Local Microcode Compaction Techniques

DAVID LANDSKOV, SCOTT DAVIDSON, AND BRUCE SHRIVER

University of Southwestern Louisiana, Computer Science Department, P.O. Box 44330 U.S.L., Lafayette, Louisiana 70504

PATRICK W. MALLETT

Computer Sciences Corporation, 6565 Arlington Boulevard, Falls Church, Virginia 22046

Microcode compaction is an essential tool for the compilation of high-level language microprograms into microinstructions with parallel microoperations. Although guaranteeing minimum execution time is an exponentially complex problem, recent research indicates that it is not difficult to obtain practical results. This paper, which assumes no prior knowledge of microprogramming on the part of the reader, surveys the approaches that have been developed for compacting microcode. A comprehensive terminology for the area is presented, as well as a general model of processor behavior suitable for comparing the algorithms. Execution examples and a discussion of strengths and weaknesses are given for each of the four classes of local compaction algorithms: linear, critical path, branch and bound, and list scheduling. Local compaction, which applies to jump-free code, is fundamental to any compaction technique. The presentation emphasizes the conceptual distinction between data dependency and conflict analysis.

Keywords and Phrases: compaction, data dependency, horizontal architecture, horizontal optimization, microcode compaction, microinstruction, microprogram, parallel, resource conflict, scheduling

CR Categories: 4.12, 6.21, 6.33

INTRODUCTION

As the use of microprogramming increases, it becomes costly to write microprograms in unstructured unwieldy languages. The same pressures that led to the widespread acceptance of conventional high-level languages now apply to microprogramming [DAVI78]. The time is long overdue for the development of machine-independent, high-level microprogramming language compilers.

Microprogrammable processors that allow simultaneous control of several hardware resources present special challenges to the implementor of a high-level language compiler. These horizontal processors must have their microprograms compacted in order to run efficiently. Compaction involves choosing from the possible arrangements of concurrent activities one that will minimize the execution time of the microprogram and possibly its size as well.

The first step in such a compaction is the division of the program into branch-free segments. The analysis, or local compaction, of one of these segments is an expo-

CONTENTS

nentially complex problem, i.e., one that is time consuming to calculate. In the past, because of this complexity, it was feared that a compiler capable of generating efficient horizontal microcode would run too slowly to be usable. Because microprograms are at the lowest programmable level, their efficiency directly affects the efficiency of the entire system.

Recent research [TOKO77, MALL78, WOOD79, FISH79] indicates that local compaction algorithms can be practical. Despite the theoretical computational complexity, optimal or near-optimal results can be found in a reasonable (i.e., nonexponential) amount of time. Thus one of the major obstacles to practical microcode compilers for horizontal machines is surmountable.

This paper surveys the various approaches that have been taken for compacting microcode [RAMA74, TABA74, TSUC74, YAU74, DEWI75, AGER76, DASG76, MALL78, WOOD78, FISH79]. Clear definitions are presented for the terms required to understand this area, since the variations in terminology and the lack of a comprehensive vocabulary have constituted a major problem. This paper presents a unifying terminology for studying the issues involved and applies this terminology to the different approaches.

The development of compaction algorithms in existing literature is based on a number of different models of the processor environment. These differences make comparison of the algorithms difficult, and sometimes model differences are confused with the differences in the algorithms themselves. This paper presents a model which incorporates the major features of existing models, yet avoids unnecessary details. All of the algorithms presented are explained in terms of this general model.

Local compaction is a fundamental part of any compaction process. There are four major classes of algorithms for locally compacting microcode: linear analysis, critical path, branch and bound, and list scheduling. Each of these classes is explained using the terms defined in this paper. A common example illustrates the execution of each kind of algorithm. There are also detailed presentations of two nontrivial support algorithms which are rarely explained in the literature.
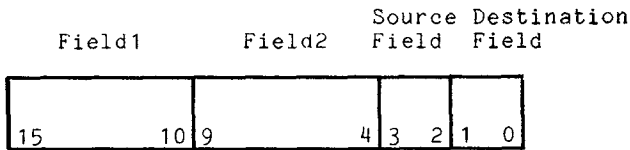
Definition of the Problem

A microprogram is a sequence of **microinstructions (MIs)**. A microprogram is stored in a memory, often a special memory called the control store, from which the instructions are executed one at a time. During its execution, an MI is the **control word** for its machine.

Each separate machine activity specified in an MI is called a **microoperation (MO)**. Thus an MI can be characterized as a set of MOs. A **field** is a collection of control word bits that controls a primitive machine activity. An MO requires one or more fields in order to execute. The format of a control word determines how many and which MOs can be placed together in an MI. Figure 1 shows the relationship between fields and microoperations in the control word organization for a hypothetical machine. If only one or a few MOs can fit into an MI, the

```
                                    Source Destination
        Field1          Field2      Field  Field

     ┌──────────────┬────────────┬─────┬─────┐
     │              │            │     │     │
     │15         10 │9         4 │3   2│1   0│
     └──────────────┴────────────┴─────┴─────┘
```

(a)

	Field1	Field2	Source	Destination
mo_1:	ADD			d
mo_2:	MOVE		s	
mo_3:		SHIFTL	n	
mo_4:		SHIFTR	n	

where $s, d \in$ AC (accumulator), RA (reg A), RB, or SH (shifter); $n \in 1. .4$.

(b)

```
┌──────────────┬────────────┬─────┬─────┐
│              │            │     │     │
│    MOVE      │   empty    │ AC  │ SH  │
│              │            │     │     │
└──────────────┴────────────┴─────┴─────┘

┌──────────────┬────────────┬─────┬─────┐
│              │            │     │     │
│    ADD       │   SHIFTL   │  3  │ RB  │
│              │            │     │     │
└──────────────┴────────────┴─────┴─────┘
```

(c)

FIGURE 1. Example control word organization. (a) Microinstruction format. (b) Partial list of microoperations. (c) Two possible microinstructions.

machine is said to have a **vertical architecture**. Otherwise it is said to have a **horizontal architecture**. A more detailed discussion of control word format can be found in DASG79.

The microcode **compaction problem** is as follows. Suppose that for a particular machine, the **host machine**, we are given a microprogram expressed as a sequence of MOs. These MOs are to be placed into MIs so that the microprogram execution time is minimized. This must be done under the restriction that the resulting sequence of MIs must be semantically equivalent to the original sequence of MOs. "Semantically equivalent" means that if both sequences are executed, the same input always results in the same output. The original sequence of MOs is not executable as it stands but can easily be made executable by placing each MO in a separate MI. Some MOs may have to be placed in the same MI as a previous MO, as explained in the section on coupling. Informally, the problem is to "compact" the program into a small memory space.

We discuss microcode compaction for horizontal architectures. Although compaction can be performed for any architecture, including a vertical one, the compaction problem is only interesting if a useful concurrency of MO execution can be achieved. Not only are vertical architectures limited in their potential concurrency, but this limitation is one of the justifications of their design. Vertical machines are easier to microprogram precisely because they avoid the time-consuming and error-prone procedure of manually analyzing concurrency.

In order to analyze possible concurrent activity, the microprogram is divided into straight-line microcode sections.

Definition. A **straight-line microcode section (SLM)** is an ordered collection of MOs with no entry points, except at the beginning, and no branches, except possibly at the end.
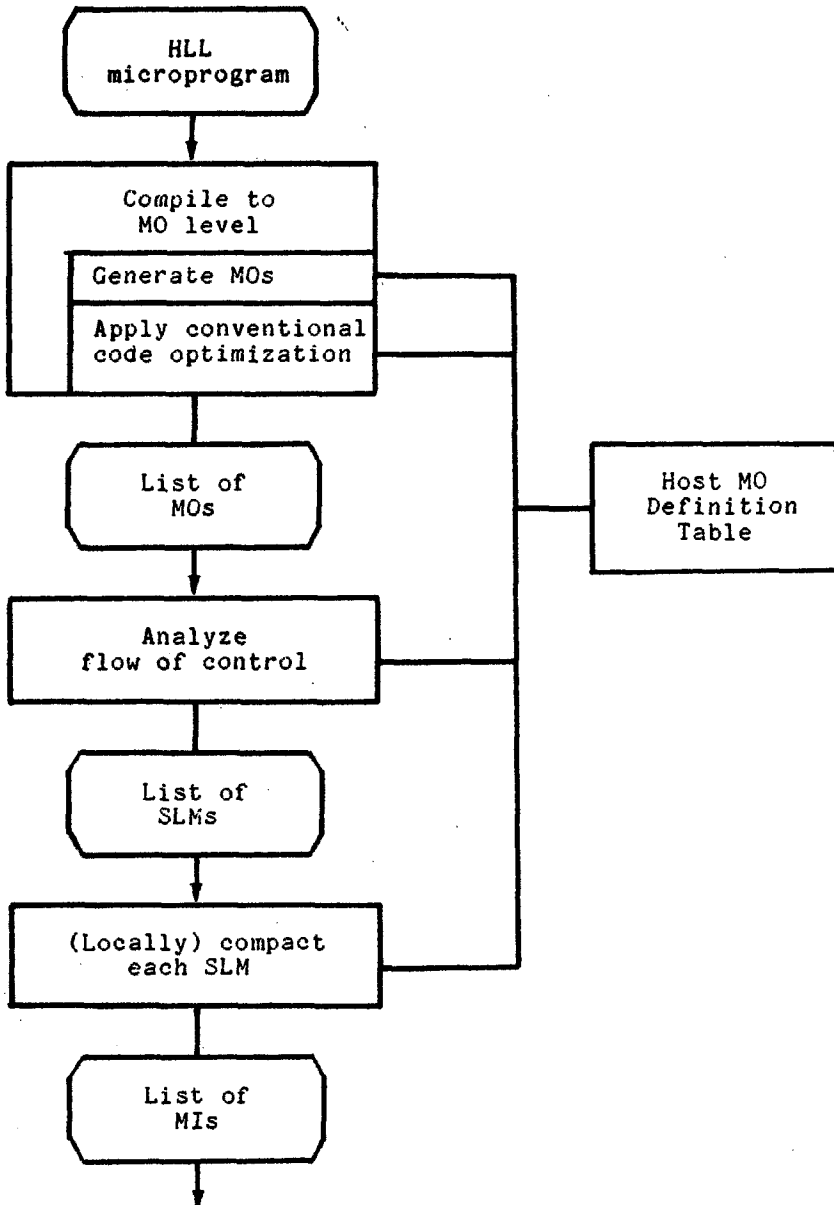
FIGURE 2.   One possible microcode compilation system.

SLMs are also known as basic blocks. Within an SLM, minimizing execution time is achieved by minimizing the number of compacted microinstructions. Analysis of a single SLM is called **local analysis**, and of more than one SLM, **global analysis**.

In global analysis, minimizing the number of microinstructions does not necessarily minimize execution time, since some SLMs may be executed many more times than others. Global analysis is very much an active research problem [TOKO78, DASG79, WOOD79, FISH79]. Interesting approaches based on treating a compacted SLM as a primitive in a more global SLM are found in WOOD79 and FISH79. Our paper is confined to the local analysis problem, which is examined in detail.

The role of local compaction analysis in a high-level microprogramming language

FIGURE 3. Subalgorithm modules used by compaction algorithms. BuildDDG and FormCIs are used in most of the approaches.

translation system [cf. MALL75] is shown in Figure 2. Two relatively distinct analyses must be performed as part of the local compaction process (see Figure 3). The way in which data are passed from MO to MO forces some MOs to be kept in the order in which they appeared in the original SLM. **Data dependency analysis** decides the partial ordering. **Conflict analysis** determines whether two MOs can fit into the same MI without conflicting over a hardware resource.
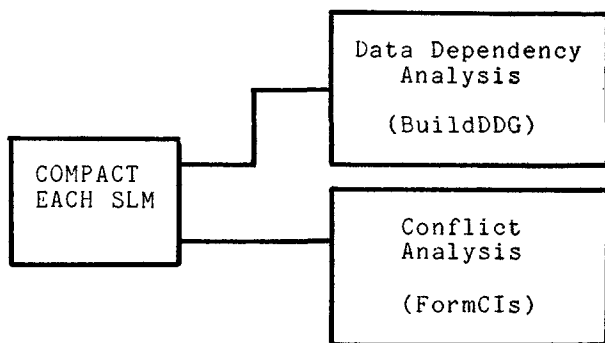
Many of the optimization techniques of conventional compilers are applicable to microprograms—which is not surprising, since a sequence of MOs strongly resembles a conventional machine language program [KLEI71]. These techniques consist primarily of code transformations that reduce the number or the execution time of MOs in sequential form. For the rest of our discussion we assume that any code transformations have already been applied.

Sometimes compaction is termed horizontal optimization, but this is misleading because conventional compiler optimization is different from compaction and because compaction can reduce microprogram size and execution time without necessarily minimizing them. Horizontal improvement would be a more accurate term.

The Minimum Manhole Shifts Analogy

The reader can gain an appreciation for some of the issues involved by considering the following analogy, in which the scheduling problems of an underground construction project are compared to those of local

microcode compaction. The appropriate compaction term is placed in parentheses following discussion in the applicable analogy.

Both compaction and this analogy are examples of job shop scheduling problems [COFF76]. Compaction also bears a direct resemblance to the processor scheduling problem [GONZ77].

The Analogy

The foreman of a crew working in a manhole has a big job to do. The job entails a large number of specific tasks, but there is no shortage of workers. There is, however, a limited amount of space in the manhole and a limited number of tools. The foreman's problem is how to perform a given job in the minimum number of shifts.

The analogy thus far:

The manhole: an MI or a control store word.
The shift: one MI cycle.
The job: an SLM.
The task and its associated worker: an MO.
The tool: a processor's operational unit (ALU, BUS, etc.)

Some workers' tasks depend on the completion of tasks by other workers. The foreman must make a list of which tasks depend on which other tasks and not send a worker down into the manhole before the necessary prerequisite tasks have been completed (*data dependency*).

