Optimization Strategies for Microprograms

R. L. KLEIR, STUDENT MEMBER, IEEE, AND C. V. RAMAMOORTHY, MEMBER, IEEE

Abstract—With increased use of microprogramming in present computer systems, the need arises to automate the checking and optimization of microcode. This paper reviews the optimization objectives, characterizes microprograms, and discusses the machine structure. A translator from machine code to microcode appears feasible and is described. Some compiler techniques are reviewed and adapted to improve microprograms by studying operational interaction. Microprogram characteristics permit additional methods to reduce the computation effort. The techniques are integrated into a scheme implemented for optimizing a simulated machine.

Index Terms—Inefficient code sources, intermediate text, machine code translator, microprogram characteristics, microprogramming, optimization, program graph.

I. INTRODUCTION

WITH increased interest in expanding the uses of microprogram routines and the advent of READ/WRITE control memory, some effort has been directed toward alternatives to hand-coded microroutines (e.g., microprogram compilers). Significant benefits accrue by removing the instruction FETCH and DECODE overhead from the user's routine [11]. However, an optimization phase, similar to that used in current compilers, would be desirable to compete effectively with hand coded programs.

This paper addresses some aspects of optimization in microprograms. Some compiler techniques are adapted and other new techniques developed.

Because the details of microprogrammable machines vary widely, the methods do not presume applicability to all machine structures. To avoid restricting the application to a particular machine example, hardware characteristics assumed are a composite of features common to a large variety of contemporary machines.

II. OVERVIEW OF OPTIMIZATION

"Optimization" here does not guarantee production of the absolute minimum computation effort form of a process. Since the minimum effort cannot in general be predicted from the process specification, improvements are judged by comparing the efficiency of different versions of the same program. Optimization in this sense is actually modification of the original program to reduce either the computation time or computer resources required without introducing possible logic flaws.

Compilers which produce programs in an algorithmic fashion (e.g., syntax direct compilers) must produce correctly functioning routines regardless of the program sequences involved. Hence, inefficiencies are often introduced to protect against special cases. It is usually desirable to later scan the object code produced (i.e., optimization pass) to check for the occurrence of the special cases; if one does not exist, the protection code may be deleted.

For example, a logarithm evaluation may check its input for being nonnegative before computing the result. If the input is always obtained from an absolute value evaluation in a particular program, the input can never be negative. Hence, the test in the logarithm routine is superfluous in this program and could be removed.

Other identifiable sources of inefficient code are the following: 1) Changes During Development and Maintenance: Although the original set of code may contain little superfluous code, changes introduced during checkout or a change in specifications may introduce modifications that would allow a compression in the program. 2) Tutorial Organization: The program may be written to ease the maintenance and understanding of the process involved. A different organization of program subparts may be more desirable to expedite computation. 3) Debug Aids: The order and manner of execution may be designed to allow easy control and sampling of intermediate results. Some values saved are not required after checkout has been completed. 4) Library Functions: If the program includes code drawn from the system library, the user is usually not familiar with the internal structure of the library routine. Therefore, he may utilize only portions of the routine or perform checks that are also performed in the library code.

Once a working set of code is achieved, there is considerable reluctance to modify the code further and remove any inefficiency; therefore, the superfluous effort may be included in the production version also.

The optimizer's function is to analyze the program subparts with an overall view of the process environment. Some basic functions of an optimizer are: 1) segmentation and analysis of the flow characteristics in the program; 2) removal of nonessential operations; and 3) reordering execution to improve the computation speed. Although other more sophisticated actions are performed for improving object code in current compilers, these do not appear applicable to reducing microprograms.

Since the purpose of optimization is to increase machine efficiency, more time should not be spent optimizing the program than could possibly be recovered by reductions in executing the improved program. Therefore, optimization should be selectively applied and some programs either not improved or only partially reduced. The conditions for which the optimizer searches should be of a general nature and occur frequently. Analysis time should not be wasted searching for limited and infrequently occurring situations.

Manuscript received August 15, 1970; revised March 1, 1971.

The authors are with the Department of Electrical Engineering, University of Texas, Austin, Tex. 78712.
III. MICROPROGRAMMABLE MACHINES AND MICROPROGRAMS

To avoid ambiguity in terms used to describe microprograms, the following definitions are presented.

1) *Machine instructions* are the basic machine functions sanctioned by the manufacturer. To users above the microprogram level, these functions are considered the lowest level of machine control.

2) *Machine code or machine programs* are processing routines realized by a specification in machine instructions (e.g., assembly language programs).

3) A *microstep* is the quantum of time required to execute one microinstruction.

4) A *microoperation* is a unit of microprogram activity which performs a particular function (e.g., GATE, ADD, SHIFT, etc.).

5) A *microinstruction* is the specification of microoperations to take place during one microstep. The microinstruction may contain a single operation (vertical microprogramming) or specify several concurrent activities (horizontal microprogramming).

Since considerable attention has been given to the structure of microprogrammable machines in the literature [7]–[11], and the design details vary widely among machines, the structure discussion here will be limited to defining terms used later.

The machine discussed is assumed to have the following hardware structures.

1) *Main Memory:* This mass storage device is assumed to be the slowest of the storage types in the machine. It is assumed to operate asynchronously under microprogram control with a REQUEST/REPLY type interface.

2) *Control Storage (Micromemory):* This storage device holds the system of microprograms invoked during the execution of a machine instruction. Since more powerful machines can be realized with a READ/WRITE control storage and current machine development appears headed in this direction, the READ/WRITE capability will be assumed here.

3) *Local Storage:* These elements comprise the machine register file and associated special event latches (e.g., overflow, comparison results, etc.). They represent the fastest of the memory types.

