AN ABSTRACT OF THE THESIS OF

Thit Siriboon for the degree of Doctor of Philosophy in Electrical and Computer Engineering presented on March 11, 1988

Title: Integrated Local and Global Microprogram Reduction

Redacted for privacy

Abstract approved: _____

V. M. Powers

Compaction of a horizontal microprogram is an attempt to exploit fully the parallelism permitted by a given algorithm and data processing structure in order to produce a short microprogram. Both viewpoints of the problem: the local compaction viewpoint which deals with compaction within a straight-line segment, or a basic block; and the global compaction viewpoint which deals with compaction across basic blocks are explored. This dissertation is presented as a composite of three papers addressing these viewpoints.

The first paper models the interactions of data dependencies with resource limitations in a manner which allows the designer to select the compaction algorithm best suited to the problem architecture and design goals. A composite graph model of the problem is shown to support a range of local compaction models and methods. Optimal solution searches and faster heuristic-based

compactions are supported by the composite graph.

The second paper describes a global compaction technique called KLITS, which is a heuristic iteration of microoperation movements. This technique can accommodate any of several measures of optimization, and retains the block structure of the original. KLITS searches for globally profitable sequences of microoperation movements even when the first few movements of some sequences do not yield immediate profit. Performance on some previously published example microprograms is encouraging.

The third paper describes tests of the KLITS algorithm. Microprograms are modelled as pseudo-microprograms. Both pseudo-microprogram generation and KLITS global compaction are implemented. For each pseudo-microprogram generated, both local compaction and the KLITS global compaction technique are applied.

As the number of basic blocks in a pseudo-microprogram increases, the pseudo-microprogram compacted by the KLITS algorithm increasingly becomes more compact than one compacted by local compaction. The experiments also suggest that the compaction time grows slowly enough for the algorithm to be used practically.

Integrated Local and Global Microprogram Reduction

by

Thit Siriboon

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Completed March 11, 1988

Commencement June 1988

APPROVED:

Redacted for privacy

Professor of Electrical and Computer Engineering in charge of major

Redacted for privacy

Head of department of Electrical and Computer Engineering

Redacted for privacy

Dean of Graduate School

Date thesis is presented March 11, 1988

Typed by researcher for Thit Siriboon

©Copyright by Thit Siriboon
March 11, 1988

All Rights Reserved

Acknowledgements

I owe a great debt to my major advisor, Dr. Victor Powers, for his encouragement and guidance though out my study and research. I would like to express my sincerely gratitude to the other member of my committee, Dr. Paul Cull, Dr. John Saugen, Dr. James Herzog, and Dr. William Firey. All have been more than generous with their time and comments which have help improve this thesis.

I would like to give my special thank to the department of Electrical & Computer Engineering of O.S.U., especaially to the chairman, Dr. SJT Owen; for providing me with the financial support and accessing to the department facility.

Finally, I wish to dedicate this thesis to my beloved parents : my father, Capt. Pakorb Siriboon and my mother, Mrs. Duangnate Siriboon; who always give me their unlimited confidence, love and understanding which greatly help me finish this thesis.

TABLE OF CONTENTS

Chapter I Introduction 1

Chapter II Integrated Constraints in Local Microprogram 5

Abstract 5

1. Introduction 6
2. Basic terminology 8
   - 2.1. Microprograms 8
   - 2.2. Example 10
   - 2.3. Local and global reduction 12
   - 2.4. Data dependency 14
   - 2.5. Resource usage 17
3. Local Compaction 18
   - 3.1. Exhaustive local compaction 19
   - 3.2. Heuristic local compaction 22
   - 3.3. Critical path example 26
4. Microprogram structure 30
   - 4.1. Introduction 30
   - 4.2. The microprogram partition 32
   - 4.3. Constrained scheduling of microoperations 36
5. Composite model 40
   - 5.1. Basic approach 40
   - 5.2. A composite graph model 41
   - 5.3. Microprogram selection 43
   - 5.4. Microprogram composition summary 46
     - 5.4.1. Example 47
     - 5.4.2. A non-deterministic approach 50
6. Summary 52
7. References 55

Chapter III The KLITS Microprogram Global Compaction Algorithm 58
Abstract 58
1. Introduction 60
2. Basic concepts 61
3. Previous work 62
3.1. The symmetric pairs method 62
3.2. MORIF compaction 62
3.3. Trace scheduling 63
3.4. Tree compaction 68
3.5. SRDAG compaction 69
3.6. Improved trace scheduling compaction (ITSC) 70
3.7. The generalized data dependency graph (GDDG) method 75
3.8. Rao and Majumdar's algorithm 77
4. The Kernighan-Lin partitioning algorithm 80
5. The KLITS algorithm 82
5.1. Moving rules 82
5.1.1. Moving up 85
5.1.2. Moving down 86
5.1.3. Additional restriction 86
5.2. Cost functions 87
5.3. The algorithm 88
6. Detailed example 92
7. Summary and conclusions 109
8. References 112

Chapter IV Experiments on the KLITS Algorithm 115
Abstract 115
1. Introduction 117
2. The model of microprogram used in the simulation (pseudo-microprogram) 118
2.1. The pseudo-microoperations 118
2.2. The control flow structure 119
3. Generation of the pseudo-microprogram 120
3.1. Generation of the control flow structure 121
3.1.1. An example 123
3.2. Generation of the pseudo-micro-operations 125
4. Compaction 130

4.1. The cost function 131
4.2. The moving rules 134
4.3. The algorithm (global compaction of the PMP) 134
5. Experiments on previous examples 139
5.1. Tree compaction example [LAH83] 139
5.2. Rao's example [RAO86] 140
6. Experiments on pseudo-microprograms 141
6.1. Experimental experience 142
6.2. Statistical significance of the experiments' results 144
6.2.1 Relationship between the population mean (μ) and the experiment mean ($\bar{X}$) 144
6.2.2 Relationship between the population variance (σ^2) and the experiment variance (s^2) 145
6.2.3 Statistical significance summary 145
6.3. Discussion 146
6.3.1 Improved Compaction 146
6.3.2 Timing 150
7. Summary and conclusions 153
7.1. Summary 153
7.2. Future work 155
7.3. Conclusions 158
8. References 160
9. Appendix A. 162

Chapter V Bibliography 167

LIST OF FIGURES

Figure II.1. Example of a data path 10

Figure II.2. Example microprogram specification. 28

Figure II.3. Critical path solution 29

Figure II.4. Cross-over problem example 32

Figure II.5. Covering problem for a microprogram 45

Figure II.6. Microprogram composition example 49

Figure III.1 (a). The basic blocks structure for 65

Figure III.1 (b). Fisher's moving rules 65

Figure III.2 (a). Su's comditional jumps example 72

Figure III.2 (b). Su's moving rules 73

Figure III.3. Basic blocks structure transfromation example 84

Figure III.4. An example microprogram from [Lah83] 93

Figure III.5. Detailed example steps group 1 95

Figure III.6. Detailed example steps group 2 101

Figure III.7 Detailed Example Steps Group 3 103

Figure III.8. Comparison summary 106

Figure III.9. Compaction results 107
(a) KLITS Compaction Results 107
(b) Tree Compaction Result 107
(c) Trace Scheduling Result 107
(d) List Scheduling Local Compaction Result 107
Figure IV.1. An example of basic blocks Lay out generation 124
Figure IV.2. An example of control flow generation 126
Figure IV.3. Number of PMIs eliminated by KLITS over local compaction 147
Figure IV.4. Percentage of PMIs eliminated by KLITS over local compaction 149
Figure IV.5. KLITS experiments compaction time 151
Figure IV.6. KLITS experiments average compaction time of each number of PMOs 152

INTEGRATED LOCAL AND GLOBAL MICROPROGRAM REDUCTION

Chapter I Introduction

In this dissertation, we address the problem of microprogram compaction from both local and global viewpoints. The dissertation is organized as a composite of three papers.

The first paper explains the relationship of two parts of the problem of local microprogram reduction. The compaction problem is the production of a small, horizontal coding for a microprogram, or a subblock, which has no internal entries or conditional branches. The two parts of the problem which need integration are satisfying the two categories of constraints. The semantics of the microprogram induce a partial order or dependency which limits the order in which component microoperations may be coded. The limited data path resources available to act out the microoperations further constrain the simultaneity of microoperations. The dependencies and resource limits both act to limit from below the ultimate length of the compacted microprogram.

The usefulness of this exposition is not limited to one algorithm or one application. The internal structure of a microprogram basic block is explained as induced by the constraints. Furthermore, a model is shown which can support the formation of a compact microprogram. Because of the computational complexity of the problem of searching for a best compaction, and because different applications might require different cost tradeoffs among the length and width parameters, choice of a single best compaction algorithm for applications is outside the scope of the present paper.

In the second paper, the global compaction problem is discussed. Global compaction has been recognized as a major issue in microprogram optimization. One of the most noticeable techniques is Fisher's trace scheduling [FISH79], [FISH81]. However, Fisher's trace scheduling has been criticized for the growth, in the worst case exponential, of memory size due to intensive copying of blocks. Also, trace scheduling usually changes the topological structure of blocks, causing the compacted microprogram to be difficult to debug.

A microprogram can be compacted globally by moving micro-operations, sometimes past basic block boundaries. The topological structure of blocks can be easily controlled and preserved. The growth of memory size can be monitored by the programmer.

> However, "...to find a sequence of very profitable moves, one often has to go through an initial sequence of moves which are either not profitable, or, worse still, actually make the code longer. Locating such a sequence involves abandoning attempts to prune this expensive search tree" [FISH81].

The Kernighan-Lin Partitioning Algorithm, which embodies an optimization strategy that may take "bad" steps (i.e. produce immediate losses) if they will lead to an overall gain later, is described in [ULLM84]. "This algorithm takes time that, experimentally, has been found to grow far more slowly than an exponential" [ULLM84].

A global compaction technique with a similar strategy and the concept of microoperation movement as its kernel, is presented. This technique can accommodate any of several measures of optimization (e.g. the average execution time, the total length of the microprogram). With this technique, the programmer has the ability to control the growth of the memory and the changing of topological structure. An heuristic evaluation function is implemented and used to demonstrated the effectiveness of the technique for reducing the total microprogram length.

In the third paper, the KLITS algorithm is implemented for testing on large amounts of microprograms to better understand its behavior. In order to test the KLITS algorithm on large amounts of microprogram some kind of automatic generation of vertical microprograms is essential. The "microprogram" generated does not need to be a real microprogram, it only needs those characteristics of a real microprogram actually used by compactions. For example, the compaction does not need the specific functions of each microoperation, but the compaction must be able to determine data dependencies and resource constraints among microoperations. A microprogram generated in the experiment is called a pseudo-

microprogram, or PMP. The process of generating a pseudo-microprogram is divided into two parts: the basic block structure generation, and the pseudo-microoperation or PMO generation.

After PMPs are generated, both local compaction (list scheduling) and the KLITS algorithm are used to compact them. The results of both compaction are compared and studied.

Chapter II Integrated Constraints in Local Microprogram

Abstract

Compaction of a horizontal microprogram is an attempt to exploit fully the parallelism permitted by a given algorithm and data processing structure in order to produce a short microprogram. For a straight-line segment, this paper models the interactions of the data dependencies of the given program with the resource limitations of the data path in a manner which allows the designer to select the compaction algorithm best suited to the problem architecture and design goals.

A composite graph model of the problem is shown to support a range of local compaction models and methods. Optimal solutions based on simplified models are seen to be searches among ways to form microprograms from the composite graph. Faster compactions and more complicated models are supported by heuristic-based selection of microprograms.

1. Introduction

This paper explains the relationship of two parts of the problem of local microprogram reduction. The compaction problem is the production of a small, horizontal coding for a microprogram, or a basic block, which has no internal entries or conditional branches. The two parts of the problem which need integration are satisfying the two categories of constraints. The semantics of the microprogram induce a partial order or dependency which limits the order in which component microoperations may be coded. The limited data path resources available to act out the microoperations further constrain the simultaneity of microoperations. The dependencies and resource limits both act to limit from below the ultimate length of the compacted microprogram.

The usefulness of this exposition is not limited to one algorithm or one application. The internal structure of a microprogram basic block is explained as induced by the constraints. Furthermore, a model is shown which can support the formation of a compact microprogram. Because of the computational complexity of the problem of searching for a best compaction, and because different applications might require different cost tradeoffs among the length and width parameters, choice of a single best compaction algorithm for applications is outside the scope of the present paper.

Furthermore, the present work is useful beyond just local compaction of interpretive code from a control store. Of course, the basic blocks of a microprogram may be compacted according to the structure uncovered here. Local compaction can play an important role as a component of global compaction, too. But the basic notion of a microprogram block which is compact in length applies whether or not the control is implemented as a microprogram consisting of stored microinstructions. Thus, the compaction described here could equally correspond to the problem of creating a compact control packet, balancing the placement of microoperation-like control signals in sequence with the multiplicity of resources chosen for the data path. For example, the design of the control of a RISC-style machine is describable in this manner.

The next section defines the terminology of the problem. Section 3 identifies the categories of constraints, and summarizes their effects by describing how previous work has dealt with separate phases of microprogram reduction. Section 4 defines how the constraints influence the structure of a given microprogram. Section 5 presents a composition model; from consideration of the constraint structure, a microprogram can be chosen to meet these constraints. Section 5 also shows an example where the composition model allows an optimum compaction. Section 6 is a summary, with concluding remarks.

2. Basic terminology

2.1. Microprograms

A microprogram directly controls a set of hardware resources organized to process data, and many higher levels of system organization depend on the microprogram for efficiency of implementation. It is often worthwhile to spend some considerable design effort optimizing the microprogram.

The hardware the microprogram controls will herein be called the data path. A typical application may have a data path organized around a few data operators such as ALUs, shifters, incrementers, counters, multiplexers, register and register banks, busses and buffers. We will consider designing a compact microprogram for a fixed algorithm on a fixed data path.

The required control for such a data path will be the appropriate sequence of patterns of control signals to select, gate, and clock the flow of data throughout the data path. For purposes of this paper, we will refer to the smallest unit of control signal as a microorder. Our design unit will be the smallest bundle of microorders which must be scheduled together. Such a bundle is a microoperation (MO). We will assume a clocked sequence of sets of MOs which each act in one clock period. Thus an MO may be a simple

command to increment a counter or complement a register, or it may be a register-transfer level operation to fetch two operands from registers, perform a data operation (such as subtract) and save the result in another register during the appropriate phases of a clock cycle.

The set of MOs which occur together during a control period comprise a microinstruction (MI). In the case that a microinstruction is fetched from a control store, the sequence control (opcode and address functions) will be treated similarly to the control of the data path, and it is considered itself to be effected by MOs. Thus the general idea of a microinstruction applies whether the MI is fetched from a control store, or is generated by some means such as a clocked sequence generator.

A microprogram (MP) is a sequence of microinstructions. If the MIs of a microprogram are (M1,M2,...), then each MI is a subset of all the MO types {a,b,c,...} in the MP. If the same microoperation appears in more than one MI, we may identify the separate appearances of the same signal by numbering the different instances of the same MO type: a1,a2,a3, b1,b2, ... If the microprogram includes conditional jumps, then different control paths through the MP may be followed, depending on data values. A basic block is a longest section of a microprogram with one entry point and one exit point; each MP can be partitioned into a set of basic blocks, and the flow of control in the microprogram can follow a complex path among the basic blocks.

The problem is to construct a microprogram which is efficient in some regard. For now, we focus on the length of each basic block.

2.2. Example

The example presented here is to illustrate some of the definitions mentioned in previous section. The section of the microprogram listed in this example will also be used later to demonstrate local compaction techniques.

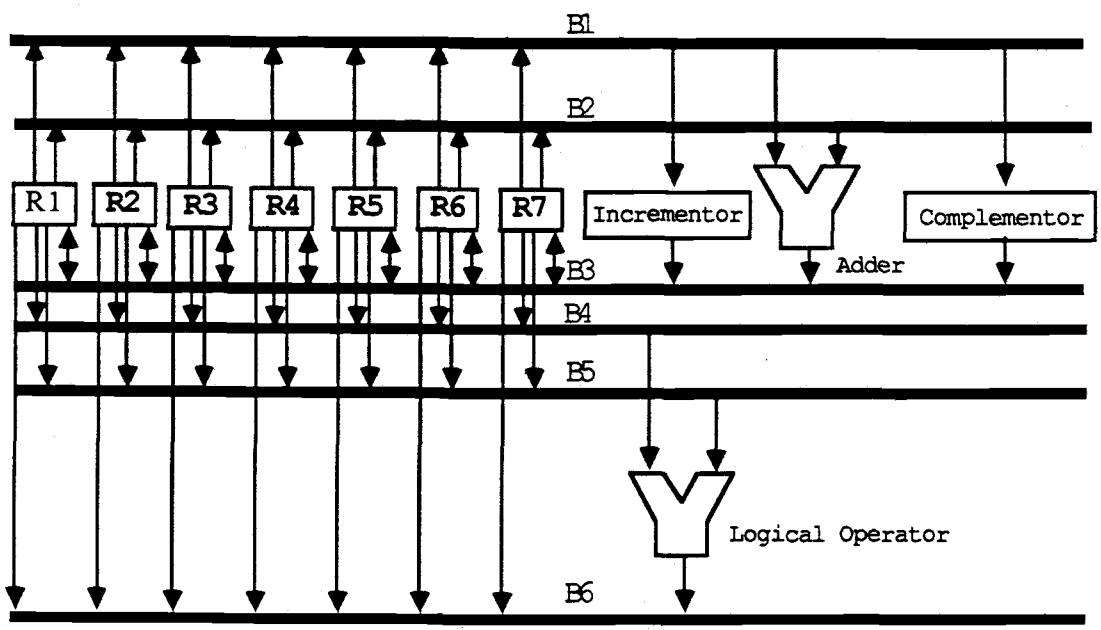


Figure II.1. Example of a data path

From the architecture of a hypothetical data path as shown in Figure II.1, the resources of the machine are:

1) seven registers R1-R7,

2) six buses B1-B6,

3) one incrementor,

4) one complementor,

5) one adder, and

6) one logical operator (**AND, OR, XOR**)

A basic block of a possible microprogram is shown below.

MO's Name	Operation	Resource Usage	Comments
a	R1<--R2 **AND** R3	B4, B5, B6 and the Logical Operator	Logical Operator perfroms **A N D** operation
b	R4<--R4'	B1, B3 and the Complementor	Complement R4
c	R4<--R4^{++}	B1, B3, and the Incrementor	Increment R4 by one (MO b and MO c together perfrom two's complement on R4)
d	R5<--R1+R4	B1, B2, B3 and the Adder	Subtract the original value of R4 in MO c from R1
e	R1<--R5^{++}	B1, B3 and the Incrementor	Increment the result and store it in R1
f	R4<--R5	B3	Store the result in R4

g	R6<--R2+R7	B1, B2, B3 and the Adder	Add R2 and R7 and store the result in R6
h	R7<--0	None	Clear R7

2.3. Local and global reduction

Two major properties which distinguish horizontal micro-program optimization from conventional program optimization are concurrency of microoperations and allocation of microprogram resources [TSUC76]. A compact horizontal microprogram must be "semantically equivalent" to the original microprogram in the sense that both have to produce the same output when given the same input.

It has been traditional to work on two aspects of MP reduction separately [LAND80]: Reduction of the length of a basic block (local compaction), and reduction of a measure of the length, or time of execution, of the global MP. The techniques and concepts addressed in this paper apply to local reduction: that is, to reduction of a basic block. Optimization of the overall microprogram, comprised of a network of basic blocks connected by microprogram control transfers (conditional jumps), is addressed in a separate paper. One useful step in global compaction is to compact one, or each of a set of, basic blocks. One approach to global compaction is to conduct a number of

manipulations of the MP, changing the compositions of basic blocks and compacting them.

Previous work on microprogram basic block compaction has proceeded to establish a microprogram structure by separately applying one constraint (i.e., the critical precedence ordering) and the other constraint (i.e., resource conflict). Separating the problem into two distinct phases allows sequential application of distinct algorithms, but prevents exploiting tradeoffs across boundaries of the phases of the problem. The approach in this paper integrates the two phases of constraint calculations, precedence and resource usage, and proceeds by describing the characteristics of an ordered partition which satisfies both types of constraints and their interaction.

As described later in this section, from a given microprogram or other specification, we have the partial order relation of precedence or data dependency, and the relation of resource conflict (for 1 copy of each resource, this relation is binary and symmetric). The required partial ordering of component microoperation instances forces some minimum separation of MOs even in a horizontal microprogram. In addition, limited facilities or resources in the data path architecture tend to further lengthen the microprogram when microoperations which otherwise could proceed together, must instead be scheduled serially in order to avoid conflict over the use of a resource which each requires.

Each microprogram can be regarded as an ordered partition of the microoperations,

$$P = (M1, M2, \ldots)$$

where each block Mj is a microinstruction, and the microinstruction containing a given microoperation instance, x, is denoted M(x). Packing the microinstructions into a microprogram requires that we satisfy, for each pair of microoperations x and y:

1. precedence: If $x \Leftarrow y$, x precedes y in the precedence relation, then $M(x) \Leftarrow M(y)$, the block of the microprogram containing x precedes the block containing y.
2. resource limits: If x and y conflict in the usage of a resource, then x and y do not appear in the same microinstruction block Mj.
3. interactions: Microoperations are assigned to microinstructions so that neither constraint is violated.

2.4. Data dependency

Definition: MO j is <u>data dependent</u> on (or dependent on) MO i if i must be executed before j in order to preserve data integrity.

The data dependence partial order of MO instances is the transitive extension of pairwise direct dependence. If a microprogram is given as a sequence of operations (typically, highly

vertical), direct dependence is usually determinable by calculations on the input, or read set, I_i, and the output, or write set, O_i, of the data used by each MO i. If j follows i in the given sequence, then we say there exists an ordering hazard if any one of the following three pairs has a nonzero intersection:

1) O_i and I_j ("read after write," or RAW hazard),
2) I_i and O_j ("write after read," or WAR hazard),
3) O_i and O_j ("write after write," or WAW hazard),

Whether or not a hazard constitutes a constraint on the sequencing or concurrency of the MOs involved depends upon the semantics of the program. For example, some or all of the values of O_i and O_j may be irrelevant to the algorithm (unused outputs or side effects). In considering the local compaction problem, we will usually assume that the hazards and their associate semantics have been translated into dependencies. In some places, the methods or the alternative results will make a difference.

In a RAW hazard, if j is executed before i, the only way it can have the right values in its input set is if some operations other than i have left them in the proper state. This semantic condition is only detectable by analyzing the values involved, whether by the techniques of constant unfolding or by symbolic execution with variable values [VEGD82]. We will exclude the corresponding semantic transformation, as others have, from the local compaction problem. Thus, each RAW hazard corresponds to a sequencing constraint; i before j.

Unless they happen to involve equivalent values, switching the order of i and j in the case of a WAR hazard could force i to read a value in I_i that was set by the write of O_j, giving an invalid result.

A WAW hazard also is a constraint only if the values are different (assumed through lack of contrary knowledge, as for RAW). Even so, the "write" of O_i and all of its reads or uses must be complete either before the usage by O_j, or after all the reads of O_j. Unless the "write" value of O_j is not used, violating the MO order, i before j, for a WAW hazard may change the program result.

The most usual model of local compaction makes j data dependent on i if i occurs before j in the vertically coded specification of the microprogram, and if any one of the hazards exists. Another interpretation, which allows more opportunity for concurrency, treats a RAW hazard as a data dependency and WAW and WAR hazards as "antidependencies" [VEGD82].

Depending on the particular application of microprogram reduction, the data dependence may be required to be derived from such I_i/O_i usage, or from some other ordering requirements. Alternatively, the partial ordering may be given as a fully described precedence relation. In any event, in most of the following the relation will be assumed known as one of the forms of constraints on allowable schedules of MOs.

Some subtler sequential constraints among MOs have been noted. Sometimes, due to the relative phasing of read and write actions within a clock period, we can say that one MO must be in a period later than or equal to another. In some designs, there may be multiple signals which need be scheduled in the same MO time. Fortunately, these variety of detailed requirements can generally be accommodated by grouping together as one MO a bundle of coupled control signals [LAND80]. We will proceed assuming that a partial order characterizes the sequencing constraint.

Definition: Two microoperations are parallel executable if there is no data dependency between them.

2.5. Resource usage

Each MO may require the use of one or more resource in the data path, such as an adder, a shifter, a bus or a register. For some means of analysis and scheduling, it is convenient to distinguish among transient uses of a resource (such as a bus with no memory) and resident uses of a resource (such as setting a flag or storage in a register). A set of MOs cannot be executed concurrently if they require more resources than are available in the hardware; such a set are said to have a resource conflict.

More complex hardware interaction will be ignored in the following. For example, a few microinstructions may need to be scheduled in some fixed relative order because of interactions through the hardware; a floating point operation may have a side effect which another MO must handle some fixed number of MI times later. Such tight coupling will be here simplistically assumed to be accounted for by replacing the tightly coupled set by a representative of the bundle, to be scheduled as a single MO.

Two parallel executable microoperations may be executed simultaneously if they have no resource conflict [RAMA74].

Definition: A horizontal microprogram is a sequence of sets of microoperations, called microinstructions, such that:

1) Each microoperation is a member of exactly one set;
2) The microoperations in the same set are parallel executable and have no resource conflict;
3) If MO_i is in the set S, all microoperations which are dependent on MO_i must be in the sets following S in the sequence.

3. Local Compaction

Most published previous works on microprogram improvement center on reports of algorithms or programs to reduce the length of a

microprogram. These action-oriented descriptions often concentrate more on procedures than on models of the microprogram structure. Most of the work done in solving the local compaction problem can be divided into two classes. The first class comprises algorithms which always find the optimal solution. Because the local compaction problem is an NP hard problem [VEGD82], the time required to find the optimal solution, in the worst case, grows exponentially with the number of microoperations being compacted. Therefore, algorithms which intend to find the optimal solution are not appropriate for use on large basic blocks.

The second class of methods for local compaction are ones which might not find the optimal solution. These algorithms use some kind of heuristic techniques which, although they can not be guaranteed to find the optimal solution for all problems, find close to the optimal solution in reasonable time for most expected problems.

3.1. Exhaustive local compaction

Exhaustive search methods for finding the best of all combinations form some of the earliest and some of the most recent references. Apparently the first report on the microcode compaction problem is by Astopas and Plukas [ASTO71]. They calculate all possible microprograms which preserve the data dependency relation; then they select the shortest one which has no resource conflict.

This method is improved by Yau, Schowe, and Tsuchiya [YAU74]. The paper describe a branch and bound algorithm that guarantees an optimal solution. The cost of the algorithm is an exponential function of the number of microoperations to be optimized. The branch and bound algorithms apply the branch and bound technique to a search of a tree of all the possible MI orderings. A branch of the tree is pruned and the search down it stopped, when it is discovered that every possible microprogram length resulting from the current path will be longer than a previously discovered solution, or current least upper bound.

Landskov characterized the general exhaustive search method in a manner that was used by others later to describe refinements [LAND80]. In this formulation, the optimum compaction is found by searching a tree of possible microinstructions. At any point in the tree, a set of data-available MOs (a Dset) enumerates all the MOs which have no as-yet-unscheduled predecessors in the data dependency relation. A selection procedure, FormCI, is required which can produce a set of maximal MIs from this Dset; each such MI is as large as it can be without violating a resource restriction. One more node of the search tree is established for each such distinct MI, the next Dset for that node is constructed, and the search continues. A leaf node is one for which the Dset is empty; all MOs have been scheduled. A microprogram is the sequence of MIs encountered in traversing the tree from root to a leaf. The shortest microprogram corresponds to the path to the shallowest leaf node of the tree. The tree may be completely generated, some pruning may be done such

as by the branch and bound technique, or some heuristics may lead to selecting one or a few "promising" paths through the tree.

