APPLICATION OF MICROCODE TO IMPROVE PROGRAM PERFORMANCE

An Honors Thesis (ID 499)

by

Dawn D. Greenwood

Thesis Director

Clinton P. Fuelling

Ball State University

Muncie, Indiana

May 18, 1981

Note: Spring 1981
CONTENTS

INTRODUCTION ................................................. 1

MICROPROGRAMMING OPERATING SYSTEMS. ...................... 1

ARCHITECTURE REDEFINITION VIA MICROPROGRAMMING. .......... 2

Manual Tuning .............................................. 3

Automated Process ......................................... 9
  Frequency of Sequential Occurrence .................... 9
  Frequency of Execution .................................. 10
  Combination Methods .................................... 11

OPTIMIZATION ............................................. 13

Vertical Optimization ...................................... 13
  Removal of Nonessential Operations .................. 13
    Redundant Actions .................................. 13
    Negated Actions .................................... 14
  Code Motion .......................................... 14
  Code Consolidation .................................... 14

Horizontal Optimization .................................. 15
  Local Compaction ...................................... 15
  Compaction Algorithms ................................ 17
    Linear Algorithm .................................. 18
    Critical Path Algorithm ............................ 18
    Branch and Bound Algorithm ......................... 19
    List Scheduling .................................... 20

Global Optimization ...................................... 21

Register Allocation ...................................... 21
INTRODUCTION

The use of microcoding to improve program performance has introduced new areas of study in recent years. Microcoding operating systems and architecture redefinition via microprogramming are two such areas. High-level microprogramming languages are also commanding great interest and research. All these techniques require optimization algorithms to assure optimal execution times. These areas will be considered in this paper.

Due to the difficulty in locating resources, many articles were not available for review. In this case, care has been taken in guiding the reader to the article by referencing it when appropriate. A number of books and articles in the bibliography offer a basic look at microprogramming in general as well as the subjects covered in this paper [AGRA74, AGRA76, BROA74, CHUR70, DAVI78, FULL76, HIGB78, HINS72, JONE76a, JONE76b, KRAF81, LEVE77, RAUS80, ROSI74, SALI76].

MICROPROGRAMMING OPERATING SYSTEMS

Since "users seldom (if ever) perform a computation without assistance from the operating system," [DENN71] any improvements in execution rates of most frequently used segments should reflect favorably on application programs. Microcoding primitives used throughout the operating system is one approach to improving operating system time. Larger segments can also be selected. Choosing the most appropriate segments is important to performance improvements [RAUS80, BROW77]. In [BROW76] is a list of fourteen functions which should be
considered for microcoding. These functions tend to be too exact for hardware, relatively consistent, and noticeably affected by software overhead. Several applications in the area of software have been discussed [HUBE70, BELG76, DHAU77, CHAT76, LUND76, DENN71, STOC77, SOCK75].

ARCHITECTURE REDEFINITION VIA MICROPROGRAMMING [RAUS76]

Called "tuning" by [ABDA74, ABDA73], the central concept is to change a system's architecture to more efficiently solve a particular problem. Architecture in this sense refers to "the attributes of a computer as seen by the programmer" [RAUS76]. This includes the memory and particularly the machine instruction set which the programmer interacts with. The microprograms in control store which interpret the machine instructions define a particular architecture. By modifying these microprograms, the architecture changes accordingly.

Such modification is necessary for several reasons. Typical instruction set design has followed hardware constraints instead of considering the problems to be solved and their structure. The results are instruction sets which are awkward and inefficient [RAUS76, WADE75]. In addition, the general-purpose architecture was conceived through a series of compromises that permit it to perform the widest range of services. Like the jack-of-all-trades, it is frequently master of none and handles all its functions in a suboptimal fashion [HUSS70].

To counteract these compromises and tune the instruction set, carefully chosen segments of machine code are microcoded, placed in control
storage and referenced by one new machine language instruction. The major advantage of this approach is the decrease in machine instruction fetch-and-decode time. Considering the average (FORTRAN) program spends forty percent of its execution time on this function, it is estimated that savings can easily be twenty percent [RAUS76]. Additionally, the microcode can be optimized. It is important to stress that optimization strategies are vital to architecture redefinition success [ABDA74]. The final section covers optimization strategies.

