Comput. & Ops Res., Vol. 4, pp. 101–110. Pergamon Press, 1977. Printed in Great Britain

DYNAMIC PROGRAMMING FOR COMPUTER REGISTER ALLOCATION

William W. Agresti†
University of Michigan-Dearborn Campus, Dearborn, Michigan 48128, U.S.A.

Scope and purpose—Computers and operations research admit to a rich and complex interface. Most examples of problem solving in this domain use both computers and operations research as tools to solve problems[1, 2]. In particular, the computer is used as a tool in implementing a solution technique arising from an operations research model. A different aspect of this interface is explored here: operations research methodology is used as an analytical tool for a computer problem. With their widespread use, computers themselves comprise an application area of vital concern. Operations research approaches can be valuable in this regard, as evidenced by queueing models of computer operating systems[3] and mathematical programming models of information systems[4]. In the present work, dynamic programming is used to solve a problem in the optimization of computer programs. Programs written in a high-level language (like FORTRAN) are translated into machine language by a compiler. The general problem is to make this machine language version efficient (i.e., execute fast and conserve memory). A specific problem of this type—namely, index register allocation—is formulated in this paper. The solution is a dynamic programming-based procedure which could be included in the compiler to generate more efficient programs.

Abstract—A procedure for optimal index register allocation in loops is described. The procedure is a result of the dynamic programming formulation of the index register allocation problem for other than straight-line code. An example involving a simple loop is solved.

1. INTRODUCTION

The impact of the computer is undeniable; and part of the effect is due to the availability of high-level languages. The rapid symbol-processing ability of the computer would be less often tapped if the only access to it were the direct manipulation of a machine's instruction set. Better suited to problem definition, FORTRAN and COBOL and the others facilitate the diverse applications which characterize the broad influence of computers. There is a price to pay for such flexibility: programs written in these languages must be translated to become machine-sensible. Compilers are the software which effect this translation into machine language.

Even with the earliest compilers, the concern was not merely to accomplish the translation, but to translate *efficiently*. Attention has been focused on describing an *optimization phase* for the compilation process. The use of the word "optimization" owes more to history than to accuracy. By compiler optimization is meant that transformations are introduced during compilation which change one version of a user's program into an equivalent version which executes faster or consumes less memory. Such transformations have been called "optimizations" but there is usually no attempt to prove that the transformed program is "best" with respect to some cost function. (Indeed, there are serious theoretical hindrances to such a proof.) A more accurate term would be "code-improving transformation" which allows that the machine language program (code) has been improved but not optimized.

From the construction of the first FORTRAN compiler until the present, register allocation has offered one of the best opportunities for program optimization by the compiler. The general issue in register allocation is to describe rules by which a compiler will use a machine's available registers in the best possible manner. 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

†William W. Agresti graduated from Case Institute of Technology with a B.S. in Management Science. He received a M.S. in Industrial Engineering and a Ph.D. from New York University. Dr. Agresti is Assistant Professor of Industrial and Systems Engineering and Director of the Computer and Information Science Program at the University of Michigan–Dearborn. His research interests are in the area of computer software and simulation.

statements. In this way, the program will run faster because the required quantities will not have to be fetched from memory. Of course, there are only a limited number of registers; 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 scope 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 is arranged in storage in some systematic way. As an example, suppose that we wish to calculate the sum of the contents of 100 consecutive memory locations, beginning at location K. Our computer has instructions with the format,

```
OPER   REG,ADDR
```

where "OPER" is the operation; "REG" is the register which holds one operand (and which will hold the result); and "ADDR" is the address of the second operand. We can accumulate the sum in register 1 with the sequence,

```
LOAD    1, K
ADD     1, K+1
ADD     1, K+2
 .
 .
 .
ADD     1, K+99
```

A computer which provides indexing offers a less tedious alternative. With indexing, the effective address of the second operand is calculated as ADDR *plus* the contents of the specified index register. Indexing is indicated by enclosing the index register in parentheses:

```
OPER    REG,ADDR(INDEX REG)
```

If register 2 were used for indexing, we can add one to its contents each time through a loop, thereby adding one to the effective address of the operand. Calculating the sum by indexing would now be accomplished by:

```
        LOAD    1,ZERO
        LOAD    2,ZERO
LOOP    ADD     1,K(2)
        ADD     2,ONE
       { if not through  }
       { loop 100 times, }
       { then go to LOOP }
```

where "ZERO" and "ONE" are two locations which contain the numbers zero and one, respectively.

