COO-3077-161

Courant Mathematics and
Computing Laboratory

U.S. Department of Energy

The Optimization of Horizontal Microcode Within and Beyond Basic Blocks:

An Application of Processor Scheduling with Resources

Joseph A. Fisher

U.S. Department of Energy Report

Prepared under Contract EY-76-C-02-3077
with the Office of Energy Research

Mathematics and Computing
October 1979

New York University

DISCLAIMER

UNCLASSIFIED

Courant Mathematics and Computing Laboratory

New York University

Mathematics and Computing

COO-3077-161

THE OPTIMIZATION OF HORIZONTAL MICROCODE WITHIN AND BEYOND BASIC BLOCKS: AN APPLICATION OF PROCESSOR SCHEDULING WITH RESOURCES

Joseph A. Fisher

October 1979

U. S. Department of Energy

Contract EY-76-C-02-3077

UNCLASSIFIED

TABLE OF CONTENTS

FIGURES, TABLES AND EXAMPLES

1. Abstract and Summary

Microprogram optimization is the rearrangement of microcode written vertically, with one operation issued per step, into legal horizontal microinstructions, in which several operations are issued each instruction cycle. The rearrangement is to be done in a way that approximately minimizes the running time of the code.

We identify this problem with the problem of processor scheduling with resource constraints. As a result of this identification, the problem of optimizing basic blocks of microcode can be seen to be np-complete; however, we are able to use approximate methods for basic blocks which have good records in other, similar, scheduling environments. We use a method of scheduling called "priority list scheduling" in which the tasks are ordered according to some evaluation function, and then schedules are found by repeated scans of the list. Several evaluation functions are shown to perform very well on large samples of various classes of random data-precedence graphs with characteristics similar to those derived from microprograms. An evaluation function we produced is sensitive to both the data-precedence graph and the resource constraints; it performed best of those tested, though the differences among the four best functions, while statistically significant, were small.

A method of spotting resource bottlenecks in the derived data-precedence graph is adapted from a lower bound suggested by Fernandez and Bussel [FERN73]. This method permits us to produce the above-mentioned "resource considerate" evaluation

function, in which tasks which contribute directly to or precede bottlenecks have their priorities raised. We were also able to greatly reduce the complexity of the calculations necessary to compute the lower bound, thus making the above strategy more practical. The lower bound is further used to bound the percentage differences between the lengths of schedules produced and the optimal.

A method is suggested for optimizing beyond basic blocks. We treat groups of basic blocks as if they were one block, encoding the information necessary to control the motion of tasks between blocks as data-precedence constraints on the conditional tasks. We are thus able to optimize long paths of code, with no back branches, using the same methods used for basic blocks. These methods are efficient (order n^2), and are capable of handling the long blocks obtained this way quite well. When loops are encountered, the contents of the loop are optimized, and then the loop is treated as a unit, with its own data-precedence constraints, permitting other tasks to move past, ahead of, or into the loop, as is appropriate. The code obtained seems as optimized as, and remarkably similar to, that obtained by hand.

2. Introduction to the Problem

Microprograms are sequences of microoperations (MOPs) which control the most fundamental resources of the computer. A MOP might, for example, control whether a register is written into during a particular clock cycle, or select which of several possible data paths might be fed into an adder. In many microprogrammable computers, the fact that several different parts of the hardware can operate simultaneously may be taken advantage of in the microprogram, and a collection of MOPs may be specified for a single microprogram cycle, rather than just one. Such microprograms are said to be horizontal (rather than vertical) and a collection of MOPs specified for a single cycle is called a microinstruction.

This is an investigation into the practicality of taking a sequential microprogram written for a horizontal machine and gathering the MOPs into microinstructions in a way that approximately optimizes the running time of the microprogram. Within basic blocks (no transfers of control), this corresponds to gathering the MOPs into as few microinstructions as possible; subject to, of course, the data-precedence requirements on the MOPs and the resource usage constraints, which will prohibit certain combinations of MOPs from being specified in the same microinstructions. Beyond basic blocks dynamic considerations apply and we

can no longer guarantee that improvements that shorten some branches of code at the expense of others will improve running time. We investigate such improvements at length but, as is the case with some compiler optimizations, it is difficult to measure their effectiveness except empirically. We remark here that the problem at hand does not otherwise bear much resemblance to compiler optimization, except in the use of some flow graph techniques. Indeed, we assume that all ordinary compiler optimizations have already been applied to the vertical code before the gathering into microinstructions.

The reader is invited to look at the source and object codes in the examples presented in Chapters 5 and 8. The code is written for the Courant Institute PUMA System, and, with proper documentation, the source code could be easily understood, having the flavor of assembly language level code. The corresponding horizontal code, however, is usually quite obscure.

Why This Is Important

After extensive experience with the highly horizontal PUMA microcode, it is clear that this aspect of microprogramming, producing parallel code, is most unpleasant, very time consuming, and very error prone. Furthermore, those not very familiar with the techniques would seem

essentially prevented from producing any practical microcode at all. Even for a skilled programmer, writing a large interpreter would be a most formidable task without automatic parallelization.

The same consideration appears to be true when one is compiling high level languages (machine dependent or independent) into horizontal microcode. While most compilation tasks involve the same concepts as compilation into machine language, it is essential that the compiler be able to make reasonably full use of the machine's resources. If user microprogramming of horizontal machines is going to become somewhat common, and this appears likely, it seems clear that the techniques investigated herein will be important. It is often mentioned in the literature that automatic parallelization is a necessary and missing systems aid; however, it is also felt that finding practical methods of parallelizing is a difficult problem toward which little progress has been made [AGER76], and is regarded as "next to impossible" [ROSS75].

CHAPTER 3. Previous Investigations of This Problem

Fortunately, an excellent survey of methods of optimizing microprograms exists, Agerwala [AGER 76]. That survey, updated in March 1976, refers to the gathering of MOPs into microinstructions as "word dimension reduction" and generally considers it to be the most promising area of optimization. Nonetheless, the first paragraph of the conclusions section states

> "Most of the important work to date on microprogram optimization has been surveyed in this paper and, unfortunately, the results are disappointing. Very few techniques exist that can be profitably applied in any practical environment."

The survey presented here supports that conclusion. It seems to be unknown whether any of the work done to date can provide enough help to the microprogramming systems designer to enable the writing of large programs using sequential code.

We now survey all the parallelization methods we were able to find. The main reduction algorithms are presented in some detail and an annotated bibliography contains all relevant references. The references are generally from two sources: the IEEE Computer Society Transactions on Computers and the SIGMICRO Yearly Workshop preprints. The SIGMICRO papers, unfortunately, are very loosely edited, and the algorithms are often imprecise. Despite that, they are the best source of current information.

In our descriptions of the algorithms we will use terminology from processor scheduling theory; indeed, in Chapter 5 we will formalize an identification between optimizing microcode and scheduling processors. Most terms should be clear enough from their contexts to provide a general understanding of the algorithms, but all terms used are defined carefully in the next chapter. For the time being, we may think of the MOPs as having a data-dependency relation upon them; that is, some MOPs will have to be placed in earlier microinstructions than others to preserve data validity. Using this relation, we form the data dependency graph referred to below. The tasks referred to in the following chapter definitions (Fig. 4.1) will be representing the MOPs, as we will explain in Chapter 5.

Optimal Solutions

We can break the work done into two categories, algorithms which always find the optimal solution, and those which might not. We consider the former first; though none of them seems useful in a practical environment. Evidently, the first algorithm was proposed by Astopas and Plukas [ASTO 71]. They consider all possible gatherings of MOPs (partitionings) which don't violate the data dependencies; they then accept the shortest one which doesn't violate the resource conflict criterion.

An improvement upon this algorithm is made by Yau, et al. [YAU 74]. They generate only valid microinstruction partitionings, and not all of those; they then select the shortest, or stop if they obtain one which is provably minimal. Unfortunately, this still seems to use far too much time to be of much use. Of interest in that paper, though, is the fact that they use heuristics to guide the order in which the operations are grouped, stopping if they reach a provably minimal partition. Since they then present a not necessarily minimal algorithm which uses much the same heuristics, this paper is referred to later in this section.

Finally, an algorithm is given by Tabandah and Ramamoorthy [TABA74]. It is considered in the context of the SIMPL compiler (see the next reference) and is similar in style to the nonoptimal algorithm of Ramamoorthy and Tsuchiya, which is presented next. Again, this algorithm requires immense amounts of time and space and is not suggested as a practical solution.

Approximate Solutions

We now consider algorithms which produce suboptimal results but are possibly practical. The most important work seems to be that connected with the SIMPL compiler, Ramamoorthy and Tsuchiya [RAMA74]. SIMPL is a high level microprogramming language using the rather restrictive

single identity principle, in which variables may be assigned values only once. It is first compiled into sequential microcode and is then parallelized. Briefly, their parallelizing algorithm is as follows:
Ignoring resource constraints, they identify critical MOPs as those on the critical path(s), which are paths of maximal length on the data-precedence graph. The natural partitioning of these MOPs is taken, in which MOPs the same distance from the top are in the same partition. Each of these is then split up into the minimum number necessary to avoid resource conflict. (Here a potentially large loss of optimality is evident, as data and resource independent MOPs from adjacent partitions are never placed in the same new intermediate partition.) Finally the non-critical MOPs are placed in this scheme from earliest to latest; occasionally a new partition is formed for one of them when it would delay its successors too much to place it legally in an existing partition.

The next algorithm we consider is that of Tsuchiya and Gonzalez [TSUC74], also developed in relation to the SIMPL compiler and meant to be an improvement over the one given above. Briefly, they do the following:
The latest partitioning is formed, in which MOPs are placed as late as possible without increasing the number of partitions over the minimum, and with no regard paid to resource conflict. Individual partitions are then considered earliest

to latest. If no resource conflicts exist, as many MOPs as can be brought in legally from later partitions are and this becomes a permanent microinstruction. If conflicts do exist, all possible MOPs whose data dependencies will allow are brought into the partition and a choice of a permanent set of MOPs for this microinstruction is made. The remaining MOPs are pushed into the next partition, where they may have a ripple effect on their successors, possibly causing new instructions after what had been the last.

Unfortuntately, the algorithm was not given in enough detail for us to resolve conflicts between the criteria given for choosing which MOPs to delay, and those actually delayed in a rather detailed example.

Next we consider the previously mentioned algorithm of Yau, Schowe, and Tsuchiya [YAU74]. This is rather different in spirit from the previous two in that it constructs microinstructions an instruction at a time, rather than starting with a more global partitioning and then altering it. Their algorithm is essentially:

> The weight of a MOP is defined as its number of descendants (not necessarily direct descendants), and the weight of a microinstruction as the sum of the weights of its constituent MOPs. Microinstructions are formed one at a time, from the earliest to the latest, by considering every possible legal micro-

instruction producable from the MOPs not yet used and selecting the one with the greatest weight. We remark that only microinstructions which cannot have another MOP legally added to them need be considered for maximal weight.

As will be explained in more detail later, we found this the most interesting of the algorithms here, and investigated it in some detail.

Dasgupta and Tartar [DASG76] present an algorithm which they claimed to be optimal; however, it simply considers MOPs in their source order and places them each as early as possible without violating resource and data-dependency rules. Its nonoptimality is evident from very short examples [DASG78], but it is difficult to ascertain that from the algorithm, which is quite complex due to their handling of "poly-phase" microinstructions (which we will briefly discuss in Chapter 9, and which we feel would not present a great deal of difficulty in an implementation).

Finally, we consider the work of Tokoro et al. [TOKO77]. Just as was the case in [DASG76], they have a somewhat more general model, involving "microtemplates", which deals with many more options in microprogrammed machine design than we are considering here. As such, the optimization algorithm, as we would consider it, is somewhat obscured.

When projected down to our environment, however, their algorithm is the same as that in [YAU74], with the exception that they select the microinstruction at each level which has the most MOPs whose longest path distance to the bottom of the data dependency graph is greatest among the remaining MOPs. Ties are broken by simply picking the instruction with the most MOPs.

Beyond Block Optimization

We were only able to find two references to optimization beyond basic blocks, a subject about which we will have much to say in Chapter 8.

The first, again an algorithm of Dasgupta [DASG77], only looks for pairs of basic blocks, (B_i, B_j), with the property that B_j is executed during a run of the code if and only if B_i is. (For basic block definitions, see Chapter 8.) The earlier block, B_i, is parallelized, using the nonoptimal algorithm in [DASG76]. Then MOPs in B_j whose data dependencies allow are moved up into B_i, if they can be fitted into holes in the already existing optimization of B_i. (That is, without lengthening B_i.) Otherwise, they are scheduled in B_j according to the basic block algorithm.

The other reference to beyond basic block optimization, that of Tokoro et al. [TOKO78], is considerably more ambitious. Although a great many details are omitted, we can describe the spirit of their methods. They produce a

small catalog of the types of MOP motions from one block to another, such as moving a MOP from a block into all of the blocks that must follow it. (We have a catalog of that sort in Chapter 8.) They then proceed in an upward direction, moving MOPs into holes in already optimized blocks. Finally, the same thing is attempted in a downward direction.

CHAPTER 4: Practical Results in Processor Scheduling Theory

Our approach to this problem has been to identify the main aspects of it with special cases of the processor scheduling problem with resource constraints. Processor scheduling theory has received wide study (e.g. see [COFF76]), and we have been able to use some of the results and methods of attack used for the more general problem.

Processor Scheduling

The processor scheduling problem we are interested in can be described as follows: we are given a set of tasks to be processed, $t_1, t_2, \ldots, t_s$ and an acyclic partial order on those tasks (the partial order specifies a time precedence on the tasks, i.e. if $t_i < t_j$, then t_i must be completed before t_j begins). Each task takes some length of time to be processed, and we have m identical processors, $P_1, \ldots, P_m$ with which to process these tasks. Furthermore, there is a set $R = \{r_1, r_2, \ldots, r_n\}$ of resources and a function U where $U(t_i, r_j)$ is between 0 and 1 and specifies the proportion of resource j which is used by task t_i in one time unit. In fact, we will restrict ourselves to tasks with identical times (or unit execution time – UET scheduling), but will in later sections make reference to situations in which tasks take longer than 1 unit (e.g.

the PUMA's 2 cycle add).

A schedule is an assignment of the tasks to discrete time units (this assignment is called partitioning; the time units, partitions) such that:

(1) No more than m tasks are assigned to any time unit, corresponding to the m processors available to process them

(2) If $t_i < t_j$, t_i is assigned to an earlier time unit than t_j

(3) Given any resource, r, and any time unit, the sum of the U(t,r) for all t's assigned to the time unit is less than 1; that is, we don't use up more of a resource than there is.

We are generally interested in finding schedules in which the number of time units used (the "length of the schedule") is near optimally small. Finding the actual minimum is np-complete; indeed it's np-complete under the restriction that there are but 2 processors, and one resource which tasks use either completely or not at all (see Ullman's paper in [COFF76]), and under other, similar restrictions [GARE74].

Our formal definitions for scheduling theory, given in Figure 4.1, make no mention of processors. Indeed, if we had an m-processor system, we could define a resource r_p such that $U(t_i,r_p) = 1/m$ for all i, and r_p would completely

Figure 4.1

A Formal Description of Task Scheduling with Resources

A. Schedule Definitions.

1. We have a *set of tasks* $T = \{t_1, t_2, \cdots, t_s\}$ of size s (intuitively, the tasks are jobs which we are going to process).
2. We have a *partial order* < on the tasks, and, thus an associated dag. We distinguish some of the edges by writing <= rather than simply < , and in any picture we draw of the dag we place an "=" next to any edge distinguished by a <=.
   (Intuitively, $t_i < t_j$ means that we are required to process t_i before t_j. $t_i <= t_j$ means that we are required to process t_i no later than t_j.)
3. We have a *set of resources* $R = \{r_1, r_2, \cdots, r_u\}$ of size u.
   (Intuitively, the r_i's are things used in the processing of the tasks, We think of them as being present in limited amounts and we think of the tasks as competing for their use.)
4. We have a *map* $U: \{T \times R\} \rightarrow [0,1]$ specifying the percentage of each resource used by each task
   (i.e. $U(t_3, r_7) = .125$ means that task 3 uses 1/8 of the available amount of resource 7).
5. We define a *partition* as any subset of T, and a *partitioning* as an ordered tuple of partitions which are mutually disjoint and exhaustive. That is, if for each i, $P_i \subset T$, the P_i's are partitions. If $P = (P_1, \ldots, P_k)$ such that $P_j \cap P_i = \emptyset$, unless i = j, and such that $\bigcup_i P_i = T$, then P is a partitioning.
   Note that the partitioning is determined not only by the P_i's , but also their order.
   When we refer to a specific partitioning $P = (P_1, \ldots, P_k)$, we say *a task t is at level* n when $t \in P_n$; we also may refer to P_n as a *level* or *level* n, and call P_n a *cycle*.
6. We further say a partitioning P is a *legal partitioning* if both of the following hold:
   (a) Given any t and t' $\in$ T such that t < t' with t at level n and t' at level m, then n < m. Similarly, if t <= t', then n <= m.
   (b) Given any partition P_j in P, and any resource $r_i \in R$,

$$\sum_{t_k \in P_j} U(t_k, r_i) <= 1 .$$

A legal partitioning is more commonly called a *schedule*.

(Intuitively, we think of our tasks as each taking one time unit to process. Each partition, then, is one time unit, and the tasks belonging to that partition are all thought of as being done in parallel in that time unit. Condition (a) then assures that the specified task precedence is not violated. Condition (b) assures that in no time unit is more than the available amount of any resource used.)

Figure 4.2

B. Directed Graph Definitions (used in describing algorithms):

1. If $t_i < t_j$ or $t_i <= t_j$ we say that t_i is a *predecessor* of t_j and that t_j is a *successor* of t_i. When we wish to distinguish between the two types of precedence, we sometimes use the terms *strict predecessor* and *equal predecessor* with the obvious meanings. Similarly for successor.
2. We formally define the *height* of a task, HEIGHT(t), as follows:
   (a) If a task 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, then the given task has height h + 1. Otherwise the given task has height h.

   (The height may be thought of as the smallest number of time units required from the time processing starts on the given task to the end of the shortest possible schedule, given infinite resources.)
3. A *critical task* is any sequence of tasks such that:
   (a) each task is a predecessor of the following task.
   (b) the first task has no predecessors and is of the highest height in T
   (c) the last task has no successors

   We will refer to C, the "critical path length" in a graph, which is the height referred to in (b). Note that a critical path may have more than C tasks along it, due to = edges. Note also that C is a theoretical lower bound on any schedule, and that the bound would be achieved given infinite resources.

   Any task which belongs to any critical path is called a *critical task*.
4. We define the depth of a task, DEPTH(t), precisely as height was defined, with the word predecessor substituted for successor throughout.
5. The *earliest partitioning* is that in which the level of each task is its depth. The *latest partitioning* is that in which each task is at level (C - height) + 1, which we call *LATEST(t)*. Note that each of these partitionings has exactly C partitions.

   (In the earliest partitioning, every task is done as early as possible, with no regard for resource usage. In the latest, each task is done as late as possible, without adding a level, with no regard for resource usage.)
6. Given a particular partitioning P, we say that task t is *data ready at level ℓ* if all of its strict predecessors are contained in levels $1,2,\ldots,\ell-1$ and all of its equal predecessors in levels $1,2,\ldots,\ell$.

describe the processor constraint. Since the systems we will be investigating do not in general have anything corresponding to the processor set, we left it out of the definitions. Also note that some partial order edges are distinguished by equals signs in our definitions. We will need the full generality of those edges later, for the purposes of this discussion, however, we may ignore the "= edges".

Approximately Optimal Solutions and List Scheduling

We were not able to find anything in the literature but the roughest upper bounds on approximate algorithms for the full problem we are interested in. If one eliminates the resources, however, or eliminates the precedence relation, experiments have been done to rank some suggested strategies. In the case of no resources, a paper by Adam , Chandy and Dickson [ADAM74] studied various "list scheduling" strategies. List scheduling, which is summarized more formally in the last section of this chapter, basically involves choosing a heuristic function to assign a priority value to each task. The first partition is scheduled by choosing tasks, in order of their priorities, from those that are data-ready. Each task is examined to see if it can be placed in the time unit without any resources being used above capacity, and it is so placed, if possible. The partition is fixed when either no more data ready tasks

exist or all of the processors have been used up. Following partitions are then filled in the same way; scheduling is finished when no tasks remain. It is clear that a legal schedule is formed in this manner.

The attractions of list scheduling include:

(1) It is fast and straightforward; in particular, no scheduled task is ever moved by a later step.

(2) The heuristic portion of it is totally isolated from the scheduling aspects.

(3) It has a record of good performance in some environments.

Adam et al. studied the list schedules produced by five strategies, namely:

(1) The priority is the length of the largest chain from the given task to the exit. Since their model included nonunit task times, the priority is the total length of all tasks on the chain. They referred to this as HLFET (Highest levels first, estimated times); in our definitions (Fig. 4.2) we have referred to this as the height of a task, for UET scheduling.

(2) As above, except ignoring tasks times, referred to as HLFNET (no estimated times).

(3) Random priorities.

(4) The priority is the closeness to the entrance of the graph, referred to as SCFET (Shortest co-levels). For UET scheduling, we could use -DEPTH to get the same ordering.

(5) As above, with no tasks times - SCFNET.

These strategies were used to produce list schedules for hundreds of precedence graphs, some containing hundreds of edges or nodes; some of the graphs were randomly produced, some culled from real programs. In all cases (and also when the same thing was done for tasks with stochastic task times), the results were in the same order: HLFET, HLFNET, SCFNET, RANDOM, SCFET with HLFET being the superior. What's more, the best known lower bound for this case, that of Fernandez-Bussel [FERN73], was rarely exceeded by HLFET in any class tested by more than 0.2 percent, even for very large graphs (one case was 16 percent worse, one 4, the rest under 2, and a great many hit the lower bound). As we will discuss in Chapters 6 and 7, we were able to put the ideas in [FERN73] to several good uses, but, for the moment, we note that these results demonstrate both the effectiveness of highest-level priorities and the tightness of the bound in this environment. The good performance of the highest level first strategies is not a great surprise; it has been in the folklore for some time that that's the right way. The CDC FTN Fortran compiler optimizes basic blocks this way, as does an optimizer for the CRAY-1 written by Richard Sites, and for some restricted classes of problems, optimal schedules can be formed using highest level type lists (see, e.g., [COFF72]).

Resource Constrained Problems

Naturally, such strategies could be applied directly to resource constrained problems, but they would be what we have dubbed "resource inconsiderate". It is certainly true that one is often faced with a resource bottleneck that indicates priorities opposed to what would be suggested by a level heuristic; we shall see examples of that shortly. There is, as we have mentioned, a study [ECKE78] which has compared two resource considerate strategies, but in an environment in which the precedence relation was empty. Under such conditions, the problem is called "generalized bin packing" and the heuristics used generalize bin packing strategies. We refer to the two strategies tested as RMAX and NEIGHBORHOOD:

(1) RMAX: the priority of a task is the maximum component in its resource usage vector, that is,
PRIORITY(t_i) = $\max_j \{U(t_i,r_j)\}$.

(2) NEIGHBORHOOD: we define a relation CLOSE as: two tasks, t_i and t_j, are close if they could ever be scheduled together without a resource being overused, i.e. for all k, $U(t_i,r_k) + U(t_j,r_k) \leq 1$. Then the priority of a task is the number of tasks it is CLOSE to.

Eckert chose his simulation parameters in such a way that RMAX would seem to be favored, since individual resources are not often pairwise overused. Nonetheless, the NEIGHBORHOOD

strategy did noticably, though not decisively, better.

Our concern is with the full problem. In the next two chapters the ideas of [ADAM74] and [ECKE78] are combined and extended, and many other ideas along similar and dissimilar lines are suggested and tested.

We close this chapter with a formal description of list scheduling.

A Formal Description of List Scheduling

In the absence of = edges, we have what is called, in the scheduling theory literature: *unit execution time (UET) scheduling with resources*. The quality of schedules is measured by the number of levels produced, and an aim of scheduling theory is the derivation of methods which produce short schedules.

We now present a method of UET scheduling called *list scheduling*, the desirable properties of which have been outlined in the accompanying text. This will later be modified to include = edges.

Algorithm: List Scheduling

Input: T, R, U, < given as in Figures 4.1 and 4.2, with accompanying definitions.

Uses: A separate routine — PRIORITYSET — forms a function PRI: T → real numbers. PRI(t_i) is thought of as the *priority* that task t_i be scheduled early.

An ordered list of tasks, READY.

A set of tasks, NOT READY. NOT-READY $\subset$ T.

Output: A schedule, namely a legal partitioning $P = \{P_1, P_2, \ldots, P_\ell\}$.

