A SURVEY OF REGISTER ALLOCATION

Richard K. Johnsson

Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

May, 1973

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-C-0107) and is monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited.

A SURVEY OF REGISTER ALLOCATION

ABSTRACT

One of the most important functions a compiler must perform is the allocation of registers for the instruction stream it generates. In addition to assuring that the allocation of registers is consistent with the semantics of the program being compiled, many compilers attempt to make 'optimal' use of the registers. A survey of work in the area of optimal register allocation is presented, and the algorithms used in a particular optimizing compiler (BLISS-11) are discussed.

A SURVEY OF REGISTER ALLOCATION

INTRODUCTION

One of the important functions a compiler must perform is the allocation of registers for use in the instruction stream it generates. In addition to the requirement that the allocation of registers be consistant with the semantics of the program being compiled, it is usually desirable for the compiler to make "clever" use of the registers. "Clever" may be taken to mean reordering the computation so that a minimum number of registers is used, minimizing the number of data transfers between registers and core, minimizing the number of temporary core locations which must be used, or any other optimization involving the registers.

In generating an instruction sequence to evaluate an expression, a compiler must be prepared to deal with the case that the number of partial results generated exceeds the number of registers available to hold them. It may be possible to avoid this condition by changing the order in which the values of the subexpressions are computed. For example the value of the expression (A)+(B) where (A) and (B) are arbitrary expressions, is independent of whether the value of (A) is computed before the value of (B) or vice versa (ignoring for the moment the possibility that the evaluation of (A) or (B) might have side-effects which change the value of the other expression). On the other hand, the number of registers required for the evaluation sequence (A)(B)+ (Polish postfix) may be different from the number of registers required for the sequence (B)(A)+. Within the limits of the semantics of a given program, a compiler is free to choose an evaluation order for subexpressions which minimizes the number of registers and temporary storage locations required. Note that when there is only one register available the same procedures may be used to minimize the number of temporary values which must be stored.

Independent of the decision to evaluate (A) or (B) first in the above example is the decision of whether the sum will be formed in the register containing (A), the register containing (B), or some third register. This decision is especially important in the case that either (A) or (B) is a common subexpression \ *cse*[*].

In addition to these local register allocation considerations, there are questions of allocating registers to hold the values of *cse*'s. It is also possible to assign variables to registers either for segments of a program in which they are heavily used or for their entire extent. This paper is a survey of work which has been done in solving these problems of register allocation.

HISTORY

Horwitz, et. al. [H66] discussed index register allocation in a paper published in the JACM in 1966. The thrust of this paper is clearly toward FORTRAN-like programs which have simple array accessing mechanisms. An index is presumed to be a simple variable whose value must be retained either in a register or in core at all times.

[*] A common subexpression is an expression whose value is needed at more than one point in a part of a program in which the values of the constituents of the expression do not change, hence its value need be computed only once. The backslash notation will be used in this paper to introduce an abbreviation \ abr.

Given the future index requirements of a program, the allocation of the index registers of the machine to the indices is considered. When all of the index registers contain values that will be needed again later in the program, a decision must be made to replace one of those values when a new index is required.

Horwitz considers the possibility that an index may be changed while it resides in a register. If an index is changed in a register, and subsequently that register must be allocated to another index, the changed value must be stored in core. If the value is not changed, it is not necessary to store the value back into core when the register is needed. This problem is analogous to the problem of page replacement in a virtual memory system. It is less expensive to replace a page which has not been changed since it was read from secondary storage because a valid copy still exists elsewhere.

For the purpose of this problem, a program can be considered to be a sequence of steps each of which requires a specific index. The fact that there may be steps in the program that do not require indices is not important. Consider the set of program steps and associated indices

step	index
1	x_1*
2	x_2*
3	x_3
4	x_1
5	x_2
6	x_2*
7	x_4

where x* means that index x is changed in the step where x* appears. When a step calls for an index, that index must be in one of the index registers. The other index registers may contain any configuration of indices. The indices in the other index registers may or may not be in a modified state.

We may construct all of the allowable configurations for each step i, i.e. all combinations of n of the indices used by the program which include the index required by step i (where n is the number of index registers available). Consider the configurations to be nodes in a directed graph with branches from each configuration of the ith step to each configuration of the i+1st step. Each of these branches can be assigned a weight which is the cost of making the change in configurations between steps i and $i+1$ represented by the branch. The cost of changing between configurations is defined as the number of memory references required to make the change. Thus each new index which is loaded has a cost of one, each starred index which is replaced has an additional cost of one. Changing an unstarred occurance of an index to a starred occurance of the same index, or replacing an unstarred index require no memory references and therefore have a cost of zero.