Atkins [ATKI84] reports implementation detail and data structures used in a Pascal implementation of the tree search. He explicitly points out that the FormCIs operation can be phrased as an intersection of the Dset with each of a collection of maximal compatible classes (there called "prime implicants") characterizing a binary resource compatibility relation. Also, the computation of a new Dset for a newly formed MI is aided by precomputation of a lookahead or LA set, one for each MO in the precedence ordering.

Henry [HENR83] considers a microprogram as a set of assignments of MOs to MI time slots. A binary relation built among pairs of MO assignments is restricted from violating data dependency. As each MO is placed in a potential microprogram during a tree search of possible microprograms by selecting one assignment allowed by the relation, the relation is further reduced by eliminating all MO assignments which could conflict with this choice. In the paper, the procedure is presented as similar to the "consistent labeling problem" also used in scene analysis.

[VEGD82] provides a dynamic programming algorithm that compacts straight-line microcode in polynomial time when the number of resources is limited.

3.2. Heuristic local compaction

Landskov characterizes the nonoptimal local compaction methods as critical path, linear and list scheduling [LAND80]. Whereas the critical path approach proceeds based upon an analysis of the data dependencies of the entire basic block, the other two forms only examine the dependencies among small subsets of MOs, incrementally chosen as the algorithms proceed. Although some experiments [DAVI81] have shown microprograms in which a critical path algorithm failed to find the optimum compaction, for reasons which are demonstrated below, yet that approach is here considered instructive. For one thing, it initially discovers a key structure which provides an irreducible (though not always achievable) lower bound to the minimum microprogram length. In addition, it shows a structure which can be related to the structure described in a following section. The idea of critical MOs was used early on.

Ramamoorthy and Tsuchiya [RAMA74] present a paper concerning the development of a high level microprogramming language, SIMPL (single identity microprogramming language). Along the way, a non-optimal compaction algorithm using the concept of critical microoperations is presented.

Tsuchiya and Gonzalez [TSUC76] discuss a polynomial-time non-optimal method of local compaction. A careful algorithm is

presented for the general problem, including the updating of a least upper bound of a microprogram. Their algorithm starts by generating the earliest and the latest precedence partitions for the program data dependency graph. The latest partition is modified by grouping microoperations in each level according to their resource usage. For each level, if resource conflict exists, some microoperations are shifted to the additional following level. The microoperation to be shifted is determined by the number of its successors. Then, microoperations are shifted up to any additional levels that do not create resource conflict or violate data dependency. This step is an improvement on their previous version [RAMA74].

The critical path is the longest minimum path in the data precedence graph, and gives a lower bound on the length of the microprogram. Considering only the precedence relation and not the resource usage, the MOs are composed into an Early Partition (EP). Each MO is placed in the earliest block it could be in, and is delayed only to be in the block immediately after an MO on which it directly depends. Similarly, a Late Partition (LP) puts every MO into the latest block it could be in, subject only to preceding any other MO which depends on it. The critical partition is the set of blocks of MOs which are in the same block of the EP as the LP. A critical path is a path through the precedence graph of the MOs in the critical partition.

The length of the critical path is a lower bound on the length of the basic block, but this bound is generally prevented by resource

usage conflicts. The critical path algorithm considers the blocks of the critical partition to be frames of MIs, and the microprogram is developed only by refining the partition and splitting frames into multiple MIs as the other MOs are added. Even the critical path may contain resource conflicts, and frames must be split to allow several MOs to be in different MIs. As each MO not in the critical partition is added one at a time, it may fit in an MI in the range of frames from where that MO was in the EP to where it was in the LP. If it conflicts with others in each such MI, another split of a frame must add another MI. This approach leads to suboptimal results because MIs created in different frames are not merged. An example from the literature of the critical path method is shown in the next section.

Linear and list scheduling methods attempt to reduce the complexity of the optimization search. They sweep once through the microprogram, one item at a time. A linear algorithm goes through a given list of MOs one time. Each MO is taken and used to search the microprogram so far constructed (initially empty) for an MI which accommodates the MO without precedence or resource conflict; if none exists, the MO is placed in a new MI. List scheduling constructs MIs one at a time, attempting at each step to find the incrementally best group of MOs. With either single-sweep method, the size and form of the resulting microprogram can depend on the initial ordering of the list of MOs.

Dasgupta and Tartar [DASG76] provide an algorithm, originally described as optimal. Their algorithm considers microoperations in

their original source order and places them as early as possible. The algorithm is quite complex due to its handling of polyphase microinstructions. The cost of the algorithm is quadratic in the number of microoperations. And, although the algorithm allow levels to be recombined which is an improvement from [RAMA74], its non-optimality is shown later [DASG78].

List scheduling [FISH79] uses a strong heuristic to thread through the search tree; at each step, some heuristic selects which maximal MI is to be formed as the next MI. Fisher compared several alternate priority evaluation criteria for selecting the next MI. In addition to the work on optimal compaction mentioned above, Yau *et al.* [YAU74] also present a heuristic criterion that chooses the microinstruction with the largest number of successors as being the best complete microinstruction from the data available set at each node of the search tree. With this heuristic, the cost of the algorithm decreases to a polynomial function of the number of microoperations. However, the algorithm can no longer guarantee the optimal solution every time.

Davidson, *et al.* [DAVI81] compared critical path, linear (or first-come, first-served, FCFS) and heuristic tree searching algorithms for three microprograms with basic blocks of tens to hundreds of MOs. Both FCFS and tree searching simplified to the equivalent of list scheduling were judged to be reasonable in time used, producing very good results.

The method described by Tokoro *et al.* [TOKO81] uses a complex description of each MO as a "microtemplate" in two dimensions (timing and resources). Against this more elaborate MO model they show a local compaction method which iteratively chooses MIs from the given sequence of MOs. Each MI is chosen by two steps. "Compaction" finds parallel executable MO sets. MOs first are taken in the order of their first specification. Concurrency is sought by moving an individual MO forward in time until it is limited by data dependency conflict. "Packing" chooses for the next MI a subset of one of those sets, which is free of resource conflicts; a heuristic selects as the next MI the set containing the largest number of critical MOs [TSU76]. Provision is made to back up when necessary to include certain MOs.

Tokoro's technique, called MORIF (microprogram optimization technique considering resources occupancy and instruction formats), includes both local and global optimization. For global optimization, microtemplates are transferred between straight-line segments in which they can be more compacted. Rules for transferring are presented.

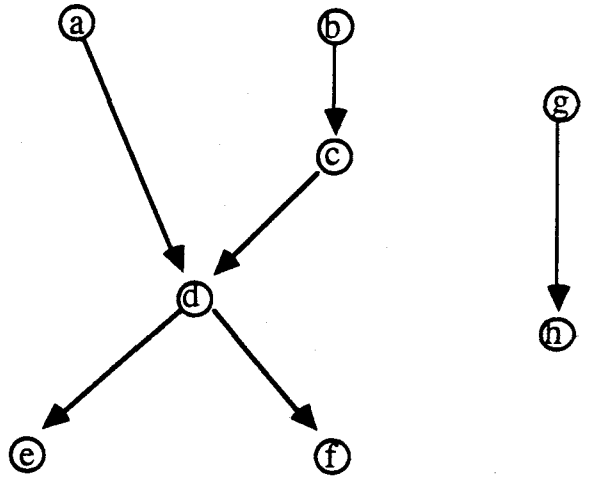
3.3. Critical path example

The critical path method is shown summarized below, as applied to the example of section 2.2. This example has the same

precedence relation graph and resource conflicts as the example of [LAND80]. Figure II.2 shows a microprogram specification in terms of the precedence relation on eight microoperations and a list of resource conflicts. Figure II.3 shows steps in using the critical path in constructing a microprogram. We will return to this same example in Section 5, with a different approach.

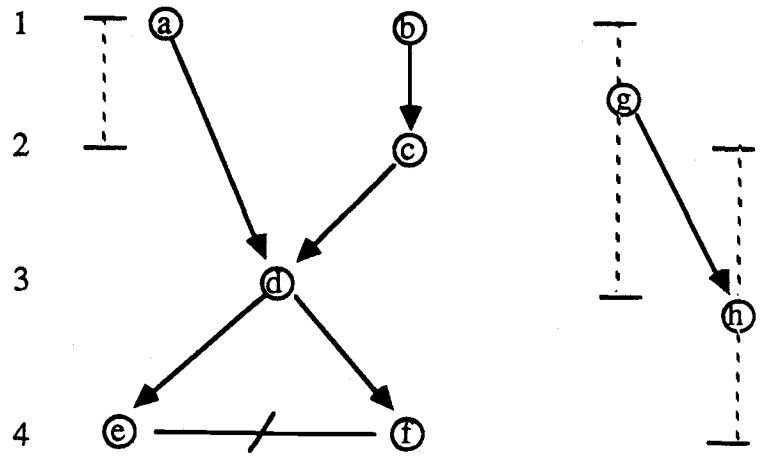
Figure II.3(a) shows each MO as located in a range of positions, from earliest to latest, as determined by the EP and LP mentioned above. In that Figure, a vertical dashed line indicates the extent of range. Any MO whose range covers only the one point is a point in the critical path. In Figure II.3(a), the critical path, as determined only from examination of the precedence relation, is: b; c; d; e,f. The microprogram is derived by adding the other MOs and splitting incompatible MIs. First, as indicated in Figure II.3(a) by the hash mark, the MOs of the critical path are checked for resource compatibility. In this case, MOs e and f conflict in resource usage, and must be separated into different MIs.

Figure II.3(b) shows MO a placed as early as possible, in the first MI with b. The MO, g, however, has resource conflicts with not only the first but every level in the range determined in Figure II.3(a). The solution is to split the MI containing d into two. Then when, as in Figure II.3(c), h is added as early as possible, all the MOs are fitted into levels (MIs) which have no resources conflicts and whose contents and order obey the precedence described in the specification.

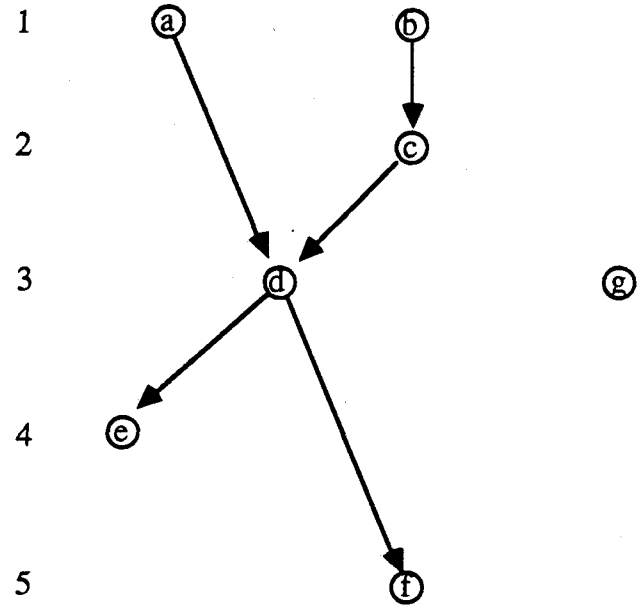


Resource conflicts: {b,g}, {c,g}, {e,f}, {d,g}

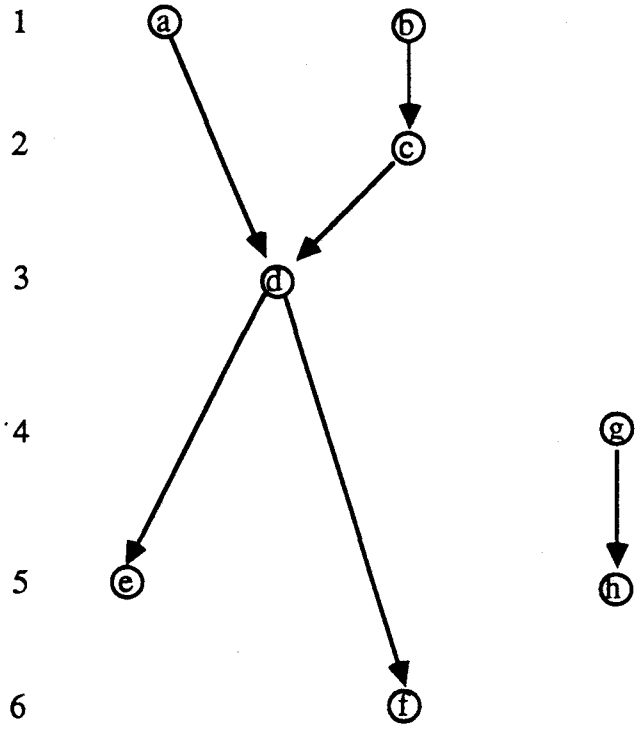
Figure II.2. Example microprogram specification.



a) Early, Late and Critical Paths



b) Add noncritical g to revised CP.



c) Add h.

Figure II.3. Critical path solution

4. Microprogram structure

4.1. Introduction

As indicated in the previous section, construction of a good microprogram requires simultaneously satisfying precedence and resource constraints. This section examines the interaction of those constraints by means of discovering conditions on the structure of a microprogram as a schedule of microoperations. These conditions can be considered necessary conditions on a schedule which is under consideration for use as a microprogram.

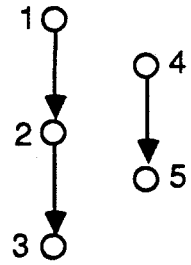
As defined in Section 3, a microprogram or MP is a sequence of microinstructions or MIs. The microinstructions contain disjoint subsets of the instances of microoperation types. A complete microprogram is thus an ordered partition of the microoperation instances or MOs; the microinstructions are the sequentially ordered blocks of the microprogram partition.

Clearly, partitions of the MOs which violate either the dependency or the resource constraints cannot be microprograms.

Definition: A candidate partition is a partition on the set of microoperation instances such that no pair of microoperations in the same block is ordered by the precedence ordering, and no subset of the MOs in a block have a resource conflict.

There is one major difference between the concept of a partition and the concept of a horizontal microprogram: the blocks in a partition do not have order, but the MOs in a horizontal microprogram must have an order of execution. Nevertheless, each horizontal microprogram can be derived from some candidate partition by using the order of the data dependency graph to determine the order of execution of microinstructions (i.e. the blocks in the partition). It is possible that more than one horizontal microprogram can be derived from one candidate partition. However, there are some candidate partitions from which no horizontal microprogram can be derived.

For example, one of the candidate partitions of the set of microoperations: {MO1,MO2,MO3,MO4,MO5}, with its data dependency graph and its resource usage shown in Figure II.4, is {(MO1,MO5),(MO2), (MO3,MO4)}. This partition of the five MOs is a candidate partition because no pair of MOs in a block conflicts by data dependency or resource usage. But no horizontal microprogram can be derived from this partition because, from the data dependency graph (Fig. 1a), MO1 must be executed before MO3, and MO4 must be executed before MO5. If MO1 and MO5 are in the same microinstruction, and MO3 and MO4 are in another microinstruction, neither MI may precede the other. A conflict of this nature is called a "cross-over problem."



Microoperation	Resource Usage
MO1	Ra
MO2	Rb
MO3	Rc
MO4	Ra,Rb
MO5	Rb,Rc

(a) data dependecy graph (b) resource usage table

Figure II.4. Cross-over problem example

4.2. The microprogram partition

For a candidate partition, from each microoperation instance, x, in turn we can describe a sequence of sets of MOs which could follow x in the corresponding microprogram. We define a sequence of levels with regard to the chosen MO. Each level includes the MOs which are separated from the previous or higher level in sequence by the requirements of the precedence relation, and also includes those others which, according to the given partition, are grouped in the same microinstructions.

The first level following microoperation x: $L_1(x)$ is the union of all blocks B_i of the partition, containing a direct successor of x.

$L_1(x) = \bigcup \{ B_i \mid$ a direct successor of x is in $B_i. \}$

Successive levels: For k = 1,2,...

$L_{k+1}(x) = \bigcup \{ L_1(y) \mid y$ is in $L_k(x) \}$.

We can collect the occupants of successive levels into sets including successively higher-order successors:

$$S_1 = L_1$$

$$S_{k+1} = S_k \cup L_{k+1} \text{ for } k = 1, 2, \ldots$$

Successive levels $L_1, L_2, \ldots$ may include MOs already seen in earlier levels, or a level may be empty. In either case, the process is terminated when at some point the next level adds no new microoperations. The resulting set of successors is called $S^*(x)$:

$$S^*(x) = S_k \text{ when } S_{k+1} = S_k.$$

Of course, $S^*(x)$ may not include all the MOs of the MP.

Because of the existence of the cross-over problem, although being a candidate partition is necessary for a horizontal microprogram, it is not sufficient.

Definition: A valid partition is a candidate partition of all the MOs such that for each MO, x, x is not in $S^*(x)$.

Lemma 1: A horizontal microprogram is a valid partition.

Proof: Obviously, a horizontal microprogram is a candidate partition. Take an MO, x, in block B. Each MO which is directly dependent on x is in some block B_i of the horizontal microprogram, following and distinct from B. Thus x is not in $S_1(x) = L_1(x)$. For some level k, assume that x is not in $S_k(x)$. Any successor of a member y of $L_k(x)$ is in a block B_j which must follow the corresponding block of L_k. These blocks B_j form $L_{k+1}(x)$. $L_{k+1}(x)$

and therefore $S_{k+1}(x)$ is composed of blocks which follow, so do not contain x. By induction, the horizontal microprogram is a valid partition.

Lemma 2: From each valid partition, at least one horizontal microprogram, whose microinstructions are the blocks of the valid partition, can be derived.

Proof: Since the valid partition is a candidate partition, microoperations in each block in the partition are parallel executable and have no resource conflict. And, by definition of a partition, each microoperation is a member of exactly one block. These properties satisfy the first two properties of a horizontal microprogram. If for each instance of an MO, x, all blocks in $S^*(x)$ are placed after the block containing x, then the third property of a horizontal microprogram is satisfied. Therefore, at least one horizontal microprogram exists corresponding to each valid partition.

Given a valid partition, one way to schedule the blocks to form a horizontal microprogram starts with all the blocks in one layer. Then, we repeatedly split each layer which has more than one block, always preserving the precedence.

1) Start with all the blocks of the partition in one first layer, L_1. If in the following, all blocks are removed from a layer, then the layer disappears, or all blocks below it move up one layer.
2) For each layer L, starting with $L = L_1$:

2.1) Choose a block, B_i in layer L. Create two new layers, L+1 and L+2, after moving any other later layers down. For each MO, x, in that block:

2.1.1) If any block in $S^*(x)$ is in an earlier layer, move it to L+2.

2.2) Move B_i to layer L+1.

2.3) If B_i was the only block in layer L, set L = L+2. Otherwise, stay at layer L to choose another block.

2.4) Repeat from 2.1) until there are no more blocks at lower layers.

An alternate procedure would be to build a directed acyclic graph of the blocks by searching the MOs of each block to find which block must follow each block B_i; and then to choose an arbitrary schedule which does not violate the explicit block precedence order.

Theorem 1: For any given microprogram, the set of all valid partitions generates the set of all horizontal microprograms.

Proof: From Lemma 1, all horizontal microprograms are valid partitions. Then, the set of all horizontal microprograms is a subset of the set of all valid partitions. And, from Lemma 2, at least one horizontal microprogram can be generated from a valid partition. Therefore, the set of all valid partitions generates the set of all horizontal microprograms.

Since, from Theorem 1, the set of all valid partitions generates the set of all horizontal microprograms, the shortest horizontal microprogram (i.e. the horizontal microprogram that has the smallest number number of microinstructions) is generated by the valid partition which has the smallest number of blocks. Therefore,:

Corollary 1: For a microprogram, the smallest number of microinstructions in any horizontal microprogram is equal to the number of blocks in the maximal valid partition.

4.3. Constrained scheduling of microoperations

The previous section defined necessary conditions for a partition to represent a horizontal microprogram. This section makes one step toward taking full advantage of all the latitude allowed within the constraints. We start by defining which pairs of MOs could rest in the same block of a candidate partition. As mentioned before, microoperations can be put into the same block in the partition (i.e. in the same microinstruction) if none have either data dependency or resource conflict with the rest. For this section we only consider pairwise resource conflicts. Instead of forming all candidate partitions, we first test and reduce the possibilities by further examining the defined relation.

Definition: The scheduling compatible relation, sc, is the relation between microoperations that are parallel executable and

have no resource conflict. Scheduling compatible MOs can be members of the same microinstruction (MI).

If MO i is scheduling compatible with MO j, written i sc j, then i and j may possibly be in the same MI. This relation is reflexive (i sc i), symmetric (if i sc j then j sc i) but not transitive (i sc j and j sc k does not require that i sc k). A reflexive, symmetric, nontransitive relation is known as a compatibility relation.

Corollary 2: A partition such that microoperations in the same block are scheduling compatible, is a candidate partition.

As stated before, due to the existence of the cross-over problem, a candidate partition is not sufficient for a horizontal microprogram. In considering a collection of compatibility classes as precursors to candidate partitions, we know it will be necessary to prohibit crossover problems. However, the definitions in Section 4.2. are expressed in terms of partitions and their blocks. A new set of definitions which are similar, but based on compatibility classes, are needed to express the concept of crossover. They are shown below.

Designate as a compatibility collection a set of compatibility classes C_i under the compatibility relation, sc:

$CC = \{C_1, C_2, ..., C_q\}$.

For microoperation x:

$LC_1(x)$, the first level of successors, is the union of all the

(possibly overlapping) compatibility classes containing a direct successor of x.

$LC_1(x) = \bigcup \{ C_i \mid$ a direct successor of x is in $C_i \}$. For (k = 1, 2, ...), $LC_{k+1}(x)$ is the union of all first levels of members of $LC_k(x)$.

$LC_{k+1}(x) = \bigcup \{ LC_1(y) \mid y \text{ is in } LC_k(x) \}$.

We again collect the successive levels into various orders of successors:

$SC_1(x) = LC_1(x)$

$SC_{k+1}(x) = SC_k(x) \cup LC_{k+1}(x)$ for k = 2, 3, ... until there is no change, when $SC^*(x) = SC_{k+1}(x) = SC_k(x)$.

Definition: CC is a <u>valid</u> compatibility collection for the microprogram if for all x, x is not in $SC^*(x)$.

Definition: A <u>compatible partition</u> on a compatibility collection CC, P(CC), is a partition whose blocks are subsets of the classes of the CC.

We want to compare the components of SC^*, using successors and the compatibility relation of CC, with the corresponding components of S^*, using successors and the equivalence relation of P(CC).

Lemma 3: For any compatible partition P(CC) of a valid compatibility collection CC, for any MO, x:

$L_k(x)$ of P(CC) is a subset of $LC_k(x)$ of CC.

Proof: Each block B_i of P(CC) is a subset (possibly empty) of a compatibility class C_i, for each compatibility class, i = 1,2,...,q. For any MO, x, a direct successor of x is in Bi only if it is in the superset, C_i:

$$L_1(x) = \bigcup \{ B_i \mid \text{a direct successor of } x \text{ is in } B_i \} \cup \{ C_i \mid \text{a direct successor of } x \text{ is in } C_i \}$$

$$L_1(x) \subseteq LC_1(x).$$

If $y \in B_i$ in some $L_k(x)$, then $y \in C_j$, $B_i \subseteq C_j$. Thus, $L_1(y) \subseteq LC_1(y)$ so that $L_k(x) \subseteq LC_k(x)$, for all k.

Since each level $L_k(x)$ is restricted, the collections are.

Lemma 4: For any compatible partition P(CC) of a valid compatibility collection CC, for any MO, x:

$S_k(x)$ of P(CC) is a subset of $SC_k(x)$ of CC.

Finally, we see that a partition formed from the compatibilities can be crossover-free:

Theorem 2: A compatible partition P(CC) of a valid compatibility collection, CC, is a valid partition.

Proof: Since the blocks of P(CC) are subsets of compatibility classes C_i, P(CC) is a candidate partition. For some microoperation, x, since $S_k(x) \subseteq SC_k(x)$ for each k, therefore if x is not in $SC^*(x)$, then x is not in $S^*(x)$.

5. Composite model

5.1. Basic approach

The semantic data dependencies among microoperation instances and the resource conflicts among microoperations were introduced in Section 3. The manner in which these constraints interact was discussed. Section 4 described conditions on a partition of microoperation instances, or on the compatibility classes which can be used to generate the partition, which must obtain if an ordering of the partition is to serve as a horizontal microprogram.

These necessary conditions were presented from the passive point of view. Given a partition representing a microprogram, we are sure they obtain; given the constraints, a collection of selected compatibility classes on the MOs must be crossover-free if they are to be taken as valid for forming a horizontal microprogram. The current section takes an active point of view. Definitions based on a graph model are presented, and it is shown how a satisfactory microprogram could be formed while satisfying all of the constraints.

5.2. A composite graph model

A graphical method is used to determine the valid compatibility collections within a given microprogram.

A graph is a set of vertices or points or nodes plus a set of edges connecting pairs of vertices. In our model, the vertices will represent microoperation instances, and they will be labelled as such. The edges will represent relations. As we are concerned with two relations, the partial ordering of dependency and the scheduling compatibility relation, we will use labelled edges of two types.

G=(V;E) is a graph on a set of vertices, V and a set of edges, E, where $E = \{(i,j,t)\} \subseteq V \times V \times T$, for a set of edge label types T.

Definition: A graph G'=(V';E') is a subgraph of G=(V;E) if V' and E' are, respectively, subsets of V and E such that an edge (v_i, v_j, t) of E is in E' only if v_i and v_j are in V' [SWAM81].

Definition: A graph G is a supergraph of a graph G' if G' is a subgraph of G.

Definition: A graph G'=(V';E') is a spanning subgraph of G=(V;E) if G' is a subgraph of G and V'=V [SWAM81].

A graph can represent the potential parallel structure for a microprogram. For microprogram p, let the vertices V_p represent the microoperation instances. The edges will be of two disjoint sets of different type, $E_{p,d}$ and $E_{p,sc}$. The direct data dependent pairs (MOs i and j) are connected by edges of type "d", where (i,j,d) is an element of $E_{p,d}$ iff j directly depends on i. $E_{p,sc}$ edges will connect the scheduling compatible pairs of MOs: (k,l,sc) is in $E_{p,sc}$ iff k and l are scheduling compatible. Of course, sc is symmetric so (l,k,sc) is also in $E_{p,sc}$, but sc is not transitive. $E_{p,d}$ is transitive and antisymmetric.

Definition: With the notation above, define:

$$E_p = E_{p,d} \cup E_{p,sc}.$$

Now, for microprogram p, $M_p = (V_p; E_p)$ is the extended graph of the microprogram.

Definition: An alpha loop is a connected graph with more than one vertex which has a spanning cycle which does not contain two edges of type sc in succession.

Definition: A beta subgraph, m, of microprogram p,

$$B_{p,m} = (VB_{p,m} ; EB_{p,m})$$

is a spanning subgraph of M_p such that:

1) $EB_{p,m}$ includes all data dependencies and some scheduling compatibilities:

$$E_{p,d} \subseteq EB_{p,m} \subseteq E_{p,d} \cup E_{p,sc}.$$

2) No subgraph of $B_{p,m}$ is an alpha loop. and

3) If (k,l,sc) is in $B_{p,m}$ then so is (l,k,sc).