Method: We use a list called READY, the data-ready list, which initially contains all tasks without predecessors and which is sorted by priority. The first level is formed by considering the tasks on the list in order and placing each one in that level if it does not cause any resource to be overused. A task so placed is deleted from the data ready list. After no more tasks can be placed, the data ready list is updated to contain all tasks which will, as a result of their predecessors' being scheduled, be ready at the next level. The next and following levels are scheduled in the same way.

Algorithm:

1. Call PRIORITYSET, defining PRI(t_i) for each task $t_i \in T$.
2. c = 1, READY = empty, NOT-READY = T, $P_1 = \emptyset$
3. For each task $t_i \in$ NOT-READY which is now data ready, do:
   Place t_i on READY in order of PRI(t_i)
   Delete t_i from NOT-READY
   End.

4. Scan the READY list top to bottom (i.e. in order of priority). For each t_i on READY, if

   for all k, $\sum_{t_j \in P_c} U(t_j,r_k) + U(t_i,r_k) \leq 1$

   then do:

   Place t_i in P_c

   Delete t_i from READY

   End.

5. If tasks remain on READY or NOT-READY,

   then do:

   $c = c + 1$

   $P_c = \emptyset$

   go to step 3

   end.

   Otherwise, *STOP*. A schedule has been formed from $P_1, P_2, \ldots, P_c$.

Note: The above algorithm was chosen for clarity. It appears to require that $|T|^2$ elements be scanned at step 4 in a total run of the algorithm, and this seems unlikely to be improvable. In practical terms, however, it is probably a significant constant factor faster to:

1. Keep a count of the predecessors of each task
2. Whenever a task is scheduled, decrement each of its successor's counts by 1
3. Whenever a task's count reaches 0 during the scheduling of some level, put the task on an ALMOST-READY list.

4. After scheduling is completed for some level, insert the the ALMOST-READY tasks onto the READY list.

Indeed, our implementation does this.

It is also worth noting that a sophisticated PRIORITYSET routine is apt to dominate the efficiency considerations.

5. Optimizing Basic Blocks of Microcode

Formal Identification between Optimizing and Scheduling

We now do what we have been alluding to all along, that is, we recast our problem as one of processor scheduling with resource constraints. To make our formal identification, we will need to specify what our tasks are, how the partial order is defined, and what the resource mapping is to be.

Suppose we are given a basic block of sequential microcode (which will be defined carefully in Chapter 8, but which we informally say has no jumps out of the block, except at the end, and no jumps in, except at the beginning). We define as our tasks the individual MOPs. The resource usages of each MOP will be completely machine dependent, but will involve such resources as busses, ALU's, multiplexers, etc.

Our algorithm for determining the partial order on the MOPs is given in Figure 5.2. The algorithm presupposes that the registers read and written by each MOP are known and from those sets determines $<$, with some edges distinguished by an $=$, as explained earlier.

Formal Identification between Processor Scheduling and Basic Block Microcode Optimization

Processor Scheduling	Microcode
Set of tasks	Micro operations
Acyclic partial order	Data precedence relation preserving data validity (see Figure 5.2)
Resources	Hardware resources in computer (e.g., ALU, BUSSES)
One time unit in a task processor resource schedule	A horizontal microinstruction

Figure 5.1

Rules for Formation of Partial Order on Micro-operations

Given: 1. A set of MOPs $T = \{t_1, t_2, \ldots, t_s\}$, where T represents, with subscript order equal to source order, a basic block of MOPs.

2. A set of registers $A = \{A_1, A_2, \ldots, A_v\}$

3. For each t_i , two sets, $READ(t_i)$ and $WRITE(t_i) \subseteq A$, not necessarily disjoint.

We produce a partial order < on T, with some edges distinguished by writing <= , as follows:

For each pair of MOPs t_i, t_j with $i < j$ (i.e. t_i comes before t_j in the source code):

1. If $READ(t_i) \cap WRITE(t_j) \neq \emptyset$, then $t_i <= t_j$ unless for each $a \in READ(t_i) \cap WRITE(t_j)$ there is a k such that $i < k < j$ and $a \in WRITE(t_k)$.
2. If $WRITE(t_i) \cap READ(t_j) \neq \emptyset$, then $t_i < t_j$, unless for each $a \in WRITE(t_i) \cap READ(t_j)$ there is a k such that $i < k < j$ and $a \in WRITE(t_k)$.
3. If $WRITE(t_i) \cap WRITE(t_j) \neq \emptyset$, then $t_i < t_j$. Unless for each $a \in WRITE(t_i) \cap WRITE(t_j)$ there is a k such that $i < k < j$ and $a \in WRITE(t_k)$.
4. If by the above rules, both $t_i < t_j$ and $t_i <= t_j$, then we write $t_i < t_j$.

Figure 5.2

Algorithm for the above:

Input: T, A, READ, WRITE as above, except T is augmented with a dummy task, t_0.

Uses: A function LASTWRITE A → T initially into a dummy task t_0.

A function READS_SINCE_WRITE: A → subsets of T, initially into the empty set.

Output: A function STRICTPRED: T → subsets of T

and EQUALPRED: T → subsets of T.

$t_j \in$ STRICTPRED(t_i) will mean that $t_j < t_i$, while $t_j \in$ EQUALPRED(t_i) will mean that $t_j <= t_i$.

Method: We consider the tasks in source order. For each task t_i we look at the set READ(t_i). For each element $a_j \in$ READ(t_i), that is for each register that t_i reads, we put the last task to write a_j, (that is, LASTWRITE(a_j)), in the set STRICTPRED(t_i). We then add t_i to READS_SINCE_WRITE(a_j).

Similarly, we consider each register $a_k \in$ WRITE(t_i). For each we put all the tasks belonging to READS_SINCE_WRITE(a_k) on EQUALPRED(t_i). If READS_SINCE_WRITE(a_k) is empty, we put LASTWRITE(a_k) on STRICTPRED(t_i). (If we did so even when READS_SINCE_WRITE(a_k) was not empty, it would still be correctly following rule 3 above, but would produce a redundant ("transitive") edge.) Finally

Figure 5.2
(Continued)

we set READ_SINCE_WRITE(a_k) = $\emptyset$ and we set LASTWRITE(a_k) = t_i.

After processing each task, we "cleanup" by removing all edges from t_0, by removing all duplicate edges (resolving contentions in favor of STRICTPRED), and removing all edges from a task to itself.

Algorithm:

INITIALIZE:

1. STRICTPRED(t_i), EQUALPRED(t_i) empty, for all $t_i \in T$.
2. READS_SINCE_WRITE(a_k) = empty, LASTWRITE(a_k) = 0, for all $a_k \in A$.

FORM EDGES:

3. For i = 1 to s DO:
    4. DO for each $a_k \in$ READ(t_i):
        5. STRICTPRED(t_i)=STRICTPRED(t_i)$\cup${LASTWRITE(a_k)}
        6. READS_SINCE_WRITE(a_k)=READS_SINCE_WRITE(a_k) $\cup$ {t_i}
    7. END 4
    8. DO FOR EACH $a_k \in$ WRITE(t_i):
        9. IF READS_SINCE_WRITE(a_k) = empty THEN STRICTPRED(t_i)=STRICTPRED(t_i)$\cup${LASTWRITE(a_k)}
        10. ELSE DO:

Figure 5.2
(Continued)

11. EQUALPRED(t_i)=EQUALPRED(t_i)
    ∪ READS_SINCE_WRITE(a_k)
12. READS_SINCE_WRITE(a_k) = empty
13. END 10
14. LASTWRITE(a_k) = t_i
15. END 8

CLEANUP:

16. STRICTPRED(t_i) = STRICTPRED(t_i) - {t_0}
17. EQUALPRED(t_i) = (EQUALPRED(t_i)-STRICTPRED(t_i))
    - {t_0,t_i}
18. END 3

NOTE: As implemented, the above avoids doing $o(|T|^2)$ of any operation, the order of the most frequently executed steps being o(E), the number of edges in the graph being formed. This required that for the t_i under consideration in steps 3 through 18, an array ALREADY(t_j) kept the status of the edge (t_j,t_i), which was zero if no precedence was found, 1 if only equal precedence was found, and 2 if strict (or both) precedence(s) were found. (t_j,t_j) was set back to zero after ALREADY was built. All $t_j \in$ STRICTPRED(t_i) ∪ EQUALPRED(t_i) were then scanned, and a "cleaned-up" predecessor set was built from that information, with that step also taking o(E) steps. As Figure 5.4 shows, E is o(|T|) in this type of graph.

Figure 5.2
(Continued)

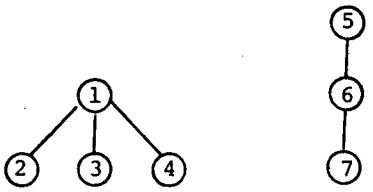
An Example

We have provided an example, so that the application of scheduling theory, and particularly list scheduling, may be easily followed. Example 5.1a contains a short sequence of MOPs, written for the PUMA. We first identify the individual MOPs as the tasks; there are 7 of them, which we can refer to as $t_1, t_2, \ldots, t_7$. Our goal is to bunch them into microinstructions, which we think of as the discrete partitions of a schedule. Corresponding to the precedence relation on the tasks we have data-prededence requirements on the MOPs, from the register usages in example 5.1b. The data-precedence requirements simply assure that no MOP reads a register before it is valid or after it has been erased, and we see in example 5.1c the data-predecence graph on $t_1, \ldots, t_7$. Note that we use the term register rather loosely here. For example, since we are unlikely to be capable of a range analysis, we consider all of main memory to be one register, and we say that any MOP which reads any memory location must follow any (earlier source) MOP which writes any memory location and that two memory writes may not be permuted. This is not likely to be serious if the code being optimized is a microcoded emulator, since the memory references will strongly depend upon the algorithm being interpreted, rather than that in the microcode, and a range analysis is unlikely to provide much help. This is, however, a potentially serious deficiency when a given

	SEQUENTIAL CODE	REGISTER(S) READ	REGISTER(S) WRITTEN	RESOURCE(S) USED *
(1)	AC = BUF	BUF	AC	ALU (1 unit)
(2)	YO = AC	AC	YO	XYREGBUS (1)
(3)	Y1 = AC	AC	Y1	XYREGBUS (1)
(4)	Y2 = AC	AC	Y2	XYREGBUS (1)
(5)	MQ = -BUF	BUF	MQ	ALU (1)
(6)	MQ = SHIFT(MQ,L1)	MQ	MQ	SHIFTER (1)
(7)	MQ = SHIFT(MQ,L1)	MQ	MQ	SHIFTER (1)

5.1 a 5.1 b 5.1 d

* We have one unit of each of these resources available.



5.1 c

Using PRIORITY(t) = HEIGHT(t)
PRIORITY LIST: 5 1 6 2 3 4 7

Schedule:

5	MQ = -BUF
1 6	AC = BUF; MQ = SHIFT(MQ,L1)
2 7	YO = AC; MQ = SHIFT(MQ,L1)
3	Y1 = AC
4	Y2 = AC

5.1 e

Using PRIORITY LIST: 1 5 6 2 3 4 7

Schedule:

1	
5	2
6	3
7	4

5.1 f

EXAMPLE 5.1 a-f

applications program is compiled into microcode. The programmer, or a smart compiler, may very well know that references differ and that no data-dependency is implied.

Example 5.1 has been chosen to have obvious data-dependencies. It is worthwhile to note that one of the characteristics of the microprogram level of a machine is that it tends to have many hidden and surprising register usages, reflecting some of the subtle aspects of the machine's design. While this complicates all aspects of microcode generation, it can make hand optimizing particularly difficult, especially when there has been a time lapse between the production and optimizing — as in debugging. This presents little difficulty to the automated optimizer, however.

Note that we mark some edges on our graph with an equals sign; this indicates that the following task can be done no earlier than the preceding one, but they may be done simultaneously. In many machines, PUMA included, master-slave flip-flops permit the valid reading of a register up to the time that the register writes occur. Thus a write to a register following a read of that register may be done in the same cycle as the read, but no earlier. Because of "= edges", a task may become data-ready in the course of scheduling a microinstruction if all of its remaining unscheduled predecessors had "= edges" to it at the start of the formation of the microinstruction

and all were scheduled in the microinstruction. Figure 5.3 specifies the changes necessary to our list scheduling algorithm, given at the end of Chapter 4, to allow for "= edges".

Resource Constraints

We finally need to consider the resource constraints on MOPs. In the PUMA, and we suspect in most machines, the full generality of the resource usage function is never used. In most cases, each MOP uses a set of resources, usually one or two, and each resource it uses, it uses completely. Thus we would expect function values of all zeros, except for a few ones. A somewhat different form of resource conflict occurs when one considers hardware which has mode settings. That is, an arithmetic-logic unit might be able to operate in any of 2^k modes, depending on the values of some lines. Two MOPs which require the ALU to operate in the same mode might not conflict, yet they both use the ALU, and would conflict with other MOPs using the ALU in different modes. A similar situation occurs when a multiplexer selects data onto a data path; two MOPs might select the same data, and we would say they have compatible use of the resource. The possibility of compatible usage makes efficient determination of whether a MOP conflicts with already placed MOPs more difficult. An interesting and efficient way of dealing with this is discussed elsewhere.

Alterations of List Scheduling to Account for "= Edges"

In the algorithm for list scheduling at the end of Chapter 4, replace step 4 with:

4a. NEXT-READY = $\emptyset$

4b. Find the highest priority task on the READY list, call it t_i.
If READY = $\emptyset$, *then do*:

READY = NEXT-READY
GO TO STEP 5
end

4c. If for all k, $\sum_{t_j \in P_c} U(t_j,r_k) + U(t_i,r_k) \leq 1$
then do:
Place t_i in Pℓ.
For each equal successor t_j of t_i on the data precedence graph,
if: (i) All of t_j's strict predecessors were scheduled in P_{c-1} or earlier
and (ii) All of t_j's equal predecessors were scheduled in P_c or earlier
then do: remove t_j from NOT-READY
place t_j on READY
end

else Place t_i on NEXT-READY

4d. remove t_i from READY; go to step 4b.

Figure 5.3

In example 5.1d, we see that it is sufficient to consider only three resources to determine the conflict relations among the tasks — all other resources have been left out for simplicity. If we ignore these resources in forming our priorities and schedule using a highest level first list, as in example 5.1e, we see that five microinstructions are generated. With a little reflection, though, we can see that t_2 , t_3 , and t_4 all form a resource bottleneck, and t_1 must be given priority over t_5 to get through this bottleneck quickly, even though t_5 has a higher level. It isn't just resource inconsiderate strategies which are unable to deal with this, though. Neither RMAX nor NEIGHBORHOOD would distinguish between t_1 and t_5 , since they both have precisely the same resource usages. Even a strategy like taking the sum of a task resource priority and level priority would fail here, since t_5 would still have priority over t_1 , and, as example 5.1f shows, putting t_1 ahead of t_5 would generate only four microinstructions.

Naturally, one can invent a clever example which will make any efficient strategy look bad and we were quite curious about whether this is a common situation. In Chapter 7 we report on experiments we have done to test many strategies for the production of list priorities.

A Note on Efficiency

List scheduling seems to be an n^2 (in the number of tasks) time complexity algorithm, but when coded efficiently appeared to run linearly. After examination, a possible explanation occurred to us; namely, the code seemed linear in the number of edges in the data-precedence graph. While, normally, the edges of a dag grow as n^2, the number of edges derived according to the rules in Figure 5.2 is limited in one dimension by the number of registers used, which would not grow with the number of tasks (unless, possibly, if the memory locations were thought of as individual registers and a range analysis were done, which does not seem relevant to these optimizations). Thus the number of edges, and the algorithms used, grow linearly with the number of tasks. The argument that the number of edges grows as the product of the number of tasks and the number of registers is presented in Figure 5.4.

In summary, then, we see that the basic block problem is very little different from the scheduling problem presented in Section 4, but that the methods used on restrictions of the problem are possibly not effective enough on the full problem, even when combined.

Before presenting the results of our experiments we, in the next chapter, concern ourselves with a lower bound which will help us interpret the results of our experiments and will provide a basis for some of the strategies tested.

E is $o(|T|)$ in Data Precedence Graph:

We show here that the number of edges in the data dependency graph of a set of MOPs grows linearly in the number of MOPs, despite the fact that the in-degree and out-degree of any single MOP may itself grow linearly with the number of MOPs.

We are given a set of MOPs of size s, and a set of v registers, as in Figure 5.2.

According to Figure 5.2, there are three sources for edges (t_i,t_j) defined on the MOPs:

1. where t_i reads a register which is next written by t_j
2. where t_i writes a register which is read by t_j, before any other writes to the register
3. where t_i writes a register which is written by t_j, before any other reads or writes to the register (we say that t_i is an unreferenced write).

We claim that each of the above contributes at most v * s edges to the graph. For edges of type 1, we maintain that at most v edges could leave any task, since for each of the v registers that t_i reads, there will be at most one MOP which next writes that register. Thus each of s MOPs could follow t_i via a type 1 edge, and only v * s of them could exist.

Similarly, if t_j reads a register, then only one t_i could be the immediately preceding write of that register, and only v type 2 edges could have t_j as their target. Thus only v * s edges of type 2 could exist.

Finally, for each of the up to v registers that t_j writes, only one unreferred write could immediately precede it, and again only v * s edges could exist.

We see, then, that fewer than 3 * v * s edges could be generated, and the number of edges is o(s). This is somewhat surprising in light of the fact that one MOP could have in-degree of v * (s-1)/2 and out-degree of v * (s-1)/2. This would happen, for example, if all MOPs read all registers, the middle MOP was the only one to write any register, and it wrote them all.

For a given machine, v is a small constant, but s, while possessing a theoretical upper bound, can grow large enough that algorithms requiring $o(s^2)$ operations or space can take significantly longer than those requiring o(s).

Figure 5.4

A More Complex Example

We close this chapter with an actual example from the PUMA's existing, hand-optimized, microcode; the code is part of the emulation of the CDC 6600 central exchange jump. The example is of interest because, although it is not a frequently executed portion of code, much attention was paid to the hand optimization of the whole emulator. Most list schedules would produce code which requires eight cycles instead of the nine cycles found in the PUMA. Upon investigation, it is clear that the hand optimization was defeated by the intricacies of MOP compatability, rather than data-dependency. It is also of interest to note that this is the only block in the PUMA code which wasn't done in obvious minimum time, and it seems that any reasonable strategy would produce minimum length code for every PUMA block. We'll have more to say about the implications of this in Chapters 7 and 8.

The exchange jump example is presented briefly and without comment as Example 5.2. Note that it includes two-cycle MOPs, which we consider in Chapter 9, and a jump MOP, which we force to the end, but which we consider at length in Chapter 8.

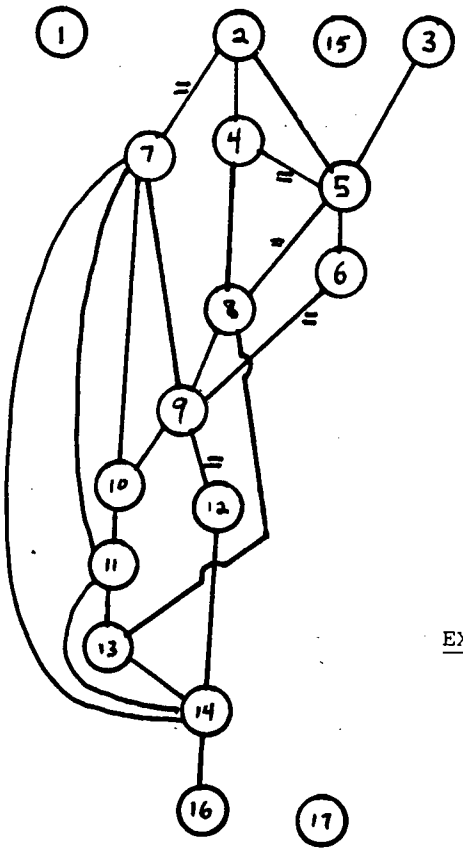
SOURCE CODE (AS PARALLELIZED IN PUMA):

```
CLEAR;  AC = MQ;  BUF = Y0
Y2 = AC;  AC = AC & ∿BUF
BM = AC;  MQ = 0
BUF = Y2
AC = SHIFT(BUF:MQ, R16)
AC = SHIFT(AC:MQ, R1)
AC = SHIFT(AC:MQ, R1);  BUF = Y0
Y2 = AC;  AC = AC & ∿BUF;  P = P+1
AM = AC;   = 7 + E1; IF EALUPOUT_
                    THEN XJEXTP
```

TASKS:

1 ; 2 ; 3 (i.e. 1 ≡ CLEAR etc.)
4 ; 5
6 ; 7
8
9
10
11 ; 12
13 ; 14 ; 15
16 ; 17

DATA-PRECEDENCE GRAPH



PRIORITY LIST:

2 3 4 5 7 8 6
9 10 11 12 13 14
1 15 16 17

(using PRIORITY(t) = HEIGHT(t))

SCHEDULE:

1 ; 2 ; 3 ; 7 ; 15
4 ; 5
8
6 ; 9
10 ; 12
11
13 ; 14
16 ; 17

EXAMPLE 5.2

6. A Lower Bound and its Uses

The Fernandez-Bussel Lower Bound and How It Works

Fernandez and Bussel [FERN73] have produced a lower bound on the number of cycles needed to schedule a set of tasks, given data-precedence, but no resource constraints. This bound was used in [ADAM74] to bound the distance of various list schedules from the optimal; the fact that any schedule is an upper bound on the length of an optimal schedule, and that the derived schedules were very near the bound, show that the bound was very tight in that environment. We have extended their bound, greatly reduced the computation necessary to calculate it, and have some suggested uses for it, beyond the obvious use as an experimental measure of the optimality of derived schedules.

Before explaining how the lower bound is found, we note that in [FERN73] the bound is given for systems that include tasks with arbitrary task times. Everything we do could be similarly presented, but we are primarily interested in unit execution time systems, and will, for clarity, restrict our presentation to such systems.

Finding the Bound

Given m processors, we look at all intervals (i,j), $1 \leq i \leq j \leq C$, where i and j are integers, and C is the length of a critical path in the data-precedence graph.

For any such interval, say (i_0,j_0), we consider the set of tasks t with the property that DEPTH(t) is i_0 or later, and LATEST(t) is j_0 or sooner. (For definitions, see Figure 4-2). Those tasks could not, by the definitions of DEPTH and LATEST, be scheduled any earlier than time i_0 in a schedule of length C, nor any later than time j_0. Thus if an optimal schedule were to be only C units long, all of these tasks would have to be scheduled in i_0-j_0+1 time units. But if there are T such tasks, then it will take at least T/m time units, and $(T/m) - (i_0-j_0+1)$ extra units above C will be required. We look at all intervals to find the one that contributes the greatest number of extra cycles, E. The shortest possible schedule will then be $\geq$ (C+E) units long. We thus have the formula presented in Figure 6-1 for our bound, which has been extended to include resources, as explained in 1, below.

How the Bound Loses Accuracy

The Fernandez-Bussel bound is excellent at finding local bottlenecks. Unfortunately, we can only be sure that the number of extra cycles is the largest number found for any one interval. That is, if intervals (i,j) and (i',j'), with $i < j < i' < j'$ each contributed three extra cycles, one cannot, in general, be sure whether three, six, or some intermediate number of extra cycles would

The Fernandez-Bussel Lower Bound:

Given: R the set of resources, C the length of a critical path

EARLY, LATEST: T → integers [1,C]

USAGE: T × R → [0,1]

all as defined previously.

then if

$$E = \max_{\substack{1 \le j \le C \\ 1 \le i \le j \\ r \in R}} \left\{ \left(\sum_{\substack{t \text{ such that} \\ i \le DEPTH(t) \\ LATEST(t) \le j}} USAGE(t,r) \right) - (j+1-i) \right\}$$

Then the length of an optimal schedule L_{OPT} is

$$L_{OPT} \geq C + E \quad .$$

Figure 6.1 .

suffice to relieve both bottlenecks. We searched rather hard for a set of criteria to help measure that number, but were unsuccessful. It is our belief that the number is generally the sum of the two disjoint bottlenecks, especially when the graph has many edges. As a result, the greater the length of the critical path, the less accurate the bound is likely to be. In Chapter 7 our experimental results will speak to that point.

Our Work On and Suggested Uses of the Bound

We have done the following in relation to this bound:

1. Extension to processor scheduling with resource constraints. It is possible, in the obvious way, to consider each resource separately and to calculate the usage of each resource in each interval. (In fact, the processors themselves can be considered a resource of which each task uses 1/m.) The resource-interval which contributes the most extra cycles will determine the lower bound. Unfortunately, the interaction of a set of tasks restricted to a certain interval will generally involve several resources and, in practice, the bound seems to miss the heavy resource bottlenecks. If, however, the tasks tend to use only one resource apiece, that is, if the tasks form equivalence classes with respect to which resources they use, then we are much more likely to find the worst

bottleneck. Microoperations seem to have approximately this property; the relation of clashing is generally transitive and is certainly reflexive. Some insight into the utility of this extension may be gained from the results of the experiments reported on in Section 7.

