INFORMATION SCIENCES **18**, 83–94 (1979)

Register Assignment in Tree-Structured Programs

WILLIAM W. AGRESTI

School of Engineering, University of Michigan—Dearborn, 4901 Evergreen Road, Dearborn, Michigan 48128

Communicated by John M. Richardson

ABSTRACT

Much complex decision-making is performed routinely by the software of a computer system. It is appropriate to study more thoroughly the performance of this built-in decision-making, because it can strongly influence the efficiency of the entire system. One objective of compilers is to produce a reasonably efficient machine-language version of a user's program. Traditionally, one of the best opportunities for improving the compiler-produced machine-language program has been in devising efficient policies for assigning quantities to the computer's registers. The programs of interest here involve flow of control which can be represented by a tree structure. The problem of assigning index registers in such programs is formulated as a (nonserial) dynamic-programming problem. Following the resulting recursion equations leads to a policy which the compiler could follow to minimize costs. The policy decisions specify those steps in the program where particular quantities should be loaded or stored into registers. An example involving a branching program is solved by this method.

INTRODUCTION

A host of resource allocation problems are found in the operation of computer systems. Much of the decision-making in such environments is automated as part of the system software. Operating systems, for example, regularly manage the resources of the computer—conserving memory, exploiting parallelism wherever possible, and scheduling the processors for high utilization. Compilers, as well, contain a wealth of decision-making apparatus which is routinely exercised as part of the process of translating user programs into machine language. Because of the high usage of such software, the quality of this built-in decision-making can strongly influence the efficiency of the entire system.

Underlying the present study is the belief that some of the complex decision-making that is automated in computer software can benefit from more thorough analysis. This observation is certainly not new. Perhaps the

 0020-0255/79/04083-12$01.75

most outstanding example is the application of queueing theory in computer operating systems. The software of interest here is the compiler or language processor. An important objective in compiling is to consider the efficiency of the machine-language program that is generated. As a user's program in, say, FORTRAN is being translated, decisions are made regarding the sequence of machine-language instructions which should be produced to correspond to each FORTRAN statement. During this compiling process, transformations are applied which change parts of the program into equivalent versions which will execute faster or consume less memory. It is suggested that this process of successive transformation of the program is a staged decision process whose behavior might be clarified by a dynamic-programming formulation.

The availability of these so-called "program optimizations" is extremely important—especially to the construction of large application and systems programs which will be run on a regular basis. Their presence in compilers makes it feasible for programmers to use high-level languages and still get a relatively efficient program without having to resort to assembly language. Structured programming, which puts a premium on clarity and organization, can sometimes lead to inefficient programs. With the program optimizations automated, structured programming can be safely pursued and its benefits realized.

From the earliest compilers until the present, register assignment has offered one of the best opportunities for program optimization. The general problem is to describe rules by which a compiler can make the best possible use of a computer's available registers. When several program statements are scanned, the compiler can note what values are required in each of the calculations. This pattern of usage suggests a plan for keeping certain values in high-speed registers over a span of several statements. In this way, the program will run faster because the required quantities will not have to be fetched from memory. Of course, the number of registers is limited; so the problem becomes one of devising a schedule for loading these values into registers—ideally, a schedule which permits the fastest possible execution of the program. The problem has been widely studied in several versions, depending on such factors as the range of the allocation (over a few statements or an entire program), the existence of common subexpressions, and practical considerations of implementation on a particular machine.

Registers which are used for indexing are the special concern here. Indexing is a valuable programming technique for operating on data which are arranged in storage in some systematic way. It can help reduce the running time of the machine-language programs produced by compilers. The problem of assigning index registers has been studied by Horwitz et al. [1] and Kennedy [2]. In [1], a procedure was given for specifying which quantities should occupy index registers at each point in a program so that the number of memory references

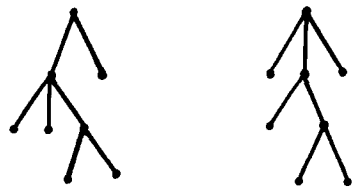

Fig. 1. Examples of tree structures.