As even this simple example shows, indexing is an efficient technique. It can help reduce the running time of the machine language programs produced by compilers. The allocation problem for index registers has been studied by Horwitz *et al.*[5] and by Luccio[6]. In [5], a procedure is given for specifying which quantities should occupy index registers at each point in a program so that the number of memory references (from LOAD and STORE operations) is minimized. The programs which were examined in [5] and [6] involved only linear flow of control—i.e. no branches or loops. More recent work by Kennedy[7] 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 loops and branches which more realistically characterize computer programs. A new methodology, dynamic programming, is introduced to find solutions.

2. THE INDEX REGISTER ALLOCATION PROBLEM

We are interested in those computers which provide some number of index registers. Two types of instructions are of interest. One type *refers* to the contents of an index register; the last section had an example,

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

The second type of instruction *modifies* the contents of the index register, e.g.,

```
ADD     2,ONE
```

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:

$$x_1x_3^*x_2x_4x_2^*x_3x_1x_2.$$

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 $P(1) = x_1$, $P(2) = x_3^*, \dots, P(8) = x_2$.

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 an n-step program P is a sequence of configurations,

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

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,

$$P(i) \in Q_i \quad 1 \leq i \leq n.$$

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 work[5, 7]; 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.*[5] 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 straight-line code. When the flow of control involves branches and loops, this definition will require modification.

To further reduce the number of configurations, a cleansing rule[5, 7] can be applied: if Q_i^1 and Q_i^2 are two configurations associated with step i such that

$$w(Q_i^1) + c(Q_i^1, Q_i^2) \leq w(Q_i^2)$$

then eliminate Q_i^2. As soon as the minimum change configurations have been generated and the weights assigned, this cleansing rule can be tried. The effect of removing a configuration Q_i^2 is that we need not use it as a basis for generating states at the next step $i+1$.

3. 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 state: 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 $P(i) \in Q_i$, $1 \leq i \leq n$. 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[8] 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

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

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 function is separable and monotonic, which are sufficient conditions for decomposition[9].

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. The control flow graph of a simple loop appears in Fig. 1.

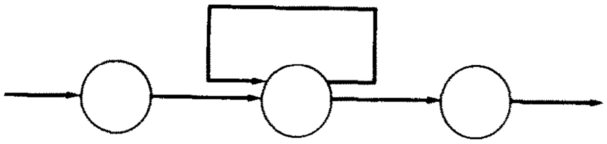

Fig. 1. Control flow graph of a simple loop.

For a program with n steps, the simple loop structure of Fig. 2 suggests the dynamic programming treatment of the problem. In Fig. 2, the squares represent the stages (or program steps). The state variables Q_i capture the essential information and describe the input and output at each stage. Stages j through k comprise the loop: these program steps are executed $n > 0$ times.

Solving the index register allocation problem for a simple loop involves first expressing the correct recursion equations for each step:

1. Stages 1 to $j-1$,

$$w(Q_1) = \min_{p(Q_1)} c(p(Q_1), Q_1)$$

$$w(Q_i) = \min_{p(Q_i)} \{c(p(Q_i), Q_i) + w(p(Q_i))\} \quad i = 2, \ldots, j-1$$

2. Stages j to $k-1$

$$w(Q_j, Q_k) = \min_{p_{j-1}(Q_j)} \{c(p_{j-1}(Q_j), (Q_j) + c(Q_k, Q_j) + w(p_{j-1}(Q_j))\}$$

$$w(Q_i, Q_k) = \min_{p(Q_i)} \{c(p(Q_i), Q_i) + w(p(Q_i), Q_k)\} \quad i = j+1, \ldots, k-1$$

