Trace Scheduling: A Technique for Global Microcode Compaction

JOSEPH A. FISHER, STUDENT MEMBER, IEEE

Abstract—**Microcode compaction is the conversion of sequential microcode into efficient parallel (horizontal) microcode. Local compaction techniques are those whose domain is basic blocks of code, while global methods attack code with a general flow control. Compilation of high-level microcode languages into efficient horizontal microcode and good hand coding probably both require effective global compaction techniques.**

In this paper "trace scheduling" is developed as a solution to the global compaction problem. Trace scheduling works on traces (or paths) through microprograms. Compacting is thus done with a broad overview of the program. Important operations are given priority, no matter what their source block was. This is in sharp contrast with earlier methods, which compact one block at a time and then attempt iterative improvement. It is argued that those methods suffer from the lack of an overview and make many undesirable compactions, often preventing desirable ones.

Loops are handled using the reducible property of most flow graphs. The loop handling technique permits the operations to move around loops, as well as into loops where appropriate.

Trace scheduling is developed on a simplified and straightforward model of microinstructions. Guides to the extension to more general models are given.

Index Terms—**Data dependency, global microcode optimization, microcode compaction, parallel instruction scheduling, parallel processing, resource conflict.**

I. Introduction

THIS paper presents trace scheduling, a solution to the "global microcode optimization problem." This is the problem of converting vertical (sequential) microcode written for a horizontally microcoded machine into efficient horizontal (parallel) microcode, and as such, is properly referred to as "compaction" rather than "optimization." In the absence of a general solution to this problem, the production of efficient horizontal microprograms has been a task undertaken only by those willing to painstakingly learn the most unintuitive and complex hardware details. Even with detailed hardware knowledge, the production of more than a few hundred lines of code is a major undertaking. Successful compilers into efficient horizontal microcode are unlikely to be possible without a solution to the compaction problem.

Local compaction is restricted to basic blocks of microcode. A basic block is a sequence of instructions having no jumps into the code except at the first instruction and no jumps out except at the end. A basic block of microcode has often been described in the literature as "straight-line microcode." Previous research [1]-[3] has strongly indicated that within basic blocks of microcode, compaction is practical and efficient. In Section II we briefly summarize local compaction and present many of the definitions used in the remainder of the paper.

Since blocks tend to be extremely short in microcode, global methods are necessary for a practical solution to the problem. To globally compact microcode it is not sufficient to compact each basic block separately. There are many opportunities to move operations from block to block and the improvement obtained is significant. Earlier methods have compacted blocks separately and searched for opportunities for interblock operation movement. However, the motivating point of this paper is the argument that these methods will not suffice. Specifically:

> Compacting a block without regard to the needs and capacities of neighboring blocks leads to too many arbitrary choices. Many of these choices have to be undone (during an expensive search) before more desirable motions may be made.

As an alternative, we offer trace scheduling. Trace scheduling compacts large sections of code containing many basic blocks, obtaining an overview of the program. Unless certain operations are scheduled early, delays are likely to percolate through the program. Such critical operations, no matter what their source block, are recognized as such and are given scheduling priority over less critical operations from the outset. Trace scheduling works on entire microprograms, regardless of their control flow, and appears to produce compactions strikingly similar to those laboriously produced by hand. Trace scheduling is presented in Section III. In Section IV suggestions are given to extend trace scheduling to more realistic models of microcode than that used in the exposition.

II. Local Compaction

It is not the purpose of this paper to present local compaction in detail. Thorough surveys may be found in [1]-[3]. However, trace scheduling uses local compaction as one of its steps (and we have developed an approach that we prefer to the earlier methods), so we briefly summarize an attack on that problem.

A. A Straightforward Model of Microcode

In this paper we use a relatively straightforward model of microcode. Trace scheduling in no way requires a simplified model, but the exposition is much clearer with a model that represents only the core of the problem. Reference [1] contains a more thorough model and references to others.

Manuscript received July 15, 1980; revised November 15, 1980. This work was supported in part by the U.S. Department of Energy under Grant EY-76-C-02-3077.

The author was with the Courant Institute of Mathematical Sciences, New York University, New York, NY. He is now with the Department of Computer Science, Yale University, New Haven, CT 06520.

During compaction we will deal with two fundamental objects: microoperations (MOP's) and groups of MOP's (called "bundles" in [1]). We think of MOP's as the fundamental atomic operations that the machine can carry out. Compaction is the formation of a sequence of bundles from a source sequence of MOP's; the sequence of bundles is semantically equivalent to the source sequence of MOP's.

Both MOP's and bundles represent legal instructions that the processor is capable of executing. We call such instructions microinstructions, or MI's. MI's are the basic data structure that the following algorithms manipulate. A flag is used whenever it is necessary to distinguish between MI's which represent MOP's and those which represent bundles of MOP's. (Note: In the definitions and algorithms which follow, less formal comments and parenthetical remarks are placed in brackets.)

Definition 1: A *microinstruction* (*MI*) is the basic unit we will work with. The set of all MI's we consider is called P. [An MI corresponds to a legal instruction on the given processor. Before compaction, each MI is one source operation, and P is the set of MI's corresponding to the original given program.]

There is a function *compacted:* $P \rightarrow \{\text{true, false}\}$. [Before compaction, each MI has its compacted flag set false.] If m is an MI such that compacted (m) = false, then we call m a *MOP.*

In this model MI's are completely specified by stating what registers they write and read, and by stating what resources they use. The following definitions give the MI's the formal properties necessary to describe compaction.

Definition 2: We assume a set of *registers.* [This corresponds to the set of hardware elements (registers, flags, etc.) capable of holding a value between cycles.] We have the functions *readregs, writeregs:* $P \rightarrow$ subsets of registers. [readregs(m) and writeregs(m) are the sets of registers read and written, respectively, by the MI m.]

Definition 3: We are given a function *resource compatible:* subsets of $P \rightarrow \{\text{true, false}\}$. [That a set of MI's is resource_compatible means that they could all be done together in one processor cycle. Whether that is the case depends solely upon hardware limitations, including the available microinstruction formats. For most hardware, a sufficient device for calculating resource_compatible is the **resource vector**. Each MI has a vector with one component for each resource, with the kth component being the proportion of resource k used by the MI. A set is resource_compatible only if the sum of its MI's resource vectors does not exceed one in any component.]

B. Compaction of MOP's into New MI's

During compaction the original sequence of MOP's is rearranged. Some rearrangements are illegal in that they destroy data integrity. To prevent illegal rearrangements, we define the following.

Definition 4: Given a sequence of MI's $(m_1, m_2, \cdots, m_t)$, we define the partial order $\ll$ on them. When $m_i \ll m_j$, we say that m_i *data precedes* m_j. Data precedence is defined carefully in Section IV, when the concept is extended somewhat. For now we say, informally, given MI's m_i, m_j with $i < j$:

- if m_i writes a register and m_j reads that value, we say that $m_i \ll m_j$ so that m_j will not try to read the data until it is there;
- if m_j reads a register and m_k is the next write to that register, we say that $m_j \ll m_k$ so that m_k will not try overwrite the register until it has been fully read.

Definition 5: The partial order $\ll$ defines a directed acyclic graph on the set of MI's. We call this the *data-precedence DAG.* The nodes of the DAG are the MI's, and an edge is drawn from m_i to m_j if $m_i \ll m_j$.

Definition 6: Given a DAG on P, we define a function *successors:* $P \rightarrow$ subsets of P. If m_i, m_j belong to P, $i < j$, we place m_j in successors(m_i) if there is an edge from m_i to m_j. [Many of the algorithms that follow run in $0(E)$ time, where E is the number of edges in the DAG. Thus, it is often desirable to remove redundant edges. If $m_i \ll m_j \ll m_k$, then a redundant edge from m_i to m_k is called a "transitive edge." Techniques for the removal of transitive edges are well known [4].]

Definition 7: Given a P, with a data-precedence DAG defined on it, we define a *compaction* or a *schedule* as any partitioning of P into a sequence of disjoint and exhaustive subsets of P, $S = (S_1, S_2, \cdots, S_u)$, with the following two properties.

- For each k, $1 \le k \le u$, resource_compatible(S_k) = true. [That is, each element of S could represent some legal microinstruction.]
- If $m_i \ll m_j$, m_i is in S_k, and m_j is in S_h, then $k < h$. [That is, the schedule preserves data-precedence.]