(from LOAD and STORE operations) is minimized. The programs in [1] involved only a straight-line sequence of statements, i.e., no branches or loops. More recent work in [2] improved the procedure and suggested extensions to programs containing simple loops.

The major departure here is to consider programs which possess *non*linear flow of control—the branches which are found in real computer programs. In FORTRAN terms, we can now include some programs which contain conditional and branching statements like IF and GO TO. The programs we study are those whose flow of control can be represented as a tree structure. Figure 1 gives some examples. To solve the index register allocation problem in such programs, a new methodology, dynamic programming, is introduced.

1. THE INDEX-REGISTER ASSIGNMENT PROBLEM

We are interested in those computers which provide some number of registers which can be used for indexing. Two types of instructions are of interest. One type simply *refers* to the contents of an index register. For example,

```
ADD 1,K(2)
```

would mean that two quantities are to be added together. One operand is in register 1. The second operand is in memory at the address given by K plus the contents of index register 2. The second type of instruction *modifies* the contents of the index register. For example, if we added one to the contents of index register 2, then the instruction would be of the second type. Where there are more indices than registers, the problem is to specify what indices should occupy index registers at each step in the program.

Because our only concern is with the references to indices in a program, we use "program" to mean a sequence of such references. Where the index has been modified, an asterisk is placed next to it. A program, assuming one reference per step, might look like the following:

$$x_1 x_3^* x_2 x_4 x_2^* x_3 x_1 x_2$$

This program tells us that index x_1 is referenced at step 1, index x_3 is referenced and modified at step 2, and so on. If we denote by $P(i)$ the index referenced at the ith step, then $P(1)=x_1$, $P(2)=x_3^*,\ldots,P(8)=x_2$.

We assume that the machine has N_R index registers, and we define a register *configuration* Q_i to be an unordered set of N_R indices which occupy the index registers at step i in the program. An allocation A for a n-step program P is a sequence of configurations,

$$A=(Q_1,Q_2,\ldots,Q_n).$$

Every legal allocation for P requires that the index called for at step i in the program must be in a register at that step. In our notation,

$$P(i)\in Q_i, \qquad 1\leqslant i\leqslant n.$$

The problem then is this: when there are more indices than index registers, what procedure will provide an allocation which satisfies the condition above and minimizes costs? Costs will be determined in the same way as earlier works [1,2]; and we begin by assigning a memory location to each index. Because some indices may be modified at steps in the program, we can identify an index in a register as being in one of two states. An index is *passive* if the value of the index in the index register is the same as the value in memory. If the value of the index in a register has been modified since the index was last loaded from memory, the index is *active*. For an active index, the value in the index register is different from the value in memory. If we decide to remove an active index from a register, we must store its current value in its memory location. To remove a passive index, no "store" operation is necessary because the two values agree.

If we assign a cost of one unit to a load or a store operation, we can easily list all of the possible elementary costs:

(a) Replace an active index. Cost=2 (store the value of the active index and load the new index).

(b) Replace a passive index. Cost=1 (load the new index).

(c) Change an index from active to passive. Cost=1 (store the value of the active index).

(d) Change an index from passive to active. Cost=0 (no memory references required).

If a + is appended to an active index, then x_j^+ is an active index and x_j is a passive index.

The cost $c(Q_1,Q_2)$ of changing from configuration Q_1 to configuration Q_2 involves simply identifying occurrences of each of the four cases above. For

example, to change from $Q_1=(x_1^+,x_2,x_3^+,x_4)$ to $Q_2=(x_1,x_2^+,x_5^+,x_6)$, we change x_1^+ to x_1 (cost=1), change x_2 to x_2^+ (cost=0), replace x_3^+ (cost=2), and replace x_4 (cost=1). In this example, $c(Q_1,Q_2)$, the total cost of changing from Q_1 to Q_2, is 4.