4) *Functional Units:* These devices perform actual manipulations of the data for a microinstruction. They represent selector gates, arithmetic units, Boolean operators, comparators, and the like.

5) *Data Paths:* This collection of hardware routing paths interconnect the functional units and various memory types. The paths provided in a machine realization apply an important constraint upon the microoperations allowed.

To evaluate compiler optimization techniques and develop special modifications for application to microprograms, the characteristics of the microprogram must be examined. Since the original proposal of Wilkes, microprograms have retained a distinct data routing flavor. That is, microprograms fundamentally direct the sequence of data flows within the internal machine structure. Additionally, microprograms (particularly as compared to machine programs) display the following characteristics.

1) *Machine Timing:* Microprograms must consider the internal timing of the host machine. Both the data path propagation delay and the execution time required by functional units determine when a particular sequence of operations may produce a timing hazard. A timing hazard occurs whenever the microprogram assumes valid data availability too early or late for the machine electronics.

2) *Machine Realization:* The microoperation repertoire and interconnectable elements are limited by the internal structure of the host machine. The microlanguage can be designed to maximize parallel activities within the hardware.

3) *Parallelism:* Parallel activity is incorporated on a cost effective basis. Parallel hardware is more expensive and its microprograms are more complex, but if the parallel feature sufficiently increases the machine capability, the cost can be justified.

4) *Operation Complexity:* The fundamental operations performed are fairly simple. Complex functions (e.g., floating-point arithmetic, multiplication, division, etc.) are often formed from a sequence of microsteps.

5) *"Variables":* The machine storage devices (e.g., registers) are analogous to machine program variables. The number of variables, their structure, and some implied information on the frequency characteristics is known for all microprograms on a machine. Efficient microprogram algorithms access the local storage data more often than main memory data. Hence, scalar variables of machine language programs can be likened to the registers and the machine program array to one-dimensional arrays of mass memory and control store.

IV. EXPANSION OF MACHINE CODE TO MICROCODE

Rather than discuss a compiler with output code in microinstructions, we will describe a technique for translating the program from assembly type code into microcode. Assume that a program is available that describes some process in machine code. If the program were executed on the host machine in normal fashion, each instruction would be interpreted in sequence. The flow chart of Fig. 1 exemplifies the activities a microprogram might pursue for the interpretation. The activities OP1, OP2,... correspond to the microcode which performs the functional operation. Other activities are the required overhead functions and certain run time modifications in the execution (e.g., indexing and indirect addressing). Hence, each machine instruction invokes some "macro" of microcode to perform the function.

If we were to design a translator such that the instruction FETCH and DECODE were performed by the translator, and the corresponding macros expanded sequentially retaining run time modifying sequences as required, the result would be a functionally equivalent set of microinstructions that produce the same output results. The value of removing the instruction FETCH/DECODE overhead has been demonstrated [11]. It is not hard to imagine, however, that such a translation would result in some inefficient operations, particularly at the interfaces of the "macros," since each macro is locally concerned with its particular functional operation.

For translation to be a credible approach, however, several constraints must be imposed upon the original program. Self-modifying machine code must be excluded. That is,
Machine programs that alter the instruction stream during execution would offer enormous problems to both translation and optimization analysis. Further difficulties are presented by indexing and indirect addressing.

Indexing can be classified into data references and indexed branches. If we assume the data references to be proper data items (not executable instructions) then a translation that preserved the relative order and spacing of data items would preserve indexed references. Indexed branches could be resolved by using transfer vectors (i.e., branching to a table of branch instructions for proper linkage to the expanded version).

Simple indirect addressing (i.e., single word pointers) would be resolved by techniques similar to those used for indexing. More complex indirect addressing (e.g., multi-level) might be more formidable. Since the routines to be encoded would be of fairly limited complexity (necessary to fit into the control storage), probably the logical effect of indirect addressing could be formulated using some other mechanism.

V. Analysis and Representation of Microprograms

Since the optimization procedure requires a comprehensive description of both the various operations to be performed and the interrelationships between code sections, several tools will be described to aid this analysis.

The Microprogram: As previously mentioned, the actual structure and microlanguage of machines vary widely. To avoid limitation to a particular implementation, the following abstract description of microprograms is developed. It is assumed that the host machine can be described in terms of both structure and allowed operations. The structure will be a partitioning of the machine into $M$ distinct sections called machine subparts. The partitions will indicate the lowest level of microcontrol. For example, if the register file consists of 16 registers addressable to the half register level, 32 subparts will be required. Operations utilizing a full register specify two half register subparts. Some machine subparts possess memory characteristics (e.g., registers) while others can be considered as combinational in nature (e.g., data paths and asynchronous functional units).

Microoperations allowed by the machine are enumerated by a listing of primitive machine operations; microinstructions are a set of these operations and machine subparts. A microaction is a specification of a machine operation with a particular set of machine subparts. Microactions will be represented by the notation:

$$I_q \{i\} OP_j \{k\} \{u\}$$

where

- $\{i\}$ and $\{k\}$ Machine memory subparts used as inputs and outputs respectively for the microoperation.
- $OP_j$ One of the host machine’s primitive operations.
- $\{u\}$ Machine subparts that are used in some transitive fashion to accomplish the operation. These are usually functions of the operation desired and the memory subparts used.
- $I_q$ Some unique index assigned to the particular operation and machine subpart description used to identify this combination.