Given this representation of the possible allocation of index registers, the problem becomes one of finding the shortest, i.e. least expensive, path through the graph from the first step to the last step. Although there are several algorithms for finding the shortest path through a directed graph, the number of calculations required for other than a small number of nodes makes these solutions impractical. Since it is necessary to find only one of the possibly many shortest paths through the graph, we may restrict attention to a subgraph which contains a shortest path. Horwitz et. al. devote the bulk of their paper to developing properties of these graphs which lead to rules for eliminating nodes and branches from consideration. Horwitz proves that the subgraph obtained by applying these rules does contain a shortest path, and gives a procedure for finding that path. Horwitz gives six rules for generating the subgraph from which an optimal index register allocation may be derived. In summary they are

Rule 1: Generate only minimal change branches and eliminate any node which has no minimal change branches entering it. A minimal change branch is defined as a branch from node n at step i to node n' at step i+1 such that either nodes n and n' are identical or n' differs from n only in the index required at step i+1.

Rule 2: If n_1 and n_2 are nodes of step i and $W(n_1)+w(n_1,n_2)\leq W(n_2)$, eliminate n_2. Here $w(n_1,n_2)$ is the cost of changing from the configuration of n_1 to that of n_2. $W(n')$ is the weight of node n' obtained by considering the nodes of the previous step and the branches entering n'. $W(n') = \min_n(W(n)+w(b))$ where $w(b)$ is the cost associated with the branch from n to n'. The initial node has weight zero.

Rule 3: n_1 and n_2 are nodes at step i which differ in exactly one element. Let z_1 be the element of n_1 which is replaced by z_2 in n_2. Although the exact explanation is somewhat more complex, the idea is that node n_2 can be eliminated when $W(n_1)\leq W(n_2)$ and in the future z_1 will be used before z_2. This requires the ability to look ahead in the program.

Rule 4: This rule is a consequence of Rule 3 and prevents generation of nodes that would later be eliminated by Rule 3. If z_1 and z_2 are elements of a node n at step i and the next use of z_1 comes before the next use of z_2, do not form a node at step i+1 which replaces z_1 by the index required at step i+1.

Rule 5: Since we need only one shortest path, generate only one branch b into each node n' such that $W(n')=W(n)+w(b)$.

Rule 6: If a node n of step i which is not the last step has no branches leaving it, eliminate node n.

Figure 1 shows the result of applying Rule 1 to the graph of the example program above when there are two index registers available. Step 0 is added to indicate the initial configuration which contains two indices not used in the program (x5 and x6). Each branch is labeled with the cost of the change between the indicated configurations and each configuration is labeled with the minimum cost to reach the configuration from step 0.

F. Luccio, writing in the CACM [L67], showed that Horwitz's rules may restrict the graph so that at some steps only one configuration is possible. The program steps before and after such a step may be treated separately. Luccio neatly describes his technique in terms of link diagrams. Six types of links are used to connect various combinations of starred and unstarred indices (figure 2). Links of types 1, 2, 3, and 4

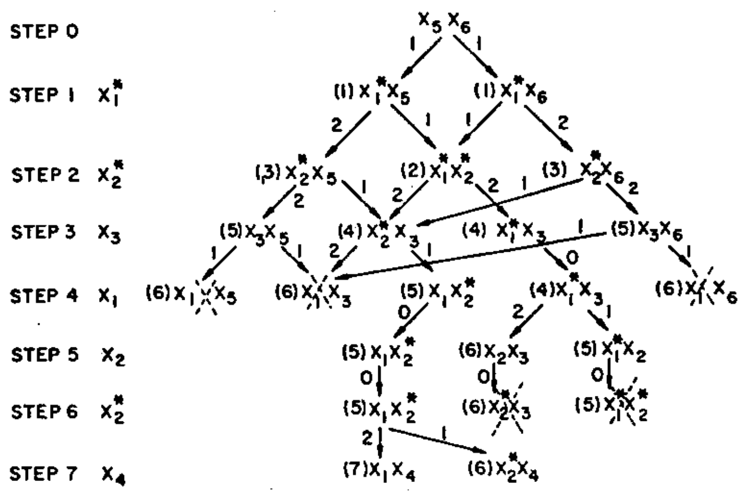

Figure 1

are built whenever a second occurance of an index is seen. Links of types 5 and 6 are built following occurances of starred indices and are maintained up to the current step. These are called temporary links since they will be changed to one of the other types when a succeeding occurance of the particular index is encountered.