Definition: A delta collection, $DC_{p,m}$, of the beta subgraph, $B_{p,m}$, is a collection of subsets of microoperations in microprogram p such that i and j are in the same set in the $DC_{p,m}$ only if (i,j,sc) is in $EB_{p,m}$.

Theorem 3: A $DC_{p,m}$ is a valid compatibility collection.

Proof: We define MOs i and j compatible if i and j are in the same $DC_{p,m}$ subset; this is a reflexive and symmetric relation. Because no alpha loop exists in the generating beta subgraph, $B_{p,m}$, then no MO, x, is found in any of its successor levels: x is not in $SC^*(x)$.

5.3. Microprogram selection

In order to form a horizontal microprogram, a partition on the set of all microoperations can be constructed in such a way that each block in the partition is a subset of a set in the $DC_{p,m}$. This problem is similar to a covering problem.

Given a collection U of subsets u_i of a set of elements S = {a,b,c,...x}, a cover of S is a subset of U, K = $\{u_j\}$, such that each element of S is contained in one of the classes u_j. The covering problem is to find, for U and S, a minimal-cost cover, K. If the cost

function is additive, the cost C(K) is the sum of the costs of the selected classes, $C(K) = \sum_{\forall u_j \in K} C(u_j)$. If all classes have equal cost, the minimal cover objective is the smallest number of classes which together encompass, or cover, the set S.

For a horizontal microprogram, S is the set of all microoperation instances in the given microprogram P, and K is a delta collection, $DC_{P,m}$, of P. If minimization of microprogram length is the goal, the cost function of a covering collection, C(K), is defined as the number of classes selected:

$$C(K) = |K|$$

There is one major difference between a covering problem and a horizontal microprogram forming problem: while in a covering problem the selected sets are not necessarily disjoint, in a horizontal microprogram the microinstructions must not "share" microoperations. However, if the selected classes "share" some microoperations, it is easy to generate a horizontal microprogram which has a number of microinstructions equal to the number of classes in the covering collection by "moving" the intersected microoperations in or out of the microinstructions. For example, the covering collection: { {1,2,3}, {3,4,5} }, will generate two horizontal microprograms of the same length. They are {(1,2,3), (4,5)} and {(1,2), (3,4,5)}. Therefore, methods developed for simplifying covering problems may be used for the construction of the horizontal

microprogram.

In a covering chart [KOHA78] used for constructing a horizontal microprogram, the columns correspond to microoperations in the given microprogram and the rows correspond to classes in the delta collection. For example, the chart for the delta collection: {(1,2,5);(2,3);(3,6,7);(4,5,7)} is shown in Figure II.5:

	1	2	3	4	5	6	7
(1,2,5)	x	x			x		
(2,3)		x	x				
(3,6,7)			x			x	x
(4,5,7)				x	x		x

Figure II.5. Covering problem for a microprogram.

In this example, the row (1,2,5) is called essential, because no other row include MO 1. Thus, (1,2,5) is in each cover. Further, for this example, a minimal-size cover consists of just the three essential rows:

(1,2,5), (3,6,7), (4,5,7)

From this cover, four valid partitions can be generated:

{(1,2,5);(3,6);(4,7)},

{(1,2,5);(3,6,7);(4)},

{(1,2);(3,6);(4,5,7)}, and

{(1,2);(3,6,7);(4,5)}.

From Lemma 2 of Section 4.2, we know that each valid partition can be ordered in such a way that it becomes a horizontal microprogram.

5.4. Microprogram composition summary

The results of this section suggest the following plan, given the data dependencies and resource conflicts specifying an example microprogram, to select a compact horizontal microprogram. The general idea is to reduce the scheduling compatible relation, sc, to an equivalence relation. The blocks of the resulting partition, when ordered so as to agree with the data dependencies, form a microprogram.

1. Form the graph with edges labelled either d (data dependent, antisymmetric) or sc, (scheduling compatible, symmetric).
2. Find B, a maximal spanning subgraph without alpha loops. Form maximal compatible classes of the diminished sc relation.
3. Choose a covering collection of MCCs.
   - 3.1. The MOs in an MI must be a subset of some MCC.
   - 3.2. The MIs of a MP are blocks of a partition of the MO instances. The MP is thus a sequence of sets, each of which may be formed from one of a covering set of MCCs by eliminating overlap.
   - 3.3. A compact MP is formed from a minimal cover of the MO instances by the MCCs.

5.4.1. Example

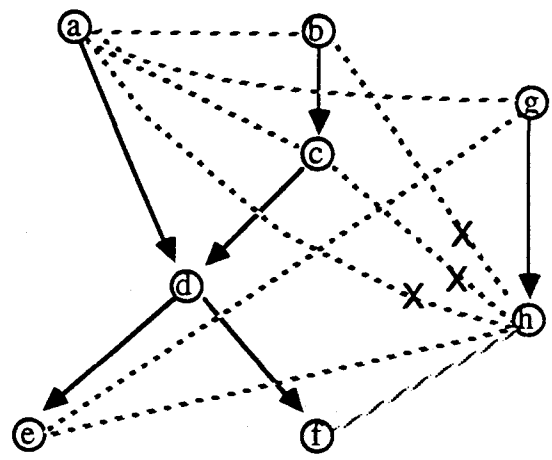
We return to the example microprogram shown in Section 2.2 and 3.3. Figure II.6(a) below shows the composite graph. The data dependencies are shown on the diagram with solid lines as edges of the two unconnected parts of the directed graph which together represents the partial order, d. Some MO pairs are related by this partial order, and others conflict in their usage of system resources, as specified in section 3.3. The others are scheduling compatible (sc), and are shown in Figure II.6(a) as dashed lines.

Several alpha loops appear in that graph: for example, a,d, e,g,h,a. One way to remove them is to cut the edges between e and g; this could lead to the same microprogram previously shown in Section 3.3. An alternative solution is to form the diminished sc relation by removing all edges between (a,h), (b,h), (c,h) and (d,h).

The maximal delta collection uses all the sc links remaining in Figure II.6(a) to form the compatibility classes used in the covering problem of Figure II.6(b), and listed along the left side of that table. There, the compatibility classes {a,b}, {a,c}, {d}, and {f,h} are essential (some subsets of them must be in every solution), and of the remainder, {e,g} is a dominant choice because it alone covers the rest of the MOs.

Finally, the minimal cover of MOs, the delta collection {(a,b), (a,c), (d), (e,g), (f,h)} must be converted to a partition, and the blocks ordered in agreement with the data dependency partial order. One such solution is shown in Figure II.6(c). There, sc links (a,b), (e,g), and (f,h) tie MOs which are scheduled in the same MIs. Links (a,c), (a,g), and (e,h) do not bind in this manner.

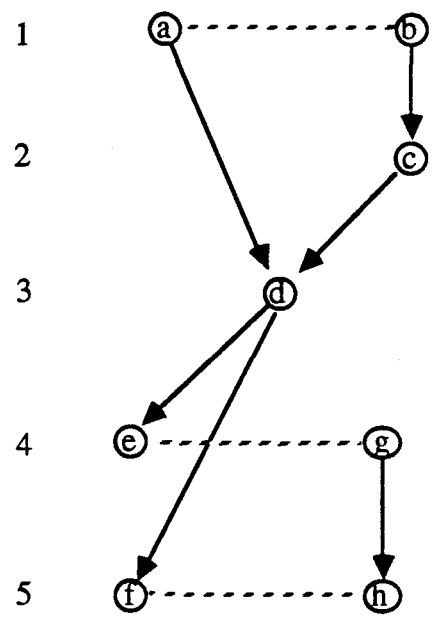
Notice that drawing the composite graph with both dependencies and scheduling compatibilities, and reducing it by eliminating the alpha loops which demonstrate crossover problems, allowed us to find the microprogram depicted in Figure II.6(c), which has only five microinstruction, rather than the six generated by applying the classical critical path method of considering the precedence and the resource constraints in separate phases, as described in Section 2.2 and 3.3.



(a) Composite graph

	a	b	c	d	e	f	g	h
ab	x	x						
ac	x		x					
ag	x						x	
d				x				
eg				x			x	
eh				x				x
fh						x		x

b) Covering matrix for MI selection



c) Microprogram

Figure II.6. Microprogram Composition Example

5.4.2. A non-deterministic approach

A composite graph model of local compaction indicates that a horizontal microprogram which is transformed from a vertical microprogram is a partition of microoperations such that each block of the partition contains only microoperations that are in the same compatible class of a composite graph with all the alpha loops eliminated. Therefore, an obvious procedure to compact a basic block is as described in section 5.4 and demonstrated in the example in section 5.4.1. The problem of identifying alpha loops and the problem of minimum cost covering are *NP*-complete problems [HOPC79]. However, since the number of microoperations in a basic block is typically small (a basic block that has more than 10 microoperations is uncommon [WOOD78]), the procedure can be used for compaction.

Since the problem of finding the optimal local compaction is an *NP*-hard problem [VEGD82], any procedures to find an optimal valid partition are bound to have non polynomial complexity. One of these procedures is presented in non-deterministic format as following.

1) Initialize all nodes as unmarked. Initialize the U-set as an empty set; the U-set is used to record all nodes (microoperations) that are already placed in the current microinstruction.
2) Find an unmarked node which has no unmarked predecessor, N_1, then mark it.

3) Find an sc edge that connects N_1 to an unmarked node, N_2, which also has no unmarked predecessors.

4) If there exists any node in the U-set which has no sc edge connecting it to N_2, eliminate the sc edge that connects N_1 and N_2. Go to step 6).

5) If N_2 has sc edges connecting it to all nodes in the U-set, mark it and add it to the U-set.

6) Eliminate all the sc edges that connect N_1 to any successors of N_2 by the data-dependency edges (d edges).

7) Repeat step 3 to 6 until all the sc edges of N_1 connect to only marked nodes.

8) Clear the U-set.

9) Repeat step 2 to 8 until all nodes are marked.

10) Find the delta collection of this graph (at this point, the graph has no alpha loop). Each block of the delta collection is a microinstruction. And, the delta collection is a valid partition (i.e. a horizontal microprogram).

From this non-deterministic procedure, a search tree can be constructed. The optimal solution can be found by performing exhaustive tree search. This procedure is the same as FromCI procedure described in [LAND80]. The time spending for the tree searching can be reduced by using some heuristic schemes to determine search paths. List scheduling uses a heuristic value, called

priority value, calculated for each node to guide the search. The time used by the procedure described above can be reduced by using these priority values for selecting N_1 and N_2 (i.e. select the node that has the highest priority value first). This procedure cannot guarantee an optimal solution but neither can list scheduling.

6. Summary

Examining the structural constraints among the microoperations of a basic block yields an understanding which can be useful in attempts at both local compaction and global compaction. This paper has described how the two classes of constraints on horizontal microcode of a basic block can be examined, along with their interactions, in order to limit the search for a compact microprogram. Of course, one part of global compaction is to use basic blocks which themselves are compact. But a potentially more powerful way to use these results is to examine the interactions of the basic block constraints with the dynamics of control flow among blocks, and allow tradeoffs in moving microoperations among blocks. Such considerations will be left for another paper.

In the present work, two constraints on the microoperations of a basic block are taken as the starting point. The precedence relation derivable from data dependence among MOs is a partial order which must be preserved in order to preserve the semantics of the microprogram's calculation. The resource conflicts prohibit certain

MOs from acting simultaneously, because of the lack of enough copies of some resources (clearly, one way to design a data path for a given microprogram would be to trade off additional data-handling resources for shorter microprograms).

Since a given horizontal microprogram is a sequenced partition of the set of MO instances, it seems worthwhile to consider the conditions required of a partition because of the two kinds of constraints. Section Four describes conditions on a partition, necessary that it describe a microprogram. Such a "valid" partition satisfies precedence and resource constraints, and their interactions. Because compaction requires discovery of such partitions, not merely their evaluation, the necessary conditions are also formulated there for the more naturally discoverable compatible collections of MOs which may be scheduled together in the same microinstruction.

In Section Five, a graphical method is used to determine the valid compatibility collections within a given microprogram. A composite graph shows both precedence and resource compatibility. Eliminating a certain class of cycles in the graph allows selection of the compatibility collections. An example shows how this composition model can be used to compose an optimum compaction of an example microprogram from the literature. Optimum compaction still involves a search among alternatives, and if the search grows too tedious it can be shortened by quickly choosing a (possibly suboptimal) microprogram, either by heuristic methods similar in effect to the ones previously published elsewhere, or by

algorithms yet to be devised.

The descriptive success of the model in the present work suggests that search methods based upon exploiting this description may be a profitable future area of research. A problem which seems interesting is to attack the interaction of two previously separated microprogram reduction problems: the reduction of the number of microinstructions, as treated in the present paper; and the reduction of the length in bits of a microinstruction.

7. References

[ASTO71] F. Astopas and K. Plukas;
"Method of Minimizing Computer Microprograms", *Automatic Control*, 5,4, 1971, pp.10-16.

[ATKI84] R. P. Atkins,
"Improved Instruction Formation in the Exhaustive Local Microcode Compaction Algorithm", *Proceedings of the 17th Annual Microprogramming Workshop*,
October 30—November 2, 1984 pp. 105-111.

[DASG76] S. Dasgupta and J. Tartar;
"The Identification of Maximal Parallelism in Straight-line Microprograms",
IEEE Transactions on Computers, C-25,10 October 1976, pp.986-991.

[DASG78] S. Dasgupta,
"Comment on the Identification of Maximal Parallelism in Straight-line Microprograms", *IEEE Transactions on Computers*, C-27,3; March 1978, pp.285-286.

[DAVI81] S. Davison, D. Landskov, B. Shriver,
and P. Mallet;
"Some Experiments in Local Microcode Compaction for Horizontal Machines",
IEEE Transactions on Computers, C-30,7; July 1981, pp.460-477.

[FISH79] J. A. Fisher,
"The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling with Resources", PhD. Dissertation, the Department of Computer Science, New York University, New York, NY, October 1979.

[HENR83] S. D. Henry, R. A. Mueller, and M. Andrews;
"Local and Global Microcode Compaction Using Reduction Operators",
Proceedings of the 16th Annual Microprogramming Workshop,
October 11-14, 1983, pp. 34—43. October, 1983, pp. 34 - 43.

[HOPC79] J. E. Hopcroft and J. D. Ullman,
"Introduction to Automata Theory, Language, and Computation", Addison-Wesley Publishing Company, Reading, Massachusetts, 1979.

[KOHA78] Z. Kohavi,
"Switching and Finite Automata Theory",
McGraw-hill Book Company,
Second Edition,1978.

[LAND80] D. Landskov, S. Davidson, B. Shriver,
and P. W. Mallett;
"Local Microcode Compaction Techniques",
Computing Surveys, Vol.12, No. 3, September 1980, p 248.

[RAMA74] C. V. Ramamoorthy, and M. Tsuchiya;
"A High-Level Language for Horizontal Microprogramming",
IEEE Transactions on Computers, C-23,8; August 1974, pp. 791-801.

[SWAM81] M. N. S. Swamy, and K. Thulasiraman;
"Graphs, Networks, and Algorithms",
Wiley Interscience, 1981.

[TOKO81] M. Tokoro, E. Tamura, and T. Takizuka;
"Optimization of Microprograms",
IEEE Transaction on Computers, C-30,7; July 1981, pp. 491-50.

[TSUC76] M. Tsuchiya, and M. J. Gonzalez;
"Toward Optimization of Horizontal Microprogram",
IEEE Transactions on Computers, C-25,10, October 1976, pp. 992-999.

[VEGD82] S. R. Vegdahl,
"Local Code Generation and Compaction in Optimizing Microcode Compilers", PhD Dissertation, the Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA., December 1982.

[WOOD78] G. Wood,
"On the Packing of Micro-operations into Microinstruction words",
Proceedings of the Eleventh Annual Microprogramming Workshop, November 19-22, 1978; pp. 51-55.

[YAU74] S. S. Yau, A. C. Schowe, and M. Tsuchiya;
"On Storage Optimization of Horizontal Microprogramming",
Proceedings of the 7th Annual Microprogramming Workshop, September 30—October 2, 1974, pp 98-106.

Chapter III
The KLITS Microprogram Global Compaction Algorithm

Abstract

Global compaction has been recognized as a major issue in microprogram optimization. A microprogram can be compacted globally by moving microoperations, sometimes past basic block boundaries. However, "to find a sequence of very profitable moves, one often has to go through an initial sequence of moves which are either not profitable, or, worse still, actually make the code longer. Locating such a sequence involves abandoning attempts to prune this expensive search tree" [FISH81].

Fisher's trace scheduling [FISH79], [FISH81] offers an alternative by compacting traces without recognition of basic block boundaries. The topological structure of the microprogram is altered later to preserve its semantics. However, trace scheduling has been criticized for the growth, in the worst case exponential, of memory size due to intensive copying of blocks, and the difficulty of its implementation. Also, since trace scheduling changes the topological structure of blocks, the compacted microprogram may be difficult to

debug.

A technique called *KLITS*, which is based on the concept of microoperation movement and a heuristic iteration similar to that described in the Kernighan-Lin Partitioning Algorithm [KERN70], [ULLM84] is developed and tested. This technique can accommodate any of several measures of optimization (e.g. the average execution time, the total length of the microprogram), and allows the programmer to control the growth of the memory.

KLITS searches for globally profitable sequences of microoperation movements even when the first few movements of some sequences do not yield immediate profit. The compacted microprogram retains the block structure and control flow of the original.

1. Introduction

Global compaction has been recognized as a major issue in microprogram optimization. One of the most noticeable techniques is Fisher's trace scheduling [FISH79], [FISH81]. However, Fisher's trace scheduling has been criticized for the growth, in the worst case exponential, of memory size due to intensive copying of blocks. Also, trace scheduling usually changes the topological structure of blocks, causing the compacted microprogram to be difficult to debug.

A microprogram can be compacted globally by moving micro-operations, sometimes past basic block boundaries. The topological structure of blocks can be easily controlled and preserved. The growth of memory size can be monitored by the programmer.

> However, "...to find a sequence of very profitable moves, one often has to go through an initial sequence of moves which are either not profitable, or, worse still, actually make the code longer. Locating such a sequence involves abandoning attempts to prune this expensive search tree" [FISH81].

The Kernighan-Lin Partitioning Algorithm, which embodies an optimization strategy that may take "bad" steps (i.e. produce immediate losses) if they will lead to an overall gain later, is described in [ULLM84]. "This algorithm takes time that, experimentally, has been found to grow far more slowly than an exponential" [ULLM84].

A global compaction technique with a similar strategy and the concept of microoperation movement as its kernel, is presented. This technique can accommodate any of several measures of optimization (e.g. the average execution time, the total length of the microprogram). With this technique, the programmer has the ability to control the growth of the memory and the changing of topological structure. An heuristic evaluation function is implemented and used to demonstrated the effectiveness of the technique for reducing the total microprogram length.

2. Basic concepts

A microoperation is free at the top of a basic block if that microoperation is not data dependent on any other microoperation of the same basic block.

A microoperation is free at the bottom of a basic block if there is no microoperation in the same basic block which is data dependent on it.

A register is live at the top of a basic block if that register is read before it is written to in some trace that starts with the basic block.

The side-set of a basic block is the union of all its successors' live-at-the-top register sets if the basic block terminates with a

microinstruction containing a conditional-jump microoperation, and is an empty set otherwise.

3. Previous work

3.1. The symmetric pairs method [DASG79]

Dasgupta [DASG79] defined a symmetric pair $<B_i,B_j>$ as a pair of basic blocks such that basic block B_i is executed whenever B_j is, and vice versa. A microoperation can be moved up from a basic block to its symmetric pair partner if the microoperation is dependent neither on any microoperations in the same basic block nor on any microoperations in any basic block separating the symmetric pair. Dasgupta's algorithm moves a microoperation up if it can be placed in the upper basic block of the symmetric pair without increasing the basic block length.

3.2. MORIF compaction [TOKO78], [TOKO81]

Tokolo, Takizuka, and Yamaura described a technique for microprogram compaction utilized a microoperation modeling technique called microtemplating; later, Tokolo, Tamura, and Takizuka revised the technique and called it MORIF (microprogram optimization technique consider resource occupancy and instruction formats). "A microtemplate is a two-dimensional representation (timing and resources) of a microoperation. Timing is expressed on

both machine cycle and subcycle base" [TOKO81]. MORIF includes both local compaction and global compaction. A microprogram is partitioned into basic blocks. Each basic block is compacted separately (i.e. local compaction). Then, microtemplates are transferred between contiguous segments (basic blocks) which are any two basic blocks that connect directly by an edge in the control flow structure. Toko et al. [TOKO78], [TOKO81] provides conditions for transferring microtemplates between contiguous basic blocks; these conditions are similar to Fisher's moving rules [FISH81]. A microtemplate is transferred if the transferring improves the overall microprogram.

3.3. Trace scheduling [FISH79], [FISH81]

In his doctoral dissertation [FISH79], Fisher introduced two microprogram compaction techniques: list scheduling for local microprogram compaction and trace scheduling for global microprogram compaction. Both have become well known and widely studied.

Although list scheduling is a local compaction technique, it plays an important role in global compaction. Several global compaction techniques include list scheduling or its variations in them [ISOD83], [LAH83], [SU84], [RAO86]. In list scheduling, each microoperation in the basic block is assigned a priority. Then, a set is constructed of microoperations which have no predecessors except those that were already placed in microinstructions; this set is called

the data ready set or D—set. The D-set is initialized with all micro-operations that do not have any predecessor. A microinstruction is formed by grouping microoperations from the D—set according to their priority. The microoperation with the highest priority is taken out from the D-set and placed first in the empty microinstruction. The microoperations remaining in the D—set are examined in order of their priority. The microoperations are taken out from the D—set and added into the newly formed microinstruction if they are not in conflict with those already in there. If such a conflict does exist, the microoperation is not selected. The process is repeated until either no more microoperations can be added into the forming microinstruction or the D—set is exhausted. After the new microinstruction is completed, the D-set is updated. The compaction is terminated when all microoperations in the basic block are placed in microinstructions.

In his dissertation, Fisher categorized 3 conditions of interblock microoperation motions, and later [FISH81] he revised them into 6 rules, as shown in Figure III.1, which have been called moving rules. These moving rules illustrate both conditions that need to be satisfied before the interblock movement can begin, and the micro-operation position in the microprogram after the movement is finished. By applying these moving rules, Fisher pointed out that a global compaction, which he named "The Menu Method" [FISH81], can be performed on microprograms. The menu method first locally compacts each basic block separately. Then, by using the moving rules, microoperations are moved past basic block boundaries if the

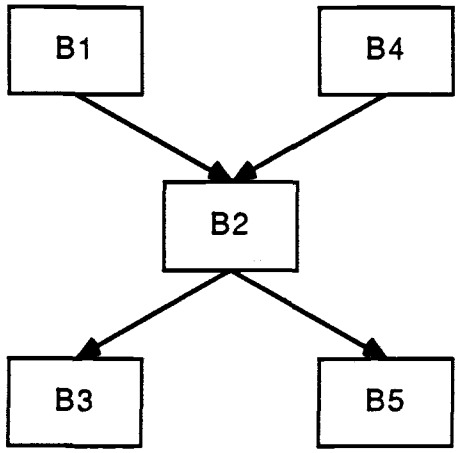


Figure III.1 (a) The basic blocks structure for Fisher's moving rules

Rule Number	MO can move From	To	Under the Conditions that
R1	B2	B1 and B4	The MO is free at the top of B2.
R2	B1 and B4	B2	Identical copies of the MO are free at the bottoms of both B1 and B4.
R3	B2	B3 and B5	The MO is free at the bottom of B2.
R4	B3 and B5	B2	Identical copies of the MO are free free at the tops of both B3 and B5.
R5	B2	B3 (or B5)	The MO is free at the bottom of B2 and all registers written by the MO are dead in B5 (or B3).
R6	B3 (or B5)	B2	The MO is free at the top of B3 (or B5) and all registers written by the MO are dead in B5 (or B3).

Figure III.1 (b). Fisher's moving rules

movement "appears to save a cycle".

Fisher introduced trace scheduling which considers a "more *global view* of all the code [microprogram] being optimized [compacted]" [FISH79]. In trace scheduling, instead of compacting basic blocks, traces are scheduled.

"A trace is a loop-free sequence of microoperations which might be executed contiguously for some choice of data" [FISH81]. Traces are scheduled in the order of their probability of execution. The trace with the highest probability of execution will be the first to be scheduled. After a trace is determined, a data precedence graph is built for the trace. The data precedence graph is similar to the data dependency graph which has been used in most local compaction techniques (i.e. a path from node i to node j represents the data dependency of MO_j on MO_i). However, in order to prevent absolutely illegal microoperation movements between basic blocks, movements that would destroy the values of registers that will be referenced by microoperations off the traces, new, special edges are added to the data precedence graph. If a conditional jump microoperation C is in the trace but may jump to a basic block B off the trace that has a register R in its live-at-the-top registers set, the new edges must be drawn from C to every microoperation in the trace that writes to R. A register is live at the top of a basic block if that register is read before it is written to in the trace that starts with the basic block. The data precedence graph constructed for the trace contains all the necessary restrictions on microoperation movements among basic

blocks in the trace. Without any knowledge of original basic block boundary locations, the trace is scheduled (i.e. compacted) as if it were a single basic block using the list scheduling local compaction technique.

During trace scheduling, microoperations may be moved past the original basic block boundaries. Microoperations in the trace that the basic blocks off the trace jump to, or the old trace entrances, and the microoperations that jump to the basic blocks off the trace, or the old trace exits, can be moved. Therefore, the new trace entrances and the new trace exits need to be identified. And, to preserve the semantics of the original microprogram, after the trace is compacted, copies of microoperations are inserted off the trace, above the new trace entrances and/or below the new trace exits. The microoperations that need to be copied, with few exceptions, are the microoperations that have been moved above the new trace entrances and/or below the new trace exits. This phase is called the bookkeeping phase of the trace scheduling.

After the bookkeeping phase is finished, another uncompacted trace is determined, and the process is repeated until no microoperation is left uncompacted.

Loops are handled by unrolling them for many iterations. "The unrolled loop is a stream, all intermediate loop tests are now conditional jumps, and the stream gets compacted as above" [FISH83].

Trace scheduling has proven to be able to produce significant reduction in execution time of compacted microprograms. Trace scheduling has become the most well known and widely studied method of global compaction. Although it has been regarded as "the most promising global compaction", [RAO86] it also has been criticized for the growth of memory size by extensive copying of basic blocks [FISH81], [LAH83], [SU84], [MUEL86], [RAO86]. And, since trace scheduling performs compaction on basic blocks regardless of their boundaries, the topological structure of the micro-program compacted by the technique is, usually, different from the original one. This will make the debugging process more difficult [LAH83], [RAO86].

There are various proposals for mitigating the copying problem and other drawbacks of trace scheduling.

3.4. Tree compaction [LAH83]

The tree compaction method was proposed by Lah and Atkins to reduce the memory size and the complexity required by the book-keeping phase of trace scheduling. With tree compaction, the topological structure of the control flow of the compacted micropro-gram is preserved.

The basic blocks of a microprogram are partitioned into tree-shaped subsets of basic blocks: top trees and bottom trees. "A top

tree (bottom tree) is a connected subgraph which is itself a tree and has a unique node r with in-degree zero [entry] (out-degree zero [exit]) in the tree. Node r is the root of this top tree (bottom tree)" [LAH83]. First, only top trees are considered. Trace scheduling is applied to each top tree separately. Then, the bottom trees are identified and trace scheduling is performed on each bottom tree separately. The microoperations are not allowed to move past either top tree or bottom tree roots. With this restriction, the number of basic blocks copied during the bookkeeping phases can be reduced, and the algorithm is less complex than trace scheduling. However, since "less" movements than trace scheduling are allowed, the tree compaction may not compact the major trace as well as trace scheduling.