Elements in the input partition will be ordered as required by the symbol "$;\$". For example, if the operation is subtract, inputs must be classified as minuend and subtrahend. Hence, the notation,

$$I_1 \{1; 2\} SUB \{3\} \{10, 11, 12\}$$

describes an action, $I_1$, which subtracts the data in subpart 2 from the data in 1 and places the results in subpart 3. Machine subparts 10, 11, and 12 are used in the execution of the operation. Action descriptions may be truncated here when partial specification is not misleading.

A machine subpart is defined by the last action in which it appeared as an output. That is, the data in the subpart is a result of having executed the action. For the previous example, subpart 3 is defined by action $I_1$.

Graphical Description: To assist in the environmental analysis and to study the interaction of the various program parts, a directed linear graph representation is used. Nodes of the graph correspond to microactions in the program. Arcs describe the set of possible successor nodes. A path is an ordered list of nodes that correspond to some execution.
sequence. A loop is a path with the restriction that the last node and first node in the path are the same. A region is some subset of nodes and the interconnecting arcs between them (also called a subgraph).

A region is said to be strongly connected if the subgraph possesses the property that any node can be reached from any other node by traversing only arcs within the subgraph. A node is a predecessor of a node if a path exists from to . If there is an arc directly from to , is an immediate predecessor. Similarly, in the first case, is a successor of ; in the second, is an immediate successor.

A node is said to be a region entry node if there exists an arc from some node, , such that,

\[ k \in R, \quad l \notin R \quad \text{where } R \text{ is the region.} \]

Similarly, is called a region exit node if there exists a node , such that an arc exists from to and

\[ k \in R, \quad m \notin R \quad \text{for the region } R. \]

A node is called an articulation node (or essential node) for a region if there exists no path from any region entry to any region exit node that does not include . Therefore, if the region is entered, the code in an articulation node must be executed before region exit can be accomplished.

Two regions, and , are called properly nested (or simply nested) if either

\[ R \cap G = \emptyset \]

or

\[ R \cap G = R \]

or

\[ R \cap G = G. \]

That is, either one is contained in the other or they are nodewise disjoint. If , then is said to be an inner region. If some region contains no other region, then is said to be the innermost region (notice that several disjoint regions of a graph may be innermost). A method by which a graph can be analyzed and regions produced which are both properly nested and strongly connected is given in [1]. This list of regions allows separation of the program into computationally simpler portions, allowing the analysis to proceed in a piecewise fashion rather than considering the total program at once.

VI. TECHNIQUES

Having presented some sources of inefficiency and discussed the graphical tools and microprogramming notation, this section will describe the types of improvements sought by the optimizer. The conditions to trigger the improvements will be presented here, while the integration and implementation methods will be deferred to the next section.

Intermediate Text: The compilation or translation process could produce microcode in an analysis-oriented intermediate form that would facilitate inspection and subsequent manipulation beyond a sequential listing of micro-instructions. In addition, the intermediate text format provides enough detail to allow the generation of the final microinstruction text. During the translation, the code is grouped into nodes of sequentially executed actions (i.e., no imbedded branch operations) and the interconnection between the nodes is recorded. Node entry is allowed only to the first action. Node exit is allowed only at the end of the action set. Hence, if branch type operations appear in the node, they occur only at the end of the node. Several contiguous branches however may appear at the end of the node with no other type of action intervening (intervening operations would force creation of a new node). A simple flow chart appears as Fig. 2 to indicate the process of separating the actions into nodes.

As the translation is performed, the tabulation of the action table and the sequence table occurs. The intermediate text consists of the following.

1) Connectivity Matrix: For a program of nodes, this is an matrix of Boolean values. The matrix contains elements such that

\[ c_{ij} = 1 \text{ if an arc exists from node } i \text{ to } j \]

\[ c_{ij} = 0 \text{ otherwise.} \]

2) Action Table: This tabulation appears in the form

\[ \{i_1, i_2\} \text{ OP} \{j\} \{u\} \text{ aux} \]
KLEIR AND RAMAMOORTHY: OPTIMIZATION STRATEGIES FOR MICROPROGRAMS

INPUTS  OPERATION  OUTPUT  AUX
[1:2]  ADD  [3]  
[1:2]  ADD  [3]  

(a)

<table>
<thead>
<tr>
<th>INDEX</th>
<th>[1;2]</th>
<th>OP</th>
<th>[j]</th>
<th>[u]</th>
<th>aux</th>
</tr>
</thead>
<tbody>
<tr>
<td>I_1</td>
<td>[1:2]</td>
<td>ADD</td>
<td>[3]</td>
<td>[10,11,12,13]</td>
<td></td>
</tr>
<tr>
<td>I_3</td>
<td>[3:]</td>
<td>GATE</td>
<td>[5]</td>
<td>[10,13]</td>
<td></td>
</tr>
<tr>
<td>I_4</td>
<td>[5:]</td>
<td>SHIFT</td>
<td>[7]</td>
<td>[10,15,13]</td>
<td>4</td>
</tr>
<tr>
<td>I_5</td>
<td>[5:]</td>
<td>GATE</td>
<td>[6]</td>
<td>[10,13]</td>
<td></td>
</tr>
<tr>
<td>I_7</td>
<td>[3:1]</td>
<td>ADD</td>
<td>[7]</td>
<td>[10,11,12,13]</td>
<td></td>
</tr>
</tbody>
</table>

(b) (c) (d)

SEQUENCE INDEX  ILOC  FP  BP
1  I_1  2  0
2  I_2  7  1
3  I_3  4  1
4  I_4  6  3
5  I_5  6  3
6  I_6  0  5
7  I_1  8  3
8  I_7  0  7

Fig. 3. Intermediate text activity tabulations. (a) Program actions for node. (b) Action table. (c) Sequence table for node. (d) Sequence directory node entry.

where

{i_1; i_2}  Input subparts with allowance for possible ordering between left and right side inputs in noncommutative binary operations (e.g., subtraction).
OP_k  Designation of the primitive microoperation applied to the inputs.
{f}  Output subparts into which the operation result is placed.
{u}  Description of the transient machine subparts used.
aux  Auxiliary modifier to hold such data as shift counts, branch target locations, and other operation modifications that do not fit the partition input/output scheme. Its purpose is to distinguish between otherwise identical actions and to assist in the code generation.