Isolation of the appropriate segments can be a manual process based on environmentally determined areas of processing inefficiency. An automated process can be based on the frequency of sequential occurrence in an environment, the frequency of execution, or a combination thereof. [RAUS76]

MANUAL TUNING

Manual tuning is the creation of specialized microcoded instructions when the code is written by a programmer directly or in some microprogramming language. [ABDA74] applied the term to tuning efforts prior to when writable control store made automated tuning feasible. It is no less applicable to vendor packages and installation-written code designed to "speed up" a particular feature or activity.

Those activities most likely to benefit by microprogramming are CPU bound, use a number of intermediate results, are highly repetitive, or are not easily handled by existing machine language instructions [CLAP72]. Such features include multiply routines, square root algorithms, matrix operations, table searches, array address calculation,
number generation, stacking routines, and tree searches [COOK70, HUSS70, TUCK71, CLAP72, REIG72, REIG73, PARK73, SNYD75, HABI74, TAFR75].

In addition, applying manual tuning concepts when an instruction set is being designed will assure an application oriented architecture [WADE72, WADE73, HARR73, WADE75]. [WADE75] capitalized on the similarity of languages' constructs to design a "general-purpose" architecture capable of supporting a number of widely-accepted programming languages. Wade's basic instruction set included simple arithmetic, compare, branch, logic, and load/store instructions. In addition to this "kernel" [RAUS76] language, Wade designed a set of "super instructions" to efficiently execute five common high-level constructs. These operations include arrays and array indexing, repetition loops, block structure housekeeping, input/output editing, and character string manipulation. Each feature will be examined briefly, with somewhat more detail on array indexing to allow for better understanding of the process of developing application oriented instructions.

ARRAYS AND ARRAY INDEXING

Wade designed a new machine language instruction ADGEN. This instruction enables a microprogram which was specially written and placed in control storage by Wade to calculate the address of an array element and use that location in one of several ways. A high-level language version of the algorithm used to calculate the address is given below.
SUM = ∅

For i = 1 step 1 until n do
begin
  sum = sum * (upper-bound [i] - lower bound [i] + 1)
  sum = sum + (subscript + [i] - lower bound [i])
end

sum = sum * element - size
computed - address = array - address + sum

The microcoded algorithm uses information stored in the instruction and the array descriptor. These formats are given below.

ARRAY DESCRIPTOR FORMAT

n# array # lower-bound - 1# upper-bound - 1# ... # lower-bound - n# upper-bound - n# element size (in bytes)

where:

n = number of dimensions of array
array = address of array in memory
INSTRUCTION FORMAT

ADGEN$5\$ address of array descriptor $b$ subscript $-1\$...

subscript $-n\$ flags $b$ index register

where:

$5\$ = instruction operation code

flags = indicate action to be performed with location

index register = received calculated address

The algorithm is designed for arrays stored in row order internally. For those stored by columns (FORTRAN), the subscript fields are interchanged.

Repetition Loops (DO- or FOR-loops)

Observing that such loops can vary in intricacy, Wade designed four different instructions. Two instructions handle the simplier cases requiring little more than an increment-compare-branch sequence, while the remaining instructions were used for the more complicated loops.

Block Structure

Block structures generate overhead due to the housekeeping involved in the dynamic manipulation of storage necessary for nested BEGIN blocks and procedures CALL/RETURN statements. When handled by inappropriate instruction sets, that overhead becomes excessive. To implement these
functions in microcode, BEGIN Block and END Block instructions, two procedure calls, and a single RETURN statement were designed.

**Input/Output Editing**

I/O editing refers to the conversion of decimal character numbers into binary form for internal storage and manipulation and the conversion back to decimal numbers for external use. The instruction CONVERT TO DECIMAL converts one binary number to its decimal representation. Input instructions were provided for conversion to binary form.

**Character String Manipulation**

Wade's goal was to provide for the majority of the manipulation features of PL/1. A string descriptor similar to that of arrays provided direct and indirect addressing capabilities. This simplifies usage of varying-length strings. MOVE and COMPARE statements were designed to best handle these features.

