J. C. Beatty

Register Assignment Algorithm for Generation of Highly Optimized Object Code

Abstract: A register assignment algorithm is described that, in contrast to traditional methods, permits a high level of optimization at both local and global levels. This involves splitting local register optimization into two phases, with global assignment intervening. Because novel techniques are used in the global assignment procedure, it is described in detail. Experimental results with a prototype implementation are presented in which object code improvements on the order of 25 percent over a production optimizing compiler were obtained. No attempt was made to assess manpower costs of a final implementation nor to weight them against expected improvements in generated code.

1. Introduction

The register assignment process in compilers is generally divided into local and global phases, primarily because of the much greater difficulty of global assignment. The dividing line may vary among methods, but the basic criterion is the same: that local assignment operate in a restricted context, in which enough information is available to make correct decisions. These decisions at the same time should have a high likelihood of improving the efficiency of the code generated. In global assignment, one must chose between doing something very rudimentary, such as a one-to-one assignment of selected items to registers, or going to considerable pains to get enough information for a more nearly optimal approach. Day [1] has shown, using random data, that the second alternative offers potential for substantial gains in efficiency. In spite of the pioneering work of Yershov [2] in this direction, the practical application of advanced global register assignment techniques in the western world has been limited. One reason for this is undoubtedly the complex interaction between local and global assignment. (Yershov's work was motivated primarily by storage economy and did not have this constraint.)

Because of its restricted context, local assignment is likely to have greater payoff than global assignment. It is both faster and more likely to result in the right decisions (from the point of view of efficiency of the generated code). Thus, to the extent that local and global assignment procedures compete for registers, preference should be given to the former. This was indeed done in the design of the IBM System/360 FORTRAN H compiler [3], in which local assignment for a region (loop) of the program is completed and then the remaining unused registers are assigned globally on a one-to-one basis. In fact, it is not clear how a more sophisticated approach to global assignment, such as Day's "many-to-few" algorithm [1], can be used effectively in an environment in which local registers have been completely bound. One appears to be faced with a dilemma: Either give priority to local assignment and sacrifice the benefits of an ambitious global assignment scheme, or do global assignment first, using a relatively sophisticated many-to-few strategy and reserving a fixed number of registers for a subsequent local assignment procedure.

This paper describes a register assignment method that resolves this dilemma. The approach is basically to separate local register optimization into *allocation* and *assignment* phases, with global assignment intervening. Local allocation claims all register resources needed for local communication without the premature binding that would impede global assignment. This binding is completed later, during the local assignment phase, consistently with the results of global assignment. In the local register optimization phases, relatively straightforward, but possibly highly machine-dependent, techniques are used. These are not discussed in great detail. The global assignment algorithm, however, is applicable to any machine with multiple registers, and novel techniques are used; it is described in considerable detail and illustrated by an example.

A prototype implementation of the proposed register assignment method in the context of the FORTRAN H compiler is described, and some experimental results are presented. Limitations of the prototype are outlined, indirectly suggesting the magnitude of a production implementation. However, no attempt is made to assess the tradeoff between anticipated improvements in object code and the cost of final implementation.

2. Register assignment strategy

A *basic block* (or simply a block) is defined as a sequence of nonbranch instructions followed by a (possibly null) sequence of conditional branches, followed optionally by an unconditional branch. Register assignment is customarily broken into local, or intrablock, assignment and global, or interblock, assignment. The reason for this dichotomy is probably that local assignment is easier than global assignment. However, the two processes are so interrelated that neither can be done well independently of the other. To resolve this dilemma, let us, for a program point p and a data item x, distinguish *allocating* a register for x at p from *assigning* a specific register to x at p. The former action implies a decision that x is to reside in some register at p without specifying which one. Thus the consistency of an allocation at p is dependent only on the count of available registers at p being greater than zero.

The register assignment process is then broken into the following three steps:

1. Local allocation
2. Global (allocation and) assignment
3. Local assignment.

This organization permits local allocation to be given priority over global assignment without unnecessarily impeding it by premature binding of registers.

3. Context for global assignment

The principal concern of this paper is the global assignment phase. Its functional specifications are described in this section; to a large extent they determine the division of local register optimization between the local allocation and the local assignment phases. More details about a specific implementation of these phases are given in section 12.

The inputs required for global assignment may be summarized as follows:

1. Instruction text and dictionary
2. Count of available registers at each program point
3. Control flow information
4. Live variable information at selected points.

To describe these components in greater detail, a few definitions are required.

We speak of the instants between the execution of successive instructions as *program points* and distinguish the point after the last instruction of a basic block b (the *end* of b) from the point before the first instruction of any successor b' of b (the *beginning* of b').

We will have occasion to refer to program points in their control flow context. The corresponding flow graph is obtained from the customary graph (whose nodes are basic blocks) by replacing each basic block by all of its points, connected in their linear order.

We assume that there is a hierarchical decomposition of the flow graph into strongly connected, nested subgraphs, called *regions*. These are used for the redistribution of load and store instructions in an inner-to-outer direction. These regions should thus ideally be correlated with execution frequency in the obvious way and would normally be the same regions used for other loop-oriented optimization techniques, such as backward motion of loop-invariant expressions and strength reduction. A point or another region is said to be *immediately contained* in region R if it is a subgraph of R but not of any subregion of R.

A region R, in addition to being a mere subgraph, has certain points distinguished as *entries* corresponding to the beginnings of the blocks of R with a predecessor outside of R, and *exits*, corresponding to the ends of the blocks of R with a successor outside of R. An entry p of region R is assumed to have a unique predecessor p' not in R, called the *entry target* of p. Moreover, p' has p as its unique sucessor and has the same immediately containing region as R. Similarly an exit q of R is assumed to have a unique successor q' not in R, called the *exit target* of q. Moreover, q' has q as its unique predecessor and has the same immediately containing region as R. These restrictions are motivated primarily by convenience and can always be met, if necessary, by introducing empty blocks.

A *track* is defined as a nonrepeating sequence of program points, each of which is a successor (in the context of the flow graph) of the preceding point in the sequence. The track is said to be *closed* if its first point is a successor of its last point.

A data item x is said to be *live* at a point p if there is a track T from p to a point of possible use (e.g., a load) of x, such that no store of x occurs between points of T. (The term used in [3] is "busy.") In [4], an efficient algorithm is given for determining where items are live. Sometimes the liveness of x at p is defined as requiring additionally that there be a track from a store of x to p. We refer to this as the *restrictive* definition. It is valid only under certain assumptions. Static data in a PL/I or FORTRAN subprocedure, for instance, must in general be assumed to have a store at the procedure entry in order for this definition to apply.

Now we can describe in greater detail each of the above components of the global assignment input.

Instruction text and dictionary
The instructions are partitioned into basic blocks and are expressed in a form identical to that of the object machine language except that specific registers are not assigned. An operand of an instruction i in block b may be either the result of a prior instruction in b or a data item, as represented by a dictionary entry. The latter alternative would normally occur only for an operand (such as that of a load or store) that must be in storage. (Options such as RR vs RX referencing in System/360 computer architecture are discussed in more detail in section 12.) This implies, in particular, that for a data item to be used as a register operand of i, it must be the subject of a load instruction in b the result of which would be referenced by i. Similarly, unless an instruction has a storage result, its result, in order to be assigned to a variable x, must be the subject of a store instruction referencing x. We assume that all questions of intrablock communication of values in registers have been resolved at the allocation level, i.e., that local allocation is complete. Thus global assignment for data item x need be concerned at most with the elimination (or motion) of one load and one store of x in any basic block.

Available register count
The second essential input for global assignment consists, for each program point p, of the count $RAVAIL$ (p) of registers available at p. These counts must reflect the results of local allocation. Thus $RAVAIL$ (p) is the number of registers in the machine minus the number of ordered pairs (i, i') of instructions (in the block containing p) such that i and i' are separated by p, and i' references the result of i.

Control flow information
The control flow information required for global assignment consists basically of the possible successors of each basic block and of the regions described above.

Live variable information
For each exit target q' of each region R we require the list of data items that are referenced in R and are live at q'. If the restrictive definition of liveness is used, then such a list is also required for each entry target of R.

The ouput of global assignment consists of the program text in the same form as on input except that certain loads and stores may have been marked for deletion, others may have been inserted in entry and exit targets, and the result register of certain instructions may have been assigned. Specifically, any load that has been marked for deletion and any instruction whose result must remain in a register until the end of its block must have a specific result register assigned. Moreover, a block b may have certain registers reserved for passing data items through b, even though they are not referenced in b. The available register counts in $RAVAIL$, though no longer needed after global assignment, are maintained throughout the process and reflect its results. This is done because global assignment is in part a continuation of the allocation process. Assignments are made only when an interblock communication is involved, and at any two points taking part in a communication of values of a data item x, the same register must be assigned to x for each. Thus all interblock matching has been resolved during global assignment.

4. Global assignment strategy
The classical approach to global assignment for a data item x in a region R involves the assignment of a single register to x at least at each point of R at which x is live. This clearly permits all loads and stores of x to be removed from R, given that x is loaded at the entry targets and stored at the exit targets of R at which it is live. Some classical methods of global assignment are discussed in [1]. Considerably greater freedom is obtained by allowing individual loads and stores of x to be removed from R even though not all such loads and stores can be moved. We next describe a method for doing this:

1. Regions are processed in an inner-to-outer order. Loads and stores deposited at entries and exits of a region R are considered for removal from the immediately containing region.
2. The first step in processing a region involves the computation of bit vectors, called *status vectors*, which are used to record the status of allocation and assignment.
3. For a given region R, all loads and stores of a data item x are processed together. The order in which data items are processed is significant; however, processing order is not addressed in this paper beyond commenting that reference frequency should probably be an important factor. The processing of x in R consists of the following subphases:

   a. *Load motion* This involves processing the loads of x sequentially, moving each one for which a consistent assignment can be found, based on the status vectors. Because the processing order may be significant, the most frequently executed loads, if known, should be considered first.

   b. *Store motion* Stores of x are processed sequentially as in load motion. Their motion depends partly on the results of load motion.

c. *Reservation, or status update* This involves recording the effect of the load-store motion for x on the status vectors, as well as updating the register counts in $RAVAIL$.

5. Underlying concepts for load-store motion

A *live exit* for data item x in region R is defined as an exit of R at whose exit target x is live. A live exit for x corresponding to exit q plays a role analogous to that of a load of x at the point q and may be thought of as a dummy load at this point.