As the microactions are generated from the translation process, the table is searched for a match to the current action description. If a match is found, the location of the entry is noted and the current duplicate description discarded; otherwise, the action description is appended to the end of the list and this position is remembered.

3) Sequence Table: The location of the action is recorded in the sequence table whose entries are of the form,

ILOC  FP  BP

where

ILOC  Index of the action in the action table.
FP  Forward mobility pointer.
BP  Backward mobility pointer.

The mobility pointers will be described later.

4) Sequence Directory: This K x 2 matrix indicates the bounds of the K nodes in the sequence table. Entries in the directory are of the form,

From  To

where

From  Pointer to the first action in the node described in the sequence table.
To  Pointer to the last action of the node in the sequence table.

5) Region List: This list is generated after the translation process is complete. It contains the properly nested strongly connected region description of the nodes in the program graph.

The intermediate text is exemplified in Figs. 3 and 4. While the amount of data required by the intermediate text appears at first glance rather imposing, it should be noted that much of the description can be represented by bit vectors using few computer words. For example, subparts of the host machine can be described using the bit vector structure.
actions.

gories
involve the
zation, action
dictable and
the
further notice that Conditions 3 through 7 were satisfied and the action in node 4 was deleted. Then execution of the path 1-2-3-4-5 would perform properly but the path 1-4-5 could cause errors.

Fig. 4. Graphical description of a program. (a) Program graph. (b) Connectivity matrix. (c) Region list.

Compiler Techniques: Since compiler object code optimization techniques already exist, it would appear prudent to modify some of these methods for the task at hand. The first involves the removal of nonessential operations. Two categories of this class are redundant actions and negated actions.

Redundant actions are those whose outputs are predictable and currently available from previous identical action executions. Notice in the example of Fig. 3 that the seventh action in the sequence is the same as the first. Further notice that the result of the seventh is the same as the first since subparts 1 and 2 are not redefined between the executions. Then, if the seventh action were deleted, the program state would not be disturbed.

Therefore, sufficient conditions for deleting a subsequent action as redundant are the following.

Condition 1: Two actions have the same inputs, operation, and output specified (i.e., there are two pointers in the sequence table to the same location in the action table).

Condition 2: No input subpart is redefined between the two identical actions (i.e., the input subpart set is not specified as an output subpart in the intervening actions between the two).

Condition 3: The output subpart data is preserved (i.e., no intervening action redefines the output subpart).

Condition 4: If one of the two actions is executed the other action is also.

Condition 4 is required to avoid removing a possible necessary action. For example, suppose one action resided in node 3 of Fig. 4 and an identical action resided in node 4. Suppose further that Conditions 1 through 3 were satisfied and the action in node 4 was deleted. Then execution of the path 1-2-3-4-5 would perform properly but the path 1-4-5 could cause errors.

Negated actions are those activities whose output results are never used. These conditions occur when the output from an action (or some part of it) appears as an output in a succeeding action without first appearing as an input. That is, the output result from the first action is lost after the execution of the second. For example, in Fig. 3(a) the results of action 4 are negated by action 8.

A special case of negated results is the removal of entire nodes. When the program graph contains a node which has no successor (predecessor) nodes and is not a final (entry) node of the process, the entire code set in the node can be deleted. Although this often indicates a programming error, the condition can occur in correctly operating routines when library functions are retrieved and only a subset of the code actually utilized.

Another valuable technique from compiler optimizers is code motion. Here, execution time is reduced by minimizing the action population of the most frequently executed portions in the program. Two basic approaches exist for determining an area’s execution frequency.

Static analysis applies the heuristic that execution frequency is dependent on nesting level in the program graph.
Code is moved from inner to outer regions under this assumption.

Dynamic analysis assigns execution frequency to areas based upon program operation. The program can be sampled (either externally or internally) during execution of a representative run to obtain a profile of activity. Alternatively, the program can be analyzed with graphical techniques by assigning probabilities to branch points and applying modeling techniques. Although a more powerful quantitative technique, dynamic analysis is usually more time consuming and particularly susceptible to errors from invalid assumptions.

For present purposes, the static approach is adopted and strongly connected nested regions are used. The activities of code motion can be summarized as: 1) structure analysis of the program graph; 2) selection of move candidate actions; and 3) hazard analysis to confirm that the action can safely be moved out of the region. Actions can be moved either forward (along region exit paths) or backward (along region entry paths). Sufficient conditions for motion out of a region are the following.

*Condition 1:* Input subparts for the action are not defined within the region. That is, the input subpart does not appear as an output for any of the actions within the region.

*Condition 2:* The action is resident in an articulation node for the region.

*Condition 3:* The output is not defined elsewhere in the region. If forward motion is desired the following holds.

*Condition 4a:* The output subpart is not used as an input within the region. If backward motion is desired the following holds.

*Condition 4b:* The action output is not used on any path from a region entry to the position of the action.

The actions removed from the region produce the same results after each execution (Condition 1). They are always executed whenever the region is entered (Condition 2) and the repositioning of the execution will not cause a partition to be erroneously defined when other actions in the region are executed (Conditions 3 and 4). The actions satisfying these conditions can then be removed from the region and executed only once upon region entry (backward motion) or upon region exit (forward motion).