**PERFORMANCE RESULTS**

Wade's improved machine language instruction set was implemented and its performance compared to that of the IBM 370 architecture. A number of PL/1 and FORTRAN programs were executed on both machines. The following chart lists the execution time improvements realized by the high-level language oriented architecture over the IBM general purpose architecture.
<table>
<thead>
<tr>
<th>Program Function</th>
<th>Percent Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finding prime numbers by sieve of Eratosthenes</td>
<td>32.2%</td>
</tr>
<tr>
<td>Generates random sentences from English grammar</td>
<td>64.2%</td>
</tr>
<tr>
<td>Convert arithmetic expression from infix to postfix notation</td>
<td>87.4%</td>
</tr>
<tr>
<td>Adds record to linked list</td>
<td>54.2%  27.9%</td>
</tr>
<tr>
<td>Solving differential equation by fourth order</td>
<td></td>
</tr>
<tr>
<td>Runge-Kutta method</td>
<td>51.7%  50.5%</td>
</tr>
<tr>
<td>Multiplication of two matrices</td>
<td>72.3%  31.4%</td>
</tr>
<tr>
<td>Evaluation of a determinant</td>
<td>85.9%</td>
</tr>
</tbody>
</table>

In general, those programs performing non-numeric processing showed greater performance improvements. This is in keeping with the findings of Hans Jeans [JEAN65]. Jeans found that arithmetic operations were hardware bound and that the decrease in instruction fetch and execute time was less significant than in logical operations.
With the increased use of the writable control store, thought turned to automating the manual process of tuning. In this way each group of similar programs, or each program, can have its own machine architecture with instructions chosen to assure its optimal execution [ELAY77]. This architecture is dynamically loaded into control store prior to program execution. When a unique architecture is created for each program, the process is called dynamic problem oriented redefinition of computer architecture via microprogramming [RAUS76, RAUS75, RAUS78, RAUS80]. When similar programs are grouped together and an improved instruction set created for each group, this is called heuristic tuning [ABDA74].

Several methodologies exist for determining which groups of instructions should be replaced by new machine instructions to yield the greatest execution gains with minimal control store requirements. (Because of its expense, control store must be considered a limited resource. If this were not the case, entire programs could be microcoded.) These include the frequency of sequential occurrence and the frequency of execution. Heuristic tuning [ABDA74] and [RAUS76]'s dynamic redefinition of architecture enlist both methods in an attempt to achieve optimal performance improvement.

Frequency of Sequential Occurrence

This method is based on the observation that instructions charact--

ristically occur in identical sequences. This is found not only on the machine instruction level, but also on the intermediate level generated
by high-level language compilers. The intermediate language code is analyzed to locate such sequences and to determine how frequently they occur. From this information, sequences which would yield the greatest execution time savings if microcoded are selected for new machine instructions. The difficulty here is the length of sequences to be chosen. Longer sequences occur less frequently, while shorter sequences occur more often.

While execution time improvements are achieved through this process, they are not optimal. Typically, no attempt is made to determine relationships between sequences. Additionally, run time information is not available. There is no way to determine if some sequences are executed repeatedly as in a loop. [RAUS76]

Frequency of Execution

This method uses the execution behavior of a program to choose the instructions to be microcoded. The program is analyzed into program blocks. These are straight-line sections of code such that if the first operation in the block is performed, all succeeding code is executed. By knowing the number of times the block is executed and the execution time improvement of a single microprogrammed block over that of the corresponding machine instructions, total execution time savings can be calculated. Those blocks with the greatest savings are chosen as new machine language instructions.
Combination Methods

Rauscher and Agrawala [RAUS76, RAUS75, RAUS78] devised a dynamic redefinition scheme combining the previous two methods. Program sequences are considered for microcoding on the basis of both their frequency of occurrence and their frequency of execution.

The intermediate representation from the high-level language compiler is analyzed to determine the different instruction sequences and where they occur in the program. The expected savings from microcoding each segment are calculated. Estimates are made on how often each segment will be executed. With this additional information, total projected time saved can be calculated. Those instruction sequences with the highest potential time savings are microcoded as new instructions. Experimentally, execution time improvements were found to be between twenty-five and fifty percent.