The cost of an allocation A is simply the sum of the costs of the successive changes of configurations:

$$\text{cost}(A)=\sum_{i=1}^{n} c(Q_{i-1},Q_i),$$

where Q_0 is some initial configuration.

While the number of legal configurations at each step is finite, it may be impractically large. A result of Horwitz et al. [1] allows us to restrict the number of configurations we must evaluate and still be certain that we will find the optimal allocation from this reduced collection. The restriction involves considering only those configurations at step i which can be reached from a configuration at step $i-1$ by a *minimal change*. If Q_{i-1} is a configuration at step $i-1$, the configurations which can be reached from Q_{i-1} by minimal change are the following:

(1) A configuration Q_i which is identical to Q_{i-1}.

(2) A configuration Q_i which differs from Q_{i-1} only in that $P(i)$ is passive in Q_{i-1} and active in Q_i.

(3) All configurations Q_i which differ from Q_{i-1} only in that $P(i)$, which is not in Q_{i-1}, appears in Q_i replacing one of the indices in Q_{i-1}. To find the optimal allocation we will use the minimal-change definition above, beginning with some initial register configuration Q_0. To Q_0 we assign a weight of zero. Using the minimal-change rule, we generate configurations at step i, associating a weight and a parent pointer to each configuration. The parent pointer for configuration Q_i, $p(Q_i)$, points to the configuration at step $i-1$ from which Q_i was reached by minimal change. The weight of a configuration Q_i is

$$w(Q_i)=\min_{p(Q_i)}\{w(p(Q_i))+c(p(Q_i),Q_i)\}.$$

The weight of a configuration is defined in the context of a straight-line program. When the flow of control involves branches, this definition will require modification.

2. THE DYNAMIC-PROGRAMMING MODEL

The index-register allocation problem will now be recast as a dynamic-programming problem. The interest at this point is on the motivation for such

a model and the adequacy of dynamic programming as a methodology. The characteristics of an archetype dynamic program will be presented briefly, followed by the corresponding element in the index-register allocation problem.

Our first observation is that the problem is divisible into stages, which are the steps in the program. There is a policy decision at each stage: replacing an index or changing the state of an index. Further, there are a number of states associated with each stage—the states being the legal register configurations at each step. Because there are two problem statements, we will be using two names to describe the same thing: *step* in the program and *stage* in the process; and likewise, *configuration* and *state*. A feature of dynamic-programming models is that the policy decision at each stage transforms the current state into a state associated with the next stage. The decision in the register allocation problem accomplishes this transformation, with the "association with a stage" provided by the requirement that $P(i) \in Q_i$, $1 \leqslant i \leqslant n$. The Markov property that an optimal policy for the remaining stages must depend only on the current state is satisfied because only the current configuration can affect remaining allocations. Finally, a recursive relationship $w(Q_i)$ is available to identify the optimal policy.

In the definition of $w(Q_i)$, the weight of a configuration, the minimization over $p(Q_i)$ represents decision inversion [3] and arises from the following situation. Even with the minimal-change principle, there are often several ways of generating the same configuration. For example, on a machine with two index registers, suppose that three legal configurations were x_1x_2, x_1x_3, and x_1x_4, each with a weight of 5. At the next step, $P(i) = x_3^*$, so that x_3^+ must be present now in every configuration. By minimal changes, the configuration $x_1x_3^+$ can be reached by each of the three configurations above. But the weight associated with $x_1x_3^+$ is 5, because

$$\begin{aligned} w(x_1x_3^+) = \min\{ & w(x_1x_2) + c(x_1x_2, x_1x_3^+), \\ & w(x_1x_3) + c(x_1x_3, x_1x_3^+), \\ & w(x_1x_4) + c(x_1x_4, x_1x_3^+)\} \\ = \min\{ & 5+1, 5+0, 5+1\} \\ = 5. & \end{aligned}$$

A condition for the use of dynamic programming is the decomposition of the cost function. In index-register allocation, the cost is simply the number of memory references needed to change configurations. Such an additive cost

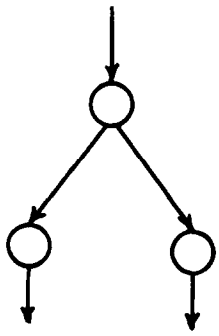

Fig. 2. Flow graph of diverging branch.

function is separable and monotonic, which are sufficient conditions for decomposition [3].