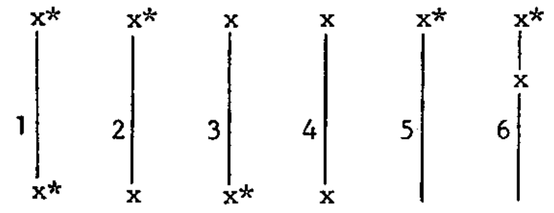

Figure 2

A link is said to cover all the steps along its extension excluding the extremes. Only the first extreme is excluded for temporary links so that they cover the current step. Luccio gives two rules for changing links of types 1, 2, 3, or 4 to links of definite allocation (type 0). The index corresponding to a link of type 0 must be kept in its register throughout the entire extension of the link.

If there are N registers available then

1) a link *l* of type 1 becomes type 0 if for each step *k* covered by *l* the number of other links of types 0, 1, 2, 3, or 4 covering *k* is less than N-1.

2) A link *l* of type 2, 3, or 4 becomes type 0 if for each step *k* covered by *l* the total number of other links covering *k* is less than N-1.

When the number of type 0 links covering a step *k* is N-1, the configuration for *k* is fixed. The *n* registers must contain the N-1 indices corresponding to the type 0 links and the index required by step *k*. At such steps the Horwitz method may be applied independently to the preceeding and succeeding steps.

The Horwitz method is related to Belady's algorithm for page replacement in a virtual storage computer [B66]. Belady showed that in a paging environment, the page to be replaced should be the page whose next use is farthest in the future. In addition he noted that if a page has not been written into it need not be written out (to secondary storage) but merely deleted. The ability to determine which page (register) is next used farthest in the future depends on knowing the future behavior of a program.

Ikuo Nakata addressed the question of evaluation order in his paper describing the register allocation part of a FORTRAN compiler for the HITAC-5020 [N67]. Nakata shows that the order of evaluation of the subexpressions of an expression can affect the number of temporary values that are required at any one time. Consider the expression a*b+(c+d)/(e+f). A straight forward code sequence to evaluate this expression is:

$a*b \rightarrow R_1$
$c+d \rightarrow R_2$
$e+f \rightarrow R_3$
$R_2/R_3 \rightarrow R_2$
$R_1+R_2 \rightarrow R_1$

Suppose, however, that this expression must be evaluated on a computer with fewer than three registers. To use the same evaluation order with only two registers available would require one of the intermediate results (namely a*b) to be stored in some temporary memory location. On the other hand, by changing the order of evaluation of the subexpressions, the expression may be evaluated using only two registers and without storing intermediate results.

$c+d \rightarrow R_1$
$e+f \rightarrow R_2$
$R_1/R_2 \rightarrow R_1$
$a*b \rightarrow R_2$
$R_2+R_1 \rightarrow R_2$

The central point of this example is that the subexpression (c+d)/(e+f) requires two intermediate values. Since those intermediate results are not needed after the division is performed, one of the registers may be used to compute a*b. Since the result of the evaluation of an expression occupies only one register, it follows that for any binary operator, the operand whose evaluation requires the larger number of registers should be evaluated first.

The number of registers required to evaluate the expression (a) <binop> (b) where (a) and (b) are arbitrary expressions and <binop> is some binary operator is given by the following analysis. Let l and m be the number of registers required to evaluate (a) and (b) respectively. If either (a) or (b) contains no operators (it is a constant or a simple variable) it requires zero registers. (Note that require here means the minimum number of registers necessary to evaluate an expression without storing any intermediate results).

There are two cases:

1) $l=m=p$. (a) can be evaluated first leaving the result in one of the l registers used. Evaluation of (b) will require one more than the l-1 registers remaining giving a total of p+1 registers for the expression.

2) $l \neq m$; $\max(l,m)=p$. In this case the operand requiring the larger number of registers is evaluated first leaving $p-1$ registers for the other operand. Since the other operand requires at most $p-1$ registers no additional registers are needed and the expression can be evaluated using only p registers.

In both cases p is a lower bound on the number of registers required and $p+1$ is an upper bound. In case 2, p is also an upper bound.

Nakata gives an algorithm for labeling the nodes of a tree with the number of registers required for evaluation of the node. Briefly, this algorithm assigns a label Ln to each node n of the tree such that if n is a leaf then $Ln=0$, otherwise the immediate descendents of n have labels l and r and $Ln=\min(\max(l+1, r), \max(l, r+1))$.

Nakata's algorithm for code production involves first labeling the nodes of the tree by the above method, and then beginning at the root node, walking through the tree generating code to evaluate the expression represented. At each node the operand requiring the larger number of registers is evaluated first. If the operands require the same number of registers, the left operand is evaluated first. Nakata does not consider formally the question of what to do when the number of simultaneous temporary values exceeds the number of registers. He does, however, offer some heuristics for deciding which temporary value should be stored. On most machines the left operand of a division or subtraction operation must be in a register, so the left operand of these operations should not be stored. This may conflict with the other assertion that the value to be stored should be the one whose use is farthest in the future, but Nakata conjectures that the efficiency of the code produced will not be significantly affected by the choice of either of these courses of action.