2. Efficient computation of the bound. The computational methods suggested in [FERN73] require looking at all $o(|T|^2)$ intervals and doing a set formation of complexity at least $o(|T|)$ for each interval. Thus their methods require $o(|T|^3)$ operations, at least. Our methods, presented at the end of this chapter, do a constant amount of work for each of the $o(|T|^2)$ intervals, thus requiring $o(|T|^2)$ operations.

We report on the actual computation time used by an implementation of our algorithm in Chapter 7.

3. Use of the bound as a guide to places to invest more time. Although it is true that when a derived schedule is significantly longer than the bound the fault may lie with either, such cases give some indication that an investment of more time may be worthwhile. In particular, finding spots in the list schedule where a data-ready task was delayed due only to resource constraints and trying again with a different task delayed may pay off. It may be worthwhile, in view of the reason for the loss of accuracy

of the bound, to sum up the extra cycles yielded by some set of disjoint intervals. If that sum were significantly less than the derived schedule length, further search would be indicated.

4. Resource considerate heuristics. Were it not for the data-precedence graph, scheduling with resources would be a generalization of bin-packing to weight-vectors, rather than simple weights. Various heuristics have been suggested for the generalized bin-packing problem [ECKE78], but it is not clear how to apply these heuristics to tasks on a data-precedence graph. We have attempted to use the bottlenecks found by the bound as an aid in the production of resource considerate schedules. Our method was successful, in that it consistently produced the shortest schedules of any method we tested, and could probably be "fine tuned" to do even better. Whether it offers enough improvement over simpler strategies is environment dependent; Chapter 7 contains an experimental measurement of that improvement.

Our method involves altering highest level priorities to compensate for resource bottlenecks. Rather than use the lower bound to spot only the worst bottleneck, we consider all resource-interval pairs which need extra cycles. For each such interval, we note which tasks contributed to the resource bottleneck, and we boost their priorities (and those of their

predecessors) beyond what is obtained from strictly data-precedence considerations. This would seem intuitively, to zero in more firmly on the resource constraints than strategies which permit the measurement, for example, of the resource contentions of tasks which would be unlikely to compete for scheduled places. Again, Section 7 contains a precise statement of the heuristics used, as well as a summary of experiments done.

Efficient Calculation of the Bound

Algorithm: Efficient calculation of Fernandez-Bussel Lower Bound, extended to resources.

Input: T, set of tasks

LATEST, DEPTH functions: T → integers [0,C]

U function: T × R → [0,1]

C length of critical path of dag defined on T

all as previously defined

Output: E where C + E is a lower bound on the length of a schedule for T

Uses: LPTR function: integers [1,C] → subsets of T

where LPTR(i) is $\{t_j \in T$ such that $LATEST(t_j)=i\}$

SUME, SUML functions: integers [0,1] × R → real numbers

where, initially,

$$SUME(i,r_k) = \sum_{\substack{t_j \text{ s.t.} \\ DEPTH(t_j)=i}} U(t_j,r_k) .$$

That is, SUME(i,r_k) is the amount of r_k used by all tasks whose "earliest issue time" (DEPTH) is i. SUML is the same, with LATEST replacing DEPTH

TOTAL USAGE function: R → real numbers where

$$\text{TOTALUSAGE}(r_k) = \sum_{t \in T} U(t,r_k)$$

B fixed right endpoint of the major and minor intervals

A varying left endpoint of the minor intervals

MAJOREXCESS the amount of the resource currently under consideration which is used by tasks constrained to the major interval under consideration, in excess of the amount that interval could process.

MINOREXCESS Same as MAJOREXCESS, for minor intervals

MAXEXCESS the maximum of the MINOREXCESSes.

METHOD: For each resource, we consider all of the $o(C^2)$ intervals [i,j], with $1 \leq i \leq j \leq C$, called the minor intervals. For each minor interval we determine what the excess resource requirement is, that is, how much of a resource, r_k, is used by tasks "critically constrained" (see below) to [i,j], above the j+1-i units of r_k which could be processed in [i,j] with no additional cycles. A task t is critically constrained to [i,j] if $i \leq$ DEPTH(t) $\leq$ LATEST(t) $\leq j$. We call those excesses the MINOREXCESSes, and, from Figure 6-1, we are looking for the largest such

excess. We will gather the minor intervals into C chains in such a way that the excess for any interval is the excess of its predecessor minus some already known value. The first interval on each chain is referred to as the major interval, and the chains are arranged as follows:

Major Intervals		Minor Intervals
[1,C]	→	[2,C] → [3,C] → ... → [C-1,C] → [C,C]
[1,C-1]	→	[2,C-1] → ... → [C-1,C-1]
.		
.		
.		
[1,2]	→	[2,2]
[1,1]		

To process the minor intervals, we set up a nested loop. The outer loop iterates through the major intervals from [1,C] to [1,1]. The inner loop processes the chain headed by the major interval.

Consider the first chain. The MINOREXCESS of [1,C] is TOTALUSAGE - C. For [2,C] we need to eliminate the tasks which are critically constrained to [1,C] (which all tasks are) but whose DEPTH is 1. These tasks, however, have a total resource usage of SUME(1,r_k), and so we need only subtract SUME(1,r_k) from MINOREXCESS, and then add 1 because we are losing one cycle and can process one unit less of r_k. We continue this way, subtracting SUME and

adding 1 until we do [C,C].

We now consider the second chain. [1,C-1] has the same excess as [1,C] had, minus SUML(C,r_k), plus 1, by the same reasoning as above. A problem arises, however, on the transition from [1,C-1] to [2,C-1]. This transition cannot be done by subtracting SUME(1,r_k) because some of the tasks which contributed to SUME(1,r_k) had LATEST of C, and thus were not critically constrained to [1,C-1] and were not included in the excess. Thus, after we chop off an end point to go from one major interval to the next, we must update all of the SUME function values affected. This is straightforward enough; when we go from [1,C] to [1,C-1], we take all tasks t in LPTR(C) and, for each, subtract its usage from SUME(DEPTH(t),r_k).

The formal algorithm:

```
1. In one pass through T, Form LPTR, SUME, SUML, TOTALUSAGE
2.  MAXEXCESS=0
3.  Do for each r_k ε R
    4.  MAJOREXCESS = TOTALUSAGE(r_k) - C
      /* MAJOREXCESS is likely to start negative */
    5.  Do B=C to 1 BY -1
        6.  MINOREXCESS = MAJOREXCESS
        7.  DO A=1 TO B
            8.  MAXEXCESS = MAX(MAXEXCESS, MINOREXCESS)
            9.  MINOREXCESS=MINOREXCESS + 1 - SUME(A,r_k)
       10.  END 7
       11.  MAJOREXCESS=MAJOREXCESS + 1 - SUML(B,r_k)
```

12. /* Update the SUME's to reflect new B */

FOR EACH $t_i \in$ LPTR(B),

SUME(DEPTH(t_i),r_k)=SUME(DEPTH(t_i),r_k)-U(t_i,r_k)

13. END 5

14. END 3

15. E = MAXEXCESS

16. END

Notes on the Efficiency of the Above

The loop at 3 is done a constant number of times, once for each resource in the machine. Step 5 defines a loop which is iterated C (which is o(T)) times. Within the loop, steps 7-10 are done o(C) times. In step 12, each t belongs to only one LPTR, so step 12 is iterated only once per t no matter what. Thus the worst steps, 7-10, are $o(C^2)$; in fact, they are done C(C+1)/2 times.

In any system in which many resources are sparsely used, which is probably true of most systems, it would make sense to keep a running count of the SUM value at the fixed endpoint, stopping consideration of that endpoint whenever the SUM value went below 1. If the endpoint does not contribute to the constraining of tasks enough to overcome the additional unit of resource it makes available, then one of its sub-intervals is a worse bottle-

neck than the interval with that endpoint.

Finally we note that if C^2 units of storage were used, we could just list all of the MINOREXCESSes, eliminating step 8, and use a faster routine to find the max.

7. Experimentally Obtained Measures of the Effectiveness of Several Basic Block Optimizing Strategies

Introduction and General Conclusions

In this chapter we describe experiments done to measure the effectiveness of several evaluation functions used to produce scheduling lists. The functions were tested on random task sets chosen to have characteristics similar to those we would expect to cull from a wide range of basic blocks of microcode. The schedule lengths obtained are compared to the theoretical lower bound and to schedules produced using some previously suggested methods of optimizing microcode.

We believe that our results support the contention that: Microcode optimization within basic blocks is not a critical issue. Specifically, we maintain that:

1. The differences among the best strategies we could find are very small. One of those strategies is the simple "highest levels first" mentioned earlier which produces lists in $0(E)$ time. (Though any list method will be $0(|T|^2)$, worst case, to produce the schedule given the priority list.)
2. The best strategies were very close to optimal. In an environment meant to resemble the PUMA, the four best strategies were within 8% of the theoretical bound for task sets as large as 60

tasks, and within 4% for sets of size 20. Indeed, it is our strong belief that the actual difference between the strategies and the optimal is much closer to zero. (Further experiments are underway to test this hypothesis.)

3. The PUMA microcode emulating the CDC 6600 contains about 360 basic blocks, and evidently any of the four best strategies would produce optimal code for every one. Indeed, almost all of the blocks are one microinstruction long, and so even a very poor scheduler would do very well. By hand, two of the blocks were larger than optimal by one cycle apiece.

We do not suggest that optimizing basic blocks is unimportant. To the contrary, our beyond block methods, presented in the next chapter, and which we believe to be of critical importance, produce task systems which are formally the same as basic blocks of code containing many tasks (perhaps hundreds in some environments.) Thus methods of solving the basic block problem do take on an importance in that environment. Highest level lists, however, would seem to do well enough to make the effort of more elaborate strategies not worthwhile. In particular, we would suggest that research in microcode optimization be directed to issues beyond blocks, given these measurements.

We are not alone in having these opinions, despite the direction of the research to date. A short, refreshing paper by Graham Wood [WOOD78] appeared in the Proceedings of the 11th Annual Microprogramming Workshop in December, 1978. Wood does not report on the measurement of any strategies, but did code an optimizer, and concludes:

> On reading the referenced papers, one unavoidably gains the impression that the automatic packing of micro-operations into micro-instruction words is a critical area of research into which a great deal of effort must be invested before user-microprogrammable systems become feasible.
>
> Experience with the above program, however, has led the author to conclude that, in practical situations, very little scope exists in which to practise the art of optimal packing. Straight line segments of vertical micro-operations typically are not very long - sequences of more than about ten statements without a jump or a label are uncommon, and the data inter-dependency in such cases is liable to be great. Scope for optimisation increases with the degree of parallelism within the micro-instruction word, but, in that case, resource contention is reduced and packing them becomes easy anyway.
>
> In most practical situations, each of the four algorithms compared above would probably produce the same size of output. Where one does produce a smaller output than another, the difference is quite likely not to be significant compared to the possible speed-ups which could be generated by careful optimisation of the micro-program itself.

We maintain that, given efficient sequential code, it is the beyond block optimization which is difficult and long.

The Model

In order to simulate the task sets which would be produced by microcode, we produced randomly generated simplified MOPs and then formed the task set from them.

The parameters we used for a given run were:

#TASKS: the number of tasks per task set

#RES: the number of resources available

#RESU: the number of resources used by each task

#REGS: the number of registers available

#REGSR: the number of registers read per task

#REGSW: the number of registers written per task

#JOBS: the number of task sets generated

Thus we would, for each of the #TASKS MOPs, pick #RESU integers uniformly from the set [1,#RES] and we would say that the MOP used the one available unit of each picked resource. Note that we allow repetition, so the MOPs would use somewhat fewer than #RESU resources on average. Similarly, we pick #REGSR registers from the set [1,#REG] and we call that the set READ for that MOP; the same is done for the write registers, using #REGSW. We then used the algorithms from Chapter 5 to produce the data-precedence graph.

A setting of the parameters which figured prominently

in our experiments was:

#TASKS=40	#REGS=6
#RES=4	#REGSR=2
#RESU=1	#REGSW=1

which was felt to resemble the PUMA. (40 tasks was considered a realistic large set producable using the beyond basic block methods - see Chapter 8.)

Limitations of the Model

We intentionally kept our model simple; it seemed to us that more information about the practicalities of the situation could be gleaned from statistically sound runs with many parameter settings than fewer runs (due to the time usage of a more complex model) on a model which more closely resembled an actual machine.

In particular, we did not allow arbitrary amounts of each resource to be used, feeling instead that in most systems almost every resource is used entirely or not at all by any task. Thus, while the model might not be able to serve as part of an implementation of a system, the measures obtained from it would be likely to accurately reflect the characteristics of the system. We similarly did not allow general distributions of numbers of registers read or written, nor any correlation between the register and resource usages of neighboring MOPs, etc., again feeling that the simpler case accurately enough

resembles the actual situation. Our feelings along this line are somewhat borne out by the fact, as we shall discuss shortly, that the order among the strategies tested remained stable under rather dramatic shifts in the values of the parameters, and even the percentage differences from the theoretical optimal varied in an unsurprising fashion with the changes in the parameters.

The Strategies Tested

We tested twelve strategies for producing list schedules, and then considered the various methods of optimizing microcode suggested in the literature.

The twelve list scheduling strategies: (The numbers to the left are the keys for the strategy numbers in Figures 7.1 and 7.2.)

1. & 2. Random. Each task is assigned a random integer between 1 and 999.

4. Highest levels. $PRIORITY_4(t) = HEIGHT(t)$.

5. Criticality. $PRIORITY_5(t) = (1/(LATEST(t)-DEPTH(t)+1))$

6. Dense neighborhoods. As defined in Chapter 4, $PRIORITY_6(t)$ is the number of tasks which do not resource

conflict with t.

8. Smallest co-levels. $PRIORITY_8(t) = -DEPTH(t)$.

11. Reverse source order. PRIORITY (t) = -t. Thus the earlier a task, the higher its priority.

9. Dense neighborhood plus highest levels. $PRIORITY_{12}(t) = PRIORITY_6(t) + PRIORITY_4(t)$.

12. Dense neighborhood plus criticality. #9 with criticality replacing highest levels.

13. Dense neighborhood times highest levels. #9 with * replacing +.

7. Number of successors. The priority is the number of (not necessarily direct) successors of a task.

3. Coffman-Graham. Given tasks t and t' - if one precedes the other, it has higher priority. Otherwise, suppose that all of the successors of t and t' have been assigned priorities.

Then the higher priority is assigned to the one whose set of successors' priorities has higher dictionary ordering when sorted. If k tasks have no successors, they are randomly assigned priorities 1-k (we used reverse source order). If one removes transitive edges, this is optimal for UET Scheduling with 2 processors and no resources. We did not remove transitive edges. See [COFF72] for proof of optimality.

10. Resource bottleneck compensation. The tasks are first assigned priority according to method 4, highest levels. Each task is checked, in reverse source order, to see if it contributed to a Fernandez-Busell bottleneck. If so, then its priority is boosted by the number of extra cycles implied by the worst such bottleneck. Its predecessors' levels are then boosted, if necessary, to keep them at a higher level than the boosted task.

More formally:

```
PRIORITY = HEIGHT:
DO T=NUMBTASKS TO 1 BY -1;
  PRIORITY(T)=FIND-WORST-BOTTLENECK(T)+
                                PRIORITY(T);
  CALL BOOST-PREDS-PRIORITIES(T);
END;
```

where: FIND-WORST-BOTTLENECK(T) considers all intervals (i,j) and resources r

such that: U(T,r)=1 (assuming all values of U are 0 or 1.)

T is critically restrained to (i,j)

E (the extracycles for (i,j) and r) is > 0

FIND-WORST-BOTTLENECK(T) returns the largest E so found.

and: BOOST-PREDS-PRIORITIES(T) looks at all predecessors, T', of T.

if T' <=T then PRIORITY(T')=MAX(PRIORITY(T'),PRIORITY(T))

if T'< T then PRIORITY(T')=MAX(PRIORITY(T'),PRIORITY(T)+1).

Other Suggested Basic Block Methods

The following are sources containing suggested methods of basic block optimization, as obtained in Chapter 3. For each we explain what action, if any, was taken to test the method.

[DASG76] Appears to be strategy 11, above, once the complications of "sub-micro cycles" are removed. See also comments in [WOOD78] and [DASG78]. Strategy 11, as we shall see, does relatively poorly.

[RAMA74] Generally recognized in literature as very likely to produce non-optimal code even for simple examples; we did not test it.

[YAU74] When MOPs fall into equivalence classes of resource usage, it can be shown that this corresponds to strategy 7 above. We believe that, even though an occasional MOP may break the pattern on many machines, most basic block sets of MOPs will have that property, and thus the YAU strategy is essentially equivalent to strategy 7. Nonetheless, we did code the YAU method and tested it not only with the suggested weight (our $PRIORITY_7$), but with weights from 3, 4, and 10 above. The surprising results of those tests are reported later in this chapter.

[TOKO77] This method, once the concept of "micro-templetes" is removed, roughly corresponds to the YAU method, with $PRIORITY_4$ used to provide the weights, and which we have tested. We say roughly because tasks below the highest level are not considered

unless a tie exists among highest level tasks. Nonetheless, we feel that YAU with $PRIORITY_4$ is a good measure of the [TOKO77] method; when MOPs form resource equivalent classes, strategy 4 above represents it extremely well.

[TSUC74] We were unable to get a clear enough picture of the parallelization portion of this algorithm to code and test it.

The Experiments

We now report on the various experiments done to test the effectiveness of the above strategies. For each experiment, we first explain what was done in terms of the parameter settings and the strategies tested. We then draw conclusions, with references to the accompanying tables and figures.

Experiment 1. Change in number of tasks.

MODEL: (a) The PUMA-like model: #RES=4 #REGS=6 #REGSW=1 #RESU=1 #REGSR=2 #JOBS=200

(b) We varied #TASKS from 5 to 200, using the values 5, 10, 15, ..., 75, 80, 85; also 140 and 200.

STRATEGIES: For each task set size, and for each of the 200 task sets, we formed a list schedule for each of the 13 priority list methods. Thus 49,400 schedules were formed for this experiment.

CONCLUSIONS: (a) If we divide the strategies into groups as follows: GROUP 1 = {3,4,9,10,13}

GROUP 2 = {5,7,11,12}

GROUP 3 = {1,2,6,8}

then it is evident from Figure 7.1 and Table 7.1 that for any size task set, *any strategy in group 1 is better than any strategy in group 2 which is in turn better than any strategy in group 3.* Notice that group 1 is those strategies which are close to being highest level lists, group 3 is those which have no correlation to highest levels, whereas group 2 has some, but not a strong, correlation.

(b) *The group 1 strategies are closely bunched, and are all within 4% of optimal for #TASKS* $\leq$ *20, within 8% of optimal for #TASKS* $\leq$ *60, and within 12% of optimal for #TASKS* $\leq$ *200.* We remark that due to the nature of the lower bound, it is realistic to expect that

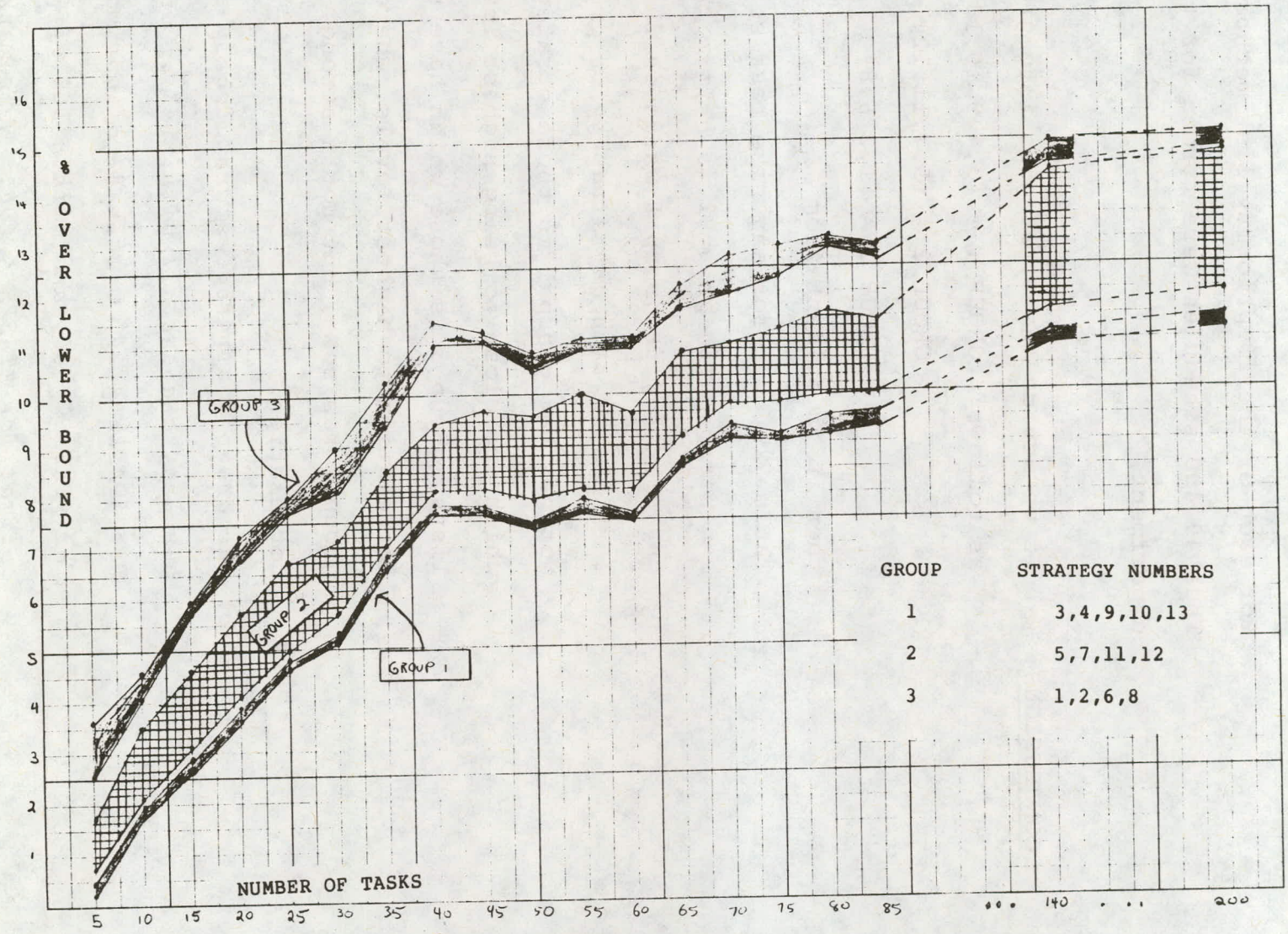


FIGURE 7.1 THE PERFORMANCE OF THE THREE LIST SCHEDULING STRATEGY GROUPS.

METHOD	NUMBER OF TASKS:							
	5	10	15	20	25	30	35	40
1	3.470	6.550	9.670	12.810	15.895	19.075	22.190	25.125
2	3.490	6.530	9.660	12.800	15.870	19.140	22.045	25.210
3	3.395	6.385	9.380	12.375	15.400	18.500	21.480	24.390
4	3.400	6.390	9.395	12.395	15.400	18.495	21.450	24.365
5	3.410	6.420	9.430	12.480	15.475	18.630	21.605	24.615
6	3.510	6.555	9.660	12.750	15.900	19.050	22.070	25.165
7	3.420	6.405	9.430	12.475	15.490	18.580	21.535	24.455
8	3.495	6.550	9.680	12.770	15.855	19.015	22.015	25.100
9	3.400	6.390	9.390	12.390	15.395	18.510	21.465	24.370
10	3.395	6.385	9.390	12.390	15.400	18.475	21.435	24.360
11	3.445	6.490	9.560	12.620	15.720	18.835	21.830	24.755
12	3.410	6.410	9.425	12.430	15.455	18.595	21.555	24.535
13	3.400	6.390	9.395	12.395	15.405	18.515	21.475	24.380
CRIT PATH	3.060	5.750	8.370	11.070	13.725	16.410	18.920	21.290
LOWER BND	3.385	6.270	9.135	11.935	14.720	17.570	20.115	22.615

Table 7.1a Lengths of list schedules for the PUMA-like model and varying size task sets.