The *location* of an instruction i is the point immediately following i and denoted loc (i).

Given points p and q of region R and a data item x, we say that p is in the *affect* (R,x) *relation* to q, denoted $AFFECT\ (p,q,R,x)$, if there is a track from p to q in R not containing any redefinition (e.g., store) of x. Live exits and the affect relation play a central role in store motion, because to move a store i of x out of R, it is necessary, in effect, to remove the live exits of x (considered as loads of x) to which loc (i) is in the affect (R,x) relation.

A point p is called a *register point* for data item x if p is the point immediately following an instruction whose result is in a register and is the current value of x, i.e., it would be correct to insert a store of this result into x as the next instruction. In particular, the point following a load of x would be a register point for x. A register point for x is used to eliminate subsequent loads of x in the manner in which, in global common expression elimination (e.g., as in [5]), one instance of an expression is used to eliminate a subsequent one computing the same value.

Given a point p in a region R and a data item x, we define the *register requirement set* for removing from R a load of x at p, abbreviated $RR(R,x,p)$, as the set of points q in R such that there is a track from q to p not containing a register point for x. In fact, the removal of such a load can be effected by maintaining x in a register in $RR(R,x,p)$ and, if necessary, loading it at the entry target of any entry in $RR(R,x,p)$. If the restrictive definition of liveness (section 3) is used, then loads are only needed at those entry targets at which x is live.

Let q be the location of a store of x in R, and let $p_1, \cdots, p_n$ be the locations of all the loads and live exits of x in R to which q is in the affect (R,x) relation. The store at q can be removed from R by removing all these loads from R, maintaining x in a register in each of the register requirement sets $RR(R,x,p_1), \cdots, RR(R,x,p_n)$ and storing x at each exit target of R corresponding to one of the live exits p_j of x.

A load or live exit L of data item x in region R is referred to as *absolutely infeasible* for removal from R if the corresponding register requirement set $RR(R,x,\text{loc}(L))$ contains a point p at which no registers are available, i.e., $RAVAIL(p) = 0$. A specific register r is said to be *available* to L if r has not received any specific assignment in $RR(R,x,\text{loc}(L))$. Prior to global assignment, this would normally be true of all registers.

A block b is *transparent* if $RAVAIL(p) > 0$ for all p in b. Data item x is a *potential initial quantity* for b if x is used before it is defined in b and if $RAVAIL(p) > 0$ for all p between the beginning of b and the first register point of x in b. The significance of the potential initial quantities for b is that only the loads for these items in b can be candidates for load motion, because any other loads are absolutely infeasible. Similarly, x is a *potential terminal quantity* for b if x has a register point in b and $RAVAIL(p) > 0$ for all p between the last such point and the end of b. The fact that x is a potential terminal quantity for b can thus be used to extend the register point for x in b to the end of b when this is required for removing a load in some other block.

6. Bit vector definition

For a region R, we define a number of Boolean vectors spanning the loads and live exits immediately contained in R, i.e., having a coordinate for each such load and live exit. These are called R-vectors and are partitioned into subvectors by data item. The subvector of R-vector V whose coordinates correspond to loads and live exits of a data item x is called an $[(R)]$*x-vector*, and is denoted by $V(x)$. In case V is itself indexed, say $V(j)$, its x-vector is denoted $V(j,x)$. We also have occasion to use unimbedded x-vectors, i.e., those not contained in any R-vector. In defining these vectors, we use L to denote both a load (or live exit) and its corresponding R-vector (or x-vector) coordinate. The data item loaded by L is denoted item (L).

For each basic block b immediately contained in region R, we define the R-vectors:

$RRB(b)$, whose Lth coordinate indicates membership of the beginning of b in $RR(R,\text{item}(L),\text{loc}(L))$, and
$RRE(b)$, whose Lth coordinate indicates membership of the end of b in $RR(R,\text{item}(L),\text{loc}(L))$.

Among the register requirement sets for the different loads and live exits $L_1, \cdots, L_n$ of a data item x in region R, it is important for assignment consistency to know which pairs of such sets mutually intersect. This is conveniently represented by the x-vectors:

$INT(x,L_j)$, whose L_kth coordinate indicates that $RR(R,x,\text{loc}(L_j))$ intersects with $RR(R,x,\text{loc}(L_k))$ for $1 \leq j \leq n, 1 \leq k \leq n$.

For interrogation of individual x-vector components, we make use of the x-vectors $UNIT(x,L)$, which have zero in all but the Lth coordinate position.

The R-vector *AIF* indicates by its Lth coordinate the absolute infeasibility of removing the load or live exit L from R.

For each specific register r, the R-vector $RNA([R,]r)$ represents by its Lth coordinate the nonavailability of register r to the load or live exit L. These vectors (in contrast to *AIF*) are normally all zero prior to global assignment, because no specific assignments have occurred. A specific assignment to r in general rules out the use of r for certain loads of other items, which is reflected by modifying $RNA(r)$ appropriately.

For each basic block b in region R, we define the R-vector $NPT(b)$, whose Lth coordinate indicates that the data item x associated with the load or live exit L was not a potential terminal quantity for b at the outset of global assignment for R. Thus, by definition, $NPT(b)$ need not be updated as allocation proceeds in R.

7. Global assignment for a single data item in an innermost region

The essence of the global assignment for a data item x in a region R may be seen by considering the case in which R has no subregions.

As can be observed in sections 4 and 5, the central operation in global assignment is that of moving a single load (or live exit) L of data item x from R. As described in section 5, this involves finding a register r available in $RR(R,x,\text{loc}(L))$. It may be, however, that another load L' of x has a register requirement set intersecting that of L. In this case we would, in order to move both L and L', need a common register for both unless register move instructions were introduced. Since the latter alternative may nullify the benefit of load motion, we reject it, though a few register moves may be required during local assignment.

Given any set U of loads of x to be moved, let U be partitioned into sets $U_1, \cdots, U_m$ such that for each k there is a unique register r_k to which x has been assigned at $\text{loc}(L)$ for all $L \in U_k$. We call the set $P = \{(r_1,U_1), \cdots, (r_m,U_m)\}$ a *direct partial assignment* for x. The general requirement to avoid register moves is that P be *consistent*, in the sense that if L and L' are in distinct U_k, then $RR(R,x,\text{loc}(L))$ and $RR(R,x,\text{loc}(L'))$ are disjoint.

Consider now the problem of moving a load L of x not in U. If $RR(R,x,\text{loc}(L))$ is disjoint from all of the U_k, any register available in $RR(R,x,\text{loc}(L))$ may be chosen. In general, however, we must do some merging to preserve consistency. For convenience we order the set P so that, for some index l $(0 \le l \le m)$, U_k intersects $RR(L,x,\text{loc}(L))$ for $1 \le k \le l$ but not for $l < k \le m$. Let $U' = \{L\} \cup U_1 \cup \cdots \cup U_l$. It is necessary for consistency to find a common register r' for all members of U'. The members $(r_1,U_1), \cdots, (r_l,U_l)$ of P are then replaced by the single member (r',U'). We call this process *connecting L to p.*

Let us consider how to find r'. We assume this is done simply by trying all registers in a specific order, called the *register trial sequence* for x in R. This sequence need not include registers inappropriate for x (such as floating-point registers if x is not a floating-point number). Let $s_1, \cdots, s_n$ be this sequence. Some criteria for choosing the sequence for different data items are discussed in section 9. It is convenient to represent the registers in a direct partial assignment for x by their indices in the register trial sequence for x. We call the result an *indirect partial assignment.* In these terms, the problem we are addressing is how to find j' such that register $s_{j'}$ is available in $RR(R,x,\text{loc}(L'))$ for all $L' \in U'$. Of course, this could be done simply by trying $j' = 1, \cdots, n$. However, if we let j_k be the register trial sequence index of r_k for $1 \le k \le l$, it suffices to try $j' = \max(1,j_1,\cdots,j_l), \cdots, n$. For if $j_k = \max(j_1,\cdots,j_l)$, then, because the same register trial sequence was used in obtaining j_k, it is the minimum index for which s_{j_k} is available for the elements of U_{j_k}, so no smaller index can suffice for all elements of the larger set U'.

Now we are ready to summarize the connecting algorithm discussed above in a more specific form, using the bit vectors described in section 6. Given a register trial sequence $s_1, \cdots, s_n$, a partial assignment $P = \{(j_1,U_1), \cdots, (j_m,U_m)\}$ for x, where the U_i are interpreted as x-vectors, and a load or live exit L of x, the procedure CNCT (L,P) attempts to connect L to P, i.e., to replace P by a partial assignment including L.

Procedure CNCT (L,P)

```
1. Order $P$ so that for some $l$ $(0 \le l \le m)$,
   $U_k \wedge INT(x,L) \ne 0$ for $1 \le k \le l$,
   and $U_k \wedge INT(x,L) = 0$ for $l < k \le m$.
2. Let $U' = UNIT(x,L) \vee U_1 \vee \cdots \vee U_l$.
3. For $k = \max(1,j_1,\cdots,j_l),\cdots,n$, do:
   if $U' \wedge RNA(s_k,x) = 0$, then replace
   $P$ by $\{(j_{l+1},U_{l+1}),\cdots,(j_m,U_m),(k,U')\}$ and return.
4. Return in failure ($L$ cannot be connected to $P$).
```

Given a register trial sequence s for x, load motion for x is accomplished simply by initializing the partial assignment P to the empty set and successively invoking CNCT(L,P) for the loads L of x that are not absolutely infeasible (i.e., are such that $UNIT(x,L) \wedge AIF(x) = 0$).

The motion of a store S is done as follows:

1. If there is a load L of x such that AFFECT(loc(S)), loc(L),R,x) and L could not be removed during load motion, S cannot be removed.

2. Otherwise let $L_1, \cdots, L_k$ be the live exits (and loads, if any) satisfying AFFECT(loc(S),loc(L_j),R,x), which may still be movable, and let P be the current partial assignment (based on load-store motion up to this point).
3. Let $P' = P$, and apply CNCT(P',L_j) for $j = 1, \cdots, k$.
4. If each application was successful, S can be removed. Let $P = P'$.
5. Otherwise S cannot be removed.