3.5. SRDAG compaction [LINN83]

SRDAG compaction is an extension to trace scheduling [LINN83]. Linn indicates that, in trace scheduling, "interactions with basic blocks off the trace are not considered." And, when microoperations are moved from later basic blocks of the trace into the first basic block of the trace, the most likely trace of the microprogram may change. Thus, he suggests that the trace "should be reselected after the first basic block has been compacted". In the SRDAG compaction, a single rooted directed acyclic graph (SRDAG) of basic blocks is considered instead of a trace. The SRDAG to be compacted next is determined by, first, using the microprogram branch probabilities to

select an uncompacted basic block which has no uncompacted predecessors. The basic block becomes the root of the SRDAG (i.e. the next basic block to be compacted). Then, the maximal SRDAG of that basic block such that all basic blocks in it are uncompacted, is selected. The compaction is processed in an iterative manner. However, proof of termination is not presented, since the compaction of a basic block may, unfortunately, require other basic blocks to be created so that the microprogram semantics are preserved.

Linn also demonstrated rearranging for (conditional and unconditional) jump microoperations to preserve microprogram semantics when jump microoperations are moved beyond basic block boundaries.

3.6. Improved trace scheduling compaction (ITSC) [SU84]

The ITSC method is presented by Su, Ding and Jin as an improvement of trace scheduling to reduce the extra microprogram size sometimes generated during the bookkeeping phase and to increase the efficiency of microprogram loop compaction. The improvement can be divided into 3 parts:

1) Modification of the moving rules suggested by Fisher [FISH79], [FISH81].
2) The improved trace scheduling algorithm (ITS) for reducing extra copies.
3) The URCR (unrolling, compacting, and rerolling the

loop) algorithm for loop compaction.

Fisher's moving rules are modified to correct a semantic error which may occur when a conditional jump microoperation is moved down. Beside, they also added two more rules (R7 and R8) to symmetrize previous rules (R5 and R6). With these additions, they increase the chance of moving a microoperation beyond basic block boundaries. The modified moving rules, for the basic blocks structure in Figure III.1(a), are shown in Figure III.2(b).

The ITS (improved trace scheduling) algorithm consists of two major parts. The first part is to partition the microoperations in the trace into two sets: MAIN and the remainder. MAIN is the set of microoperations which are in the critical path of the trace and their non-jump successors (i.e. any non-jump microoperations whose predecessors are also in MAIN). The microoperations in the set MAIN are compacted by using the list scheduling algorithm. Those microoperations in MAIN can be scheduled as early as possible, by the list scheduling algorithm, without extra copies. The rest of the microoperations in the trace are gathered into the remainder set. For each uncompacted microoperation in the remainder set, find its DAG_RANGE which is the schedule range in the already compacted microinstructions (MAIN), regardless of resource conflict, but restricted by the data dependency relationship.

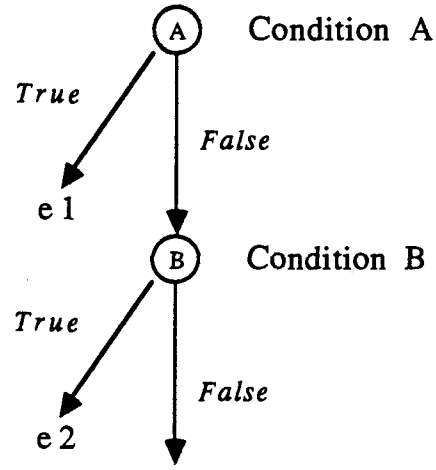


Figure III.2 (a). Su's comditional jumps example

Rule Number	MO can move From	To	Under the Conditions that
R1	B2	B1 and B4	The MO is free at the top of B2.
R2	B1 and B4	B2	Identical copies of the MO are free at the bottoms of both B1 and B4.
R3	B2	B3 and B5	The MO is free at the bottom of B2.
R4	B3 and B5	B2	Identical copies of the MO are free at the tops of both B3 and B5.

R5	B2	B3 (or B5)	For unbranch MO: The MO is free at the bottom of B2 and all registers written by the MO are dead in B5 (or B3). Additional conditions for branch MO: (See Figure III.2a.) Also, its exit is the same as the exit of the branch MO B at the bottom of B2 (e_1 and e_2 are the same) or Condition(MO A) *Condition(MO B) = *False*
R6	B3 (or B5)	B2	The MO is free at the top of B3 (or B5) and the rest of conditions are same as R5.
R7	B2	B1 (or B4)	The unbranch MO is free at the top of B2 and all registers written by the MO are dead below this MO.
R8	B1 (or B4)	B2	The unbranch MO is free at the bottom of B1 (or B4) and all registers written by the MO are dead the top of B2.

Figure III.2 (b). Su's moving rules
(for the same basic blocks structure as in Figure III.1a.)

The uncompacted microoperation which has minimum DAG_RANGE is picked; call it MO_I. For MO_I, find its OPT_RANGE which is the schedule range where MO_I can be scheduled, regardless of resource-conflict, without causing any copy depending on the characteristic and type (jump or rejoin) of the MO_I. The MO_I is scheduled if possible within RANGE, the union of DAG_RANGE and OPT_RANGE. If the MO_I is a jump microoperation, the scheduling attempt starts from the bottom to the top of RANGE. If MO_I is not a jump microoperation, the scheduling starts from the top to the bottom of RANGE. If the MO_I cannot be scheduled within RANGE without causing resource conflict, it is tried again with the rest of DAG_RANGE. If this also fails, then MO_I needs to be inserted in a new microinstruction. The next each MO_I is picked from the remainder set until all microoperations in the trace are compacted. A new trace is then picked and compacted until all microoperations in the microprogram are compacted.

The URCR is used for loop compaction. The loop body A is simply unrolled into two bodies[1] A and A'. The unrolled body is compacted with list scheduling; then the new loop body is found. After all the redundant microoperations are deleted, the loop is rerolled.

---

1. Fisher [FISH83] suggests that a loop body may be unrolled into k bodies for compaction. Su et al. believe that there is no advantage for k>2 over k=2.

3.7. The generalized data dependency graph (GDDG) method [ISOD83]

Isoda, Kobayashi, and Ishida describe a global compaction algorithm which depends on a generalized data dependency graph (GDDG). The GDDG can "integratedly express the concurrency of microorders[microoperations] and their mobility past boundaries of basic blocks" [ISOD83]. The GDDG is a combination of the data dependency and two primitives: Y-joint, which represents multiple entrances to the same basic block, and Y^{-1}-joint, which represents multiple exits from the same basic block (i.e. the basic block terminated with conditional jump microoperation). At first, the edges of the GDDG do not represent data dependency, but the order of execution of microoperations in the given microprogram. Isoda et al. provide five rules for transforming the GDDG. The first rule is used to transform edges within the same basic block into data dependency represented edges. The second and the third rules are used for passing a microoperation upward or downward past a Y^{-1}-joint. These two rules are similar to Fisher's moving rules R6 and R5, respectively. The fourth and the fifth rules deal with passing a microoperation upward or downward past a Y-joint, when the Y-joint "is a loop's entry joint and movement is through the entry inlet of the loop, and through other inlets of the joint". Certain bookkeeping process are required to prevent subsequent illegal transformations.

After the GDDG is completed, the basic blocks are classified into 4 categories: $Y^{-1}Y$, YY, $Y^{-1}Y^{-1}$, and YY^{-1} according to the joints

connected to their entries and the joints connected to their exits. The microoperations are also classified into 4 categories: bounded, L-bounded, U-bounded, and unbounded. A bounded microoperation is a microoperation which cannot be moved past its resident basic block boundaries. An L-bounded microoperation of the basic block B is a microoperation which has been moved past B's top boundary (either moved down from B's predecessor or moved up from B itself) but has never been moved past B's bottom boundary. A U-bounded microoperation of the basic block B is a microoperation which has been moved past B's bottom boundary (either moved up from B's successor or moved down from B itself) but has never been moved past B's top boundary. A unbounded microoperation of the basic block B is a microoperation which has been moved past both B's bottom boundary and B's top boundary (either moved down from B's predecessor to B's successor or moved up from B's successor to B's predecessor). A bounded microoperation is also called a MUST microoperation since it must be compacted only in its resident basic block. In contrast, a MAY microoperation (either L-bound, U-bounded or unbounded) may be compacted into more than one of several basic blocks.

The idea of compaction is to try to compact MAY microoperations away from the most frequently executed basic blocks unless they can be included without increasing the length of those basic blocks. Isoda et al. give the order of processing basic blocks as below:

"1st) YY^{-1}-type blocks,

2nd) YY- or $Y^{-1}Y^{-1}$-type blocks, and

3rd) $Y^{-1}Y$-type blocks.

If two YY-type blocks($Y^{-1}Y^{-1}$-type blocks) are adjacent, the succeeding(preceding) one is processed first. YY^{-1}- or $Y^{-1}Y$-type blocks are never adjacent [so there is no need for determining their order of processing]."

Since this technique does not generate all possible microoperation movements (e.g. microoperation movements according to Fisher's moving rules R1, R2, R3, and R4 are not considered), the technique may not compare with the trace scheduling [MUEL83].

3.8. Rao and Majumdar's algorithm [RAO86]

The global compaction algorithm developed by Rao and Majumdar [RAO86] uses "a local compaction technique in which the microoperation priority function is dynamically modified." Each basic block, I, is given a weight called block dependency weight, BDW_I, which equals the number of descendent basic blocks of basic block I. The basic block with the largest BDW is to be compacted first and the one with the smallest BDW is the last to be compacted (any ties are broken by the source order). The movability of microoperations are determined by eight moving rules which are almost exactly the same as those given in [SU84] except that rule Su's R7 is

ignored and a rule prohibiting movement of a conditional jump microoperation beyond its resident basic block boundary is added. The microoperations are classified into delay, early, and critical microoperations. A delay microoperation is a microoperation that can be moved down from its resident basic block to the successors basic blocks. An early microoperation of a basic block B_I is a microoperation that can be moved up to basic block B_I from B_I's successor basic blocks. A critical microoperation is a microoperation that cannot be moved from its resident basic blocks. These classifications are similar to L-bounded, U-bounded, and bounded microoperations described in [ISOD83]. Associated with each early microoperation J of basic block B_I is a dynamic weight called a priority reduction value, P_J, which is the number of basic blocks in the trace from B_I to the resident basic block of J (i.e. the resident basic block of J must be one of B_I's successor basic blocks).

The global compaction starts with selecting the uncompacted basic block, B_I, which has the maximum BDW. If B_I's predecessor is compacted, move from B_I to its immediate predecessor, microoperations which can be put in the B_I's immediate predecessor without increasing its length. The path which has B_I as its root and contains the highest BDW is chosen as the trace. Determine a set of delay microoperations of B_I. As in trace scheduling, the trace is compacted as if it were a single basic block with list scheduling except:

1) Microoperations which are considered are those that are in the trace but not in the set of delay microoperations of B_I.

2) The priority assigned to microoperation J is calculate by $D_J + C_J - P_J$, where

D_J is the data dependency weight of J which is the number of microoperations dependent on J,

C_J is the resource conflict weight of J which is the number of potential microoperations that have resource conflict with J, and

P_J is the priority reduction value as defined before.

3) The list scheduling compaction stops when all microoperations originally in B_I are compacted; in trace scheduling the compaction stops when all microoperations in the trace are compacted.

After the list scheduling is completed, the microoperations in the set of delay microoperations of B_I which can stay in B_I without increasing its length are put in to the microinstructions in B_I. The rest of the microoperations in the set are moved down to the immediate successor basic block of B_I in the trace. Bookkeeping is performed to preserve the semantics of the microprogram. Then, another root basic block and a trace are selected and compacted again until all basic blocks are compacted.

Since moving conditional jump microoperations are forbidden, the topological structure of the control flow of the compacted program is unchanged.

4. The Kernighan-Lin partitioning algorithm

In [KERN70], Kernighan and Lin explain an algorithm for partitioning a weighted, undirected graph into subsets of nodes no larger than a given maximum size in such a way that the summation of weights (or costs) of edges connecting between nodes of different subsets is minimum [KERN70], [ULLM84].

The simplest partitioning problem is called the 2-way partitioning problem. It can be described as follow:

> " Let S be a set of 2n points, with an associated cost matrix C = (c_{ij}), i,j =1, ... , 2n. We assume without loss of generality that C is symmetric, and that $c_{ii} = 0$ for all i. There is no assumption about nonnegativity of the c_{ii}'s. We wish to partition S into two sets A and B, each with n points, such that the 'external cost' $T = \sum_{A X B} c_{ab}$ is minimal" [KERN70] p. 295.

Given a 2-way partition problem of S and C, let A and B be any arbitrary 2-way partition, and let A^* and B^* be a (minimal) solution of the problem. Obviously, there must exist subsets "$X \subseteq A$, $Y \subseteq B$ with $|X| = |Y| \leq n/2$ such that interchanging X and Y produces A^* and B^*".

The algorithm developed by Kernighan and Lin finds X and Y approximately, without considering all possible choices, by sequentially identifying their elements.

First, select a pair of nodes: $a_1' \in A$ and $b_1' \in B$ such that the gain, g_1, (i.e. the reduction in cost between the old external cost before the exchange and the new external cost after the exchange) produced by their exchanging is maximum. Second, select other pair: a_2', b_2' from A-$\{a_1'\}$ and B-$\{b_1'\}$ such that the gain, g_2, of their exchange, after the exchanging of (a_1',b_1'), is maximum. Third, select (a_3',b_3') from A—$\{a_1',a_2'\}$ and B-$\{b_1', b_2'\}$ such that the gain, g_3, of their exchange, after the exchanging of (a_1',b_1') and (a_2',b_2'), is maximum. Continue selecting node-pairs until all nodes are selected.

Chose k such that the partial sum $\sum_{i=1}^{k} g_i = G$ is maximum. If G>0, a reduction of the external cost can be achieved by moving $a_1',a_2',\ldots,a_k'$ from A to B and $b_1',b_2',\ldots,b_k'$ from B to A. After the movements are done, the resulting partitions are treated as the initial partitions and the process is performed again from the first step. If G=0, the partition is a locally optimum partition, which Kernighan and Lin call a phase 1 optimal partition.

By performing numerous experiments, Kernighan and Lin approximate the p(m): the probability that a phase 1 optimal solution of an m-node graph is a globally optimum partition, to be $2^{-m/30}$ with the running time that grows as $m^2\log(m)$.

5. The KLITS algorithm

The Kernighan-Lin partitioning algorithm has demonstrated a heuristic method dealing with a problem of optimizing the cost of a system by moving objects which causes the system cost to be changed. This underlying nature of the graph partitioning problem is not much different from the microprogram global compaction problem. In the microprogram global compaction, microoperations are moved across basic block boundaries and the movements are effect the microprogram (i.e. the system) cost. Because of these similarities, the underlying technique of the Kernighan-Lin partitioning algorithm can be useful for handling the microprogram global compaction problem.

In this paper, we develop and test a global compaction technique based on an adaptation of the Kernighan-Lin partitioning algorithm. We name this technique the Kernighan-Lin Like Iterative Topology and Structure-Preserving (KLITS) algorithm.

5.1. Moving rules

The semantics of the microprogram need to be preserved because the compacted microprogram, when given the same input as the original microprogram, must produce the same result as the original microprogram.

When a microprogram is globally compacted, moving micro-operations among basic blocks makes the task of debugging the compacted microprogram difficult. If the topological structure of the compacted microprogram is also different from the original, the debugging becomes much more difficult [LAH83].

The KLITS algorithm is developed for microprogram global compaction while preserving both semantics and the topological structure of the original microprogram.

In order to preserve the topological structure of the microprogram, the conditional-jump microoperation must not be moved out of its resident basic block. And, in order to preserve the semantics of the microprogram, when a microoperation is moved from one basic block to other(s), some conditions have to be satisfied before the movement can take place. In [FISH79], Fisher has cataloged "interblock microoperation motions" which consist of 3 conditions. Later, [FISH81] he broke them down to 6 rules "for the motion of MOP's [microoperations] to blocks other than the one they started in". These 6 rules are sometimes called moving rules. In [TOKO81], similar rules are use for transferring of microtemplates between contiguous segments. Two more rules and some additional conditions for conditional-jump microoperations movement are added in [LINN83], [SU84]. Since "it is also believed that the movement of conditional [jump] microoperations beyond basic blocks will not improve the microprogram compaction" [RAO86], Rao and Majumdar add the rule that prohibits conditional-jump microopera-

tions movement beyond basic blocks.

In this paper, the moving rules are combined and broadened so that they can be applied to any control flow structures such that if a basic block has immediate successors: B_I and B_J, neither B_J is a successor (either immediate or otherwise) of B_I nor vice versa. If the control flow structure does not fit in this category, as in Figure III.3(a)., it can be transformed so that the rules can be used, by adding some dummy empty basic blocks as shown in Figure III.3(b).

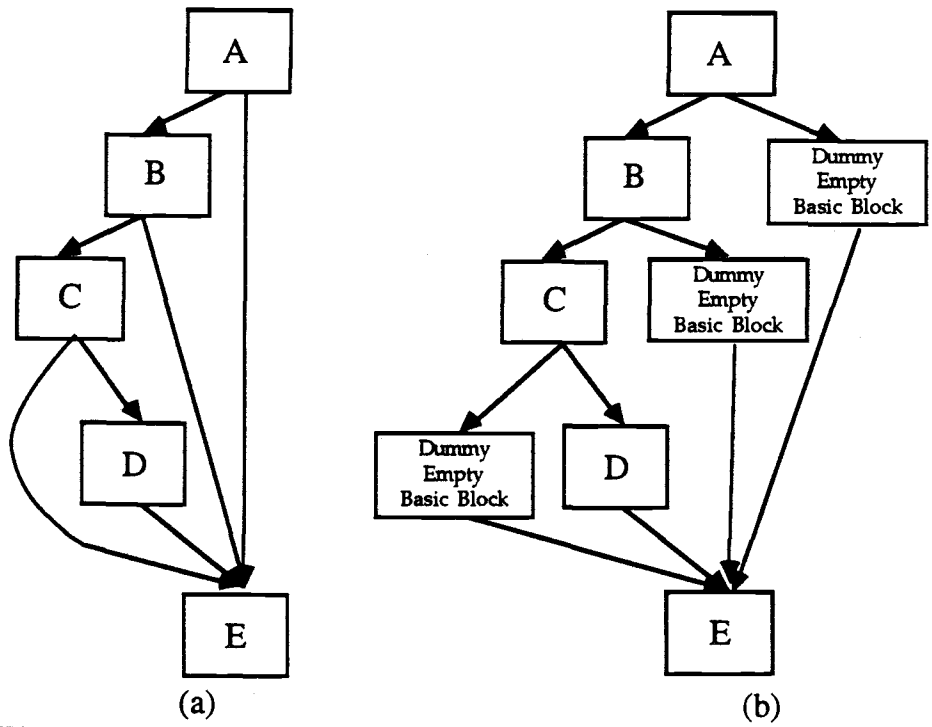


Figure III.3. Basic blocks structure transfromation example

The moving rules are divided into rules (or conditions) for moving a microoperation up and for moving a microoperation down. Both of them are presented in recursive form.

5.1.1. Moving up

To move a microoperation up, it must be free at the top of its resident basic block. If a microoperation is moved up (i.e. moved from its resident basic block to basic blocks preceding its resident basic block), copies of it must be put at the bottom of all its original basic block's predecessors. If a copy cannot be put at the bottom of any predecessor basic block, the microoperation cannot be moved up at all.

If a basic block does not terminate with a conditional-jump microoperation, a microoperation (or a copy of a microoperation) can always be added at its bottom. However, if the last microinstruction of a basic block contains a conditional-jump microoperation, a microoperation can be promptly added to the basic block's bottom only if the microoperation neither depends on the basic block's conditional-jump microoperation nor writes to the registers in the side-set of the basic block.

For moving a microoperation up, if the microoperation writes to the side-set of a basic block but is not data dependent on the conditional-jump microoperation, it still can be added at the bottom of the basic block if there are microoperations identical to it, which can be moved up to the basic block, in all the basic block's successors.

5.1.2. Moving down

To be moved down, the microoperation must be free at the bottom of its resident basic block. If a microoperation is moved down (i.e. moved from its resident basic block to the basic blocks following its resident basic block), its copies must be added to the top of all its former basic block's successors which have any of the microoperation's write registers in their live-at-the-top register sets. If its copies can not be put at the top of some of those successor basic blocks, the microoperation cannot be moved down at all.

A microoperation can be added at the top of a basic block that has some of the microoperation's write registers in its live-at-the-top register set, if there are microoperations identical to it which can be moved down to the basic block, in all the basic block's predecessors. However, if none of the microoperation's write registers are in the live-at-the-top register set of that basic block, the microoperation can be promptly added to the top of the basic block.

5.1.3. Additional restriction

To preserve the topological structure of the microprogram, an additional restriction is put on the conditional-jump microoperation movement:

The conditional-jump microoperation cannot be moved out from its original resident basic block.

5.2. Cost functions

The cost function is used to measure some of the microprogram properties that in which we have interested. The value that is produced by a cost function is called the cost of the microprogram. And, usually, we want the cost of the microprogram to be minimum. Hence, the best cost is the smallest cost. Since, for different application, we are interested in different properties of the microprogram, the cost functions are varied. For example, if we are interested in the size of a microprogram, the cost function for this application should reflect the total number of microinstructions in the microprogram.

In some global compaction techniques, the microprogram properties to be reduced are integrated into the algorithm. For example, in trace scheduling, the goal of the technique is to reduce the execution time of the most likely trace (and, probably, the average execution time of the microprogram). If other properties of the microprogram are emphasized, the algorithm has to be altered.

In the KLITS algorithm, the specific measure of the microprogram is not a part of the algorithm. Therefore, any cost functions are accommodated by the KLITS algorithm. With the KLITS algorithm,

reducing different properties of the microprogram can be done, without changing the algorithm itself, by using suitable cost functions.

For testing the KLITS algorithm in Part III, we limited the cost function to reflect the size of the microprogram. Hence, we use the total number of microinstructions to be the cost of the microprogram.

5.3. The algorithm

The Kernighan-Lin partitioning algorithm needs to be modified to apply to the global compaction problem since there are some differences between the graph partitioning problem and the global compaction as follow:

1) When a microoperation is moved, its copies may need to be added into several basic blocks; this process has no match in the graph partitioning problem.

2) Since the graph partitioning problem works on partitions of nodes, the microprogram have to be divided into partition also. The most obvious choice of partitions are basic blocks. However, there are more than two basic blocks in the microprogram which make the global compaction problem closer to the graph multiple partitioning problem.

In the Kernighan-Lin partitioning algorithm, after a node—pair is exchanged, the cost of unselected node—pairs are recalculated. Kernighan and Lin provided a short—cut method that reduces the recalculation times. In a microprogram global compaction, such a recalculation may take a long time and the short-cut method given in the Kernighan-Lin partitioning algorithm is not applicable to the problem. So, the algorithm for global compaction is as presented below.

1) Perform the live register analysis of all microoperations in the given microprogram. With this information, the live-at-the-top registers sets of each basic blocks are constructed. All microoperations are marked unselected. This is the beginning of a pass.

2) Create a ready-to-move sequence, or R sequence by

2.1) Initialize the R sequence as an empty sequence.

2.2) Chose an unselected microoperation from the microprogram. Use the moving rules to determine whether it can be moved. If it can be moved, its representative cost is calculated. The representative cost of a microoperation is the best (i.e. the smallest) incremental cost of the microprogram due to the movement of the microoperation beyond its current basic block boundary. The representation cost can be either positive (i.e. the

movement increases the microprogram cost), or negative (i.e. the movement decreases the microprogram cost), or zero (i.e. the movement does not change the microprogram cost). Mark the microoperation selected and add it to the R sequence in the order of its representative cost. Also record the direction of movement (up or down) that produced the representative cost.

2.3) Repeat 2.2) until all unselected microoperations are examined.

3) Take the first microoperation of the R sequence (i.e. the microoperation with the smallest representative cost) and move it, in its direction that was recorded in 2.2, according to the manner dictated by the moving rules. If it can be moved, put it (and its copies) in the furthest basic block(s) that causes the cost of the microprogram to be smallest. The change of the microprogram cost may be different from the microoperation representative cost calculated in 2.2 since the position of microoperations in the microprogram is changed due to the fact that other microoperations have been moved. Delete the microoperation from the R sequence but keep it (and its copies) marked as selected. If the microoperation cannot be moved in the direction recorded, the microoperation is also deleted from

the R sequence but it now is remarked as unselected. The cost of the microprogram is recorded.

4) Update the live-at-the-top register sets of basic blocks affected by the movement. Usually, if the direction of movement is up, only those basic blocks that the MO and its copies were moved from need to be updated; if the direction of movement is down, only those basic blocks that the MO and its copies were moved to need to be updated. For more detail about how to update live-at-the-top registers sets see Part III Section 4.3.

5) Determine if there are any unselected MO's that have now become movable because of the MO movement of step 3. This is the same logic as the process described in 2.2). However, in order to save time, the previously calculated values of representative costs for the remaining R—sequence are not reevaluated. Rather, the values from before the first MO movement are kept as estimates.

6) Repeat 3) to 5) again until the R sequence is empty. This is the end of the pass.

7) Examine the series of MO movements, and identify the last MO movement which produced the smallest net MP cost; call it MO*. Then, start from the end of the series, reverse the MO movements after MO* to bring them back to the basic blocks from which

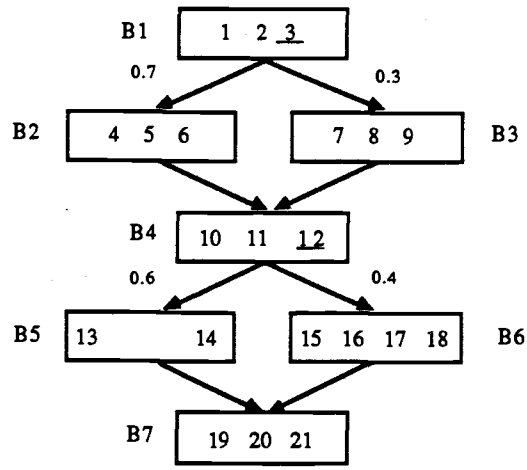
they were moved.

8) Repeat 1) to 7) until the MO* in 7) is the beginning of the moving sequence. Since some MO movements have zero-cost, it is possible that the MO movements from the beginning of the pass to the MO* movement are all zero-cost movements. And, sometimes, all these movements are reversed in the next pass, which can cause the algorithm to run perpetually. To prevent this undesirable situation, a integer k is selected as a bound of how many consecutive occurrences of zero-cost passes are allowed.

6. Detailed example

This example is a compaction of the same example microprogram used by Lah and Atkins to demonstrate the tree compaction algorithm and to compare results with trace scheduling [LAH83]. The purpose of this example is to illustrate in detail the steps of KLITS compaction and also, at the same time, to compare results with both tree compaction and trace scheduling.

The basic blocks structure with probability of execution of each branch taken from [LAH83] is shown in Figure III.4(a).