METHOD	NUMBER OF TASK					
	45	50	55	60	65	70
1	28.185	31.235	34.300	37.680	40.590	44.040
2	28.190	31.320	34.295	37.665	40.525	43.980
3	27.310	30.365	33.305	36.480	39.290	42.730
4	27.285	30.350	33.250	36.450	39.270	42.695
5	27.475	30.555	33.415	36.790	39.600	43.000
6	28.180	31.330	34.300	37.625	40.455	43.880
7	27.405	30.555	33.455	36.655		
8	28.140	31.280	34.240	37.665	40.405	43.765
9	27.280	30.365	33.270	36.450	39.275	42.710
10	27.285	30.345	33.225	36.470	39.270	42.635
11	27.805	31.005	33.970	37.180	40.105	43.395
12	27.435	30.510	33.390	36.705	39.505	42.915
13	27.310	30.380	33.320	36.480	39.305	42.755
CRIT PATH	24.010	26.895	29.400	32.510	34.625	37.495
LOWER BND	25.340	28.275	30.865	33.910	36.175	39.065

TABLE 7.1b

METHOD	NUMBER OF TASKS:				
	75	80	85	140	200
1	46.590	49.940	53.090	87.63	124.66
2	46.480	49.875	53.035	87.42	124.77
3	45.040	48.305	51.510	84.73	120.79
4	44.990	48.270	51.425	84.68	120.66
5	45.370	48.655	51.800	85.32	121.60
6	46.390	49.970	53.000	87.32	124.42
7					
8	46.320	49.940	52.945	87.27	124.37
9	45.000	48.325	51.465	84.75	120.79
10	45.005	48.240	51.400	84.58	120.60
11	45.930	49.295	52.275	86.31	122.96
12	45.315	48.575	51.710	85.08	121.32
13	45.025	48.375	51.485	84.83	120.84
CRIT PATH	39.675	42.605	45.250	74.21	106.19
LOWER BND	41.235	44.170	47.005	76.22	108.38

TABLE 7.1c

the actual percentages are much closer to zero.

(c) As shown on Table 7.2, *the times to form the graph, calculate the priority and schedule are very short.* In particular, all priorities which do not involve the (obviously $0(|T|^2)$) dense neighborhoods are $0(|T|)$. The number of successors is only apparently $0(|T|)$, however, due to the use of the CDC bit count instruction. The lower bound calculation, which is $0(|T|^2)$, deserves to be counted in as part of strategy 10. Using highest levels, (strategy 4,) *a set of 40 MOPs, having been read in along with the READ and WRITE sets, and USAGE functions, would seem to be schedulable on the CDC 6600 in about 142 ms.* (And that measurement includes 4 system calls to do the timing!)

Experiment 2. Changing other parameters, keeping #TASKS fixed at 40.

MODEL: All models have #TASKS=40. We test models 1-9, with parameters as listed on Table 7.3. #JOBS again is set to 200.

TIME TO	NUMBER OF TASKS					
	20	40	60	80	140	200
FORM PRIORITY 1	0.14	0.12	0.11	0.11	0.11	0.11
2	0.14	0.12	0.11	0.11	0.11	0.11
3	1.51	1.65	1.70	1.71	1.73	1.75
4	0.06	0.04	0.04	0.04	0.03	0.03
5	0.07	0.06	0.05	0.05	0.04	0.04
6	0.67	1.26	1.86	2.46	4.11	5.87
7	0.17	0.17	0.17			
8	0.06	0.04	0.04	0.03	0.03	0.03
9	0.71	1.30	1.89	2.49	4.13	5.90
10	0.66	0.66	0.67	0.70	0.78	0.89
11	0.05	0.03	0.03	0.03	0.02	0.02
12	0.73	1.31	1.90	2.50	4.15	5.92
13	0.71	1.30	1.89	2.49	4.13	5.90
SCHEDULE (GIVEN PRI)	1.18	1.15	1.13	1.14	1.15	1.14
FORM PRECEDENCE REL	2.21	2.35	2.40	2.42	2.46	2.48
CALCULATE LOWER BND	1.17	1.37	1.64	1.87	2.56	3.35

ALL TIMES ARE MILLISECONDS PER TASK

TABLE 7.2 TIMES TO FORM DATA-PRECEDENCE GRAPH, PRIORITY LISTS, LOWER BOUNDS AND SCHEDULES FOR VARIOUS TASK SET SIZES.

STRATEGIES: For each parameter setting and for each of the 200 jobs, we again tested all 13 list scheduling strategies. Thus 23,400 schedules were formed for this experiment.

CONCLUSIONS: (a) *Except as noted below, the order of the strategies was essentially what it had been for the full range of task sizes.* (See Tables 7.3, 7.4, and 7.5). The averages of the rankings were remarkably similar to that in Experiment 1.

(b) *For most of the models, the best strategies were again close to the bound.* Exceptions occurred, again, in places where we would predict the failure of the bound rather than the scheduling.

(c) For models with very short critical paths, that is, *for very wide graphs, the number of successors was a strikingly better priority measure than it had been.* See model 7, especially. We have no explanation for this.

(d) Other strategies, notably 5 and 12, seemed to do better for some settings of the parameters than they had previously, again for reasons unknown to us.

Paremeter \ Model Number	1	2	3	4	5	6	7	8	9
# RES	4	3	2	11	3	11	4	4	4
# RESU	1	1	1	2	1	2	1	1	1
# REGS	3	3	3	3	6	6	12	12	12
# REGSR	2	2	2	2	2	2	1	1	2
# REGSW	1	1	1	1	1	1	0* 1*	1	1

* With equal probability.

TABLE 7.3. THE VARYING PARAMETERS FOR EXPERIMENT 2, WITH NUMBER OF TASKS FIXED AT 40.

METHOD	MODEL NUMBER: (FROM TABLE 7.3)								
	1	2	3	4	5	6	7	8	9
1	32.300	32.840	34.305	32.905	26.615	26.455	14.890	17.650	19.940
2	32.295	32.805	34.345	32.930	26.665	26.280	14.810	17.700	19.960
3	32.155	32.605	34.085	32.680	25.760	25.550	13.270	15.705	18.220
4	32.145	32.600	34.060	32.690	25.720	25.485	13.250	15.585	18.145
5	32.150	32.625	34.105	32.715	25.945	25.695	13.580	16.135	18.605
6	32.255	32.845	34.250	32.890	26.655	26.400	13.700	16.345	18.995
7	32.180	32.640	34.075	32.740	25.760	25.570	13.225	15.660	18.315
8	32.250	32.815	34.290	32.880	26.525	26.310	14.360	16.875	19.390
9	32.145	32.615	34.055	32.760	25.745	25.780	13.250	15.595	18.150
10	32.145	32.590	34.055	32.690	25.645	25.420	13.260	15.660	18.145
11	32.245	32.720	34.210	32.835	26.175	26.025	13.700	16.290	18.965
12	32.155	32.640	34.095	32.740	25.880	25.685	13.370	16.350	18.565
13	32.145	32.615	34.055	32.840	25.760	26.090	13.265	15.635	18.200
CRIT PATH	30.425	30.365	30.435	30.610	21.690	21.720	6.085	10.680	14.140
LOWER BND	31.345	31.425	32.080	31.555	23.480	22.930	13.185	14.160	16.355

TABLE 7.4 LENGTHS OF LIST SCHEDULES FOR THE MODELS OF TABLE 7.3

List Method	Model Number (from Table 7.3)									Avg Rank These Models	Avg Rank Exp. 1
	1	2	3	4	5	6	7	8	9		
1	13	12	12	12	11	13	13	12	12	12.22	11.75
2	12	10	13	13	13	10	12	13	13	12.11	11.50
3	6	3	6	1	4	3	6	6	5	4.44	2.83
4	1	2	4	2	2	2	2	1	1	1.89	2.50
5	5	6	8	4	8	6	8	7	8	6.67	7.50
6	11	13	10	11	12	12	9	9	10	10.78	11.67
7	8	7	5	5	4	4	1	4	6	4.89	6.75
8	10	11	11	10	10	11	11	11	11	10.67	10.75
9	1	4	1	7	3	7	2	2	3	3.33	2.33
10	1	1	1	2	1	1	4	4	1	1.78	1.50
11	9	9	9	8	9	8	9	8	9	8.67	9.00
12	6	7	7	5	7	5	7	10	7	6.78	6.50
13	1	4	1	9	4	9	5	3	4	4.44	4.25

TABLE 7.5 RANKINGS OF PERFORMANCES OF LIST SCHEDULES FOR THE MODELS OF TABLE 7.3.

(e) We would guess that the relatively poor performance of the Coffman-Graham Algorithm (strategy 3) for some of the models is caused by transitive edges, which their algorithm mandates the removal of.

Experiment 3. Already suggested optimization methods.

MODEL: We only tested YAU's strategy on graphs in which the equal edges were considered to be strict edges; this was the model for which it was suggested, and the coding was considerably less difficult. The #TASKS setting was kept at 40, and 7 sets of 200 jobs were run. Since models with equivalence classes of resource usages (i.e. with #RESU=1) will cause the YAU methods to produce list schedules, only one model with that property was run. The models were:

Model Y1: #RES=3 #REGS=6 #REGSW=1
#RESU=1 #REGSR=2 #JOBS=200

Model Y2: #RES:11, #RESU=2, otherwise like 1.
(Run twice)

Model Y3: #RES=24, #RESU=3, otherwise like 1.

Model Y4: #RES=24 #REGS=6 #REGSW=0 or 1
#RESU=3 #REGSR=1 (with 0 or 1 equally likely)

Model Y5: #RES=24, #RESU=6, otherwise like 1.

Model Y6: #RES=24 #REGS=6 #REGSW=0 or 1
#RESU=6 #REGSR=1 (each equally likely)

Model 1 was used to verify that YAU schedules were indeed list schedules with PRIORITY=WEIGHT when #RESU=1. Models 4 and 6 were chosen to have a short average critical path. This would give the YAU method more to do, with many more tasks data ready at each cycle. Thus we guessed its performance would be best in such a situation, at the cost of much more time used. The resource usages of 2 out of 11 and 3 out of 24 were carefully chosen to have the property that two MOPs will clash with probability about .33, just as was the case with 1 out of 3. We would expect the list schedules to be about the same in all three cases, but we expected the YAU method to be progressively better as the number of resources used increased, since the interdependency among them also increased.

STRATEGIES: The strategy of [YAU74] was tested on each of the above models with WEIGHT equal to list scheduling priority methods 3, 4, 7, and 10. Thus an indication was also obtained of the effectiveness of the methods of [TOKO77].

For comparison, list schedules were also formed for priorities 3, 4, 7, and 10. This was our first test of these methods in the absence of equal edges. In total, 11,200 blocks were optimized.

CONCLUSIONS: (a) Although it might be reasonable to expect the YAU method to almost always be an improvement over simple list scheduling with the same weight, *the YAU method was slightly but uniformly worse than simple list scheduling in the most realistic models (2,3,4).* Only when we used the rather artificial models (5,6), in which the resource interrelationships among MOPs are very complex, were we able to get improvement from the added complexity of this kind of optimization. Evidently it is generally worse to try to schedule more of the less critical tasks than fewer more important tasks. The results of Experiment 3 are summarized in Table 7.6.

(b) Evidently due to the removal of the equal edges, $PRIORITY_7$, *the number of successors, was as good as the group 1 methods.*

Experiment 4. Breaking priority ties via source order.

METHOD	MODEL NUMBER						
	Y1	Y2	Y2	Y3	Y4	Y5	Y6
LIST (3)	29.265	29.115	29.250	29.200	15.625	34.955	24.405
YAU (3)	29.265	29.130	29.275	29.200	15.930	34.925	24.275
LIST (4)	29.190	29.065	29.185	29.105	15.590	34.850	24.250
YAU (4)	29.190	29.080	29.200	29.120	15.735	34.810	24.145
LIST (7)	29.150	29.075	29.220	29.090	15.540	34.745	24.145
YAU (7)	29.150	29.095	29.250	29.100	15.600	34.710	23.845
LIST (10)	29.185	29.075	29.190	29.105	15.575	34.840	24.200
YAU (10)	29.185	29.090	29.205	29.120	15.695	34.790	24.035
CRIT PATH	26.895	27.065	27.100	27.040	12.615	27.060	12.190
LOWER BND	28.050	27.980	27.980	27.860	13.690	28.380	15.690

LIST (N) MEANS A LIST SCHEDULE FORMED USING PRIORITY METHOD N.

YAU (N) MEANS A SCHEDULE FORMED USING YAU'S METHOD WITH PRIORITY METHOD N AS A WEIGHT.

TABLE 7.6 LENGTHS OF SCHEDULES PRODUCED BY YAU'S ALGORITHM USING VARIOUS WEIGHTS; LIST SCHEDULES FOR SAME WEIGHTS GIVEN FOR COMPARISON.

MODEL: the PUMA-like model with #TASKS=40

#RES=4 #REGS=6 #REGSW=1

#RESU=1 #REGSR=2 #JOBS=200

STRATEGIES: List schedules were formed using the 13 strategies, then, using the same seed, and thus generating the same set, list schedules were formed with eleven of the strategies altered so that priority ties were broken in favor of the earlier source task. Method 3 does not produce ties, and we had already coded it in such a way that if two tasks have identical sets of successors, higher priority went to the earlier source task. Method 11 is source order, so is unaffected. The other methods had ties broken in scheduling in rather arbitrary fashion. The test was done on four sets of 200 jobs, so 20,800 schedules were formed.

CONCLUSIONS: *Among the strategies that did well, no significant difference was noticeable.* Many sets of schedules were longer and many shorter. Some of the poorer methods (e.g. 6, which produces very many ties.) did improve a bit, this seemed due to the imposition of some good strategy upon methods that were little better than random.

Experiment 5. Test of statistical significance.

MODEL: As in Experiment 4.

STRATEGIES: Methods 1-13 of list scheduling, ties broken arbitrarily. The 200 jobs were run five times in the previous experiments, plus once more for this; all runs used a different seed for the random number generator, thus different task sets were produced.

CONCLUSION *The small differences detected among even most closely ranked strategies were statistically significant*. We used the well known Student's t test as follows. The total schedule length of 200 schedules is approximately normal, being the sum of a large number of samples from a multinomial distribution. Given any two strategies, then, we have six pairs of totals, one for each run. The paired elements are not independent, being schedules for the same graphs, so we cannot test for the difference in the two means, as is usually done with the Student t distribution. However, the difference between them should still be normal, and we can use that test to see whether the difference is zero.

The two best strategies, 4 and 10, yielded pairs with differences of 10, 3, 2, 1, 5, and 6 for a mean of 4.5. The Student t test rejects the hypothesis that these could have come from a normal population with mean zero at the .01 level. Indeed, not until the average difference gets down to 1.92 does the test fail to reject at the .05 level with the sample variance. We may thus be confident that strategy 10 is slightly better than 4 for this model, despite the fact that their difference was about .8% of the total schedule length.

Other pairs of strategies were almost all farther apart than 10 and 4, and the test showed significance to an even greater degree for most pairs. A few pairs of strategies from within the same groups did fail to reject at this level; however, beyond strategy groups (that is, groups 1, 2, and 3 from Experiment 1) the differences were relatively massive and the mean=0 hypothesis was overwhelmingly rejected.

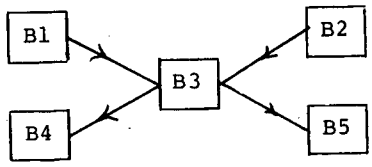
8. A Unified Approach to Interblock Optimization

In this chapter we will present a method of moving microoperations past jumps into basic blocks other than the one they started in. A small amount of experience hand optimizing convinces one that to optimize well, such movements are necessary. In particular, blocks tend to be short in microcode, and a MOP may very well be movable from a block in which it takes up a cycle to a block in which it can be done for free.

It is not difficult to produce a catalog of the types of allowable motions, see Figure 8.1. When we hand optimize microcode, we generally produce horizontal code which we then iteratively improve, using motions like those in Figure 8.1. When we first considered automatic optimization, we imagined that the same course would be followed.

We were not particularly optimistic about the effectiveness of this approach and, indeed, initial investigation bore out our feelings. This seemed a familiar situation; one is given a starting position (in this case, the individual blocks each separately optimized), and a very large tree of possible changes. One then wants to travel the tree to a final position which is more desirable (in this case, with code speed and possibly space used somewhat comparable to hand produced code). When these iterative methods don't work well, they tend to fail for two strongly related reasons:

Given blocks B1,B2,B3,B4,B5 with the following flow graph:



It may be profitable to move a MOP as follows, if it can be scheduled "for free" in at least one of the target blocks:

(a) From B3 to (B4 and B5) or to (B1 or B2)

(b) From (B4 and B5) or from (B1 or B2) to B3 if an identical MOP appears in both source blocks

(c) From B3 to B4 or from B4 to B3 if all registers written by the MOP are dead in B5.

FIGURE 8.1. CATALOG OF INTERBLOCK MOP MOTIONS.

(a) Width of tree. There is a vast number of choices at each position, and the cost of moving to and evaluating the next position is large.

(b) Depth of tree. Real improvement can only be found at a great distance from the starting position. Worse yet, all paths to the improved positions start by going through several positions in which the situation appears worsened.

Working through PUMA code by hand, it becomes obvious that both of these problems would defeat an attempt to produce code nearly as good as that produced by hand. What was needed, then, was a more *global view* of all of the code being optimized. After many false starts, we were able to come up with a radically different approach which provides precisely such a view; when put to the test, it efficiently (though somewhat complicatedly) generated code which compares very well with hand optimized versions of extremely complex code.

Our description of the method will again be done both formally and informally. We will first present an undetailed and hopefully clear description, starting with how code with relatively simple flow structures would be handled and explaining how the methods generalize to code with an arbitrary flow of control. We then present the algorithm in more detail, as follows:

(a) The definitions needed are given formally.

(b) The calling sequence for the algorithm is presented, uncommented so that the flow of control may be seen.

(c) The calling sequence is repeated, with comments designed to give a clear view of what each routine accomplishes.

(d) The algorithms are presented carefully, using Pidgin PL/1 and English description when we wish to avoid getting too caught up in details.

Finally, examples are presented which follow the algorithms in some detail.

Our Method of Interblock Optimization

Our approach to beyond-basic-block optimizing is centered about embedding the information necessary to make a large set of these optimizations into one very large data-precedence graph. The graph includes jumps and loops as ordinary tasks, and uses the precedence relation to constrain or allow the motion of MOPs past jumps. This permits us to again use the techniques and ideas of scheduling theory to great advantage, and the above mentioned problems of tree searches are both dealt with rather decisively.

To begin a description of our method, we note an unfortunate, but unavoidable, fact about this kind of optimization: one is often faced with the prospect of shortening one branch from a condition at the potential expense of our ability to shorten the other. The hand optimizer is apt to be keenly aware of this, particularly since most conditions seem obviously and heavily weighted in one direction, and shortening the main branch seems very natural. In any event, we will assume that at each conditional jump, we have an estimate of the probabilities of each branch being taken. This may be obtained either by programmer guess (some people, of course, might dub this the "guaranteed incorrect method"), by simulation, by running the vertical code, modified to count branches (after all, the vertical code is perfectly valid, if inefficient, hori-

zontal code, and can run as is), or by some other heuristic method.

We now outline our method, which is given more formally later in the section. For the moment we deal with sections of code containing jumps forward, but not back — that is, loop free code. Our first step is to follow one path of the code from the entrance to the exit, choosing a path which seems most likely to be followed in running typical data. Places where two branches join are, momentarily, ignored (that is, the join will be remade later, when possible; when not, the code is duplicated. This will be discussed in more detail shortly.) We now build a data-precedence graph consisting of all tasks (even the conditional jumps) on this path. Except for edges going to and from the jumps, the graph building precedes as before, see Figure 5.2. Jumps tend to read the contents of certain registers, and so must follow the tasks which wrote those registers. Edges from the jumps are a more complicated matter; namely, those tasks which would destroy registers read in the branch not under consideration must not be permitted to go above the jump, and so we drew an edge from the jump to all such tasks.

We have now encoded our MOP motion into a data precedence graph, and the resource conflict rules are what they would have been had this been one big basic block. Thus all the pieces are there to allow us to schedule, just as we scheduled basic blocks, and we do exactly that.

Having obtained a schedule, certain adjustments must be made to legalize some of the motions implied by the schedule. For example, suppose a task moved below a jump which it was previously above — such a task would generally have to be duplicated into the branch that we haven't considered. We look for all such inverted task-jump pairs, and duplicate tasks where appropriate. Furthermore, other paths will need to be rejoined to the newly scheduled path, but there may be no point below which are those and only those tasks which we would like to rejoin to. Thus we rejoin as far up as we possibly can, and all those tasks which we have not rejoined to are duplicated into the joining branch before the splice. We mention that some of the duplicated tasks will be conditional jumps — this presents no particular problem, but care needs to be taken that the right sequence of instructions is followed.

Before actually duplicating tasks, we look at the schedule produced and determine whether some tasks can be moved down into holes in the schedule without lengthening it. List scheduling puts tasks into early cycles, and often space can be recovered by undoing some arbitrary choices made by the scheduler. Thus, inverted jump-task pairs may often be uninverted, and tasks which are above the legal rejoin for a path may be placed below it.

In carrying out this space recovery phase, we suggest precisely the sort of tree search procedure that we previously avoided. This is due to the fact that we regard space

recovery as something of a "luxury item", and we are glad to grab whichever savings are easily done, but are generally unwilling to invest a great amount of time looking very far or setting up an elaborate structure to carry this out well. Furthermore, in contrast to speed optimization, it seems that most of the space recovery lies right near the surface. We remark, however, that in a space-critical environment, elaborate reparallelizations and careful measurements of the time/space tradeoffs may be most worthwhile.

Scheduling the Remainder of the Path

We are now in a position to schedule the remaining tasks, i.e. those off the main path. From among the remaining tasks we again select one well traveled path to optimize next, again ignoring all joins. Just as before, we form a data-precedence graph, using the conditionals as gates to control the code motion, and we form a priority list. However, instead of directly scheduling, we pull data-ready tasks off the list, and try to move them up into holes in the already scheduled path that this path splits from. When this process is exhausted, we schedule the remaining tasks as before, complete with space recovery and task duplication.

When this has finished, we choose another path and repeat the process, continuing this way until all tasks have been optimized. We then hunt for duplicate lines of code to eliminate, and we are finished optimizing.

Code Containing a Single Loop

We will shortly present a method of optimizing any microcode, including that containing back branches. We will assume, for this discussion, that we are dealing with reducible flow graphs, although node-splitting (actually duplicating blocks as suggested in the split node flow graph) is appropriate here, since one would not expect to run into many irreducible flow graphs, and since we have already seen that we are willing to duplicate some code.

We first deal with the situation of a single loop with a single back branch contained in code which otherwise has no back branches. We first optimize the loop itself, using the basic block methods above. (We note that all normal compiler optimizations are assumed to have been done during production of the vertical code, thus moving tasks out of the loop is of no concern to us.) Having optimized the loop, we now schedule the rest of the tasks. We form a data precedence graph as before from the tasks not in the loop, but our graph now contains a new node called L, say, which will represent the entire loop. Now, some of the other tasks, due to data constraints, will have to precede L, and some follow; this information is encoded in the graph in the usual way, and we are ready to begin scheduling. Until L is data-ready, we proceed in the usual fashion, as if this were a basic block. When L is data-ready and the next task to be

considered for scheduling, however, special considerations arise. We assume that some tasks have already been scheduled in the current cycle. We then place L in the current cycle only if all of the tasks already in the cycle are loop invariant, and only if all of them may be fitted into the loop's internal schedule without lengthening it. If those conditions are not met, we act as though L had a resource conflict with the tasks already placed. Eventually, L will be placed, if only because it must get to the top of the priority list at the start of some cycle. Once L is placed, we continue to try to schedule tasks in that cycle, but now the situation is reversed. That is, tasks are scheduled with L only if they are loop invariant and can fit into holes remaining in L's internal schedule. When scheduling is completed, we stop treating the loop as one task, and write it out in full, according to its already produced schedule, filling in holes with tasks scheduled in the same cycle, as required. We then duplicate tasks and recover space as before.

In summary then, the method outlined above permits the motion of tasks into, ahead of, and beyond loops, with the motion again controlled by (hopefully) appropriate scheduling heuristics. We point out that several details have been left out of this discussion but are covered more carefully in the accompanying algorithm.

Code with a General Flow Structure

We now consider code with an arbitrary reducible flow graph. Our method can generally be described as a recursive descent; that is, given the flow graph, we apply the process we are describing to each outermost loop, optimizing each separately and with no regard for the optimization of the others. Each such loop, naturally, contains its own outermost loops (since in reducible flow graphs two loops are disjoint or else one completely contains the other), and we attempt to optimize each of those. Eventually, an outermost loop must itself contain no loops, and we optimize it using the techniques just explored for loop-free code.

Having optimized all of the innermost loops, we may work our way outward. At each stage, we have loop-free code except for some disjoint, already optimized loops. This situation is handled just as the code containing a single loop was. That is, each loop is regarded as one task, and the data-precedence graph is formed and scheduling done so as to allow motion beyond, above and within loops. The only special consideration here is that one loop-task is never permitted to enter another. Thus we pick a path, schedule, duplicate tasks where necessary, pick another path, and so on, until all tasks are scheduled. Eventually, we will have worked our way outward to the outermost code — when that is scheduled, we are done.

Definitions for the Interblock Optimization Problem

1. We have a set of microoperations,

   $M = \{m_1, m_2, \cdots, m_s\}$, and a set of *exit nodes*

   $E = \{e_1, e_2, \cdots, e_n\}$.