After load-store motion and assignment have been determined for x, the reservations implied by this assignment must be carried out. This involves

1. Allocation by counting and
2. Assignment of specific registers at block boundaries.

In addition to the information recorded locally (e.g., *RAVAIL* and the assigned result registers), it is necessary to update the *AIF* vector (based on step 1) and the *RNA* vectors (based on step 2), because these are what control the load-store motion and assignment for subsequent data items.

Allocation by counting takes one of three forms for data item x in a block b:

1. *Initializing* This occurs when x is a potential initial quantity for b and the corresponding load is removed. It involves decrementing $RAVAIL(p)$ for each point p from the beginning of b to the first register point of x.
2. *Terminalizing* This occurs when x is a potential terminal quantity for b and its last register point in b is used to remove a load or live exit elsewhere. It involves decrementing $RAVAIL(p)$ for each point p from this point to the end of b.
3. *Passing through* This occurs when x has no register point in b, but b is transparent and is on a track from such a register point elsewhere to a load or live exit being removed. It involves decrementing $RAVAIL(p)$ for all points p in b.

Allocation by counting in block b can affect the absolute infeasibility status, as reflected by the R-vector *AIF*, in one or more of the following three ways when *RAVAIL* goes to zero at some point in b:

1. One or more data items can cease to be potential initial quantities for b. For each such data item x, $RRB(b,x)$, which contains only the single load L of x in b, is ORed into $AIF(x)$, because L is now absolutely infeasible.
2. One or more data items can cease to be potential terminal quantities for b. For each such data item x, $RRE(b,x)$ is ORed into $AIF(x)$, because this causes any load L of x that has the end of b in $RR(R,x,\text{loc}(L))$ to become absolutely infeasible.
3. Block b may cease to be transparent. In this case $RRE(b) \wedge NPT(b)$ is ORed into *AIF*, because this causes any load L of an item x that is not referenced in b to become absolutely infeasible if the end of b is in $RR(R,x,\text{loc}(L))$. Note that if x is referenced in b but is not a potential terminal quantity of b, L would already be absolutely infeasible, so that the bit vector operation is still correct.

Let $A = \{(r_1,U_1), \cdots, (r_m,U_m)\}$ be the assignment generated by load-store motion for x, where the x-vector U_k represents the loads and live exits of x that were assigned to register r_k. The reservation implied by A is carried out by performing the procedure RSRV(b) for each block b in R.

```
Procedure RSRV (b)
For j = 1 to m do:
1. If $U_j \wedge RRE(b,x) \neq 0$ then (the last register point of
   x in b was used in moving one of the loads of x) do:
   a. If x is a potential terminal quantity in b then do:
      i. Terminalize x in b.
      ii. Assign $r_j$ as the result register of the instruc-
          tion last defining x in b.
      iii. OR $RRE(b)$ into $RNA(r_j)$.
      iv. Update AIF as required (see 1-3 above):
   b. Else (x is not referenced in b) do:
      i. Pass x through b.
      ii. Reserve $r_j$ as a pass-through register of b.
      iii. OR $RRE(b) \vee RRB(b)$ into $RNA(r_j)$.
      iv. Update AIF as required (see 1-3 above).
      v. Return.
2. If $U_j \wedge RRB(b,x) \neq 0$ then (a load of x in b was
   moved) do:
   a. Initialize x in b.
   b. Assign $r_j$ as the result register of the initial load of
      x in b, which, though it is marked for deletion, is
      left as a place holder for local assignment.
   c. OR $RRB(b)$ into $RNA(r_j)$;
   d. Update AIF as required (see 1-3 above).
```

8. Computation of RRE, RRB, and INT vectors

The first step in computing the *RRB* and *RRE* vectors for a region R (as defined in section 6) is to select a set $\{e_1, \cdots, e_n\}$ of edges of the control flow graph of R so that the graph $\overline{R}$ resulting from the removal of the e_i from R has the following property: $\overline{R}$ has no closed tracks that are not totally contained in proper subregions of R. The method of [6] always produces regions for which this can be achieved for $n = 1$. For the present, we assume that R contains no proper subregions. Thus, the graph $\overline{R}$ is acyclic (has no closed tracks), and its

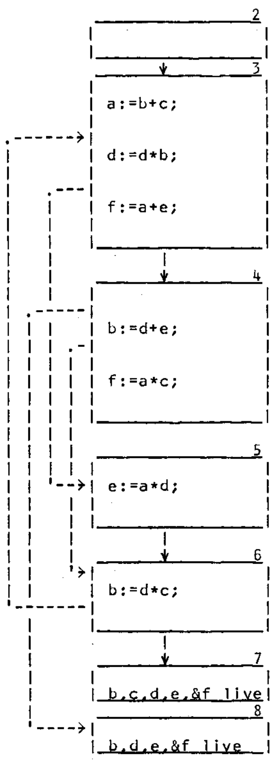
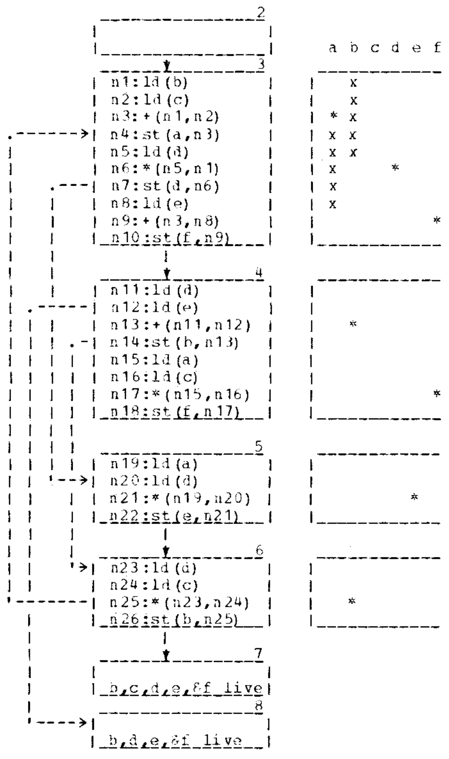


Figure 1 An example for global assignment in an innermost region (3–6).

Figure 2 Effect of local allocation on status.



nodes (basic blocks) can be topologically sorted, i.e., given a linear order $b_1, \cdots, b_m$, such that if b_i is a predecessor of b_j, then $i < j$.

Then apply the following steps:

1. For $1 \leq j \leq m$ do:
   a. Set the Lth component of $RRB(b_j)$ to 1 if L is a potential initial load in b_j, else to 0 (the beginning of b is in $RR(R,\text{item}(L),\text{loc}(L))$ if item(L) is a potential initial quantity of b).
   b. Set the Lth component of $RRE(b_j)$ to 1 if L is a live exit corresponding to the end of b_j, else to 0 (the end of b is in $RR(R,\text{item}(L),\text{loc}(L))$ if it is the exit corresponding to L).
2. Repeat $n + 1$ times:
   a. For $j = m, m - 1, \cdots, 1$ do;
      i. OR $RRB(b_k)$ into $RRE(b_j)$ for each successor b_k of b_j in R, including those corresponding to removed edges (if the beginning of b_k is in $RR(R,\text{item}(L),\text{loc}(L))$, then so is the end of b_j).
      ii. OR $RRE(b_j) \wedge NPT(b_j)$ into $RRB(b_j)$ (if the end of b is in $RR(R,\text{item}(L),\text{loc}(L))$, then so is the beginning of b if item(L) is not referenced in b).

This algorithm strongly resembles that used in [5] to compute the bit vectors for global redundancy elimination, and it has a similar justification. See also [7], in which it is viewed as solving linear Boolean equations. The algorithm results, in general, in register requirement sets that are larger than those defined in section 5, because an item may not be a potential terminal quantity of b even though it is referenced in b. However, this can only occur for a load or live exit that is absolutely infeasible, so that there is no net effect of the inaccuracy.

For its application in store motion, the affect (R,x) relation is conveniently represented by an x-vector $AFCT(b,x)$ for each block b of the region R whose Lth coordinate has the truth value of the relation AFFECT (end(b),loc(L),R,x). Such a vector is required only if x is stored in R. By appending $AFCT(b,x)$ to $RRE(b)$ for all such x, and by appending similar x-vectors $AFCTNPT(b,x)$ to $NPT(b)$ and $AFCTRRB(b,x)$ to $RRB(b)$ and initializing them appropriately, the above iterations also compute the $AFCT(b,x)$ vectors. Specifically, $AFCT(b,x)$ and $AFCTRRB(b,x)$ are initialized exactly as the corresponding vectors $RRE(b,x)$ and $RRB(b,x)$ are in step 1 above, whereas $AFCTNPT(b,x)$ is set to all zeros if x is redefined in b and to all ones otherwise.

The AIF vector is first set to all zeros if we assume that only loads corresponding to potential initial quantities have R-vector coordinates. Then, for each block b in R that is not transparent, $RRE(b) \wedge NPT(b)$ is ORed

```
         _________________2    component:register requirement      /affect
        |                  |   variable: a  b   c    d      e    f /a  b   d      e    f
        |__________________|   ref.blk.: 45 378 3467 345678 3478 78/45 378 345678 3478 78
         ________V________3
        | n1:ld(b)         |   RRB(3)    00 100 1000 100000 1000 00/00 100 100000 1000 00
        | n2:ld(c)         |
        | n3:+(n1,n2)      |
.------>| n4:st(a,n3)      |
|       | n5:ld(d)         |
|       | n6:*(n5,n1)      |
|   .---| n7:st(d,n6)      |
|   |   | n8:ld(e)         |
|   |   | n9:+(n3,n8)      |
|   |   |_n10:st(f,n9)_____|   RRE(3)    00 000 0000 000000 0000 00/00 000 000000 0000 00
|   |            |             NPT(3)    00 000 0000 000000 0000 00/00 111 000000 1111 00
|   |    ________V________4
|   |   | n11:ld(d)        |   RRB(4)    10 000 0100 010000 0100 00/10 000 010000 0100 00
| .-----| n12:ld(e)        |
| | |   | n13:+(n11,n12)   |
| | | .-| n14:st(b,n13)    |
| | | | | n15:ld(a)        |
| | | | | n16:ld(c)        |
| | | | | n17:*(n15,n16)   |
| | | | |_n18:st(f,n17)____|   RRE(4)    00 001 0000 000001 0001 01/00 001 000001 0001 01
| | | |                        NPT(4)    00 000 0000 000000 0000 00/11 000 111111 1111 00
| | | |  _________________5
| | | | | n19:ld(a)        |   RRB(5)    01 000 0000 001000 0000 00/01 000 001000 0000 00
| | '-->| n20:ld(d)        |
| |   | | n21:*(n19,n20)   |
| |   | |_n22:st(e,n21)____|   RRE(5)    00 000 0000 000000 0000 00/00 000 000000 0000 00
| |   |          |             NPT(5)    00 111 1111 000000 0000 11/11 111 111111 0000 11
| |   |  ________V________6
| |   '>| n23:ld(d)        |   RRB(6)    00 000 0010 000100 0000 00/00 000 000100 0000 00
| |     | n24:ld(c)        |
'-------| n25:*(n23,n24)   |
  |     |_n26:st(b,n25)____|   RRE(6)    00 010 0001 000010 0010 10/00 010 000010 0010 10
  |              |             NPT(6)    11 000 0000 000000 1111 11/11 000 111111 1111 11
  |      ________V________7
  |     |                  |
  |     |_b,c,d,e,&f_live__|
  |      _________________8
  '---->|                  |
        |_b,d,e,&f_live____|
```

Figure 3 Vectors as initialized prior to iteration for RRB's and RRE's.