[The elements of S are the "bundles" mentioned earlier. Note that this definition implies another restriction of this model. All MI's are assumed to take one microcycle to operate. This restriction is considered in Section IV.]

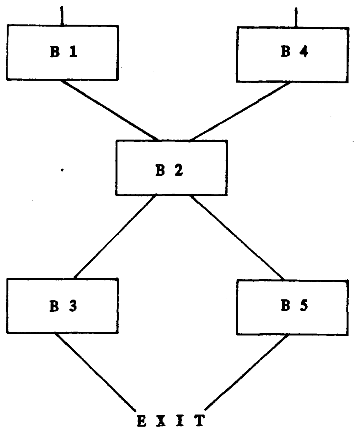
It is suggested that readers not already familiar with local compaction examine Fig. 1. While it was written to illustrate global compaction, Fig. 1(a) contains five basic blocks along with their resource vectors and readregs and writeregs sets. Fig. 1(d) shows each block's DAG, and a schedule formed for each. Fig. 5(a) contains many of the MOP's from Fig. 1 scheduled together, and is a much more informative illustration of a schedule. The reader will have to take the DAG in Fig. 5(a) on faith, however, until Section III.

Compaction via List Scheduling

Our approach to the basic block problem is to map the simplified model directly into **discrete processor scheduling theory** (see, for example, [5] or [6]). In brief, discrete processor scheduling is an attempt to assign tasks (our MOP's) to time cycles (our bundles) in such a way that the data-precedence relation is not violated and no more than the number of processors available at each cycle are used. The processors may be regarded as one of many resource constraints; we can then

MOP NUMBER	BLOCK NAME	REGISTERS WRITTEN	REGISTERS READ	FOLLOWERS (DEFAULT FALLS THROUGH TO NEXT MOP ON LIST)	RESOURCE VECTOR
ENTRANCE				m1	
ENTRANCE				m15	
m1	B1:	R3	R1,R2		[0, 1, 1, 0]
m2		R5	R3,R4		[1, 0, 0, 0]
m3		R6	R3		[1, 0, 0, 0]
m4		R7	R5,R6		[0, 1, 1, 0]
m5		R8	R7,R3		[1, 0, 0, 0]
m6	B2:	R10	R9		[1, 0, 1, 0]
m7		R11	R9		[0, 1, 1, 0]
m8			R10	m9, m17	[0, 0, 1, 0]
m9	B3:	R13	R12		[0, 0, 0, 1]
m10		R15	R10		[0, 0, 0, 0]
m11		R16	R13,R14		[0, 0, 0, 1]
m12		R17	R16,R11		[0, 0, 0, 1]
m13		R18	R17,R5		[0, 0, 0, 1]
m14		R19	R18	EXIT	[0, 0, 0, 1]
m15	B4:	R5	R3,R4		[0, 0, 0, 1]
m16		R8	R7,R3	m6	[0, 0, 0, 1]
m17	B5:	R12	R13		[1, 0, 0, 0]
EXIT		the following registers dead at this point of the code: R1-7, R9-11, R13, R18			

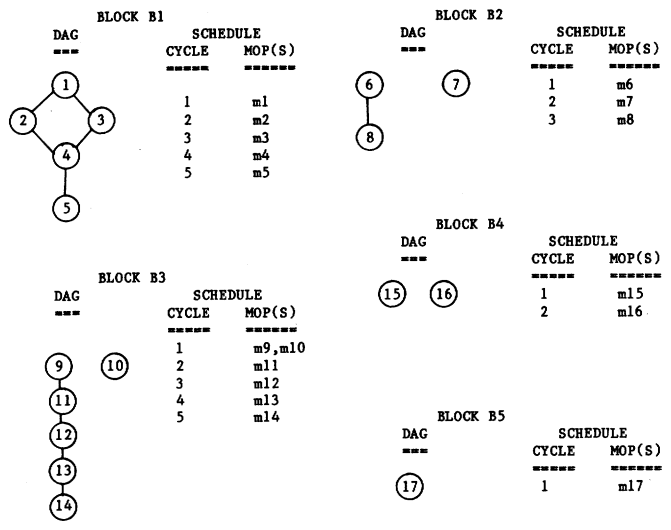
(a)



(b)

BLOCK NUMBER	MOPS FREE AT TOP	MOPS FREE AT BOTTOM	REGISTERS LIVE AT TOP
B1	m1	m5	R1-2,R4,R9,R12-17
B2	m6,m7	m7,m8	R5,R8,R9,R12-17
B3	m9,m10	m10,m14	R5,R8,R10-12,R14
B4	m15,m16	m15,m16	R3,R4,R7,R9,R12-17
B5	m17	m17	R8,R13-17

(c)



BLOCK B1 SCHEDULE

CYCLE	MOP(S)
1	m1
2	m2
3	m3
4	m4
5	m5

BLOCK B2 SCHEDULE

CYCLE	MOP(S)
1	m6
2	m7
3	m8

BLOCK B3 SCHEDULE

CYCLE	MOP(S)
1	m9,m10
2	m11
3	m12
4	m13
5	m14

BLOCK B4 SCHEDULE

CYCLE	MOP(S)
1	m15
2	m16

BLOCK B5 SCHEDULE

CYCLE	MOP(S)
1	m17

(d)

Fig. 1. (a) Sample loop-free code, the essential instruction information. (b) Flow graph for Fig. 1. (c) Flow graph information for Fig. 1. (d) Data precedence DAG's and schedule for each block compacted separately.

say that basic block compaction is an example of unit execution time (UET) scheduling with resource constraints. As such it can be shown that the basic block problem is NP-complete, since even severely restricted scheduling analogs are [5]. Thus, we would expect any optimal solution to be exponential in the number of MOP's in a block.

Despite the NP-completeness, simulation studies indicate that some simple heuristics compact MOP's so well that their lack of guaranteed optimality does not seem to matter in any practical sense [2], [3]. The heuristic method we prefer is **microinstruction list scheduling** (2). List scheduling works by assigning priority values to the MOP's before scheduling begins. The first cycle is formed by filling it with MOP's, in order of their priorities, from among those whose predecessors have all previously been scheduled (we say such MOP's are **data ready**). A MOP is only placed in a cycle if it is resource compatible with those already in the cycle. Each following cycle is then formed the same way; scheduling is completed when no tasks remain. Fig. 2 gives a more careful algorithm for list scheduling; the extension to general resource constraints is straightforward.

Various heuristic methods of assigning priority values have been suggested. A simple heuristic giving excellent results is **highest levels first**, in which the priority of a task is the length of the longest chain on the data-precedence DAG starting at that task, and ending at a leaf. Simulations have indicated that highest levels first performs within a few percent of optimal in practical environments [7], [2].

III. Global Compaction Using Trace Scheduling

We now consider algorithms for global compaction. The simplest possible algorithm would divide a program into its basic blocks and apply local compaction to each. Experiments have indicated, however, that the great majority of parallelism is found beyond block boundaries [8], [9]. In particular, blocks in microcode tend to be short, and the compactions obtained are full of "holes," that is, MI's with room for many more MOP's than the scheduler was able to place there. If blocks are compacted separately, most MI's will leave many resources unused.

A. The Menu Method

Most hand coders write horizontal microcode "on the fly," moving operations from one block to another when such motions appear to improve compaction. A menu of code motion rules that the hand coder might implicitly use is found in Fig. 3. Since this menu resembles some of the code motions done by optimizing compilers, the menu is written using the terminology of flow graphs. For more careful definitions see [10] or [11], but informally we say the following.

Definition 8—Flow Graph Definitions: A *flow graph* is a directed graph with nodes the basic blocks of a program. An edge is drawn from block B_i to block B_j if upon exit from B_i control may transfer to B_j.

```
GIVEN:  A set of tasks with a partial order (and thus a DAG) defined on them
        and P identical processors.

        A function PRIORITY-SET which assigns a priority value to each task
        according to some heuristic.

BUILDS: A schedule in which tasks are scheduled earlier than their successors
        on the DAG, and no more than P tasks are scheduled each cycle.  The
        schedule places high priority tasks ahead of low priority tasks when
        it has a choice.

ALGORITHM:

        PRIORITY-SET is called to assign to a priority value to
        each task.

        CYCLE = 0

        DRS (the DATA READY SET) is formed from all tasks with no
        predecessors on the DAG.

        While DRS is not empty, do

                CYCLE = CYCLE + 1

                The tasks in DRS are placed in cycle CYCLE in order of
                their priority until DRS is exhausted or P tasks
                have been placed.  All tasks so placed are removed
                from DRS.

                All unscheduled tasks not in DRS whose predecessors
                have all been scheduled are added to DRS.

                end (while)

        Scheduling is finished, CYCLE cycles have been formed.
```

Fig. 2. An algorithm for list scheduling.