2. The following functions are given:

   READ, WRITE, U

   and the sets A & R, all as defined for intrablock optimization.

3.a. We have the pair of functions:

   TRUEJUMP, FALSEJUMP: $M \rightarrow (M \cup E)$

   and when $\text{TRUEJUMP}(m_i) \neq \text{FALSEJUMP}(m_i)$ we say that m_i is a *conditional jump*. We further call $\text{TRUEJUMP}(m_i)$ and $\text{FALSEJUMP}(m_i)$ *targets* of m_i.

   b. We also have the function

   LVTOP: E → subsets of A, the set of registers,

   where $a_k \in \text{LVTOP}(e_i)$ means that the exit e_i from the code we are optimizing has a reachable use of a_k.

   We will, in the course of optimizing, extend LVTOP to the set of basic blocks of microinstructions.

4. We now define a *microinstruction* as a set of MOPs with the properties that:

   a. No resource is overused (as defined earlier)

   b. No more than one conditional jump appears in a micro-instruction.

5. We will be searching for a *parallelization* of M, which will be the gathering of MOPs into microinstructions. The microinstructions will not generally be disjoint, as they had been for intrablock code, but we will again require that they be exhaustive.

6. During our algorithm, we will work with P, a sequence of microinstructions, $P = \{P_1, P_2, \cdots, P_k\}$, which we will eventually convert into our desired parallelization.

   We will start off with P = M, that is, each partition will be a single MOP. We will define

   TRUEJUMP and FALSEJUMP: $P \rightarrow (P \cup E)$

   in the obvious way, given the definitions on M, and some microinstructions will be referred to as conditional jumps. The sets $READ(P_i)$ and $WRITE(P_i)$ will also be obvious extensions of the definitions on M, that is:

   $READ(P_i)$: $\bigcup_{m_j \in P_i} READ(m_j)$, and similarly for WRITE.

7. As we parallelize we will redefine the targets of the P_i's as the order of the P's changes; as m_j's are added to P_i's, the READ and WRITE functions will change.

8. We gather the P's into disjoint, exhaustive sets, $B = \{B_1, B_2, \cdots, B_t\}$, with, for each B_i , $B_i = \{P'_j, \cdots, P'_k\}$. We call the B_i's *basic blocks* (or just BLOCKS) if all of the following are true:

a. The P_i's form a chain $P'_j < \ldots < P'_k$ where $P'_m < P'_\ell$ means that P'_ℓ is a target of P'_m. We call P'_j the *initial microinstruction* and P'_k the *terminal microinstruction* of the block.

b. either i. $j = 1$

or ii. P_j is the target of a conditional jump

or iii. P'_j is the target of two or more microinstructions.

c. either i. P'_k is a conditional jump

or ii. the target of P'_k is the target of another microinstruction

or iii. the target of P'_k is an exit node.

d. There is no other partitioning of P into t or more subsets with the above conditions being true.

9. We have a relation $<_B$ on B, defined as

$B_i <_B B_j$ if the initial microinstruction of B_j is a target of the terminal microinstruction of B_i.

We call the graph determined by $<_B$ the flowgraph and we call the element of B containing P_1 the *entrance block*. We use the terms *SUCCESSOR* and *PREDECESSOR* in the ordinary way. When the context is clear, we will write (B_i, B_j) for the arc corresponding to $B_i <_B B_j$.

10. Given a flowgraph, we define *reducible flowgraph*, *dominator*, *backedge*, and *natural loop* as in Aho-Ullman [AH77], Chapter 12.

Briefly, we say:

a. *Block B_i dominates block B_j* if every path from the entrance block to B_j goes through B_i.

b. If block B_i dominates block B_j and $B_j < B_i$, then we call the arc (B_j,B_i) a *back edge*.

c. For each back edge (B_j,B_i), we form a subset of B, called the *natural loop* of the backedge, which is all nodes that can reach B_j without going through B_i, plus B_i itself.

d. A flowgraph is *reducible* if the edges are partionable into two disjoint classes:

   i. forward edges, which form a dag in which every block is reachable from the initial block, and

   ii. backedges, as above.

   The key properties of reducible flowgraphs, for our purposes, are that:

   i. Each loop has only one entrance — the B_i in c. above

   ii. The loops are nested, that is, two loops are either disjoint, or one is included in the other; two otherwise disjoint loops may share a header, however.

Calling Sequence

```
OPTIMIZE: PROC(M,E,LVTOP,READ,WRITE,U,TRUEJUMP,FALSEJUMP);
     P = M;
     CALL MAKE_BASIC_BLOCKS;
     CALL MAKE_FLOW_GRAPH;
     CALL LIVETOP_ANALYSIS;
     CALL ASSIGN_JUMP_PROBABILITIES;
     CALL SCHEDULE(B)
END OPTIMIZE;

SCHEDULE: PROC(L)
     DO FOR EACH L ∈ OUTER_LOOPS(L);
          CALL SCHEDULE(L_i);
     END;

     DO WHILE (UNOPTIMIZED BLOCKS REMAIN IN  L);
          CALL PICK_A_PATH;
          CALL MAKE_TASKS;
          CALL MAKE_DATA_PRECEDENCE_GRAPH;
          CALL FORM_PRIORITY_LIST;
          CALL TASK_LIFT;
          CALL SCHEDULE_PATH;
          CALL RECOVER_SPACE;
          CALL CUT_AND_PASTE;
     END;
END SCHEDULE;
```

Comments on the Optimizing Routines

```
OPTIMIZE: PROC(M,E,LVTOP,READ,WRITE,U,TRUEJUMP,FALSEJUMP);
     P = M;
     /* We start with the set of microinstructions equal to
        the source MOPs and with the jump functions also
        identified with the functions on M.  */
     CALL MAKE_BASIC_BLOCKS;
     /* Produces B = {B1,B2,B3,...,Bt}, the set of basic
                     blocks of P */
     CALL MAKE_FLOW_GRAPH;
          /* MAKE_FLOW_GRAPH:
          1.  Builds <B , the flowgraph  for B.
          2.  Determine  if <B  is reducible; if not, uses
              node-splitting to revise P & B so that a
              revised <B is reducible. Duplicates loop
              headers where needed to assure  loops are
              disjoint or nested.
          3.  Builds the set LOOPS, which is originally a
              set of subsets of B; each element of LOOPS is
              a natural loop; note that we include B in LOOPS
              as it if were a loop.
          4.  Builds the function: OUTER_LOOPS: LOOPS → subsets
              of LOOPS, where OUTER_LOOPS(Li) is the set of
              loops properly contained in Li but no smaller loop.
```

5. Builds EXITS, where EXITS(L_i), $L_i \in$ LOOPS, is the set of names which follow blocks in L_i on $<_B$ but which aren't in L_i.
6. Changes LOOPS so that within each $L_i \in$ LOOPS, all blocks contained in some loop interior to L_i are removed and the set OUTER_LOOPS(L_i) is added. Thus, L_i contains only names of objects contained in no smaller loop.
7. Builds $<_{L_i}$ for each $L_i \in$ LOOPS. $<_{L_i}$ is $<_B$ restricted to $L_i \cup$ EXITS(L_i), except that each arc (B_i,B_j) where B_j is the entrance of some loop in L_i is changed to (B_i,L_m), with $L_m \in$ OUTER_LOOP(L_i). Thus $<_{L_i}$ is a subset of $L_i \times (L_i \cup$ EXITS(L_i)). */

```
CALL LIVETOP_ANALYSIS;
/* For each B_i ∈ B , we perform a live variable analysis,
   to produce LVTOP(B_i), a subset of A, the set of
   registers. a_k ∈ LVTOP(B_i) means that there is some
   path through the source code, starting at the initial
   microinstruction of B_i , which reaches a use of
   register a_k before a_k is written. Recall that part
   of our input is the set of LVTOP's for the exit nodes.
   LVTOP is also extended to LOOPS; that is LVTOP(L)
   = LVTOP(B_i) where B_i is the entrance node of L.
   We mark LVTOP_VALID = true for each block. */
```

```
        CALL ASSIGN_JUMP_PROBABILITIES;
        /* Forms two functions: ARC_PROB and EXPECT.
        ARC_PROB((B_i,B_j)) where B_i <_L B_j,is the estimated
          probability that if control flows to B_i it will next
          flow to B_j.
        EXPECT(B_i)  is the probability, calculated from ARC_PROB
          and <_L that, given that the entrance node of L is
          reached, B_i is traversed before an exit node.
        We will have several suggested methods for
        estimating ARC_PROB.  The estimate need not be very
        tight, but occasionally will influence which sections
        of code are shortened at the expense of others. */
        CALL SCHEDULE(B);
    END OPTIMIZE;

    SCHEDULE: PROC(L);
        DO FOR EACH L_i ∈ OUTER_LOOPS(L);
            CALL SCHEDULE(L_i);
        END;
        DO WHILE(UNOPTIMIZED BLOCKS REMAIN IN L);
            CALL PICK-A-PATH;
           /*  PICK-A-PATH selects a set of blocks in L.
            These blocks are called PATH={B_1,B_2,...,B_v} and have
            the property that on <_L , B_1 < B_2 < ... < B_v.
            The selection is  made in such a way that blocks
            which are likely to be travelled are generally
            preferred over less traveled ones, as determined by
                                                    EXPECT. */
```

```
CALL MAKE_TASKS;
/* We form a set of tasks T = {t1,t2,...,tw}. Each loop
   Li ∈ OUTER_LOOPS(L) which is in PATH is considered
   one task ℓi , and each element P ∈ Bi ∈ PATH, where
   Bi is a block, is one task. These latter elements
   will, in fact, be microinstructions containing
   single MOPs, since no parallelization will yet have
   been done to them. */

CALL MAKE_DATA_PRECEDENCE_GRAPH;
/* We now must specify the successors and predecessors
   (equal and strict) for each element in T. We have
   the register sets READ(ti) and WRITE(ti) when ti
   is an ordinary microinstruction, just as we did in
   the case of intrablock scheduling. When our tasks
   are microinstructions, then, our criteria for
   the precedence relation are what they were
   before.
```

If a task t_i represents a microinstruction which was a conditional jump, however, we have an additional set of registers to concern ourselves with. There are two target nodes on $<_L$ for the jump from the block ending with t_i. Let B_j be the one which is not the immediately following node on our path (though B_j may very well be further down the path). The next t_k which writes one of the registers in LVTOP(B_j) and which is on the path down from t_i must be a strict successor of t_i.

When t_i is a task representing one of the loops (which have already been optimized), the situation is more complex, and the details are left to a formal description of the algorithm. Essentially, though, we note those tasks which read or write registers which are read or written within the loop or are live at one of the exits from it, and draw an equal arc between such tasks and the loop task to prohibit the invalidation of data; more careful precedence considerations are used during scheduling.*/

CALL FORM_PRIORITY_LISTS;

/* We use any appropriate heuristic, as discussed in the chapter on intrablock scheduling. This routine assigns a value PRI(t_k) to each $t_i \in T$. */

CALL TASK_LIFT;

/* TASK_LIFT attempts to move tasks which have no predecessors up from the path being considered to some already optimized blocks in L which have jumps to the beginning of the path being considered.

Generally, TASK_LIFT looks first at the blocks on L which have jumps to the path being considered. Unless the probability that control flows to the path via arcs from already optimized blocks in L is greater than some threshold (say 50%) no LIFTing is attempted. A task, t_i, is eligible for LIFTing in priority order only if it meets *all* of the following criteria:

(a) t_i has no predecessors on the path, or all of its predecessors have been LIFTed.

(b) t_i is not a conditional jump or a loop task.

(c) t_i comes from a block which is above the first block (other than the path entrance) to which there is an arc from a block off the path.

as tasks are LIFTed, their successors might satisfy (a) above, and need to be considered when they do.

When a task is eligible for lifting, we call a routine, TRY_LIFT, once for each arc from an already optimized block in L to our path entrance. TRY_LIFT(P_i,B_j) will succeed (and will report the place(s) in blocks B_j and above on L to insert a t_i into the already formed parallelization) if t_i may be so inserted without added cycles being generated; otherwise, TRY_LIFT will fail. The task will only be LIFTed if TRY_LIFT succeeds on arcs which total more than some threshold (again, say, 50%) probability of being the route by which control reaches our path.

We temporarily insert a buffer block, initially empty, between each immediate predecessor block and the block we are lifting from. When a task is lifted, we actually insert the MOP into those positions dictated by the calls to TRY_LIFT. If there are immediate predecessor blocks the task is not lifted into, either because TRY_LIFT failed or because of an unoptimized predecessor, we place the task at the bottom of that block's buffer.

When lifting is completed, we eliminate all still empty buffers, and make a single buffer for each set of identical buffers, adjusting addresses accordingly. */

CALL SCHEDULE-PATH;

/* We now use our ordinary methods of forming list schedules, as in intrablock scheduling, with the major exception of the loop tasks. When a loop task is encountered, we look at the partition being formed. If the loop, ℓ_i , is the first task being placed in the partition, then we so place it and continue as described below. If other tasks are already scheduled in the partition, we call TRY_DROP for each t_j in the partition, in reverse order of when they were scheduled in the partition. TRY_DROP attempts to place tasks in loop L_i (from the top) without adding a cycle to L_i's already formed schedule, just as TRY_LIFT does from the bottom. If TRY_DROP succeeds for all such t_j's, then each corresponding MOP is added to the loop in P , and ℓ_i is scheduled in the partition otherwise. ℓ_i is treated as if it had a resource conflict and is delayed. Note that TRY_DROP and TRY_LIFT recognize loop invariance and take it into account when considering moving a task into a loop; also both routines balk at moving conditional jumps or loops at all, so we do not attempt to place either in a loop.

Eventually, ℓ_i will be placed in some partition. Tasks after ℓ_i which are considered for placement in the same partition are passed to TRY_LIFT which tries to fit them into the remaining holes in L_i's schedule. If they can be so placed, they are, and we consider them scheduled in that partition. */

```
CALL RECOVER_SPACE;
/* A task which moves below a conditional jump it was previ-
   ously above will, generally, have to be copied into the
   off-the-path target block of that jump in order to preserve
   data validity, as will a task which was below a join to
   the path, but is now above the earliest spot for a legal
   rejoin.
```

RECOVER_SPACE alters our temporary schedule in an attempt to reduce the amount of copying done. In particular since, due to the nature of list scheduling, we would not expect to be able to move tasks up from their scheduled positions, we identify conditional jumps which have been moved up past tasks they followed, and tasks which are too far up to allow them to participate in a rejoin in which they are required, as candidates to be moved down into holes in PTEMP.

In the course of trying to recover space, the routine will notice that at certain arcs coming to or leaving the path, new blocks will have to be created to hold tasks which must be duplicated — namely the space that was unable to be recovered. If the arcs lead to or from already optimized blocks, we try to fit the tasks into the blocks in a way analogous to TASK_LIFT. Those that are not so fitted are regarded as single MOP or loop microinstructions and PTEMP is updated to include these, with the addresses adjusted appropriately.

```
*/
```

```
CALL CUT_AND_PASTE;
/* CUT_AND_PASTE:
   1. Updates P and related functions to reflect the
      new parallelization.
   2. Finds the basic blocks of the parallelization
      and adds the new names to L and B, removing the
      elements of PATH from L.
   3. Updates <L to reflect the new blocks.      */
```

Detailed Algorithms for Interblock Optimization

OPTIMIZE:

INPUTS: (1) M, a set of MOPs as defined earlier

(2) E, a set of exit node names

(3) The functions TRUEJUMP, FALSEJUMP, LVTOP, READ, WRITE, U as defined in Definitions 2 and 3

OUTPUT: A set P of microinstructions as defined in (4) and (5), with revised functions TRUEJUMP and FALSEJUMP.

USES: (1) B, a set of block names, with values basic blocks of microinstructions as in definition (8)

(2) LOOPS, a set of loop names, each with value a subset of B ∪ LOOPS

(3) OUTER_LOOPS, a map: LOOPS → subsets of LOOPS

(4) EXITS, a map: LOOPS → B ∪ E

(5) $<_{L_i}$ for $L_i \in$ LOOPS. A subset of $L_i \times (L_i \cup \text{EXITS}(L_i))$

(6) ARC_PROB, a map: $\left\{ \bigcup_{L_i \in \text{LOOPS}} <_{L_i} \right\} \rightarrow [0,100]$

(7) EXPECT, a map: $\left\{ \bigcup_{L_i \in \text{LOOPS}} L_i \right\} \rightarrow [0,100]$

(8) T, a set of tasks, with data precedence relation $<_T$

(9) Predicates OPTIMIZED and LVTOP_VALID on (B ∪ LOOPS)

METHOD: Calls the routines described below as indicated by the previously given calling sequence. The above variables are to be considered global to all following procedures.

MAKE_BASIC_BLOCKS:

INPUT: P, the set of microinstructions as defined in Definition (6).

OUTPUT: $B = \{B_1, B_2, \ldots, B_t\}$, a set of basic block names, with values basic blocks as defined in Definition (8).

METHOD: See Algorithm 12.1, page 412, Aho-Ullman [AHO77].

MAKE_FLOW_GRAPH

INPUT: P, B

OUTPUT: (1) A revised P and B revised to account for the property of LOOPS described below.

(2) The set $LOOPS = \{L_1, L_2, \ldots, L_p\}$ where each L_i is a subset of LOOPS $\cup$ B. (That is, LOOPS is a set of names with values a subset of LOOPS $\cup$ B.) The elements of LOOPS are originally subsets of B alone, and as such any two loops are either disjoint or one contains the other. After forming LOOPS, however, this routine replaces all of the blocks in L_i contained in an outermost loop L_j of L_i, with L_j itself. So then an element $L_i \in$ LOOPS will consist of the names of all blocks which are in L_i but no smaller loop, plus all names of loops in L_i but no smaller loop.

(3) The functions OUTER_LOOPS, EXITS, $<_L$ for $L \in$ LOOPS. OUTER_LOOPS(L) is the set of all names of LOOPS in the set L. EXITS is the set of all names of blocks (or elements of the initial E) jumped to by microinstructions in blocks of L, but not contained in L. $<_L$ is a subset of $L \times (L \cup \text{EXITS}(L))$ where $(B_i, B_j) \in <_L$ means that B_i is a predecessor block of B_j in the usual way.

METHOD: (1) $<_B$ is built as in the algorithm on pp. 450-454 of [AHO77], those algorithms also determine whether $<_B$ is irreducible.

(2a) If $<_B$ is irreducible, node splitting is used to produce a new P, B, and $<_B$ so that $<_B$ is reducible. For details on the transformations and duplications necessary to accomplish node splitting, see Chapters 4-6 of [HECH77].

(2b) If two loops L_i and L_j share a header H, make a copy of H, call it H', and have the back edge from L_j to H jump to H' instead. Then L_i will be as before, but L_j will be $(L_j - \{H\}) \cup \{H'\}$. Now any two loops will be either disjoint or nested. Update P, B and $<_B$ in the obvious way to reflect these changes.

(3) Form the set LOOPS where L $\in$ LOOPS is a set of block names $\subseteq$ B, such that L is a natural loop of B. Include B itself as an element of LOOPS. Again use pp. 450-454 of [AHO77].

(4) Build OUTER_LOOPS, EXITS, change LOOPS, and build $<_{L_i}$ for each $L_i \in$ LOOPS. All were defined in the comments for this routine, and all may be done in a completely straightforward way.

/* Unfortunately, node-splitting is something of a headache, particularly considering the expected rarity of its use. The algorithms known for finding the fewest nodes necessary to duplicate to accomplish node-splitting all use procedures known to be np-complete, such as minimum covering [HECH77]. However, one would assume that occasionally running an exponential program would not be too great a burden, in practice. Furthermore, even if a few too many nodes were produced, we are in an environment where the running time of the produced code is generally more critical than the space it uses.

One might very well expect an implementation, especially one which will not have wide, but will have critical use to simply harass the programmer into redoing code that produced an irreducible flow graph. Certainly a routine to convert such code could be added later. */

LIVETOP_ANALYSIS:

INPUT: (1) P, B, $<_B$ as before.

(2) The set $E = \{E_1, E_2, \ldots, E_n\}$ of exit nodes.

(3) The function LVTOP: E → subsets of A, where A is the set of registers.

OUTPUT: LVTOP is extended so that it maps E ∪ B into subsets of A. $LVTOP(B_i)$ is the set of registers whose values are live at the entrance to B_i. LVTOP is then further extended to elements of LOOPS. LVTOP_VALID is marked TRUE for each name LVTOP is defined on.

METHOD: Live variables can be found using Algorithm 146 and the accompanying discussion on pp. 489-490 in [AH77].

/* The above algorithm uses the following definitions:

$DEF(B_i)$ = the set of registers written into by B_i before any uses of those registers.

$USE(B_i)$ = the set of registers used in B_i before any writes to those registers.

$LVBOT(B_i) = \bigcup_{B_i <_B B_j} LVTOP(B_j)$ (they use IN and OUT for LVTOP and LVBOT)

Then the formula used for $LVTOP(B_i)$ is

$$LVTOP(B_i) = \left(LVBOT(B_i) - DEF(B_i)\right) \cup USE(B_i) .$$

To extend LVTOP to LOOPS, we just note that if

$L \in LOOPS$,

$LVTOP(L) = LVTOP(B_i)$, where B_i is the entrance block of L.*/

ASSIGN_JUMP_PROBABILITIES:

INPUT: LOOPS, $<_L$ for all $L \in LOOPS$.

OUTPUT: Two functions are produced: ARC_PROB and EXPECT, as defined earlier.

(1) ARC_PROB((B_i,B_j)) where $B_i <_L B_j$ is the estimated probability that if control flows to B_i it will next flow to B_j.

(2) EXPECT(B_i) is the probability, calculated from ARC_PROB and $<_L$ that, given the entrance node is reached, B_i is traversed before an exit node.

METHOD: For ARC_PROB we suggest several methods:

(1) Programmer estimate

(2) Simulation of unparallelized microcode on sample data.

(3) Running unparallelized microcode in the hardware on sample data, with added code to trace branches.

EXPECT can be calculated from ARC_PROB as follows:

(1) Consider the elements of LOOPS separately.

(2) If $L \in LOOPS$ with entry node B_k, $EXPECT(B_k) = 1$.

(3) For all other nodes, $B_j \in L$,

$$EXPECT(B_j) = \sum_{B_i <_L B_j} (EXPECT(B_i) * ARC\_PROB((B_i,B_j)))$$

Note that this calculates the probability of a node being reached. It does not measure the expected number of iterations, in case the node is in a loop, although we could do such a calculation. What we have is good enough, since we will only want to compare nodes within a loop.

/* As a further explanation of two of the methods of deriving ARC_PROB estimates:

Simulation. It is reasonable to expect that a microcode optimizer would only be one part of a highly automated design system. If such a system contained a microcode simulator, it would presumably take a small change to the simulator to count jumps taken. Counting could be done during the debug phase of code development, or if that is too biased toward pathological code, by running an appropriate mix of sample code. We note that the simulator would presumably not have to be altered to run the vertical source code, as such code is legal, if inefficient, horizontal code.

In the hardware. In a user microprogrammable system, already functioning, a preprocessor might insert a microcode subroutine call before every jump. The subroutine would test

the same condition, record its truth or falsity, and return the machine to its previous state. This could be done on a suitable mix of data far faster than option (b). (A macro may be more appropriate on many machines than a subroutine.)

SCHEDULE:

INPUT: (1) L, an element of LOOPS

(2) All global variables.

OUTPUT: (1) P is revised to reflect a parallelization of all blocks contained in L.

(2) L, $<_L$ and all associated maps and functions are similarly revised.

METHOD: Uses the routines described below as indicated by the previously described calling sequence.

PICK_A_PATH:

INPUT: L, EXPECT, OPTIMIZED

OUTPUT: An ordered set PATH, a subset of L,

PATH = $\{B_1, B_2, \ldots, B_n\}$ and on $<_L$, $B_1 < B_2 < \cdots < B_n$.

METHOD: (1) Recall that when we refer to already optimized elements of L, we mean in this call of SCHEDULE; that is, the previously optimized loop nodes have not yet had OPTIMIZED set to true; only

their contents have on previous calls to SCHEDULE.

(2) Examine all not yet optimized nodes, all of whose successors are elements of EXITS(L), already optimized nodes, or the entrance node for L.

(3) From among the nodes found in step (2), pick the one whose EXPECT value is greatest, let PATH equal the set containing that node.

(4) Now work backwards. For each node placed on PATH, pick the predecessor whose EXPECT value is greatest and add that to PATH, ignoring predecessors which are already optimized.

(5) Stop when the entrance node for L is reached, or when a node is picked which has all of its predecessors already optimized.

/* Our method of picking a path is rather arbitrarily selected; one could surely produce flow graphs for which the path so selected is rather poor. It is our strong belief, though, that as long as one does not get sidetracked onto very lightly traveled paths, the method chosen is not critical.

