merit and removing it. This greedy algorithm cannot, however, fully anticipate the "aggregate effects" of different allo-

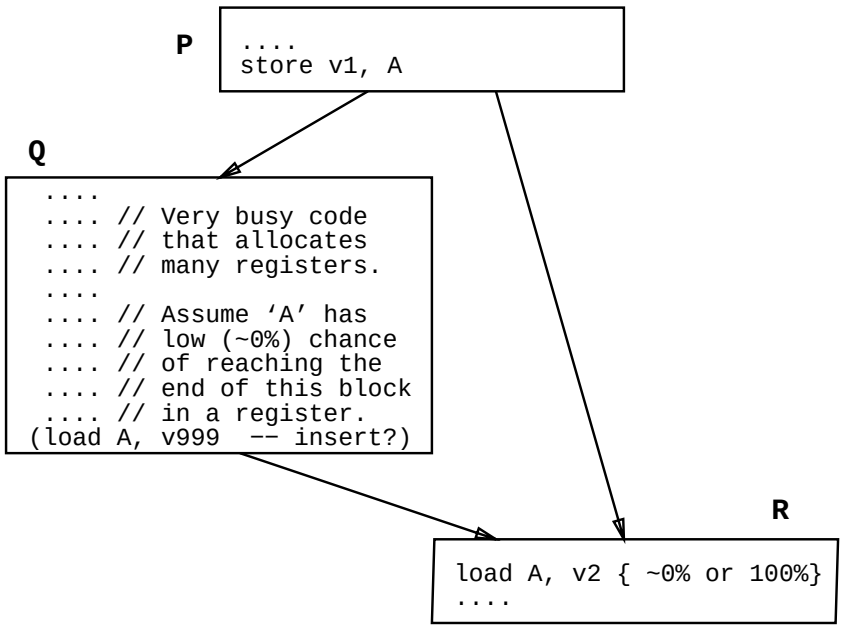


Fig. 12. Inserting a helpful load.

cation choices. For instance, it may be possible that three candidates, `A`, `B`, and `C` have comparable estimates, but that an allocation for `A` is slightly more beneficial than for either `B` or `C`. Furthermore, assume that `B` and `C` do not interfere with each other, but they both interfere with `A`. In this situation, it would be possible to allocate a single register to both `B` and `C` at a greater total benefit than allocating a single register to `A`.

By ignoring the aggregate effects and global constraints of allocation, the greedy nature of demand-driven register allocation may make suboptimal decisions. Because of the exponential number of ways grouping register candidates, how to best compute aggregate benefits for purposes of guiding register allocation is problematic. The previous example suggests grouping nonconflicting register candidates as a place to start. Finding an efficient means of utilizing a measure of aggregate benefit will require future research.

8. COMPLEXITY

The run-time complexity of our global register allocation algorithm is $O(n^3)$ where n is the number of instructions in the procedure. This assumes that the number of loads, the number of register candidates, and the size of register-live-ranges are all proportional to the number of instructions.

Computing a single Δ-estimate takes only a constant amount of work. Therefore, the complexity of computing all the Δ-estimates within a loop is

$$O(|loop| \times candidates).$$

After Δ-estimates have been computed, computing the estimate for any given load

requires multiplying all the Δ-estimates in the register-live-range, which is

$$O(|\textit{register-live-range}|).$$

Therefore, computing estimates for all the loads is

$$O(|\textit{register-live-range}| \times \textit{loads}).$$

Because the computation of an estimate is done before each load removal, the algorithm takes

$$O(|\textit{register-live-range}| \times \textit{loads}^2).$$

Assuming the register-live-ranges are proportional in size to the program, and that the number of loads removed is too, the complexity of our algorithm is $O(n^3)$. This is a very conservative bound that reflects the complexity of the current algorithm. As noted in Section 6, it may not be necessary to recompute estimates after each load is removed. This would decrease the complexity to $O(n^2)$.