RULE NUMBER	MOP CAN MOVE FROM	TO	UNDER THE CONDITIONS THAT
1	B2	B1 and B4	the MOP is free at the top of B2
2	B1 and B4	B2	identical copies of the MOP are free at the bottoms of both B1 and B4
3	B2	B3 and B5	the MOP is free at the bottom of B2
4	B3 and B5	B2	identical copies of the MOP are free at the tops of both B3 and B5
5	B2	B3 (or B5)	the MOP is free at the bottom of B2 and all registers written by the MOP are dead in B5 (or B3)
6	B3 (or B5)	B2	the MOP is free at the top of B3 (or B5) and all registers written by the MOP are dead in B5 (or B3)

Block numbers refer to the flow graph in example 2(b).

Any of the above motions will be beneficial if the removal of the MOP allows (at least one) source block to be shorter, with no extra cycles required in (at least one) target block. It may be necessary to recompact the source and/or targets to realize the gain.

Fig. 3. The menu – rules for the motion of MOP's to blocks other than the one they started in.

The graph may be acyclic, in which case we say it is *loop free*. If it has cycles, the cycles are called loops. [Although a formal definition of a loop is somewhat more difficult than it may at first seem.]

A register is *live* at some point of a flow graph if the value stored in it may be referenced in some block after that point, but before it is overwritten. A register not live at some point is *dead* at that point. We say a MOP is *free* at the top of its block if it has no predecessors on the data-precedence DAG of the block, and free at the bottom if it has no successors. [A MOP may be free at both the top and the bottom of its block.]

Flow graph concepts are illustrated in Fig. 1; see especially (c).

B. Earlier Methods of Global Compaction

Previous suggestions for global compaction have explicitly automated the menu method [12], [13]. This involves essentially the following steps.

1) Only loop-free code is considered (although we will soon consider a previous suggestion for code containing loops).

2) Each basic block is compacted separately.

3) Some ordering of the basic blocks is formed. This may be as simple as listing pairs of basic blocks with the property that if either is ever executed in a pass through the code, so is the other [12], or it may be a walk through the flow graph [13].

4) The blocks are examined in the order formed in 3), and legal motions from the current block to previously examined ones are considered. A motion is made if it appears to save a cycle.

Limitations of the Earlier Methods

The "automated menu" method appears to suffer from the following shortcomings.

• Each time a MOP is moved, it opens up more possible motions. Thus, the automated menu method implies a massive and expensive tree search with many possibilities at each step.

• Evaluating each move means recompacting up to three blocks, an expensive operation which would be repeated quite often.

• To find a sequence of very profitable moves, one often has to go through an initial sequence of moves which are either not profitable, or, worse still, actually make the code longer. Locating such a sequence involves abandoning attempts to prune this expensive search tree.

We summarize the shortcomings of the automated menu method as follows:

Too much arbitrary decisionmaking has already been made once the blocks are individually compacted. The decisions have been shortsighted, and have not considered the needs of neighboring blocks. The movement may have been away from, rather than towards, the compaction we ultimately want, and much of it must be undone before we can start to find significant savings.

An equally strong objection to such a search is the ease with which a better compaction may be found using trace scheduling, the method we will present shortly.

C. An Example

Fig. 1 (a)–(d) is used to illustrate the automated menu method and to point out its major shortcoming. The example was chosen to exaggerate the effectiveness of trace scheduling in the hope that it will clarify the ways in which it has greater power than the automated menu method. The example is not meant to represent a typical situation, but rather the sort that occurs frequently enough to call for this more powerful solution to the problem. Fig. 1 is not written in the code of any actual machine. Instead, only the essential (for our purposes) features of each instruction are shown. Even so, the example is quite complex.

Fig. 1(a) shows the MOP's written in terms of the registers written and read and the resource vector for each MOP. The jump MOP's show the names of the blocks it is possible to jump to, and the flow graph obtained is shown in Fig. 1(b). In Fig. 1(c) is a table showing the registers live and MOP's free at the top and bottom of each block. Finally, Fig. 1(d) shows each block's data dependency DAG and a compaction of the block. The compaction would be obtained using list scheduling and almost any heuristic (and probably any previously suggested compaction method).

Given the blocks as compacted in Fig. 1(d), an automated menu algorithm could then choose to use rules $R1$–$R6$ to move MOP's between blocks. Fig. 4 shows the possible application of some of these rules, and using them we see that some of the blocks may be shortened. If, for the moment, we suppose that the code usually passes through the path $B1$-$B2$-$B3$, we can see that the length of that path may be reduced from the initial 13 cycles (with each block compacted separately) to 11 cycles, an important savings. Nonetheless, we shall see that this is an example of an unsatisfactory compaction obtained using the automated menu method.

D. Trace Scheduling–Compacting Several Blocks Simultaneously

The shortcomings of the automated menu method are effectively dealt with using the technique we call **trace scheduling**. Trace scheduling operates on **traces** instead of basic blocks. A trace is a loop-free sequence of instructions which might be executed contiguously for some choice of data. More formally, we define the following.

Definition 9: There is a function *followers:* $P \rightarrow$ subsets of P. [Given an MI, say m, the set followers(m) is all MI's to which control could next pass after m is executed.] If m_i is in followers(m_j), then we say that m_j is a *leader* of m_i. If there is more than one MI in followers(m), then m is called a *conditional jump* MI.

We define a *trace* as any sequence of distinct MI's $(m_1, m_2, \cdots, m_t)$ such that for each j, $1 \leq j \leq t-1$, m_{j+1} is in followers(m_j). [Thus, a trace is a path through the code which could (presumably) be taken by some setting of the data. Note that if no MI in T is a conditional jump, T might be a basic block, or part of one. In general, T may contain many blocks.]

To allow us to consider P to be a portion of a larger program, we have dummy MI's which are on the boundaries of P. We call these dummy MI's entrances and exits, and they are used to interface P with the rest of the program.

Definition 10: We assume that some of the MI's are distinguished as *exits* and some as *entrances*. Exits are dummy MI's each representing the start of a section of code outside of P. Exits have no followers. Entrances are dummy MI's each representing some jump into P from outside P. Entrances are not followers of any MI in P.

Before compaction we set the compacted value of both entrances and exits to true, so they never appear on a trace. After compaction is completed, we replace all jumps to exits by jumps to the code location, outside P, represented by the exit. Similarly, we change all jumps from code outside P to entrances by having the jump be to the followers of the entrance.

RULE NUMBER	EXAMPLE OF MOTION	REALIZABLE SAVINGS
1	move MOP 7 from block B2 to blocks B1 and B4	block B2 goes from 3 to 2 cycles while, by placing a copy of MOP 7 next to MOPs 2 and 15, B1 and B4 stay the same size
2	move MOPs 5 and 16 from blocks B1 and B4 to form a new MOP in B2, if MOPs 5 and 16 are identical	since the new MOP may be placed next to MOP 8, it costs nothing in B2, while saving a cycle in both B1 and B4
5	move MOP 7 (if it has not been moved by rule 1) from block B2 into block B3 (possible since register R11 is dead at the entrance to block B5)	since MOP 7 may be placed next to MOP 11, it costs nothing in block B3, while saving a cycle in block B2

Fig. 4. Examples of the savings available in Fig. 1(d) via the menu method.

Building Data-Precedence Graphs on Traces

The automated menu method hunts for specific cases of interblock motions and examines each only after compacting basic blocks. Trace scheduling takes the opposite tack. Here, the scheduler is given in advance the full set of MOP's it has to work with and is allowed to produce whatever schedule is most effective. No explicit attention is given to the source code block structure during compaction. Sometimes, though, it is not permissible for an operation to move from one block to another. That new information and ordinary data-precedence determine edges for a DAG. Given this DAG, it is possible to explain the main technique of trace scheduling as follows.