(a) Basic blocks structure (with probability of execution of each branch) [LAH83]

Block	Mo's	Name	Read Registers		Write Registers	Resource Vector
B1	1		R2,	R3	R1	0 1 1 0
	2		R1,	R5	R10	0 0 1 0
	3	CJUMP	R3,	R4	R2	1 0 1 0
	4		R9,	R5	R8	0 1 0 0
B2	5		R2,	R5	R2	0 0 0 1
	6	GOTO	R4,	R8	R15	0 1 0 0
	7		R8,	R8	R9	0 1 0 1
B3	8		R1,	R5	R1	0 1 0 0
	9		R11,	R12	R8	0 0 0 1
	10		R2,	R13	R14	0 1 0 1
B4	11		R3,	R5	R7	0 1 0 0
	12	CJUMP	R4,	R10	R6	1 0 0 0
B5	13		R3,	R4	R9	1 0 1 0
	14	GOTO	R3,	R6	R6	0 1 0 0
	15		R6,	R8	R2	0 0 0 1
B6	16		R2,	R14	R7	0 1 0 1
	17		R2,	R5	R4	0 1 0 0
	18		R11,	R12	R14	0 0 1 0
	19		R6,	R10	R9	0 0 0 1
B7	20		R7,	R3	R13	0 0 0 1
	21	STOP	R4,	R11	R3	0 1 1 0

(b) List of microoperation repersentations [LAH83]

Figure III.4. An example microprogram from [Lah83]

The list of microoperation representations is shown in Figure III.4(b). In the example, the microprogram consists of 7 basic blocks: B1 to B7, and 21 microoperations: 1 to 21. From the probability of execution of each control branch, the major (most likely) trace is B1, B2, B4, B5 and B7. In order to make a comparison with local compaction, we apply list scheduling local compaction to the basic blocks of the microprogram. The compacted microprogram has 15 microinstructions as shown in Figure III.5(a). We also use this value as the original cost of the MP.

In order to distinguish the step number taken in the example from the segment number of the algorithm described in 4.3, the step of the example is shown in Roman numerals while its corresponding algorithm segment number is shown in Arabic numbers in square parenthesis. The steps of the KLITS compaction are as following:

Pass 1

Step I [1]: The live-at-the-top register set of each basic block is shown below. For example, R7 is live at the top of B7 because it is read by MO 20 and there is no microoperation in B7, before MO 20, that writes to it. R7 is also live at the top of B5 since no microoperation in B5 writes to it either. However, R7 is not live at the top of B6 since it is written by MO 16 in B6.

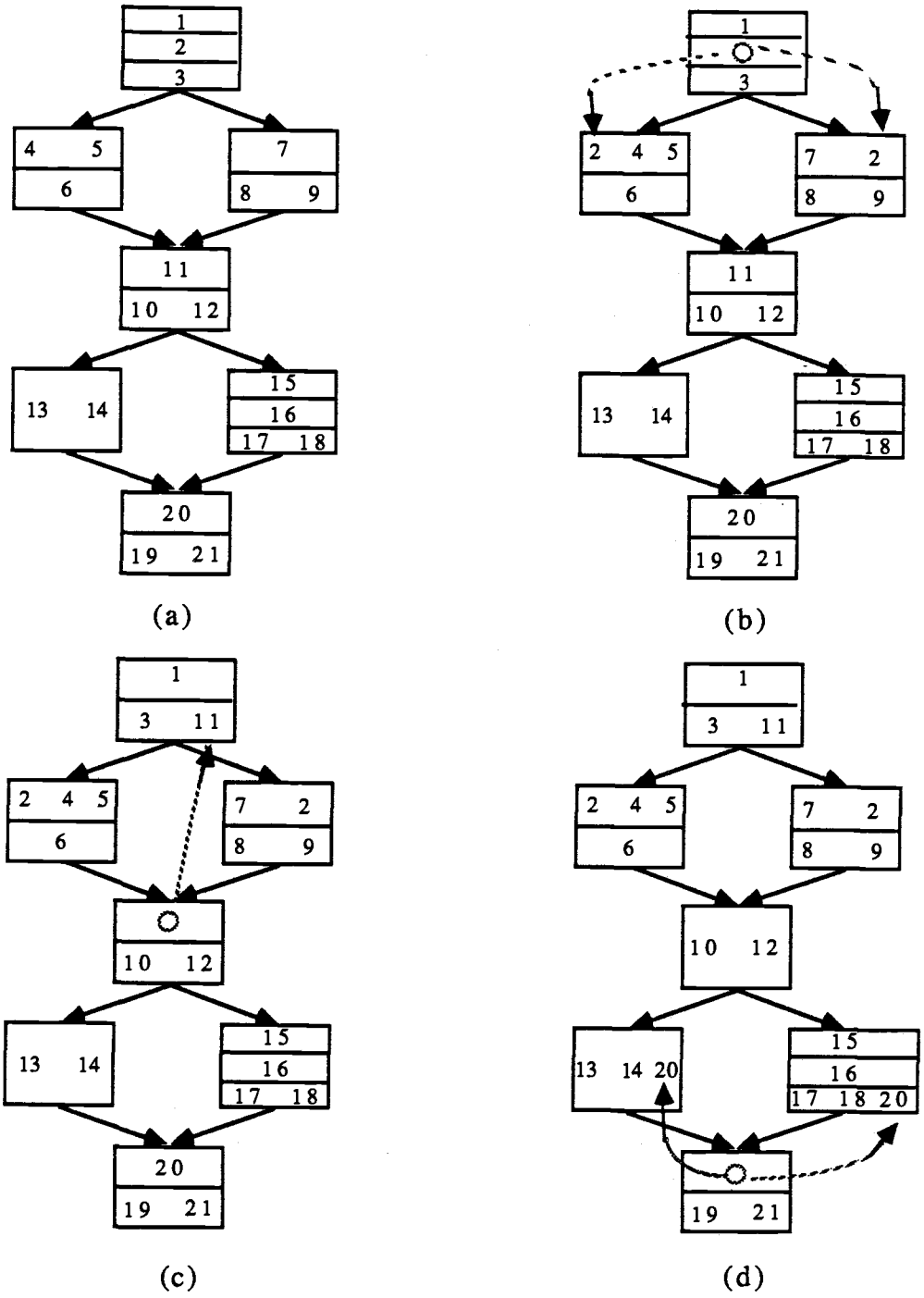


Figure III.5. Detailed example steps group 1

Basic Block	Registers Live at the Top
B1	R2, R3, R4, R5, R8, R9, R11, R12, R13
B2	R2, R3, R4, R5, R9, R10, R11, R12, R13
B3	R1, R2, R3, R4, R5, R8, R10, R11, R12, R13
B4	R2, R3, R4, R5, R8, R10, R11, R12, R13
B5	R3, R4, R6, R7, R10, R11
B6	R3, R5, R6, R8, R10, R11, R12, R14
B7	R3, R4, R6, R7, R10, R11

Then, every microoperation is initially marked as unselected.

Step II [2]: Each microoperation is examined for its movability. If the microoperation is movable, we put it into the ready-to-move sequence, R sequence, in the order of its cost. For example, from the basic blocks structure, MO 1 can possibly be moved down only because B1 is the first basic block of the microprogram (i.e. there is no basic block for MO 1 to move up to. But, since MO 2 is dependent on MO 1, MO 1 is not free at the bottom of B1 and, therefore, MO 1 cannot be moved down from B1 (i.e. below MO 2) without changing the microprogram semantics. On other hand, 2 is free at the bottom of B1; therefore it can be moved down. If 2 is moved out of B1, the cost of B1 (i.e. the number of microinstructions in B1) will be reduced by 1 (from Figure 2a.). Then, since 2 writes to R10 and R10 is live at the tops of both B2 and B3, 2 must be moved into both B2 and B3. After MO 2 is moved to B2 and B3, R10 is not live at the top of B2

and B3 any more. MO 2 has neither resource conflict nor data dependency with 4 nor 5; so, it can be put into the same microinstruction with 4 and 5 which makes the cost of adding 2 to B2 equal to 0. Similarly, 2 can be combined with 7 to form a microinstruction; the cost of adding 2 to B3 is also 0. The total cost of moving 2 from B1 to B2 and B3 is -1+0+0 = -1. MO 2 cannot be moved further because R10 is live at the top of B4 and although 2 is free at the bottom of B2, it is not free at the bottom of B3 since 8 is data dependent on 2. So, the best moving cost of 2 (i.e. the changing in the total number of microinstructions of the microprogram) is —1. We mark 2 as selected and put it into the ready-to-move sequence, or R sequence. We keep examining the microoperations until all microoperations are examined. The R sequence is shown below in the (Name,Cost,Dir) format, where Name is the name of a microoperation, Cost is the microoperation moving cost (i.e. representative cost) and Dir is the direction of moving (U for up and D for down).

R Sequence : { (2,-1,D); (11,-1,U); (20,-1,U); (8,0,U);
(10,0,D); (13,0,U); (18,0,D)}.

Note:

1.) All these MOs are marked as selected.

2) The MOs in the R sequence are ordered according to their costs. The order of their appearance in the vertical microprogram (i.e. their names) is used as a tie-breaker.

3) The (representative) cost of a MO is the smallest number of MIs that are added when the MO is moved according to the current configuration of the

MP (i.e. as in Figure III.5(a)). In the case that a MO can be moved in either directions (up and down) with the same representative cost, the direction of movement recorded for that MO is up (U).

Step III [3]: From step II, 2 is the first microoperation in the R sequence; therefore, we first move 2 down from B1 to both B2 and B3. The cost of the MP is now 14 as shown in Figure III.5(b).

Step IV [4]: After 2 is moved, it is deleted from the R sequence and the live-at-the-top register sets of B2, B3 and B1 are updated.

Step V [5]: All unselected MOs are checked again to determine whether any of them can be added to the R sequence (Note: 2's in both B2 and B3 are still marked as selected). However, there is no MO that can be added to the R sequence; therefore,

R Sequence : { (11,-1,U); (20,-1,U); (8,0,U); (10,0,D); (13,0,U); (18,0,D)}.

Step VI [6,3]: From the R sequence, 11 is the next microoperation to be moved and the direction of movement is up. Since the register that is written by 11, R7, is live at the top of B5, we must move 11 to both B2 and B3. However, two extra microinstructions, one in B2 and the other one in B3, are then needed because 11 has resource conflict with 4 and 6 in B2, and 7 and 8 in B3. Because of these conflicts, the cost of moving 11 from B4 to B2 and B3 is 1 (—1 from B4 +1 for B2 and +1 for B3). But, in both B2 and B3, 11's are

free at the top; so, we move both 11's up to B1 where it shares an MI with 3, saving two MIs for a net cost of +1—2=—1. The cost of the MP is now 13 as illustrated in Figure III.5(c). Again, we delete 11 from the R sequence.

Step VII [4]: Update live-at-the-top register sets of B4, B3, B2 and B1.

Step VIII [5]: We try to find unselected MOs to add to the R sequence, but none is found. The R sequence is shown below.

R Sequence : {(20,-1,U); (8,0,U); (10,0,D); (13,0,U); (18,0,D)}

Step IX [6,3]: From the R sequence, 20 is the first MO in the sequence. Since 20 is still free at the top of the B7 it can be moved up. We move 20 up from B7 to both B5 and B6 where in each case it fits in an existing MI (i.e. {13,14} and {17,18}, respectively) without conflict. The cost of the movement is -1. The cost of the MP is now 12 as shown in Figure III.5(d). As before, 20 is deleted from the R sequence.

Step X [4]: The live-at-the-top registers set of B7 is updated.

Step XI [5]: Again, we cannot find any unselected MOs to be added to the R sequence. The R sequence is now as shown below.

R Sequence : {(8,0,U); (10,0,D); (13,0,U); (18,0,D)}

Step XII [6,3]: The R sequence hints that 8 should be moved up since in step I, 8 could be moved from B3 to B1 because it was free at the top of B3. However, in step III, 2 was moved to B3 and 8 is data dependent on 2: 8 writes over R1 which is read by 2, so, at this step, 8 is no longer free at the top of B3. Hence 8 cannot be moved up as illustrated in Figure III.6(a). We remove 8 from the R sequence and mark it as unselected.

Step XIII [4,5]: When all unselected MOs are examined, 8 is examined as well. We find that 8 can be moved down and the cost of the moving is 0. Therefore, we add 8 to the R sequence. The R sequence is shown below.

R Sequence : {(8,0,D); (10,0,D); (13,0,U); (18,0,D)}

Step XIV [3]: 8 is moved down from B3 to B6. The cost of the movement is 0. So, the cost of the MP is still 12 as shown in Figure III.6(b). After all information is update [4], no MO can be added to the R sequence [5]. The R sequence is shown below.

R Sequence : {(10,0,D); (13,0,U); (18,0,D)}

Step XV [6,3,4,5]: 10 is moved down from B4 to B6 [3]. However, the cost of moving 10 down at this step is 1 which is not the same as the cost calculated at step I. In step 1 (i.e. the original configuration of the MP: Figure III.5(a)), 10 had to be moved down to B6 since the register that it writes (R14) was live at the top of B6 (but was not live at the top of B5). When 10 was moved out of B4,

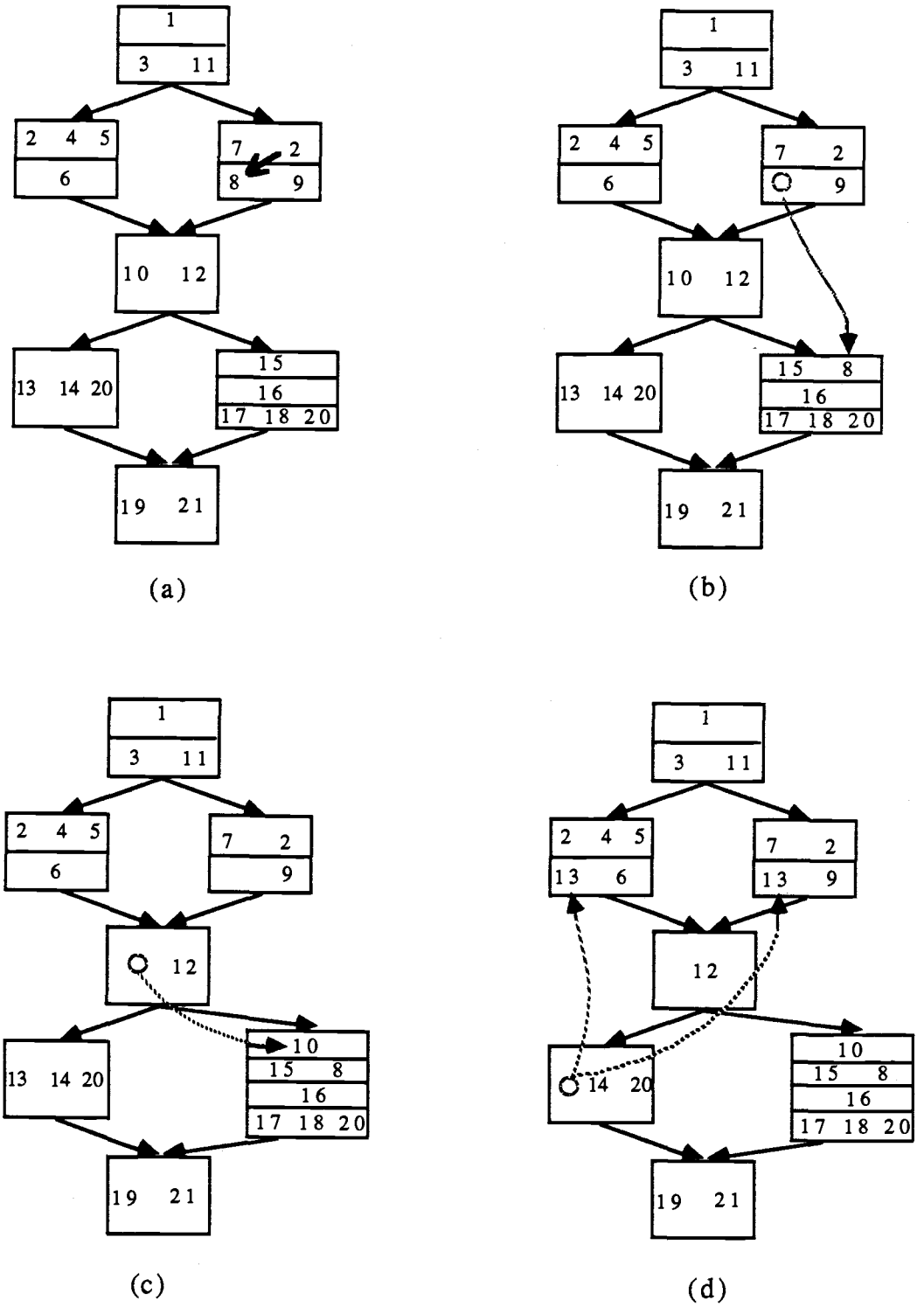


Figure III.6. Detailed example steps group 2

11 and 12 could be combined into a single microinstruction; so, the cost of moving 10 out was —1. But to add 10 into B6, the new microinstruction had to be created for 10, since it was in conflict with 15. Therefore, at step I, the total cost of moving 10 was 0.

At this step (step XV), however, B4 has only one microinstruction: {10,12}. The cost of moving 10 out of B4 is, therefore, 0. This makes the total cost of moving 10 down to be 1. The cost of the MP is increased to 13 as shown in Figure III.6(c). After performing [4,5], the R sequence at the end of this step is

R Sequence : {(13,0,U); (18,0,D)}

Step XVI [6,3,4,5]: We move 13 up from B5 to B2 and B3 [3]; the cost of the movement is 0. The cost of the MP is 13 as shown in Figure III.6(d), and the R sequence is

R Sequence : {(18,0,D)}

Step XVII [6,3,4,5]: The only MO left in the R sequence is 18. We move it down from B6 to B7. The movement makes the MP cost to be 14 as shown in Figure III.7(a). Since no unselected MO can be added to the R sequence, the R sequence is now empty.

Step XVIII [7]: The costs of the MP from step I to step XVII are compared; the smallest ones appeared at step IX and step XIV which are 12. The latest step of them is chosen. The rest of the

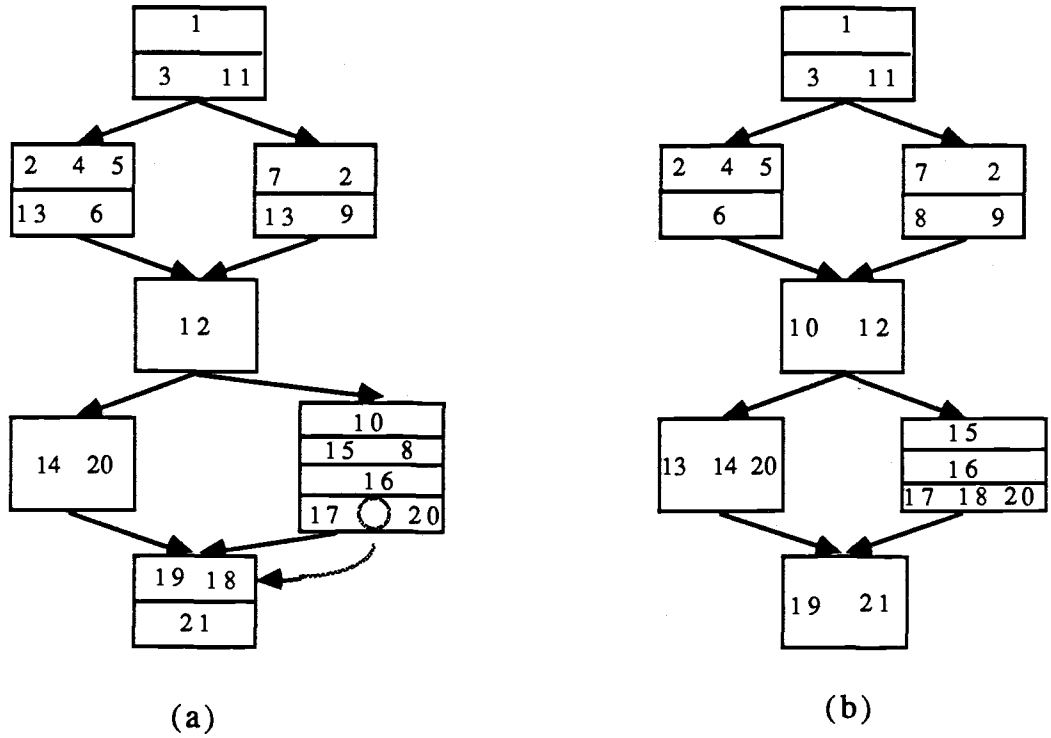


Figure III.7 Detailed Example Steps Group 3

sequence of the microoperation movements (step XV to step XVII) are reversed: 18 is moved from B7 back to B6, 13 from B2 and B3 back to B5, and 10 from B6 back to B4. The MP is now as shown in Figure III.7(b).

Pass 2

Step XIX [1]: All MOs are now marked as unselected and examined. Those that are movable are put in the R sequence as follow:

R Sequence : { (13,0,U); (2,1,U); (2,1,U); (8,1,U); (10,1,D); (11,1,D); (18,1,D); (20,1,D); (20,1,U)}.

Note: The cost of moving 8 up is 1 because the moving rules that are used to determine the movability of MOs are not "symmetric". They allow a MO which writes to a register that is dead at the top of a successor basic block to be moved down to that basic block without any additional conditions. However, when a MO is moved up from its current resident basic block, its copies have to be added to all predecessors of the current basic block.

Step	MO's Name	From	To	MO's Cost	MP's Cost	R Sequence
XXI	13	B5	B2,B3	0	12	(2,1,U); (2,1,U); (8,1,U); (10,1,D); (11,1,D); (18,1,D); (20,1,D); (20,1,U)
XXII	2	B2,B3	B1	1	13	(8,1,U); (10,1,D); (11,1,D); (18,1,D); (20,1,D); (20,1,U)
XXIII	8	B6	B1	0	13	(10,1,D); (11,1,D); (18,1,D); (20,1,D); (20,1,U)
XXIV	10	B4	B6	1	14	(11,1,D); (18,1,D); (20,1,D); (20,1,U)
XXV	11	B1	B4	0	14	(18,1,D); (20,1,D); (20,1,U)
XXVI	18	B6	B7	1	15	(18,1,D); (20,1,D); (20,1,U)

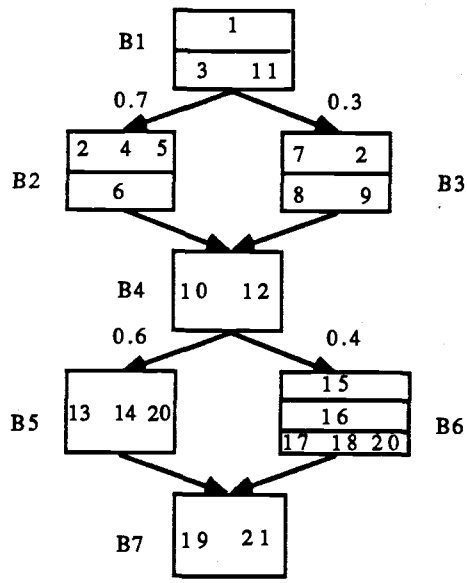
Step XXVII: Move 20 from B6 and B5 to B7, the moving cost is 0. The microprogram cost is 15. The R sequence is now empty again; the best cost is at step 10. The movements from step XXI to step XXVII are revised. All the MOs are marked unselected and examined. The R sequence is reconstructed. In this pass (step XX to step XXVII), all the moving from the beginning to the best cost are zero-cost movements. If the movements in the sequence to achieve the best cost in the next pass are all zero-cost too, the compaction will terminate.

The next pass has 7 steps (step XXVIII to step XXXIII), and the sequence of microoperation movements are very similar to the previous pass (i.e. pass 2). The sequence from the beginning of the pass up to the best microprogram cost consists of only one step: step XXVIII which is moving 13 from B2 and B3 to B5 with moving cost of 0. Since for the second pass in a row the sequence consists of only zero-cost movements, the compaction is terminated. The final compacted microprogram is shown in Figure III.9(a).

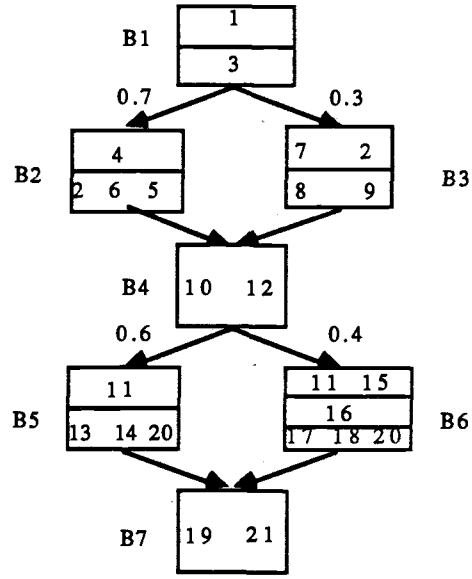
The results of tree compaction and trace scheduling on the same microprogram are shown in Figure III.9(b) and (c), respectively. The comparison summary is shown in Figure III.8.

	Time in the major trace	Weighted time	Space needed	Space time product
KLITS Compaction	7	7.8	12	93.6
Tree Compaction	8	8.4	13	109.2
Trace Scheduling	7	7.52	21	157.92
List Scheduling local Compaction	10	10.8	15	162.0

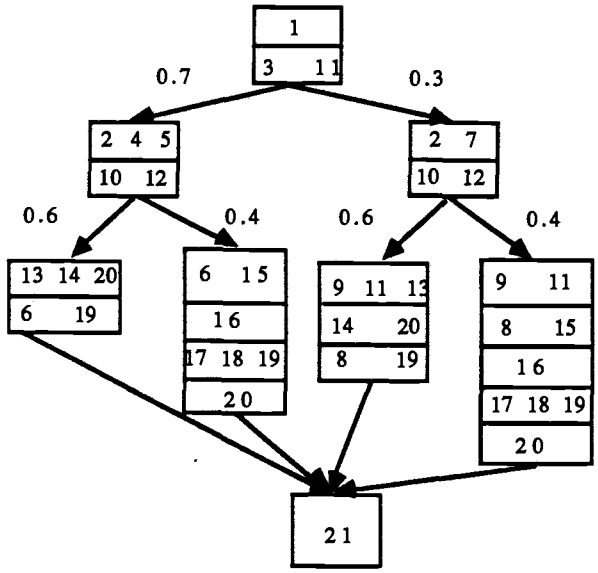
Figure III.8. Comparison summary



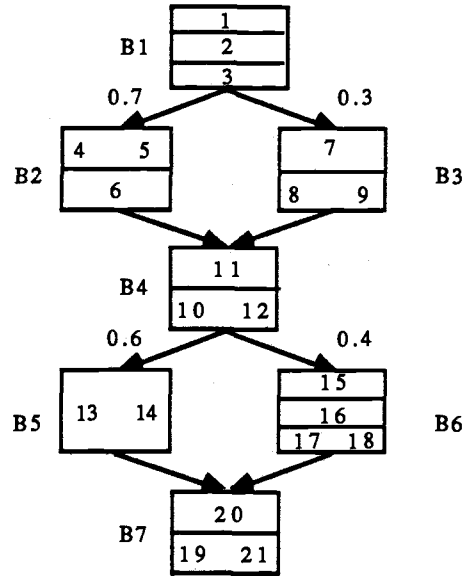
(a) KLITS compaction result



(b) Tree compaction result



(c) Trace scheduling result



(d) List scheduling local comapction result

Figure III.9. Compaction results

In this example, for each basic block containing a conditional-jump microoperation, the probabilities of execution of each branch were given. So, the probability of each basic block to be executed can be calculated. Using these probabilities, the major trace (i.e. the trace which has the highest probability of execution) can be picked.

The time in trace is the number of microinstructions in the major trace after the compaction. The weighted time is the summation of the product of the number of microinstructions in the basic block and the probability that the basic block is executed, for every basic block in the microprogram. This should be the same as the average execution time of the compacted microprogram, assuming each microinstruction has the same execution time. The space needed is the total number of microinstructions in the compacted microprogram. And, the space-time product is the product of the weighted time and the space needed.