Using a graph theoretic approach, R. R. Redziejowski later proved that Nakata's algorithm does use the minimum number of registers [R69]. Redziejowski transformed

Nakata's tree into a "lineup" or linear sequence of verticies. Each vertex represents a single operation in the tree and an arc is drawn from vertex x to vertex y to represent a partial result which is computed at y and used at x. Choosing a feasible evaluation order is equivalent to ordering the sequence of verticies so that vertex x preceeds vertex y if there is an arc from y to x. (This is equivalent to requiring that any partial result be computed before it is used).

At any vertex x the number of partial results created before x and used after x is represented by the number of arcs passing over vertex x. Redziejowski calls this number the width of the lineup and develops an algorithm for producing a lineup of minimum width. Redziejowski's algorithm is in principle the same as Nakata'a algorithm and therefore Redziejowski's proof of his algorithm can be considered as a formal proof of Nakata's algorithm. Redziejowski generalizes the algorithm to include operators with more than two operands.

Sethi and Ullman [S70] consider the more general problem of minimizing the number of program steps and/or the number of storage references in the evaluation of an expression with a fixed number of general registers. They exploit the associative and commutative properties of operators and assume that all elements are distinct (no *cse*'s) and that there are no non-trivial relations between operators (e.g. no distributive law).

Nakata's tree labeling scheme is modified slightly to account for commutative and non-commutative operators. This change assigns a label of one rather than zero to a leaf node which is the left descendent of its ancestor. The change means that the left and right operands of a binary operator may have different weights and accounts for the gains which may be made by exploiting commutativity.

First considering only non-commutative operators, Sethi and Ullman prove that their Algorithm 1 (which is essentially Nakata's algorithm) uses the minimum number of registers as well as the minimum number of loads and stores. Since the number of binary operators is not changed by the allowed transformations, a program which has a minimum number of loads and stores has a minimum number of program steps.

In Algorithm 2, Sethi and Ullman consider commutative operators by adding a step to Algorithm 1 which interchanges the left and right descendents of a commutative operator when the left descendent is a leaf and the right descendent is a non-leaf.

Associativity is treated only in conjunction with commutativity since in practice most associative operators are also commutative. The approach used by Sethi and Ullman is to make the associative-commutative operators into n-ary operators, reorder the operands so that the one or two operands requiring the largest number of registers appear on the left, and then change back to binary operators associating to the left. This is conceptually similar to Redziejowski's treatment of n-ary operators.

Sethi and Ullman prove that each of their algorithms generates an evaluation sequence containing the minimum number of loads and stores under the assumptions of the algorithm. They then show that this leads to the conclusion that the algorithms also minimize the number of storage references.

In their conclusion, Sethi and Ullman point out that all of their algorithms can be performed in time proportional to the number of nodes in the tree. They also show that the algorithms can easily be modified to allow operations which require more than one register.

Beatty [B72] recasts the ideas of Sethi and Ullman in terms of axiom systems.

Beatty extends the Sethi-Ullman algorithm for associative-commutative operators to include the unary minus and its relations to the other operators. These relations include the equalities

$$a-b = a+(-b)$$
$$-(a*b) = (-a)*b$$
$$-(a/b) = (-a)/b = a/(-b)$$

Beatty's proof of minimality is considerably more complicated than the Sethi-Ullman proof due to the properties of the unary minus.

The work thus far discussed has dealt only with the question of optimal use of registers in expressions. A paper by W. H. E. Day in the IBM Systems Journal [D70] considers the much broader problem of global assignment of data items to registers. Before describing Day's work, it is necessary to explain the distinction between what Day calls global assignment and what he considers local assignment.

Consider a programming language L with statements that are ordered sequences of delimiters, operators, constants, and identifiers. The constants and identifiers form a subset of L whose elements represent data items. Statements in L are either descriptive or executable, the latter specifying operations to be performed on data items. A data item is said to be 'defined' in a statement when execution of the statement causes a new value to be assigned to the data item. A data item is 'referred to' when the value of the data item is required for correct statement execution.