> The DAG built for a trace already contains all of the necessary restrictions on interblock motion, and only those restrictions. A scheduler may compact the trace without any knowledge of where the original block boundaries were. The scheduler's sole aim will be to produce as short a schedule as possible for the trace, making implicit interblock motions wherever necessary to accomplish this goal. This may be done at the expense of extra space, and may sometimes lengthen other traces. Thus, the process is applied primarily to the traces most likely to be executed.
>
> More formally, we build the DAG as follows.

Definition 11: Given a trace $T = (m_1, m_2, \cdots, m_t)$, there is a function *condreadregs:* the set of conditional jumps in T $\rightarrow$ subsets of registers. If $i < t$, register r is in condreadregs(m_i) if r is live at one or more of the elements of followers(m_i) $- \{m_{i+1}\}$. [That is, at one of the followers besides the one which immediately follows on the trace. Algorithms for live register analysis are a standard topic in compiler research [10], [11]. We assume that updated live register information is available whenever it is required.] For the last element on T, we define condreadregs(m_t) as all registers live at any follower of m_t.

Definition 12: Given a trace $T = (m_1, m_2, \cdots, m_t)$, we define the successors function to build a directed acyclic graph (DAG) called the *trace data-precedence graph.* [Or, just the data-precedence graph or DAG if the context is obvious.] This is calculated exactly as if T were a basic block, using the sets readregs(m) and writeregs(m), except for the DAG edges from conditional jumps. If m is a conditional jump, then all the registers in condreadregs(m) are treated as if they were in the set readregs(m) for purposes of building successors(m). [This is to prevent values which may be referenced off the trace to be overwritten by an instruction which moves from below m to above m during compaction. Again, the DAG is defined more carefully when we extend it slightly in the next section.]

Scheduling Traces

In brief, trace scheduling proceeds as follows.

To schedule P, we repeatedly pick the "most likely" trace from among the uncompacted MOP's, build the trace DAG, and compact it. After each trace is compacted, the implicit use of rules from the menu forces the duplication of some MOP's into locations off the trace, and that duplication is done. When no MOP's remain, compaction has been completed.

To help pick the trace most likely to be executed, we need to approximate the expected number of times each MOP would be executed for a typical collection of data.

Definition 13: We are given a function *expect*: $P \rightarrow$ nonnegative reals. [Expect(m) is the expected number of executions of m we would expect for some typical mix of data. It is only necessary that these numbers indicate which of any pair of blocks would be more frequently executed. Since some traces may be shortened at the expense of others, this information is necessary for good global compaction. For similar reasons, the same information is commonly used by the hand coder. An approximation to expect may be calculated by running the uncompacted code on a suitable mix of data, or may be passed down by the programmer.]

Given the above definitions, we can now formally state an algorithm for trace scheduling loop-free code.

Algorithm: Trace Scheduling

```
Given:      P, a loop-free set of microinstructions with
            all of the following predefined on P:
            leaders, followers, exits, entrances,
            readregs, writeregs, expect,
            resource_compatible.

Builds:     A revised and compacted P, with new MI's
            built from the old ones. The new P is
            intended to run significantly faster than
            the old, but will be semantically equivalent
            to it.

Uses:       T, a variable of type trace.
            S, a variable of type schedule.
            pick_trace, schedule, bookkeep, all
            subroutines. [Explained after this
            algorithm.]

Algorithm:  for all $m_i$ in P, compacted($m_i$) = false;
            for all exits and entrances compacted =
            true;
            while at least one MI in P has compacted
            = false do;
              call pick_trace(T);
                [Sets T to a trace. T is picked to be
                the most frequently executed path
                through the uncompacted MI's left in
                P.]
              call schedule(T);
                [Produces a trace schedule S on T
                after building a DAG.]
              call bookkeep(S);
                [Builds a new, compacted MI from
                each element of S, changing all of the
                predefined functions as necessary,
                both within and outside of T.
                Duplicates MI's from T and places
                them in P, where necessary.]
              end;
```

The Subroutines Pick_trace, Schedule, Bookkeep

Algorithm: Pick Trace

```
Given:      P, as defined above.

Builds:     A trace T of elements from P. [The trace
            is intended to represent the "most likely"
            path through the uncompacted portion of
            the code.]

Method:     Picks the uncompacted MOP with the
            highest expect, calling that m. Builds a
            trace around m by working backward and
            forward. To work backward, it picks the
            uncompacted leader of m with the highest
            expect value and repeats the process with
            that MOP. It works forward from m
            analogously.

Uses:       m, m', $m_{\max}$, $m_i$ all MI's.
            F, G sets of MI's.

Algorithm:  $m_{\max}$ = the MOP m in P such that if m' is
            a MOP in P, then expect(m) $\geq$
            expect(m'). Break ties arbitrarily; [Recall
            that if $m_i$ is a MOP, compacted($m_i$) =
            false, so $m_{\max}$ is the uncompacted MI with
            highest expect.]
            m = $m_{\max}$;
            T = (m); [The sequence of only the one
            element.]
            F = the set of MOP's contained within
            leaders(m);
            do while F not empty;
              m = element of F with largest expect
              value;
              T = AddToHeadOfList(T, m); [Makes
              a new sequence
              of the old T preceded by m.]
              F = the set of MOP's contained within
              leaders(m);
              end;
            m = $m_{\max}$;
            G = the set of MOP's in followers(m);
            do while G not empty;
              m = element of G with largest expect
              value;
```

```
        T = AddToTailOfList(T, m); [Makes a
        new sequence
        of the old T followed by m.]
        G = the set of MOP's contained within
        followers(m);
        end;
    end;
```

Algorithm: Schedule

Given: P, as defined above.
T, a *trace* of elements from P.

Builds: S, a *trace schedule.*

Method: Builds a DAG on the trace as described earlier. Once the DAG has been built, list scheduling is done just as it is in local compaction.

Uses: *Build_DAG*, a subroutine that builds the trace data-precedence DAG by filling in the *successor* MOP's for each MOP on the trace. This uses the *condreadregs* sets as explained in the definition of trace data-precedence.
Priority_set, a subroutine that gives a priority value to each MOP on the trace. Any method (such as highest levels first) that works well for basic blocks may be used here, although it may be advantageous to always favor elements with relatively high *expect* values over those without.
list_schedule, the subroutine used for local compaction.

Algorithm:

```
call build_DAG(T); [As described
above.]
call priority_set(T); [As described
above.]
call list_schedule(T, S); [Builds S, a
trace schedule for T.]
end;
```

By scheduling without paying attention to whether MOP's were conditional jumps, we have given the scheduler carte blanche to apply many possible code motions, and the scheduler has used a heuristic to choose which ones to implicitly apply. Menu rules $R1$ and $R3$ refer to the motion of MOP's up past a code join and down past a code split, respectively. We may find places in which the scheduler implicitly applied motion rules $R1$ and $R3$ without placing the moved MOP into both necessary blocks. We now must complete the motions to make them legal, as follows.

Algorithm: Bookkeep

Given: P, as defined above.
T, a *trace* of elements from P.
S, a *trace schedule*.

Builds: A revised P, with operations duplicated where necessary to make the code semantically equivalent to the original P. The operations placed in individual cycles are coalesced into new, compacted MI's.

Method: The details of the bookkeeping phase are very complex and their formal presentation is unintuitive. Instead of a careful algorithm, a more informal, hopefully clearer explanation is given.

Explanation—Repairing Rejoins: When there were jumps to the trace, we now must find a location in the new schedule to jump to. This may be difficult, because MOP's may have moved above and below the old join. We may only rejoin to places that have no MOP's at or below them which had been above the old join, since we don't want to execute such MOP's. When we find the highest point for a rejoin, menu rule $R1$ requires that all MOP's which had been below the join, but are now above it, be copied into the joining block. That is,

- We first consider each MOP m_i on T, $i > 1$, which has a leader besides m_{i-1}. Find the minimum S_j, an MI in the new schedule, with the property that for any k, $k \geq j$, S_k contains only MOP's m_h with $h \geq i$. That is, S_j is the spot on the schedule below which are found only MOP's which were at or below the old joining position. S_j is the new rejoining position corresponding to jumps to m_i in the original program.
- For each such rejoin, create a new block B_i which jumps to S_j. Change all jumps to m_i, besides the one from m_{i-1}, to jump to the new block B_i. (In other words, place B_i between all jumps to the rejoin and the rejoin itself.)
- Populate B_i with copies of all MOP's m_h with $h \geq i$ but which are now in some S_k, where $k < j$. In other words, populate this block with copies of MOP's which had been below the old join, but are above the new rejoin. We are now finished legalizing the scheduler's uses of rule $R1$.