Assume for now that a worker's task always takes exactly one shift to complete (*monophase MI*). Thus a worker depend-

ing on the job done by another should not go into the manhole until the shift after that of the first worker. Of course, the tasks of two workers do not always involve a dependency. If this is the case, they can go down into the manhole together, assuming there are no other problems between them (*data independence*).

When can workers with independent tasks not be sent down into the manhole together? There are two possibilities. First, they may require the same tool to do their jobs. If there is only one such tool, one worker must wait for the next shift (*resource unit conflict*). Second, the two workers may not fit into the manhole together. This can happen if the two workers must work in the same place (at an exposed water pipe, for instance) in the manhole (*microoperation field conflict*).

Now, what if the assumption that each worker needs an entire shift is unrealistic? Suppose that some workers need only part of a shift to perform their tasks (*polyphase MIs*). A union rule states that a worker must stay in the manhole during the entire eight-hour shift, but also that a worker may be idle some of that time. Consider two workers with independent tasks who need the same tool. If they can work at separate times, one can use the tool for the first few hours, the other, for the remaining hours. Thus two workers who need the same tool during different parts of the shift can be sent into the manhole together (*resource unit compatibility*). They still must fit together in the manhole, however (*microoperation field compatibility*).

Suppose worker 2's task can only be performed after the completion of worker 1's task. If worker 1 starts in the morning and takes two hours to finish and worker 2 needs four hours, they can be sent into the manhole together, worker 2 waiting until worker 1 is finished. Thus, assuming that the workers can fit together in the manhole, they can be sent down together (*weakly dependent MOs*).

Now, let worker 3 perform a delicate task, such as cleaning a joint in preparation for welding. Worker 4 does the welding. If the cleaned joint is left overnight, it gets dirty again, and the job must be redone. Therefore worker 4 must be sent into the manhole

on the same shift as worker 3 (*coupled MOs*). The foreman decides to simplify the analysis of task dependencies by considering two or more inseparable tasks as one task with multiple workers (*MO bundles*).

Some tools may be multifunctional. A drill can be used for drilling or modified and used for polishing, but a polisher can be used only for polishing. If worker 5 needs a drill and worker 6 a polisher, worker 6 should be given the polisher, not the drill/polisher. The foreman should tell the workers which tools to use, thus eliminating this kind of conflict (*versions of resource units*). Similarly, if worker 7 can work either in a corner or by a wall (needing access to pipes, say), and worker 8 can work only on a junction box in a corner, worker 7 should be told to work by a wall (*versions of microoperation fields*).

These are some of the issues the foreman must contend with when attempting to minimize the number of shifts required to complete a job.

The manhole analogy makes the scheduling nature of the local compaction problem clear. The terms in parentheses are explained in the appropriate sections of this paper. The reader should refer to this analogy when studying these terms.

How This Paper Is Organized

Many of the sections in this paper are explanations of algorithms. Although the details of an algorithm can be skipped without a serious loss of understanding of subsequent sections, it is crucial to understand the problem that each algorithm is designed to solve.

Section 1 analyzes the data dependency problem and presents an algorithm for building a data dependency graph from an SLM in sequential form. Section 2 explains a model of processor behavior and shows how it can be used to detect conflicts in resource usage. Section 3 examines an algorithm that uses this model to form microinstructions. Finally, Section 4 presents examples from each of the four classes of compaction algorithms and discusses the computational complexity of each class.

Little or no prior knowledge of micropro-

gramming is needed in order to understand this article.

1. DATA DEPENDENCY ANALYSIS

Data dependency analysis is the first step performed in the local compaction process. It is based on an examination of the input and output resources of each microoperation.

1.1 The Definition of Data Dependency

Most of the microoperations of a given machine operate on **registers**. A register whose value is used by an MO is called an **input** storage resource or an input operand. Similarly, a register whose value is changed by an MO is called an **output** storage resource or an output operand. ("Source" is often used instead of "input" and "sink" or "destination" instead of "output.")

As long as the number of variables used in the entire microprogram (not just one SLM) does not exceed the number of registers available, a register is essentially a variable. Assume for now that this is the case; the problem of reallocating registers is discussed in Section 4.6.

Given an SLM to be compacted, the final list of MIs must be "semantically equivalent" to the SLM in the sense that both must produce the same results when given the same input values. If they are not semantically equivalent, data integrity has been violated. Some of the MOs cannot change their order of execution without producing different answers. In particular, the order of two MOs cannot be changed if they satisfy the following definition.

Definition. Two MOs, mo_i and mo_j, have a **data interaction** if they satisfy any of the following conditions (assuming that mo_i precedes mo_j in the original SLM):

(1) An output resource of mo_i is also an input resource of mo_j (if mo_j were first, it would have an old value in its input resource, one that should have been updated by mo_i but was not).

(2) An input resource of mo_i is also an output resource of mo_j (if mo_j were first, it would be able to change the value that mo_i was expecting as input before mo_i had a chance to use it).

(3) An output resource of mo_i is also an output resource of mo_j (if mo_j were first, mo_i would be able to overwrite mo_j's output value, when mo_i's value is the one that should remain after both MOs are finished).

The definition of data interaction can be applied to any two MOs without reference to their order in the original SLM. Section 2 presents a representation for the input and output resources of MOs that allows data interaction to be tested by examining set intersections.

The remainder of our development of data dependency analysis rests on the following assertion.

A compacted list of MIs will be semantically equivalent to its original SLM if, for every two MOs in the MI list that have a data interaction, the MO occurring earlier in the SLM finishes with each of the resources causing the data interaction before the later MO starts to use it.

Several definitions are based on this assertion, as shown in the following.

Definition. Given two MOs, mo_i and mo_j, where mo_i precedes mo_j in the SLM, then these MOs are **order preserving** if their execution in the same MI obeys the following rule (assume it is otherwise possible): For each resource causing a data interaction between them, mo_i finishes with that resource before mo_j starts to use it.

If two MOs are order preserving, they can be in the same MI without violating data integrity. If an MO is order preserving with respect to every MO in an MI, it is order preserving with respect to that MI.

The next definition defines a partial order over the MOs.

Definition. Given two MOs, mo_i and mo_j, where mo_i precedes mo_j in the original SLM, mo_j is **directly data dependent** on mo_i (written mo_i **ddd** mo_j) if the two MOs have a data interaction and if there is no sequence of MOs, $mo_{k1}, mo_{k2}, \ldots, mo_{kn}, n \geq 1$, such that mo_i **ddd** mo_{k1}, mo_{k1} **ddd** $mo_{k2}, \ldots, mo_{k(n-1)}$ **ddd** mo_{kn}, mo_{kn} **ddd** mo_j.

The second part of the definition ensures that two directly data-dependent MOs will

have no "chain" of directly data-dependent MOs between them.

Data dependency is the transitive closure of the direct data dependency relation.

Definition. Given two MOs, mo_i and mo_j, mo_j is **data dependent** on mo_i (written mo_i **dd** mo_j) if

$$mo_i \text{ ddd } mo_j$$

or if there exists an MO, mo_k, such that

$$mo_i \text{ ddd } mo_k \quad \text{and} \quad mo_k \text{ dd } mo_j.$$

If mo_i **dd** mo_j, then mo_j cannot execute before mo_i without violating data integrity. Usually they cannot execute in the same MI either; this situation is discussed in the next section. Two MOs that are not data dependent are said to be **data independent**. It should be clear that data independence implies order preservation.

Suppose a list of MIs is being constructed from an SLM and the MOs in the SLM are being considered one at a time. The data dependency concept is used to determine whether adding a particular MO to a particular MI in the list will violate data integrity. If the answer is no, the MO is said to be **data available** with respect to that MI. Data availability is discussed more formally in the next section.

The direct data dependency relation defines a partial order over the MOs of an SLM. The representation of this ordering in graph form is called a **data dependency graph (DDG)**. Each node on a DDG, node i say, corresponds to a unique MO in the SLM, mo_i. If there is a link from i to j on the graph, then mo_i **ddd** mo_j. The definition of direct data dependency ensures that this link is the only path in the graph from i to j. Figure 4 shows a simple DDG where mo_1

$$
\begin{array}{c}
(1) \\
/ \\
(2) \\
\backslash \\
(3)
\end{array}
$$
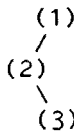
FIGURE 4. A data dependency graph. Nodes 1 and 3 cannot be linked.

ddd mo_2 and mo_2 **ddd** mo_3. There cannot be a link between node 1 and node 3 because they are already linked indirectly.

Many compaction algorithms use a DDG of the SLM as input (see Section 4). A well-designed microcode compilation system might have the code generator produce the MOs in graph form. Otherwise, the first step of compaction is to form a DDG from the SLM.

1.2 Extending the Data Dependency Concept

There are several machine features that can be incorporated into data dependency analysis.

1.2.1 Finishing with a Resource Before the End of a Cycle

Sometimes an MO does not affect machine resources during the entire time that the machine allows an MI to execute (i.e., the instruction cycle time). In Section 2, a notation for specifying the parts of an MI cycle in which an MO executes is developed. If mo_i and mo_j are in the same MI, and if mo_i finishes executing before mo_j begins, then data integrity is preserved even if mo_j is data dependent on mo_i. This motivates the following definition.

Definition. Given two MOs, mo_i and mo_j, then mo_j is **weakly dependent** on mo_i (mo_i **wd** mo_j) if mo_j is directly data dependent on mo_i, and if for every resource causing a data interaction between them, mo_i finishes with that resource before mo_j starts to use it.

Clearly, taking advantage of weak dependencies makes compactions with fewer MIs possible, since two MOs related by a weak dependency may be able to fit into the same MI. In DASG 76, the term "conditionally disjoint" is almost a synonym for weakly dependent (the difference in meaning arises from model differences and is not significant). If mo_i **ddd** mo_j, and it is known that mo_j is not weakly dependent on mo_i, then mo_j is **strongly dependent** on mo_i (mo_i **sd** mo_j). If MOs are placed in a list of MIs so that the MOs are order preserving,

$$mo_i \text{ sd } mo_j \quad \text{implies} \quad mo_i < mo_j$$
$$mo_i \text{ wd } mo_j \quad \text{implies} \quad mo_i \leq mo_j$$

The weak dependency concept allows a more precise analysis of some of the MO

relationships. One conclusion that can be drawn is

> If two MOs are data independent, or if one is weakly dependent on the other, then they are order preserving.

These are the only two conditions considered here which allow two MOs to fit into the same MI and still preserve data integrity.

The term **data available** applies when a list of MIs is considered instead of just an individual MI. We observe that

> Given an SLM of MOs, and a list of MIs constructed from some of these MOs, mo_i is called data available with respect to mi_j if
>
> (1) every MO in the SLM on which mo_i is data dependent appears in an MI which is above mi_j in the list, or appears in mi_j itself, and
> (2) mo_i is not strongly dependent on any MO in mi_j.

Item 2 is a rewording of order preservation. Notice that this definition applies even if mi_j is empty, or if the list of MIs has no other elements.

1.2.2 Transitory-Data Resources

One particular kind of weak dependency has a special importance. The example in Figure 5 depicts an MI sequence in which mo_1, mo_2, and mo_3 combine to take the data stored in register A1 and move (gate) it through latches 1 and 2 to register A2.

Registers A1 and A2 are examples of **static** resources. A static storage resource is one whose contents are maintained indefinitely until explicitly overwritten by the execution of an MO. A **transitory-data** storage resource is any storage resource that is not static, that is, one whose contents can become undefined at the termination of an MI in which it has been given a value. Latches are examples of transitory-data resources.

Two MOs are said to be **directly coupled** if one of them passes data to the other through a transitory-data resource. Two MOs are said to be **coupled** if a sequence of MOs exists such that one of the two MOs

mo_1:	Input = A1, Output = Latch1
⋮	(MOs activating devices that use the contents of Latch1)
mo_2:	Input = Latch1, Output = Latch2
⋮	(MOs activating devices that use the contents of Latch 2)
mo_3:	Input = Latch2, Output = A2
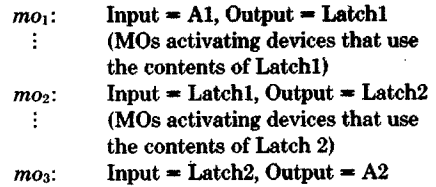
FIGURE 5. MOs coupled through transitory-data resources.

is the first element of the sequence, the other is the last, and each element of the sequence is directly coupled to its adjacent elements. Coupled MOs must be placed in the same MI to work properly. Coupling can occur in a variety of microprogram instruction sets [AGRA 76, SHRI 73].

It is the authors' experience that the only way to accommodate coupling in the compaction algorithms without causing serious confusion is to incorporate coupling into the definitions of the MO-to-MO relationships. This can be done using the concept of bundling.

Definition. **A microoperation bundle (MB)** is a set of MOs, all of which are coupled to one another.

Thus every MO in an MB must go into the same MI because of the nature of transitory-data resources.

The SLM is changed from a list of MOs to a list of MBs by putting coupled MOs into the same MB and by putting each uncoupled MO into its own MB. All of the relations over MOs can be defined over MBs in a straightforward manner. For example, mb_j is data dependent on mb_i (mb_i dd mb_j) if there exists an mo_i in mb_i and an mo_j in mb_j such that mo_i dd mo_j.

After the MB relationships are defined, the compaction algorithms can operate with MBs just as they previously operated with MOs. The nodes of a DDG are now MBs, and compaction algorithms change an SLM of MBs into a list of MIs. An MI is now a set of MBs.

For the rest of this paper we discuss compaction in terms of MBs. When relating this paper to other papers that do not discuss coupling, the reader may substitute the word microoperation for microoperation bundle.

1.2.3 Multicycle Operations

In many machines there are microoperations which need more than one instruction cycle to execute. A main memory reference is a common example of such a multicycle operation. An MB which is data dependent on an MB containing a multicycle operation must be **delayed** the proper number of cycles following the multicycle MI's start-up. This is easily accomplished in data dependency analysis by using **dummy microoperation bundles**. An operation requiring n cycles to execute is represented by a sequence of n MBs, each one data dependent on the previous MB in the sequence. The first MB has all of the input and output resources of the operation; the others are dummies which use the delayed output resources of the operation for both input and output. Figure 6 shows an example DDG where mb_1 has a three-cycle operation. Nodes 1a and 1b represent dummy MBs.
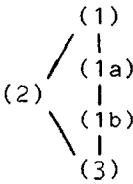

FIGURE 6. A data dependency graph with a three-cycle microoperation.