Day defines a program P in L to be a finite ordered set of statements in L. A basic block in P is an ordered subset of elements of P which intuitively is "straight line code," i.e. a sequence of statements which can only be entered by branching to the first statement and which can only be left by branching from the last statement. P_b is a representation of P as an ordered set of basic blocks. P_g is a representation of P as a directed graph with the elements of P_b as the verticies and a set of arcs representing the flow of control among the basic blocks of P. A region R_i is a strongly connected subgraph of P_g, and P_r is a representation of an ordered set of regions:

$$\begin{aligned} P_r &= \{R_1, R_2, \ldots, R_n\} \\ &= \{R_i \mid R_i \neq R_j \text{ for } i \neq j, \\ &\qquad R_i \cap R_j = 0 \text{ or } R_i \subset R_j \text{ for } i < j, \\ &\qquad R_n = P_g\ \} \end{aligned}$$

A computer has a set of registers G^* whose elements are g_i, and for most situations requiring the use of a register any available $g_i \in G^*$ may be assigned. Let d represent an element of P, P_b, or P_r and define:

$G' = \{\ g_i \mid g_i \in G^*, g_i$ is available for assignment everywhere in $d\ \}$

$N' = \{\ n_i \mid n_i$ is a data item in P, n_i may be assigned to registers in $d\ \}$

Given these representations, Day offers the following definitions:

1) A local assignment is a (possibly multi-valued) mapping of $N \supseteq N'$ onto $G \supseteq G'$ for $d \in P_b$.

2) A global assignment is a (possibly multi-valued) mapping of $N \supseteq N'$ onto $G \supseteq G'$ for $d \in P_r$.

3) A one-one assignment is a one-one mapping of $N \supseteq N'$ onto $G \supseteq G'$. A one-one assignment defines a one-to-one correspondence between N and G.

4) A many-few assignment is a single-valued mapping of $N \supseteq N'$ onto $G \supseteq G'$ with $C(N) \geq C(G)$. ($C(X)$ represents the cardinality of the set X).

5) A many-one assignment is a many-few assignment in which $C(G) = 1$.

A data item is 'active' at a point in d if it may be referred to before being defined subsequent to the point. Two data items interfere in d if they are both active at some point in d. A necessary condition for the assignment of $N \supseteq N'$ to $g \in G$ in d is that n_i must not interfere with n_j for every $n_i, n_j \in N$, $i \neq j$.

Local assignment, as defined by Day, occurs entirely within basic blocks of a program. The methods described by Horwitz et. al., Nakata, Sethi-Ullman, and Beatty provide algorithms which may be used to obtain optimal local assignments under the assumptions dictated by those authors. Local assignment is not, however, able to cope with data items which may be active on block entry or exit.

Global one-one assignment partially solves the problem of active data items at block boundaries by assigning data items to registers throughout an entire region. With this type of assignment, precautions need be taken only at region boundaries to assure that values of active data items are retained.

Assigning a data item to a register for an entire region may lead to inefficient use of the registers. With accurate program flow information, it is possible to determine the points at which a data item is active. When the active points of all data items are known, a set of data items which do not interfere may be determined and the elements of that set assigned to the same register. The avalability of complete and accurate flow information is critical to efficient of global many-one or many-few assignments. In the absence of flow information, many-one and many-few assignments degenerate to one-one assignments.

Day formulates global one-one, many-one, and many-few assignment problems as integer programming problems. He makes the reasonable assumption there is some profit (>0) associated with the assignment of a data item to a register and that this profit depends on the frequency and context of the use of the data item. Day gives several algorithms for solving the assignment problems. Some of these give optimal results while others may produce non-optimal feasible results at a much lower cost in computational complexity. Day's formulations of the problems are summarized below.

The global one-one assignment is the simplest of the three problems since no interference data is required. Refering to the definition of a one-one assignment let $n = C(N')$ and $m = C(G')$ and let p be a vector of profits such that p_i is the profit associated with assigning $n_i \in N'$ to a register. Vector x is a selection vector such that $x_i = 1$ if $n_i \in N'$ is assigned to a register, otherwise $x_i = 0$. The problem is

maximize	$z = px$
subject to	$Ix \leq m$ [Ix is the sum of elements of x]
where	$x_i \in \{0, 1\}$ and $p_i > 0$

The solution to the one-one assignment is simple: assign the m data items with the largest profits to registers.

The global many-one assignment problem is similar to the one-one problem except for the added restriction that no two data items which are assigned to the register may interfere. Day expresses this condition in terms of a matrix of data item interference values ($C \mid c_{i,j} = 1$ if $n_i, n_j \in N'$, $i \neq j$ interfere; $c_{i,j} = 0$ otherwise).

The many-few assignment problem is an extension of the many-one assignment problem to more than one register. The problem is to select the best combination of many-one assignments. Day explicitly excludes multi-valued mapping which might assign a single data item to different registers at different points in a region.