The reason for using dynamic programming is that it provides a unified methodology for handling the index-register allocation problem in programs with nonserial flow of control. To represent such programs, we use control flow graphs. A flow graph is a triple $G=(N, E, n_0)$ where

(i) N is a finite set of nodes,
(ii) $E \subseteq N \times N$ is a finite set of edges,
(iii) $n_0 \in N$ is the initial node.

The nodes in G represent basic blocks, that is, sequences of instructions which are executed in order. The edges represent possible transfers of control from one block to another.

In tree-structured programs the flow pattern which dominates is the diverging branch. The flow graph for the diverging branch in Fig. 2 might correspond to a LOGICAL IF statement in FORTRAN. A tree-structured program would be composed of straight-line sequences and, in general, several occurrences of this diverging branch structure, with possibly more branches than just two.

The diverging branch structure of Fig. 3 suggests the dynamic-programming formulation of the problem. In Fig. 3, the squares represent the stages (or

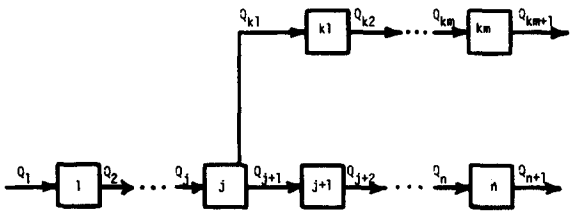

Fig. 3. Diverging branch as a dynamic program.

program steps). The state variables Q_i capture the essential information and describe the input and output at each stage. At stage j, the process branches.

The recursion equations from the last section require some alteration in view of this nonserial structure. In keeping with earlier studies of diverging branch systems [3], the solution will proceed backward. The diverging branch stages of Fig. 3 will be analyzed first, beginning with stage km and continuing until values for $w(Q_{k1})$ are found. Next, stages, $n, n-1,\ldots,j+1$ will be processed, resulting in values for $w(Q_{j+1})$. The returns from these two branches will be combined at stage j, where another straight-line segment from j to 1 will complete the solution.

The procedure will use the minimal change states at each stage. However, because of the backward direction, the minimization at each stage will take place over all *daughters* $d(Q_i)$ of a given configuration Q_i, i.e., those configurations (at the next stage) which can be reached by minimal change from the given configuration.

The recursion equations at each step are as follows:

(1) Stages $km, km-1,\ldots,k1$,

$$w(Q_{km}) = \min_{d(Q_{km})} c(Q_{km}, d(Q_{km})).$$

$$w(Q_{ki}) = \min_{d(Q_{ki})} \{c(Q_{ki}, d(Q_{ki})) + w(d(Q_{ki}))\}, \qquad i = m-1,\ldots,2,1.$$

(2) Stages $n, n-1,\ldots,j+1$,

$$w(Q_n) = \min_{d(Q_n)} c(Q_n, d(Q_n)),$$

$$w(Q_i) = \min_{d(Q_i)} \{c(Q_i, d(Q_i)) + w(d(Q_i))\}, \qquad i = n-1,\ldots,j+1.$$

(3) Stage j,

$$w(Q_j) = \min_{d(Q_j)} \{c(Q_j, d(Q_j)) + w(d_{j+1}(Q_j)) + w(d_{k1}(Q_j))\}.$$

(4) Stages $j-1,\ldots,2,1$,

$$w(Q_i) = \min_{d(Q_i)} \{c(Q_i, d(Q_i)) + w(d(Q_i))\}, \qquad i = j-1,\ldots,2,1.$$

When the two branches diverge at step j, the set of daughters $d(Q_j)$ contains configurations associated with step $j+1$ or $k1$. To identify these two groups, we denote them by $d_{j+1}(Q_j)$ and $d_{k1}(Q_j)$ in the recursion equation at stage j above.