The KLITS compaction shows improvement over the tree compaction in all categories. The trace scheduling shows the same number of microinstructions in the major trace and 3.58% improvement in weighted time over the KLITS compaction. However, the extra space used to make that 3.58% improvement is a 75% increase. The space-time produce of the trace scheduling is 74.76% bigger than the KLITS result.

7. Summary and conclusions

The microprogram global compaction algorithm called KLITS uses moving rules to move microoperations across basic block boundaries; however, by utilizing Kernighan-Lin like iterations, the whole search tree need not be considered before a good solution can be found. Instead, the KLITS algorithm is divided into passes. Each pass constructs and revises a sequence of microoperation movements. This sequence will produce a microprogram that has no worse cost than the one at the beginning of the sequence, although it may contain some movements that, by themselves, increase the microprogram cost. A sequence of microoperation movements is constructed by ordering microoperation movements according to the changed in the microprogram cost due to those movements. In each sequence, each microoperation may be moved no more than once. Then, the sequence is revised in such a way that the sequence ended at the microoperation that produces the microprogram with the smallest cost. The algorithm is terminated when the sequence of the pass cannot produce a better microprogram than the one it starts from.

By not integrating any specific cost measure to its algorithm, the KLITS algorithm is more adaptable to the needs of users than trace scheduling and global compactions that following its foot steps.

The KLITS algorithm retains the original topological structure. In contrast, the microprogram compacted by trace scheduling may have a changed topological structure as illustrated by Figure III.4(a) and Figure III.9(c). This may make debugging much more difficult.

Tree compaction also preserves the topological structure of the original microprogram. However, in the KLITS algorithm, micro-operations can be moved farther than allowed by tree compaction. Hence, the chance that a microoperation can be combined with other microoperations to form a microinstruction is greater in the KLITS algorithm.

The KLITS algorithm presented here showed encouraging result with a previously published example. The KLITS compaction showed improvement over tree compaction in all categories. Trace scheduling showed the same number of microinstructions in the major trace and 3.58% improvement in weighted time over the KLITS compaction. However, the extra space used to make that 3.58% improvement was a 75% increase. The space-time product of trace scheduling was 74.76% bigger (worse) than the KLITS result.

Both tree compaction and trace scheduling have the primary goal of reducing the execution of the major trace while the goal of the KLITS compaction can be easily changed, without altering the algorithm, by using a different cost function. The cost function used in the example was the number of total microinstructions of the microprogram. And from the example, the KLITS produced the

smallest number of the microinstructions.

In order to understand more of the KLITS algorithm behavior, the algorithm was implemented and used on samples from a model of microprograms (pseudo-microprograms). The results of experiments are reported and discussed in Part III.

8. References

[DASG79] S. Dasgupta,
"The Organization of Microprogram Stores",
ACM Computing Surveys, Vol.11, No. 1, March 1979, pp. 39-65.

[DAVI81] S. Davidson, D. Landskov, B. D. Shriver, and P. W. Mallett;
"Some Experiments in Local Microcode Compaction for Horizontal Machines",
IEEE Transactions on Computers, C-30,7; July 1981, pp. 460-477.

[FISH79] J. A. Fisher,
"The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling With Resources", Phd. Dissertation, Department of Computer Science, New York University, New York, 1979.

[FISH81] J. A. Fisher,
"Trace Scheduling: A Technique for Global Microcode Compaction",
IEEE Transactions on Computers, C-30,7 July 1981, pp. 478-490.

[FISH83] J. A. Fisher,
"Very Long Instruction Word Architectures and the ELI-512",
Proceeding of the 10th Symp. Computer Architecture, Stockholm, June 1983, pp. 140-150.

[ISOD83] S. Isoda, Y. Kobayashi and T. Ishida;
"Global Compaction of Horizontal Microprograms Based on the Generalized Data Dependency Graph"
IEEE Transaction on Computers, C-32,10; October 1983, pp. 922-933.

[KERN70] B. W. Kernighan, and S. Lin;
"An Efficient Heuristic Procedure for Partitioning Graphs",
The Bell System Technical Journal, Vol. 49, No. 2; February 1970, pp. 291-307.

[LAH83] J. Lah, and D. E. Atkins;
"Tree Compaction of Microprograms",
Proceedings of the Sixteenth Annual Microprogramming Workshop, October 11th-14th, 1983, pp. 23-33.

[LAND80] D. Landskov, S. Davidson, B. Shriver, and P. W. Mallett;
"Local Microcode Compaction Techniques",
ACM Computing Surveys, Vol.12, No. 3, September 1980, pp. 261-294.

[LINN83] J. L. Linn,
"SRDAG Compaction: A Generalization of Trace Scheduling to Increase the Use of Global Context Information",
Proceedings of the Sixteenth Annual Microprogramming Workshop, Oct. 1983, pp78—85.

[MUEL86] R. A. Mueller, and V. H. Allan,
"Phase-Coupled Methods in Automated Microcode Generation: Assessment and Strategy",
Proceedings of the Nineteenth Annual Hawaii International Conference on System Science, 1986, pp 505-519.

[RAO86] S. U. Rao and A. K. Majumdar;
"An Algorithm for Global Compaction of Horizontal Microprograms", *SIGMICRO TCMICRO Newsletter*, SIGMICRO: Vol 17, No. 2; TCMICRO: Vol 1, No. 2; July 1986, p.19-30.

[SU84] B. Su, S. Ding, and L. Jin;
"An Improvement of Trace Scheduling for Global Microcode Compaction",
Proceedings of the Seventeenth Annual Microprogramming Workshop, New Orleans, Louisiana, October 30th - November 2nd, 1984, pp. 78-85.

[TOKO78] M. Tokoro, T. Takizuka, E. Tamura, and T. Yamaura;
"A Technique of Global Optimization of Microprograms",
Proceedings of the Eleventh Annual Microprogramming Workshop, November 19-22,1978; pp 41-50.

[TOKO81] M. Tokoro, E. Tamura, and T. Takizuka;
"Optimization of Microprograms",
IEEE Transaction on Computers, C-30,7 July 1981, pp. 491-504.

[ULLM84] J. D. Ullman,
"Computational Aspects of VLSI", Computer Science Press, Inc. Rockville, Maryland, 1984.

Chapter IV Experiments on the KLITS Algorithm

Abstract

Microprogram global compaction has been widely studied. In Part II we developed a global compaction algorithm based on an adaptation of the Kernighan-Lin partitioning algorithm. This new algorithm, called KLITS, has been applied to some previously published example microprograms. The results on these examples are encouraging.

This paper further tests and studies the KLITS algorithm. A method for modeling microprograms as pseudo-microprograms is described. In order to test the KLITS algorithm, both pseudo-microprogram generation and KLITS global compaction are implemented. For each pseudo-microprogram generated, both local compaction (list scheduling) and the KLITS global compaction technique are applied. The results from both techniques are reported and compared. The behavior of the KLITS algorithm in experiments on pseudo-microprograms is discussed in terms of compaction time and improvement

over local compaction.

1. Introduction

The simulation experiments in Part III are intended to test a global compaction technique named the Kernighan-Lin Like Iterative Topology and Structure-Preserving (KLITS) algorithm. The purpose of testing the KLITS algorithm, which was constructed in Part II, is to better understood its behavior. In order to test the KLITS algorithm on large amounts of microprogram some kind of automatic generation of vertical microprograms is essential. The "microprogram" generated does not need to be a real microprogram, it only needs those characteristics of a real microprogram actually used by compactions. For example, the compaction does not need the specific functions of each microoperation, but the compaction must be able to determine data dependencies and resource constraints among microoperations. A microprogram generated in the experiment is called a pseudo-microprogram, or PMP. The process of generating a pseudo-microprogram is divided into two parts: the basic block structure generation, and the pseudo-microoperation or PMO generation.

2. The model of microprogram used in the simulation (pseudo-microprogram)

The microprograms used in the experiments are not real microprograms. They are representation of the information about microprograms needed for compaction. These model microprograms, or pseudo-microprograms, are described in two sections: the model microoperations, or pseudo-microoperations, and the control flow structures.

2.1. The pseudo-microoperations

The pseudo-microoperations (PMOs) encode the information required for proper compaction of microoperations in a microprogram. The PMO representation is adapted from the six-tuple, <name, I, O, U, T, F>, used for representing microoperation semantics described in [LAND80], [DAVI81]. The tuple elements there were

> "(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 required 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)." [LAND80]

The elements (5) and (6) are dropped[1] to make local compaction easier. Therefore, each PMO consists of

1) the name[2] or identifier of the microoperation,
2) Iset, the set of registers read by it,
3) Oset[3], the set of registers written by it, and
4) Uset, the set of operators (or transitory storage units, or resources) used by it.

2.2. The control flow structure

A microprogram is composed of basic blocks. A basic block is a longest sequence of microoperations such that it has only a single entry and a single exit. The next microoperation of each microoperation is, usually, found at the next address. However, sometimes the next address may not be in the same basic block if the next address is a destination of a jump (either conditional or unconditional) microoperation.

A basic block is usually, but not always, terminated by a jump microoperation. If the basic block is terminated by a jump microoperation, the microoperation executed next after the jump microoperation is the first microoperation of the destination basic block. In the case of a conditional-jump microoperation, there may be more

1. The PMO becomes a single phase uncoded microoperation.
2. In [LAND80], [DAVI81] the name indicates the function of the MO (e.g. ADD, MOVE). For a PMO, its name is only used to distinguish it from another.
3. For the conditional-jump pseudo-microoperation, the Oset is empty [SU84].

than one next microoperation.

The control flow of a microprogram is illustrated by the connections between basic blocks in the microprogram. Control flow is usually represented by a directed graph whose nodes represent basic blocks of the microprogram. An edge connects the bottom of one basic block to the top of each basic block which may contain the next microinstruction. For each microinstruction in the microprogram, the next microinstruction may be either the one at the next address (which may or may not be in the same basic block with the current microinstruction) or the first microinstruction of another basic block which is the destination of either a conditional-jump or an unconditional-jump microoperation that terminates the current basic block. If a block B_J may be executed right after the block B_I is executed for some set of data then, there must exist an edge from B_I to B_J.

3. Generation of the pseudo-microprogram

For testing the KLITS algorithm, pseudo-microprograms are generated and then compacted by the algorithm. In this section the process of generating a pseudo-microprogram is discussed. The process is divided in two: generation of the basic block structure, or the control flow structure, and generation of the pseudo-microoperations. A control flow structure is generated first with all its basic blocks initially empty. Then, PMOs are generated and put into these basic blocks.

3.1. Generation of the control flow structure

In these experiments, the control flow is generated by randomly assigning empty basic blocks into levels, then randomly connecting basic blocks from one level to basic blocks in the immediate successor level, while observing some conventions adopted to ensure reasonable-looking control-flow graphs.

The control flow for a PMP is generated level by level until the specified total number of basic blocks in the PMP (#BLOCK) is reached. Except for the first and the last level which have only one basic block each, the number of basic blocks in each level is a pseudo-random integer chosen from the range [1,#MAXBLOCKPERLEVEL]. However, if any level has a single basic block, the next level must have more than one basic block. This is forced by continuing to generate a new number of basic blocks for this next level until it is greater than one. If the number picked for the number of basic blocks of that level is greater than or equal to the number of basic blocks left, the number of the basic blocks used is instead the number of basic blocks left minus one (e.g. if the total number of basic blocks generated so far is, say, #BLOCK-4 and the number picked for the next level is 4 or greater, the number of basic blocks for that level is changed to 3). If the total number of basic blocks in all levels generated so far (including the first level) is two less than #BLOCK, this means that the next level will be the level immediately before the last level and it would have a single basic block. Instead of building one more level with one basic block, the

"extra" basic block is added to the current level generated, and the program terminates with one block in the last level.

After the total number of basic blocks in all levels equals #BLOCK, the next step of generating the control flow is to determine the control flow paths between these basic blocks. For our simulation, only loop-free structures are considered. This means there exists no control flow path through one level back to an earlier level.

Control flow paths only exist between basic blocks in one level and the basic blocks in the immediate following level. For each basic block, B_I, basic blocks from the level immediately following its resident level are picked randomly #MAXBRANCH times. These basic blocks are used as the destination of the control flow paths which are originated from B_I. Repetition is allowed in the picking; therefore, some basic blocks may have less than #MAXBRANCH branches radiated from them. When all basic blocks in the current level have successors, the next level becomes the current level. Before the control flow paths from this level to the next level are defined, all basic blocks have to be checked for two qualifications:

1) every basic block must have at least one predecessor (except, of course, in the first level), and
2) if a basic block has only a single predecessor in the previous level, that predecessor must have other successors.

If the first qualification is not true, a basic block from the previous level is picked randomly and a control flow path from that picked basic block is established.

If a basic block, B_I, fails the second qualification, another basic block of the same level is randomly picked to be another successor of the predecessor of B_I.

Because of these remedies, some basic blocks may be a source of more than #MAXBRANCH number of control flow paths.

3.1.1. An example

This example will demonstrate the process of generating the control flow of a pseudo-microprogram. The parameters used in the generation process are #BLOCK (i.e. the total number of basic blocks in the PMP), #MAXBLOCKPERLEVEL (i.e. the setting number of maximum basic blocks per level), and #MAXBRANCH (i.e. the target branch capacity per basic block). The values of these parameters are shown below.

#BLOCK	= 7
#MAXBLOCKPERLEVEL	= 4
#MAXBRANCH	= 2

The first level and the last level have only one block each. So, there are 5 basic blocks left to be assigned to other levels. Random

integers from 1 to 4 (#MAXBLOCKPERLEVEL) are used to determine the number of basic blocks in each level. For level 2, the number is 3; therefore, 3 basic blocks: 2, 3, and 4 are assigned. The random integer for level 3 is 1. The basic block 5 is, then, assigned to level 3. However, only one basic block is left to be added. If an extra level is constructed, the control flow will have two adjacent levels which have one basic block in each as illustrated in Figure IV.1(a). Instead, the extra basic block is added to the level 3 as demonstrated in Figure IV.1(b). The final lay out of the control flow is shown in Figure IV.1(c).

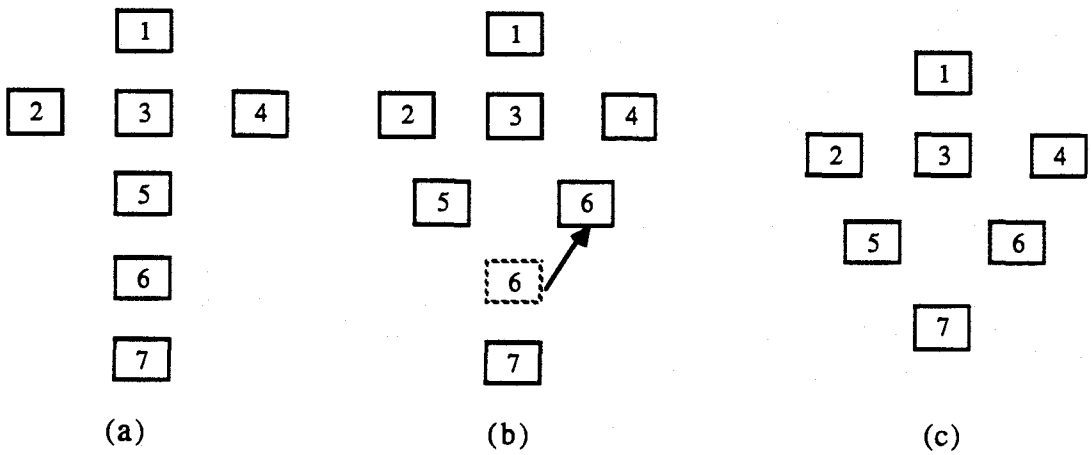


Figure IV.1 An example of basic blocks Lay out generation

After the basic blocks layout of the control flow is finished, we start to generate paths between basic block. Because #MAXBRANCH is 2, for each basic block, 2 random integers are picked for determining its destinations. Start with level 1, from the basic blocks layout in figure 1, random integers from 2 to 4 are picked twice; both

are 2. The path from basic block 1 to basic block 2 is connected as in figure 2. Basic blocks 3 and 4 do not yet have predecessors, so, since level 1 has only 1 basic block, the paths from basic block 1 to basic blocks 3 and 4 are added. Similarly, random integers between 5 to 6 are picked for basic block 2, basic block 3 and basic block 4; these integers are 5, 5, 5, 5, 6 and 6, respectively. The paths corresponding to these numbers are connected. With these connections, basic block 6 has one predecessor: basic block 4, and basic block 4 also has only one successor: basic block 6. A random integer from 5 to 6 is picked again to be used as another successor of basic block 4; the number is 5 (we keep picking the number until it is not 6). Finally, the paths from basic block 5 and basic block 6 to basic block 7 are connected. The process of generating the control is now complete as shown in Figure IV.2(i).

3.2. Generation of the pseudo-microoperations

During the control flow structure generation, all basic blocks are created empty. After that process ends, pseudo-microoperations are generated and placed in each basic block. No basic block is left empty when generation of the pseudo-microoperations finishes. The number of microoperations generated for each basic block is randomly chosen from [2, #MAXMO]. If a basic block has more than one branches leading from it, a conditional-jump pseudo-microoperation is add at the end of the basic block.

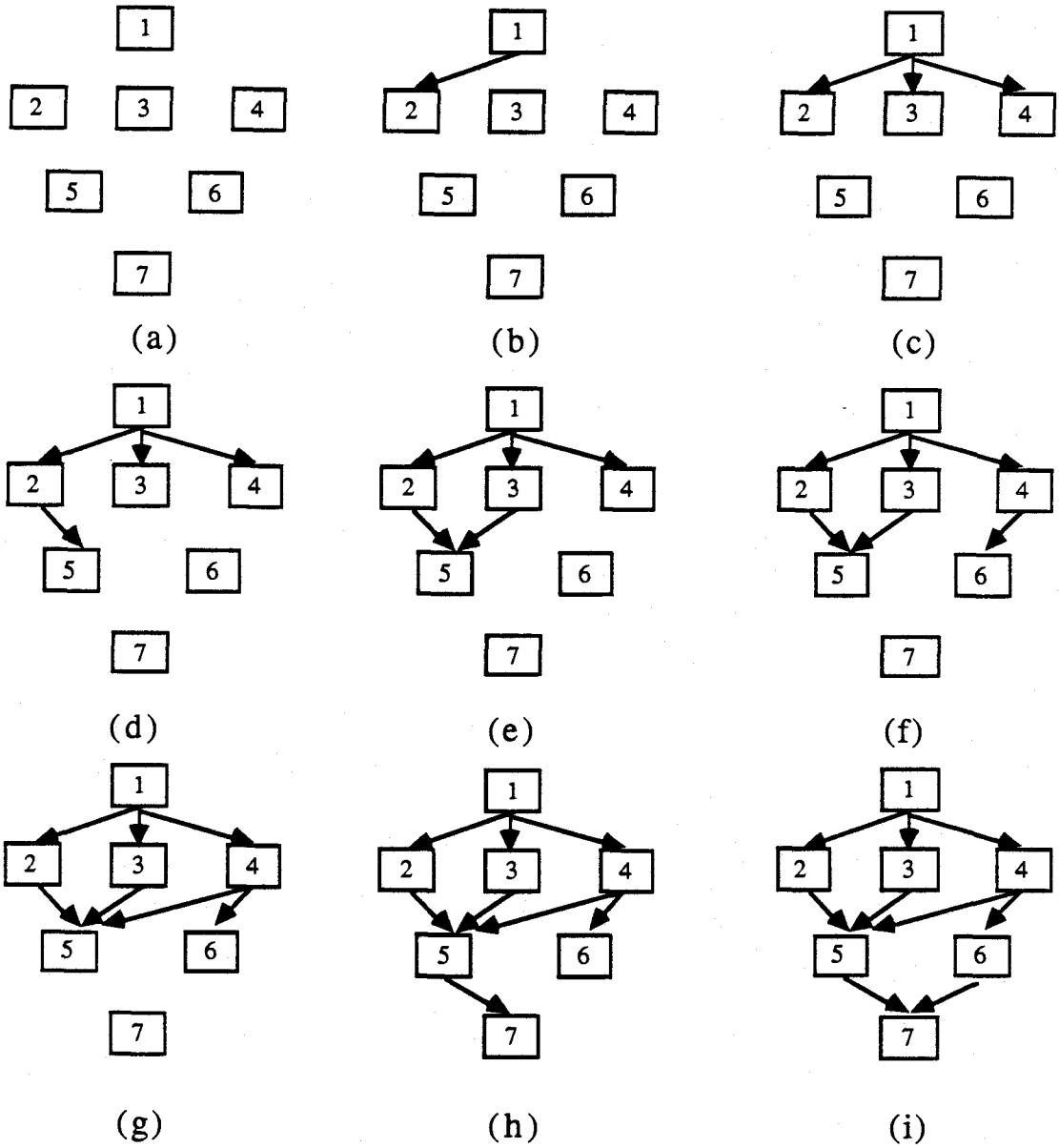


Figure IV.2. An example of control flow generation

The process of pseudo-microoperation generation used here is similar to one described in [FISH79]. There are 5 parameters used in generating a pseudo-microoperation; they are:

#RES	:	the number of operators (resources) available,
#RESU[4]	:	the number of operators used by each pseudo-microoperation (i.e. the number of operators in Uset),
#REGS	:	the number of registers available,
#REGSR	:	the number of registers read per pseudo-microoperation (i.e. the number of registers in Iset), and
#REGSW	:	the number of registers written per pseudo-microoperation (i.e. the number of registers in Oset).

For each of the pseudo-microoperations, its Uset is constructed by picking #RESU integers from uniformly distributed integers in the range [1,#RES]. The pseudo-microoperation uses the one available unit of each operator picked. Repeated selection of the same operators is possible, so some pseudo-microoperations possibly use fewer than others when #RESU is greater than one. A similar process is

4. All conditional-jump pseudo-microoperation use a unique operator which is not use by other pseudo-microoperations. This is accomplished by assigning the #RESU+1 operator to every conditional-jump pseudo-microoperation. Hence, a conditional-jump pseudo-microoperation is always conflict-free with any non-conditional-jump pseudo-microoperations.

also used for determining the Iset and Oset of each pseudo-micro-operation, so each may read from 1 to #REGSR registers and write from 1 to #REGSW.

Values for each parameter used by Fisher[5]

#RES	=	4
#RESU	=	1
#REGS	=	6
#REGSR	=	2
#REGSW	=	1

Values used in the simulation

#RES	=	10
#RESU	=	1
#REGS	=	30
#REGSR	=	2
#REGSW	=	1
#MAXMO	=	5

These values are used because we want our PMPs to have wide PMIs (i.e. PMIs can contain several PMOs).

Each PMO has a unique name. If two PMOs have the same Iset, Oset and Uset, they are identical pseudo-microoperations. If PMO_I precedes PMO_J in the vertical PMP, PMO_J is data dependent on PMO_I if at least one the followings is true:

5. Values used in [FISH79] which he felt to represent the PUMA.

1) The Iset of PMO_J and the Oset of PMO_I are not disjoint; or

2) The Iset of PMO_I and the Oset of PMO_J are not disjoint; or

3) The Oset of PMO_I and the Oset of PMO_J are not disjoint; or

4) There exists a pseudo-microoperation PMO_K such that PMO_K is data dependent on PMO_I and PMO_J is data dependent on PMO_K.

This modeling does not make any distinction between data dependency (i.e. condition 1) and data antidependency (i.e. conditions 2 and 3). Since most compactions (both local and global) do not treat them differently, in order to allow comparison of our algorithms with others, the different nature of data antidependency is deliberately ignored.

Two pseudo-microoperations are conflict-free if their Usets are disjoint. A pseudo-microinstruction, or PMI, is a set of PMOs that are conflict-free and have no dependency among them. If PMO_J is data dependent on PMO_I, PMO_I must be placed in a PMI "above" the PMI in which PMO_J is placed.

4. Compaction

In Part II, a microprogram global compaction (KLITS) algorithm, based on an adaptation of the Kernighan-Lin iterative method, was shown. This KLITS algorithm is used to compact a number of pseudo-microprograms in our experiments. The algorithm moves only those microoperations that can be moved without affecting either the semantics or the topological structure (control flow pattern) of the microprogram.

The conditions required for moving a microoperation across basic block boundaries without changing the semantics of the microprogram are called the moving rules (for more discussion on moving rules see Part II).

The purpose of any global compaction is to improve microprogram compaction further from the result that can be achieved by local compaction of each basic block. This by no mean implies that a global compaction needs to start with a local compaction. In our simulation, a local compaction is performed at the first step of algorithm. This is not a necessary step for the KLITS compaction. The local compaction is done to provide a comparison with global compaction.

4.1. The cost function

In each experiment, the total number of (pseudo)microinstructions in a (pseudo)microprogram is used as the cost measure. The actual cost of each microoperation movement is calculated by recompacting the basic block from which the microoperation is moved and the basic block(s) to which it is moved. Then, the minimum actual cost that can be achieved by moving a pseudo-microoperation across basic block boundaries is used for that pseudo-microoperation representative cost. The local compaction technique called list scheduling [FISH79] is employed for compacting and recompacting basic blocks because

1) it has been used by several global compaction techniques.
2) It is easy to implement.
3) When the number of microoperations in the basic block is small, list scheduling produces very good results.

In list scheduling local compaction, each microoperation in the basic block is assigned a priority. Then, a set of microoperations which have no predecessors except those that were already placed in microinstructions is constructed; this set is called the data ready set or D—set. The D-set is initialized with all microoperations that do not have any predecessors. A microinstruction is formed by grouping microoperations from the D-set according to their priority. The microoperation with the highest priority is taken out from the D-set

and placed first in the empty microinstruction. The microoperations remaining in the D—set are examined in order of their priority. Each is taken out from the D—set and added into the newly formed microinstruction if it does not conflict with those already there. If such a conflict does exist, the microoperation is not selected. The process is repeated until either no more microoperations can be added into the forming microinstruction or the D—set is exhausted. After the new microinstruction is completed, the D-set is updated (since some microoperations are placed in the microinstruction, their successors may become data ready). The compaction terminates when all microoperations in the basic block are placed in microinstructions.

A basic block may have a conditional-jump or unconditional-jump microoperation, but by the definition of a basic block, there can not be any microoperation in the same basic block after a jump microoperation. A special consideration is made when compacting a basic block with a jump microoperation: a jump microoperation is given the lowest priority (0) and can only be put in a microinstruction which has no successor microoperations. Although, in the real microprogram, the microinstruction containing a conditional-jump microoperation must be the last microinstruction of the basic block, in our implementation, it is possible that after the conditional-jump microoperation was placed in the microinstruction the D-set may not be empty, which means that there might be microinstructions following the one that has the conditional-jump microoperation. However, these microinstructions do not have any data dependency rela-

tion with the microinstruction containing the conditional-jump micro-operation.

Thus in an actual microprogram the remaining microinstructions would be placed immediately before the jump, but in the simulation that rearrangement is unnecessary.

Fisher experimented with strategies for assigning priorities. He reports that the performance of the strategies which are related to the height of microoperation in the successor graph are very good compared with other heuristics tested. The height of a microoperation is defined as follow:

> "(a) If a task [microoperation] has no successors, its height is 1.
> (b) Otherwise, find the successors of the task whose height is the largest, say height h. If one of those tasks is a strict successor[6], then the given task has height h+1. Otherwise, the given task has height h." [FISH79].