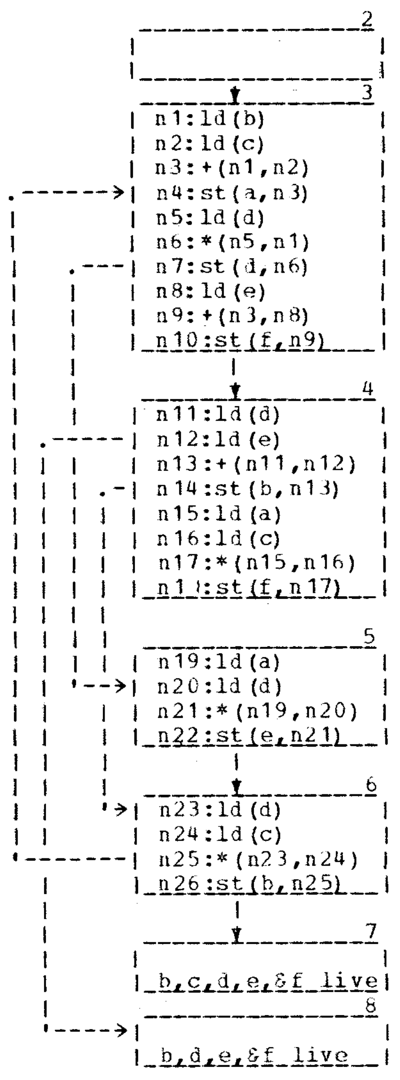
into *AIF*, because if *RR*(*R*,item(*L*),loc(*L*)) contains the end of *b*, then *L* is absolutely infeasible unless item (*L*) is a potential terminal quantity for *b*.

The intersection vectors are computed as follows:

1. Initialize *INT*(*x*,*L*) to all zeros for all *x* and all loads *L* of *x* in *R*.
2. For each block *b* and each data item *x* in *R* do:
   a. Set *RREI* = *RRE*(*b*,*x*).
   b. If *x* is both a potential initial and a potential terminal quantity in *b* corresponding to a single load *L* in *b*, then OR *RRB*(*b*,*x*) into *RREI*. (At this point, *RREI* represents the set of loads of *x* whose register requirement sets have a common point in block *b*.)
   c. For each component *L* of an *x*-vector, if the *L*th component of *RREI* = 1, then OR *RREI* into *INT* (*x*,*L*).

9. Example

The global assignment algorithm has been described for a region with no subregions. The reader may now consider its application to the example shown in Figs. 1 and 2. The region *R* consists of the blocks 3, 4, 5, and 6. By removing the edge from block 6 to block 3, all closed tracks in *R* are eliminated, and the blocks are topologically sorted. The only entry block is 3, with entry target 2. Both 4 and 6 are exit blocks with exit targets 8 and 7, respectively. The indicated control flow would, of



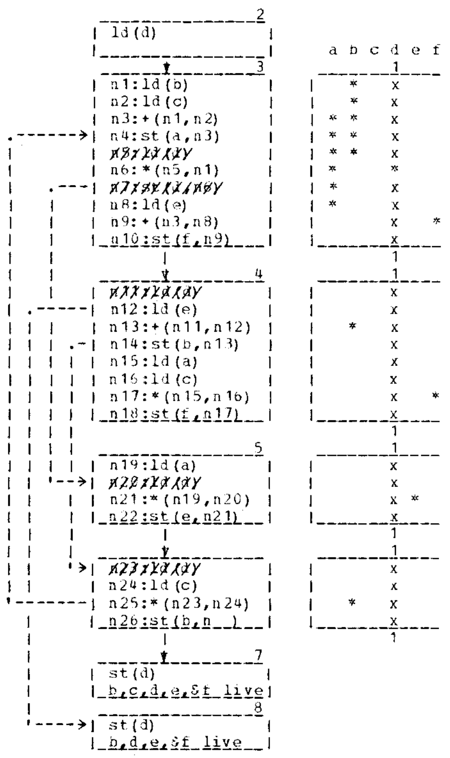
```
2

3
n1:ld(b)
n2:ld(c)
n3:+(n1,n2)
n4:st(a,n3)
n5:ld(d)
n6:*(n5,n1)
n7:st(d,n6)
n8:ld(e)
n9:+(n3,n8)
n10:st(f,n9)

4
n11:ld(d)
n12:ld(e)
n13:+(n11,n12)
n14:st(b,n13)
n15:ld(a)
n16:ld(c)
n17:*(n15,n16)
n18:st(f,n17)

5
n19:ld(a)
n20:ld(d)
n21:*(n19,n20)
n22:st(e,n21)

6
n23:ld(d)
n24:ld(c)
n25:*(n23,n24)
n26:st(b,n25)

7
b,c,d,e,&f live
8
b,d,e,&f live
```

```
component:register requirement       /affect
variable: a  b   c    d      e    f /a  b   d      e    f
ref.blk.: 45 378 3467 345678 3478 78/45 378 345678 3478 78

RRB(3)    00 100 1000 100000 1000 00/00 100 100000 1111 00

RRE(3)    11 000 0110 011000 0100 10/11 000 111111 1111 10

RRB(4)    10 000 0100 010000 0100 00/10 000 110111 1111 00

RRE(4)    00 001 0010 000101 1011 11/00 001 100111 1111 11

RRB(5)    01 000 0010 001000 0000 10/01 000 101110 0000 10

RRE(5)    00 000 0010 000100 1010 10/00 000 100110 1111 10

RRB(6)    00 000 0010 000100 1010 10/00 000 100110 1111 10

RRE(6)    00 110 1001 100010 1010 10/00 110 100010 1111 10
```

Figure 4 Results of iteration for RRB's and RRE's.

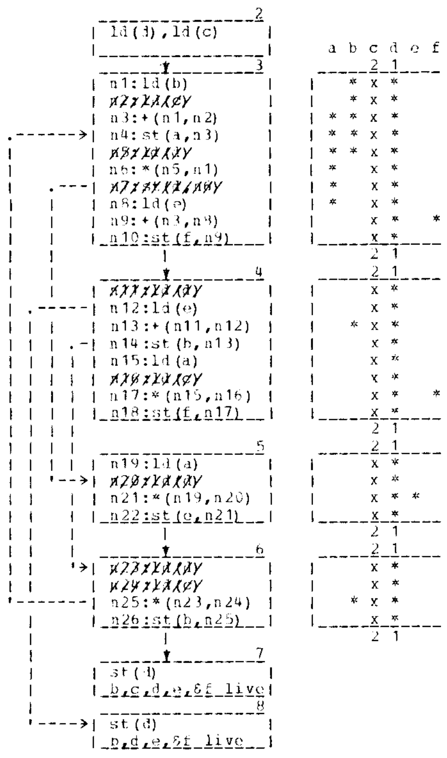
course, require branches in addition to the instructions shown. The variables that are live on entry to the exit targets are listed. The box to the right of each block in R is used in Figs. 2 and 5-10 to illustrate the allocation and assignment status of the block. A character under a particular variable v in such a box indicates that a register has been allocated for v at the corresponding point. New allocations (i.e., changes from the prior state depicted) are indicated by x's, old ones by *'s. We assume an object machine with four registers of a single type. The register availability count at any point is thus four minus the number of characters in the corresponding row of the appropriate box. Figure 2 shows the effect of local allocation: a and b have been maintained in registers between successive references in block 3. A global assignment of v to register r is indicated by r at the appropriate block boundary (beginning or end) under the v.

Figures 3 and 4 show the bit vectors before and after application of the algorithm of section 8 for computing *RRB* and *RRE* with $n = 1$. In addition to the register requirement components as defined in section 6, we have included, to the right of the slash, the *affect* component, as suggested in section 8, for computing the affect relation needed for store motion. Each R-vector (section 6) is separated by spaces into its component x-vectors. An x-vector is labeled by variable, and its bit positions are



```
variable a  b   c    d      e    f
ref.blk. 45 378 3467 345678 3478 78

AIF      00 000 0000 000000 0000 00
RNA(1)   xx xxx xxxx xxxxxx xxxx xx
RNA(2)   00 000 0000 000000 0000 00
RNA(3)   00 000 0000 000000 0000 00
RNA(4)   00 000 0000 000000 0000 00
```

Figure 5 Results of processing d; register trial sequence = 1,2,3,4.



```
variable a  b   c    d      e    f
ref.blk. 45 378 3467 345678 3478 78

AIF      00 000 0000 000000 x000 00
RNA(1)   11 111 1111 111111 1111 11
RNA(2)   xx xxx xxxx xxxxxx xxxx xx
RNA(3)   00 000 0000 000000 0000 00
RNA(4)   00 000 0000 000000 0000 00
```

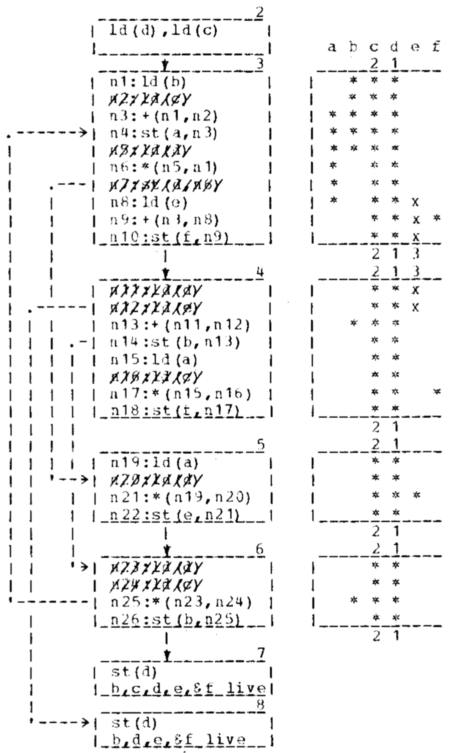
Figure 6 Results of processing c; register trial sequence = 2,3,4,1.

labeled by the numbers of the blocks of the corresponding reference (load or live exit) of that variable. Thus, a c-vector has positions labeled 3, 4, and 6, corresponding to the loads of c in blocks 3, 4, and 6, and a position labeled 7, corresponding to the live exit of c at the exit target 7. No c-vector is included in the affect component, because the affect relation is not required for variables not stored in R. The live exit of c is not required, either, for processing R, because it is only needed for store motion.