Although dummy MBs do not correspond to actual microoperations, an MI containing only dummy MBs has to be created as a no-operation (NOOP). Such a NOOP indicates that the machine has nothing else to do while waiting for the completion of a multicycle operation. Global analysis can eliminate NOOPs at the end of an SLM by moving the corresponding dummy MBs to all the SLMs that can execute immediately after this one [MALL 78].

1.3 Forming a Data Dependency Graph

Few of the compaction articles referenced by this paper define an algorithm for constructing a data dependency graph from a list of MBs (or MOs). Although algorithms of this kind exist, they are not widely known and have not been explained in terms of the microcode compaction problem. Thus the following algorithm should be of general interest.

The algorithm presented (BuildDDG) constructs the graph one MB at a time, starting with the first MB in the list and proceeding through the list in sequential order. Each MB is added as a new node to the graph formed by the MB's predecessors in the list. Then the graph is searched to find which nodes should be linked to this new node. We say that we are adding the "current MB" to the "current graph."

The directed graph is defined in a conventional way. "Node A is a parent of node B" means that "B is directly data dependent on A"; we write node A above node B with a link, or line, connecting them. The nodes without parents are the **roots**, and are drawn at the top of the graph. The nodes with no children are the **leaves**. A **path** is a sequence of distinct nodes, each of which is a parent to the next node in the sequence. Node A is an **ancestor** of node B if a path exists from A to B.

While adding the current MB to the graph, we test an MB already in the graph (a graph MB) to see if the current MB is data dependent on it. Whenever the test indicates a data dependency, a link is formed. Each graph MB needs to be tested once at most when adding the current MB. The data interaction part of the test can be performed by looking at only the two MBs involved. However the definition of direct data dependency tells us when data interaction does *not* imply direct data dependency. We can restate this part of the definition as follows:

> If the current microoperation bundle, mb_k, is directly data dependent on a microoperation bundle, mb_j, then mb_k cannot be directly data dependent on an ancestor of mb_j.

It immediately follows that a graph MB should not be linked if it is an ancestor of the current MB. Such links are not formed, and the unnecessary testing of such MBs for data interaction is avoided by observing the following rule:

Data Interaction Testing Rule.   A graph MB is tested for data interaction with the current MB if and only if all of the graph

```
Direct Data Dependencies:    1 ddd 2, 1 ddd 6, 4 ddd 5.
                             1 ddd 4, 3 ddd 6,
```

Current Graph	Current MB	Compare to	ddd?	Next Action
(1) / (2)	3	-	-	Add node for 3; Test next leaf (2).
		2	no	Check 2's parents.
		1	no	(No parent); Next MB.
(1) (3) / (2)	4	-	-	Add node for 4; Test next leaf (2).
		2	no	Check 2's parents.
		1	yes	Form link from 1 to 4; Test next leaf (3).
		3	no	(No parent); Next MB.
(1) (3) / \ (2) (4)	5	-	-	Add node for 5; Test next leaf (2).

		2	no	(2's parent has untested
				  child, 4);
				  Test next leaf (4).

		4	yes	Form link from 4 to 5; Test next leaf (3).
		3	no	(No parent); Next MB.
(1) (3) / \ (2) (4) | (5)	6	-	-	Add node for 6; Test next leaf (2).

		2	no	(2's parent has untested
				  child, 4);
				  Test next leaf (5).

		5	no	Check 5's parents.
		4	no	Check 4's parents (1's children now verified)
		1	yes	Form link from 1 to 6; Test next leaf (3).
		3	yes	Form link from 3 to 6; Stop (out of MBs).

```
      (1)    (3)
     / | \ /
    /  |  \              Resulting
 (2) (4) (6)     Data Dependency Graph
     |
    (5)
```

FIGURE 7.   Example of formation of a data dependency graph.

MB's children are verified nonancestors of the current MB.

By following this rule a positive test for data interaction implies direct data dependency.

Now an algorithm that searches for graph nodes on which the current MB is directly data dependent can be specified. A search progresses upward from each leaf, since each leaf automatically satisfies the testing rule. If the test of an MB for data interaction is positive, then a link is formed to the current MB. If the test is negative, then that MB has been verified as a nonancestor of the current MB. Each of its parents is informed that another child has been veri-

fied. Each parent is also checked to see if it now satisfies the testing rule. If it does, then the parent is tested. Searching ends when there are no more MBs to be tested.

Figure 7 shows the formation of a data dependency graph from an SLM of six MBs. The direct data dependencies which should be detected by the algorithm are listed at the top of the illustration. The algorithm starts by placing mb_1 in the graph. Then mb_2 becomes the current MB. Since mb_1 is a leaf of the current graph, it is tested against mb_2 for data interaction. The test is positive and a link is formed. No more MBs remain to be tested, so mb_3 becomes the new current MB. Refer to Figure 7 for a trace of the rest of the execution.

The resulting data dependency graph shows that it is not possible for mb_5 to be directly data dependent on mb_1 (since they are indirectly linked on the graph), even though there may be a data interaction between them. The algorithm did not perform the needless test for this data interaction. Notice also that the algorithm searches parents (and leaves) from left to right. The order in which parents are searched makes no difference. The searches could proceed in parallel.

No reference has been made yet to the data structure used to represent the graph. Any structure capable of representing a directed graph will work. An adjacency matrix where $DDG_{i,j}$ = true means mb_i ddd mb_j is a reasonable choice.

2. DESCRIBING THE HOST MACHINE

Section 1 examined the ordering that must be preserved while placing microoperation bundles into microinstructions. There remains the question of the restrictions imposed by the host machine itself. Two MBs cannot be placed in the same MI if they both need exclusive control over the same resource at the same time. Such a situation is called a **conflict**. For example, usually two MBs cannot use the same ALU at the same time.

To be able to detect conflicts, a compaction algorithm must operate within the framework of a machine model. More precisely, this framework should be a model of a machine control word, since control word

behavior defines the legality of an MI. This model should have the following properties.

Machine independence. The model should be applicable to a variety of physical machines. Table-driven models can isolate machine-dependent information, such as the host instruction set or the number of resources of a given type. In any case the data structures used by the compaction algorithms should be general.

Manageability. The data structures used by the model should support clear, well-structured design. Appropriate primitives should be defined and used in the code. The model should not contain hardware details unnecessary for the execution of the algorithms. Efficiency should also be considered, since some potential uses of the model involve computations of exponential complexity.

Completeness. The model must be able to represent real-world machines. This implies that some machine features ignored in the past will have to be supported by the model. Designing a simple model and adding extensions later can have disastrous effects on its manageability.

In the following sections we present a model which is a synthesis of most of the models in the literature [KLEI74, DASG76, DEWI76, MALL78, RAMA74, YAU74]. This model demonstrates that the above goals can be (and have been) met. Several features of this model contribute to its completeness. One previously mentioned is the ability to represent microoperations that are not active throughout an entire MI cycle. A machine that supports such MOs is said to have **polyphase timing**. Many compaction algorithms deal only with monophase timing [DEWI76, RAMA74, YAU74]. Another important feature of the model we are presenting is the ability to *delay resource binding* until MIs are formed, that is, until the compaction process. Some models do this for registers only [DEWI76]. The binding concept allows an MO to have a choice of other kinds of resources as well.

This model is not claimed to be the best choice for all compaction problems but is offered to demonstrate the feasibility of describing a variety of realistic machine features. Although coding details are not

```
mo[i] = <ADD, {A, B}, Latch1, ADDER1, Phase2, Field1>

explanation: Latch1 <- A + B;
    Adder ADDER1 is used to add the contents of its two
    input registers A and B. The results are placed in the
    register Latch1. Execution occurs during clock phase
    2. The encoding for the MO is contained in MO field
    Field1.  The name of the MO is "ADD."
```

```
mo[j] = <MOVE, A, B, BUS, Phase0, Field2>

explanation: B <- A;
    Using the bus BUS, data is gated from register A to B.
    Execution is at clock phase Phase0 and the MO's
    encoding is in MO field Field2.
```

```
mo[k] = <JUMP, empty, empty, CSAU, Phase5,
                                    {Field5, Field6="9"}>

explanation: Unconditional Branch;
    The Control Store Address Unit (CSAU) uses the
    immediate data (9) contained in MO field Field6 as
    the target address for the branch. Execution occurs at
    Phase5, and Field5 contains the encoding of the MO.
    No input or output resources are required.
```

```
mo[l] = <SHIFT, ShiftA, ShiftA, SHIFTER, Phase3, Field3>

explanation: ShiftA <- SHIFTER(ShiftA);
    The shifter SHIFTER is used to shift the contents of
    register ShiftA. Execution is during clock phase
    Phase3. The encoding occupies MO field Field3.
```

FIGURE 8. Examples of MO tuple representation.

presented here, implementations of the features do exist [MALL78, WOOD78, FISH79].

2.1 Microoperation Tuples

The MO is the most primitive activity accommodated in our model. An MB is represented as a set of MOs. The semantics of an MO are represented by a six-tuple, ⟨name, I, O, U, T, F⟩, where the tuple elements are

(1) name—an identification of the MO to be performed,
(2) I—the set of all storage resources whose contents are required by the MO as input,
(3) O—the set of all storage resources into which the MO places output,
(4) U—the set of all functional units re-

quired by the MO while the MO is executing,
(5) T—the set of processor clock phases required for MO execution, and
(6) F—the set of all microinstruction fields required by the MO (they contain the encoding of the MO and possibly immediate data).

Elements 2 through 6 are known as tuple sets. Figure 8 shows examples of four MOs with their tuple sets enumerated. Note that the "{" and "}" braces are not written around singleton sets. An MI is a group of MOs contained in one word of control store. The execution time for one MI is called the instruction cycle time and may consist of several subcycles called clock phases. A microprogram is an ordered list of MOs or

MIs which realize the logic of a particular program.

The six-tuple specification of MO semantics is a machine-independent representation of a microoperation. When the MOs and resources of a particular machine are enumerated in this format, the representation becomes tailored to that machine. The tuple is used by a code compaction algorithm to detect conflicts and analyze resource usage between MOs.

The definition of the individual sets of the six-tuple, as given in Figure 8, is not complete. We extend it to take into account **resource allocation**. The tuple sets are considered to be **bound** or **unbound** with respect to their allocated processor resources. An unbound set contains, as elements, the enumeration of *all* possible resources that *can* be allocated to the set. A bound set contains, as elements, those resources that *were* actually assigned to the set. Consider the following unbound Unit set (Uset).

$$U_i = \{\text{ALU1 or ALU2}\}$$

This set is characterized as unbound by its "or" operator, which indicates an Orlist. The Orlist signifies that at some point before the MO can be executed, a choice must be made as to which hardware resource will be assigned to be used by the MO. An Orlist in which the assignments have been made (i.e., the Orlist has only one item in the list) is a **version** of the set. For example, binding the foregoing Uset generates two bound Orlists and thus two versions of the Uset:

$$U_i\text{Ver1} = \{\text{ALU1}\} \qquad U_i\text{Ver2} = \{\text{ALU2}\}$$

Another resource allocation grouping is the **Andlist**, which groups one or more Orlists in an unbound or bound tuple set. Consider an example of an unbound Uset:

$$U_j = \{(\text{ALU1 or ALU2}),\\ (\text{ALU3 or ALU4})\}$$

The Andlist indicates that one unit must be assigned from each Orlist. One version of U_j is

$$U_j\text{Ver1} = \{\text{ALU1, ALU3}\}$$

Tuple sets are either bound or unbound sets. Resource allocation imposes a particular binding upon all Orlists within a tuple set.

The elements of a field set (Fset) specify the MI fields used for storage of the encodings of the MOs and immediate (literal) data. Immediate data are data that an MO needs available within the same MI. An example of an Fset is

$$F_i = \{\text{Field1 or Field2}\}$$

An MO with this field set would require one of the two fields, "Field1" or "Field2," to store the bit encoding of the MO. Next we have an MO that, in addition to a control field, requires that "Field5" contain the decimal constant "8" to be used as immediate data. The Fset of such an MO might look like

$$F_j = \{\text{Field1, Field5} = \text{"8"}\}$$

2.2 Relationships Between Tuples

Using the tuple set model to detect resource conflicts is done primarily by testing for nonnull set intersections. Consider the following definitions.

> Two MOs are **unit compatible** if their Usets are disjoint or if their Usets are not disjoint but their Tsets are disjoint.

This definition simply states that a unit cannot be used by two different MOs at the same time.

> Two MOs are **field compatible** if their Fsets are disjoint or if all of the common elements of their Fsets contain the same value.

Since the use of a field lasts an entire instruction cycle, timing sets need not be considered. The last clause of the definition allows two MOs to share the same literal value. For example, two field-compatible MOs might both need a "5" in Field9. Not only does our model allow two MOs to share a literal, there is no way to prevent them from sharing one. With most machines this would cause problems. An extremely flexible representation of field semantics can be found in DEWI76.

For the use of registers to be conflict-free, the same register cannot be used by two MOs at the same time, unless both MOs

are reading. But this is just the negation of data interaction, with a timing consideration added:

Two MOs are **data compatible** if there is no data interaction between them, or if their Tsets are disjoint.

Notice that this definition applies to two dependent MOs which execute at separate times in the MI cycle, but in the wrong order. Thus data compatibility does not imply order preservation. It is easy to verify, however, that if two MOs are order preserving, they are also data compatible.

If two MOs are unit compatible, field compatible, and data compatible, then they are said to be **resource compatible**. In this model, resource compatible implies conflict-free.

Two MOs which are order preserving and conflict-free are defined to be **parallel**. Parallel MOs can be placed in the same MI. During the construction of a list of MIs from an SLM, an MO which is data available to an MI in the list and is also resource compatible with that MI is said to be **compactable** into that MI. The word composable has been used synonymously with compactable.