In our simulation, the simplest strategy is utilized. That is, the height of the microoperation is used as its priority. In the case that two PMO's have the same height, the numbers of their successors are used as the tie-breaker: the one which has more successors is given the higher priority.

---

6 . The strict successor is the successor that must be executed after the microoperation (not at the same time). In these experiments, all successors are strict successors.

4.2. The moving rules

The moving rules are used to decide if the microoperation can be moved beyond its current basic block and also govern the changing of the microprogram content after such a movement is performed if it can be performed at all. The cost function is then used to determine the changing of the microprogram cost due to the microoperation movement.

The moving rules that are discussed in Part II are used in the implementation of the KLITS. For more informations see Section 4.1 in Chapter III.

4.3. The algorithm (global compaction of the PMP)

After the pseudo-microprogram is generated, we perform a local compaction (using the list scheduling technique [FISH79]) on every basic block; the number of total pseudo-microinstructions is used as the current cost of the PMP.

All pseudo-microoperations are initially marked as unselected. Then, each PMO is checked for its movability according to the moving rules.

If a PMO is movable, its cost is determined by calculating the effect of its movement on the total number of PMIs. In our implementation, this calculation is done by recompacting both the basic

blocks to which the PMO is moved and the basic blocks from which the PMO is moved. The best (i.e. the smallest) change of the PMP cost (i.e. the number of total pseudo-microinstructions) that can be achieved by moving the PMO out of its current resident basic block, even if that cost may be worse than the current one, is used as the representative cost of that PMO. This representative cost can be positive (if the movement increases the PMP cost), or negative (if the movement decreases the PMP cost), or zero (if the movement does not change the PMP cost at all). In the case that a PMO can be moved in both directions (i.e. up and down), and the minimum representative costs of both direction are the same, then, the up direction is (arbitrary) chosen. Its representative cost and the direction of movement (i.e. up or down) are recorded for each PMO.

After the representative costs and movability of all PMOs are determined, the PMOs which are movable are put, in order according to their representative costs, into the ready to move sequence or R-sequence, and marked as selected. The PMO with the best (minimum) representative cost is the first in the R-sequence. If the representative costs of pseudo-microoperations are the same, their order of appearance in the original vertical pseudo-microprogram (i.e. their names) is (arbitrary) used as a tie-breaker.

The first PMO (i.e. the one with minimum representative cost) in the R-sequence is chosen and taken out from the R-sequence. The PMO is moved in the same direction which was recorded earlier. If the PMO can be moved in that direction, it must be moved to the

basic block (or basic blocks) which produces the minimum cost. This basic block (or basic blocks) may not be the same as the one that produced the representative cost which has been calculated earlier. Similarly, the cost of the PMO movement may not be the same as its representative cost. If there are several places which produce the same cost, the PMO is moved to the basic block (or group of basic blocks) that is farthest from its previous resident basic block. The live-at-the-top register sets are updated as follows:

1) If a PMO is moved down out of a basic block, the live-at-the-top registers set of that basic block is unchanged.

2) If a PMO is moved up out of a basic block, the live-at-the-top registers set of that basic block needs to be recalculated. This process is not complicated since not all live-at-the-top registers sets of the basic block successors are affected. We can simply union all of those sets, delete all registers that become "dead" because PMOs in the basic block write to them and add registers that become live because PMOs in the basic block read them.

3) If a PMO is moved up into of a basic block, the live-at-the-top registers set of that basic block is unchanged.

4) If a PMO is moved down into of a basic block, the live-at-the-top registers set of that basic block needs to be recalculated. The process is similar to 2).

5) Any time that the live-at-the-top registers set of a basic block is changed (i.e. when it is recalculated and the result is not the same as before), all live-at-the-top registers sets of that basic block's predecessors have to be recalculated.

If a PMO cannot be moved in the predetermined direction, it is not moved at all. It is still taken out of the R-sequence, but it is marked unselected.

When a PMO is moved, the PMP is changed. This change usually changes the cost of the PMP. The new cost of the PMP after the PMO movement is recorded. In order to save time, the previously calculated values of representative costs for the remaining R—sequence are not reevaluated. Rather, the values from before the first MO movement are kept as estimates. Then, all unselected PMOs are checked again; if any PMO becomes movable, it is put into the R-sequence.

When the R-sequence is empty (i.e. no movable unselected PMO is left), the sequence of PMO movements is examined. The minimum cost of the sequence is located; the PMO that produced the minimum cost is marked. In the event that more than one location in the sequence yield the same minimum cost, the latest PMO movement that gives the minimum cost is marked. Then, starting from the last PMO to the PMO that is moved immediately after the marked PMO in the sequence, the PMOs are moved back one at a time until

the PMP's cost is the same as the minimum cost. At this point, we say that another pass of the algorithm is completed. If the minimum cost in the sequence is worse than the starting cost that existed before the first PMO movement of the sequence, the PMP is reversed all the way back to the state of the starting cost and the compaction is now completed. Otherwise, a new pass is begun by marking all PMOs unselected; and the compaction continues. To prevent the algorithm from running perpetually, if, in two consecutive passes, all PMO movements are zero-cost movements, the algorithm is terminated.

5. Experiments on previous examples

5.1. Tree compaction example [LAH83]

This example is described in a paper on "Tree Compaction" [LAH83]. The microprogram has 21 microoperations and 7 basic blocks. When local compaction (list scheduling) is applied to the example, the compaction result is 15 microinstructions. The comparison summary is shown below.

Comparison Summary

	Time in Trace (MIs In Trace)	Weighted Time	Space needed (Total MIs)	Space-time product
Tree Compaction	8	8.4	13	109.2
Trace Scheduling	7	7.52	21	157.92
KLITS Compaction	7	7.8	12	90.36

In this example, for each basic block containing a conditional-jump microoperation, the probabilities of execution of each branch are given. So, the probability of each basic block to be executed can be calculated. Using these probabilities, the major trace (i.e. the trace which has the highest probability of execution) can be picked.

The time in trace is the number of microinstructions in the major trace after the compaction. The weighted time is the summation of the product of the number of microinstructions in the basic block and the probability that the basic block is executed, for every basic block in the microprogram. This should be the same as the average execution time of the compacted microprogram, assuming each microinstruction has the same execution time. The space needed is the total number of microinstructions in the compacted microprogram. And, the space-time product is the product of the weighted time and the space needed.

The KLITS compaction shows improvement over the tree compaction in all categories. The trace scheduling shows the same number of microinstructions in the major trace and 3.58% improvement in weighted time over the KLITS compaction. However, the extra space used to make that 3.58% improvement is a 75% increase. The space-time produce of the trace scheduling is 74.76% bigger than the KLITS result.

5.2. Rao's example [RAO86]

Rao and Majumdar describe an algorithm for global compaction. They demonstrated their algorithm on an example in [RAO86]. The example consists of 37 microoperations and 9 basic blocks. The list scheduling local compaction yields 32 microinstructions.

Comparison Summary[7]

	Rao's algorithm[8]	Trace scheduling	Tree compaction[9]	KLITS compaction
Total MI	23	23	28	23

The KLITS compaction result shows an improvement over tree compaction. It also produce the same number of microinstructions as Rao's algorithm and trace scheduling.

6. Experiments on pseudo-microprograms

The KLITS algorithm and a pseudo-microprograms generator have been implemented in Turbo Pascal Version 3.0[10]. All of the experimented were run on an IBM AT computer[11] with 4.77 MHz clock speed and 640 Kb of memory under the PC DOS 3.2 operating system.

---

7. In their paper, Rao and Majumdar did not report any detail about other compactions (i.e. trace scheduling and tree compaction) except their final results.
8. In their paper, Rao and Majumdar did not report any detail about other compactions (i.e. trace scheduling and tree comaction) except their final result.
9. This result is similar to that produced by GDDG (Generalized Data Dependency graph [ISOD83]) compaction [RAO86].
10. Turbo Pascal is a registered trademark of Borland International, Inc.
11. IBM AT is a registered trademark of International Business Machines Corp.

For each set of values of #MAXBLOCKPERLEVEL and #MAXBRANCH, the value of #BLOCK had a range of values. For each value of #BLOCK, 4 control structures of pseudo-microprograms were generated. And, for each control structure 15 pseudo-microprograms were generated and compacted by KLITS. So, the total of 60 pseudo-microprograms were presented for each set of #MAXBLOCKPERLEVEL, #MAXBRANCH, and #BLOCK. The values which were used for #MAXBLOCKPERLEVEL, #MAXBRANCH, and #BLOCK are:

#MAXBLOCKPERLEVEL	#MAXBRANCH	#BLOCK
2	2	10, 13, 16, 20, 25
4	2	10, 13, 16, 20, 25
6	3	13, 16, 20

6.1. Experimental experience

For each number of basic blocks: 13,16 and 20; there are 180 experiments (i.e. 180 PMPs tested). For 10 and 25 basic blocks, there are 120 experiments each. Hence, the total number of experiments are 780.

For each experiment, a PMP was generated by the process discussed before. All the random numbers used in the process were recorded, so that the same PMP can be reconstructed if necessary.

These data were recorded: the number of PMOs, the number of PMIs produced by local compaction, the number of PMIs produced by the KLITS algorithm, the difference between them, the time used for the experiment, the total number of passes, the number of passes wherein all movements were zero-cost movements, the total number of movements, the average number of conditional-jump PMOs per basic block, the average number of out-going branches per basic block, and the total number of conditional-jump PMOs.

Because, in our implementation of the KLITS algorithm, dynamic variables were used to represent PMOs and basic blocks and these variable were used very often, sometimes it was not possible to allocate them, either because there was not enough free memory or not a big enough contiguous free block of memory. For this reason, some of the experiments stopped before the compaction finished. The number of occurrences are showed below.

	10	13	16	20	25
B2L2[12]	None	None	None	3 stops Out of 60	2 stops out of 60
B4L2	None	None	2 Stops Out of 60	3 Stops Out of 60	4 stops out of 60
B6L3	*****	None	5 Stops Out of 60	7 Stops Out of 60	*****

12. B*n*L*m* means the parameters of the basic blocks structure are #MAXBLOCKPERLEVEL = *n* and #MAXBRANCH = *m*.

6.2. Statistical significance of the experiments' results

6.2.1 Relationship between the population mean (μ) and the experiment mean ($\bar{X}$)

The probability relation between the population mean and the experiment mean can be expressed as follow:

$$P\{\mu-\delta \leq \bar{X} \leq \mu+\delta\}=1-\alpha$$

where μ is the true population mean , $\bar{X}$ is the experiment mean, and $1-\alpha$ is the probability that the interval $\mu\pm\delta$ contains $\bar{X}$. If we can assume that population is normally distributed, it has been shown that $\delta = (\sigma Z_{\alpha/2})/\sqrt{n}$ where n is the experiment size, σ is the population standard deviation, and $Z_{\alpha/2}$ is the two-tailed standardized normal statistic [SHAN75].

Assuming that the population of pseudo-microprograms is normally distributed on each set of #MAXBLOCKPERLEVEL, #MAXBRANCH, #BLOCK; for α=0.05 (i.e. the probability that the interval $\mu\pm\delta$ contains $\bar{X}$ is 0.95) and n=60 (i.e. our number of pseudo-microprograms), then

$$\delta = 0.253034\sigma = \sigma/3.952023 \approx \sigma/4.$$

6.2.2 Relationship between the population variance (σ^2) and the experiment variance (s^2)

The relation between σ^2 and s^2 can be demonstrated by the following probability statement below:

$$P\{(1-\delta)\sigma^2 \leq s^2 \leq (1+\delta)\sigma^2\}=1-\alpha$$

where δ is a number between 0 and 1 that expresses how close the S^2 are to be to the σ^2. If the experiment size, n is large enough, then

$$\delta = Z_{\alpha/2}\sqrt{\frac{2}{n-1}}$$ [SHAN75].

For $\alpha=0.05$ and n=60, we can estimate that the experiment variance (s^2) is within $\pm 36.0865\%$ of the population variance (σ^2) with probability of 0.95.

6.2.3 Statistical significance summary

The number of experiments performed for each set of #MAXBLOCKPERLEVEL, #MAXBRANCH, and #BLOCK are enough so that the experiment average is within the population mean $\pm$ one quarter of the population standard deviation. The values of the experiments statistical data and the estimated ranges of population means and standard diviations are shown in appendix A.

6.3. Discussion

6.3.1 Improved Compaction

In every experiment, before the KLITS algorithm is applied to the pseudo-microprogram, list scheduling local compaction is used to compact each basic block separately. The total number of pseudo-microinstructions produced by the local compaction of each experiment is recorded and compared to the result obtained from the KLITS global compaction. The KLITS technique can almost always (776 cases out of 780 cases or 99.487%) improve microprogram compaction from local compaction. The number of PMIs per PMP that are reduced by the KLITS algorithm over the local compaction range from 0 (4 cases out of 780 cases) to 14 (1 case out of 780 cases).

The frequencies of number of reduced PMIs, categorized by number of basic blocks, are shown in Figure IV.3. Since the number of experiments in each number of basic blocks are not the same, the frequencies are presented as a percentage of the total number of PMPs tested for that number of basic blocks.

From the graph in Figure IV.3, when the number of basic blocks increases, the peak of each subgraph shifts to the right. Therefore, when the PMP becomes bigger, we can expect that the KLITS compaction algorithm reduces more PMIs from the local-compacted form.

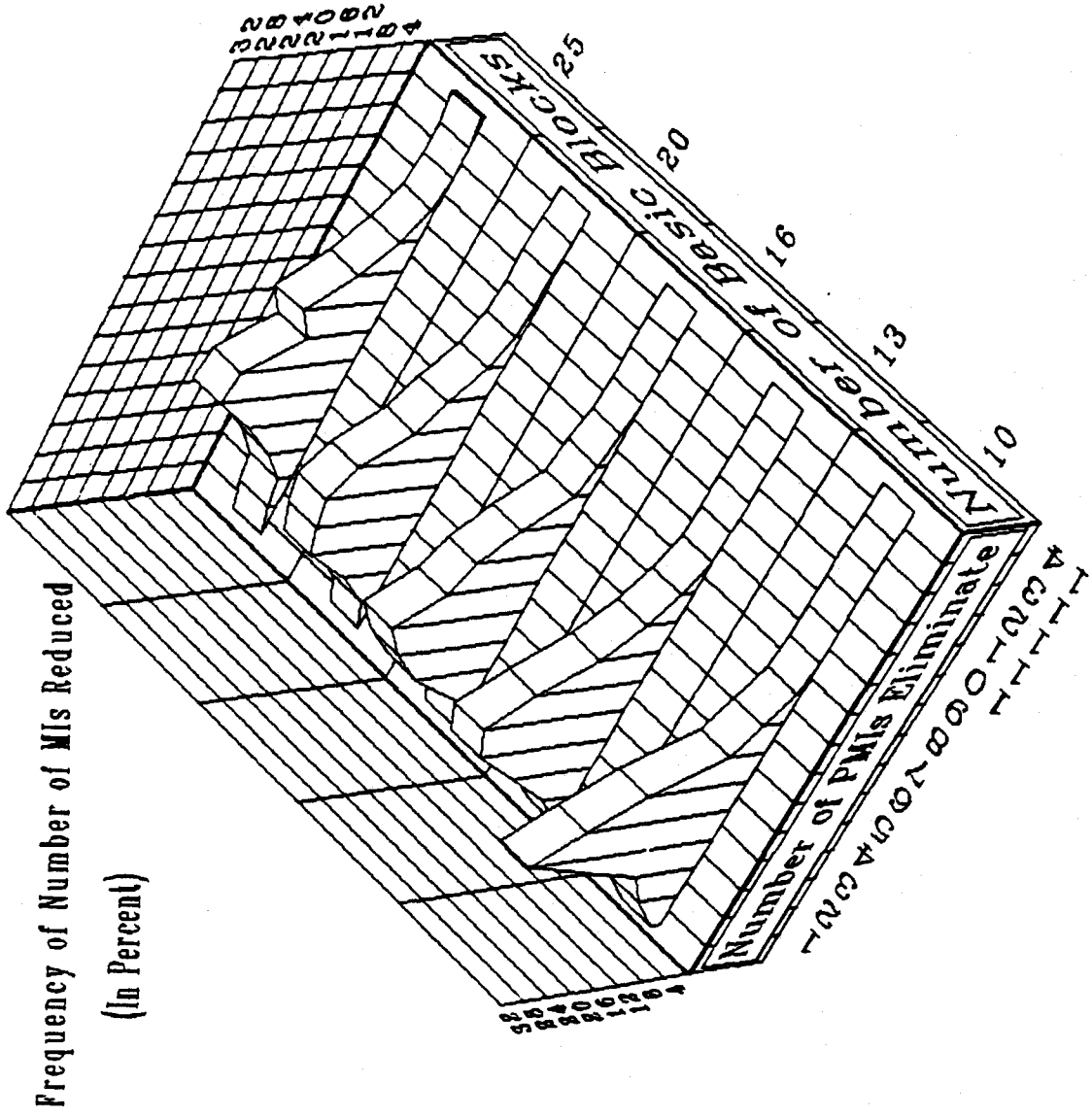


Figure IV.3. Number of PMIs eliminated by KLITS over local compaction

In our experiments, about 20 - 70 percent (average of 47.5%) of the basic blocks in each pseudo-microprogram generated contain conditional-jump pseudo-microoperations. Since the KLITS algorithm preserves the basic block structure, these conditional-jump PMOs are not moved beyond their basic block boundaries. These numbers of conditional-jump PMOs is a lower bound for the number of pseudo-microinstructions in the compacted PMP. The PMIs which do not contain conditional-jump PMOs are candidates for elimination.

The improvement of the KLITS algorithm over local compaction is also shown as the percentage of the number of total PMIs that can possibly be reduced (i.e. the number of PMIs produced by local compaction minus the number of conditional-jump PMOs). The frequencies of percentage of PMIs reduced, catego-rized by number of basic blocks, are shown in Figure IV.4. Again, due to the different of number of experiments performed on each basic blocks number, the frequencies are graphed as a percentage of the number of experiments performed for that number of basic blocks.

From the graph in Figure IV.4, except for the PMPs which have 10 basic blocks, for each number of basic blocks, most of the PMPs were improved over local compaction by the KLITS algorithm by removing 15-20 % of the elimination candidates.

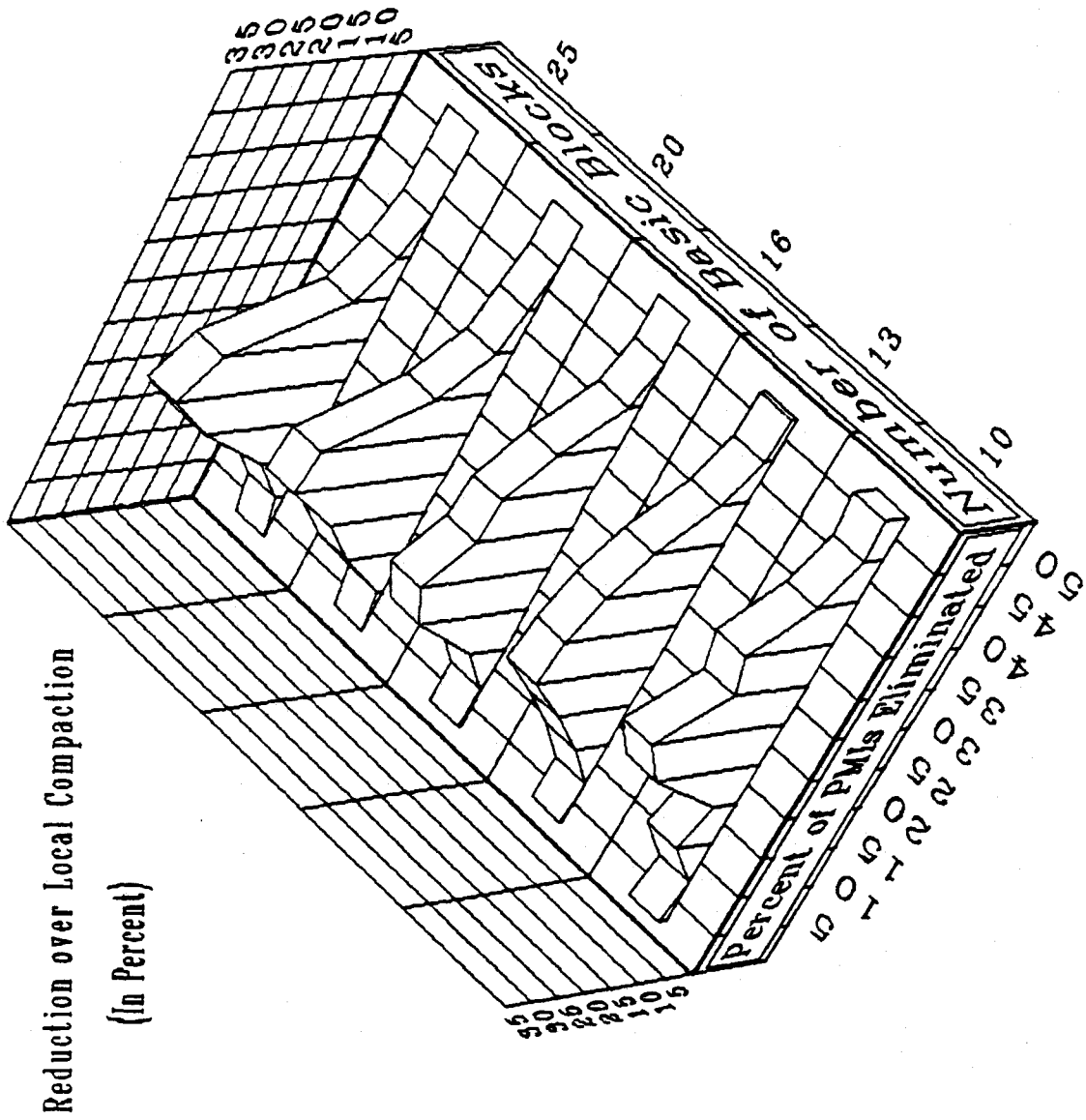


Figure IV.4. Percentage of PMIs eliminated by KLITS over local compaction

6.3.2 Timing

For each experiment, the time that it started and the time that it ended were marked and the difference between them was used as the compaction time for that experiment. The compaction time of each experiment (780 experiments) plotted against the number of PMOs is shown in Figure IV.5(a). The average compaction time of each number of PMOs is shown in Figure IV.5(b).

There is a special case that occurred during our experiments. The compaction time for on PMP with 63 PMOs was very long (125019 seconds or approximately 34.727 hours). From the other data form this experiment, we speculate that the reason for this extra long compaction time might be that the movement of PMO_A produced a positive cost: **x**, then later PMO_B movement produced a negative cost: **-x**, when that pass was ended, the revised moving sequence ended at PMO_B. In the next pass, PMO_B was moved with cost **x**; then PMO_A was moved with the cost of **-x**. The total cost of this pass was 0. This situation repeated again and again since the total cost at the end of each pass was not worse than the cost that it started with. When a PMO was moved from a basic block to another basic block, we did not actually "move" it. Rather, we marked the PMO in the "from" basic block as inactive; then we created a new PMO which had attributes identical to the old one except the name (since each PMO must have a unique name). In our implementation of a PMO, the integer data type was used to represent the name of a

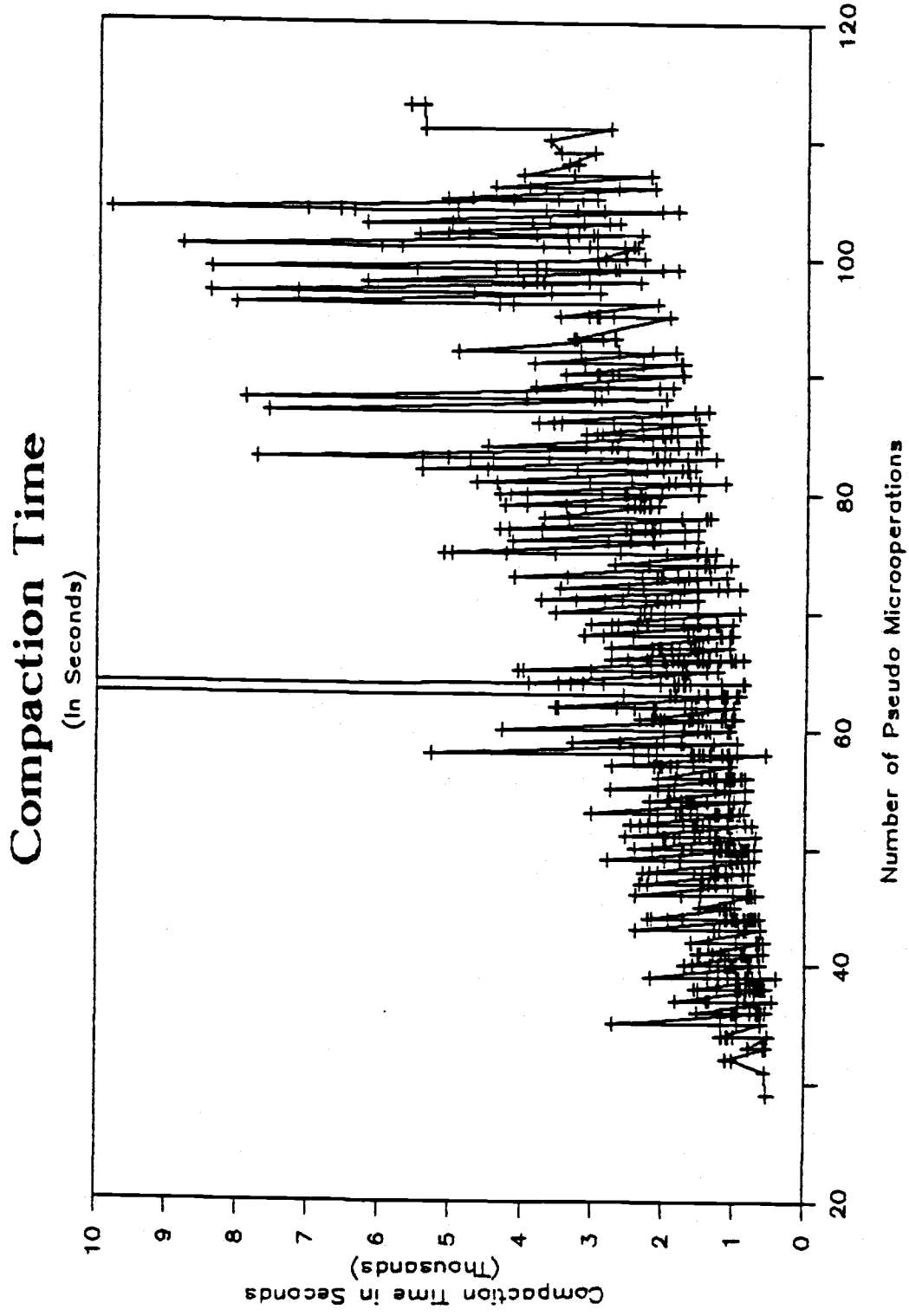


Figure IV.5. KLITS experiments compaction time

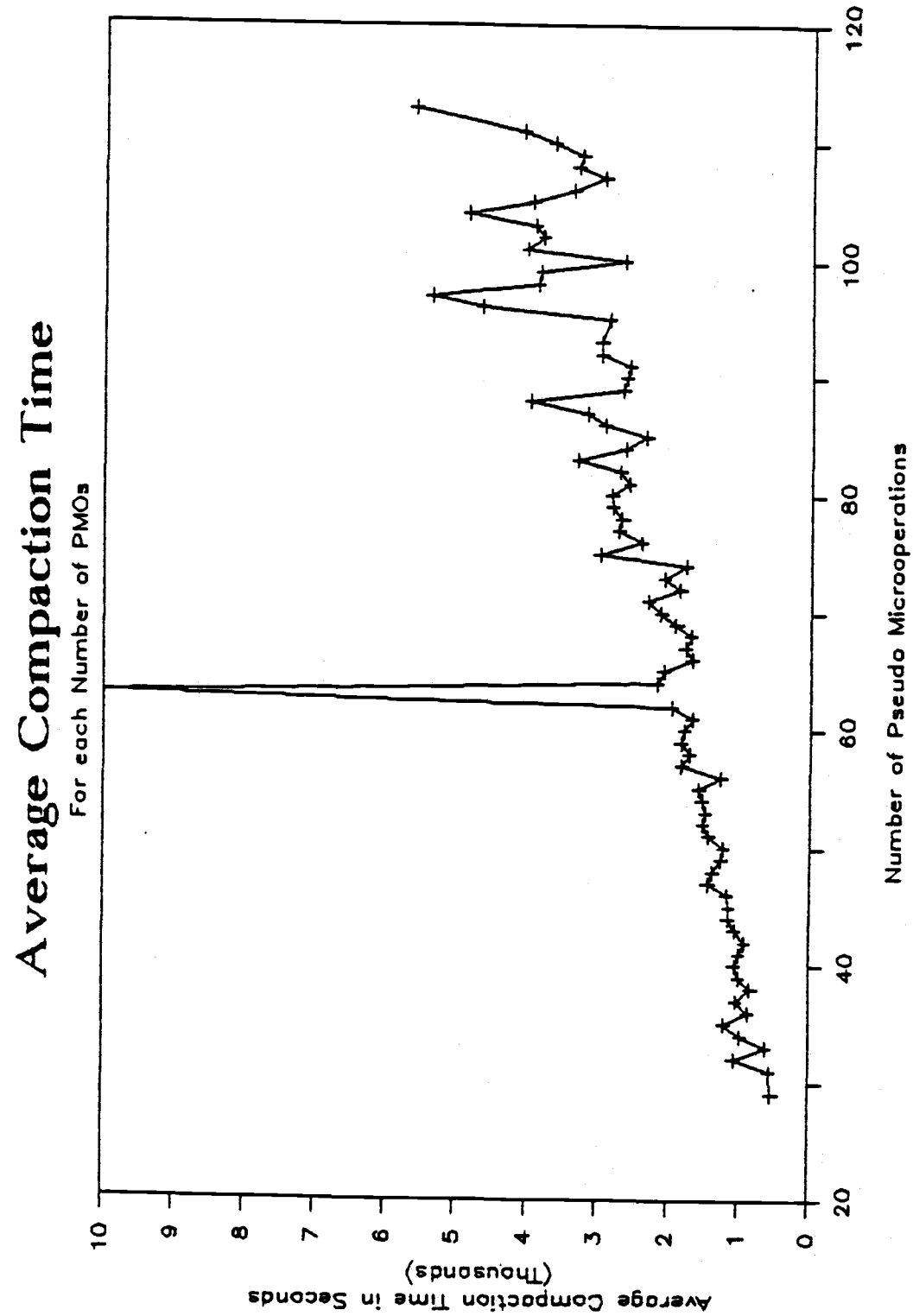


Figure IV.6. KLITS experiments average compaction time of each number of PMOs

PMO, and when a PMO was created the new name was given to it by adding one to the last "name" used. In Turbo Pascal, the largest integer is 32767; if 1 is added to this number the result will be -32768 since Turbo Pascal does not detect integer overflow. If 1 is added to this number 32769 more times, the result will become positive again. Then, we have duplicate PMO names. This condition caused the process to stop (after a very long time). A method that can prevent the wasteful long time will be discussed in the next section.