In his conclusion, Day reports the results of several tests of the actual execution characteristics of his algorithms for many-few assignment. The OPTSOL alorithm (which provides an optimal solution) requires much longer execution time for relatively gain over the estimating algorithms. (Sample values: for one register and 48 data items t(optimal) = 6 sec., t(estimate) = 0.06 sec.). The total profits produced by the extimating algorithms are consistently greater than 90% of the profit produced by the OPTSOL algorithm and are significantly better than a one-one solution to the same problem. Day concludes that his algorithms are sufficiently fast to be included in an optimizing compiler.

ACUTAL COMPILER ALGORITHMS

Lowery and Medlock give an example of a global one-one assignment in their paper on the optimization in the IBM FORTRAN-H compiler [L69]. FORTRAN-H uses a simple forward-backward scan of the code sequence for loops to find the first and last uses of registers which are needed for intermediate values. This type of register allocation for straight line code is described by Hopgood [H69]. After local assignment, the most important (i.e. most frequently used) variables in the loop are assigned (one-one) to the remaining registers. Some care is taken to be sure that a sufficient number of registers is available for use in inner loops. In some cases values which have been assigned to registers in outer loops are stored-loaded at inner loop boundaries to make an additional register available. Lowery and Medlock do not go into detail in the area of register allocation prefering to treat the more glamorous optimizations of FORTRAN-H.

Wulf and Johnsson [W73] describe a method of temporary name binding (register allocation) which is a global many-few assignment with some heuristic embellishments. The DEC PDP-11 (target machine for the described BLISS-11 compiler) has the property that registers are not required for any operation. While the hardware requires registers for some functions, e.g. indexing, those functions can be simulated in software without using registers at the expense of program size and speed. The compiler takes advantage of this property by assigning a measure of importance to each quantity which might be assigned to a register and then choosing as many of the "most important" items as will fit in the available registers.

BLISS-11 divides a program into a set of regions which correspond to the (sub)routines of the program. Each routine is responsible for saving and restoring the contents of any registers it uses. At the time of register allocation, the compiler has global flow analysis information which identifies the *cse*'s. Earlier phases of the compiler have also decided the evaluation order of the expressions based on a variation of Nakata's algorithm. The temporary name binding proceeds in three phases - temporary name \ *TN* assignment, ranking, and packing.

The assignment phase consists of a left-to-right-depth-first tree walk during which temporary names are generated for the intermediate values required by the computation. The declaration processing phase of the compiler generates temporary names for the variables which are local to a routine. There are two types of local variables, those which must be assigned to a register (due to programmer specification) and those which may be assigned to either a register or a core location at the compiler's discretion. During this treewalk the compiler also assigns to each node in the tree a unique pair of numbers indicating the position of the node in the tree relative to linear order (linear order number \ ***lon***) and to flow order (flow order number \ ***fon***). The ***lon*** is a monotonicaly increasing value along a linear traversal of the program (essentially the position of the node in the source program). The ***fon*** increases along possible flow paths and is reset at the beginning of each of any group of parallel flow paths. The diagrams in figure 3 should illustrate more clearly the way in which the ***lon*** and ***fon*** are assigned.

As the compiler walks through the tree and assigns ***lon-fon*** values and temporary names, it records for each *TN* the following six values:

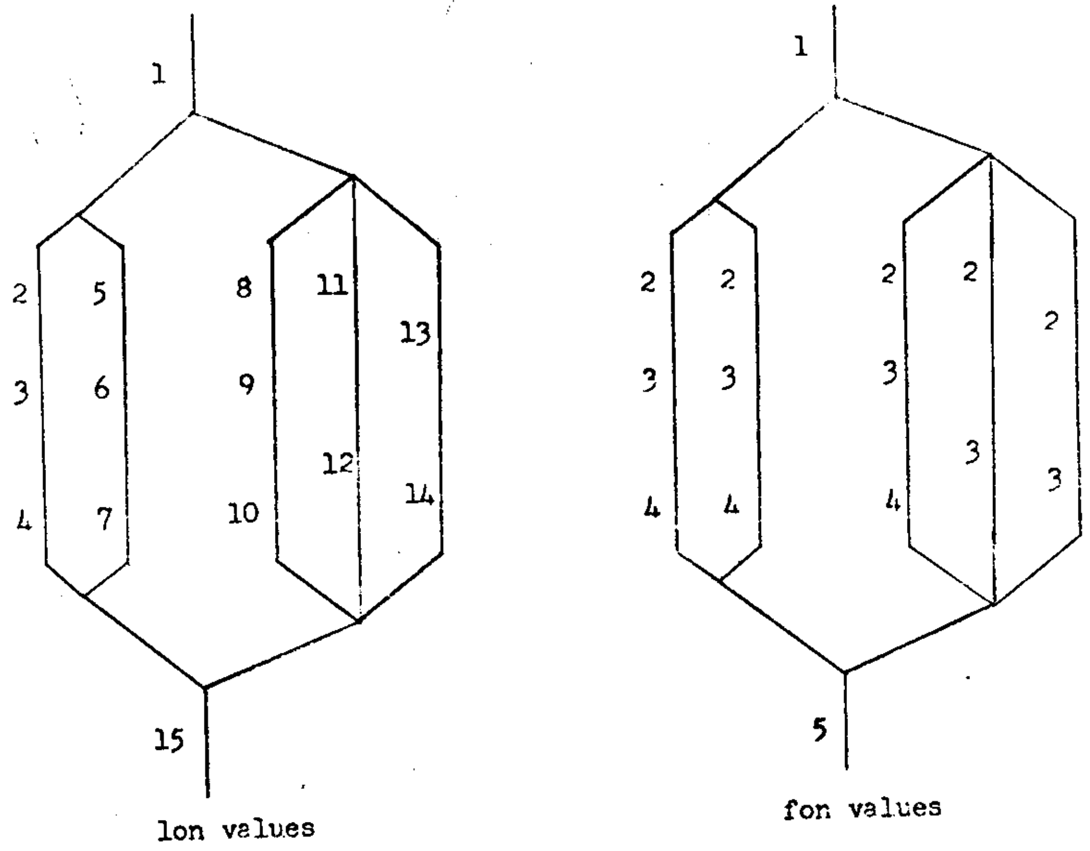

Figure 3

1) ***lon*** of first use
2) ***lon*** of last use
3) ***fon*** of first use
4) ***fon*** of last use
5) minimum use complexity
6) maximum use complexity

The first four values are the smallest (largest) ***lon*** (***fon***) values at which the ***TN*** is in use. Items 5 and 6 represent the estimated total cost (in terms of storage references) if the ***TN*** is assigned to a register (5) or to a core location (6). The complexity of use at a specific node is a function of the addressing mode, the operator involved, and the depth of loop nesting. The difference between items 5 and 6 is a particular instance of the profit described by Day.

After all *TN*'s are assigned and the ***lon-fon***-complexity data computed, the *TN*'s are ranked according to their "importance." Importance is defined to be proportional to the maximum use complexity and inversely proportional to the ***fon*** span (last use minus first use). Since the maximum use complexity is directly related to the number of times a *TN* is used, the importance measure is related to the use density of the *TN* throughout the region (routine) with additional consideration being given to *TN*'s used in loops, for indexing, or in other complex situations.

The four ***lon-fon*** values associated with each *TN* define a (possibly degenerate) rectangle in the ***lon-fon*** space. This rectangle is called the "lifetime" of the *TN*. The third phase of temporary name binding binds the *TN*'s to the available registers and core locations. Necessary conditions for binding a *TN* t to a location l are:

1) l is available. (i.e. if l is a register then the contents of l will be saved at routine entry; if l is a core location then l has been "allocated" by the routine.)

2) The lifetime of l does not overlap the lifetime of any other *TN* already bound to l.

TN's are bound in the following order:

1) *TN*'s which must be bound to a specific register. (e.g. the value of a routine call is a *TN* which must be bound to a specific register)

2) *TN*'s which must be bound to a register, but the compiler is free to choose any available register.

3) all other *TN*'s in order of their importance.

If *TN*'s in group 2 cannot fit into an available register, a new register is "opened" (marked so that its contents will be saved at routine entry). Because *TN*'s are placed in groups 2 and 3 only through programmer specification (either explicit or implicit), the condition of not having a register to satisfy a request in group 1 or 2 is considered semantic error. Since the structure of the PDP-11 does not require registers, the *TN*'s in group 3 need not be assigned to registers so that the problem of not having enough registers is not encountered.

The algorithm for binding each *TN* in group 3 is:

1) try an available register.

2) if the cost of opening another register is greater than the difference in the maximum and minimum use complexities (if cost is greater than profit) then try an available core location.

3) try to open another register.

4) try an available core location (if not tried in 2).

5) allocate another core location.

The algorithm terminates with the first successful step.

There are two additional actions taken during the assignment which increase the probability of "clever" assignments. These are "targeting" and "preferencing." Targeting tries to use the same *TN* for several intermediate steps in evaluating an expression. Target temporary names are passed from ancestors to descendents in the treewalk. The descendent tries to use the *TN* it is passed if possible. Except in the case that the descendent is a control expression or a *cse*, the target can usually be met. Obviously only one of the operands of a node can receive the target. The decision about which one is the "target path" is made by an earlier phase of the compiler. It is important to note that the target path decision and the evaluation order decision are independent.