In heuristic tuning [ABDA74], programs are classified according to their application, language and/or special requirements. A trace file for each application class is built by monitoring the execution of numerous programs from that class. This data may be collected through hardware, software or microprogrammed means. The trace file data is then processed to obtain statistics on instruction execution frequencies, instruction dependencies and similar information. These statistics make it possible for a synthesis algorithm to modify existing microcode or create new microprograms which will allow optimal execution of class programs.
The first step performed by the algorithm is the location of program loops using trace file information. Within the most frequently executed loop, an optimization step takes place. In it, arrangements are made so that the most frequently accessed data is preloaded into internal registers. Next, the instructions within the loop are decomposed into micro-operations, optimized, and loaded into control store. The trace file is updated and an estimate of the performance improvement is made. This algorithm repeats, each time choosing the loop to be tuned from those remaining. It terminates when the algorithm determines that any further execution improvements could not be greater than the overhead they generate.

If the algorithm determines there are either no loops or very long loops, synthesis of new instructions can still take place. The trace file supplies the most frequently occurring code sequences and an algorithm similar to the one used above combines the dependent code into new instructions.

The new instruction set is tested to determine that it is functionally equivalent to the general instruction set. Once its results can be trusted, the new architecture is ready to be incorporated into the existing system. Several software modifications must be made.

Two programs were heuristically tuned and their original execution compared with that using the synthesized instruction. A data movement example resulted in a 87.4% improvement in execution time. A Fibonacci number generation program achieved a 77% improvement [ABDA74]. It should be emphasized that these examples reflect specific algorithms.
More complicated application programs would probably achieve somewhat less of an improvement.

OPTIMIZATION

Optimization has several meanings depending on what concepts the particular author wants to emphasize. While optimization can refer to decreasing the number of bits in a microinstruction, this material will concentrate on methods of reducing the execution time by decreasing the number of microinstructions in a microprogram [MALL78]. There are two categories of algorithms which optimize microcode. Vertical optimization deals with decreasing the number of microoperations in a sequence of microoperations. Horizontal or microcode optimization, also known as compaction, is the process of creating microinstructions in a machine whose control word allows a great deal of parallelism. This means a large number of hardware devices can perform concurrently, and therefore a number of microoperations can execute simultaneously. Compaction can be performed locally within straight-line sequences or globally throughout the program. Additionally, register allocation is discussed because of its relationship to program performance.

Vertical optimization

Until recently, microcode was painstakingly optimized by hand. When automation of the process was considered, the natural route was to modify processes that were used in standard compilers on predominantly sequential code [ALLE69, ALLE72]. Kleir and Ramamooty [KLEI71] were
the first to consider vertical optimization algorithms for microcode as an alternative to hand optimization.

One method of optimization borrowed from traditional compilers is removal of nonessential operations such as redundant actions and negated actions. Redundant actions are those available from a previous operation which used identical input and identical destination of output. Additionally, the output destination must have remained unchanged, and the execution of one action must guarantee execution of the other to assure consistent performance. Negated actions are those from which the output is never used.

Code motion is the act of decreasing the number of operations performed in those segments of the program most frequently executed. Using program activity as a guide, dynamic analysis rates areas as to their execution frequency to determine proper code movement. Unfortunately, dynamic analysis is prone to suffer from false assumption and extensive time usage. Code motion without the benefit of execution statistics is called static analysis. It is assumed that the level of nesting indicates frequency of execution. Actions are migrated from inner to outer segments [KLEI71].

Code consolidation is another possible vertical optimization. It is feasible on certain machines that a sequence of simpler instructions can be replaced by one complex instruction. This concept has been used in the PL/MP compiler [TAN78].

The PL/MP compiler takes a high-level language and by applying both machine dependent and machine independent optimizations, produces
microinstructions. Similar compilers and the corresponding high-level languages are discussed in a number of articles [ECKH71, TIRR73, BLAI74, BOND74, CLAR72, DASG78b, DASG80, LEWI79, DEWI76a, SCHR74, FRIE76, DEWI76b, LLOY74b, RAMA74, TSUC72].