*Other Techniques:* Other modifications to the program may provide reduction benefits in the microprogram that may be of little value to compilers. These benefits accrue because of the concern in microprograms with machine structure, timing, and limited operation complexity.

One of the most valuable of these is the recognition of parallel actions among those specified in the node. This analysis provides not only a list of parallel action opportunities, but also reveals the mobility range of each action within the node; that is, if two actions can be executed simultaneously, then it follows that the computation result is independent of which is executed first serially [13]. Two actions may be executed in parallel if the following conditions hold.

*Condition 1:* The output of one action is not used as an input to the other and vice versa.

---

**Fig. 5.** Reduced form of example in Fig. 3(a). (a) Redundant and negated actions removed. (b) Tomasulo's algorithm applied.

*Condition 2:* The two actions have disjoint output subparts.

*Condition 3:* The two actions do not use conflicting machine resources for execution.

Conditions 1 and 2 produce order independence on the data base while Condition 3 is required by the physical hardware structure.

Another technique is a software realization of Tomasulo's algorithm for multiple hardware units [12]. The scheme is to discover from the transitive nature of data flow when results may be fanned-out to machine subparts in parallel and to eliminate temporary storage when possible. Consider the example of Fig. 5(a). Since the GATE operation simply copies the input data to the output subpart, the information in subparts 3, 5, and 6 is the same after the fourth action is complete. If the result from the first action could be fanned-out to more than one machine subpart in parallel, the gate operations could be removed. Notice that subpart 5 is used only for temporary storage and can be deleted. Hence, the reduced set of actions in Fig. 5(b) will produce the same results. If parallel fan-out is not possible because of machine structure, a gate operation must remain, but the temporary storage use of subpart 5 has been deleted.

The process requires recognizing microoperations that simply transfer data (e.g., gate, shift by 0 and, add with 0 transfer, etc.). The action defining the transfer action's input is located. The output subpart specified by the transfer is appended to the defining action's output list, and the transfer deleted. Temporary storage is removed by simple negated analysis on the output fan-out list. If necessary, gate actions are recreated after this analysis.

Because the microprogram must consider internal hardware timing, some microactions may not follow one another within the same or adjacent microsteps. That is, some delay may be required between the execution of one action and the use of its result by some succeeding action to prevent a timing hazard. This delay is often provided by inserting a timing "pad" of other actions to insure completion of busy hardware. If the macro corresponding to the execution of the machine instruction exhausts productive activities before the required time has elapsed, a pad of no-operations is often introduced. Since translation produces a longer list of activities, further candidates for filling the timing gap appear. Replacing the timing no-ops with productive poten-
tially parallel code would reduce the processor idle time and improve the program execution time. Sufficient conditions for moving a new action in to fill the timing gap are the following.

1) The productive action is mobile (by code motion constraints) from its current position to the no-op location in the program.

2) Removing the productive action will not create a new timing hazard in its old location.

3) Placing the productive operation in the timing gap will not create a new timing hazard in the new location.

Another technique of resolving timing problems is a REQUEST/REPLY type interface for periodic activities that require more than one microstep. Here, the activity is initiated for the slower device (e.g., main memory) and before the result is used, an idle loop is executed awaiting the device completion signal. If sufficient other microactivity is introduced between the initiation of the device and the check for completion, one iteration of the idle loop will confirm device completion eliminating the time wasted in nonproductive spinning.

If the cycle time is known to be \( T \) for the periodic device and the microstep time is \( t \), then providing \([T/t]\) instructions between initiation and the completion check will minimize the idle loop iterations.

Physically relocating part of the execution code in memory may allow streamlining frequently traversed arcs in the program graph, reducing the average execution time. It should be noted that the optimal placement of code may not be obvious if the branch structure and input data environment are complex. One approach is to determine which arcs are most frequently traversed and streamline those paths by rearranging the branch action clusters at the end of nodes. If “fall through” types of branches (i.e., where the next sequential instruction is a branch target address) are used, the optimal ordering of branches may require a specific placement of code modules in memory and hence produce a significantly different program structure. Details of the technique to reorder the branches and determine the memory load configuration are presented in the Appendix.

VII. IMPLEMENTATION OF OPTIMIZATION STRATEGIES

Clearly, not all machine subparts (e.g., all main memory words) can be enumerated in a bit vector for analyzing each action. Only those subparts used most frequently are assigned bits to describe the actions. If mass memory is considered as a one-dimensional array, only the array name (i.e., which memory) and the index to the proper datum (i.e., address register) are required. Then two bits for each memory are used in the vector to indicate memory usage.

It is noteworthy that with each transformation of code to another form, operational identities become more obscure. For example, a memory address identity that is perfectly obvious from symbolic label references requires examination of base registers, program relative address, indexing, and indirect addressing in machine language versions. The analysis is even more complex on the microprogram level. Hence, some optimization should probably be performed on each level of code. Optimization at each level would reduce those inefficiencies that become obscure at the next level.

Rather than apply each of the techniques separately, the methods should be integrated in the optimizer and share as much analysis as possible. The application sequence can also influence the optimization effectiveness since there is a cascade effect in improvements. That is, one improvement may make another one possible. In the example of Fig. 3, removing the negated shift operation reduced machine partition 5 to a temporary storage use. Furthermore, moving one action out of a region may permit other actions to be moved.