The graph in Figure IV.6. suggests that the compaction time growth may be a little faster than linear in the range below 120 PMOs and slower than an exponential function (if it is an exponential function, it must have a very small exponent).

7. Summary and conclusions

7.1. Summary

The technique used in the KLITS algorithm is similar to a technique microprogrammers use to compact microprograms by hand, moving microoperations "when such motion appears to improve compaction" [FISH81]. In contrast, the KLITS algorithm allows movements that may initially lengthen a microprogram, if the overall sequence of movements will improve the compaction.

Unlike trace scheduling, the KLITS algorithm compacts a microprogram without altering the topology of the original microprogram. The KLITS algorithm can also accommodate other cost functions; this fact makes the algorithm useful for use in reducing any selected properties of a microprogram without changing the algorithm itself.

The KLITS algorithm has been tried on two microprograms from previously published papers. The compacted microprograms produced by the KLITS algorithm were better than the results produced by tree compaction, and as good as or better than the results produced by trace scheduling and Rao's algorithm.

To further study the behavior of the KLITS algorithm, we implemented the algorithm and tested it on model of microprograms called pseudo-microprograms. The pseudo-microprograms, that were used in our experiments, were adapted from the models that had been developed and used by Fisher [FISH79] to study his local compaction technique: list scheduling.

Pseudo-microprograms have been generated in different topological structures and sizes. They were first compacted by list scheduling local compaction and the number of PMIs were record for later comparison. Then they were compacted by the KLITS algorithm with the total number of PMIs as the cost function. The most obvious method for calculating the cost: recompacting the basic blocks from which microoperations were moved and the basic blocks to which they were moved, was chosen for our experiments. The

total number of PMIs after KLITS compaction were compared with the total number of PMIs after list scheduling local compaction. The results from the experiments showed that the KLITS algorithm almost always (99.487% of cases) improves upon local compaction and the number of (pseudo) microinstructions reduced by the KLITS algorithm increases when the size of (pseudo) microprograms increase.

Another aspect of the KLITS algorithm that we are interested in is the time the algorithm needs for compacting a microprogram. From these values, we believe that the compaction time of the KLITS algorithm grows not much faster than linear with the size of the microprogram in the range of sizes tested; if it is exponential, the coefficient is small.

7.2. Future work

The moving rules that are used in KLITS are the conditions for moving microoperations across basic block boundaries without altering the topological structure nor changing the semantics of the original microprogram. However, it would be possible to add moving rules to allow moving a microoperation while preserving the semantics of the microprogram, and causing only a minor change of the topological structure of the microprogram. An example of this type of movement is moving a conditional-jump microoperation up.

A conditional-jump microoperation can be moved up when it is the only microoperation left in the basic block and the other conditions are the same as moving an ordinary microoperation up. After the conditional-jump MO is moved up, the basic block is now empty (since the conditional-jump MO must be the only MO in the basic block); the basic block is deleted from the microprogram. Each predecessor basic block now has a conditional-jump MO which jumps to the successor basic blocks of the deleted basic block. Hence, the topological structure of the microprogram is slightly changed. If the conditional-jump MO (and its copies) can be put into an already-existing MI of each of its new resident basic blocks, the number of MIs is reduced. Hence, the cost of the microprogram decreases.

This type of movement may improve the compaction and, although it alters the topological structure of the microprogram, the users have the ability to control the change of the topological structure by altering the cost evaluation, and to decide if the movement is worthwhile.

In our experiments, our cost of a (pseudo) microprogram is the number of total number of microinstructions. This cost (i.e. the number of microinstructions) is calculated by using list scheduling to compact (or recompact) basic blocks. When a microoperation is moved across basic block boundaries, both the basic block(s) from which the MO is moved and the basic block(s) to which it is moved are recompacted. This recompaction of several basic blocks, however, may not be necessary. If a heuristic function which can

estimate the cost of the microprogram (i.e. the total number of micro-instructions) can be derived, the compaction may work faster.

Although recompaction of basic blocks gives the "actual" cost of the microprogram, this cost does not reflect the interaction of its movement with movement of other microoperations (e.g. preventing some microoperations from moving). And, several microoperations may have the same cost by this method. A cost function that takes into consideration the effect of the microoperation movement on the movement of other microoperations may improve the compaction. Also, a method for breaking a tie is needed.

The KLITS algorithm is not limited to any particular cost function. For any properties of the microprogram are that of interest, the users can select appropriate cost functions to achieve their goals. Some cost functions other than the one that was used in our experiments, should also be tested. Also, several cost functions can be combined to form a cost function that measures several properties at the same time. For example, a cost function that measures the total number of microinstructions can be combined with a cost function that measures the number of microinstructions in the major trace so that both properties are monitored. Weighted values may be used to designate one measure for the primary goal and another measure for a secondary goal.

In the previous section, a special case of extra long compaction time was discussed. This long compaction was cause by repeated

occurrence of total-zero-cost passes. One of the method that can prevent this problem is limiting the number of consecutive total-zero-cost passes. Another method is limiting the total number of passes. We can estimate the lower bound of the number of microinstructions, **l**, as the total number of microoperations, **n**, divided by the maximum number of microoperations per microinstruction (the real minimum is much higher than this value). If each pass of the KLITS algorithm reduces at least one microinstruction, to reduce the microprogram down to **l** microinstructions, the algorithm should use no more than **n-l** passes. So, we can limit the number of passes to a number on the order of **n**.

In our experiments, as was discussed before, some experiments stopped before the compaction finished. To eliminate this problem, we may change our implementation of the algorithm to actually move microoperations from basic blocks to basic blocks, not just change the status of "from" microoperations to inactive and then keep creating new microoperations to add them into "to" basic blocks. The implementation would reduce the need for memory allocation since much less new PMOs would be created.

7.3. Conclusions

The KLITS algorithm developed in Part II was implemented and tested on example microprograms from previously published papers and on model of microprograms, called pseudo-microprograms, developed in this paper. The results of the KLITS algorithm

on the example microprograms were at least as good as the results from previous global compaction techniques tested in those papers.

The results from applying the KLITS algorithm on pseudo-microprograms shows that the algorithm improves upon local compaction alone. As the number of basic blocks in a pseudo-microprogram increases, the pseudo-microprogram compacted by the KLITS algorithm increasingly becomes more compact than one that compacted by local compaction. From the experiments, we can expect that, on the average, the KLITS algorithm will reduce a microprogram 15-20 % better than local compaction alone.

The data from the experiments suggests that the time the algorithm used to compact a microprogram grows slowly enough for the algorithm to be used practically.

The KLITS algorithm is usable for compacting microprograms with respect to a user-chosen measure of cost without altering the algorithm itself while also retaining the original topological structure of the microprograms.

8. References

[DAVI81] S. Davidson, D. Landskov, B. D. Shriver, and P. W. Mallett;
"Some Experiments in Local Microcode Compaction for Horizontal Machines",
IEEE Transactions on Computers, C-30,7 July 1981, pp. 460-477.

[FISH79] J. A. Fisher,
"The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling With Resources", Phd. Dissertation, Department of Computer Science, New York University, New York, 1979.

[ISOD83] S. Isoda, Y. Kobayashi and T. Ishida;
"Global Compaction of Horizontal Microprograms Based on the Generalized Data Dependency Graph"
IEEE Transaction on Computers, C-32,10 October 1983, pp. 922-933.

[LAH83] J. Lah, and D. E. Atkins;
"Tree Compaction of Microprograms", *Proceedings of the 16th Annual Microprogramming Workshop*, October 11th-14th, 1983, pp. 23-33.

[LAND80] D. Landskov, S. Davidson, B. Shriver, and P. W. Mallett;
"Local Microcode Compaction Techniques", ACM Computing Surveys, Vol.12, No. 3, September 1980, pp. 261-294.

[RAO86] S. U. Rao and A. K. Majumdar;
"An Algorithm for Global Compaction of Horizontal Microprograms", *SIGMICRO TCMICRO Newsletter*, SIGMICRO: Vol 17, No. 2; TCMICRO: Vol 1, No. 2; July 1986, p.19-30.

[SHAN75] R. E. Shannon,
"Systems Simulation the Art and Science",
Prentice-Hall, Inc.; Englewood Cliffs, New Jersey, 1975.

[SU84] B. Su, S. Ding, and L. Jin;
"An Improvement of Trace Scheduling for Global Microcode Compaction",
Proceedings of the Seventeenth Annual Microprogramming Workshop, New Orleans, Louisiana, October 30th - November 2nd, 1984, pp. 78-85.

9. Appendix A.

For #BLOCK =10

B2L2	PMIs Difference	Percentage of Possible Eliminated PMIs	Compaction Time (Seconds)
Experiments Statistical Data			
Sampling Avg.	4.033	28.041	1.0367E+03
Sampling Max.	8.000	47.059	2.7100E+03
Sampling Var.	2.033	74.102	1.7751E+05
Sampling Std.	1.426	8.608	4.2131E+02
Estimated Population Statistical Data (95% Confidence Range)			
Std. Upper Bound	1.783	10.768	5.2700E+02
Std. Lower Bound	1.222	7.379	3.6116E+02
Avg. Upper Bound	4.479	30.733	1.1684E+03
Avg. Lower Bound	3.587	25.349	9.0495E+02

B4L2	PMIs Difference	Percentage of Possible Eliminated PMIs	Compaction Time (Seconds)
Sampling Avg.	3.250	22.394	9.0367E+02
Sampling Max.	6.000	46.154	2.1750E+03
Sampling Var.	2.292	102.784	1.1053E+05
Sampling Std.	1.514	10.138	3.3247E+02
Estimated Population Statistical Data (95% Confidence Range)			
Std. Upper Bound	1.894	12.681	4.1587E+02
Std. Lower Bound	1.298	8.691	2.8500E+02
Avg. Upper Bound	3.723	25.565	1.0076E+03
Avg. Lower Bound	2.777	19.224	7.9970E+02

For #BLOCK =13

B2L2	PMIs Difference	Percentage of Possible Eliminated PMIs	Compaction Time (Seconds)
Experiments Statistical Data			
Sampling Avg.	4.233	24.412	1.3827E+03
Sampling Max.	8.000	46.667	2.4580E+03
Sampling Var.	3.063	88.927	2.6099E+05
Sampling Std.	1.750	9.430	5.1087E+02
Estimated Population Statistical Data (95% Confidence Range)			
Std. Upper Bound	2.189	11.796	6.3902E+02
Std. Lower Bound	1.500	8.084	4.3793E+02
Avg. Upper Bound	4.781	27.361	1.5425E+03
Avg. Lower Bound	3.686	21.463	1.2230E+03

B4L2	PMIs Difference	Percentage of Possible Eliminated PMIs	Compaction Time (Seconds)
Experiments Statistical Data			
Sampling Avg.	4.033	23.598	1.3392E+03
Sampling Max.	8.000	50.000	2.2040E+03
Sampling Var.	2.507	75.349	1.5314E+05
Sampling Std.	1.583	8.680	3.9133E+02
Estimated Population Statistical Data (95% Confidence Range)			
Std. Upper Bound	1.981	10.858	4.8949E+02
Std. Lower Bound	1.357	7.441	3.3546E+02
Avg. Upper Bound	4.528	26.313	1.4616E+03
Avg. Lower Bound	3.538	20.884	1.2169E+03

B6L3	PMIs Difference	Percentage of Possible Eliminated PMIs	Compaction Time (Seconds)
Experiments Statistical Data			
Sampling Avg.	3.567	20.274	1.4026E+03
Sampling Max.	8.000	42.105	3.0180E+03
Sampling Var.	2.487	64.417	3.3021E+05
Sampling Std.	1.577	8.026	5.7464E+02
Estimated Population Statistical Data (95% Confidence Range)			
Std. Upper Bound	1.973	10.039	7.1879E+02
Std. Lower Bound	1.352	6.880	4.9259E+02
Avg. Upper Bound	4.060	22.784	1.5823E+03
Avg. Lower Bound	3.074	17.764	1.2229E+03

For #BLOCK =16

B2L2	PMIs Difference	Percentage of Possible Eliminated PMIs	Compaction Time (Seconds)
Experiments Statistical Data			
Sampling Avg.	5.400	24.578	1.9548E+03
Sampling Max.	9.000	39.130	4.2880E+03
Sampling Var.	2.854	44.064	5.0190E+05
Sampling Std.	1.689	6.638	7.0845E+02
Estimated Population Statistical Data (95% Confidence Range)			
Std. Upper Bound	2.113	8.303	8.8616E+02
Std. Lower Bound	1.448	5.690	6.0729E+02
Avg. Upper Bound	5.928	26.653	2.1764E+03
Avg. Lower Bound	4.872	22.502	1.7333E+03

B4L2	PMIs Difference	Percentage of Possible Eliminated PMIs	Compaction Time (Seconds)
Experiments Statistical Data			
Sampling Avg.	4.217	20.590	1.8995E+03
Sampling Max.	10.000	40.000	4.0630E+03
Sampling Var.	3.291	61.332	5.7930E+05
Sampling Std.	1.814	7.831	7.6111E+02
Estimated Population Statistical Data (95% Confidence Range)			
Std. Upper Bound	2.269	9.796	9.5204E+02
Std. Lower Bound	1.555	6.713	6.5244E+02
Avg. Upper Bound	4.784	23.039	2.1375E+03
Avg. Lower Bound	3.649	18.141	1.6614E+03

B6L3	PMIs Difference	Percentage of Possible Eliminated PMIs	Compaction Time (Seconds)
Experiments Statistical Data			
Sampling Avg.	3.450	17.009	3.7177E+03
Sampling Max.	9.000	33.333	1.2502E+05
Sampling Var.	2.591	55.135	2.5417E+08
Sampling Std.	1.610	7.425	1.5943E+04
Estimated Population Statistical Data (95% Confidence Range)			
Std. Upper Bound	2.013	9.288	1.9942E+04
Std. Lower Bound	1.380	6.365	1.3666E+04
Avg. Upper Bound	3.953	19.331	8.7031E+03
Avg. Lower Bound	2.947	14.687	-1.2678E+03

For #BLOCK =20

B2L2	PMIs Difference	Percentage of Possible Eliminated PMIs	Compaction Time (Seconds)
Experiments Statistical Data			
Sampling Avg.	5.867	20.958	2.7802E+03
Sampling Max.	12.000	38.710	7.7470E+03
Sampling Var.	5.406	56.533	1.4005E+06
Sampling Std.	2.325	7.519	1.1834E+03
Estimated Population Statistical Data (95% Confidence Range)			
Std. Upper Bound	2.908	9.405	1.4803E+03
Std. Lower Bound	1.993	6.445	1.0145E+03
Avg. Upper Bound	6.594	23.309	3.1502E+03
Avg. Lower Bound	5.140	18.607	2.4101E+03

B4L2	PMIs Difference	Percentage of Possible Eliminated PMIs	Compaction Time (Seconds)
Experiments Statistical Data			
Sampling Avg.	4.883	18.358	2.7945E+03
Sampling Max.	9.000	36.000	7.5890E+03
Sampling Var.	3.529	40.396	1.3254E+06
Sampling Std.	1.878	6.356	1.1512E+03
Estimated Population Statistical Data (95% Confidence Range)			
Std. Upper Bound	2.350	7.950	1.4400E+03
Std. Lower Bound	1.610	5.448	9.8687E+02
Avg. Upper Bound	5.471	20.345	3.1545E+03
Avg. Lower Bound	4.296	16.370	2.4345E+03

B6L3	PMIs Difference	Percentage of Possible Eliminated PMIs	Compaction Time (Seconds)
Experiments Statistical Data			
Sampling Avg.	4.150	17.266	2.3666E+03
Sampling Max.	9.000	33.333	5.4360E+03
Sampling Var.	2.401	38.540	1.0635E+06
Sampling Std.	1.549	6.208	1.0313E+03
Estimated Population Statistical Data (95% Confidence Range)			
Std. Upper Bound	1.938	7.765	1.2900E+03
Std. Lower Bound	1.328	5.322	8.8402E+02
Avg. Upper Bound	4.635	19.208	2.6890E+03
Avg. Lower Bound	3.665	15.325	2.0441E+03

For #BLOCK =25

B2L2	PMIs Difference	Percentage of Possible Eliminated PMIs	Compaction Time (Seconds)
Experiments Statistical Data			
Sampling Avg.	6.600	19.294	4.1332E+03
Sampling Max.	14.000	35.000	8.8060E+03
Sampling Var.	5.193	33.559	2.7448E+06
Sampling Std.	2.279	5.793	1.6568E+03
Estimated Population Statistical Data (95% Confidence Range)			
Std. Upper Bound	2.851	7.246	2.0723E+03
Std. Lower Bound	1.953	4.966	1.4202E+03
Avg. Upper Bound	7.313	21.106	4.6513E+03
Avg. Lower Bound	5.887	17.483	3.6151E+03

B4L2	PMIs Difference	Percentage of Possible Eliminated PMIs	Compaction Time (Seconds)
Experiments Statistical Data			
Sampling Avg.	5.450	16.084	3.4418E+03
Sampling Max.	10.000	30.769	9.8190E+03
Sampling Var.	3.777	33.379	2.3331E+06
Sampling Std.	1.943	5.777	1.5274E+03
Estimated Population Statistical Data (95% Confidence Range)			
Std. Upper Bound	2.431	7.227	1.9106E+03
Std. Lower Bound	1.666	4.953	1.3094E+03
Avg. Upper Bound	6.058	17.890	3.9194E+03
Avg. Lower Bound	4.842	14.277	2.9641E+03

Chapter V Bibliography

[AGER76] T. Agerwala,
"Microprogram Optimization: A Survey",
IEEE Transactions on Computers, C-25,10 October 1976, pp. 962-973.

[ASTO71] F. Astopas and K. Plukas;
"Method of Minimizing Computer Microprograms",
Automatic Control, 5,4, 1971, pp.10-16.

[ATKI84] R. P. Atkins,
"Improved Instruction Formation in the Exhaustive Local Microcode Compaction Algorithm",
Proceedings of the 17th Annual Microprogramming Workshop,
October 30—November 2, 1984 pp. 105-111.

[DASG76] S. Dasgupta and J. Tartar;
"The Identification of Maximal Parallelism in Straight-line Microprograms",
IEEE Transactions on Computers, C-25,10 October 1976, pp.986-991.

[DASG78] S. Dasgupta,
"Comment on the Identification of Maximal Parallelism in Straight-line Microprograms", *IEEE Transactions on Computers*, C-27,3 March 1978, pp.285-286.

[DASG79] S. Dasgupta,
"The Organization of Microprogram Stores",
ACM Computing Surveys, Vol.11, No. 1, March 1979, pp. 39-65.

[DAVI81] S. Davidson, D. Landskov, B. D. Shriver, and P. W. Mallett;
"Some Experiments in Local Microcode Compaction for Horizontal Machines",
IEEE Transactions on Computers, C-30,7 July 1981, pp. 460-477.

[DEWI76] D. J. Dewitt,
"A Machine independent Approach to the Production of Optimized Horizontal Microcode", PhD thesis, University of Michigan, June 1976.

[FISH79] J. A. Fisher,
"The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling with Resources", PhD. Dissertation, the Department of Computer Science, New York University, New York, NY, October 1979.

[FISH81] J. A. Fisher,
"Trace Scheduling: A Technique for Global Microcode Compaction",
IEEE Transactions on Computers, C-30,7 July 1981, pp. 478-490.

[FISH83] J. A. Fisher,
"Very Long Instruction Word Architectures and the ELI-512",
Proceeding of the 10th Symp. Computer Architecture, Stockholm, June 1983, pp. 140-150.

[HENR83] S. D. Henry, R. A. Mueller, and M. Andrews;
"Local and Global Microcode Compaction Using Reduction Operators",
Proceedings of the 16th Annual Microprogramming Workshop,
October 11-14, 1983, pp. 34—43. October, 1983, pp. 34 - 43.

[HOPC79] J. E. Hopcroft and J. D. Ullman,
"Introduction to Automata Theory, Language, and Computation", Addison-Eesley publishing company, Reading, Massachusetts, 1979.

[ISOD83] S. Isoda, Y. Kobayashi and T. Ishida;
"Global Compaction of Horizontal Microprograms Based on the Generalized Data Dependency Graph"
IEEE Transaction on Computers, C-32,10;
October 1983, pp. 922-933.

[KERN70] B. W. Kernighan, and S. Lin;
"An Efficient Heuristic Procedure for Partitioning Graphs",
The Bell System Technical Journal, Vol. 49, No. 2;
February 1970, pp. 291-307.

[KOHA78] Z. Kohavi,
"Switching and Finite Automata Theory",
McGraw-hill Book Company,
Second Edition,1978.

[LAH83] J. Lah, and D. E. Atkins;
"Tree Compaction of Microprograms",
Proceedings of the 16th Annual Microprogramming Workshop, October 11th-14th, 1983, pp. 23-33.

[LAND80] D. Landskov, S. Davidson, B. Shriver, and P. W. Mallett;
"Local Microcode Compaction Techniques", *ACM Computing Surveys*, Vol.12, No. 3, September 1980, pp. 261-294.

[LINN83] J. L. Linn,
"SRDAG Compaction: A Generalization of Trace Scheduling to Increase the Use of Global Context Information",
Proceedings of the Sixteenth Annual Microprogramming Workshop, October 1983, pp. 78—85.

[MUEL86] R. A. Mueller, and V. H. Allan,
"Phase-Coupled Methods in Automated Microcode Generation: Assessment and Strategy",
Proceedings of the Nineteenth Annual Hawaii International Conference on System Science, 1986, pp. 505-519.

[RAMA74] C. V. Ramamoorthy, and M. Tsuchiya;
"A High-Level Language for Horizontal Micro-programming",
IEEE Transactions on Computers, C-23,8 August 1974, pp. 791-801.

[RAO86] S. U. Rao and A. K. Majumdar;
"An Algorithm for Global Compaction of Horizontal Microprograms", *SIGMICRO TCMICRO Newsletter*, SIGMICRO: Vol 17, No. 2; TCMICRO: Vol 1, No. 2; July 1986, p.19-30.

[SHAN75] R. E. Shannon,
"Systems Simulation the Art and Science",
Prentice-Hall, Inc.; Englewood Cliffs, New Jersey, 1975.

[SU84] B. Su, S. Ding, and L. Jin;
"An Improvement of Trace Scheduling for Global Microcode Compaction",
Proceedings of the Seventeenth Annual Microprogramming Workshop, New Orleans, Louisiana, October 30th - November 2nd, 1984, pp. 78-85.

[SWAM81] M. N. S. Swamy, and K. Thulasiraman;
"Graphs, Networks, and Algorithms",
Wiley Interscience, 1981.

[TOKO78] M. Tokoro, T. Takizuka, E. Tamura, and T. Yamaura;
"A Technique of Global Optimization of Microprograms",
Proceedings of the Eleventh Annual Microprogramming Workshop, November 19-22,1978; pp 41-50.

[TOKO81] M. Tokoro, E. Tamura, and T. Takizuka;
"Optimization of Microprograms",
IEEE Transaction on Computers, C-30,7 July 1981, pp. 491-504.

[TSUC76] M. Tsuchiya, and M. J. Gonzalez;
"Toward Optimization of Horizontal Microprogram",
IEEE Transactions on Computers, C-25,10 October 1976, pp. 992-999.

[ULLM84] J. D. Ullman,
"Computational Aspects of VLSI",
Computer Science Press, Inc. Rockville, Maryland, 1984.

[VEGD82] S. R. Vegdahl,
"Local Code Generation and Compaction in Optimizing Microcode Compilers", PhD thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA.; December 1982.

[WOOD78] G. Wood,
"On the Packing of Micro-operations into Microinstruction words",
Proceedings of the Eleventh Annual Microprogramming Workshop, November 19-22, 1978; pp. 51-55.

[YAU74] S. S. Yau, A. C. Schowe, and M. Tsuchiya;
"On Storage Optimization of Horizontal Microprogramming",
Proceedings of the 7th Annual Microprogramming Workshop,
September 30—October 2, 1974, pp 98-106.