Nonetheless, we suggest two other selection methods:

(1) Pick, from among the unoptimized nodes of L, the one with the highest EXPECT. Work backwards, as above, from that node, and similarly work forwards,

always going to the node with the highest EXPECT.

(2) Pick the path backward and/or forward from some node (e.g., the highest EXPECT), in an attempt to produce a high average EXPECT. This might be done in a manner similar to Dijkstra's Single Source algorithm [AHO74], pp. 207-209.

Other methods could be imagined indefinitely — we have no guide other than intuition to their performance. */

MAKE_TASKS:

INPUT: P, L, OUTER_LOOPS(L), PATH

OUTPUT: A set $T = \{t_1, t_2, \ldots, t_w\}$ of newly defined tasks.

METHOD: (1) $T = \emptyset$

(2) Consider each $B_i \in$ PATH in turn.

(3) If $B_i = L_j$ for some $L_j \in$ OUTER_LOOPS(L) then: $T = T \cup \{\ell_j\}$ where ℓ_j is a task created to stand for the loop L_j.

(4) Otherwise, B_i is a set of elements belonging to P, and for each such element we create a unique task t_j. Let $T = T \cup \{t_j\}$.

MAKE_DATA_PRECEDENCE_GRAPH:

INPUT: T, the set of tasks with all associated sets and functions.

OUTPUT: A partial order on T.

METHOD: We alter the algorithm given in Figure 5.2 for finding the data-precedence in a set of MOPs as follows:

(a) Conditional jump tasks

(1) For each conditional jump task t_i we have the set LVTOP(B_j) where B_j is the target block of the off the path jump at t_i.

(2) With each register we have the set COND_READS_SINCE_WRITE which is treated essentially like READS_SINCE_WRITE. When we process t_i in our graph formation, we add t_i to each COND_READS_SINCE_WRITE(a_k) , where $a_k \in$ LVTOP(B_j).

(3) For each task t_j that writes a register a_k we

(i) Draw a STRICT edge from each element of COND_READS_SINCE_WRITE(a_k) to t_j.

(ii) Clear COND_READS_SINCE_WRITE(a_k)

(b) LOOP tasks

(1) We defer consideration of loop invariance until it is needed in specific situations, thus the question of whether a task strictly or equally follows a loop task is considered during schedul-

ing, see SCHEDULE_PATH. (This is done for efficiency reasons.)

(2) Given a loop task t_i which represents the loop L_i, we define the sets

$$READ(t_i) = \bigcup_{P_j \in L_i} READ(P_j) \quad \text{and}$$

$$WRITE(t_i) = \bigcup_{P_j \in L_i} WRITE(P_j)$$

where by $P_j \in L_i$ we mean P_j is a microinstruction in a block of L_i or some loop contained in L_i.

(3) If t_j is another task in T, we use the rules in Figure 5.2 to find edges (t_i, t_j) or (t_j, t_i) in $<_T$, except that we consider all such edges to be equal edges.

(4) If L_i contains a conditional jump off (or farther down) the PATH, we follow the rules in (a) above to form strict edges.

(5) If we wish to minimize microinstruction space used, we may draw edges from some tasks to t_i or from t_i to some conditional jumps (see space saving below).

FORM_PRIORITY_LIST:

INPUT: T and the data precedence graph T.

OUTPUT: Function PRI: T → Real numbers.

METHOD: Any appropriate method as used with intrablock optimization may be applied here. Decisions must be made, once a heuristic has been chosen, about extending the heuristic, particularly for loop tasks. Loop tasks may or may not be considered unit-execution tasks, and the question of resource conflict between a nonloop task and a loop task may be somewhat subtle and time consuming to apply. Given the results of the experiments in Chapter 7, however, it is probably nearly optimal to just use highest levels, considering loops to be unit execution time tasks.

TASK_LIFT: PROC;

INPUT: (1) L, $<_L$, T, data-precedence graph $<_T$

(2) THRESHOLD1, THRESHOLD2, real numbers in [0,1].

OUTPUT: A possibly revised, L, $<_L$, P, and T, revised to reflect the lifting of tasks to optimized blocks above PATH.

METHOD: (1) Form the set $S \subset L$.

$S_i \in S$ means that $S_i <_L B_1$, where B_1 is the first element on the chain of nodes chosen for PATH.

(2) If $\sum_{S_i \in S}$ [EXPECT (S_i) * (ARC_PROB$((S_i,B_1))$)]

$<$ EXPECT(B_i) * THRESHOLD1

then return.

/* There are too few profitable candidate blocks to move tasks into at the possible expense of the others. */

(3) Place an empty "header" block in L between each element of S which has been optimized and B_1. Place a single empty header node between *all* unoptimized elements of S and B_1.

(4) Form a priority ordered list of tasks in T with no predecessors on $<_T$. Eliminate all t_j such that t_j is:

(a) a loop task

(b) a conditional jump

(c) below a join to a block on PATH besides at B_1.

(5) DO for each t_j on the list:

(6) Call TRY_LIFT(t_j,s_i) for each $s_i \in S$ where s_i has been optimized and no predecessor of t_j is in the header after s_i.

(7) If $\sum_{s_i \mid \text{TRY\_LIFT}(t_j,s_i)} [\text{EXPECT}(s_i) * (\text{ARC\_PROB}((s_i,B_1)))] < \text{EXPECT}(B_1) * \text{THRESHOLD2}$

then remove t_j from the list.

(8) Otherwise DO:

(9) Take the union of all indices returned by TRY_LIFT(t_j,s_i) and place P_j in each indicated microinstruction, revising L and P.

10. For each s_i which returned null or for which we did not call TRY_LIFT, place a copy of P_j at the bottom of its header.
11. Eliminate t_j from T and $<_T$.
12. Place any tasks which no longer have precepredecessors on $<_T$ on the list in priority position.
13. Remove t_j from the list.

14. END 8.

15. END 5.

16. From each set of identical headers, pick one, have all jumps to any of the headers in the set jump to the representative instead, delete the others.

17. END TASK_LIFT.

TRY_LIFT: PROC(P_i,B_j); /* called by task-lift */

INPUT: P_i , a microinstruction containing a single MOP which is not a conditional jump

B_j , a block $\in$ L.

OUTPUT: The index of the highest $P_k \in B_j$ that P_i can legally be moved up into from a block below B_j , and null if no such index exists.

METHOD: (1) If B_j belongs to a member of OUTER_LOOPS(L), and if P_i is not loop-invariant for that loop, return null.

(2) Let P_n be the terminal microinstruction of B_j. If P_n writes a register that P_i reads, or if P_n is a conditional jump, and the other block P_n targets has a live register in its LVTOP that P_i writes, then return null.

(3) Otherwise, we try to back P_n up the chain of microinstructions making up B_j , starting with P_n at the terminal instruction, until one of the following happens:

(a) P_n is the initial microinstruction of B_j ,

(b) P_i writes a register that P_n reads ,

(c) P_i reads a register written by an immediate predecessor of P_n

(4) If P_i does not resource conflict with P_n , we return n.

(5) Otherwise, go forward on the chain towards the terminal node. Return the index of the first

microinstruction found which does not resource conflict with P_i.

(6) Return null if step (5) fails.

/* A more sophisticated (by far) task lifting is possible: Somewhat informally:

(1) We continue beyond B_j in this fashion, backing past conditional jumps and checking loop invariance before backing into loops.

(2) If we find ourselves about to back past a join, we temporarily stop and find the highest possible legal index, as above, if any.

(3) We now generate a new copy of this procedure for each block preceding our join. If they all succeed, we return the union of the indices that they returned.

(4) If, at least one of them fails, and we had found an index in step (2), return that index. If step (2) also failed to find an index, return null.

(5) Never try to back into an unoptimized block.

*/

SCHEDULE_PATH:

INPUT: T, the set of tasks ordered by a priority function.

OUTPUT: A parallelization of T, called PTEMP.

METHOD: Scheduling here proceeds using the list scheduling algorithm given in Chapter 4. At step (4), however, resource conflict is determined by checking that no resource is overused. That is unchanged here, except if a loop task is being scheduled or has been scheduled for the current partition. In that case, we follow a totally different procedure:

(1) Our next ready task is a loop task, ℓ_i :

a) If the current instruction already has a loop task or a conditional jump task, we reject ℓ_i as if it had a resource conflict.

b) We consider the set of tasks already scheduled in the current instruction in source order latest-to-earliest. For each, we TRY_DROP it into the loop. If TRY_DROP finds that it is loop invariant and can be placed in the already existing schedule for the loop, it is temporarily so placed and the loop characteristics used to determine loop invariance are temporarily updated. LOOP invariance is discussed in detail in [AHO77], see particularly algorithm 13.4 on page 458.

c) If the whole set of tasks may be so placed, ℓ_i is scheduled for the current instruction and the tasks

are placed permanently in the positions found for them in step (b); then P is appropriately updated.

(2) Our next ready task, t_k , is being considered for scheduling in an instruction which has a loop task, ℓ_i , already placed in it.

a) If t_k is a loop task or a conditional jump task we reject t_k as if it had a resource conflict with the current instruction.

b) We attempt a TRY_LIFT of t_k into ℓ_i via the on-the-PATH exit from ℓ_i. If successful, we schedule t_k in the current instruction and we record t_k in its lifted position in the permanent copy of L_i , updating the associated sets and P.

/* TRY_DROP is identical in every respect to TRY_LIFT, which is given in detail, except that tasks are inserted into blocks from above rather than from below. */

RECOVER_SPACE

INPUT: PTEMP, L, $<_L$, P

OUTPUT: A possibly revised PTEMP, P, L, and B. PTEMP is changed to reflect task motions designed to reduce copying. L may have tasks (removed from PTEMP) added to its already optimized blocks, and may have blocks of MOPs copied from PTEMP. B has new buffer block names added.

METHOD: /* We give first a relatively straightforward method of doing this. We can paraphrase the method as:

-- Find the tasks we might want to move down

-- find the task from those which has the most potential and may be legally moved down into a hole in the schedule

-- move that task down to the hole which saves the most space

-- update and go back to the beginning

-- when no more motion is possible, update L and P. */

1. Form the set TARGET. $t_i \in$ TARGET if:

(a) $t_i \in T$ is a conditional jump and there exists a task t_j such that:

(i) t_j was earlier than t_i on PATH; and

(ii) t_j has been scheduled later than t_i in PTEMP.

or

(b) There is a joining edge to a block B_k on PATH from other than B_{k-1} such that

(i) t_i was in B_k or below on PATH; and

(ii) there is a t_j which was above B_k on PATH but has been scheduled below t_i in PTEMP

2. Assign PRIORITY(t_i) for each $t_i \in$ TARGET.

PRIORITY(t_i) is the number of tasks which need to be copied due to t_i's position in PTEMP.

(a) If t_i is of the type described in 1(a), PRIORITY(t_i) is the number of tasks t_j as described in 1(a).

(b) If t_i is of the type described in 1(b), PRIORITY(t_i) is the number of edges described in 1(b).

(c) If t_i is of both types, PRIORITY(t_i) is the sum of the PRIORITYs found in 2(a),(b).

3. DO UNTIL (no changes are made by steps 4-9)

4. t_i = element of TARGET with highest priority

5. Choose an already scheduled microinstruction in PTEMP as follows:

(a) We consider microinstructions scheduled below the instruction in which t_i is scheduled, but above the first instruction which must follow t_i for data-dependency reasons.

(b) We further restrict ourselves to those instructions which t_i could be scheduled in without conflict. If no instructions remain, let t_i be the task with the next

highest priority and return to the beginning of step 5.

(c) For each instruction under consideration, we calculate the number of tasks which must be duplicated if t_i is placed there.

(d) Pick the microinstruction which provides the lowest such value, except that if all such values are higher than PRIORITY(t_i), let t_i be the task with the next highest PRIORITY, and return to the beginning of step 5.

(e) In case of a tie at step (d), favor the earlier microinstruction, unless t_i has predecessors in TARGET. In that case, we want to give the predecessors more room, so we pick the latest.

6. Move t_i from its originally scheduled microinstruction to the new instruction chosen in step 5.
7. Remove t_i from TARGET if a recalculation shows its PRIORITY to be zero.
8. Recalculate the PRIORITY values for members of TARGET whose values may have changed by the motion of t_i.
9. Add to TARGET any elements which will now qualify under 1(a) or (b), and calculate their priorities.
10. END 3.

/* We now do the necessary copying. */

11. Find all tasks scheduled below a conditional jump which they were earlier than in the source and identify the set of all such jumps.
12. For each jump identified in 11, place a buffer block at the exit from PATH and place in the buffer copies of all tasks which moved below that jump, except for those tasks which only write registers dead in that branch.
13. For each such exit which leads to an already optimized block, try a TRY_DROP on the tasks in the buffer (in reverse source order). When successful, strike the task from the buffer and add it to the optimized block in L.
14. Do the analogue of 11-13 (using TRY_LIFT) for rejoins to the PATH.
15. Update L and P to contain the buffer blocks; they will be included in future PATHs. $<_L$ is similarly updated. Update the functions TRUEJUMP and FALSEJUMP to account for the buffer blocks and the rejoins. Mark OPTIMIZED and LVTOP-VALID false for each of the new buffer blocks.

In the case of the last task (in PATH order), if it were a conditional which has moved above another, we consider the less likely branch to be the off-the-PATH one, and we place our buffer after the less likely branch. This choice is arbitrary, but must be consistent with the address correction done in CUT_AND_PASTE.

/* Notes on space saving:

1. In many environments, space saving is a luxury item. As a result, a first implementation might well avoid it, and any implementation for which space was not critical or for which practice showed space did not increase, might not want to add it at all.
   In contrast, an on-chip ROM controlling a mass produced microprocessor might be an environment in which a much more sophisticated space saver than this might be appropriate.
2. One can easily produce a situation in which the above algorithm performs miserably. For example, we might have a test near the very end of a long sequence of code which might be moved to the beginning and may not be movable down due to the early placement of one of its successors. As a result, more sophisticated (by far) methods than these may be appropriate.
3. The problem somewhat resembles the chip placement problem [HANA76]. That is, the tasks have been initially placed in some location and may now be moved to slots elsewhere, or may displace other tasks when moved

to other microinstructions. This displacement would set off a string of motions which would end when a task was moved to an empty slot. If no such ending occurred, the entire string would be rejected (e.g. see "force-directed relaxation" in [HANA76]).

4. A form of space saving which we strongly recommend concerns loops. If a loop is long, with a schedule length of five or more microinstructions, say, we may not want to duplicate it. We can avoid the duplication in advance by, during the data precedence graph formation:
   (a) Draw an edge from the loop task to all conditional jumps which follow it on PATH; and
   (b) Locate all rejoins to PATH which are above the loop. For each task t_i above the latest such rejoin, draw an edge from t_i to the loop.

*/

CUT_AND_PASTE

INPUT: PTEMP

OUTPUT: A changed P, $\underline{L}$, $<_L$ reflecting the parallelization of PATH.

(1) Replace each element of PTEMP containing a loop task with the fully scheduled loop contained in P. This copy of the loop already contains all tasks added to the loop during scheduling.

(2) Do a flow analysis of PTEMP, producing blocks B' and flow graph $<_{B'}$, treating the loop as one entity, as before.

(3) Delete from L all elements of PATH. Each rejoin jump from L is redirected to the spot found for the rejoin in RECOVER_SPACE. (Many come from newly created buffer blocks.) Add B' to L.

(4) Remove from $<_L$ the edges which came from blocks in PATH and replace them with $<_{B'}$.

(5) Mark all elements in L which were blocks in B' OPTIMIZED.

(6) Produce new ARC_PROB, EXPECT for all revised elements of B'.

(7) Mark the LVTOP_VALID of each new block false; any future references to those LVTOPs will need a new calculation.

(8) Delete from P all microinstructions which correspond to nonloop tasks in T, and add to P all new microinstructions implied by PTEMP.

(9) If PATH contained the entrance node of L, we find all instructions in P which jumped to that node from out of L and replace the target address with that of the first element on PTEMP. All other jumps to the PATH were rejoins considered in RECOVER_SPACE.

(10) We now fix the functions TRUEJUMP and FALSEJUMP to account for our new parallelization. Each conditional jump along PATH has one jump target which was to the next block on the PATH, except that if the last P_i along PATH is a conditional jump we define the more likely jump as the on-the-PATH jump. Then given a newly scheduled conditional jump t_i, we change the on-the-PATH jump to be the microinstruction scheduled immediately after t_i, and we leave the off-the-PATH jump as is. If t_i has been scheduled in the last cycle, we change the on-the-PATH jump to what was defined as the on-the-PATH jump for the original last micro-instruction.

A Detailed Example

We next present Example 8.1, which is again code written for the CIMS PUMA System. The vertical code, shown in Example 8.1a, converts a CDC Display Code decimal integer to binary, compacting spaces. The code has a moderately complex flow structure, and was written for the purpose of demonstrating this algorithm. We, however, did not write the code, and the algorithms presented here were not considered during its writing, although we did specify some characteristics of the flow structure.

The source code, which we hope is understandable even to those not very familiar with the PUMA, is shown in Ex. 8.1 a. We consider the flow graph to have four outer-most loops, as shown in 8.1 b, and thus schedule is called four times. The third call, shown in 8.1 f to 8.1 i is the most complex, with the bulk of the optimization being done there. The data precedence graph for the main path is shown in Ex. 8.1 f, and we see that our largest schedule for this example involves 19^tasks. The object code produced is shown in Ex. 8.1 ℓ, and can be seen to contain 21 micro-instructions. This is the same length as the code produced by an experienced microprogrammer (who was asked just to parallelize, not change, the MOPs used), and looks almost identical to the code he produced.

Note that two factors are considered here which were not mentioned in this chapter but are discussed in Chapter 9. Several of the MOPs are actually 2-cycle MOPs, with both halves written out; 7 and 8 are an example of such a pair. Chapter 9 mentions how these might be handled and some difficulties associated with them; fortunately, no special measures were necessary here. The second consideration was of tests such as line 8 of the source code. Originally, line 8, and several others, were 2 separate lines, one for the arithmetic operation and one for the register test. PUMA permits the testing of the input lines of some registers, and this code was preprocessed to specify that this be done. The preprocessing should be completely straightforward. We discuss this further under "special case precedence" in Chapter 9.

```
   * AT ENTRANCE, Y0=33, Y1=12, Y2=55, I.E. DISPLAY CODE(0);
   * DISPLAY CODE(9) - [(0)+1]; DISPLAY CODE(SPACE).
   * MQ HAS ORIGINAL NUMBER, AC=0, Y6=0.

 1. ENTER     E1=10     * 10 DIGIT NUMBER

 2. OUTER     E2=6    * 6 BITS PER DISPLAY CHAR

   * SHIFT A NEW DIGIT INTO RIGHT HAND 6 BITS
 3. INNER     AC:MQ=SHIFT(AC:MQ,L1)
 4.           E2=E2-1[F]; IF ~EALU(0-3)=0 THEN INNER

   *NO PROPER INPUTS ARE BELOW 33
 6. (INNER.1) BUF=Y0
 7.             =AC-BUF
 8.           AC=AC-BUF; IF ALU(59) THEN ERROR

   * ALL DIGITS ARE BETWEEN 33 AND 44
10. (INNER.2) Y7=AC
11.           BUF=Y1
12.             =AC-BUF
13.           AC=AC-BUF; IF ~ALU(59) THEN NOTDIG

   * SET UP MULTIPLY BY 10 - DONE INEFFICIENTLY
   * TO PROVIDE LOOP FOR EXAMPLE
15. (INNER.3) BUF=Y6
16.           AC=BUF
17.           E2=9
18. MULT        =AC+BUF
19.           AC=AC+BUF
20.           E2=E2-1[F]; IF ~EALU(0-3)=0 THEN MULT

   * ADD THE NEW DIGIT TO OLD VALUE TIMES 10
22. (MULT.1)  BUF=Y7
23.             =AC+BUF
24.           AC=AC+BUF
25.           Y6=AC

   * ANY MORE DIGITS?
26. MERGE     AC=0
27.           E1=E1-1[F]; IF ~EALU(0-3)=0
                          THEN OUTER ELSE EXIT

   * IF NON-DIGIT IS A SPACE, IGNORE IT.
   * OTHERWISE, ABORT.
28. NOTDIG    BUF=Y2
29.             =AC-BUF
30.           AC=AC-BUF
31.           IF AC=0 THEN MERGE

   * STORE INFINITY IF ERROR IN INPUT
32. ERROR     E0=1777
33.           AC=0
34.           Y6=E0:AC
              GO EXIT

   * ALL REGS DEAD ON EXIT EXCEPT Y6
   * Y6 CONTAINS MQ CONVERTED FROM
   * DISPLAY CODE TO BINARY
```

EXAMPLE 8.1A SOURCE CODE. SOME LINE NUMBERS NOT USED DUE TO PRE-PROCESSING. BLOCK NAMES IN PARENTHESIS NOT SPECIFIED BY PROGRAMMER.

EXAMPLE 8.1 b. BLOCK FLOW INFORMATION.

Set of basic blocks:

B = {ENTER, OUTER, INNER, INNER.1, INNER.2, INNER.3, MULT, MULT.1, MERGE, NOTDIG, ERROR}

The set LOOPS = $\{\ell_1, \ell_2, \ell_3, \ell_4\}$

where ℓ_1 = {INNER}

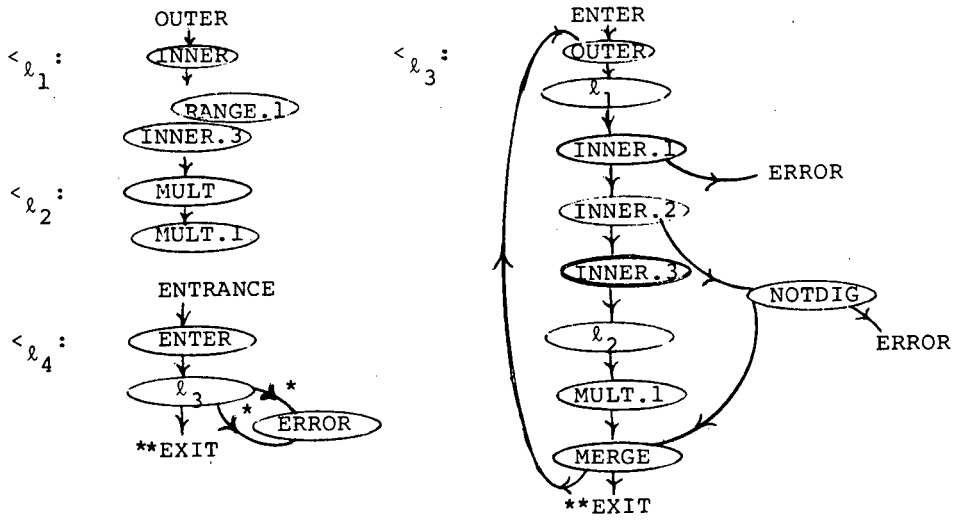
ℓ_2 = {MULT}

ℓ_3 = {OUTER, ℓ_1, INNER.1, INNER.2, INNER.3, ℓ_2, MULT.1, MERGE, NOTDIG}

ℓ_4 = {ENTER, ℓ_3, ERROR}

The map OUTER-LOOPS maps $\ell_1 \rightarrow \emptyset$,
$\ell_2 \rightarrow \emptyset$,
$\ell_3 \rightarrow \{\ell_1, \ell_2\}$
$\ell_4 \rightarrow \{\ell_3\}$

The flowgraphs $<_{\ell_1}$, $<_{\ell_2}$, $<_{\ell_3}$, $<_{\ell_4}$ — each contained loop reduced to 1 node:



* Note: We signify that two different exits exist from ℓ_3 to ERROR at instructions 8 and 31.

** EXIT is off the source code.

EXAMPLE 8.1

EXAMPLE 8.1 c. THE MAPS LVTOP, ARC_PROB, EXPECT.