Figures 5 through 9 show the results of applying the algorithm of section 7 to the variables d, c, e, a, b, and f, in that order. The application to variable a results in no change, so it is not shown. The register trial sequences (section 7) are shown for each application. They have been chosen arbitrarily for this example. The order in which the loads of a variable are processed is immaterial in this example. Changes from 0 to 1 in the reservation vectors AIF and RNA are indicated by x's. Loads and stores that have been removed are overstruck with slashes.

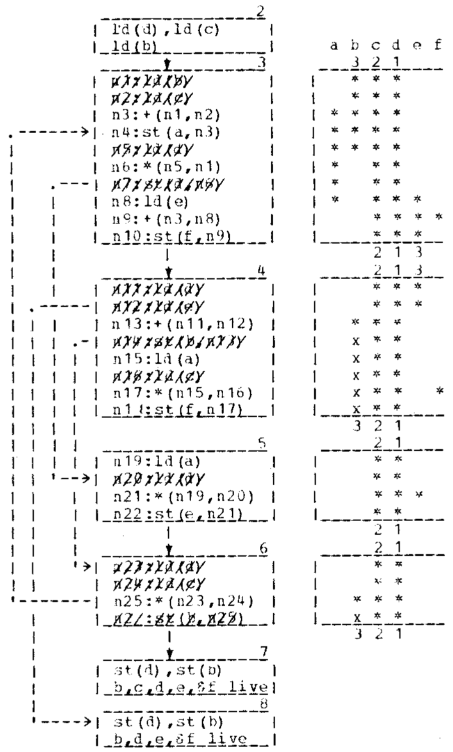
We discuss the processing of d in detail. In set theoretic notation, using the block designation for loads and live exits of d, we have (from the algorithm of section 8):

$$\text{INT } (d,3) = \{3,6,7\}$$
$$\text{INT } (d,4) = \{4,5\}$$
$$\text{INT } (d,5) = \{4,5,6\}$$
$$\text{INT } (d,6) = \{3,5,6,7,8\}$$
$$\text{INT } (d,7) = \{3,6,7\}$$
$$\text{INT } (d,8) = \{6,8\}$$



```
variable a  b   c    d      e    f
ref.blk. 45 378 3467 345678 3478 78

AIF      xx 000 0000 000000 1000 00
RNA(1)   11 111 1111 111111 1111 11
RNA(2)   11 111 1111 111111 1111 11
RNA(3)   xx 000 0xx0 0xx000 0x00 x0
RNA(4)   00 000 0000 000000 0000 00
```

Figure 7 Results of processing e; register trial sequence = 3,4,1,2.

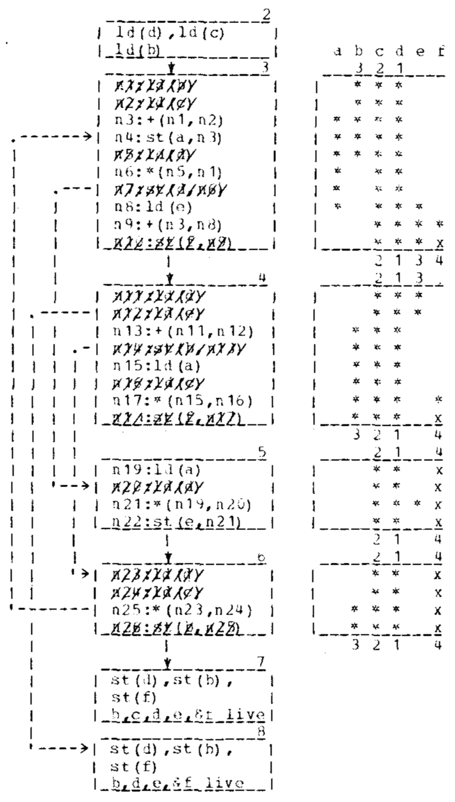


```
variable a  b   c    d      e    f
ref.blk. 45 378 3467 345678 3478 78

AIF      11 000 0000 000000 10xx 00
RNA(1)   11 111 1111 111111 1111 11
RNA(2)   11 111 1111 111111 1111 11
RNA(3)   11 xxx x11x x11xxx x1xx xx
RNA(4)   00 000 0000 000000 0000 00
```

Figure 8 Results of processing b; register trial sequence = 3,4,1,2.

Observe that $RR(R,d,\text{loc}(3))$ and $RR(R,d,\text{loc}(6))$ must be regarded as intersecting because the load in block 3 can only be moved by using the register point of d after the load in block 6, so that by the principle of consistency (section 7) they must be assigned the same register. Note that AIF and RNA are initially zero and the register trial sequence s is (1,2,3,4,). We move the loads of d in numerical order. Initially the partial assignment P is empty. The application of CNCT(3,P) returns with $P = \{(1,\{3\})\}$. In applying CNCT(4,P) we get, in step 1, $l = 0$, so in step 2, $U' = \{4\}$, and step 3 returns with $P = \{(1,\{3\}),(1,\{4\})\}$. On return from CNCT(5,$P$), $P = \{(1,\{3\}),(1,\{4,5\})\}$, and after CNCT (6,$P$), $P = \{(1,\{3,4,5,6\})\}$. The only store of d in R is in block 3, and it is in the affect (R,d) relation to all loads and live exits of d, as seen from the affect component of $RRE(3,d)$ in Fig. 4. Because all the loads have been successfully moved, the store motion procedure calls for the attempted motion of the live exists 7 and 8, using CNCT. These are successful, and the final value of P is $\{(1,\{3,4,5,6,7,8\})\}$. The load store motion for d is completed by following the instructions in section 5, which call for a load of d to be inserted at the end of block 2 and stores at the beginnings of blocks 7 and 8. The reservation for d now is accomplished by a straightforward application of RSRV(b) for $b = 3,4,5,6$, which has the effect of initializing and terminalizing d in each block. The effects on $RAVAIL$, AIF, and RNA are shown in Fig. 5.

In Fig. 6, observe that the one occurring in AIF for the load of e in block 3 is a result of e ceasing to be a potential initial quantity in this block. In Fig. 7, the ones in AIF for both loads of a result from its ceasing to be a potential terminal quantity for block 3.



```
variable a  b   c    d      e    f
ref.blk. 45 378 3467 345678 3478 78

AIF      11 000 0000 000000 1011 00
RNA(1)   11 111 1111 111111 1111 11
RNA(2)   11 111 1111 111111 1111 11
RNA(3)   11 111 1111 111111 1111 11
RNA(4)   xx xxx xxxx xxxxxx xxxx xx
```

Figure 9 Results of processing f; register trial sequence = 3,4,1,2.

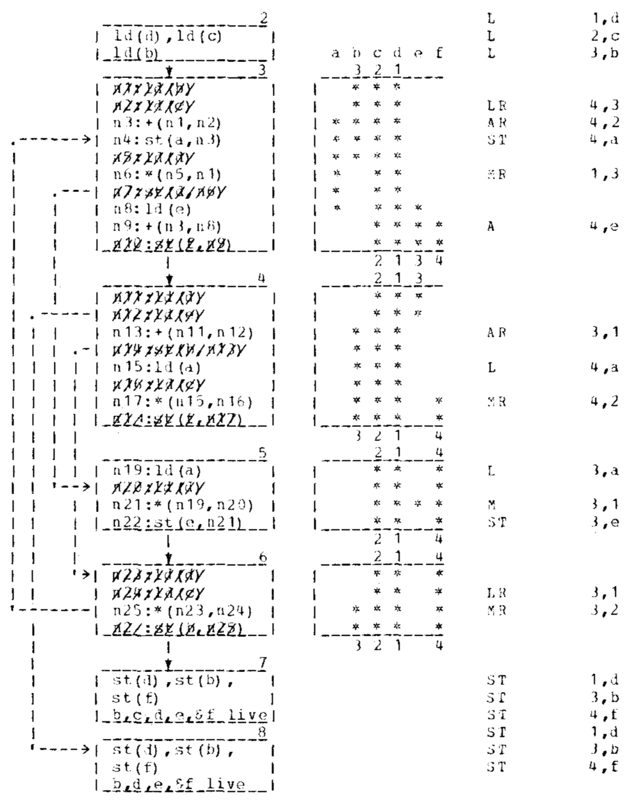


Figure 10 Generated code.

Finally, Fig. 10 shows System/360-like code generated during the local assignment phase. A register trial sequence for *e* beginning with register 4 would have inhibited the subsequent motion of the stores of *f*.

The interested reader is encouraged to try this example with variations in the number of registers, the order of processing of variables, and the register trial sequences.

10. Global assignment for outer regions

Suppose region R immediately contains region R'. Then when global assignment for R takes place, it has already been completed for R', at least in the sense that all loads and stores that are to be removed from R' have been removed. While global assignment for R proceeds basically as described in sections 7 and 8, the presence of R' must be accounted for in each of the three principal subphases—vector computation, load-store motion, and reservation.