Some techniques are applied on a nodal basis while others require regional investigation. The nodal analysis (considering the local environment of action relationships only within the node) simplifies analysis procedures. Regional analysis requires studying predecessor and successor interactions and path analysis. This investigation can become quite complex if feedback arcs, multiple successor/predecessor nodes and self loops are involved. Hence, reduction efforts across node boundaries are considerably more costly than simply reducing the node.

Because regional considerations require more computation effort, many reductions are performed only upon the linear execution sequences within the node. Results from redundant, negated, and parallel analysis as well as Tomasulo’s algorithm are restricted to nodal boundaries. More global reductions associated with these improvements can occur if actions are moved out of several region nodes and reduced in their new positions.

The mobility pointers in the sequence table (FP and BP) are used for a variety of purposes. The entries hold an index pointer to sequence table entries. The action pointed to in the sequence table is the first adjacent action that violates parallel execution data constraints (Conditions 1 and 2). The forward mobility pointer (FP) indicates the first nonparallel action encountered when examining successor actions. Similarly, the backward mobility pointer (BP) indicates the constraint in the predecessor actions. If an action is unconstrained in one of the directions, the appropriate entry is set to zero (see Fig. 3). The span of the two pointers indicates actions that are execution sequence independent.

The mobility pointers are used to locate negated actions, confirm redundant actions, implement Tomasulo’s algorithm, and locate candidates for filling timing gaps or for regional motion. Negated effects are detected by examining the cause for forward direction constraint. If an output conflict occurred (Condition 2) before an input use constraint (Condition 1) is found, the output subpart in conflict is negated.

Tomasulo’s algorithm requires examining the mobility pointers of each transfer type action. The backward mobility pointer indicates the starting place of the search for the input defining action. The forward pointer starts the search for an action using the result. If the forward search detects an output conflict, negated effects remove the transfer instruction. If the forward search locates an input using

\[ \text{The operation } [k] \text{ produces the integer least upper bound on } k. \]
the transferred result, the operation type is examined. If the operation is another transfer, a temporary storage partition may exist; otherwise, only parallel fan-out produces a reduction.

As previously mentioned, redundant operations are recognized by identical pointers to the action table. If two identical actions are discovered in the same node, the mobility pointers are used to reduce the search time for satisfying other redundancy conditions. Let FP1 be the forward pointer of the first action and BP2 be the backward pointer of the later action. If the two actions point to each other, they are obviously redundant. Otherwise, the actions between FP1 and BP2 are examined for defining a subpart of the candidate action. If this does not occur, the actions are redundant.

Code motion demands regional considerations. Strongly connected nested regions in the program graph are used for code motion. An inner to outer inspection strategy will allow expelled code to possibly propagate out of several nesting levels. All nodes considered when analyzing inner regions can be ignored for later outer regions. If actions in these inner nodes cannot move out of the inner area, their movement out of the outer area is clearly impossible. Hence, as analysis of a region is completed, it can be shrunk to a “pseudonode” representation for outer region analysis. The pseudonode is assigned the composite character of its constituent nodes.

Candidates for code motion are selected from the node contents using the conditions of: 1) residence in an articulation node not contained in an inner region, 2) inputs not defined in the region, and 3) not confined by actions within the node. The search for candidates occurs from region entry toward region exit when moving actions backward out of regions, and in the opposite direction when considering forward motion. This order of analysis allows actions to be moved that were previously constrained by another mobile action (cascade effect in code motion). It should be noted that branch type actions are not moved from their nodes. This motion would alter the program graph.

It is of course necessary to supply a destination for the displaced code. Here the alternatives are creating a new node or placing the code in an existing node. Suppose some action is displaced backward out of region R1 in Fig. 4(a) and placed at the end of node 1. Then if region R2 is entered more frequently than R1, node 1 may have a higher execution frequency than any node in R1. Hence, the execution time would be degraded rather than improved. If terminal nodes receptacles for displaced code were inserted for each region entry and exit, this degradation would be prevented. Thus, each region is appended with an initial node through which all region entries are made and a final node for all regions exists. Fig. 6 illustrates this modification for the program graph of Fig. 4.

In general, terminal nodes are considered simply as another program node during region analysis. Nonessential actions are removed from terminal nodes when the outer region is considered. The terminal node actions are considered as candidates for further motion thus allowing actions to “bubble” out of several regions.

When the outermost region has been processed, the terminal nodes are examined. If they are empty, the nodes are deleted; nonempty terminal nodes require more investigation.

If the node has a single exit arc, it can be treated as another node in the program. If multiple exit arcs exist, tradeoffs must be examined. Consider the example of node 3' in Fig. 6. Two copies of the actions could be inserted in the paths from 1 to 4 and 2 to 3 but the program size would increase. Alternatively, additional actions could be encoded to perform a selective branch to either 2 or 3 after displaced actions are executed depending upon whether entry was from node 1 or node 2. This would introduce additional execution time overhead as well as increase the program size. Lastly, the actions could be reinstated in the region. Notice that this is not identical to not having moved any actions at all, since some nonessential activity may have been removed and other actions moved further. This approach is selected for implementation.

After completing the terminal node analysis, nodes are ordered for code generation using the technique presented in the Appendix. The result is the production of a node order vector directing final microinstruction text generation from the intermediate text.

Timing hazards, parallel action merging, and smoothing periodic functions must necessarily be deferred until final code generation unless some simple relation exists between actions and the number of microinstructions required.

---

2 A strict ordering exists among articulation nodes on a distance basis from entry to exit of the region.
Parallel operations are merged on a nodal basis using the mobility pointers to determine data dependencies among the actions. The action table entry detailing transient subpart use will further limit parallelism by hardware constraints.