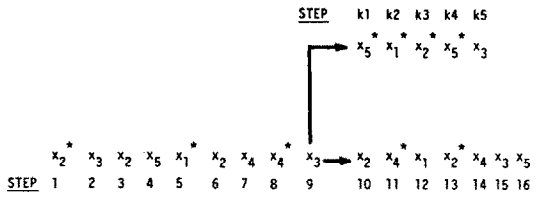

Fig. 4. Diverging branch example program with required index at each step.

3. A DIVERGING BRANCH EXAMPLE

The diverging branch in Fig. 4 involves five indices on a two-register machine. We begin by generating the minimal-change states, shown in Table 1, at each stage in the process. We follow the solution procedure above, beginning with stage $k5$ of the diverging branch. The results for the first three stages are displayed in Table 2 and explained below.

TABLE 1
Minimal-Change States for Diverging Branch Example

Step	Required index						
1	x_2^*	x_2^+					
2	x_3	$x_2^+x_3$					
3	x_2	$x_2^+x_3$					
4	x_5	$x_2^+x_5$	x_3x_5				
5	x_1^*	$x_1^+x_2^+$	$x_1^+x_5$	$x_1^+x_3$			
6	x_2	$x_1^+x_2^+$	x_2x_5	x_2x_3	$x_1^+x_2$		
7	x_4	$x_1^+x_4$	$x_2^+x_4$	x_2x_4	x_4x_5	x_3x_4	
8	x_4^*	$x_1^+x_4^+$	$x_2^+x_4^+$	$x_2x_4^+$	$x_4^+x_5$	$x_3x_4^+$	
9	x_3	$x_1^+x_3$	$x_2^+x_3$	$x_3x_4^+$	x_2x_3	x_3x_5	
10	x_2	$x_1^+x_2$	$x_2^+x_3$	$x_2x_4^+$	x_2x_3	x_2x_5	
11	x_4^*	$x_1^+x_4^+$	$x_2^+x_4^+$	$x_3x_4^+$	$x_2x_4^+$	$x_4^+x_5$	
12	x_1	$x_1^+x_4^+$	$x_1x_4^+$	$x_1x_2^+$	x_1x_3	x_1x_2	x_1x_5
13	x_2^*	$x_1^+x_2^+$	$x_2^+x_4^+$	$x_1x_2^+$	$x_2^+x_3$	$x_2^+x_5$	
14	x_4	$x_1^+x_4$	$x_2^+x_4^+$	$x_2^+x_4$	x_3x_4	x_1x_4	x_4x_5
15	x_3	$x_1^+x_3$	$x_3x_4^+$	$x_2^+x_3$	x_3x_4	x_1x_3	x_3x_5
16	x_5	$x_1^+x_5$	$x_4^+x_5$	$x_2^+x_5$	x_4x_5	x_1x_5	x_3x_5
$k1$	x_5^*	$x_1^+x_5^+$	$x_2^+x_5^+$	$x_4^+x_5^+$	$x_2x_5^+$	$x_3x_5^+$	
$k2$	x_1^*	$x_1^+x_5^+$	$x_1^+x_2^+$	$x_1^+x_4^+$	$x_1^+x_2$	$x_1^+x_3$	
$k3$	x_2^*	$x_2^+x_5^+$	$x_1^+x_2^+$	$x_2^+x_4^+$	$x_2^+x_3$		
$k4$	x_5^*	$x_2^+x_5^+$	$x_2^+x_5$	$x_1^+x_5$	$x_4^+x_5$	x_3x_5	
$k5$	x_3	$x_3x_5^+$	$x_2^+x_3$	$x_1^+x_3$	$x_3x_4^+$	x_3x_5	