With respect to load-store motion, the completed assignment of R' to some extent forces, or implies, certain specific assignments in R, because of the principle of consistency introduced in section 7. In the first place, any load inserted into an entry target of R' upon removal from R' clearly has its assignment implied by that in R'. In the second place, if there is a register point in R' at which x has a specific assignment and which is to be used in removing a load L of x from R, then by the principle of consistency the assignment of L would be implied. In more formal terms, suppose P' is the final assignment of x in R', L is a load (or live exit) of x in R, q' is an exit of R', q is the exit target of q', L' is the corresponding live exit for x, and q is in $RR(R,x,\text{loc}(L))$. If there exists (r,U) in P' such that $INT(x,L') \wedge U \neq 0$, then L has the implied assignment r in R.

Other situations, such as fixed subroutine interfaces, may have a similar effect. Whatever the source, we express this effect for each data item x in the same form as a partial assignment for x in R. Thus an *implied assign-*

ment for x in R is a set $IA = \{(j_1,U_1), \cdots, (j_m,U_m)\}$, where the j_k are indices of the register trial sequence s for x in R and U_k is the (R) x-vector representing the loads and live exits of x in R that must be assigned to s_{j_k} if they are removed.

The application of the above criterion for determination of the implied assignment may lead to inconsistency, i.e., a load L common to U_k and U_l with $j_k \neq j_l$. However, this situation is assumed to have been handled by making L absolutely infeasible in R.

It is primarily in this context that the choice of the register trial sequence seems significant, because it can be used to reduce the likelihood of such inconsistency arising. Suppose, for instance, that prior to any global assignment, we determine the processing order of the data items in each region, using reference frequency or a similar criterion. Let N be the number of registers in the object machine. Provisionally assign the first N data items (in processing order) of the outermost region to registers 1 through N. Then repeat the following process for all remaining regions S' in an outer-to-inner order: For each of the first N data items x of S', if x has a provisional assignment in the region S immediately containing S', give it the same provisional assignment in S'; otherwise provisionally assign x to any register not provisionally assigned in S to any of the first N data items of S'. These provisional assignments, having been obtained prior to global assignment, can be used to determine the register trial sequences. In particular, whenever a data item x has a provisional assignment to register r in R, r should be first in the register trial sequence for x in any region R' immediately contained in R.

In the presence of an implied assignment $IA = \{(k_1,W_1), \cdots, (k_p,W_p)\}$ for x in R, the basic function performed by the CNCT procedure in load-store motion (section 7) is performed by the procedure CNCTIA (L,P,IA), where $P = \{(j_1,U_1), \cdots, (j_m,U_m)\}$ is the current partial assignment for x in R, s the register trial sequence for x in R, and L the load (or live exit) being moved.

Procedure CNCTIA(L,P,IA)

```
1. First determine whether the assignment of $L$ is
   forced, i.e., whether $UNIT(x,L) \wedge W_q \neq 0$ for some $q$
   $(1 \leq q \leq p)$. If so, then do the following:
   a. Partition $P$ as in CNCT so that for some $l$
      $(0 \leq l \leq m)$
      $U_i \wedge INT(x,L) \neq 0$ for $1 \leq i \leq l$, and
      $U_i \wedge INT(x,L) = 0$ for $l < i \leq m$.
   b. If $j_i > k$ for some $i \leq l$, then return in failure
      (since the implied assignment would have already
      been tried in vain for one of the intersecting loads).
   c. Let $U' = UNIT(x,L) \vee U_1 \vee \cdots \vee U_l$.
   d. If $U' \wedge RNA(s_{k_q},x) = 0$ then do:
      i. Replace $P$ by $\{(j_{l+1},U_{l+1}), \cdots, (j_m,U_m), (k_q,U')\}$.
      ii. OR $INT(x,L)$ into $W_q$.
      iii. Make inconsistent implied assignments in-
           feasible, i.e., OR $W_q \wedge W_i$ into $AIF(x)$ for
           $1 \leq i \leq p, i \neq q$.
      iv. Return.
   e. Return in failure.
2. Apply CNCT$(L,P)$.
```

In addition to register reservation in blocks immediately contained in R, as described in section 7, the load-store motion in R may require additional reservation in the contained region R'. This would be the case, for instance, if an exit target q of R' were in $RR(R,x,\text{loc}(L))$, where L is a load (or live exit) of x moved from R, and no register had been reserved for x at the corresponding exit. We call this *secondary reservation* in R', and it has one of two forms:

1. Data item x may be *passed through* R', i.e., a register is reserved for x throughout R'. This can only happen when R' is transparent and x has no register point in R' but is required to be in a register at some exit of R'.
2. Data item x may require a *partial reservation* in R'. This can happen when x has a register point in R' and is required to be in a register at some exit of R' for which the corresponding live exit of x is "uncovered," i.e., cannot be removed based on prior reservations in R'.

While passing through can be handled rather simply, secondary partial reservation requires essentially the same status information and processes as primary reservation (as discussed in section 7). For this reason it may be preferable simply to suppress the motion of any load in R that would necessitate secondary partial reservation in a contained region. This could be done by making such loads absolutely infeasible. This key to carrying out a secondary partial reservation to produce x in a register at an exit q' of R' is provided by the live exit L' of x corresponding to q'. (If x is not live at the exit target q of q', there could be no load-store motion in R requiring x to be in a register at q.) Specifically, the reservation is just that required to remove L' from R', i.e., a register is needed in the set $S = RR(R',x,\text{loc}(L'))$.

Next we consider the role that the contained region R' plays in the computation of the RRB, RRE, and INT vectors for R. For most purposes R' can be considered as a node (basic block) of R whose predecessors and successors are the entry and exit targets of R'. It may be regarded as one of the b_j in the linear order defined in section 8. If no secondary partial reservation is being made, the basic algorithm for computing RRB and RRE (section 8) can stand unchanged, given only that the vectors are properly initialized for the nodes corresponding to contained regions. Specifically, if b is the node for R', then $RRB(b)$ and $RRE(b)$ are initialized to zero and $NPT(b)$ is set to one in coordinate L if and only if item (L) is not referenced in R'.

In case secondary partial reservation is used, the role of NPT for the contained region R' is played by an R-vector $JUMP$ for each entry-exit pair of R'. Let p' be an entry of R' and p its entry target in R. Let $q_1', \cdots, q_k'$ be the exits of R' and $q_1, \cdots, q_k$ their exit targets in R. We define the R-vector $JUMP(p,q_j)$ with a one in component L if L corresponds to data item x and the R'-vector $RRB(p')$ has a one in the component corresponding to the live exit of x at q_j. Then the iteration for RRE at block p has the form

OR $JUMP(p,q_j) \wedge RRB(q_j)$ into $RRE(p)$ for $j = 1, \cdots, k$.

For the purpose of computing the $INT(x,L)$ vectors, the algorithm of section 8 may be used with the contained region R' treated as one would a basic block of R. Although with a more refined method the assignment constraints could be somewhat relaxed, the net effect is probably insignificant.

The RNA vectors for R must, of course, reflect the register availability in R' relevant to loads and live exits in R. In case secondary partial reservation is not being done, this can be accomplished as follows (where the q_j and q_j' are as defined above):

For each register r do:

1. Initialize $RNA(R,r)$ to 0.
2. For each data item x (referenced in R), if x is not referenced in R' and r is reserved anywhere in R', then OR $RRB(q_j)$ into $RNA(R,r)$ for $1 \leq j \leq k$.
3. If x is referenced in R', then for $j = 1$ to k do:
   a. If x is not assigned to r at q_j', then OR $RRB(q_j,x)$ into $RNA(R,r,x)$.

If secondary partial reservation is being done, the procedure is more tedious and depends on the $RNA(R',r)$.

The effect of R' on the initialization of the AIF vector is determined in an analogous way.

11. Logical constraints of load-store motion

Until now we have assumed, for simplicity of exposition, a one-to-one fixed relation between data items to be assigned to registers and their storage locations. Thus the only constraint involved in load-store motion was the physical one of register availability. One would like to be able to apply these techniques to dynamically addressed (pointer or subscript qualified) data items, which do not satisfy the fixed one-to-one relation with storage; e.g., $A(I)$ may refer to different storage locations at different times and may refer to the same storage location as $A(J)$. This requires the imposition of certain logical constraints as well as the physical one already dealt with. These constraints should also cover the possibility of aliasing among data items. Finally the problem of *safety* must be faced in order to move loads and stores of dynamically addressed items; safety here means the possibility of introducing unwanted side effects due to a change in conditionality of execution of the moved instruction. These are serious problems for all forms of compiler optimization, particularly the global ones. The aim of this section is not to present general solutions for these problems, but to show how the above global assignment algorithm can effectively make use of such solutions.

Consider the case of a subscripted variable $A(I)$. It is possible to apply machine-independent methods of expression commoning and backward motion to remove the loads of $A(I)$ (see [7], for example.) This involves introduction of a scalar temporary to carry the value of $A(I)$ from one point of reference to a subsequent one, independently of register availability. This is the strategy used in the FORTRAN H optimizer, in which, however, apparently no attempt is made to remove subscripted stores. Although this approach reduces the index register requirements, the effect is frequently outweighed by the additional stores required for the scalar temporaries. A more flexible approach involves removing only those loads and stores of $A(I)$ for which registers are available. This machine-dependent approach has the drawback that the potential index register saving is not only smaller but often cannot be conveniently realized, because the necessary information is not known soon enough. A test made to compare the two approaches indicated no significant difference (see section 13). Nevertheless, the basic machine-dependent method is now described, because with some elaboration, it has the potential of more promising results.

Each formally distinct subscripted variable is treated as a separate data item subject to load-store motion. Thus $A(I)$,$B(I)$, and $A(J)$ would all be considered as separate items. Assuming that global commoning and backward motion for expressions (but not for subscripted references) have been completed, if e is an expression other than a simple variable reference, a reference to $A(e)$ need not be considered, at least for load motion. For this would be possible only if e were redundant, in which case the reference presumably would have been changed to the form $A(T)$.

For the purpose of load motion for a data item $x = A(I)$, the absolute infeasibility vector AIF may be used to record the logical constraints as follows:

For any block b of the region R in which x is not a potential terminal quantity, we OR $RRE(b,x)$ into $AIF(x)$ if either I or $A(I)$ could be modified by the execution of b. The latter alternative would in general obtain if, for instance, $A(J)$ or $A(e)$ were stored in b. We have begged the question slightly in the use of the notion of potential terminal quantity. In section 5 this, as well as potential initial quantity, was defined in terms of register availability alone. For a subscripted item x, this concept should include the logical identity of x at the end of b (at the beginning in the case of an initial quantity). In other words, x is a potential terminal quantity for b if there is an instruction i referencing x in b and if a load of x could correctly be moved from the end of b back to loc(i). Similarly, x is a potential initial quantity for b if there is a load i of x in b that can be correctly moved back to the beginning of b. In both cases we require, as in section 5, that the available register counts along the course of motion are positive.