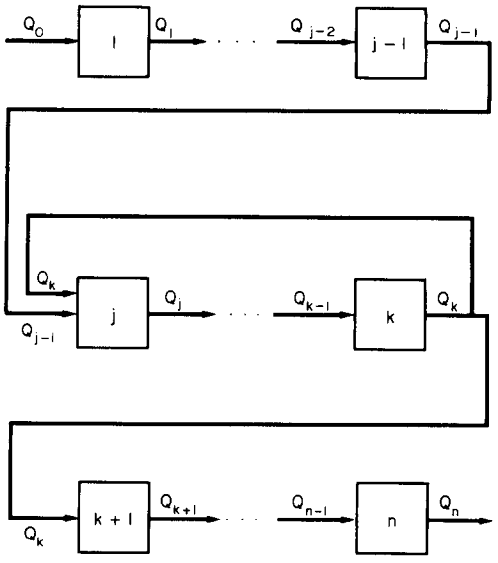

Fig. 2. Steps in a simple loop program as stages in a dynamic programming model.

3. Stage k

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

4. Stages $k+1, \ldots, n$

$$w(Q_i) = \min_{p(Q_i)} \{c(p(Q_i), Q_i) + w(p(Q_i))\} \quad i = k+1, \ldots, n.$$

When the two branches converge at step j, the set of parents $p(Q_j)$ contains configurations associated with step $j-1$ or k. We will denote by $p_{j-1}(Q_j)$ those parents associated with step $j-1$. This notation appears in the recursion equations at stage j.

Given an initial configuration Q_0, we will consider each stage in turn from 1 to n. At stage i we use the appropriate recursion equation above to find the minimum weight as a function of all output stages Q_i. At the last stage n, we identify the output state Q_n with the minimum weight and use the parent pointers to identify the other configurations $Q_{n-1}, Q_{n-2}, \ldots, Q_1$ which make up the optimal allocation.

A special problem arises when loops are considered. When the backlatch arc is taken, it is not sufficient that the desired quantities are merely in registers; they must be in the *same* registers as they were at the start of the loop. For this matching problem, we make the following assumption, used by other investigators[7, p. 62]. We assume that the register assignments are made identical by using register-to-register moves. Such instructions are much cheaper than loads and stores on many machines. They are not represented in our cost function which counts only memory references.

The effect of the loop on the solution process is to increase the dimensionality of the optimization problem for states j through k. For those states, instead of expressing the minimum weight in terms of the output state Q_i alone, we must express the minimum weight as a function of two states Q_i and Q_k. The reason for this change is that the configuration Q_k affects the decision at stage j. As a consequence, Q_k must remain as a variable throughout the loop portion, stages j through k. Of course, the possible configurations Q_k ordinarily would not be known earlier in the program at stages $j, j+1, \ldots, k-1$. But because the states Q_k must be available earlier, we must effectively unroll the loop once and use as configurations Q_k those states which are associated with the second occurrence of step k. In summary, we will follow the procedure below for the simple loop. (Those configurations reached by minimum change are called minimum change states.)

Solution procedure

(a) Generate minimum change states Q_1
(b) Obtain $w(Q_1)$

(c) For stages 2 through $j-1$,
  (i) Generate minimum change states
  (ii) Apply cleansing rule
  (iii) Obtain $w(Q_i)$
(d) By unrolling the loop once, generate minimum change states for stages j through k
(e) For stages j through $k-1$, obtain $w(Q_i, Q_k)$
(f) Obtain $w(Q_k)$
(g) For stages $k+1$ through n,
  (i) Generate minimum change states
  (ii) Apply cleansing rule
  (iii) Obtain $w(Q_i)$.

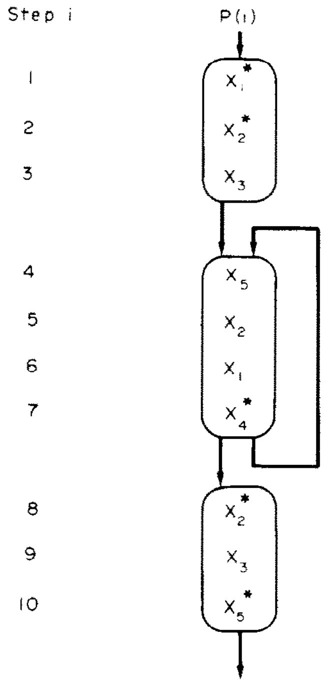

Fig. 3. Simple loop example.