3. FORMING COMPLETE INSTRUCTIONS

An algorithm for forming complete instructions (CIs) examines a set of microoperation bundles and constructs conflict-free microinstructions from them. The resource conflicts are detected using a control word model such as the one discussed in Section 2. An MI is **complete** with respect to a set of MBs if no other members of the set can be added to the MI; otherwise it is **incomplete**. An algorithm that forms complete instructions is used in some form by all of the compaction algorithms in Section 4 except the linear algorithm.

A particular algorithm (FormCIs) is presented in the following two subsections. FormCIs builds combinations of MBs from the input set by examining one MB at a time. Each MB is alternately considered as included in or excluded from the combination. A combination can be rejected because of (1) resource conflict or (2) incompleteness.

As seen in Section 3.2 the algorithm presented here is not the only possible approach, but it is consistent with our model, and a good demonstration of the model's use.

3.1 The FormCIs Algorithm

FormCIs is applied to a set of data-available MBs, which are called the **Dset**. Since the MBs are known to be data available, only units and fields need to be checked for conflicts. However weakly dependent children of MBs in the Dset may be added to an MI if their parents are added—as explained in the following.

Given a Dset, FormCIs generates all possible complete instructions from it. Each MB in the Dset can be either included in or excluded from the instruction currently being generated. If its inclusion or exclusion is not yet decided, the MB is said to be a **remaining MB**. The set of all remaining MBs is called the **remaining Dset (RDset)**. An instruction for which remaining MBs exist is said to be a **partial instruction**. An instruction under construction is considered a partial instruction until the disposition of all MBs has been decided.

We wish to detect when a partial instruction is incomplete. A useful concept is that of the **subinstruction**. A partial instruction is a subinstruction to an instruction if the union of the partial instruction and all MBs that can potentially be added is a subset of the instruction. The only MBs that can be added are the remaining MBs, unless adding weakly dependent MBs is allowed. This possibility is discussed in Section 3.1.2.

As an example of the subinstruction relation, suppose that

$$I1 = \{1, 2, 4\}$$

is a microinstruction and that

$$PI1 = \{1\}$$

is a partial instruction associated with the remaining Dset

$$RDset = \{4\}$$

In this case, PI1 is a subinstruction to I1 because the union of PI1 and RDset, $\{1, 4\}$, is a subset of I1.

A partial instruction that is a subinstruction to a previously generated complete instruction will be incomplete under any disposition of its remaining MBs. A partial instruction that is not a subinstruction to any known complete instruction is said to be **potentially complete.**

The set of complete instructions is formed by repeatedly applying the procedure outlined in the following.

FormCIs

(1) Let I represent the current partial instruction. I is initially empty and the remaining Dset is initially equal to the Dset.

(2) Pick an MB from the remaining Dset. Consider including this MB in I, possibly in more than one version. If it does not conflict with an MB already in I, reapply this procedure with this MB in I (once for each version).

(3) Then consider excluding the MB from I. If this does not make I a subinstruction to an already found complete instruction, reapply this procedure with this MB excluded from I.

(4) If the remaining Dset is empty, select I to be a generated instruction.

FormCIs generates all combinations of the MBs in the Dset except those having conflicts and those known to be incomplete. Thus all complete instructions will be generated. It is not immediately obvious that each instruction is guaranteed to be complete. This is, in fact, the case; the CI "purging extension" presented in Section 3.1.2 makes proving completeness trivial for the full algorithm.

Besides conflict and incompleteness, there is one other case where a partial instruction does not require exhaustive examination of its remaining Dset. Suppose that a complete instruction has just been generated and consider the last MB that was excluded from this instruction (if any). All MBs selected after the last excluded MB were found conflict-free. Any other disposition of these MBs could not possibly lead to an instruction with a different MB in it. Thus, whenever a complete instruction is generated, the examination of partial instruction combinations backs up to the last excluded node.

If desired, the rest of Section 3.1 can be skipped on first reading.

3.1.1 *Introductory Example of FormCIs Execution*

The execution of the FormCIs algorithm forms a tree. Each node on the tree corresponds to an MB picked for inclusion or exclusion. The first of these, an "include node," is drawn as the name of the MB in parentheses. The second, an "exclude node," is drawn as the name of the MB in square brackets surrounded by parentheses. The tree is referred to as a FormCIs tree. An instance of a node is also represented numerically by two digits separated by a decimal point. The first digit indicates the vertical position of a node counting ascending from 1. The second digit represents the horizontal position of a node counting ascending from 1. The Check-NextMB procedure, shown in a later diagram, corresponds to a node on the tree. The tree is created in depth-first, left-to-right order. Notice in Figure 9 that the partial instruction at a node is identified by the path from the root to that node. Instructions are written as a list of MBs delimited by commas. Excluded MBs are also shown because they can be used in checking completeness. The final instructions produced by this algorithm are a set of included MBs.

Figure 9 is an introductory example of the formation of complete instructions. Suppose the Dset contains three MBs: 1, 2, and 3. Furthermore, assume that 1 conflicts with 2 (1 c 2) and that 1 conflicts with 3 (1 c 3). For this example, it is not important to know what resources cause the conflicts. We now examine the execution of the algorithm, one tree node at a time.

First, MB 2 is picked arbitrarily and included in I, the current partial instruction (node 1.1). Then MB 1 is picked for inclusion in I, but 1 conflicts with 2, and so we are finished with this node (node 2.1). Next 1 is considered as excluded from I (node 2.2). There are not yet any selected instructions to compare with I, so I is potentially complete. Now MB 3 is picked and checked

```
                    .                 FormCIs Tree:
            /               \
        (2)               ([2])           MBs:        1, 2, 3.
       /   \             /    \
    (1)  ([1])        (1)    ([1])        Conflicts:  1 c 2,
    c2    / b        /  \     i                       1 c 3.
        (3)        (3) ([3])
         c1
                                          Where:  c - conflict,
                                                  i - incomplete,
        SelectedIs                                b - backup.

      ┌───┬──────────────┐
      │ 1 │ {2, [1], 3}   │
      ├───┼──────────────┤
      │ 2 │{[2], 1, [3]}  │
      ├───┼──────────────┤
      │   │              │
      └───┴──────────────┘
```

Execution Trace:

Node	MB	Conflict?	Remaining Dset	Partial Instruction	Incomplete?	Selected Instructs.
0	-	-	{1,2,3}	empty	-	{}
1.1	2	no	{1,3}	{2}	no	{}
2.1	1	yes				
2.2	[1]	no	{3}	{2,[1]}	no	{}
3.1	3	no	{}	{2,[1],3}	no	{2,[1],3}
1.2	[2]	no	{1,3}	{[2]}	no	{2,[1],3}
2.3	1	no	{3}	{[2],1}	no	{2,[1],3}
3.2	3	yes				
3.3	[3]	no	{}	{[2],1,[3]}	no	{2,[1],3; [2],1,[3]}
2.4	[1]	no	{3}	{[2],[1]}		

Comments About the Trace:

Node   0: The remaining Dset is initially the same as Dset.
Node 2.1: 1 conflicts with 2, so return to parent node.
Node 3.1: remaining Dset empty, so add the current partial
           instruction to the SelectedIs, then backup to the
           first ancestral exclude node, which is Node 2.2.
Node 3.2: 3 conflicts with 1, so return to parent node.
Node 3.3: Select the current partial instruction. Since
           the current node is an exclude node, there is no
           backup to an exclude node. Return to parent node.
Node 2.4: The partial instruction {[2],[1]} is a
           subinstruction to the first selected instruction,
           {2,[1],3}. Return to parent node.

FIGURE 9. *Introductory example of forming complete instructions.*

for inclusion in I. There is no conflict and the remaining Dset is empty, so I ({2, [1], 3}) is selected (node 3.1). We back up to the most recent exclude node, indicated by the b next to node 2.2. Now we consider excluding 2 (node 1.2). I ({[2]}) is not a subinstruction to the selected instruction {2, [1], 3} since I might be able to contain 1. Next 1 is picked, found to have no conflict with I, and added to I (node 2.3). Then 3 is picked, but its inclusion conflicts with 1 (node 3.2). Excluding 3 does not cause I ({[2], 1, [3]}) to be a subinstruction to {2, [1], 3}, so I is selected (node 3.3). Since the last node on the path is an exclude node, there is no backup. The next node in the normal order is [1], but the resulting I({ [2], [1]}) is found to be a subinstruction to the first selected I ({2, [1], 3}). Thus I is incomplete relative to the first selected instruction (node 2.4). This completes the FormCIs execution.

3.1.2 The Full FormCIs Algorithm

Accommodating unbound resources in the proposed algorithm is straightforward. We produce a version of an MB for each possible binding of its resources. Whenever an MB is considered for inclusion, an include node for each version is generated. However it now becomes possible for a new selected instruction to "supersede" an old one. In other words, an already selected instruction might be a subinstruction to the newly selected one; the new one cannot be a subinstruction because such an instruction fails the completeness test. The algorithm discards superseded instructions. As a consequence each instruction in the final list of instructions is guaranteed to be complete. (Recall that in Section 3.1 we stated that every complete instruction is in the list.)

We now redefine subinstruction to incorporate the concept of weak dependency. First, we define the extended remaining Dset. An MB is in the extended remaining Dset if it is in the remaining Dset or if it is below the remaining Dset in the graph and each of its parents satisfies one of the following:

(1) the parent is above the original Dset in the graph, or
(2) the parent is in the extended remaining Dset and the MB is weakly dependent on it.

Now we can say that a partial instruction is a subinstruction to an instruction if the union of the partial instruction and its extended remaining Dset is a subset of the instruction.

Alternatively, an analysis of subinstruction could be based on the excluded MBs instead of the remaining Dset, but the algorithm would operate in essentially the same manner. The application of the extended algorithm in a simple example can be seen in Figure 10. A flow diagram for the full algorithm appears in Figure 11.

3.2 Alternative Approaches

List scheduling algorithms (Section 4.4) consider only one possible complete instruction from each Dset. This allows a corresponding simplification in the calculation of complete instructions [WOOD78,

FISH79]. The tree of MBs is collapsed into a linear list of MBs. Each time a choice of MBs is possible, only one of them is considered. A weighting function is applied to make the choice.

There are two advantages to this approach. First, the computational complexity of forming complete instructions is reduced from exponential to linear. This is significant since sets of complete instructions are calculated many times during the course of a compaction. Second, this algorithm is much easier to program. Since only one instruction is being generated, there is no need to do subinstruction analysis.

Another approach to forming complete instructions has been developed by DeWitt [DEW176]. DeWitt's algorithm, AP-PLYCWM, is significant because it generates all possible complete instructions yet is computationally less complex than the FormCIs algorithm of the previous section. An execution tree generated by AP-PLYCWM has fewer nodes than a FormCIs tree for the same Dset.

APPLYCWM begins with an empty MI and starts adding MBs one at a time, just as FormCIs does. However no new node is generated until a resource conflict actually occurs. Thus one node can contain several MBs. When a conflict occurs between two MBs, two new nodes under the current node are created. One contains the MBs in the current node and one of the conflicting MBs, and the other contains the current node's MBs and the other conflicting MB. Then APPLYCWM continues adding MBs from the Dset to both of the new nodes. This process continues until all the nodes from the Dset have been used.

Since the execution tree for AP-PLYCWM only branches if a new MB conflicts with the MBs already in a node, it will be smaller than the tree for FormCIs, which has multiple branches for every new MB. However APPLYCWM cannot be used with the model from Section 2, because it makes no provision for handling different versions of the same MB.