Store motion for $A(I)$ requires data in addition to that described in section 5. For the simple case addressed there, a store S of item x could be removed from region R if each load and live exit of x to which S was in the affect (R,x) relation could be removed from R. The additional requirement for the case $x = A(I)$ can be satisfied by allowing for a dummy load of x at any point of R at which a modification of I or either a use or a modification of $A(I)$ under any other name (e.g., $A(J)$) could occur. These dummy loads are to be considered immovable from R. Thus if S is in the affect (R,x) relation to any such load L, then S is not movable from R. These dummy loads of x can be represented collectively by a single bit position, which we denote as $SLI(b,x)$ for store logically immovable, appended to each $AFCT(b,x)$ vector as defined in section 8. Thus if $SLI(b,x)$ is on, the end of b is in the affect (R,x) relation to some immovable dummy load and hence a store of x in b is logically immovable. By appropriate initialization the $SLI(b,x)$, as the $AFCT(b,x)$, are computed by the algorithm of section 8.

Other forms of dynamic qualification, such as based references, can be handled in exactly the same manner as subscripted references.

We have until now assumed that different variables correspond to disjoint storage locations. In this section, we used $A(I)$ and $A(J)$ as examples of possible aliases for the same storage location. Language constructs, such as EQUIVALENCE in FORTRAN, DEFINED or BASED in PL/I, REDEFINES in COBOL, or parameters in many languages, allow numerous other possibilities for aliasing. Moreover, calls to external routines can in certain instances implicity reference variables of the calling program. To do global optimization of any sort correctly, all possibilities for such side effects must be taken into account, if necessary by assuming worst cases. Our technique for handling subscripted references can be applied to reflect these possibilities in the bit vectors for global assignment. For example, if X and Y could be aliases for the same storage, they would be handled as $A(I)$ and $A(J)$ were above, in that a change of X could cause a load of Y to become absolutely infeasible, and either a use or a change of X could cause a store of Y to become logically immovable.

What has been shown thus far is how the global assignment algorithm can be applied to a data item x, given that for x and its dynamic qualifiers (subscripts or pointers) the program points are known at which the storage allocated to them can be used or modified. In the case of store motion for x in region R, it is generally necessary to know additionally that the storage for x is not deallocated in R.

The question of safety is relevant for a dynamically qualified reference of data item x that must be moved to a region entry or exit, at which it could be executed, even though it might not have been executed if left in place. This is because the qualifier may be invalid and cause an unwanted interrupt. Let p be an entry of region R. We assume it is known whether a load L of x would be unsafe at the corresponding entry target, i.e., whether such a load could cause an interrupt even though the original program could not. There are means of determining this in some cases, e.g., [8]. Should L be unsafe, we can inhibit the motion of any load or live exit of x requiring L simply by ORing $RRB(p,x)$ into $AIF(x)$. Similarly, if q is an exit of R, then we assume it is known whether a store of x would be unsafe at the corresponding exit target. If so, then if L is the live exit of x corresponding to q, a store S of x in R cannot be removed if loc(S) is in the affect (R,x) relation to loc(L). This fact can be reflected in the appropriate SLI bit by considering L to be a dummy immovable load.

12. Register assignment prototype

A prototype register assignment program was written to evaluate the strategy proposed in section 2 and consists of the three phases described there. To provide realistic input, a program was written to convert the internal text of the FORTRAN H compiler after the final optimization phase (phase 20) to a form acceptable to the prototype. The prototype generates IBM System/360 code from this input. The phase implementations in the prototype are described below.

• *Local allocation*

The local allocation phase retains the instruction order and merely attempts to keep data items in registers between successive references in the same basic block b, without making specific register assignments. This is done by keeping counts of available registers at each point (i.e., instruction) of b. Separate counts are kept for floating-point and general registers. These are initialized to the number of registers of the respective type in the machine. Then the basic register requirement for each instruction is subtracted. Normally, this amounts to one register of the appropriate type for the instruction at its corresponding point, although more are obviously required for certain instructions, such as fixed-point multiply. Thus we only account for the result register requirement at this point. The operand registers are accounted for later.

The instruction text is in the form described in section 3, with data references made only by means of loads or stores. Because some of the loads may eventually be effected by RX references, it is advisable not to assume any register requirement for such loads at this time. This assumption is valid as long as the loads immediately precede the instruction referencing their results. The subsequent register allocation corrects this assumption appropriately.

We define a *gap* of b as the interval between two successive references to the same data item in b, the second of which is not a store. The gaps of b are assigned priorities and sorted by increasing priority. Then for each gap g in order, we attempt to close g by decrementing the appropriate register counts in the interval g. When a register count of zero occurs in a gap, then it cannot be closed, in which case a load and possibly a store must be generated. The priority used for a gap is its length in terms of the number of intervening instructions. An exception to this is made when the gap is between a store of x and a unique load of x (x ceases to be live after this load). In this case we use half the gap length, because both the store and the load are removed by its closing.

Any optimization procedure that alters the instruction sequence should, for obvious reasons, precede the gap-closing process. This applies not only to the usual machine-independent optimization steps but also to instruction reordering of the machine-dependent types, such as discussed in [9–11]. However, in design and evaluation experience with instruction scheduling algorithms for optimizing running time on pipelined machines, we found it advisable to close certain time-critical short gaps before applying a general gap-closing algorithm such as the one described above. This involves a partial gap closing during, rather than after, the determination of the instruction order for the block.

- *Global assignment*

The algorithm described in sections 4 through 8 was implemented reasonably faithfully, including the method for the handling of outer regions outlined in section 10, with partial secondary reservation, and the handling of subscripted references described in section 11.

- *Local assignment*

The function of the local assignment phase is to generate code consistent, insofar as possible, with the results of local allocation and global assignment. This is done one basic block at a time and involves assignment of the registers used solely for intrablock communication (i.e., those allocated during local allocation), as well as certain local optimization steps, such as commuting operands to avoid unnecessary register moves (System/360 load register) and generating RX-type references where possible. The instructions of a block b are processed in order except that a look-ahead procedure is used to avoid unnecessary register moves due to register assignments at the end of b that have already been fixed by the global assignment phase. This procedure uses chains constructed during local allocation, which link each instruction having a register result to the last instruction in b using this result, or to the end of b in case the result is to remain in a register beyond this point. A spill situation can arise during local assignment, even though the available register counts are not allowed to go negative during the prior phases. Experience with the prototype has indicated that this possibility is almost never realized in practice. When it is, the necessary corrective action is straightforward.

- *Limitations of the prototype*

It is appropriate to identify the principal limitations and inadequacies of the prototype:

1. No attempt was made to compile subroutine linkages or in-line functions.
2. The general purpose registers were treated uniformly, so that the impossibility of using register 0 for address modification can be reflected only by not using it at all (by an appropriate parameter setting).
3. The paired register requirements of the fixed-point multiply and divide instructions were ignored, so that these are not correctly compiled.
4. No attempt was made to use the branch-on-index instructions for loop closing, again because of the paired register requirement.
5. Only full-word arithmetic and shift instructions were compiled.
6. All branch targets and scalar data were considered to be addressable with a single base register, which is assumed to be constantly loaded and thus not available to the assignment algorithm.
7. Indexed (i.e., variable target) branches were not compiled.
8. No attempt was made to use the load address instruction for loading or incrementing by a constant, or to use the subtract register instruction for zeroing a register.
9. No attempt was made to handle parameter or equivalenced data.

Table 1 Results of evaluation experiment.

Example programs	Percent reduction of FORTRAN H generated code: Liberal m.d.	Conservative m.d.	Conservative m.i.	Conservative m.d. (all-or-none)	No. of statements	No. of fixed-point ×'s and ÷'s
DTF	18	16	5	8	36	0
CONNECT	30	25	27	20	43	2
MINV	31	24	26	24	51	0
DETERM	31	27	28	24	36	1
MATINV	32	28	27	25	62	5
STRESS	28	24	25	24	77	2
TRNPROB	35	33	32	30	45	4
CLOSP	22	17	17	12	122	6
GEOLAT	11	11	12	3	105	0
averages	26	23	22	19	64	2

10. The storage mapping and region (loop) structure produced by the H compiler are used, including the existing linear ordering of blocks. The predecessor ordering required for the algorithm of section 8 must exist at the source level. Any exception is flagged as an error.

Most of these deficiencies can be remedied by a straightforward addition of detail. However, a few remarks may be of value to a prospective implementer. Consider first the problem of general purpose register (GPR) 0. The simplest solution would be to avoid its use in global assignment and to use it when possible in local assignment. A better approach involves the use of the bit vectors described in section 6 to indicate the feasibility of using GPR 0 for a particular global assignment. For example, if data item x is used in block b as an address modifier (base or index register), then $RRE(b,x)$ should be ORed into $RNA(\text{GPR0},x)$, thus inhibiting the assignment of GPR 0 in the motion of any load based on the availability of x at the end of b.

Now consider the problem of paired, aligned registers. In the case of fixed-point divide and multiply, the alignment, rather than the pairing, presents the major problem for global assignment, because the high-order half of the product is usually discarded and represents only an ephemeral register requirement. Similarly, only one result of the divide is normally required. The alignment can be forced by use of the RNA vectors. For instance, if x is set in b to the result of a fixed-point multiply, then $RRE(b,x)$ is ORed into $RNA(r,x)$ for each even numbered GPR r. The use of the branch-on-index instructions, however, presents a genuine pairing problem, which could be handled by adjusting the register trial sequences of the increment and comparand to increase the likelihood of their being paired and aligned properly. With the exception of a single base register for the most critical data, it seems unwise to reserve any base registers unconditionally, as the FORTRAN H compiler does for addressing the code of a large module. A more flexible alternative is to make, before register assignment, a conservative estimate of code size in each basic block and to use this and control flow structure to assign labels (i.e., branch targets) to as few distinct symbolic address constants as possible. A branch to such a label must have, as an operand, a load of its address constant, which will compete for registers on an equal basis during register assignment.