The map LVTOP: (EXIT→ {Y6}, given)

ERROR → ∅
MERGE →{Y0,Y1,Y2,Y6,AC,MQ,E1}
NOTDIG → {Y0,Y1,Y2,Y6,AC,MQ,E1}
MULT.1 → {Y0,Y1,Y2,Y7,AC,MQ,E1}
MULT → {Y0,Y1,Y2,Y7,AC,MQ,BUF, E1,E2}
INNER.3→ {Y0,Y1,Y2,Y6,Y7,MQ,E1}
INNER.2 → {Y0,Y1,Y2,Y6,AC,MQ,E1}
INNER.1 → {Y0,Y1,Y2,Y6,AC,MQ,E1}
INNER → {Y0,Y1,Y2,Y6,AC,MQ,E1,E2}
OUTER → {Y0,Y1,Y2,Y6,AC,MQ,E1}
ENTER → {Y0,Y1,Y2,Y6,AC,MQ}

The map ARC - PROB (programmer guess), extended to the loop-reduced flowgraphs within

$<_{\ell_1}$: no arcs
$<_{\ell_2}$: no arcs
$<_{\ell_4}$: (ENTER,ℓ_3) → 100%
(ℓ_3,ERROR) → 1 (via INNER.1)
(ℓ_3,ERROR) → 1 (via NOTDIG)

within $<_{\ell_3}$: (OUTER,ℓ_1) → 100
(ℓ_1,INNER.1) → 100
(INNER.1,INNER.2) → 99
(INNER.2,INNER.3) → 90
(INNER.2,NOTDIG) → 10
(INNER.3,ℓ_2) → 100
(ℓ_2,MULT.1) → 100
(MULT.1,MERGE) → 100
(NOTDIG,MERGE) → 99

The map EXPECT on the loop-reduced blocks:

within ℓ_1: INNER → 100%
ℓ_2: MULT → 100
ℓ_4: ENTER → 100
ℓ_3 → 100
ERROR → 100

within ℓ_3: OUTER → 100
ℓ_1 → 100
INNER.1 → 100
INNER.2 → 99
INNER.3 → 89
ℓ_2 →89
MULT.1 → 89
NOTDIG → 10
MERGE → 99
ERROR → 1

EXAMPLE 8.1

EXAMPLE 8.1 d. THE CALL SCHEDULE(ℓ_1).

PATH = {INNER} $\underline{T}$ = {3,4}

Data-Precedence Graph: 3 4

Priority List: 3 4 (using priority(t) = height(t)
breaking ties using earliest source order)

Task Lifting: None possible

Schedule: 3 , 4

Space Recovery: None necessary, TARGET = $\emptyset$

Cut and Paste: (1)

New elements of P	From old P's (removed)
5	3,4

(2) Updated function values:

P	TRUEJUMP	FALSEJUMP
5	5	6
2	5	5

(3) ℓ_1 = {B1} where B1 = {5}

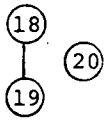
(4) $<_{\ell_1} = \emptyset$ (only one block)

EXAMPLE 8.1

EXAMPLE 8.1 e. THE CALL SCHEDULE(ℓ_2).

PATH:={MULT} T = {18, 19, 20}

Data-Precedence Graph:



Priority List: 18 19 20

Task Lifting: None possible

Schedule: 18, 20
19

Space Recovery: TARGET = {20}
20 is moved down to the second cycle
to get improvement.

Revised Schedule: 18
19 20
no copying

Cut and Paste: (1)

New elements of P	From old P's (removed)
9	18
14	19,20

(2) Updated function values:

P	TRUEJUMP	FALSEJUMP
9	14	14
14	9	22
17	9	9

(3) ℓ_2 = {B2} where B2 = {9,14}

(4) $<_{\ell_2}$ = Ø (only one block)

EXAMPLE 8.1

EXAMPLE 8.1 f. THE CALL SCHEDULE(ℓ_3);
MAIN PATH, TASK SET, DATA PRECEDENCE GRAPH.

PATH: {MERGE,MULT.1,ℓ_2,INNER.3,INNER.2,INNER.1,ℓ_1,OUTER}

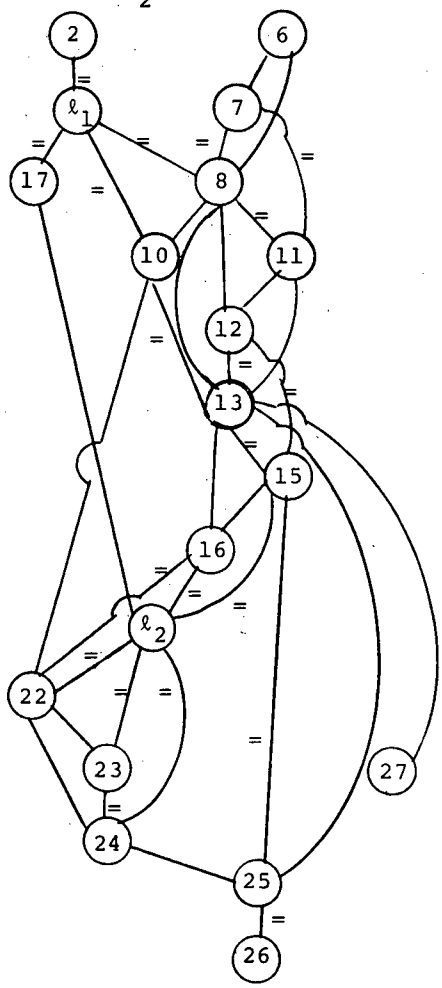
T = {2,ℓ_1,6,7,8,10,11,12,13,15,16,17,ℓ_2,22,23,24,25,26,27}

Data-Precedence Graph:

Formed Using:

TASK	RDREGS	WRREGS
2	Ø	E2
ℓ_1	AC,MQ,E2	AC,MQ,E2
6	Y0	BUF
7	AC,BUF	Ø
8	AC,BUF	AC
10	AC	Y7
11	Y1	BUF
12	AC,BUF	Ø
13	AC,BUF	AC
15	Y6	BUF
16	BUF	AC
17	Ø	E2
ℓ_2	AC,BUF,E2	AC,E2
22	Y7	BUF
23	AC,BUF	Ø
24	AC,BUF	AC
25	AC	Y6
26	Ø	AC
27	E1	E1

Also, 13 has Y2,Y6,AC,E1
live at the off-path jump.



EXAMPLE 8.1

EXAMPLE 8.1 g. SCHEDULING THE MAIN PATH

Level List	H	{t|HEIGHT(t) = H}
	1	26, 27, 25
	2	24, 23
	3	22, ℓ_2, 16, 17
	4	15, 13, 12, 10
	5	11, 8, 7, ℓ_1, 2
	6	6

Priority List: 6 2 ℓ_1 7 8 11 10 12 13 15
16 17 ℓ_2 22 23 24 25 26 27

Task Lifting: None possible
(Doesn't split off an already optimized path.)

Schedule:

(New Micro-Instruction)*	Cycle	Schedule	Delayed Due to Resource Conflict
(37)	1	6, 2	ℓ_1
-	2	ℓ_1	7, 17
(38)	3	7, 17	none
(39)	4	8, 11	none
(40)	5	10, 12	none
(41)	6	13, 15	none
(42)	7	16, 27	ℓ_2
-	8	ℓ_2	22
(43)	9	22	none
(44)	10	23	none
(45)	11	24	none
(46)	12	25, 26	none

* Used - in 8.1 i.

EXAMPLE 8.1

EXAMPLE 8.1 h. MAIN PATH, SPACE SAVING AND TASK COPYING.

Space Saving: Type(a) - 27 has moved above ℓ_2,22,23,24,25,26
Type(b) - 25,26 prevents a rejoin to 26 and 27, so 26 and 27 will have to be copied.

Thus TARGET = {27,26} with PRIORITY(27) = 8 (ℓ_2 counts as 2)
PRIORITY(26) = 1.

We attempt to move 27 down. 27 conflicts with cycle 8, but not with any of 9-12. No other rejoins are affected, and no conditional jumps become inverted, so the most space is saved by moving 27 to cycle 12.

We cannot move 26 down, since it is already in the last cycle.
So the revisions are: cycle 7: 16 (42)
cycle 12: 25,26,27 (46)

Task Copying: The space saver was not able to permit a legal rejoin to 26 and 27, so we create new micro-instructions 35 and 36, with 35 identical to 26 and 36 identical to 27. We set
TRUEJUMP(31) = 35 to make the rejoin, and set
TRUEJUMP(35),FALSEJUMP(35) = 36
TRUEJUMP(36) = 2, FALSEJUMP(36) = EXIT
P = P ∪ {35,36}
$\ell_3 = \ell_3 \cup \{B_3\}$ where B_3 = {35,36}
and in $<_{\ell_3}$,
Delete(NOTDIG,MERGE)
Add (NOTDIG,B3) with ARC_PROB = 99
(B3,OUTER), ARC_PROB not necessary
(B3,EXIT), ARC_PROB not necessary
EXPECT(B3) = 10

EXAMPLE 8.1

EXAMPLE 8.1 i. CUT_AND_PASTE FOR THE MAIN PATH.

CUT-AND-PASTE: (1) We form 10 new microinstructions 37-46, with their values given in the schedule table in 8.1 g and revised in space saving, so
$P = (P - (T - \{\ell_1, \ell_2\})) \cup \{37,38,\ldots,46\}$.

(2) Updated function values:

P	TRUEJUMP	FALSEJUMP
36	37	- (unchanged)
46	-	EXIT
20	-	43
5	-	38
37	5	5
38	39	39
39	32	40
40	41	41
41	28	42
42	9	9
43	44	44
44	45	45
45	46	46
46	37	EXIT

(3) $\ell_3 = \{B3,B4,\ldots,38,\ell_1,\ell_2\}$ where B4={37}, B5={38,39}, B6= {40,41}, B7= {42},B8={43,44,45,46}

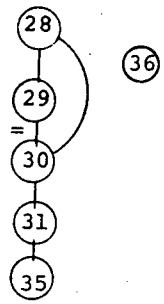
(4) New arcs in $<_{\ell_3}$ (B4,ℓ_1), (ℓ_1,B5), (B5,B6), (B6,B7), (B6,NOTDIG), (B7,ℓ_2), (ℓ_2,B8), (B5,ERROR), (B8,B4), (B3,B4)

EXAMPLE 8.1

EXAMPLE 8.1 j. THE REMAINDER OF ℓ_3.

The next PATH = {B3,NOTDIG}, T = {28,29,30,31,35,36}

Data-Precedence Graph:



Priority List: 28 29 30 31 35 36

Task Lifting: All branches to NOTDIG are optimized, so we have exceeded TRESHOLD1, and lifting may proceed. 28 and 36 are possible tasks to lift, but we eliminate 36 since it is a conditional jump. 28 may not be moved up, since it must follow 41. Thus no lifting is possible.

Schedule

(47) 28, 36
(48) 29
(49) 30
(50) 31
(51) 35

Space Recovery: TARGET = {36}.
The most improvement is obtained by moving it to (51), so now:

(47) 28
(51) 35,36 are the changes.
No tasks need be copied.

Cut_and_Paste: (1) We form 5 new microinstructions as given above, so
P = (P - T) ∪ {47,48,49,50,51}
(2) Changed jumps:

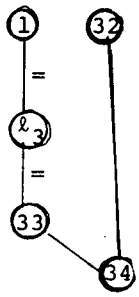
P	TRUEJUMP	FALSEJUMP
41	47	-
47	48	48
48	49	49
49	50	50
50	51	32
51	37	EXIT

(3) ℓ_3= {ℓ_1,ℓ_2,B4,B5,...,B10} where B9 = {47,48,49,50}, B10 ={51}

EXAMPLE 8.1 k. THE CALL SCHEDULE(ℓ_4).

PATH:= {ENTER, ERROR, ℓ_3} $\underline{T}$ = {1,ℓ_3,32,33,34}

Data-Precedence Graph:



PRIORITY: 1 ℓ_3 32 33 34

TASK LIFTING: None possible

SCHEDULE: (52) 1
- ℓ_3, 32 (Note: 32 temporarily placed in microinstruction 50)
(53) 33
(54) 34

SPACE SAVING: TARGET = {32} (rejoin from an exit of ℓ_3)
We move 32 from ℓ_3 to 53, which makes a full rejoin possible.
(32 data precedes 54.) And we remove 32 from 50
Revised schedule
- ℓ_3
(53) 33, 32

CUT_AND_PASTE: (1) We form 3 new microinstructions 52,53,54.

P = (P - {1,32,33,34}) ∪ {52,53,54}

(2)

P	TRUEJUMP	FALSEJUMP
52	37	37
53	54	54
54	EXIT	EXIT
39	53	-
50	-	53

(3) ℓ_4={ℓ_3,B11,B12};
B_{11}= {52}, B12 = {53,54}

(4) new $<_{\ell_4}$ (B11,ℓ_3), (ℓ_3,B12)(via 39),
(ℓ_3,B12)(via 50),(B12,EXIT)

EXAMPLE 8.1 ℓ SYMBOLIC OBJECT CODE – FULLY PARALLELIZED VERSION OF 8.1 a.

```
 P     B      SYMBOLIC
 52    B11    E1 = 10
 37    B4     E2 = 6;  BUF = Y0
  5    B1     AC:MQ =   SHIFT(AC:MQ,L1)
       +      E2 = E2-1[F];  IF ∿EALU THEN B1
 38    B5        = AC - BUF;  E2 = 9
 39           AC = AC - BUF; BUF = Y1; IF ALU(59) THEN B12
 40    B6     Y7 = AC; = AC - BUF
 41           AC = AC - BUF; BUF = Y6; IF ∿ALU(59) THEN B9
 42    B7     AC = BUF
  9    B2        = AC + BUF
 14           AC = AC+BUF; E2=E2-1[F]
       +           IF ∿EALU(0-3) = 0 THEN B2
 43    B8     BUF = Y7
 44              = AC + BUF
 45           AC = AC + BUF
 46           Y6 = AC; AC=0; E1=E1-1[F]
       +           IF ∿EALU(0-3)=0 THEN B4 ELSE EXIT
 47    B9     BUF = Y2
 48              = AC - BUF
 49           AC = AC - BUF
 50           IF AC = 0 THEN B10 ELSE B12
 51    B10    AC=0; E1=E1-1[F]; IF ∿EALU(0-3)=0
                        THEN B4  ELSE EXIT
 53    B12    AC=0; E0=1777
 54           Y6 = E0:AC
       EXIT
```

EXAMPLE 8.1

Two Examples from the PUMA 6600 Emulator

We now present, with most details left to the reader, two examples from the PUMA CDC 6600 emulator. In both cases, the source code was written by the person who had originally written the hand optimized object code, with the aim being a clear exposition of the algorithm used.

Example 8.2 is the code which sets up the multiply loop. 8.2a contains the uncommented source code. Since the code contains no loops, there is only one call to SCHEDULE; the main path of which, along with the associated data-precedence graph, is shown in 8.2.b. Note that the main path includes 34 of the 38 source tasks, so is almost all of the optimization.

8.2 c and d show the derived object code, and for comparision, the original hand optimized code. Note that both use 19 lines of code, and that to set up a floating point multiply, both take 14 cycles for positive or negative arguments. The derived integer multiply takes 16 cycles instead of the 14 cycles in 8.1 d. This is the result of the space saver; the first schedule produced also took 14 cycles for an integer multiply set-up, but required 5 extra lines of microcode. The space saving algorithm as presented here will never lengthen the main path, but may move an off-the-path jump down and then possibly lengthen a non-main-path.

A more sophisticated space saver would presumably have decided on the extra space rather than time for so important a path; real effort might have located code that used neither, as in the hand optimized version. We note in passing that the first 5 lines of the hand optimized version contain an elaborate trick which allows the shift to proceed before X_k is loaded into the MQ. The straightforward version produced by our algorithms did just as well.

Example 8.3 is the OP CODE 24 (normalize) instruction from the emulator. 8.3 a shows the unoptimized source code, while 8.3 b,c are the derived object code and the hand optimized version used in the emulator. Note that these two sections of code were chosen to be tests of these algorithms, with the feeling that they are the hardest sections of the emulator code to optimize. While the methods were changed slightly after we had gained experience with OP CODE 24, the earlier versions of the algorithms performed essentially as well.

Our estimates of the time and space requirements of the three examples for our algorithms vs. the hand optimized code are as follows.

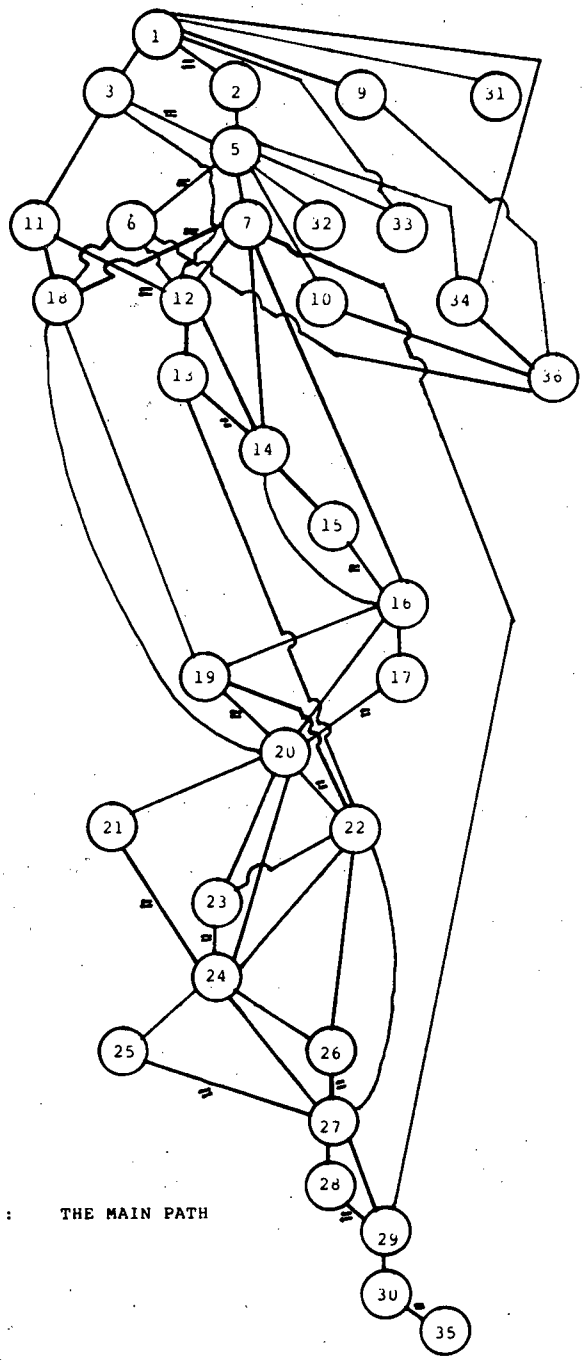
MULTIPLY SET-UP:	Algorithms are 4% slower than PUMA (expected cycles). No extra space used.
OP CODE 24:	Algorithms are 10% slower than PUMA. 16% more space used than PUMA.
CODE CONVERSION:	Time and space identical to that used in the hand optimized version.

```
                * SETS UP MAIN MULTIPLY LOOP. AT EXIT, Y REGISTERS
                * CONTAIN APPROPRIATE MULTIPLES OF ABS(XJ), MQ=ABS(XK),
                * AC=0, E0=EXPONENT OF PRODUCT. E2=15 HAS BEEN INITIAL-
                * IZED TO COUNT THE NUMBER OF CYCLES IN THE LOOP.

 1 40           E1:BUF=XJ
 2              IF BUF(59) THEN 40XJNEG

 3 (40.1)       AC=BUF; GO 40GETXK

 4 40XJNEG      AC=-BUF

 5 40GETXK      E2:BUF=XK
 6              IF BUF(59) THEN 40XKNEG

 7 (40GETXK.1)  MQ=BUF; GO 40TESTILL

 8 40XKNEG      MQ=-BUF

 9 40TESTILL    IF ILL(E1) THEN 40ILLEXP

10 (40TESTILL.1) IF ILL(E2) THEN 40ILLEXP

11 (40TESTILL.2) Y1=AC
12              AC=SHIFT(AC:MQ,L1)
13              Y2=AC
14              AC=SHIFT(AC:MQ,L1)
15              Y4=AC
16              AC=SHIFT(AC:MQ,L1)
17              Y0=AC
18              BUF=Y1
19              =AC-BUF
20              AC=AC-BUF
21              Y7=AC
22              BUF=Y2
23              =AC-BUF
24              AC=AC-BUF
25              Y5=AC
26              =AC-BUF
27              AC=AC-BUF
28              Y3=AC
29              AC=SHIFT(AC:MQ,L1)
30              Y6=AC
31              IF ZERO(E1) THEN 40XJZERO

32 (40TESTILL.3)   IF ZERO(E2) THEN WXIZERO

33 (40TESTILL.4) =E1+E2
34              E0=E1+E2; IF XFOFL THEN FLRESFLO

35 40INTMUL     AC=0
36              E2=15; GO EXIT


37 40XJZERO     IF #ZERO(E2) THEN WXIZERO

38 (40XJZERO.1) E0=6000; GO 40INTMUL



   WXIZERO      * LIVE VARIABLES: NONE
   FLRESFLO     * LIVE VARIABLES: E0 (ALSO AC DUE TO BUG IN PUMA CODE)
   EXIT         * LIVE VARIABLS: Y0-Y7, MQ, AC, E0, E2
   40ILLEXP     * LIVE VARIALBES: XJ, XK, E1, E2
```

EXAMPLE 8.2 A : MULTIPLY SET-UP SOURCE CODE

PATH = {40INTMUL, 40TESTILL.4, 40TESTILL.3, 40TESTILL.2,
40TESTILL.1, 40TESTILL, 40GETXK.1, 40GETXK, 40.1, 40}

T = {1, 2, 3, 5, 6, 7, 9, 10, 11, 12-31, 32, 33, 34, 35, 36}



EXAMPLE 8.2 B : THE MAIN PATH

```
40          E1:BUF = XJ; IF REG(59) THEN 40XJNEG
            AC=BUF; E2:BUF = XK; IF REG(59) THEN 40XKNEG
40XKPOS     MQ=BUF; Y1=AC; IF ILL(E1) THEN 40ILLEXP
40TESTE2    AC=SHIFT(AC:MQ,L1); BUF=Y1; IF ILL(E2) THEN 40ILLEXP
            Y2=AC; AC=SHIFT(AC:MQ,L1)
            AC=SHIFT(AC:MQ,L1); Y4=AC
            Y0=AC; =AC-BUF
            AC=AC-BUF; BUF=Y2
            =AC-BUF; Y7=AC
            AC=AC-BUF; =E1+E2
            Y5=AC; =AC-BUF; E0=E1+E2; IF XFOFL THEN FLRESFLO
            AC=AC-BUF; IF ZERO(E2) THEN WXIZERO
            AC=SHIFT(AC:MQ,L1); Y3=AC; IF ZERO(E1) THEN 40XJZERO
            Y6=AC; AC=0; E2=15; GO EXIT

40XJNEG     AC=-BUF; E2:BUF=XK; IF #REG(59) THEN 40XKPOS
            MQ=-BUF; Y1=AC; IF ILL(E1) THEN 40ILLEXP ELSE 40TESTE2

40XJZERO    AC=0; IF #ZERO(E2) THEN WXIZERO
            E0=6000
            E2=15; GO EXIT
```

EXAMPLE 8.2 C

```
40          E1%BUF=XJ;  MQ=0;  IF REG(59) THEN 40XJNEG ELSE 40XJPOS
40XJPOS     AC=BUF;  E2%BUF=XK;  IF ILL(E1) THEN 40ILLEXP ELSE 40FORMMP
40XJNEG     AC=-BUF;  E2%BUF=XK;  IF ILL(E1) THEN 40ILLEXP
40FORMMP    Y1=AC;  AC=SHIFT(AC%MQ,L1);  IF ILL(E2) THEN 40ILLEXP
            Y2=AC;  AC=SHIFT(AC%MQ,L1);  IF BUF(59) THEN 40XKNEG
            BUF=Y1;  MQ=BUF;  GO 40B
40XKNEG     BUF=Y1;  MQ=-BUF
40B         Y4=AC;  AC=SHIFT(AC%MQ,L1)
            Y0=AC;  =AC-BUF
            BUF=Y2;  AC=AC-BUF
            Y7=AC;  =AC-BUF
            AC=AC-BUF;  IF ZERO(E1) THEN 40XJZERO
            Y5=AC;  =AC-BUF;  IF ZERO(E2) THEN WXIZERON
            AC=AC-BUF;  =E1+E2
            Y3=AC;  AC=SHIFT(AC%MQ,L1);  E0=E1+E2;  IF XFOFL THEN FLRSFLON
40INTMUL    Y6=AC;  AC=0;  E2=15


40XJZERO    Y5=AC;  =AC-BUF;  IF ^ZERO(E2) THEN WXIZERON
            AC=AC-BUF; IF ^OPCODE(1) THEN WXIZERON
            Y3=AC;  AC=SHIFT(AC%MQ,L1);  E0=6000;  GO 40INTMUL
```

EXAMPLE 8.2 D

EXAMPLE 8.2 C AND D : DERIVED SYMBOLIC OBJECT CODE AND CODE AS HAND PRODUCED FOR PUMA

```
24              E0:BUF = XK
                IF ILL(E0) THEN 24ILL

24TESTXK        IF BUF(59) THEN 24XKNEG
                AC=BUF; GO 24B

24XKNEG         AC=-BUF

24B             MQ=0
                IF AC=0 THEN 24ZERO

                E2=0
                IF AC(47) THEN 24SHFTDN

NORMLOOP        AC:MQ = SHIFT(AC:MQ,L1)
                IF AC(47) THEN 24PLUS1

                AC:MQ=SHIFT(AC:MQ,L1)
                =E2+2
                E2=E2+2
                IF AC(47) THEN 24SHFTDN ELSE NORMLOOP

24SHFTDN        =E0-E2
                E0=E0-E2; IF FOFL THEN NORMUFLO ELSE 24SHFTDN.1

                IF #BUF(59) THEN 24WXI

                AC=-AC

24WXI           XI=E0:AC

24WBJ           IF J=0 THEN NEWINSLO

                E0=E2
                AC=E0
                BJ=AC
                NEWPARCEL; GO NEWINSTR

NORMUFLO        AC=0; GO 24WXI2

24ZERO          E2=60

24WXI2          XI=AC

24PLUS1         =E2+1
                E2=E2+1; GO 24SHFTDN

24ILL           E2=0
                AC=BUF; GO 24WXI
```