It is not clear how APPLYCWM and the model can be extended in order to work together. The version concept could possibly be modified to allow an individual resource to be bound without binding an entire MO. Then APPLYCWM could be mod-

```
FormCIs Tree:              .
                        /     \              MB:        1, 2, 3
                    (1A)        (1B)
                    /   \       / b          Versions:  ┌1A 2A 3A
                (2A)  ([2])  (2A)            │1B
                /  \    i    / b
            (3A) ([3])    (3A)               Conflicts: 1A c 3A.
            c1A
                                      ┐      DDG:  (1)   (2)
                                      │                   ¦ wd
                    SelectedIs        │                  (3)
            ┌───┐ ┌──────────────┐    │
            │ ──┼→│ 1 {1A, 2A, [3]}│ *─┘      *-The indicated Selected
            │   │ ├──────────────┤           I is discarded because
            └───┤ │ 2 {1B, 2A, 3A}│←─        it is a subinstruction
                │ └──────────────┘           to the new Selected I

Execution Trace:
            Con-    Remaining  Partial       Incom-    Selected Is
Node  MB    flict?  Dset       Instruction   plete?
0     -     -       {1,2}      empty         -         {}
1.1   1A    no      {2}        {1A}          no        {}
2.1   2A    no      {3}        {1A,2}        no        {}
3.1   3A    yes
3.2   [3]   no      {}         {1A,2A,[3]}   no        {1A,2A,[3]}
2.2   [2]   no      {}         {1A,[2]}      yes
1.2   1B    no      {2}        {1B}          no        {1A,2A,[3]}
2.3   2A    no      {3}        {1B,2B}       no        {1A,2A,[3]}
3.3   3A    no      {}         {1B,2A,3A}    no        {1B,2A,3A}

Comments About the Trace:
Node0:    The Dset is {1,2}. MB 3 is not data available.
Node2.1:  Choosing MB 2 causes its wd descendant,
          MB 3, to be added to the RemainingDset.
Node3.1:  3A conflicts with 1A, so return to parent node.
Node3.2:  The empty RemainingDset causes us to select the
          current partial instruction. There is no backup
          because the current node is an exclude node.
Node2.2:  The partial instruction {1A,[2]}is a subinstruction
          to the selected instruction {1A,2A,[3]} because
          excluding 2 implies excluding 3 (since 2 wd 3).
          Return to parent node.
Node3.3:  While entering the current partial instruction, we
          remove the previously selected instructions which
          are subinstructions to it. We try to backup to the
          first ancestral exclude node, and, since there
          is not one, we are back at node 0 and are finished.
```
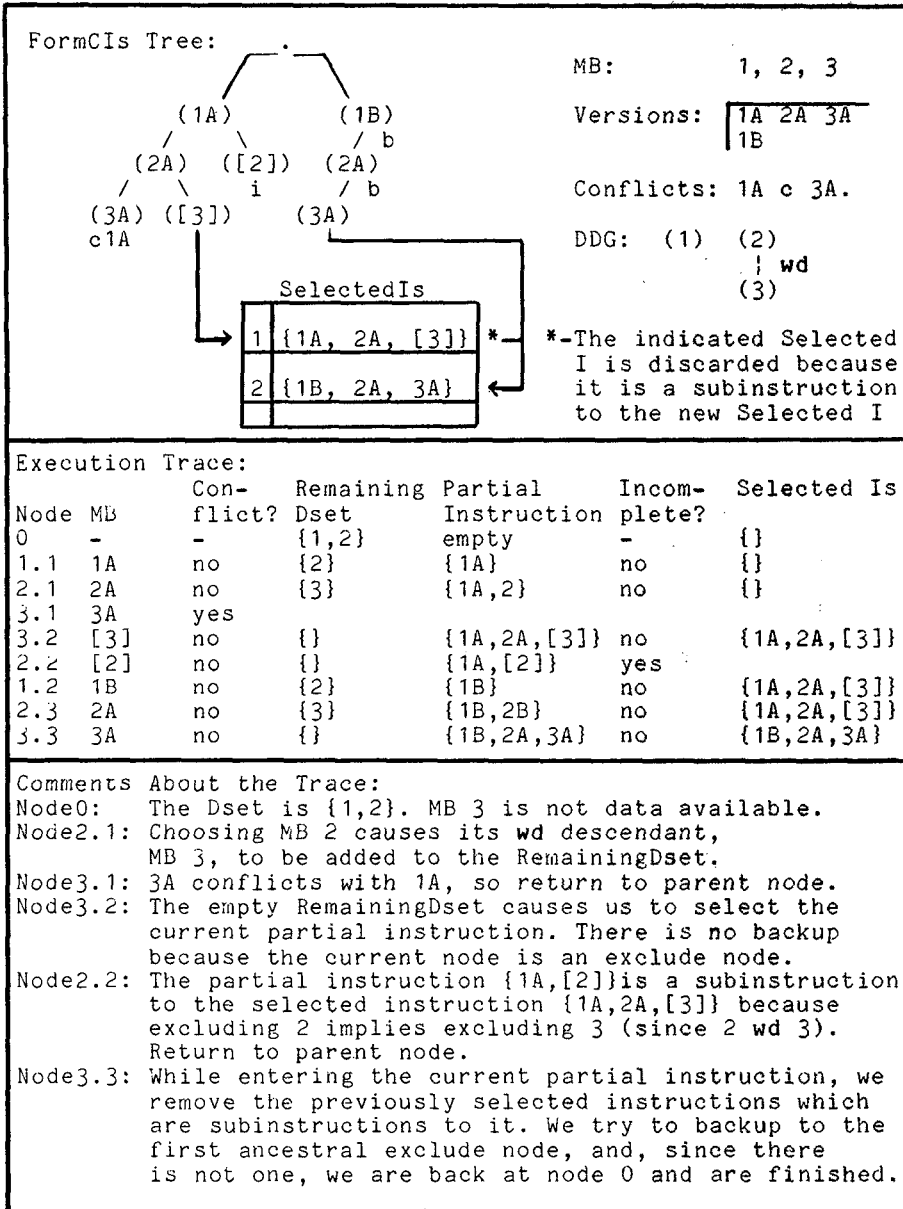
FIGURE 10. Example trace of the full FormCIs algorithm.

ified to bind resources in order to resolve a conflict. Although final instructions might still contain unbound resources, this presents no problem since an arbitrary binding is sufficient at this point.

4. COMPACTION ALGORITHMS

All of the microcode compaction algorithms that have appeared as of this writing fall into one of four categories: linear analysis,

critical path, branch and bound, and list scheduling. Representative algorithms for each of these categories are presented in the following sections.

The operation of the compaction algorithms is shown by a common example. The data dependency graph and conflicts for the example SLM are given in Figure 12. This SLM has eight microoperation bundles. MB 3 depends on MB 2, MB 4 on both
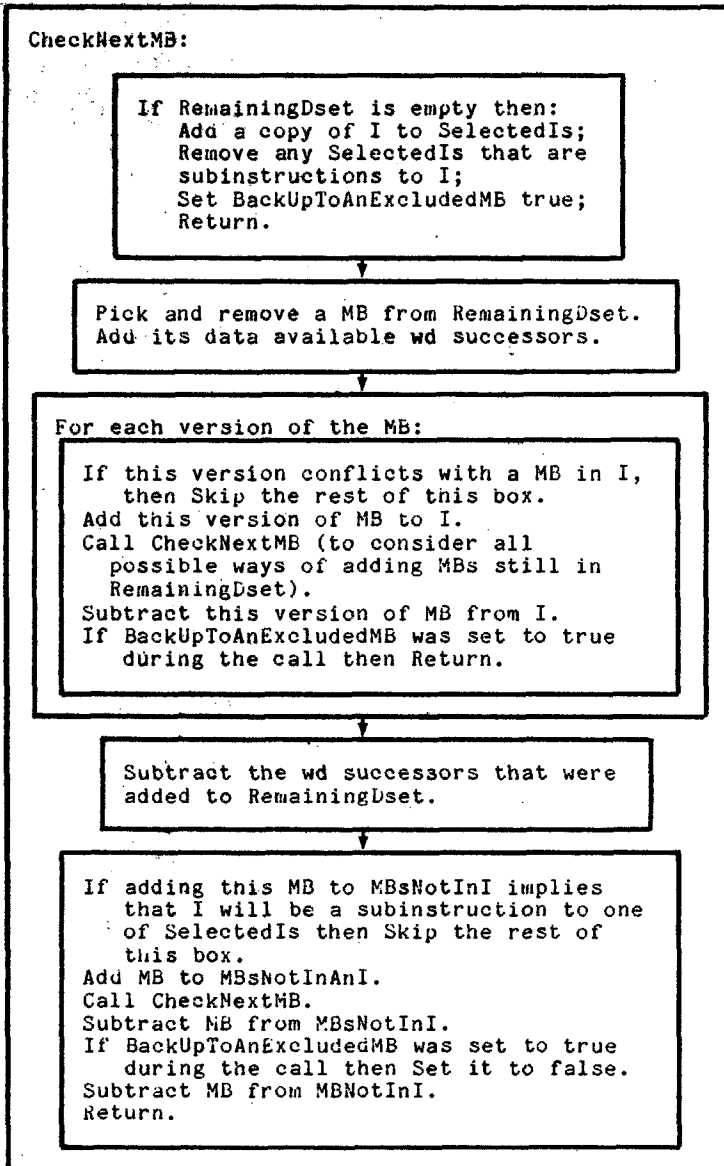
```
CheckNextMB:

        If RemainingDset is empty then:
        Add a copy of I to SelectedIs;
        Remove any SelectedIs that are
        subinstructions to I;
        Set BackUpToAnExcludedMB true;
        Return.


        Pick and remove a MB from RemainingDset.
        Add its data available wd successors.


    For each version of the MB:

        If this version conflicts with a MB in I,
            then Skip the rest of this box.
        Add this version of MB to I.
        Call CheckNextMB (to consider all
            possible ways of adding MBs still in
            RemainingDset).
        Subtract this version of MB from I.
        If BackUpToAnExcludedMB was set to true
            during the call then Return.


        Subtract the wd successors that were
        added to RemainingDset.


    If adding this MB to MBsNotInI implies
        that I will be a subinstruction to one
        of SelectedIs then Skip the rest of
        this box.
    Add MB to MBsNotInAnI.
    Call CheckNextMB.
    Subtract MB from MBsNotInI.
    If BackUpToAnExcludedMB was set to true
        during the call then Set it to false.
    Subtract MB from MBNotInI.
    Return.
```

FIGURE 11.   Flow diagram of FormCIs algorithm.

MBs 1 and 3, both MBs 5 and 6 on MB 4, and MB 8 on MB 7. There are three conflicts between MBs: MB 2 with MB 7, MB 3 with MB 7, and MB 5 with MB 6. For this example the type of conflict is unimportant.

4.1 The Linear Algorithm

The linear algorithm (Lin) is a modification of that described by Dasgupta and Tartar [DASG76, BARN78, DASG78, DASG79]. It is

restructured to fit our model and also is more efficient than their algorithm. Lin operates on an SLM which is in the form of a list of MBs. MBs from the SLM are added, in the order in which they appear on the list, to an initially empty list of microinstructions. For each MB in the SLM, an attempt is first made to add the MB to an existing MI, and if this fails, to a new MI created to hold it. (The name "linear" comes from the linear examination of the

```
(1) (2)    (7)
 |   |      |
    (3)    (8)
   /
 (4)                    2  c  7
 / \                    3  c  7
(5) (6)                 5  c  6
   (a)                      (b)
```
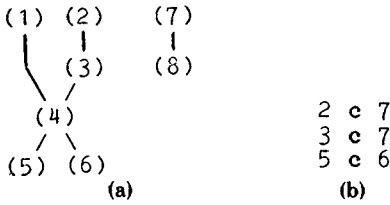
FIGURE 12. Sample compaction problem. (a) Data dependency graph. (b) MB conflicts.

original SLM and does not imply that the algorithm executes in linear time.) We now present the algorithm in more detail.

The search for an existing MI to which the MB currently being examined can be added begins with data dependency analysis. Starting at the bottom of the MI list and proceeding upward, examine each MI. Continue the search above a microinstruction in the list if the MB under current consideration is not data dependent on any MB in that MI. If the current MB is data dependent on an MB in the ith microinstruction, the current MB cannot be placed in any microinstruction earlier than i. It can only be placed in the ith MI if it is weakly dependent with every MB on which it is data dependent. In other words, it can be placed if it will not execute until these MBs are finished. The object of this search is to find the earliest (highest) microinstruction in which the new MB can be placed without violating the ordering imposed by the data dependencies in the SLM. This MI is called the **rise limit**.

The next step in the search for an existing MI is the examination of resource conflicts. To be placed in a microinstruction, the current MB must not conflict with any MB already in that microinstruction. Assuming that a rise limit i was found, search down-

ward in the list, starting with microinstruction i, for some microinstruction in which the new MB can be placed. When such a microinstruction is found, add the new MB. If no such microinstruction is found, add the new MB to the end of the microinstruction list, thus forming a new microinstruction containing only one MB.

If no rise limit was found, then the current MB was not data dependent on any MB in the microinstruction list, and the MB can be added to any microinstruction with which it has no conflicts. In this case begin the downward search at the top of the MI list. If there are no microinstructions to which the MB can be added without conflict, use the MB to form a new microinstruction at the top of all the other microinstructions. Placing this MB at the top will keep it from blocking any new MBs that depend on it.

Figure 13 shows the algorithm at work on the MBs of the example SLM. The first MB processed is MB 1. Since the list of MIs is initially empty, MB 1 is placed by itself in MI 1. The next MB, MB 2, is not data dependent on MB 1. It can therefore be placed in MI 1 along with MB 1, assuming no conflicts exist. Since MB 1 and MB 2 do not conflict, they are combined into a two-MB MI.

The next MB is MB 3. It is data dependent on MB 2, so it can be placed no higher than MI 2. Therefore a new MI, MI 2, that holds MB 3 is created.

MB 4, being data dependent on MB 3, similarly forms a new MI, MI 3. In the same fashion, MB 5 forms MI 4. MB 6, data dependent on MB 4, can rise no higher than MI 4. MB 6 cannot be placed in MI 4, however, because of a conflict between MBs 5 and 6. MB 6 is placed in a new microinstruction, MI 5.

FIGURE 13. Lin execution trace for the example in Figure 12.

List of MIs after Adding Microoperation Bundle:

MI	1	2	3	4	5	6	7	8
1	1	1,2	1,2	1,2	1,2	1,2	1,2	1,2
2			3	3	3	3	3	3
3				4	4	4	4,7	4,7
4					5	5	5	5,8
5						6	6	6

```
(* predefined const maxMBs, maxInstrs;
   predefined type MOBUNDLE, INSTR;
   type SLMB = array [0. .maxMBs] of MOBUNDLE; 0th elem = length
   type SLMI = array [0. .maxInstrs] of INSTR; 0th elem = length
   The procedures and functions used must be defined.
*)

procedure L in (slmB : SLMB; var slmI : SLMI);
(*
   Approximate minimization of number of microinstructions constructed from microoperation bundles in slm
   form (branches only at end, entries only at start).
*)
   var b : MOBUNDLE;
       bNum : 1. .maxMBs;
       i : integer;
       riseLimit : 0. .maxInstrs; (* instr that b cannot precede *)
       lastI : 0. .maxInstrs;
begin
   AppendI (InstrFromMB (slmB[1]), slmI);
   lastI := 1;
   for bNum = 2 to NumElements (slmB) do begin
      b := slmB[bNum];
      (* Find RiseLimit *)
      i := lastI;
      while (i > 0) and CanPrecedeI (b, slmI[i]) do i := i - 1;
      riseLimit := i;
      (* Add b to an instr <= riseLimit *)
      if (riseLimit <> 0) and
         CanAddToUnprecedableI (b, slmI[riseLimit])
      then AddToI (b, slmI[riseLimit])
      else begin
         (* i is still = riseLimit *)
         (* note slmI[lastI + 1] might be referenced below *)
         repeat i := i + 1
         until (i > lastI) or CanAddToPrecedableI (b, slmI[i]);
         if i <= lastI then AddToI (b, slmI[i])
         else begin (* no instr where b can be added *)
            if riseLimit = 0
            then InsertIAtTop (InstrFromMB (b), slmI)
            else AppendI (InstrFromMB (b), slmI);
            lastI := lastI + 1;
         end (* if *);
      end (* if *);
   end (* for *);
end (* Lin *);
```

FIGURE 14.   Lin in PASCAL.