9. OTHER USES FOR ESTIMATES

Register residence estimates are a good model of the scarcity or demand for registers. The previous algorithm, adapted from Beatty's algorithm, is designed to do allocation from innermost loops to outermost loops and is tuned to make good use of estimates. Register residence estimates could be used advantageously beyond this algorithm.

9.1 Assisting Graph Coloring

Estimates are general enough to help graph-coloring algorithms site spills and split live ranges. Presently graph-coloring heuristics prune the interference graph by making the trivial observation that any live range with fewer conflicts than available registers can be removed from the graph and subsequently assigned a register. Once all trivially colorable nodes have been removed, a node that cannot be trivially colorable must be pruned—with no guarantee that it can be colored. Pruning a node that is unlikely to be colored is a prudent choice. Estimates can indicate, for those live ranges with too many conflicts, which will *probably not* be allocated a register—thus intelligently directing further pruning of the interference graph. A graph-coloring heuristic could alternate between pruning all trivially colorable nodes and pruning a node that is unlikely to be allocated a register anyway.

For instance, the interference graph in Figure 2 can be updated to include estimates. These estimates, given in Figure 13, indicate the level of competition between the various register candidates—not just who is competing with whom. The estimates were computed simply by multiplying the original Δ-estimates for each variable for its entire live range.[5] The lower estimates for `D` and `E` indicate that they are infrequently used relative to demands they place on the register allocator and are therefore good candidates to prune from the graph if no trivially colorable nodes exist. The higher estimates for `A` and `B` indicate that they put less pressure on the allocator and would therefore possibly be prunable later. Nodes with low

[5]The 100% values for `v3` and `v6` assume that local allocation has guaranteed them a register.

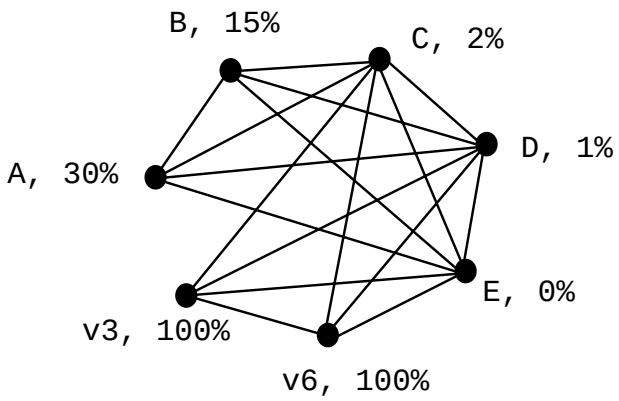


Fig. 13. Interference graph *with* estimates.

estimates are unlikely to receive a register and therefore make good candidates for early pruning from the graph when no low-degree nodes exist.

Chow's priority-based graph-coloring algorithm [Chow and Hennessy 1990] used the size of a live range (measured in instructions) as a simple measure of this competition. Of course, the expected benefit of allocating a register to a value must also be weighed when deciding between two candidates.

Similarly, demand-driven register allocation could be used *after* graph-coloring techniques have exploited all the trivial pruning opportunities available. If the pruning stage fails to allocate registers to every node in the graph, our algorithm could be used to allocate registers among the (presumably) many fewer remaining register candidates.

9.2 Assisting Interprocedural Allocation

Interprocedural register allocation attempts to allocate registers among procedures so that procedures may pass parameters in registers, avoid saves and restores around calls, and share global values in registers.

Wall [1986] built a system that allocated registers interprocedurally, at link-time when the entire program was available. All local and global variables had been previously allocated to memory, and his allocator attempted to allocate these values—when most profitable—to registers. His system allocates local variables to registers so that registers will never need to be saved around calls. That is, for any possible path in the call graph, the system guarantees that at most one local variable will be assigned to any one physical register.

It is no longer the case that local variables are competing for registers only against other variables (and temporaries) within the same procedure, but now they are competing against *all* potentially simultaneously live variables from *other* procedures. Wall's system chooses among different candidates based on estimates of execution frequencies of each candidate.