Conditional Jumps: Some MOP's which were originally above a conditional jump on the trace may have been scheduled in an MI below the jump. This is an example of the scheduler implicitly choosing to use rule $R3$. In that case we must copy these MOP's to the place jumped to, as required by $R3$. If all registers written by the MOP are dead at that place, then $R5$ tells us that the MOP's did not need to be copied.

- Consider each m_i on T which has a *follower* besides m_{i+1}, (we consider m_t, the last trace element, to fit that requirement with all of its *followers*). m_j, then, has a jump besides the one forming the trace.

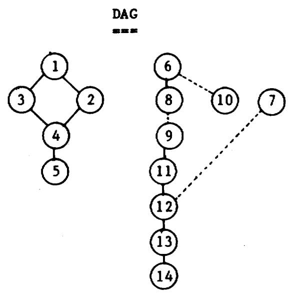
- Create a new block B_j for each m_j in *followers*(m_i) besides m_{i+1}. Change the jump to m_j from the MI containing m_i so it jumps to B_j instead, and have B_j jump to m_j. That is, place B_j between m_i and m_j.
- In B_j place a copy of any MOP m_k such that $k < i$, where m_k has been scheduled below the MI containing m_i. This completes the use of menu rule $R3$.
- Remove from B_j any MOP which writes only dead registers, that is, writes unreferenced in this branch. [This is the substitution of rule $R5$ for rule $R3$.]

After the above, we fix up the flow graph so that empty blocks are removed. The MOP's placed in a given block are written the order in which they appeared in the original code (although any topological sort of a data precedence DAG containing them would do). The *followers* of "fall through" jumps are adjusted to account for the new order. *Expect* values, which are functions of where the conditional jumps have been scheduled, are updated. (Here, the algorithm gets especially unintuitive.)

E. An Example, Revisited

Now consider Fig. 1. Suppose the most likely trace was the blocks $B1$, $B1$, $B3$. Fig. 5(a) shows the DAG for that trace, and a schedule that would be formed using the highest levels first priority function. The dotted lines in the DAG are edges that would arise between MOP's which originated in different blocks, but are treated no differently from ordinary edges. Note that, in this example, **the scheduler was able to find a significantly shorter compaction**, namely 7 cycles to the 11 which might be expected from the automated menu method. This is due to the overview the scheduling algorithm had of the whole trace. The necessity to execute m_6 early, in order to do the critical path of code, is obvious when looking at the DAG. An automated menu compactor would be unlikely to see a gain in moving m_6 into the first block, since there would be no cycle in which to place it.

The notes for Fig. 5(a) show the MOP's which would have to copied into new blocks during the bookkeeping phase. Fig. 5(b) shows the new flow graph after the copying. Fig. 5(c) and (d) display the rest of the traces being compacted.



SCHEDULE

CYCLE	MOP(S)
1	m6
2	m8,m10
3	m1,m9
4	m11,m2,m7
5	m3,m12
6	m4,m13
7	m5,m14

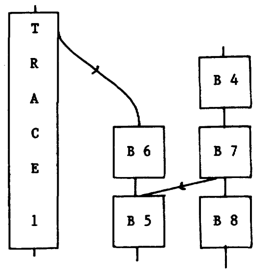
PRIORITY LIST: 6, 8, 9, 1, 7, 11, 2, 3, 12, 4, 13, 5, 10, 14

Bookkeeping phase:

Rejoin to path at MOP m6 cannot be made without including illegal earlier MOPs. MOPs 6-14 must be copied to after B4.

The following MOPs have moved to below the conditional jump at MOP m8 and must thus be copied to before B5: 1-5,7

(a)



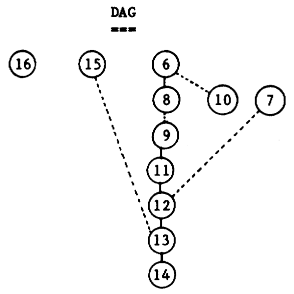
CONTENTS OF NEW BLOCKS

BLOCK	CONTAINS COPIES OF MOPs
B6	1,2,3,4,5,7
B7	6,7,8
B8	9,10,11,12,13,14

notes: TRACE 1 is the resultant compaction of B1, B2, B3 shown in example 2(a).

Blocks B4 and B7 could be merged to form a new basic block, we leave them unmerged in the example.

(b)

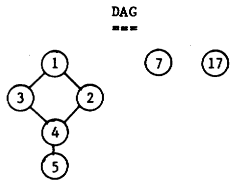


SCHEDULE

CYCLE	MOP(S)
1	m6,m15
2	m8,m10,m16
3	m9,m7
4	m11
5	m12
6	m13
7	m14

notes: MOP m7 has moved to below the conditional jump at MOP m8 and must thus be copied to before B5. Hopefully, an adequate bookkeeping phase will note the existence of m7 free at the top of B5 and will use a menu rule to move both into one MOP in B5.

(c)



SCHEDULE

CYCLE	MOP(S)
1	m1,m17
2	m2,m7
3	m3
4	m4
5	m5

notes: A rejoin is to be made at the entrance to block B5 from the path of B4, B7, B8. Thus a copy of m7 and m17 would have to be made in a new block, B9, which would fall through to the exit. Since that block is trivially parallelized, it is not shown.

(d)

Fig. 5. (a) Schedule for the main trace ($B1$, $B2$, $B3$). (b) Flow graph after the bookkeeping necessary to make the schedule formed in (a) legal. (c) Schedule for the next most commonly executed trace ($B4$, $B7$, $B8$). (d) Final blocks for the ($B5$, $B6$) of the flow graph compacted.

F. Code Containing Loops

We now extend trace scheduling to cover code with any control flow. It is important that code containing loops be handled well, since short loops are quite common in microcode. Typical examples of very short loops include byte shifts and waiting loops, which test for an infrequently true condition like "memory busy." We do not want to exclude these loops from the compacting process.

Definition 14: A **loop** is a set of MI's in P which correspond to some "back edge" (that is, an edge to an earlier block) in the flow graph. For a careful definition and discussion, see [11].

Reducible Flow Graphs

For convenience, we think of the whole set P as a loop. We assume that all of the loops contained in P form a sequence L_1, $L_2, \cdots, L_p$, such that

a) each L_i is a loop in P,

b) $L_p = P$,

c) if L_i and L_j have any elements in common, and $i < j$, then L_i is a subset of L_j. That is, we say that any two loops are either disjoint or nested, and that the sequence $L_1, L_2, \cdots, L_p$ is topologically sorted on the "include" relation.

The last requirement above is that P have a **reducible flow graph.** (For more information about reducible flow graphs, see [11].) Insisting that P have a reducible flow graph is not a problem for us for two reasons. One, programs formed using so-called "structured" control of flow, and not unrestrained GOTO's, are guaranteed to have this property. This is not a compelling argument, since it must be granted that one is apt to find wildly unstructured microcode (the nature of the micro machine level tends to encourage such practices). However, code generated by a compiler is unlikely to be so unstructured. The second reason is stronger, however, and that is that an irreducible program may easily be converted into a reducible one with the use of some simple techniques [11]. The automatic conversion produces a slightly longer program, but we have seen that small amounts of extra space is a price we are willing to pay. All known methods which guarantee that the conversion generates the least extra space rely on the solution of some NP-complete problem, but such a minimum is not important to us.

We will, then, assume that the flow graphs we are working with are reducible, and that the set of loops in P is partially ordered under inclusion. Fig. 6(a) is a sample reducible flow graph containing 12 basic blocks, $B1$–$B12$. We identify five sets of blocks as loops, $L1$–$L5$, and the table in Fig. 6(b) identifies their constituent blocks. The topological sort listed has the property we desire.