Periodic activities using a REQUEST/REPLY interface are smoothed by forcing the initiation action to occur at as near the backward mobility limit as possible and forcing the confirmation check as far forward as possible. This provides the maximum spread for the particular program environment.

A complete analysis of timing hazards requires examination of successor nodes activities. An alternative to complete inspection is to remove timing hazards that cannot be covered by node actions with a no-op pad at the node end. If the number of hazardous conditions is small, this approach may suffice. If numerous hazards are possible, nodal analysis is not likely to compress the code much beyond the original representation.

To examine the feasibility and value of the techniques described, an example system is currently being studied. The host machine is a simulated adaptation of that proposed by Cook and Flynn [11]. The optimizer presently restricts the microprogram to the following: 1) graph structure—up to 60 nodes; 2) action mix—less than 100 different actions; and 3) program size—less than 500 actions. Bounds are arbitrarily selected for this study and can be expanded fairly easily. The system allows comparison of optimized and unoptimized versions of the same routine. The various techniques can be selectively disabled to analyze the affects of each and the interaction between various modifications.

The system, including symbolic microassembler and machine simulation, is coded in four linkages in Fortran IV running on the CDC 6600 in 30K of memory. Unfortunately, preliminary results are not declarative enough to present here.

VIII. CONCLUSION

The collection of techniques presented here are by no means complete. Clearly some beneficial machine-dependent techniques can be included and some proposed techniques may be unworkable on some machine structures. Rather the purpose of this exposition is to provide an approach, insight, and historical perspective for techniques that are available. It should be obvious that all inefficiencies cannot be removed by the analysis proposed. There is certainly no substitute for good microprogramming. Optimization, however, can be used as a tool to indicate where improvements are most desirable and provide the most benefit.

It is important that the program be informed of changes made by the optimizer both for tutorial purposes and to avoid the introduction of program logic hazards. The analysis techniques can further be used to aid debugging by locating logical flaws.

One of the lamentable features of automated optimization is the investment of computation effort without any assurance of subsequent performance improvement. The optimization effort is strongly weighted towards analysis with very little effort expended in code improvement. Thus it is expedient to prescribe optimization only for well-debugged routines with long life expectancy; however, these qualities are somewhat elusive. Perhaps a more beneficial approach would be to investigate program characteristics which statistically suggest improvement yields [6]. Little effort has been directed toward this task to date. Another approach would be partial improvement directed at program areas which offer the greatest return. Here the aspects of profile analysis offer great potential. Unfortunately, techniques for obtaining the desired information remain fairly crude and merit further investigation.

Although a fairly coherent design can be formulated with the techniques presented, some areas merit further study. Their resolution will influence microlanguage design, machine architecture, and compiler/translator conventions. Among the areas of further interest are the following.

1) Context Sensitive Operations: In some microprogrammable machines, operations and instruction formats are restructured with a particular bit or mode setting by a previous operation. For example, the IBM 360/50 interprets microinstructions differently depending upon whether the machine is the CPU or I/O mode. Since some microoperations in the CPU mode are not allowed in I/O mode program area problems can develop with code motion. Some mechanism is needed to describe the program areas depending upon the mode setting. Perhaps the process can be viewed as two cooperating processes rather than a single program.

2) Equivalent Operations: Better techniques are required to manipulate logically equivalent but physically different subparts. For example, if two main memory ports are available, they are logically equivalent but manipulation code is not identical. Therefore, it is necessary to adopt a convention using one of the equivalent sets as a preferred usage and postpone the distributed assignment of tasks until the final text representation is produced. This process becomes even more complex when some items are conditionally equivalent. For example, two registers may be used interchangeably in all arithmetic operations but one is directly tied to the memory port as an address register.

3) Subroutines: The use of subroutine structures presents graphical representation problems. Subroutines can be represented graphically by: a) providing a region structure entered by all nodes calling the subroutine and with exits to all immediate successors of the calling routine; or b) inserting a region structure for each subroutine call. The first alternative is not really representative of the program. From the graph it appears possible to enter the subroutine from one node and exit to the return node for a different call. Clearly this is not indicative of the program's operation. The second alternative is equally misleading. Here, the subroutine becomes a macro and optimization is allowed on each copy independently. If Tomasulo's algorithm is applied between nodes, the linkage mechanism may be deleted.
as temporary storage partitions. Hence, either a new graphical structure must be developed, or the subroutine could be treated as a macro for investigation and only modifications universally agreed upon among all subroutine calls actually performed.

4) Machine Design Consideration: If optimization is expected to play a significant role in the utilization of the machine, it would appear prudent to investigate how machine structure aids or inhibits improvements. This information would be used by the machine designers when selecting alternative approaches for realizing the machine.

5) Microcompilers: If the microprogram level is intended for user description of custom routines, it will probably be necessary to provide higher level descriptions of the process. Surely the user will be as dissatisfied with coding at the microassembly level just as he was with the machine language assembly level. The optimizer should integrate easily with the compiler and cooperate on conventions adopted. Further, much of the information lost in the translation (symbol table mapping to instruction location and possible targets for indexed and indirect references) can be passed from the compiler to the optimizer, thereby reducing the analysis effort and decreasing the missed optimizations by resolving ambiguities.

APPENDIX

ALGORITHM TO SELECT NODE ORDER IN MEMORY

The program loading process is a mapping of the two-dimensional program graph into a one-dimensional memory space. If any of the program nodes has multiple successors, alternatives exist in the map selection. To reduce the routine’s execution time, the map should be selected that minimizes the average or expected time in traversing the graph’s arcs (performing branch operations).

Assume that each arc can be assigned a probability for leaving the node. This may be either specified by the programmer, derived from execution data, or implied from modeling or simulation.