Estimates of register residence could be used interprocedurally to isolate the competition for registers among local variables of different procedures or among locals and globals. For instance, one procedure could have such high register demands

that its performance would suffer greatly if registers were allocated to other procedures' locals or to globals. Such a procedure would necessarily *lower* the estimates that any value might be allocated a register simultaneously with it. These lower estimates would discourage the interprocedural allocator from attempting to allocate those values to registers.

10. CONCLUSION

Estimates measure the demand among register candidates for scarce registers. The higher the estimate a candidate will be allocated a register, the lower the scarcity faced by that candidate. By biasing allocation toward candidates with high likelihoods of allocation, a register allocator uses registers sparingly—eliminating as few other competing candidates as possible from being allocated registers. Thus, demand-driven register allocation can weigh competition along with benefit when making allocation decisions.

Using estimates to guide global register allocation, after a local allocation phase, provides a simple and effective algorithm that avoids graph-coloring spill heuristics. It focuses attention on the problem of *allocating* registers over *assigning* registers—a weakness inherent to previous graph-coloring schemes. Graph-coloring algorithms explicitly handle splitting or spilling live ranges when registers are exhausted, but demand-driven global allocation completely subsumes these concerns. Finally, because of the clean separation between local allocation and global allocation, our demand-driven algorithm allows existing excellent local register allocation heuristics to run unconstrained by global allocation policies.

ACKNOWLEDGEMENTS

David Callahan, Chris Fraser, Robert Henry, and Brian Koblenz provided comments previous drafts of this article that improved its content and accuracy. Steve Kurlander provided the graph-coloring version of lcc and the corresponding benchmark results.

REFERENCES

Ball, T. and Larus, J. R. 1992. Optimally profiling and tracing programs. In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages*. ACM, New York, 59–70.

Beatty, J. C. 1974. Register assignment algorithm for generation of highly optimized object code. *IBM J. Res. Devel. 18,* 1 (Jan.), 20–39.

Briggs, P., Cooper, K. D., Kennedy, K., and Torczon, L. 1989. Coloring heuristics for register allocation. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*. ACM, New York, 275–284.

Callahan, D. and Koblenz, B. 1991. Register allocation via hierarchical graph coloring. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*. ACM, New York, 192–203.

Chaitin, G. J. 1982. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*. ACM, New York, 98–101.

Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cooke, J., Hopkins, M. E., and Markstein, P. W. 1981. Register allocation via graph coloring. *Comput. Lang. 6*, 47–57.

Chow, F. C. and Hennessy, J. L. 1990. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst. 12,* 4 (Oct.), 501–536.

Fischer, C. N. and Leblanc. 1988. *Crafting a Compiler*. Benjamin/Cummings, Menlo Park, Calif.

Fraser, C. W. and Hanson, D. R. 1991a. A code generation interface for ANSI C. *Softw. Pract. Exper. 21,* 9 (Sept.), 963–988.

Fraser, C. W. and Hanson, D. R. 1991b. A retargetable compiler for ANSI C. *SIGPLAN Not. 26,* 10 (Oct.), 29-43.

Freiburghouse, R. A. 1974. Register allocation via usage counts. *Commun. ACM 17,* 11 (Nov.).

Hsu, W.-C., Fischer, C. N., and Goodman, J. R. 1989. On the minimization of loads/stores in local register allocation. *IEEE Trans. Softw. Eng. 15,* 10, 1252–1260.

Larus, J. R. and Hilfinger, P. N. 1986. Register allocation in the SPUR lisp compiler. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*. ACM, New York, 255–263.

Morel, E. and Renvoise, C. 1979. Global optimization by suppression of partial redundancies. *Commun. ACM 22*, 96–103.

Morris, W. G. 1991. CCG: A prototype coagulating code generator. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*. ACM, New York, 45–58.

Proebsting, T. A. and Fischer, C. N. 1992. Probabilistic register allocation. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*. ACM, New York, 300–301.

Wall, D. W. 1986. Global register allocation at link time. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*. ACM, New York, 264–275.

Received August 1993; revised November 1995; accepted March 1996