The next MB, MB 7, is not data dependent on any previously placed MB. It can therefore be placed in any MI with which it does not conflict. The search for such an MI starts with MI 1. MB 7 cannot be placed in MI 1 because of a conflict with MB 2. It cannot be placed in MI 2 because of a conflict with MB 3. There is no conflict with MB 4, so MB 7 can be added to MI 3. MB 8 is the last MB to be placed. It cannot be placed in an MI before MI 4 because of its data dependency on MB 7. It can be added to MI 4 since there is no conflict between MBs 5 and 8.

The final microprogram has five microinstructions and is of optimal length. That is, there is no way to form fewer than five microinstructions from this SLM. We can intuitively see that five is the minimum length by noticing that the longest path
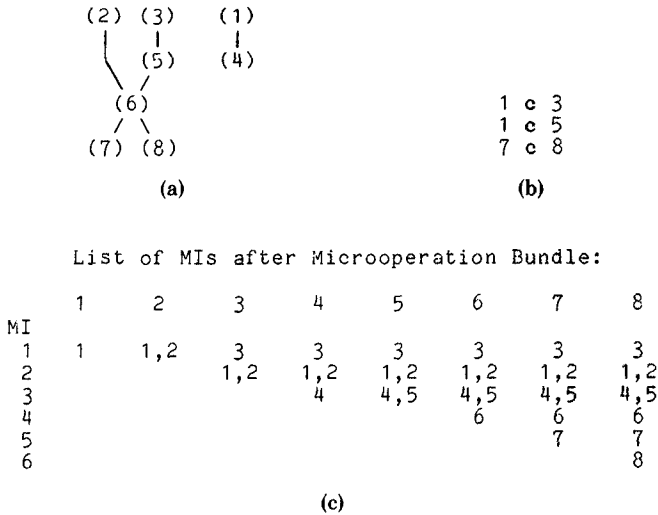
```
(2) (3)    (1)
 |   |      |
 |  (5)    (4)
  \  /
  (6)                          1 c 3
  / \                          1 c 5
(7) (8)                        7 c 8

  (a)                            (b)
```

```
List of MIs after Microoperation Bundle:

     1     2     3     4     5     6     7     8
MI
 1   1    1,2    3     3     3     3     3     3
 2              1,2   1,2   1,2   1,2   1,2   1,2
 3               4    4,5   4,5   4,5   4,5   4,5
 4                           6     6     6
 5                                 7     7
 6                                       8
```

(c)

FIGURE 15. Example of nonoptimal Lin execution. (a) Renumbered
data dependency graph. (b) Renumbered microoperation conflicts.
(c) Execution trace.

through the DDG is of length 4. To keep
data-dependent MBs in different MIs, we
need at least four MIs, and therefore 4 is
the theoretical lower bound. A fifth MI is
necessary due to the conflict between MB
5 and MB 6.

A PASCAL procedure for the Lin algo-
rithm can be seen in Figure 14. In order to
run, this procedure must be supported by
the predefined types MOBUNDLE and
INSTR, as well as by the following routines:
AppendI, CanPrecedeI, CanAddToUnpre-
cedableI, CanAddToPrecedableI, AddToI,
and InsertIAtTop.

Though the Lin algorithm performed op-
timally in the example of Figure 13, it is not
guaranteed to do so. Figure 15a shows the
DDG in Figure 12 renumbered. Since Lin
is a first-come, first-served algorithm, this
will have an effect on the performance.

Figure 15c shows the performance of Lin
on this DDG. MBs 1 and 2 are not data
dependent on each other and do not con-
flict, and so are combined into MI 1. MB 3
is not data dependent on either of the pre-
vious MBs and so can be added to MI 1 if
it does not conflict with the MBs. However
MB 3 does conflict with MB 1, so a new MI
must be formed. In this case, as mentioned
previously, we form a new MI above the
already formed MI. MB 4, data dependent
on MB 1, is placed in MI 3. MB 5, data
dependent on MB 3, can be placed no ear-

lier than MI 2. It conflicts with MB 1 in MI
2, however, and is therefore placed in MI 3.
MBs 6, 7, and 8 are placed next, forming
three new MIs, as in the foregoing example.
The resulting microprogram is six microin-
structions long and is thus nonoptimal.

How can this algorithm be made always
to produce optimal results? This can be
accomplished only by running the algo-
rithm with the MBs in the SLM in every
legal order (every order that does not vio-
late data dependency) until an optimal list
of MIs is obtained. However it is imprac-
tical to do this because the many redundant
calculations involved make this approach
much slower than the BAB algorithm of
Section 4.3.

Weak dependencies are handled in a
straightforward manner by the Lin algo-
rithm. They are tested when Lin starts to
check for conflicts with the rise limit. The
current MB was found to be dependent on
some MB(s) in the rise limit. Any strong
dependency here means the current MB
cannot be added to the rise limit MI. In the
PASCAL example of Figure 14, this analy-
sis is done by CanAddToUnprecedableI.

4.2 The Critical Path Algorithm

Critical path algorithms for microcode com-
paction were introduced by Ramamoorthy

	Early Partition	Late Partition	Critical Partition
Frame			
1	1,2,7	2	2
2	3,8	1,3	3
3	4	4,7	4
4	5,6	5,6,8	5,6

(a)

	Revised Critical Partition	Noncritical MBs
Frame		
1	2	1,7,8
2	3	
3	4	
4a	5	
4b	6	

(b)

List of MIs After Adding Noncritical MB:

	1	7	8
Frame			
1	1,2	1,2	1,2
2	3	3	3
3	4	4,7	4,7
4a	5	5	5,8
4b	6	6	6

(c)

FIGURE 16. CPath execution of the example in Figure 12. (a) Forming the critical partition. (b) The revised critical partition. (c) Adding the noncritical MBs.

and Tsuchiya [RAMA74]. Their technique was similar to the critical path approach to processor scheduling [RAMA69, GONZ77]. The following critical path algorithm (CPath) attempts to identify MBs that must be executed at a certain time in order for the list of MIs to be optimal. The MBs chosen are those which are on a longest path (the critical path) through the DDG. As noted, the minimum possible number of MIs is just the length of a longest path. If any MB on this path is delayed beyond the MI where it first becomes data available, any MB following will also be delayed, and the number of MIs in the list will be increased.

CPath proceeds as follows (see Figure 16). The first step is to create an early partition (EP). Each time frame of the EP contains those MBs that could be executed in that time, at the earliest. Figure 16a shows the EP for the example SLM. Since MBs 1, 2, and 7 are not data dependent on any other MB, they can be executed in frame 1. MBs 3 and 8 are each dependent on an MB in frame 1 and thus can be executed in frame 2 at the earliest. MB 4 is dependent on MB 3 in frame 2, so it can be executed in frame 3. Notice that although MB 4 is also dependent on MB 1, it must be placed in frame 3; otherwise it could execute at the same time as one of its ancestors in the DDG. An MB must be placed in the frame of the EP after the frame of its latest ancestor.

MB 5, being dependent on MB 4, is placed in frame 4. The same is true for MB 6. Notice that data dependencies alone determine the placement of MBs in the EP. Conflicts are resolved later.

The final early partition is of length 4. This is the length of the longest path

through the DDG and defines the minimal number of MIs in the list of MIs.

The next step is the creation of a **late partition (LP)**. In the LP the latest possible timings of the MBs are displayed. The LP is created by moving backward through the SLM. No MB is dependent on MB 8, so MB 8 can be the last MB executed. It can therefore be placed in time frame 4 of the LP. Again, the length of the LP is just the length of the longest path through the DDG.

The next MB placed is MB 7. Since MB 8 is dependent on MB 7, MB 7 must be placed in the time frame before MB 8, or frame 6. No MBs are data dependent on either MB 5 or 6, and so these MBs can be placed in time frame 4. Since MBs 5 and 6 are dependent on MB 4, MB 4 must be placed in frame 3. MBs 1 and 3, both ancestors of MB 4, must be placed in the frame preceding that of MB 4, frame 2. Finally, MB 2, the ancestor of MB 3, is placed in frame 1. The resulting LP is in Figure 16a. If the data dependency graph is represented by a matrix, the LP can be created by applying the EP algorithm to the transpose of the graph matrix.

The construction of these partitions facilitates the identification of MBs on the critical path of the DDG. These MBs, called critical MBs, are just those with the same early and late timings. It is trivial to construct a **critical partition (CP)** by examining the EP and LP. The critical partition for our example is shown in Figure 16a. MBs 2, 3, 4, 5, and 6 are critical here. The length of the CP is, of course, the length of the longest path through the DDG, in this case 4. The CP serves as a skeleton of our final list of MIs.

Since the CP was constructed by considering only data dependencies between MBs, two MBs in the same frame of the CP may conflict. And since the frames of the CP will serve as a basis for MIs in the list of MIs, conflicting MBs in a frame must be separated. In frame 4 of our example CP, MBs 5 and 6 conflict. A frame with conflicting MBs must be split so that the MBs in the resulting subframes do not conflict. In our example frame 4 is trivially split into subframes 4a and 4b, each containing a single MB. The result of this process, the **revised critical partition (RCP)**, is

shown in Figure 16b. We will continue to talk in terms of frames because they provide a link between the RCP and the early and late partitions. As seen from frame 4, one frame can contain several microinstructions. A subframe is a partially complete microinstruction. Either FormCIs or a similar algorithm must be used to form the subframes.

The next and final step of CPath is to add the noncritical MBs to the RCP, forming the final list of MIs. The noncritical MBs here are MBs 1, 7, and 8. We search the RCP from the frame containing the MB in the EP to the frame containing the MB in the LP for a microinstruction to which the noncritical MB can be added. In the case of MB 1, we start with frame 1 and end with frame 2. If the MB cannot be added to any of the frames within this range, a new subframe is created. This subframe is positioned at the end of the last frame in the range. The MB is placed in this subframe, creating an MI with only one MB. In our example, however, MB 1 can be added to the MI in frame 1 without difficulty. Figure 16c, column 1, shows the list of MIs after MB 1 is added.

MB 7 can be added to either frame 1, 2, or 3. It cannot be added to frame 1 because of a conflict with MB 2 or to frame 2 because of a conflict with MB 3. It can be added to frame 3 however.

The final MB to add is MB 8. If we used the EP to indicate the earliest frame to which MB 8 could be added, we would find that it could be added to frame 2. If it were added to frame 2, however, it would be placed before MB 7, thus violating the data dependency constraint. Therefore we must *revise the EP* to place all MBs dependent on MB 7 after MB 7. The earliest timing of MB 8 becomes frame 4, which is also the latest possible timing. We attempt to add MB 8 to the subframes of frame 4, in turn, and find that MB 8 can indeed be placed in subframe 4a. Column 8 of Figure 16c shows the final list of MIs, which is of length 5 and therefore optimal.

The critical path algorithm, like the linear algorithm, does not always produce optimal results. This can be shown by considering our example DDG, plus the following conflict: 4 *c* 7. The early, late, critical, and revised critical partitions remain the same

•

```
        List of MIs after adding Noncritical MB:

              1                    7          8
Frame                  Frame
  1          1,2         1         1,2        1,2
  2          3           2         3          3
  3          4           3a        4          4
  4a         5           3b        7          7
  4b         6           4a        5          5,8
                         4b        6          6
```

FIGURE 17. Example of nonoptimal CPath execution: Adding the noncritical MBs.

as for the foregoing example. However the result of adding the noncritical MBs is changed. MB 1 can be added to frame 1, as before. MB 7, however, cannot be added to frame 3 because of a conflict with MB 4. Since frame 3 is the latest time frame of MB 7, a new subframe must be created after frame 3 and before frame 4a. In column 7 of Figure 17 MB 4 is in frame 3a and MB 7 is in frame 3b.

As before, MB 8 must be placed within frame 4. It can be placed in frame 4a, giving the list of MIs of Figure 17, column 8. This list of MIs is of length 6 and is nonoptimal, as can be seen by moving MB 8 to frame 4b and MB 7 to frame 4a, giving the list of MIs of Figure 18. This list of MIs is five microinstructions long and is optimal.

This example shows the weakness of the critical path algorithm. Two adjacent subframes may be composed into a single microinstruction but are not because of the rigid boundaries between frames imposed by the algorithm. This problem does not arise from the first-come, first-served nature of the algorithm. The foregoing nonoptimal list of MIs would have been obtained for any ordering. Both an exhaustive search of the possible orderings and an attempt to combine adjacent frames would be necessary to make this algorithm optimal.

```
Frame          MI

  1          1,2
  2          3
  3          4
  4a         5,7
  4b         6,8
```

FIGURE 18. An optimal solution to the problem of Figure 17.

Weak dependencies are handled in CPath by putting MBs in the same frame of the EP and LP as the parents on which they are weakly dependent.

4.3 The Branch and Bound Algorithm

The third algorithm to be considered is branch and bound (BAB), a general class of tree-searching scheduling algorithms. Yau, Schowe, and Tsuchiya [YAU74] were the first to describe the application of this technique to microcode compaction.

In BAB, a tree is built, the nodes of which correspond to microinstructions. A path from the root of the tree to a leaf is an ordering of MIs, and thus a list of MIs. The tree branches whenever there is more than one microinstruction that could be placed at a point in the list of MIs. A complete tree represents every possible microinstruction ordering.

There are two variants of this algorithm. The first is BAB exhaustive, in which every branch of the tree that could possibly lead to an optimal MB ordering is explored. The other is BAB heuristic, in which pruning is done to the tree. BAB exhaustive is an optimal algorithm, thus running in exponential time, while BAB heuristic is not guaranteed optimal and can be made to run in polynomial time.

The growth of the tree can be bounded even in BAB exhaustive. As in the other algorithms, calculate the lower bound on the number of microinstructions in the best possible ordering. (Remember that this is the longest path through the DDG.) A path through the BAB tree of this length represents an optimal ordering, and the algorithm can stop once such a path is obtained.