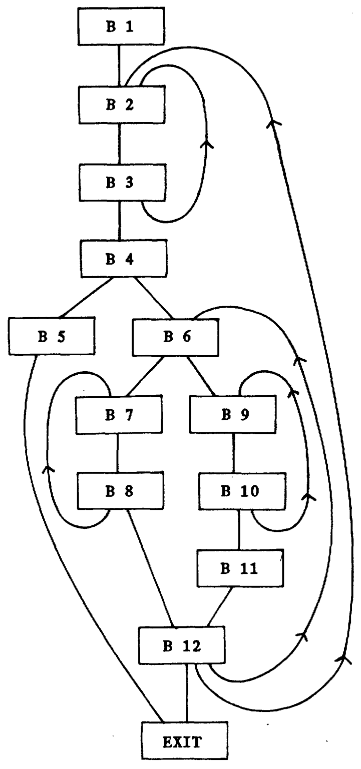
There are two approaches we may take in extending trace scheduling to loops; the first is quite straightforward.

A Simple Way to Incorporate Loops Into Trace Scheduling

The following is a method which strongly resembles a suggestion for handling loops made by Wood [14]. Compact the loops one at a time in the order $L_1, L_2, \cdots, L_p$. Whenever a loop L_i is ready to be compacted, all of the loops L_j contained within it have $j < i$, and have already been compacted. Thus, any MI contained in such an L_j will be marked as *compacted* [see Fig. 6(c)].

We can see that trace scheduling may be applied to L_i directly with no consideration given to the fact that it is a loop, using the algorithms given above. No trace will ever encounter an MI from any $L_j, j < i$, since they are all marked *compacted.* Thus, the traces selected from L_i may be treated as if they arose from loop-free code. There are still "back edges" in L_i, that is what made it a loop, but they are treated as jumps to *exits*, as are jumps to MI's outside of L_i.

When this procedure is completed, the last loop compacted will have been P. Each MOP will have been compacted in the

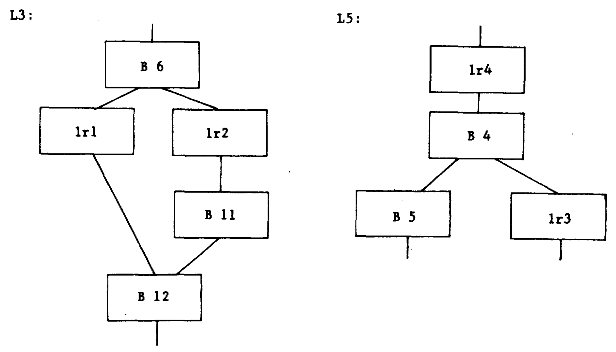


(a)

LOOP	CONTAINED BLOCKS	OUTERMOST CONTAINED LOOPS	NODES ON MODIFIED DAG
L1	<B7,B8>	none	B7,B8
L2	<B9,B10>	none	B9,B10
L3	<B6-12>	L1,L2	B6,B11,B12,lr1,lr2
L4	<B2,B3>	none	B2,B3
L5	<B2-12>	L3,L4	B4,B5,lr3,lr4
L6	(the whole set) <B1-12>	L5	B1,lr5

A TOPOLOGICAL SORT OF THE LOOPS: L4, L1, L2, L3, L5, L6
(each loop appears ahead of all loops it is contained in)

(b)



(c)

Fig. 6. (a) A reducible flow graph. (b) Loop structure information for the flow graph given in (a). (c) What the flow graphs $L3$ and $L5$ would look like at compaction time, considering each loop representative as a block.

L_i in which it is most immediately contained, and we will have applied trace scheduling to all of P. Even with the addition of trace scheduling, however, this does not provide enough power for many applications.

A More Powerful Loop Method

The above method fails to take advantage of the following potentially significant sources of compaction.

- Some operations which precede a loop could be moved to a point after the loop and vice versa. Loops should not be regarded as arbitrary boundaries between blocks. Compaction should be able to consider blocks on each side of a loop in a way that considers the requirements of the other, and MOP's should be moved between them wherever it is desirable and legal.

• In many microprograms loops tend to iterate very few times. Thus, it is often important to move MOP's into loops when they can be done without extra cycles inside the loop and are "loop invariant" (see [10] for details on loop invariance). This is an ironic reversal on optimizing compilers, which often move computations out of loops; we may want to move them right back in when we compact microcode. This is especially important on machines with loops which are rarely iterated more than once, such as waiting loops. We certainly do not want to exclude the cycles available there from the compaction process.

Loop Representative MI's

The more powerful loop method treats each already compacted loop as if it were a single MOP (called a **loop representative**) in the next outermost loop. Just as was the case for conditional jumps, we hide from the scheduler the fact that the MOP represents an entire loop, embedding all necessary constraints on the data precedence DAG. Then when a loop representative appears on a trace, the scheduler will move operations above and below the special MOP just as it would for any MOP. Once the definition of resource compatibility is extended to loop representatives, we may allow the scheduler to move other MOP's into the loop is well.

Definition 15: Given a loop L, its loop representative lr, and a set of operations N. The set $\{lr\} \cup N$ is **resource compatible** if:

• N contains no loop representatives,

• each operation in N is loop-invariant with respect to L,

• all the operations in N can be inserted into L's schedule without lengthening it. (There are enough "holes" in L's already formed schedule.)

Two loop representatives are never resource compatible with each other.

Trace Scheduling General Code

The more powerful method begins with the loop sequence $L_1, L_2, \cdots, L_p$ as defined above. Again, the loops are each compacted, starting with L_1. Now, however, consider the first loop L_j which has other loops contained in it. The MI's comprising each outermost contained loop, say L_j, are all replaced by a new dummy MOP, lr_j, the loop representative. Transfers of control to and from any of the contained loops are treated as jumps to and from their loop representatives. Fig. 6(c) shows what the flow graph would look like for two of the sample loops. Next, trace scheduling begins as in the nonloop case. Eventually, at least one loop representative shows up on one of the traces. Then it will be included in the DAG built for that trace. Normal data precedence will force some operations to have to precede or follow the loop, while others have no such restrictions. All of this information is encoded on the DAG as edges to and from the representative.

Once the DAG is built scheduling proceeds normally until some lr_j is data ready. Then lr_j is considered for inclusion in each new cycle C according to its priority (just as any operation would be). It is placed in C only if lr_j is resource compatible (in the new sense) with the operations already in C. Eventually, some lr_j will be scheduled in a cycle C (if only because it becomes the data ready task of highest priority). Further data ready operations are placed in C if doing so does not violate our new definition of resource compatibility.

After scheduling has been completed, lr_j is replaced by the entire loop body L_j with any newly absorbed operations included. Bookkeeping proceeds essentially as before. The techniques just presented permit MOP's to move above, below, and into loops, and will even permit loops to swap positions under the right circumstances. In no sense are arbitrary boundaries set up by the program control flow, and the blocks are rearranged to suit a good compaction.

This method is presented in more detail in [2].

IV. Enhancements and Extensions of Trace Scheduling

In this section we extend trace scheduling in two ways: we consider improvements to the algorithm which may be desirable in some environments, and we consider how trace scheduling may be extended to more general models of microcode.

A. Enhancements

The following techniques, especially space saving, may be critical in some environments. In general, these enhancements are useful if some resource is in such short supply that unusual tradeoffs are advantageous. Unfortunately, most of these are inelegant and rather ad hoc, and detract from the simplicity of trace scheduling.

Space Saving

While trace scheduling is very careful about finding short schedules, it is generally inconsiderate about generating extra MI's during its bookkeeping phase. Upon examination, the space generated falls into the following two classes:

1) space required to generate a shorter schedule,

2) space used because the scheduler will make arbitrary decisions when compacting; sometimes these decisions will generate more space than is necessary to get a schedule of a given length.

In most microcode environments we are willing to accept some extra program space of type 1, and in fact, the size of the shorter schedule implies that some or all of the "extra space" has been absorbed. If micromemory is scarce, however, it may be necessary to try to eliminate the second kind of space and desirable to eliminate some of the first. Some of the space saving may be integrated into the compaction process. In particular, extra DAG edges may be generated to avoid some of the duplication in advance–this will be done at the expense of some scheduling flexibility. Each of the following ways of doing that is parameterized and may be fitted to the relevant time–space tradeoffs.

If the expected probability of a block's being reached is below some threshold, and a short schedule is therefore not critical, we draw the following edges.

1) If the block ends in a conditional jump, we draw an edge to the jump from each MOP which is above the jump on the trace and writes a register live in the branch. This prevents

copies due to the ambitious use of rule $R3$ on blocks which are not commonly executed.

2) If the start of the block is a point at which a rejoin to the trace is made, we draw edges to each MOP free at the top of the block from each MOP, which is in an earlier block on the trace and has no successors from earlier blocks. This keeps the rejoining point "clean" and allows a rejoin without copying.