The intermediate representation of the compiler is a stream of primitive register-to-register operations. After other vertical optimizations are applied, the code is searched for sequences which satisfy predefined substitution rules. As an example:

- an add-immediate instruction: \( \text{ADDIM } b, @b, 100 \)
- a load indirect: \( \text{LOADI } y, b \)
- and an add instruction: \( \text{ADD } z, x, y \)

...can be performed by one instruction

- add-indirect-with-offset: \( \text{ADDIO } z, x, @b, 100 \)

The substitution would be made only if the first three instructions could be deleted. If a subsequent operation uses \( y \), this is not the case.

Horizontal Optimization

Local Compaction

Local compaction has been the subject of much recent study [YAU74, AGER76, LLOY74a, RAMA73, RAMA74, DASG76a, DASG76b, JACK74, TSUC74, TSUC76, DEWI76a, TABA74, TOKO77, MALL78, WOOD78, WOOD79, FISH79].
Because of the stringent performance expectations placed on microcode, earlier study was directed toward producing optimal code. The amount of work this requires is prohibitive since all possible mappings of microoperations to microinstructions are found [ASTO71]. The time required for such exhaustive work increases exponentially with the number of microoperations [LAND80]. Methods have been developed more recently [TOKO77, MALL78, WOOD79, FISH79] which result in optimal or near-optimal code in polynomial or linear time. This breakthrough makes horizontal microcode compilers feasible.

Local compaction algorithms typically require two inputs: a straight-line microcode section which contains no internal branches or entry points; and some representation of the relationship between the microoperations. To maintain data integrity, it is vital that the original semantics of the sequential microcode be preserved. This obviously requires that certain microoperations be executed in a particular order. Such microoperations are said to have a data interaction. Given two sequential microoperations a and b, the following three conditions define data interaction:

1. b requires an output resource of a as an input
2. a requires an input source which b modifies
3. a and b modify the same output resource [LAND80].

These interactions must be analyzed and represented so that the information can be used to create microinstructions. Landskov [LAND80] presented a general algorithm to record these data interactions in
graphical form. Each microoperation is added to the graph in a sequential order and is represented by a node. The new node must be connected to previously placed nodes to indicate the data interactions. It is linked only to those nodes on which it is directly data dependent. This implies that nodes further up the path from the lowest node having a data interaction with the new node are not also linked to it. The search first checks the last node on each path (the leaves) for data interaction. If the interaction is present the nodes are linked. Otherwise, testing takes place on each preceding node working up from the leaves until the test is positive or a node is reached which is already indirectly linked with the new node. Once all microoperations are added to the graph, the resulting tree indicates the sequencing necessary to assure semantical equivalency.

Compaction Algorithms

Landskov [LAND80] suggests four categories for compaction algorithms. These include linear analysis, critical path, branch and bound, and list scheduling. Each algorithm consists primarily of two parts. First, a data dependency analysis orders the microoperations so as to assure the semantics are not compromised when microinstructions are created. Secondly, a conflict analysis monitors the assignment of microoperations to particular microinstructions to assure there are no hardware conflicts.
Linear Algorithm

The data dependency analysis begins with a list of microinstructions (originally empty). Proceeding sequentially, microoperations are selected and the listed of microinstructions are searched from bottom to top to determine the rise limit. This is the first microinstruction in which the microoperation can be included given the data dependencies.

The microinstruction list is searched top-to-bottom in conflict analysis. Beginning at the rise limit, search proceeds downward until a microinstruction is found which can include the microoperation without hardware conflict. If a microoperation having a rise limit cannot be included in an existing microinstruction, a new one is created at the bottom of the list. If the microoperation has no rise limit, it is included in a new microinstruction at the beginning of the list. The linear algorithm does not guarantee optimal code. Dasgupta's work has produced an algorithm that is the model for Landskov's [DASG76, DASG77, BARN78, DASG78a, DASG79].