The growth of the tree can be further bounded by remembering the length of the best (shortest) path found so far. If the length of an incomplete list of MIs is greater than or equal to this length, the current path needs no further consideration.

Like the critical path algorithm, the BAB algorithm gets its information on the data dependencies of an SLM's MBs through a DDG. The first step of the BAB exhaustive algorithm is the construction of a **data available set (Dset)**. This is the set of all unplaced MBs that are not data dependent on any unplaced MB. The contents of the Dset change as execution of the algorithm progresses. The initial Dset (Dset 1) for our example in Figure 12 is 1, 2, 7 (just those MBs not dependent on any other MB).

The next step is to form microinstructions from the MBs in the Dset. We wish to form only the largest possible MIs. These are called **complete instructions (CIs)**. A complete instruction is defined as an instruction to which no other MB in the Dset can be added (see Section 3). Thus a CI is not a subset of any other legal MI. CIs make up the nodes of the BAB tree so that a path through the tree (a list of MIs) is a list of CIs.

An algorithm for the identification of CIs is found in Section 3. For our example, the CIs are easily identified. For Dset 1, the CIs are 1, 2 (I1) and 1, 7 (I2). MBs 2 and 7 *cannot appear together* because of the conflict between them. MB 1 by itself is a legal microinstruction but not a CI because it is a subset of I1.

At this point it is not known which of the CIs will lead to a better list of MIs. Therefore the tree must be split and each CI dealt with in turn. Figure 19 shows the execution of the BAB exhaustive algorithm. Each part of the figure shows one step. In Figure 19a the initial Dset, the CIs generated from it, and the tree produced by splitting the CIs are shown.

We built the BAB tree depth-first to allow us to find the length of some completed paths which may enable us to cut off some other paths. Let us proceed with the left branch of the tree.

Calculate Dset 2 by first removing the MBs in I1 from Dset 1. This leaves MB 7 in the Dset. Next add the MBs that have become data available to the Dset. At this point MB 3, data dependent on MB 2, becomes data available because MB 2 has been added to the tree. Note that MB 4 does not become data available because it is dependent on MB 3, which is still in the Dset at this point.

Since MBs 3 and 7 conflict, there are two single MB CIs that can be generated from Dset 2: I3, which consists of MB 3, and I4, which consists of MB 7. Split the tree again. Figure 19b shows the state of the tree at this point.

Again proceeding down the left branch, calculate Dset 3. This Dset consists of MBs 4 (dependent on MB 3) and 7. Since these MBs do not conflict, only one CI can be generated from this Dset, which consists of both MBs 4 and 7. Thus we do not branch here. The tree with this node added is shown in Figure 19c.

The new Dset (Dset 4) consists of those MBs dependent on MBs 4 and 7. All the remaining MBs fit this description, so Dset 4 contains MBs 5, 6, and 8. Since MBs 5 and 6 conflict, two CIs are formed from this Dset; one contains MBs 5 and 8, and the other contains MBs 6 and 8. Since there are two CIs, the tree branches, thus producing the tree of Figure 19d.

Pick I6, leaving MB 6 in the Dset. This Dset, Dset 5, obviously forms just one CI, I8. Adding I8 to the tree exhausts the MBs in the SLM, so the path from the root to I8 forms a list of MIs of elements I1, I3, I5, I6, and I8. Figure 19e shows the equivalent list of MIs, which is of length 5 and is optimal. However it does not reach the lower bound of length 4, and since there are more CIs to investigate, execution of the algorithm does not terminate. Mark the leaf of this path with a "P" to indicate that this is the best path found so far.

Now back up to Dset 4 and move down the right path, choosing I7. The new Dset, Dset 6, consists of MB 5, which forms one CI, I9. Add I9 to the tree (Figure 19f), thus exhausting our list of MBs and producing a new list of MIs. Since this list of MIs is of length 5, which is no better than the first one produced, we throw this list of MIs away—indicated by placing an "X" below node I9.

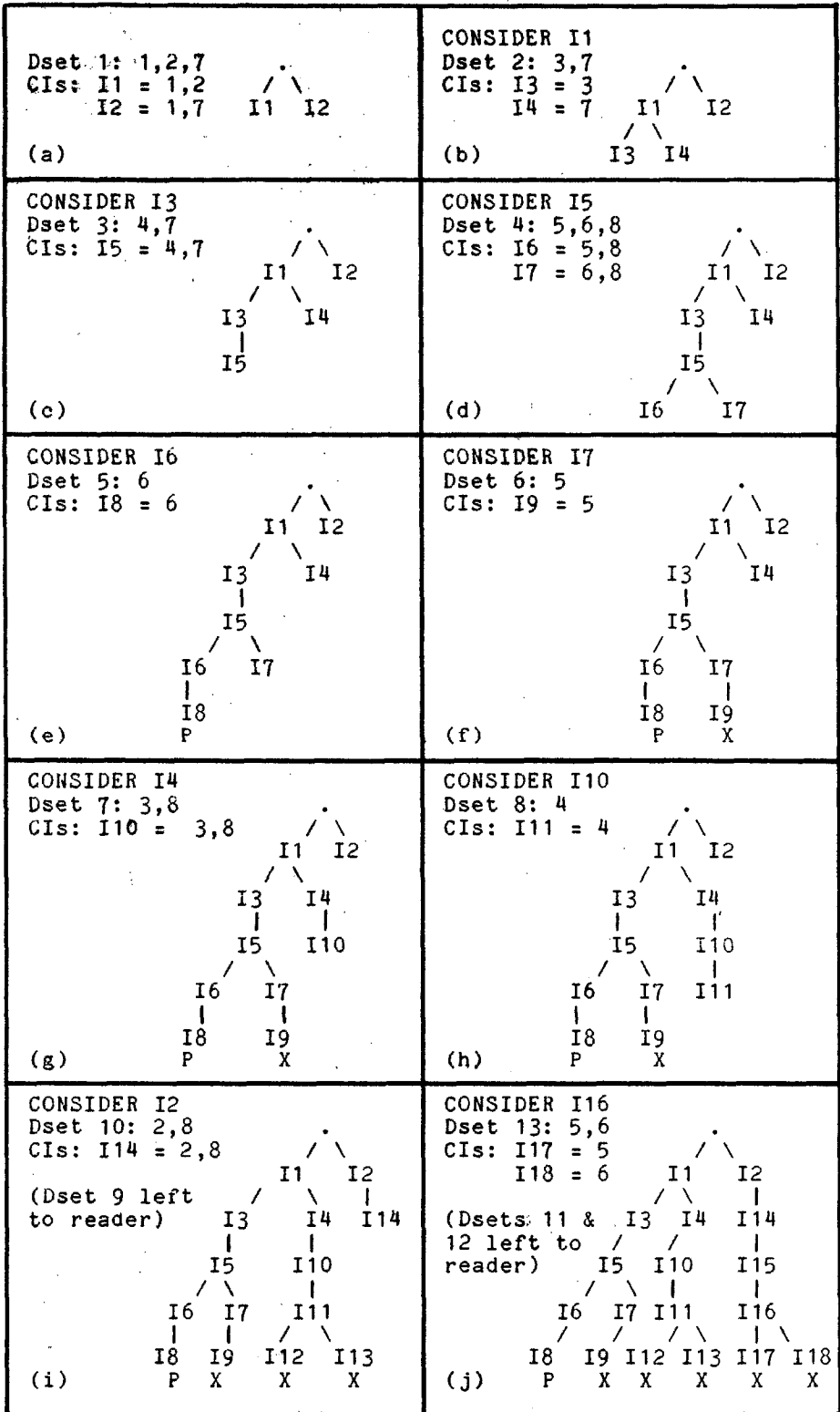The next incomplete path ends with node

FIGURE 19. BAB exhaustive execution of the example in Figure 12.

I4. Remember from Figure 19b that the Dset when this node was added to the tree was 3, 7. I4 corresponds to the choice of the CI consisting of MB 7. Thus the remaining Dset consists of MB 3. The new Dset, Dset 7, consists of the MBs dependent on MB 7 (MB 8) and MB 3. Dset 7 is shown in Figure 19g. Only one CI can be formed from this Dset. This instruction, I10, consists of both MBs 3 and 8. It is added to the tree after I4, as shown in Figure 19g.

Figure 19h shows the addition of the next CI, I11. There is nothing new in this step, or until the state shown by the tree in Figure 19i. I12 consists of MB 5. At this point we have placed all MBs except MB 6 in 5 MIs. Since we have equaled the best path length, with one more MB to place, we know we can do no better than the list of MIs shown in Figure 19e, and we cut off this path. The same holds for the path ending with I13, except that here we have placed all but MB 5.

Next back up to I2, constructing I14 out of Dset 10 as shown in Figure 19i. The end of this path, and the complete BAB tree, is shown in Figure 19j. As in the case of the path ending with I12 and I13, at nodes I17 and I18 we have equaled the best path length so far without placing all of the MBs in the SLM. Therefore we can cut off, as shown by the X's under nodes I17 and I18. At this point there are no more incomplete paths, and the algorithm has completed. The list of CIs found is the one shown in Figure 20. This list of CIs is of optimal length.

In BAB weak dependencies are handled by the FormCIs algorithm.

CI	MBs
I1	1,2
I3	3
I5	4,7
I6	5,8
I8	6

FIGURE 20. The solution found in Figure 19.

4.4 Branch and Bound Heuristics and List Scheduling

Since the branch and bound algorithm considers all possible combinations of MBs, it is guaranteed to produce an optimally improved microprogram. The cost, however, is exponential since all of the paths of the tree must be explored. It is possible to find a heuristic that will prune the BAB tree without affecting the resulting microprogram too adversely. Such a heuristic will, of course, not always produce optimal results, but it should save a great deal of time. Tests of different heuristics have been reported by Mallett [MALL78].

Consider the following heuristic: Instead of examining each complete instruction generated from a Dset, examine only the "best" CI, where the best instruction is determined by some metric. This heuristic requires only an amount of time that grows polynomially with the number of MBs in the SLM.

An algorithm that executes in this manner is an example of a list scheduling algorithm [ADAM74, FISH79]. Although list scheduling algorithms can be considered a special case of branch and bound algorithms, they are important in their own right. A list scheduling algorithm can be programmed much more easily than a full branch and bound algorithm. The tree of CIs is just one branch, and no bounds checking need be done. Furthermore, the complexity of FormCIs is greatly reduced, as noted in Section 3.2.

MB	Weight
1	3
2	4
3	3
4	2
5	0
6	0
7	1
8	0

FIGURE 21. MB weights for the example in Figure 12 (number of descendants in DDG).

The weighting function used in forming the complete instruction affects the optimality of the compaction. Extensive tests of different functions were reported by Fisher [FISH79]. We present an example using a function proposed by Wood [WOOD78].

The metric used by Wood is as follows. Assign a weight to each MB in the DDG. The weight of an MB is the number of descendants of that MB (direct or indirect)

Dset	Contents	Complete Inst.	List of MIs
1	1,2,7	2,1	2,1
2	3,7	3	2,1 3
3	4,7	4,7	2,1 3 4,7
4	5,6,8	5,8 (First-Come, First-Served)	2,1 3 4,7 5,8
5	6	6	2,1 3 4,7 5,8 6

FIGURE 22.   List scheduling execution of the example in Figure 12.

in the DDG. Figure 21 gives the weights of the MBs in our example. MB 1 has a weight of 3, since MBs 4, 5, and 6 are its descendants. MB 2 has a weight of 4, since its descendants are 3, 4, 5, and 6. MBs with no descendants, such as MB 8, are assigned a weight of 0.

Execution proceeds as follows (see Figure 22). Compute a Dset from the DDG. Find the MB in the Dset with the highest weight. Add it to the MI, which is initially empty. Then find the MB with the second highest weight. If this MB does not conflict with any MB already in the MI, add it. Repeat until all the MBs in the Dset have been examined. The resulting MI is the CI with the highest weight. Add this MI to the list of MIs and compute a new Dset. Repeat until all of the MBs in the SLM have been placed.

In Figure 22, Dset 1 contains MBs 1, 2, and 7. MB 2 is the MB of highest weight and is placed first in the CI. MB 1 has the second highest weight and can be added to the CI without conflict. The remaining MB in the Dset, MB 7, cannot be added to the CI because of a conflict between MBs 2 and 7, and so the first instruction in the list of MIs is 2, 1. The construction of the rest of the list of MIs is straightforward. In this case the final list of MIs is of length 5, is optimal, and is exactly the same as the list of MIs produced by the exhaustive variant of BAB.

4.5 Computational Complexity

In the past, the microcode compaction problem has had the reputation of being very complex [AGER76]. It was widely believed that a program for automatic compaction would be too expensive to run in a production environment. This reputation is well deserved in the optimal case. Compaction is in a class of difficult problems which includes such well-known elements as the "traveling salesman problem." But as with many of these problems, there exist practical algorithms that can produce acceptable results for many applications.

NP is a well-known class of "difficult" problems [MACH78]. Problems in this class can be solved on a nondeterministic Turing machine in polynomial time. Certain problems in NP are NP complete. If a polynomial time (deterministic) algorithm can be found to solve any problem that is NP complete, then a polynomial time algorithm can be found to solve every problem in the class. Since an algorithm with less than exponential time has never been found for an NP-complete problem, it is widely believed that all NP-complete problems are of at least exponential complexity. We now show that microcode compaction is at least as difficult as NP-complete problems.

In ULLM73 the unit time scheduling problem is shown to be NP complete. This problem is as follows:

Given

(1) a set $S = \{J1, \ldots, Jn\}$ of jobs,
(2) a partial order $\ll$ on S,
(3) that each job requires one time unit, and
(4) a number of processors, k

(up to k jobs may be executed at each time unit), minimize $t$max, the total number of time units required, under the constraint that if $J \ll J'$, then J' does not execute until at least one time unit after J.

It is straightforward to restrict our model to this problem. Let the SLM correspond to S, let data dependency be $\ll$, and let the number of fields in an MI be k, where each MB can be in any field. (This same proof is done for a slightly different model in DEWI76.) Thus, if microcode compaction can be solved in less than exponential time, so can all problems in NP complete.