Let the program graph, G, contain q nodes denoted ni, $i = 1, q$. Let the probability of a transition from $n_i$ to $n_j$ be $p_{ij}$. If no arc exists from $n_i$ to $n_j$ then $p_{ij} = 0$. Thus,

$$
\sum_j p_{ij} = 1.
$$

Let $f_i$ be the entrance frequency for node $n_i$. Further let the transition matrix $T$ have elements $t_{ij} = p_{ij}f_i$. Then the entries in $T$ can be related one to another; if $t_{ij} > t_{ik}$ then the arc from $n_i$ to $n_j$ is traversed more frequently than the one from $n_i$ to $n_k$.

Notice that $T$ can be derived from the probability specification alone since,

$$
f_i = \sum_j p_{ij}f_j.
$$

Assume we begin with a set of initial constraints on the node placement (e.g., the program entry node is first and program exit node last). Then we scan the entries in $T$ and select the largest element. If several nodes have equally large elements, the tie is arbitrarily resolved.

After selecting the node, the branch action description at the end of the node is examined and alterations made to allow the branch with the highest transition entry to execute fastest. If this requires a specific order of nodes in memory, the constraint is recorded. The alternative selected for this branch action is indicated and frozen.

The selected entry in $T$ is set to zero and a search performed again for the largest element. This process continues until the $T$ matrix contains only zeros. The constraints are examined and the order determined using the initial constraint.

To illustrate, consider a machine in which the branch instructions can take the following forms.

1) A condition branch $B(c_{ij})$ where $c_{ij}$ denotes the condition for which the transition from $n_i$ to $n_j$ occurs. If the condition is not satisfied, the next instruction is executed.

2) An unconditional branch $B_j$ where $j$ denotes the target location, $n_j$.

3) A “fallthrough” branch in which the transition from one node to another is performed by failure of all previous conditional branches.

For a node which has $m$ mutually exclusive successors, a branch cluster of at least $m−1$ branch instructions is required. Then the speed of traversing an arc can be computed as,

$$
s = (q \times b_f) + b_o
$$

where

$q$ Number of fallthrough instructions executed to reach the branch instruction,

$b_f$ Execution speed of the fallthrough,

$b_o$ Speed of executing the object branch.

Let us further suppose that the execution time for a conditional branch is asymmetrical. That is the execution time for the branch instruction if the condition is true is not the same as if it is false. Assume the execution times to be

$$
B(c_{ij}) = 3 \text{ units if condition true}
$$

$$
B(c_{ij}) = 2 \text{ units if condition false}
$$

$$
B_j = 2 \text{ units.}
$$

Consider the program graph of Fig. 7(a) where the arc numbers correspond to branch probabilities. Then the transition matrix can be derived as shown in Fig. 7(c) by solving the node entrance equations of Fig. 7(b). Suppose we constrain node 1 to be the first in memory order. Applying the method, we first select node 4 since the entry $t_{44}$ is largest. Alternative structures to realize the transition are the following.

Structure 1: Conditional branch to 4.

Structure 2: A conditional branch to 3 followed by an unconditional branch to 4.

Structure 1 requires 3 units of time, while structure 2 requires 5 units; structure 1 is selected. This imposes no constraint on the node structure.

The next largest entry in $T$ is the $t_{43}$ element. Possible branch realizations include: 1) conditional branch from 4 to 3: 5 units (includes time for failure of $B(c_{44})$); 2) unconditional branch from 4 to 3: 4 units; and 3) fall through branch from 4 to 3: 2 units. Then the third alternative is selected. Notice however, that this requires that node 3
follow node 4. This is feasible since 3 and 4 are as yet unconstrained.

In an identical fashion, \( f_{34} \) produces a conditional branch \( B(c_{34}) \) and \( t_{32} \), a fallthrough from 3 to 2 thus constraining node 2 to follow 3.

Next, node 2 is selected. Notice that all \( T \) entries for 2 are the same. Hence, alternatives must be selected for each to determine the fastest branch configuration. The smallest execution time for each is 3 units with no constraints on the node order. The next smallest for each is considered and the speed tie perpetuated until the fourth smallest alternative is examined. The fourth conditions for each of these are the following:

\[ f_{1} = \frac{1}{4} f_{2} \]
\[ f_{2} = \frac{1}{3} f_{4} + \frac{1}{4} f_{2} + \frac{1}{3} f_{3} \]
\[ f_{3} = \frac{1}{4} f_{2} + \frac{1}{3} f_{4} \]
\[ f_{4} = \frac{2}{3} f_{1} + \frac{1}{4} f_{2} + \frac{2}{3} f_{3} + \frac{2}{3} f_{4} \]

(b)

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
1 & 0 & 1/3 & 0 & 2/3 \\
2 & 1 & 1 & 1 & \\
3 & 0 & 8/3 & 0 & 16/3 \\
4 & 0 & 0 & 7 & 14 \\
\end{array}
\]

(c)

Recall that node 1 is required to be the first node by the initial constraints and cannot follow any other node. Node 3 is required to follow 4 and 2 to follow 3 from a previous result. Thus, only \( t_{22} \) is feasible under the present constraints. Since four branches have been examined thus far, we can now configure that branch cluster. The branch for \( t_{22} \) must be realized with the fourth largest alternative; other branches are arbitrarily selected from the three previous tie realizations.

Finally, node 1 is selected and the branch \( t_{14} \) realized with a fallthrough requiring node 4 to follow 1. Entry \( t_{12} \) is realized with a conditional branch. The final mapping of the nodes into memory and the branch clusters are shown in Fig. 8.

**REFERENCES**