EXAMPLE 8.3 A : OP CODE 24 SOURCE CODE

```
24          E0:BUF = XK;IF REG(59) THEN 24XKNEG
            AC=BUF; IF ILL(E0) THEN 24ILL
24B         E2=0; MQ=0; IF ~AC(47) THEN NORMLOOP
            =E0-E2;IF AC=0 THEN 24ZERO
24SHFTDN    E0=E0-E2; IF FOFL THEN NORMUFLO
            IF BUF(59) THEN 24SHFTDN.2
24WXI       XI=E0:AC; IF J=0 THEN NEWINSLO ELSE 24WBJ.1

24XKNEG     AC=-BUF; IF ILL(E0) THEN 24ILL ELSE 24B

24SHFTDN.2  AC=-AC; GO 24WXI

NORMLOOP    IF AC=0 THEN 24ZERO
NORMLOOP.2  AC:MQ = SHIFT(AC:MQ,L1); =E2+2
+           IF AC(46) THEN 24PLUS1
            AC:MQ = SHIFT(AC:MQ,L1);E2=E2+2
+           IF AC(46) THEN 24SHFTDN.3 ELSE NORMLOOP.2
24PLUS1     =E2+1
            E2=E2+1
24SHFTDN.3  =E0+-E2; GO 24SHFTDN

24ZERO      E2=60; XI=AC; IF J=0 THEN NEWINSLO
24WBJ.1     E0=E2
            AC=E0
            BJ=AC; NEWPARCEL; GO NEWINSTR

24ILL       E2=0;AC=BUF; GO 24WXI

NORMUFLO    AC=0
            XI=AC; IF J=0 THEN NEWINSLO ELSE 24WBJ.1
```

EXAMPLE 8.3 B

```
24          E0&BUF=XK;  MQ=0;  IF REG(59) THEN NORMNEG
            AC=BUF;  IF ILL(E0) THEN 24ILL ELSE NORMZT
NORMNEG     AC=-BUF;  IF ILL(E0) THEN 24ILL ELSE NORMZT
NORMZT      E2=60; IF AC=0 THEN NORMWXI2
NOSHTEST    E2=0; IF AC(47) THEN 24SHFTDN
NORMLOOP    AC&MQ=SHIFT(AC&MQ,L1);  =E2+2;  IF AC(46) THEN 24PLUS1
            AC&MQ=SHIFT(AC&MQ,L1);  E2=E2+2;  IF ^AC(46) THEN NORMLOOP
24SHFTDN    =E0-E2;  IF BUF(59) THEN 24RECOMP
            E0=E0-E2;  IF FOFL THEN NORMUFLO
NORMWXI     XI=E0&AC;  IF J=0 THEN NEWINSLO
24WBJ       E0=E2
24WBJ2      AC=E0
            BJ=AC; NEWPARCEL;  GO NEWINSTR
24RECOMP    AC=-AC;  E0=E0-E2;  IF FOFL     THEN NORMUFLO ELSE NORMWXI
NORMUFLO    AC=0
NORMWXI2    E0=E2;  XI=AC; IF J=0 THEN NEWINSLO ELSE 24WBJ2
24PLUS1     =E2+1
            E2=E2+1;  GO 24SHFTDN
24ILL       AC=BUF; E2=0;  GO NORMWXI
```

EXAMPLE 8.3 C

EXAMPLE 8.3 B AND C : DERIVED SYMBOLIC OBJECT CODE AND CODE AS HAND PRODUCED FOR PUMA

9. Extensions for More General Models of Microprogramming

Non-conforming Models

In doing this investigation we had to choose between using a relatively simple (and usually unrealistic) model of microprogramming languages, and a more complex model into which many machine structures could probably be mapped. We chose a simple structure with the aim of investigating the parallelization algorithms in as pure a light as possible, hoping, then, that the methods which proved productive here could be adapted to more complex situations. We were also aided in our choice by the fact that the PUMA nearly matches our model, by our belief that many complexities of microprogrammed central unit design are actually the result of outmoded tricks designed to save hardware, and by the fact that the bulk of the previous investigations into microprogram optimization have used this simple model.

In this somewhat anecdotal chapter we list the extensions to our model which we feel would cover many, if not most, microprogrammed machines. For each such extension we comment on how serious a problem extending our methods to a machine with such a structure would be, and we make general comments on the extension.

Compatible Uses of Resources

As mentioned in Chapter 5, the resource usage model is not adequate to describe the situation when two MOPs both use a resource, but those uses are "compatible" and the MOPs do not clash. The mode settings of an ALU or a multiplexer are two common examples of compatible uses; MOPs with different required settings would clash, but those with the same would not. To describe our method of dealing with this situation effectively, we present our suggested method for dealing with resource conflict in general.

The Left and Right Resource Bit String

The innermost loop of a list scheduler might be expected to spend most of its time determining whether a MOP under consideration may be placed in the current partition. To do this, we propose that each MOP t have associated with it two bit strings, a "left resource string," LRES(t), and a "right resource string," RRES(t). The strings would have a field associated with each resource used, and would have to be formed only once per MOP, then never changed.

If a resource has one unit available which a MOP either uses or not, then we set a single bit field in both LRES and RRES for any MOP which uses it, and we clear the field for MOPs which don't use it. Then given a

partition P, we can define PARTITIONRES(P) = $\bigcup_{t_j \in P}$ LRES(t_j). A MOP, t_i, will conflict with P in the use of resources of this type if PARTITIONRES(P) $\cap$ RRES(t_i) $\neq \emptyset$, otherwise we can add t_i to P and union LRES(t_i) in with PARTITIONRES(P) to update it.

Resources with Non-unit Availability

When we have a resource with d units available, and a task which uses c units of it, the situation is somewhat more complex. We use a d bit wide field, and we set the right hand most c bits in RRES of the MOP, while clearing the other d-c bits in the field. For LRES, we set the left hand most c bits while clearing the rest. The field in PARTITIONRES(P) has the left hand most e bits set, where e is the sum of the usages of the resource of the MOPs already in P. Then, once again, for fields of this nature, a MOP t_i will conflict with P if PARTITIONRES(P) $\cap$ RRES(t_i) $\neq \emptyset$. Note that the case d=1 corresponds to the special case described in the previous paragraph. We point out that adding a MOP to a partition is a relatively complex operation, but need be done only once per MOP, whereas testing for conflicts requires only the simple (on most machines) bit intersection operation, but may be done much more frequently.

Testing for Compatible Resources

Testing for compatible resource usage presents added difficulties. Two MOPs which, say, read the same register, might want the same select bit setting on a multiplexer. It would be convenient to again be able to code fields corresponding to the multiplexer - or whatever resource - so that incompatible usages would cause the bit intersection to be non-zero, but compatible usages would produce a field of all zeros when intersected. At first glance, this would appear to be impossible. Indeed, the PUMA microassembler, which checks the legality of all microinstructions specified by the programmer, laboriously checks all such fields against the already placed ones; though, fortunately, to assemble this need be done only once per MOP.

We were able to come up with a trick for this, using an expansion of the exclusive or. In particular, suppose the resource state can be specified by n bits. We set aside a 2n bit field for the resource. For a task t_i which does not use this resource, we clear all 2n bits in the field in both LRES and RRES. If t_i does require a bit string, B, for the resource, we let the left hand most n bits in the field in LRES be B, and the right hand be not B, the bit-by-bit complement of B. In RRES, we reverse them, letting the right hand n bits be B. PARTITIONRES(P), then, will be the union of the LRES for those fields for all MOPs in

the partition, and once again, a MOP t_i will conflict with P if PARTITIONRES(P) $\cap$ RRES(t_i) $\neq \emptyset$. This works because bits B_j and B_j' are different only if (B_j and not B_j') or (B_j' and not B_j) is 1, that is, if their exclusive or is 1. If one field is all zeros, then it will pass the test with any other string.

The discerning reader may have noticed that except for the case of multiple units of a resource, the partition's total resource can be found by simply taking the union of the LRES of each of its components. The PUMA is a machine with several "compatible" resource usages, but no multiple units; thus for the PUMA, and probably many other machines, both total resource usage for the microinstruction being formed and resource conflict can be determined very quickly. This may require more than one word of bits, even on a wide-word machine, but still would be far faster than field-by-field checking.

Many-cycle MOPs

Our model assumes that all MOPs take the same amount of time to execute, and thus that all MOPs have a resource and dependency effect during only one cycle of our schedule. In many machines, though, the difference between the fastest and slowest MOPs is great enough that allotting all MOPs the same cycle time would slow down the machine considerably.

The PUMA, for example, allows two cycles for ADDs longer than 4 bits; if combinatorial multiply chips were incorporated into the design, the longest operation would be longer still. This is more an intrinsic function of the range of complexity of the MOPs available at the microprogram level of the machine, rather than something which is liable to change with improved hardware.

The presence of MOPs taking m>1 cycle presents little difficulty to the within basic block methods suggested here. The priority calculations all extend naturally to longer MOPs; in particular, one can break the MOP into an one-cycle subMOPs, with the obvious adjustments to the graph, and calculate priorities accordingly. Scheduling is also very straightforward; indeed, one of the attractions of list scheduling is that m-cycle MOPs are handled so naturally. When an m cycle MOP is scheduled at level ℓ, we also schedule it for levels $\ell + 1$, $\ell + 2$, ..., $\ell + m-1$. (If it conflicts with other long MOPs already in some of those cycles, we treat it as if it has a conflict in the cycle in which we are trying to schedule it.) The resource usages need not be constant for each of the m cycles; the more general case is a simple extension of our resource calculations - that is, usage may be defined differently for each cycle of the MOP.

Our beyond basic block methods have some difficulties

with long MOPs, though. Within a PATH, there again seems to be no problem, but suppose a two cycle MOP, with subMOPs t_i and t_i' is scheduled so that t_i is at the same level as a conditional jump that was previously below t_i and t_i'. Then t_i' would have to be copied into the off the path branch at the jump. t_i' however, would have to be in the first cycle of the branch not taken, and the cycle containing t_i' would have to remain the first cycle. Without going deeply into details, this would require changes to and/or restrictions on the algorithms presented to account for copying into already optimized blocks, other tasks copied past the same conditional jump, etc.

Polyphase MOPs

In some systems, the MOPs represented at the micro-program language level may be further regarded as having sub-microcycles. This has the advantage that while two MOPs may both use the same resource, typically a bus, they may use it during different sub-micro-cycles; thus the apparent resource conflict may not exist, and the MOPs could be scheduled in the same cycle. This has no effect on our scheduling methods other than to complicate the resource conflict relation. We remark that the PUMA has no sub-microcycles.

Variable Instruction Formats

Some microprogrammable machines have more than one micro-instruction format. The format chosen for an instruction will allow certain subsets of the MOPs to be chosen for that instruction. In particular, each field of an instruction may allow any of several MOPs to be placed in the field, and each MOP may be placed in any of several fields.

To deal with this, we suggest first that immediately choosing one format in which to place the first MOP scheduled for the cycle seems too risky. We suggest instead that a list be kept of the formats for which the so far selected MOPs are resource compatible and that each time a new MOP is added, the formats which could not accommodate the MOP be deleted. The choice of which field to bind a MOP to within a format, however, greatly resembles the resource allocation we have been considering right along. Indeed, one would expect that the MOPs form equivalence classes with respect to which fields they may occupy, and that selecting a field for a MOP is equivalent to assigning it one unit of resource. For a discussion of this, see [WOOD78].

Necessarily Simultaneous MOPs

At the microcode level it is often possible to specify

operations which, when performed in parallel, have no sequential equivalent. A common example is the register swap: If two registers are both made of master-slave flip-flops, and each feeds one of the inputs of the other, the two may be able to exchange contents in one cycle, using no intermediate registers. That is, if A=B and B=A are both legal MOPs, the microinstruction A=B, B=A would express an operation not expressible as a sequence of single MOPs.

To accommodate the above, we suggest simply that the source code be allowed to contain lines which are sets of MOPs. Throughout optimization the set is treated as one MOP, and never broken apart.

Special Case Precedence

Our model uses the registers read and written to determine the precedence relation on the MOPs. One could expect this to be an inadequate description of the relation on most machines. As a result, it is likely that in any implementation special procedures would add edges to the dag produced by the routines described herein.

A significant example of this exists in the PUMA. In the PUMA, tests of some data may be done either before or after the data enters the register simply by specifying the appropriate test. Thus if MOP t_i writes the tested data, and t_j tests it, we can say $t_i <= t_j$ rather than

$t_i < t_j$, as long as we are careful, during object code generation, to specify the correct condition code.

We have ordinarily not been interested in code transformations of this sort, requiring instead that they be programmer specified. This particular situation is quite common however, and since it adds no difficulty to our methods, other than some simple pre- and post-processing, it would seem worth implementing in a PUMA optimizer. (Or, in general, in any environment in which such a transformation may be specified within the informational limits of the data-precedence graph.)

Flow Control Extensions and Restrictions

Our model assumes that, as is true of the PUMA, each instruction may specify two address as potential successors, one to jump to if the specified test is true, the other if it is false. Our beyond block methods use this flexibility in the rearrangement of the order of conditional jumps; without it the situation is a bit more difficult.

If a machine has a single jump address, with a fall through as the alternative, the methods in the previous chapter could lead to a situation in which several instructions want to fall through to the same instruction, which would not be permitted. In that case, one of the contending instructions would be permitted to fall through to the

intended target and the others would fall through to copies of the intended target. If the intended target were also a conditional jump, then the copies would again contend with the original, and copies of the next intended fall through instruction would have to be made and so on. When a non-conditional jump occurred, we would have all the streams use the branch address mechanism to rejoin.

Despite the dramatic sound of the above, it is our belief that, for most code, very few copies would be generated. This would be particularly so if the machine permitted reversible conditions, that is, a branch on the negation of each condition. Since this would require only one gate (an xor, say), and one extra condition field bit in the microprogram memory, it would seem worth doing. In the PUMA, the tradeoff of microprogram width versus height is not clear, and we were not able to convince ourselves whether the two address jumps were economical or not.

Finally, we mention the extension of the flow mechanism to include microprogram subroutine calls; we simply note that our recursive descent seems appropriate for handling the calls. Calls could be treated as one task which somewhat resembles the combination of a loop and a jump. That is, one task on the dag of a PATH containing a call would stand for the call, its data dependency dependent upon what registers were live entering and exiting the call. Tasks

would probably not be allowed into the subroutine, but tasks scheduled in the same cycle as the call would be processed before the call was made at the end of the cycle.

Subroutine calls are of particular interest here because the microprogram sequencer chips now being made have return address stacks built into them.

Annotated Bibliography

(# marks sources not referenced in text)

Papers concerning microprogram optimization

[AGER76] AGERWALA, T. "Microprogram optimization: a survey." Trans. Comp. 25,(Oct. 76), 962-973.

This paper surveys four broad categories of microprogram optimization: word dimension reduction, bit dimension reduction, heuristic reduction, and state reduction. The author surveys most of the papers surveyed here (on microprogram optimization), and concludes that "very few techniques exist that can be profitably applied in any practical environment."

[ASTO71] ASTOPAS, F.; AND PLUKAS, K. I. "Method of minimizing computer microprograms, "Automatic Control 5,4 (1971) 10-16

This paper is evidently the first to present an algorithm for an enumerative parallelization.

[DASG76] DASGUPTA, S.; AND TARTAR, J. "The identification of maximal parallelism in straight-line microprograms," IEEE Trans. Comp. 25,10 (Oct. 76) 986-991.

This paper claims (falsely) to present an optimal, polynomial time parallelizer. For counterexamples to the optimality, see the March 1978 correspondence in the same journal. The non-optimal algorithm does not appear useful, as even very simple cases can produce extra cycles.

[DASG77] DASGUPTA, S. "Parallelism in loop-free microprograms," in Information Processing 77, North-Holland (1977) 745-750.

An algorithm is given for moving MOPs from one basic block to another, but only when the two blocks have the property that one is executed if and only if the other is.

[DASG78] DASGUPTA, S. "Comment on the identification of maximal parallelism in straight-line microprograms," IEEE Trans. Comp., C-27, 3 (March 78) 285-286.

Letter clarifying the non-optimality of the algorithm in [DASG76].

[GRIS78]# GRISHMAN, R. "The structure of the PUMA Computer System," U.S. Department of Energy Report, Courant Mathematics and Computing Laboratory, New York University, N.Y., N.Y. 1978.

An overview of the PUMA system and the structure of the central processor are presented in detail. A chapter is devoted to microprogramming the PUMA and the entire CDC 6600 emulator microprogram is reproduced. (All references to PUMA are actually to this report.)

[KLEI71]# KLEIR, R. L.; AND RAMAMOORTHY, C. V. "Optimization strategies for microprograms," IEEE Trans. Comp. 20, 7 (July 71) 783-794.

Only optimization of sequential microprograms is discussed here. Standard compiler techniques are extended to microprograms and various micro-programming system aids and techniques are explored.

[RAMA74] RAMAMOORTHY, C.V.; AND TSUCHIYA, M. "A high-level language for microprogramming," IEEE Trans. Comp. 23, 8 (Aug. 74) 791-801.

This paper concerns the development of a high level microprogramming language, SIMPL. Along the way, a non-optimal parallelizing algorithm is presented, evidently the first polynomial time such algorithm. Although it has since been recognized as being too likely to lead to non-optimal code, this algorithm laid the groundwork and set the terminology for others.

[ROSS75] ROSSMAN, G.E.; FLYNN, M.J.; McCLURE, R.M.; AND WHEELER, N.D. "The technical significance of user microprogrammable systems," in Microprogramming, IEEE Catalog No. 75CHO98-1-1C, 249-294, IEEE NY, 1975.

A survey of various user microprogrammable systems is accompanied by an investigation into the features of user microprogramming and their effects on performance and usability. Includes useful tables, a bibliography which includes references to manuals, and a survey of user experiences with such systems.

[TABA74] TABANDEH, M.; AND RAMAMOORTHY, C.V. "Execution time (and memory) optimization in the microprogram," (SIGMICRO) 7th Annu. workshop on microprogramming preprints supplement (Sept. 30-Oct. 2, 1974)

This paper continues the discussion of the high level microprogramming language SIMPL [RAMA74]. More details of the compiler are given, along with a non-polynomial time algorithm for producing optimal parallelism. Reference is made to, and a sketchy description given to a non-optimal heuristic algorithm.

[TOKO77] TOKORO, M.; TAMURA, E; TAKASE,K; AND TAMARU, K. "An approach to microprogram optimization considering resource occupancy and instruction formats," in (SIGMICRO) 10th Annual Workshop on Microprogramming (1977) 92-108

A rather involved algorithm considers micro-operations to be represented in two dimensions, resources vs. cycles.

[TOKO78] TOKORO, M.; TAKIZUKA, T; TAMURA, E; AND YAMAURA,I. "A technique of global optimization of micrograms," in (SIGMICRO) 11th Annual Microprogramming Workshop (Nov. 19-22, 1978) 41-50.

A method of optimization beyond blocks is suggested. The method involves optimizing blocks and then trying to improve via a small catalog of interblock motions.

[TSUC74] TSUCHIYA, M.; AND GONZALEZ, M.J. "An approach to optimization of horizontal microprograms," (SIGMICRO) 7th Annu. workshop on microprogramming preprints (SEPT. 30-OCT. 2, 1974) 85-90.

This paper discusses a polynomial time non-optimal method of parallelization. While a careful algorithm is presented for the general problem, including the updating of a least upper bound, the actual details of the parallelizing are omitted.

[WOOD78] WOOD, G. "On the packing of micro-operations into microinstruction words, "(SIGMICRO) 11th Annual Microprogramming Workshop (Nov. 19-22, 1978) 51-55.

Short paper compares a few already suggested strategies for optimization and suggests a simpler alternative which amounts to list scheduling using the number of successors as a priority function. Most interesting for comments which show a different perspective on the problem from that taken in previous investigations.

[YAU74] YAU, S.S.; SCHOWE, A.C.; AND TSHUCHIYA, M. "On storage optimization of horizontal microprograms," (SIGMICRO) 7th Annu. workshop on microprogramming preprints (Sept. 30-Oct. 2, 1974) 98-106.

Two parallelizing algorithms are given, one enumerative, the other not (the latter simply uses a heuristic function to select one path of the former). The paper appears more careful and thought out than many others presenting such algorithms.

Papers and books on processor scheduling

[ADAM74] ADAM, T.L.; CHANDY, K.M.; AND DICKSON, J.R. "A comparison of list schedules for parallel processing systems, "Comm. ACM 17,12 (Dec. 74) 685-690.

The authors use precedence graphs (from programs and randomly produced) to test various list schedule producing strategies against optimal schedules, against a computed lower bound [FERN73], and against each other. Schedules are produced for systems of 2-5 identical processors. Among those tested, the latest partitioning is the best, and is nearly optimal. No significant difference in performance is noted for strategies when random graphs are used instead of "real" graphs.

[COFF72] COFFMAN, E.G. Jr.; AND GRAHAM, R.L. "Optimal scheduling for two processor systems," Acta Informatica 1 (1972) 200-213.

Among other results, particularly bounds, the authors produce an algorithm for generating optimal list schedules for tasks of equal duration done on two identical processors. The algorithm is faster than o(n-cubed) on the number of tasks.

[COFF76] COFFMAN, E.G. Jr. (Ed.), "Computer and job-shop scheduling theory," John Wiley & Sons, N.Y., 1976.

This book contains chapters on most of the important topics in deterministic processor scheduling. An introductory chapter (by Coffman) sets the definitions for the rest of the book and presents a survey of results through 1976; the results are also very conveniently tabulated.

[CONW67] CONWAY, R.W.; MAXWELL, W.L.; AND MILLER, L.W. "Theory of scheduling," Addison-Wesley Publ. Co. Inc., Reading, Mass., 1967.

An earlier work which predates most of the difficult work done, thus mostly useful for historical and definitional purposes. The problem is viewed from a management science perspective, and various applications are discussed.

[ECKE78] ECKER, K. "Analysis of a simple strategy for resource constrained task scheduling," 1978 International Conference on Parallel Processing, 181-183.

A simulation compares two simple strategies for resource constrained scheduling without data-precedence.

[FERN73] FERNANDEZ, E.B.; AND BUSSEL, B. "Bounds on the number of processors and time for multiprocessor optimal schedule," IEEE Trans. Comp. C22, 8 (Aug. 73) 745-751.

Lower bounds are calculated both on the number of cycles a schedule for a precedence graph of tasks will take given n processors, and on the number of processors necessary to process the graph in critical path time. Both bounds are improvements over all previously known bounds.

[GARE75] GAREY, M.R.; AND JOHNSON, D.S. "Complexity results for multiprocessor scheduling under resource constraints," SIAM J. Comput. 4,4 (Dec. 1975) 397-411.

Various narrow restrictions of processor scheduling with resource constraints are shown to be NP-complete; thus almost all cases are.

[GONZ77] GONZALEZ, M.J. Jr. "Deterministic processor scheduling," Computing Surveys 9,3 (Sept. 77) 173-204.

[ULLM73] ULLMAN, J.D. "Polynomial complete scheduling problem," in Fourth Symposium Operating System Principles, 1973, 96-101. (Published as Operating Systems Rev. 7,4 ACM N.Y.)

Two scheduling problems are shown to be np-hard. The first is scheduling unit time tasks on a system of identical processors where the number of identical processors is not fixed but is an input to the algorithm. The second is scheduling one and two unit time tasks on two identical processors.

Other references

[AHO74] AHO, A.V.; HOPCROFT, J.E.; AND ULLMAN, J.D. "The design and analysis of computer algorithms," Addison-Wesley, Reading, Ma., 1974.

Standard algorithms text.

[AHO77] AHO, A.V.; AND ULLMAN, J.D. "Principles of compiler design," Addison-Wesley, Reading, Ma., 1977.

Standard compiler reference. Chapters 12-14 contain much information on data-flow analysis.

[HANA76] HANAN, M.; WOLFF, P.K. Sr.; AND AGULE, B.J. "A study of placement techniques," Journal of Design Automation and Fault-Tolerant Computing, 1,1 (Oct. 76) 28-61.

Seven chip placement algorithms are applied to six large problems and the results compared.

[HECH77] HECHT, M.S. "Flow analysis of computer programs," Elsevier North-Holland, NY, NY, 1977.

A definitive reference on data-flow analysis; includes a short chapter on node splitting.