When a descendent cannot honor the request for a target, it returns to its ancestor the temporary name that was actually assigned. The ancestor then expresses the preference that the two *TN*'s be bound to the same location. The binding algorithm contains a preliminary step which tries to honor the preferences. The assignment phase must always allow for the worst case in targeting and assigning *TN*'s. The preferencing operation tends to promote "clever" assignment of registers in the simple cases. Preferencing also produces some interesting assignments in the cases of the last use of a local variable or a *cse*. The *TN* for the expression which contains the last use of a *cse* is frequently bound by preference to the same location as the *cse*.

UNSOLVED PROBLEMS

The problem of optimal register allocation for expressions has been studied extensively in the papers by Horwitz, Nakata, Sethi-Ullman, Beatty, etc. A common feature of these papers is they do not consider the implications of common subexpressions on the problem of register allocation. Clearly the overall performance of a program can be improved if the value of a *cse* is computed only once. If the value is to be computed only once, then that value must be kept somewhere. The logical place is in the register in which the value was originally computed. The algorithms described for optimal register allocation do not necessarily produce optimal results in the presence of *cse*'s. Apperson has proposed a study of the problem of register allocation when *cse*'s are considered [A73].

The algorithms used in the BLISS-11 compiler attempt to solve the much more global problem of register allocation in large regions of a program. These algorithms are largely based on heuristics and would likely benefit from an attempt to formalize the problems. Day provides some insight into how to obtain an optimal solution to the global register assignment problem, but he does not discuss the problem of gathering the information which his algorithms require.

A problem not addressed by either Day or BLISS-11 is the assigning of a data item to different registers at different points in a program. Using the BLISS-11 terminology, it would be interesting to assign a new temporary name for local variables each time a new value is assigned to the variable. Each variable would then have several "lifetimes" and these lifetimes could be considered separately when the actual assignments are made. The real difficulty in implementing this type of scheme is the difficulty of determining the separate lifetimes. There is also a difficulty in characterizing the lifetime of a temporary name. The ***lon*** and ***fon*** measures attempt to solve this problem, but they cannot adequately describe conditions of first or last uses of a temporary name on parallel paths in a program.

It is an interesting commentary on the field that while every compiler must solve some form of the register allocation problem, little discussion of the solutions found is available in the literature. This suggests that although there are some formal proofs of optimal algorithms, actual compiler implementations tend to resort to heuristics and special case analysis. The general situation for real compilers might be summed up by the treatment of register allocation by Cocke and Schwartz [C70]. Their one page discussion of register allocation can be paraphrased: "If you have a good register allocation scheme you can produce better code."

BIBLIOGRAPHY

[A73] Apperson, Jerry L., "Proposal for a Thesis on Optimal Evaluation Order for Expressions with Redundant Subexpressions," Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA.

[B72] Beatty, James C., "An Axiomatic Approach to Code Optimization for Expressions," **JACM 19**,4 (October 1972), 613-640.

[B66] Belady, L. A., "A Study of Replacement Agorithms for a Virtual Storage Computer," **IBM Systems Journal** 5,2 (1966), 78-101.

[C70] Cocke, John and Schwartz, J. T., **Programming Languages and their Compilers**, Courant Institute of Mathematical Sciences, New York University, New York, 1970.

[D70] Day, W. H. E., "Compiler Assignment of Data Items to Registers," **IBM Systems Journal** 9,4 (1970), 281-317.

[H69] Hopgood, F. R. A., **Compiling Techniques**, American Elsevier, New York, 1969, 91-103.

[H66] Horwitz, Karp, Miller, and Winograd, "Index Register Allocation," **JACM 13**,1 (January 1966), 43-61.

[L69] Lowery, E. S. and Medlock, C. W., "Object Code Optimization," **CACM 12**,1 (January 1969), 13-22.

[L67] Luccio, F., "A Comment of Index Register Allocation," **CACM 10**,,9 (September 1967), 572.

[N67] Nakata, Ikuo, "On Compiling Algorithms for Arithmetic Expressions," **CACM 10**,8 (August 1967), 494-92.

[R69] Redziejowski, R. R., "On Arimetic Expressions and Trees," **CACM 12**,2 (February 1969), 81-84.

[S70] Sethi, R. and Ullman, J. D., "The Generation of Optimal Code for Arithmetic Expressions," **JACM 17**,4 (October 1970), 715-728.

[W73] Wulf, W. A. and Johnsson, R. K., "The TNBIND Module," in Wulf, Apperson, Geschke, Johnsson, Weinstock, and Wile, **The Design of an Optimizing Compiler**, to be published.