TABLE 2
Recursion Analysis for Stages $k5$, $k4$, $k3$

Step i	Required index	States Q_i	Weight $W(Q_i)$	Daughter states $d(Q_i)$[a]
$k5$	x_3	$Q_{k5}^1 = x_2^+ x_5^+$	2	$Q_{k6}^1 = x_3 x_5^+$
		$Q_{k5}^2 = x_2^+ x_5$	1	$Q_{k6}^2 = x_2^+ x_3$
		$Q_{k5}^3 = x_1^+ x_5$	1	$Q_{k6}^3 = x_1^+ x_3$
		$Q_{k5}^4 = x_4^+ x_5$	1	$Q_{k6}^4 = x_3 x_4^+$
		$Q_{k5}^5 = x_3 x_5$	0	$Q_{k6}^5 = x_3 x_5$
$k4$	x_5	$Q_{k4}^1 = x_2^+ x_5^+$	$0+2=2$	$Q_{k5}^1 = x_2^+ x_5^+$
		$Q_{k4}^2 = x_1^+ x_2^+$	$2+1=3$	$Q_{k5}^3 = x_1^+ x_5$
		$Q_{k4}^3 = x_2^+ x_4^+$	$2+1=3$	$Q_{k5}^2 = x_2^+ x_5$
		$Q_{k4} = x_2^+ x_3$	$1+1=2$	$Q_{k5}^2 = x_2^+ x_5$
$k3$	x_2^*	$Q_{k3}^1 = x_1^+ x_5^+$	$2+2=4$	$Q_{k4}^1 = x_2^+ x_5^+$
		$Q_{k3}^2 = x_1^+ x_2^+$	$0+3=3$	$Q_{k4}^2 = x_1^+ x_2^+$
		$Q_{k3}^3 = x_1^+ x_4^+$	$2+3=5$	$Q_{k4}^2 = x_1^+ x_2^+$
		$Q_{k3}^4 = x_1^+ x_2$	$0+3=3$	$Q_{k4}^2 = x_1^+ x_2^+$
		$Q_{k3}^5 = x_1^+ x_3$	$1+3=4$	$Q_{k4}^2 = x_1^+ x_2^+$

[a]Must contain the required index.

If we expand on one of the entries in Table 2, it should clarify the use of the recursion equations. Under step $k3$, consider the first state Q_{k3}^1. At step $k3$, the index x_2^* is required, so it must be included in all of the configurations Q_{k4}. We have already, in the previous step, obtained the minimal cost weights $w(Q_{k4})$, for configurations Q_{k4}. Now we are proceeding one step backward, seeking the minimal costs in terms of configurations Q_{k3}. For state $Q_{k3}^1 = x_1^+ x_5^+$ the weight of 4 arises from the following calculation:

$$w(Q_{k3}) = \min_{d(Q_{k3})} \{c(Q_{k3}, d(Q_{k3})) + w(d(Q_{k3}))\},$$

$$\begin{aligned} w(x_1^+ x_5^+) &= \min\{c(x_1^+ x_5^+, x_1^+ x_2^+) + w(x_1^+, x_2^+), \\ &\qquad c(x_1^+ x_5^+, x_2^+ x_5^+) + w(x_2^+, x_5^+)\} \\ &= \min\{2+3, 2+2\} \\ &= 4. \end{aligned}$$

The daughter state which corresponds to this minimum value is $Q_{k4}^1 = x_2^+ x_5^+$. The analysis continues in like manner through the diverging branch until values for $w(Q_{k1})$ are obtained.

The key stage is 9, at which point the two branches are combined. The analysis at stage 9 is summarized in Table 3.

After stage 9, we are left with a straightforward serial analysis which follows the recursion equations given earlier. We conclude with a weight $w(Q_1)=20$. The interpretation is that there exists a sequence of decisions which will allow execution of the example program and require only 20 references to memory for the loading and storing of indices.