Critical Path Algorithm

An early partition is created as the first step. By working down through the microoperations using the data dependency graph, the earliest that each microoperation can be executed, assuming there is no conflicts for hardware, is determined. The total number of steps or frames needed to fit in all microoperations under these conditions is the minimum number of microinstructions which are required for that sequence. Next, working upward, microoperations are placed in the last
possible frame in which they can be executed so that the minimum number of frames determined earlier can be maintained. Critical microoperations are those with the same early and late timing and constitute the critical partition. The critical partition is then modified by dividing any frame in which there is hardware conflicts into two (or more) frames.

To form the final microinstructions, the non-critical microoperations are added to the modified critical partition. Each such microoperation can be included from their frame in the early partition to their frame in the late partition inclusive. If a hardware conflict does not allow a microoperation to be included within these frames, a new microinstruction is inserted just following the last partition position. This algorithm could result in many more microinstructions than is optimal. Its weakness is that adjacent microinstructions which could be combined are not if they are in the different frames. This algorithm was apparently devised from critical path processor scheduling [RAMA79] by Ramamoorthy and Tsuchiya [RAMA74].

Branch and Bound Algorithm

Using the data dependency graph, a data available set (Dset) is built which contains those microoperations which have all of their directly data dependent microoperations allocated to a microinstruction. Obviously, the original Dset consists of those microoperations with no parent nodes in the graph. Using the Dset, microinstructions are formed such that every possible member of the Dset are included. This is a
complete instruction and the collection of these is used to build a tree whose nodes correspond to the microinstructions. In BAB exhaustive, a complete tree is built whose paths represent every microinstruction ordering possible. From this tree, the path of optimal length is found and the corresponding complete instructions are the shortest possible microcode. [LAND80] and [MALL78] explain this algorithm in great detail. This microcode compaction algorithm was originally presented by Yau, Schowe, and Tsuchiya [YAU74]. Additionally, different heuristics can be applied to the branch and bound algorithm to decrease the number of possible paths and to cut execution time dramatically while still achieving near-optimal results [MALL78].

List Scheduling

In list scheduling algorithms, only a single branch of a tree is formed by choosing the "best" complete instruction from each Dset. "Best" is determined by a weighting function. The choice of function is important to achieving optimal microcode [FISH79]. One possible weight for a microoperation is the number of descendants that microoperation has in the data dependency graph [WOOD78]. Thus when choosing a microoperation from the Dset, the order is from highest to lowest weight. Each microoperation is added to the microinstruction unless a hardware conflict occurs. When all Dset members have been considered, a new Dset is created and a new microinstruction is started.
Global Optimization

Unlike local optimization, global optimization can look beyond segments to include loops and recursive subroutines. Its primary goal is to eliminate the NOP's positioned within microinstructions in the interest of proper timing of operations [LAND80]. Global optimization is an area where little research has been done until very recently. [WOOD79, DASG79, MALL78, FISH79, TOK078] deal with this subject.

[TOK078] has developed a global routine based on a microoperation model called a microtemplate. This microtemplate is two-dimensional, representing both timing and machine resources. [TOK078] uses an optimization algorithm which seeks primarily to eliminate NOP's positioned at the beginning and/or end of sequential segments by consolidation, or entire redundant microtemplates, in either an upward or downward direction. When compared to hand optimized code, the globally optimized code displayed an average improvement of 5.4%. Similar approaches used by [WOOD79] and [FISH79] treat a locally compacted sequence as a primitive in a larger sequence.

Register Allocation

Computers designed for efficient microprogramming typically have a large number of very fast registers. Ideally, there are enough registers so that each variable can be permanently assigned to a register during program execution. When there are too few registers, variables must sometimes be stored in memory. This means a store-and-load sequence is often required when a variable is brought into a register.
This additional overhead has been shown to affect performance. Liu [LIU75] found that having sufficient registers could improve microprogram execution time by thirty-three percent over the time with no registers. Obviously, it is important to determine which variables are most frequently referenced and to assign these variables to registers first. It is also important to have an efficient reallocation algorithm to keep reallocations to a minimum [DEWI76, TAN77].

With memory costs decreasing, manufacturers are tending to include even more high-speed microprogramming registers in their machines. This trend could make the problem of register allocation of minimal concern in the future.