4. A SIMPLE LOOP EXAMPLE

The simple loop in Fig. 3 involves five indices on a two-register machine. We follow the solution procedure above, operating on the first three stages of the process. The results are summarized below.

Step	Program	States	Weight	Parent
1	x_1^*	$Q_1^1 = x_1^+$	1	Q_0
2	x_2^*	$Q_2^1 = x_1^+x_2^+$	2	Q_1^1
3	x_3	$Q_3^1 = x_1^+x_3$	4	Q_2^1
		$Q_3^2 = x_2^+x_3$	4	Q_2^1

For example, at step 3 in the program the index x_3 is referenced. Two minimum change states are feasible—$Q_3^1 = x_1^+x_3$ and $Q_3^2 \doteq x_2^+x_3$, each with a weight of 4 and parent state Q_2^1.

Having completed the initial straight-line part of the graph in Fig. 3, we unroll the loop once to obtain the minimum change states for stages in the loop (see Table 1). Now, for stages 4, 5, and 6, we obtain the weight $w(Q_i, Q_7)$. The increased dimensionality of the problem for stages in the loop requires maintaining an array of values $w(Q_4, Q_7)$ instead of a simple list. (It should be noted that the use of a sequential search procedure along with the recursion analysis may reduce these storage requirements[9, p. 197].)

Table 1. Minimal change states for stages in loop

	Step	Program	Minimum Change States						
First iteration	4	x_5	$x_1^+x_5$	$x_2^+x_5$	x_3x_5				
	5	x_2	x_2x_5	x_2x_3	$x_2^+x_5$	$x_1^+x_2$			
	6	x_1	x_1x_2	$x_1x_2^+$	x_1x_3	$x_1^+x_2$	x_1x_5		
	7	x_4^*	$x_1x_4^+$	$x_2x_4^+$	$x_2^+x_4^+$	$x_3x_4^+$	$x_1^+x_4^+$	$x_4^+x_5$	
Second iteration	4	x_5	$x_1^+x_5$	$x_2^+x_5$	x_3x_5	x_1x_5	x_2x_5	$x_2^+x_5$	
	5	x_2	x_2x_5	x_2x_3	$x_2^+x_5$	$x_1^+x_2$	x_1x_2	$x_2x_4^+$	
	6	x_1	x_1x_2	$x_1x_2^+$	x_1x_3	$x_1^+x_2$	x_1x_5	$x_1x_4^+$	
	7	x_4^*	$x_1x_4^+$	$x_2x_4^+$	$x_2^+x_4^+$	$x_3x_4^+$	$x_1^+x_4^+$	$x_4^+x_5$	

Table 2. Recursion analysis at stage 4

	Q_7^1	Q_7^2	Q_7^3	Q_7^4	Q_7^5	Q_7^6
	$x_1x_4^+$	$x_2x_4^+$	$x_4^+x_5$	$x_2^+x_4^+$	$x_3x_4^+$	$x_1^+x_4^+$
$Q_4^1 = x_1x_5$	8	9	8	10	9	9
$Q_4^2 = x_2x_5$	9	8	8	9	9	10
$Q_4^3 = x_2^+x_5$	8	7	7	7	8	9
$Q_4^4 = x_3x_5$	9	9	8	10	8	10
$Q_4^5 = x_4^+x_5$	8	8	7	9	8	9
$Q_4^6 = x_1^+x_5$	7	8	7	9	8	7
	Values of $w(Q_4, Q_7)$					

To illustrate the use of the recursive equations, consider the entry in the upper left-hand corner of Table 2, $w(Q_4^1, Q_7^1) = 8$. This value arises from the following calculation:

$$\begin{aligned} w(Q_4^1, Q_7^1) &= \min\{c(Q_3^1, Q_4^1) + c(Q_7^1, Q_4^1) + w(Q_3^1), \\ &\qquad c(Q_3^2, Q_4^1) + c(Q_7^1, Q_4^1) + w(Q_3^2)\} \\ &= \min\{2+2+4, 3+2+4\} \\ &= 8. \end{aligned}$$

The parent state is Q_3^1. The analysis continues in like manner to stages 5 and 6, so that an array of values $w(Q_6, Q_7)$ is obtained.