3) Since the already formed schedule for a loop may be long, we may be quite anxious to avoid duplicating it. Edges drawn to the loop representative from all MOP's which are above any rejoining spot on the trace being compacted will prevent copies caused by incomplete uses of rule $R1$. Edges drawn from the loop MOP to all conditional jumps below the loop will prevent copies due to incomplete uses of rule $R3$.

In any environment, space critical or not, it is strongly recommended that the above be carried out for some threshold point. Otherwise, the code might, under some circumstances, become completely unwound with growth exponential in the number of conditional jumps and rejoins.

For blocks in which we do not do the above, much of the arbitrarily wasted space may be recoverable by an inelegant "hunt-and-peck" method. In general, we may examine the already formed schedule and identify conditional jumps which are above MOP's, which will thus have to be copied into the branch, and MOP's which were below a joining point but are now above a legal rejoin. Since list scheduling tends to push MOP's up to the top of a schedule, holes might exist for these MOP's below where they were placed. We examine all possible moves into such holes and pick those with the greatest profit. Making such an improvement may set off a string of others; the saving process stops when no more profitable moves remain. This is explained in more detail in [2].

Task Lifting

Before compacting a trace which branches off an already compacted trace, it may be possible to take MOP's which are free at the top of the new trace and move them into holes in the schedule of the already compacted trace, using motion rule $R6$. If this is done, the MOP's successors may become free at the top and movable. Reference [2] contains careful methods of doing this. This is simply the automated menu approach which we have tried to avoid, used only at the interface of two of the traces.

Application of the Other Menu Rules

Trace scheduling allows the scheduler to choose code motion from among the rules $R1$, $R3$, $R5$, and $R6$ without any special reference to them. We can also fit rules $R2$ and $R4$ into this scheme, although they occur under special circumstances and are not as likely to be as profitable. Rule $R2$ has the effect of permitting rejoins to occur higher than the bookkeeping rules imply. Specifically, we can allow rejoins to occur above MOP's which were earlier than the old rejoin, but are duplicated in the rejoining trace. This is legal if the copy in the rejoining trace is free at the bottom of the trace. When we do rejoin above such a MOP we remove the copy from the rejoining trace. This may cause some of its predecessors to be free at the bottom, possibly allowing still higher rejoins.

In the example, suppose MOP's 5 and 16 were the same. We could then have rejoined $B4$ to the last microinstruction, that containing MOP's 5 and 14, and deleted MOP 16 from block $B4$. The resultant compaction would have been one cycle shorter in terms of space used, but would have had the same running time.

Similarly, rule $R4$ applies if two identical MOP's are both free at the tops of both branches from a conditional. In that case we do not draw an edge from the conditional jump to the on the trace copy, even if the DAG definition would require it. If the copy is scheduled above the conditional jump, rule $R4$ allows us to delete the other copy from the off the trace branch, but any other jumps to that branch must jump to a new block containing only that MOP.

B. Extensions of the Model to More Complex Constructs

Having used a simplified model to explain trace scheduling, we now discuss extensions which will allow its use in many microcoding environments. We note, though, that no tractable model is likely to fit all machines. Given a complex enough micromachine, some idioms will need their own extension of the methods similar to what is done with the extensions in this section. It can be hoped that at least some of the idiomatic nature of microcode will lessen as a result of lowered hardware costs. For idioms which one is forced to deal with, however, very many can be handled by some special case behavior in forming the DAG (which will not affect the compaction methods) combined with the grouping of some seemingly independent MOP's into single multicycle MOP's, which can be handled using techniques explained below.

In any event, we now present the extensions by first explaining why each is desirable, and then showing how to fit the extension into the methods proposed here.

Less Strict Edges

Many models that have been used in microprogramming research have, despite their complexity, had a serious deficiency in the DAG used to control data dependency. In most machines, master–slave flip-flops permit the valid reading of a register up to the time that register writes occur, and a write to a register following a read of that register may be done in the same cycle as the read, but no earlier. Thus, a different kind of precedence relation is often called for, one that allows the execution of a MOP no earlier than, but possibly in the same cycle as its predecessor. Since an edge is a pictorial representation of a "less than" relation, it makes sense to consider this new kind of edge to be "less than or equal to," and we suggest that these edges be referred to as such. In pictures of DAG's we suggest that an equal sign be placed next to any such edge to distinguish it from ordinary edges. In writing we use the symbol $\underline{\ll}$. (An alternative way to handle this is via "polyphase MOP's" (see below), but these edges seem too common to require the inefficiencies that polyphase MOP's would require in this situation.) As an example, consider a sequence of MOP's such as

MOP 1: $A := B$

MOP 2: $B := C$.

MOP's 1 and 2 would have such an edge drawn between them, since 2 could be done in the same cycle as 1, or any time later. More formally, we now give a definition of the DAG on a set of MOP's. This will include the edges we want from conditional jump MOP's, as described previously.

Rules for the formation of a partial order on MOP's:

Given: A trace of MI's $m_1, m_2, \cdots, m_n$.

For each MI m_i, three sets of registers, *readregs*(m_i), *writeregs*(m_i), and *condreadregs*(m_i)

as defined previously.

For each pair of MI's m_i, m_j with $i < j$ we define edges, as follows.

1) If $readregs(m_i) \cap writeregs(m_j) \neq \emptyset$, then $m_i \underline{\ll} m_j$ unless for each register $r \in readregs(m_i) \cap writeregs(m_j)$ there is a k such that $i < k < j$ and $r \in writeregs(m_k)$.

2) If $writeregs(m_i) \cap readregs(m_j) \neq \emptyset$, then $m_i \ll m_j$ unless for each register $r \in writeregs(m_i) \cap readregs(m_j)$ there is a k such that $i < k < j$ and $r \in writeregs(m_k)$.

3) If $writeregs(m_i) \cap writeregs(m_j) \neq \emptyset$, then $m_i \ll m_j$ unless for each register $r \in writeregs(m_i) \cap writeregs(m_j)$ there is a k such that $i < k < j$ and $r \in writeregs(m_k)$.

4) If $condreadregs(m_i) \cap writeregs(m_j) \neq \emptyset$, then $m_i \ll m_j$ unless for each register $r \in condreadregs(m_i) \cap writeregs(m_j)$ there is a k such that $i < k < j$ and $r \in writeregs(m_k)$.

5) If by the above rules, both $m_i \ll m_j$ and $m_i \underline{\ll} m_j$, then we write $m_i \ll m_j$.

The algorithm given in Section II for list scheduling would have to be changed in the presence of equal edges, but only in that the updating of the data ready set would have to be done as a cycle was being scheduled, since a MOP may become data ready during a cycle. Many of the simulation experiments reported upon in Section II were done both with and without less than or equal edges; no significant difference in the results was found.

Many Cycle and Polyphase MOP's

Our model assumes that all MOP's take the same amount of time to execute, and thus that all MOP's have a resource and dependency effect during only one cycle of our schedule. In many machines, though, the difference between the fastest and slowest MOP's is great enough that allotting all MOP's the same cycle time would slow down the machine considerably. This is an intrinsic function of the range of complexity available in the hardware at the MOP level, and as circuit integration gets denser will be more of a factor.

The presence of MOP's taking $m > 1$ cycles presents little difficulty to the within block list scheduling methods suggested here. The simple priority calculations, such as highest levels, all extend very naturally to long MOP's; in particular, one can break the MOP into m one cycle sub-MOP's, with the obvious adjustments to the DAG, and calculate priorities any way that worked for the previous model. List scheduling then proceeds naturally: when a MOP is scheduled in cycle C, we also schedule its constituent parts in cycles $C + 1, C + 2, \cdots, C + m - 1$. If in one of these cycles it resource conflicts with another long MOP already in that cycle, we treat it as if it has a conflict in the cycle in which we are scheduling it. The resource usages need not be the same for all the cycles of the long MOP, it is a straightforward matter to let the resource vector have a second dimension.

Trace scheduling will have some added complications with long MOP's. Within a trace being scheduled the above comments are applicable, but when a part of a long MOP has to be copied into the off the trace block, the rest of the MOP will have to start in the first cycle of the new block. The information that this is required to be the first MOP will have to be passed on to the scheduler, and may cause extra instructions to be generated, but can be handled in a straightforward enough way once the above is accounted for.