The solution—that is, the sequence of decisions which would achieve this minimal cost—is available by tracing through the chain of daughter pointers at every stage. The results of the recursion analysis were not presented here for most of the stages. However, when these results are included, we arrive at the solution in Fig. 5. The solution is expressed as the state (configuration) which must be present at each step in the program. The decisions which lead to that optimal solution are listed in Fig. 5 next to the program step at which each

STEP	CONFIGURATION	DECISION	CUMULATIVE MEMORY REFERENCES
0	ϕ		0
1	x_2^+	LOAD x_2	1
2	$x_2^+ x_3$	LOAD x_3	2
3	$x_2^+ x_3$		2
4	$x_2^+ x_5$	LOAD x_5	3
5	$x_1^+ x_2^+$	LOAD x_1	4
6	$x_1^+ x_2^+$		4
7	$x_1^+ x_4$	STORE x_2;LOAD x_4	6
8	$x_1^+ x_4^+$		6
9	$x_1^+ x_3$	STORE x_4;LOAD x_3	8

STEP	CONFIGURATION	DECISION	STEP	CONFIGURATION	DECISION	
10	$x_1^+ x_2$	STORE x_2	k1	$x_1^+ x_5^+$	LOAD x_5	10
11	$x_1^+ x_4^+$	LOAD x_4	k2	$x_1^+ x_5^+$		11
12	$x_1^+ x_4^+$		k3	$x_2^+ x_5^+$	STORE x_1 LOAD x_2	13
13	$x_2^+ x_4^+$	STORE x_1 LOAD x_2	k4	$x_2^+ x_5^+$		15
14	$x_2^+ x_4^+$		k5	$x_2^+ x_3$	STORE x_3 LOAD x_5	17
15	$x_3\ x_4^+$	STORE x_2 LOAD x_3				19
16	$x_4^+ x_5$	LOAD x_5				20

Fig. 5. Optimal solution to diverging branch example.

TABLE 3
Recursion Analysis at Stage 9[a]

States Q_9	Weight $w(Q_9)$[b]	Daughter states $d(Q_9)$[c]
$Q_9^1 = x_1^+ x_4^+$	2+7+5=14	$x_1^+ x_3$
$Q_9^2 = x_2^+ x_4^+$	2+6+6=14	$x_2^+ x_3$
$Q_9^3 = x_2 x_4^+$	1+6+7=14	$x_3 x_4^+$
$Q_9^4 = x_4^+ x_5$	1+6+7=14	$x_3 x_4^+$
$Q_9^5 = x_3 x_4^+$	0+6+7=13	$x_3 x_4^+$

[a]Required index: x_3.
[b]$c(Q_9, d(Q_9)) + w(d_{10}(Q_9)) + w(d_{k1}(Q_9)) = w(Q_9)$.
[c]Must contain required index.

decision should be made. Alongside is the cumulative number of memory references required, totaling 20 in agreement with $w(Q_1)$.

4. SEVERAL DIVERGING BRANCHES

Tree-structured programs may divide into more than just two branches at a point in the program. Many languages have multiway switch statements, breaking a single program flow path into many possible paths. Examples include the computed GO TO or ARITHMETIC IF statements of FORTRAN, or the CASE statement in ALGOL.

With several diverging branches, the recursion analysis is no more difficult than that shown above. At the diverging stage, the weight simply includes each of the individual stage costs plus the weights from each of the separate branches.

REFERENCES

1. L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd, Index register allocation, *J. Assoc. Comput. Mach.* 13:43–61 (1966).
2. K. Kennedy, Index register allocations in straight line code and simple loops, in *Design and Optimization of Compilers*, (R. Rustin, Ed.), Prentice-Hall, Englewood Cliffs, N.J., 1972, pp. 51–63.
3. G. L. Nemhauser, *Introduction to Dynamic Programming*, Wiley, New York, 1966.

Received October 1978