It is only at this point that we are able finally to resolve the matter of dealing with that critical backlatch arc that defines the loop. Until now we have had to retain an extra decision variable Q_7 in all of our calculations. At stage 7, the effect of various states Q_7 on the loop can be assessed. A summary of the analysis at stage 7 appears in Table 3, along with the results for the remaining stages (which constitute a straightforward linear program segment).

Table 3. Results of recursion analysis for stages 7–10

Step	Program	States	Weight	Parent
7	x_4^*	$Q_7^1 = x_1x_4^+$	10	$Q_6^1 = x_1x_2$
		$Q_7^2 = x_2x_4^+$	10	$Q_6^1 = x_1x_2$
		$Q_7^3 = x_4^+x_5$	10	$Q_6^2 = x_1x_3$
		$Q_7^4 = x_2^+x_4^+$	9	$Q_6^3 = x_1x_2^+$
		$Q_7^5 = x_3x_4^+$	11	$Q_6^4 = x_1x_3$
		$Q_7^6 = x_1^+x_4^+$	9	$Q_6^6 = x_1^+x_2$
Cleansing rule eliminates Q_7^1, Q_7^2, Q_7^5				
8	x_2^*	$Q_8^1 = x_2^+x_4^+$	9	Q_7^4
		$Q_8^2 = x_2^+x_5$	11	Q_7^3
		$Q_8^3 = x_1^+x_2^+$	11	Q_7^6
Cleansing rule eliminates Q_8^2, Q_8^3				
9	x_3	$Q_9^1 = x_2^+x_3$	11	Q_8^1
		$Q_9^2 = x_3x_4^+$	11	Q_8'
10	x_5^*	$Q_{10}^1 = x_2^+x_5^+$	12	Q_9^1
		$Q_{10}^2 = x_3x_5^+$	13	Q_9^1
		$Q_{10}^3 = x_4^+x_5^+$	12	Q_9^2

To extract the solution, we examine the states associated with the final stage and identify the state with the lowest weight. Both Q_{10}^1 and Q_{10}^3 have a weight of 12. To complete this example, we will trace back from Q_{10}^1 to find one of the solutions. We use the parent pointers from Table 3 for stages 7–10. The recursion analysis with parent pointers was not presented completely for stages in the loop. However, when these results are included and the earlier summary of stages 1–3 is used, we arrive at the solution in Fig. 4. Recall that the "weight = 12" specified the total number of load and store operations that would be required to perform the allocation. But more must be said about the interpretation of the solution in light of the number of iterations through the loop.

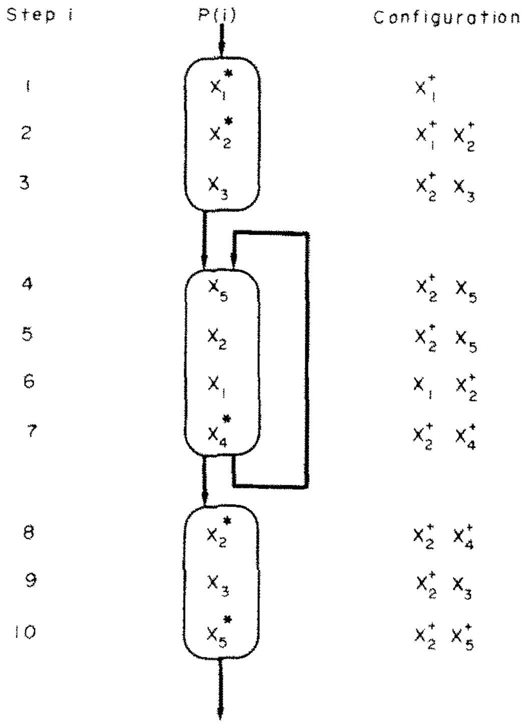

Fig. 4. An optimal solution to the simple loop example. Total cost = 12.