An alternate approach to establishing the exponential complexity of microcode compaction can be found in YAU74. It should be pointed out that some articles in the literature have claimed to have less-than-exponential solutions to the optimal compaction problem. All of these claims have been shown to be invalid.

Nonoptimal algorithms do not have to be exponential. The critical path, linear, and list scheduling algorithms work in polynomial time. It should be noted that the name linear does not imply that this algorithm has linear complexity but rather that the SLM is processed one MB at a time.

That the linear algorithm runs in polynomial time can be shown by considering the worst case SLM, wherein there are no data dependencies between MBs and where each MB conflicts with each other MB. In this case each already placed MB must be examined in order to ascertain that the new MB is not data dependent on any MB in the list of MIs. That is, $i - 1$ checks for data dependency must be done for the ith MB. The total number of checks done for an n MB SLM is therefore $(n - 1)n/2$. Each already-placed MB must be checked for conflict during the search for an MI to which the MB can be added. Again, for an n MB SLM, $(n - 1)n/2$ checks for conflicts must be done. Therefore $n^2 - n$ comparisons must be done between MBs in this worst case, showing that the complexity is $O(n^2)$.

A branch and bound algorithm with suitable heuristics is competitive with the other algorithms.

Extensive tests which support these conclusions are reported in MALL78. In one test of an SLM of 96 MBs (231 MOs), for example, Lin compacted to 55 MIs using 309 MO-to-MO comparisons, CPath compacted to 54 MIs using 376 comparisons, and a BAB heuristic restricted to list scheduling compacted to the optimal 50 MIs using 491 comparisons. The execution times on a Honeywell 68/80 processor (MULTICS) were 6.7 seconds, 11.8 seconds, and 7.5 seconds, respectively. By contrast, an exhaustive BAB compaction required 183,364 comparisons to find the optimal 50 MIs and ran in 2629 seconds.

One practical factor affecting the execution times of compaction programs is often overlooked. Because branches and entry points (i.e., labels) are common in microprograms, the typical SLM is not very long.

4.6 Register Allocation

The previous sections on compaction algorithms do not address the problem of allocating registers. This makes the presentations easier to understand. Furthermore, many applications do not need the compaction routine to allocate registers. Spurred by the rapid drop in the cost of high-speed memory, the trend in computer architecture is to include more and more registers for microprogramming. If there are more registers than microprogram variables, each variable can be allocated a register before compaction, and reallocation need never be done.

The register allocation problem is as follows. In addition to process registers, most microprogrammable processors have a local store. Data can be moved back and forth between the registers and local store. If there are more variables than registers, registers must be reallocated during the execution of the program. To reallocate a register, the value of the old variable must be saved in local store (if the register has changed value), and the value of the new variable must be loaded from local store (if it is needed). The difficulty with register

reallocation is deciding which variables to reallocate when a variable needs a register and there are no more unallocated (free) registers. The wrong choice of variables can necessitate extra reallocations later.

Register allocation can be incorporated into the compaction algorithms presented here by considering variables as unbound resources. The variable-to-register binding must be maintained in a separate data structure. Whenever a register reallocation becomes necessary, the different possible reallocations correspond to different MB versions. Store and load microoperations must also be added to perform the reallocation. The register allocation problem is discussed at length in DEWI76.

It should be noted that register allocation also affects global analysis. Since register allocations are maintained between SLMs as well as within them, an SLM is often constrained by initial and final register allocations. Under this constraint the order in which SLM compactions are performed is important. An unconstrained SLM may compact to fewer MIs than its constrained counterpart. Thus the most critical SLMs (inside loops, say) should be compacted first.

5. CONCLUSIONS

Linear analysis compaction algorithms are easy to write, and they run in a reasonable amount of time. However they can only operate on SLMs in sequential form, and their results depend on the order of the microoperation bundles in the SLM. It is easy to construct an example for which linear algorithms produce more MIs than necessary.

Critical path, branch and bound, and list scheduling algorithms operate on the SLM in graph form. Having to build a graph does not significantly increase execution time. The graph building program is also easy to write, although the compaction algorithms themselves are not straightforward to program. Critical path algorithms have difficulties with the rigid boundaries they impose between frames. These kinds of algorithms do not seem to be the best approach for microcode compaction.

Branch and bound algorithms can be exhaustively run to find optimum solutions (for the minimization of the number of MIs for the SLM), but doing so is prohibitively expensive. List scheduling algorithms and branch and bound algorithms with suitable heuristics can compact approximately as fast as linear analysis algorithms and are not nearly as dependent on the original order of the microoperations. Furthermore the probability of an optimal branch and bound compaction can be adjusted by changing the heuristics. List scheduling programs are easier to write than branch and bound programs.

These algorithms can work with realistic processor models. It is true, however, that realistic hardware details make the algorithms harder to program. In particular, whenever microoperations have a choice of resources (registers, fields, etc.) where the choice can have an adverse effect on later execution, the complexity of the compaction problem increases. Further investigation of modeling processor control is needed.

Exhaustive comparisons of the compaction algorithms have shown that all four kinds are capable of producing acceptable compactions in a reasonable amount of time. Often the small size of an SLM makes local compaction trivial. Although many issues merit further study, especially with regard to global compaction, practical compilation of horizontal microcode is feasible.

GLOSSARY

Compaction—Production of a sequence of microinstructions that is semantically equivalent to a given ordered set of microoperations. Most compactions try to minimize execution time.

Compatible—Having no resource conflicts.

Complete instruction (CI)—Instruction to which no other MB in the Dset can be added.

Conflict—Attempt to share an unsharable resource.

Data available—Addition of an MO to a particular MI in a list of MIs does not violate data integrity.

Data dependent (dd)—mo_i dd mo_j if mo_j is directly data dependent either on mo_i or on an MO which is itself data dependent on mo_i.

Data dependency graph (DDG)—Graphical representation of a partial order over the MOs of an SLM.

Data independent—Not data dependent.

Data interaction—Two MOs have a data interaction if an input operand of one is also an output operand of the other or if they have an output operand in common.

Directly data dependent (ddd)—mo_i ddd mo_j if mo_i precedes mo_j in the given MO ordering, if mo_i and mo_j have a data interaction, and if no nonempty chain of directly data dependent MOs exists between them.

Field—Collection of bits in the control word that controls a primitive machine activity.

Global analysis—Analysis that considers more than one SLM.

Horizontal architecture—Architecture that allows many MOs in the same MI.

Host machine—Computer that a given microprogram is designed to control.

Local analysis—Analysis that considers one SLM.

Microinstruction (MI)—Specification of all MOs that are to start during one machine cycle, i.e., a conflict-free set of MOs.

Microoperation (MO)—Specification of a primitive machine operation.

Microoperation bundle (MB)—A set of one or more MOs that must be executed in the same MI.

Microprogram—Sequence of microinstructions.

Remaining Dset (RDset)—Set of all remaining MBs.

Remaining MB—MB in a Dset that has not yet been included in or excluded from the current microinstruction.

Order preserving—Two MOs are order preserving if they can be placed in the same MI (ignoring resource conflicts) such that for every resource causing a data interaction between them, the MO that is earlier in the SLM finishes with the resource before the other MO starts to use it.

Parallel—Two MOs which are order preserving and conflict-free are parallel.

Partial instruction—Microinstruction for which remaining MBs exist.

Polyphase timing—An MI cycle is composed of distinct phases; MOs are active only in specified phases.

Potentially complete instruction—Partial instruction that is not a subinstruction to any known complete instruction.

Straight-line microcode section (SLM)—Ordered collection of MOs with no entry points except at the beginning and no branches except possibly at the end.

Strongly dependent (sd)—If mo_i ddd mo_j and it is known that mo_j is not weakly dependent on mo_i, then mo_j is strongly dependent on mo_i (mo_i sd mo_j).

Subinstruction—A partial instruction is a subinstruction to an instruction if the union of the partial instruction and all MBs that can potentially be added to it is a subset of the instruction.

Transitory-data resource—Storage resource whose contents can become undefined at the termination of an MI in which it has been given a value.

Vertical architecture—One that allows only a few MOs, or one MO, in an MI.

Weakly dependent (wd)—Given two MOs, mo_i and mo_j, then mo_j is weakly dependent on mo_i (mo_i wd mo_j) if mo_j is directly data dependent on mo_i, and if for every resource causing a data interaction between them, mo_i finishes with the resource before mo_j starts to use it.

REFERENCES

ADAM74   ADAM, T. L., CHANDY, K. M., AND DICKSON, J. R.   "A comparison of list schedules for parallel processing systems," *Commun. ACM* **17**, 12 (Dec. 1974), 685–690.

AGER76   AGERWALA, T.   "Microprogram optimization: A survey," *IEEE Trans. Comput.* **C-25**, 10 (Oct. 1976), 962–973.

AGRA76   AGRAWALA, A. K., AND RAUSCHER, T. G.   *Foundations of microprogramming*, Academic Press, New York, 1976.

BARN78   BARNES, G. E.   "Comments on the identification of maximal parallelism in straight-line microprograms," *IEEE Trans. Comput.* **C-27**, 3 (March 1978), 286–287.

COFF76   COFFMAN, E. G., JR., ED.   *Computer and job-shop scheduling theory*, Wiley, New York, 1976.

DASG76   DASGUPTA, S., AND TARTAR, J. "The identification of maximal parallelism in straight-line microprograms," *IEEE Trans. Comput.* C-25, 10 (Oct. 1976), 986–992.

DASG78   DASGUPTA, S. "Comment on the identification of maximal parallelism in straight-line microprograms," *IEEE Trans. Comput.* C-27, 3 (March 1978), 285–286.

DASG79   DASGUPTA, S. "The organization of microprogram stores," *ACM Comput. Surv.* 11, 1 (March 1979), 39–65.

DAVI78   DAVIDSON, S., AND SHRIVER, B. D. "An overview of firmware engineering," *Computer* 11, 5 (May 1978), 21–33.

DEWI75   DEWITT, D. J. "A control word model for detecting conflicts between microoperations," in *Proc. 8th Annual Workshop on Microprogramming* (ACM), 1975, pp. 6–12.

DEWI76   DEWITT, D. J. "A machine-independent approach to the production of horizontal microcode," Ph.D. dissertation, Univ. Michigan, Ann Arbor, June 1976; Tech. Rep. 76 DT4, Aug. 1976.

FISH79   FISHER, J. A. *The optimization of horizontal microcode within and beyond basic blocks: An application of processor scheduling with resources*, U.S. Dept. of Energy Report, Mathematics and Computing C00-3077-161, New York Univ., Oct. 1979.

GONZ77   GONZALEZ, M. J., JR. "Deterministic processor scheduling," *ACM Comput. Surv.* 9, 3 (Sept. 1977), 173–204.

KLEI71   KLEIR, R. L., AND RAMAMOORTHY, C. V. "Optimization strategies for microprograms," *IEEE Trans. Comput.* C-20, 7 (July 1971), 783–795.

KLEI74   KLEIR, R. L. "A representation for the analysis of microprogram operation," in *Proc. 7th Annual Workshop on Microprogramming* (ACM), 1974.

MACH78   MACHTEY, M., AND YOUNG, P. *An introduction to the general theory of algorithms*, North-Holland, New York, 1978.

MALL75   MALLETT, P. W., AND LEWIS, T. G. "Considerations for implementing a high level microprogramming language translation system," *Computer* 8, 8 (Aug. 1975), 40–52.

MALL78   MALLETT, P. W. "Methods of compacting microprograms," Ph.D. dissertation, Univ. Southwestern Louisiana, Lafayette, Dec. 1978.

RAMA69   RAMAMOORTHY, C. V., AND GONZALEZ, M. J., JR. "A survey of techniques for recognizing parallel processable streams in computer programs," in *Proc. 1969 AFIPS Fall Joint Computer Conf.*, AFIPS Press, Arlington, Va., pp. 1–15.

RAMA74   RAMAMOORTHY, C. V., AND TSUCHIYA, M. "A high-level language for horizontal microprogramming," *IEEE Trans. Comput.* C-23, 8 (Aug. 1974), 791–801.

SHRI73   SHRIVER, B. D. "A description of the MATHILDA system," *DAIMI PB-13*, Univ. Aarhus, Denmark, 1973.

TABA74   TABANDEH, M., AND RAMAMOORTHY, C. V. "Execution time and memory optimization in microprograms," in *Proc. 7th Annual Workshop on Microprogramming* (ACM), 1974.

TOKO77   TOKORO, M., TAMURA, E., TAKASE, K., AND TAMURA, K. "An approach to microprogram optimization considering resource occupancy and instruction formats," in *Proc. 10th Annual Workshop on Microprogramming* (ACM), Oct. 1977, pp. 92–108.

TOKO78   TOKORO, M., TAKIZUKA, T., TAMURA, E., AND YAMAURA, I. "A technique of global optimization of microprograms," in *Proc. 11th Annual Workshop on Microprogramming* (ACM), 1978, pp. 41–50.

TSUC74   TSUCHIYA, M., AND GONZALEZ, M. J., JR. "An approach to optimization of horizontal microprograms," in *Proc. 7th Annual Workshop on Microprogramming* (ACM), 1974, pp. 85–90.

ULLM73   ULLMAN, J. D. "Polynomial complete scheduling problems," in *Proc. 4th Symposium on Operating Systems Principles*; *ACM Operating Syst. Rev.* 7, 4 (Oct. 1973), 96–101.

WOOD78   WOOD, G. "On the packing of micro-operations into micro-instruction words," in *Proc. 11th Annual Workshop on Microprogramming*; *SIGMICRO Newsletter* 9, 4 (Dec. 1978), 51–55.

WOOD79   WOOD, W. G. "The computer aided design of microprograms," Ph.D. dissertation, Univ. Edinburgh, Scotland, 1979.

YAU74   YAU, S. S., SCHOWE, A. C., AND TSUCHIYA, M. "On storage optimization for horizontal microprograms," in *Proc. 7th Annual Workshop on Microprogramming* (ACM), 1974, pp. 98–106.