In what are called polyphase systems, the MOP's may be further regarded as having submicrocycles. This has the advantage that while two MOP's may both use the same resource, typically a bus, they may use it during different submicrocycles, and could thus be scheduled in the same cycle. There are two equally valid ways of handling this; using either of the methods presented here is quite straightforward. One approach would be to have the resource vector be quite complicated, with the conflict relation representing the actual (polyphase) conflict. The second would be to consider each phase of a cycle to be a separate cycle. Thus, any instruction which acted over more than one phase would be considered a long MOP. The fact that a MOP was only schedulable during certain phases would be handled via extra resource bits or via dummy MOP's done during the earlier phases, but with the same data precedence as the MOP we are interested in.

Many machines handle long MOP's by pipelining one cycle constituent parts and buffering the temporary state each cycle. Although done to allow pipelined results to be produced one per cycle, this has the added advantage of being straightforward for a scheduler to handle by the methods presented here.

Compatible Resource Usages

The resource vector as presented in Section II is not adequate when one considers hardware which has mode settings. For example, an arithmetic-logic unit might be operable in any of 2^k modes, depending on the value of k mode bits. Two MOP's which require the ALU to operate in the same mode might not conflict, yet they both use the ALU, and would conflict with other MOP's using the ALU in other modes. A similar situation occurs when a multiplexer selects data onto a data path; two MOP's might select the same data, and we would say that they have compatible use of the resource. The possibility of compatible usage makes efficient determination of whether a MOP conflicts with already placed MOP's more difficult. Except for efficiency considerations, however, it is a simple matter to state that some resource fields are compatible if they are the same, and incompatible otherwise.

The difficulty of determining resource conflict can be serious, since many false attempts at placing MOP's are made in the course of scheduling; resource conflict determination is the innermost loop of a scheduler. Even in microassemblers, where no trial and error placement occurs and MOP's are only placed in one cycle apiece, checking resource conflict is often a computational bottleneck due to field extraction and checking. (I have heard of three assemblers with a severe efficiency problem in the resource legality checks, making them quite aggravating for users. Two of them were produced by major manufacturers.) In [2] a method is given to reduce such conflict tests to single long-word bit string operations, which are quite fast on most machines.

Microprogram Subroutines

Microprogram subroutine calls are of particular interest because some of the commercially available microprogram sequencer chips have return address stacks. We can therefore expect more microcodable CPU's to pass this facility on to the user.

Subroutines themselves may be separately compacted, just as loops are. While motion of MOP's into and out of subroutines may be possible, it seems unlikely to be very desirable.

Of greater interest is the call MOP; we want to be able to allow MOP's to move above and below the call. For MOP's to move above the call, we just treat it like a conditional jump, and do not let a MOP move above it if the MOP writes a register which is live at the entrance to the subroutine and is read in the subroutine. Motion below a call is more complicated. If a task writes or reads registers which are not read or written (respectively) in the subroutine, then the move to below the call is legal, and no bookkeeping phase is needed. In case such a read or write occurred, we would find that we would have to execute a new "off the trace" block which contained copies of the MOP's that moved below the call, followed by the call itself. A rejoin would be made below the MOP's which moved down, and MOP's not rejoined to would be copied back to the new block after the call. In the case of an unconditional call, the above buys nothing, and we might as well draw edges to the call from tasks which would have to be copied. If the call is conditional and occurs infrequently, then the above technique is worthwhile. In that case the conditional call would be replaced by a conditional jump to the new block. Within the new block, the subroutine call would be unconditional; when the new block is compacted, edges preventing the motion of MOP's below the call in the new block would appear.

V. General Discussion

Compaction is important for two separate reasons, as follows.

1) Microcode is very idiomatic, and code development tools are necessary to relieve the coder of the inherent difficulty of writing programs.

2) Machines are being produced with the potential for very many parallel operations on the instruction level–either microinstruction or machine language instruction. This has been especially popular for attached processors used for cost effective scientific computing.

In both cases, compaction is difficult and is probably the bottleneck in code production.

One popular attached processor, the floating point systems AP-120b and its successors, has a floating multiplier, floating adder, a dedicated address calculation ALU, and several varieties of memories and registers, all producing one result per cycle. All of those have separate control fields in the microprogram and most can be run simultaneously. The methods presented here address the software production problem for the AP-120b.

Far greater extremes in instruction level parallelism are on the horizon. For example, Control Data is now marketing its Advanced Flexible Processor [15], which has the following properties:

- 16 independent functional units, their plan is for it to be reconfigurable to any mix of 16 from a potentially large menu;
- all 16 functional units cycle at once, with a 20 ns cycle time;
- a large crossbar switch (16 in by 18 out), so that results and data can move about freely each cycle;
- a 200 bit microinstruction word to drive it all in parallel.

With this many operations available per instruction, the goal of good compaction, particularly global compaction, stops being merely desirable and becomes a necessity. Since such machines tend to make only one (possibly very complex) test per instruction, the number of instruction slots available to be packed per block boundary will be very large. Thus, such machines will contain badly underused instructions unless widely spread source MOP's can be put together. The need to coordinate register allocation with the scheduler is particularly strong here.

The trend towards highly parallel instructions will surely continue to be viable from a hardware cost point of view; the limiting factor will be software development ability. Without effective means of automatic packing, it seems likely that the production of any code beyond a few critical lines will be a major undertaking.

Acknowledgment

The author would like to thank R. Grishman of the Courant Institute and J. Ruttenberg and N. Amit of the Yale University Department of Computer Science for their many productive suggestions. The author would also like to thank the referees for their helpful criticism.

References

[1] D. Landskov, S. Davidson, B. Shriver, and P. W. Mallet, "Local microcode compaction techniques," *Comput. Surveys*, vol. 12, pp. 261–294, Sept. 1980.

[2] J. A. Fisher, "The optimization of horizontal microcode within and beyond basic blocks: An application of processor scheduling with resources," Courant Math. Comput. Lab., New York Univ., U.S. Dep. of Energy Rep. COO-3077-161, Oct. 1979.

[3] P. W. Mallett, "Methods of compacting microprograms," Ph.D. dissertation, Univ. of Southwestern Louisiana, Lafayette, Dec. 1978.

[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.

[5] E. G. Coffman, Jr., Ed., *Computer and Job Shop Scheduling Theory*. New York: Wiley, 1976.

[6] M. J. Gonzalez, Jr., "Deterministic processor scheduling," *Comput. Surveys*, vol. 9, pp. 173–204, Sept. 1977.

[7] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Commun. Ass. Comput. Mach.*, vol. 17, pp. 685–690, Dec. 1974.

[8] C. C. Foster and E. M. Riseman, "Percolation of code to enhance parallel dispatching and execution," *IEEE Trans. Comput.*, vol. C-21, pp. 1411–1415, Dec. 1972.

[9] E. M. Riseman and C. C. Foster, "The inhibition of potential parallelism by conditional jumps," *IEEE Trans. Comput.*, vol. C-21, pp. 1405–1411, Dec. 1972.

[10] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*. Reading, MA: Addison-Wesley, 1974.

[11] M. S. Hecht, *Flow Analysis of Computer Programs*. New York: Elsevier, 1977.

[12] S. Dasgupta, "The organization of microprogram stores," *ACM Comput. Surveys*, vol. 11, pp. 39–65, Mar. 1979.

[13] M. Tokoro, T. Takizuka, E. Tamura, and I. Yamaura, "Towards an efficient machine-independent language for microprogramming," in *Proc. 11th Annu. Microprogramming Workshop, SIGMICRO*, 1978, pp. 41–50.

[14] G. Wood, "Global optimization of microprograms through modular control constructs," in *Proc. 12th Annu. Microprogramming Workshop, SIGMICRO*, 1979, pp. 1–6.

[15] CDC, *Advanced Flexible Processor Microcode Cross Assembler (MICA) Reference Manual*, 1980.

Joseph A. Fisher (S'78) was born in the Bronx, NY, on July 22, 1946. He received the A.B. degree in mathematics from New York University, New York City, in 1968 and the M.S. and Ph.D. degrees in computer science from the Courant Institute, New York, NY in 1976 and 1979, respectively.

Since 1979 he has been an Assistant Professor of Computer Science at Yale University. His research interests include microprogram optimization, microcode development environments, computer architecture, and parallel processing.