By the way in which the recursion equations operate, the solution corresponds to two iterations through the loop. That is, equal value is given to all arcs of the control graph. The cost of changing configurations from step 3 to 4 is included once; and the cost of changing configurations from step 7 to step 4 is included once. In the absence of any information about the frequency with which various paths in the program are followed, it may be reasonable to assume that each arc is equally likely to be traversed. But when the arcs, instead of being simple diverging branches, form loops in the program, those arcs typically would be followed more than once. Accordingly, the cost of changing configurations from step 7 to step 4 would be incurred more than once, as would configuration changes during other steps in the loop.

A benefit of the dynamic programming formulation is that any frequency flow information can be easily included in the analysis. If it were estimated that there would be m iterations of the simple loop, then the coefficient m would be introduced into the recursion equations for the loop stages j through k:

$$w(Q_j, Q_k) = \min_{p_{j-1}(Q_j)} \{c(p_{j-1}(Q_j), Q_j) + (m-1) \cdot c(Q_k, Q_j) + w(p_{j-1}(Q_j))\}$$

$$w(Q_i, Q_k) = \min_{p(Q_i)} \{m \cdot c(p(Q_i), Q_i) + w(p(Q_i, Q_k)\} \quad i = j+1, \ldots, k-1$$

$$w(Q_k) = \min_{p(Q_k)} \{m \cdot c(p(Q_k), Q_k) + w(p(Q_k), Q_k)\}.$$

Such a modification would preserve the meaning of the weight function as a count of the total number of load and store operations required to accomplish the allocation. Further, it would clarify the effects of packing registers with quantities over busy portions of a program.

5. CONCLUSION

The problem of optimal index register allocation in a program with a simple loop was solved by a dynamic programming model. The formulation offers a uniform method of treating index register allocation in nonserial programs. The model conveniently accepts frequency flow data and may be useful in more general register allocation problems, especially in clarifying the packing of registers in nested loops.

A larger implication in this work is a possible bridge between dynamic programming and the operation of an optimizing compiler. The input to the compiler is a source program written in a high-level language. The compiler operates on this input, successively changing it through intermediate forms into a final machine language version. It is appealing to view such a progression as a staged decision process. Add to this the goal of generating an "optimal" machine language program and the entire compilation process suggests a dynamic programming formulation. Such speculation remains to be explored.

REFERENCES

1. J. H. Engel, The philosophical underpinnings of computers and operations research, *Comput. Ops. Res.* **1**, 3 (1974).
2. A. Mjosund, The synergy of operations research and computers, *Ops. Res.* **20**, 1057 (1972).
3. L. Kleinrock, *Queueing Systems Vol. II, Computer Applications.* John Wiley, New York (1976).
4. J. R. Nunamaker, Jr., A methodology for the design and optimization of information processing systems, *Proc. Spring Joint Computer Conf.*, pp. 283–294. AFIPS Press, Montvale, N.J. (1971).
5. L. P. Horwitz, R. M. Karp, R. E. Miller and S. Winograd, Index register allocation, *J. Assoc. Comput. Mach.* **13**, 43 (1966).
6. F. Luccio, A comment on index register allocation, *Comm. Assoc. Comput. Mach.* **10**, 572 (1967).
7. K. Kennedy, Index register allocation in straight line code and simple loops, *Design and Optimization of Compilers*, R. Rustin (ed.), pp. 51–63. Prentice-Hall, Englewood Cliffs, N.J. (1972).
8. R. Aris, G. L. Nemhauser and D. J. Wilde, Optimization of multistage cyclic and branching systems by serial procedures, *Am. Inst. Chem. Eng. J.* **10**, 913 (1964).
9. G. L. Nemhauser, *Introduction to Dynamic Programming.* John Wiley, New York (1966).