13. Experimental results

In this section we present some experimental results obtained by comparing the code generated by the IBM System/360 FORTRAN H compiler for nine programs with the output of the prototype described in section 12. The prototype produced on the average about 25 percent better code. In the remainder of this section we discuss in detail the experimental results, which are summarized in Table 1.

Because the FORTRAN H compiler was used to provide input for the register assignment prototype, it served as a convenient standard against which to measure the effectiveness of the prototype. In spite of occasional user complaints about the register assignment produced by the H compiler, it has in fact remained a standard among production optimizing compilers, using a straightforward one-to-one (in the sense of [1]) global assignment strategy after first giving priority to local assignment [3]. Most of these complaints are based on the occasional appearance in the object listing of an adjacent store-load pair referencing the same memory location in the same basic block. With the version of the compiler used for the experiment (level 18, September, 1969), a visual scan of several moderately large listings turned up five such obvious redundancies.

Although the primary object of our register assignment strategy is to minimize the execution time of the object program, we have chosen the storage required for instructions as the criterion for measurement. One reason is that a realistic test of object execution time would have been beyond the scope of our project. In general the one-to-one assignment strategy used by the H compiler is adequate and, indeed, hard to beat on programs whose execution frequency is concentrated in loops that are small enough not to tax the register resources of the machine. There is some indication [12] that such programs are typical, at least of FORTRAN programs. The same study, however, points to ill-advised use of the language and poor algorithm design as a primary source of this phenomenon. In moderate to large programs that have been carefully designed and tuned, especially those of the system type, it is relatively rare that execution frequency is heavily concentrated in a few small kernels. It is for such programs that the major payoff of more advanced global register assignment methods is to be expected. Moreover, payoff in execution time will probably be about the same as in code space. This is because the loads and stores in such programs tend to be typical in execution time, and also because the redistribution of these instructions out of loops is not nearly as significant as their total elimination, i.e., commoning. Thus, for the purposes of this study, code space seemed to be the most appropriate measure of effectiveness.

The H compiler text that we used as input reflected the machine-independent handling of subscripted loads mentioned in section 11. To apply the machine-dependent method described there, our conversion program (optionally) restored the moved loads to their original positions, at least in the more obvious cases. The resulting data in Table 1 are captioned m.d. for machine dependent. The data for the unmodified FORTRAN H text are captioned m.i.

Because of the inability of the prototype to make selective use of GPR 0 and to generate BX-type branches (section 12), we have assumed no use of GPR 0 and no BX branches in the "conservative" data. Even so, because fixed-point multiplies and divides are not necessarily compiled correctly (section 12), the number of such instructions is included in the statistics for each test case. To obtain a reasonable upper bound on effectiveness, we include in Table 1 "liberal" data obtained by compiling each example assuming complete freedom in the use of GPR 0 and assuming that a BX branch is generated when all conditions except register adjacency are met. We believe that the methods suggested in section 12 can lead to results close to the liberal figures.

In an attempt to assess the effect of the individual treatment of loads and stores of a variable as opposed to the classical all or none approach typified by the many-to-few strategy [1,2], the global assignment algorithm of section 4 was modified as follows: Instead of attempting to remove individually each load and store of item x from a region R, an attempt was made to find a single register available for removing all loads and live exits of x from R (i.e., available wherever x was live in R). If such a register exists, all loads and stores of x are removed and the reservation procedure of section 7 is carried out. Otherwise, or in the case of any absolute infeasibility, no loads or stores of x are removed. The results of this modification are described under the heading "all or none."

The examples tested were typical FORTRAN programs. The only strictly fixed-point programs were CLOSP and CONNECT. However, a scan of the output indicates that in only one example, GEOLAT, were there sufficient general purpose registers to accommodate all items on a one-to-one basis. This is somewhat surprising, given the moderate size of the programs. It appears to be due in large part to the temporary variables introduced by the FORTRAN H optimizer. Although most of the examples had to be modified slightly to allow for the limitations of the prototype (see section 12), care was taken not to alter the basic data flow.

The need for partial secondary reservation was questioned in section 10. Although no systematic attempt was made to evaluate this need, experience with the prototype indicated that the code for this function was almost never invoked.

14. Notes on the prototype implementation

The prototype was implemented in PL/I by the author over a two-year period, roughly half of which was devoted to this project. It consists of approximately 4750 lines of code, of which about 3500 lines comprise the global assignment phase. The conversion from FORTRAN H represents an additional 450 lines (also in PL/I). The PL/I F compiler was used initially under System/360 Operating System (OS/360) and finally under Control Program/67-Cambridge Monitor System (CP/67-CMS). While under OS, the object program was overlayed and ran in a 400K- to 500K-byte region.

Although extensive list processing was used, the use of based storage for this purpose was kept to a minimum. Instead the list elements were accessed as elements of arrays of structures. This greatly enhanced the effectiveness of the subscript range condition for error checking. Through a minimal standardization of list formats and naming conventions and the use of the PL/I preprocessor to implement a few common list-handling constructs, a flexible and effective system was evolved for handling singly linked lists. Constructs were introduced for iterative processing of the elements of a list and for prefixing and deleting such elements.

Even allowing for the use of PL/I without optimization and with extensive subscript range checking, the execution times for the prototype were long. About half the time was spent in the global assignment phase. Because this phase was designed with considerably greater care than the rest, this ratio is not particularly significant. The global assignment time was measured in terms of the total number of global references (loads and stores) processed, i.e., considered for global motion, including those generated at entries and exits of a region by the algorithm. The time per reference on a System/360 Model 67 was in the range 0.50–.75 s, and did not vary meaningfully with the number of references. For instance, a program with 172 references and one with 505 references both required 0.66 s per reference. The global assignment time thus appears to be roughly linear in the number of references. In the examples tested, the average number of references per statement ranged from three to six.

15. Summary and conclusions

The results reported in section 13 show that the register assignment strategy outlined in section 2 can result in substantial improvement—on the order of 25 percent—in the quality of object code currently available from FORTRAN source code on System/360 machines. This figure is based on code space, as opposed to execution time. However, a comparable improvement in execution time can be expected, to the extent that execution frequency is not concentrated in small kernels.

Extrapolation of the results to other source languages and other object machines is difficult. On the one hand, the global assignment algorithm is completely general and applies to any machine with multiple registers and to any source language. The algorithm is shown in section 11 to be applicable in the principal situations that are encountered in optimizing code for current source languages, given the basic information needed for all forms of global optimization. In section 12, methods are described for adapting the algorithm to some of the asymmetries of the System/360 register architecture; these methods can be seen to apply to some of the peculiarities of other register machines. However, the viability of the register allocation (as opposed to assignment) concept depends on a high degree of register symmetry. The local assignment and to some extent the local allocation methods used are more dependent on the characteristic System/360 features, such as two-address format and RR vs RX operands. However, both phases are fairly straightforward and easily adapted to other register architectures.

The real key to effective application of any global optimization to more versatile languages, such as PL/I, is the containment of the *possibility* of side effects, aliasing, etc., to the situations in which they are actually used. For example, it is relatively rare that a BASED variable is used as an alias for another variable; yet a compiler must assume it could be an alias for any other variable, in order to generate correct code. Whereas clever global analyses may be able to mollify this assumption, a more profitable course in the long run would be to design a language encouraging, if not forcing, more disciplined access to data. There are, however, other difficult problems of semantics yet to be solved in the design of general purpose, yet highly analyzable (hence optimizable), languages. In the meantime, the fruits of global compiler optimization will continue to be harvested mainly in the domain of less ambitious or more specialized source languages.

The time and space figures in section 14 definitely put the methods of this paper in the category of a high optimization level, suitable only for compilation of debugged production code. The apparent linearity of the time figures, however, leads one to hope that, with simplification and tuning, these methods would be suitable for a production compiler.

Perhaps even more significant than the performance figures, at least to someone whose experience in compiler optimization is limited to the theoretical or algorithmic level, is the implementation experience reported in section 14. The basic global assignment algorithm is conceptually not extraordinarily difficult. However, applying it, even in a modest subset of the environment of a real compiler, took considerable effort. Probably the most important single reason for this difficulty is the great disparity between the relatively amorphous form in which a program is represented during its compilation and the regular structures (such as bit vectors) in which its details must be encoded for efficient implementation of ambitious global optimization techniques. The efficient mediation between these disparate representations requires code and data structures of considerable complexity. It is undoubtedly this mediation, rather than the bit vector manipulations per se, that accounts for most of the overhead of the implementation.

Acknowledgment
The author expresses his gratitude to K. Kennedy of Rice University for his perceptive reading of the manuscript and many helpful suggestions.

References

1. W. H. Day, "Compiler Assignment of Data Items to Registers," *IBM Systems Journal* **9**, No. 4, 281–317 (1970).
2. *The Alpha Automatic Programming System*, A. P. Yershev, Ed., Academic Press, London and New York, 1971.
3. E. S. Lowry, and C. W. Medlock, "Object Code Optimization," *Comm. ACM* **12**, No. 1, 13–22 (January 1969).
4. K. Kennedy, "A Global Flow Analysis Algorithm", *Intern. J. Computer Math.*, Section A, **3**; 5–15 (1971).
5. J. Cocke, "Global Common Subexpression Elimination," *Proc Symp. on Compiler Optimization, SIGPLAN Notices* **5**, No. 7, 20–24 (July 1970).
6. C. P. Earnest, K. G. Balke, and J. Anderson, "Analysis of Graphs by Ordering of Nodes," *J. ACM* **19**, No. 1, 23–42.
7. J. Cocke, and J. T. Schwartz, *Programming Languages and Their Compilers*, Preliminary Notes, Courant Institute of Mathematical Sciences, New York University, 1970.
8. K. Kennedy, "Safety of Code Motion," *Intern. J. Computer Math.*, Section A, **3**, 117–130 (1972).
9. J. C. Beatty, "An Axiomatic Approach to Code Optimization for Expressions," *J. ACM* **19**, No. 4 613–640 (October 1972).
10. R. Sethi and J. D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," *J. ACM* **17**, No. 4, 715–728 (October 1970).
11. J. F. Thorlin, "Code Generation for PIE (Parallel Instruction Execution) Computers," *Proc. AFIPS 1967 SJCC*, 641–643.
12. D. E. Knuth, "An Empirical Study of FORTRAN Programs," *Software Practice and Experience* **1**, 1971, 105–133.

Received February 16, 1973

The author is located at the IBM System Development Division, Poughkeepsie, New York 12602.