PARALLEL COMPUTATION: SYNCHRONIZATION, SCHEDULING, AND SCHEMES

by

JEFFREY MARTIN JAFFE

B. S., Massachusetts Institute of Technology
(1976)

S.M., Massachusetts Institute of Technology
(1977)

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE
DEGREE OF

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

(AUGUST 1979)

Signature redacted

Signature of Author .....
Department of Electrical Engineering and Computer Science, August 10, 1979

Signature redacted

Certified by .....
Thesis Supervisor

Signature redacted

Accepted by .....
Chairman, Department Committee

PARALLEL COMPUTATION: SYNCHRONIZATION, SCHEDULING, AND SCHEMES

by

JEFFREY MARTIN JAFFE

Submitted to the Department of Electrical Engineering
and Computer Science on August 10, 1979 in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy

ABSTRACT

There are two primary means of resource allocation in computer systems. There is the powerful mechanism of using a centralized resource manager that allocates the resources. An apparently weaker mechanism is for the asynchronous processes of the system to allocate resources with some type of message passing between themselves. This thesis provides a unifying treatment of these two methods. It is shown that a managed system may be simulated by the processes. As a corollary, a wide variety of synchronization algorithms may be accomplished without a manager.

The simulation works correctly even in an environment with unreliabilities. Processes may die in an undetectable manner and the memory of the system may be faulty. Thus our general simulation provides the first known algorithm for synchronizing dying processes in a faulty memory environment.

Scheduling jobs on processors of different capabilities is studied. Algorithms are presented for two machine environments that have better performance than previously studied algorithms. These environments are processors of uniformly different speeds with partially ordered tasks and unrelated processors with independent tasks. In addition, the class of all preemptive schedules for uniform processor systems are studied. These schedules are shown to be more effective than previous analyses.

Scheduling jobs on functionally dedicated processors is introduced. Algorithms are presented for scheduling jobs on such processors, both in the case that the processors are equally fast and in the case that they are of different speeds.

The expressive power of the data flow schemes of Dennis is evaluated. It is shown that data flow schemes have the power to express an arbitrary determinate functional. The proof involves a demonstration that "restricted data flow schemes" can simulate Turing Machines. This provides a new, simple basis for computability.

THESIS SUPERVISOR: Albert R. Meyer
TITLE: Professor of Electrical Engineering and Computer Science

Keywords

Chapter 2:

distributed control
fair mutual exclusion
fault tolerant computing
mutual exclusion
process systems
resource allocation
resource manager
simulation
stable state
sychronization
unreliable processes

Chapter 3:

independent tasks
list schedules
maximal usage schedules
nonpreemptive scheduling
partially ordered tasks
preemptive scheduling
scheduling
task systems
typed task systems
uniform processor system
unrelated processors
worst case performance bounds

Chapter 4:

computability
data flow schemes
effective functionals
r.e. program schemes
Turing machines

Acknowledgements

During my three years as a graduate student the primary source of direction has been provided by my thesis supervisor Albert Meyer. He has suggested exciting research directions to pursue, encouraged me to simplify and clarify my ideas, and provided inspiring technical contributions. Most importantly, he has unselfishly devoted an enormous amount of time to the development of the concepts in this thesis and to the presentation of the results.

My two thesis readers are to be credited for sparking my interest in the general topics covered in this thesis. Ron Rivest deserves primary credit for training me in the techniques of algorithm design and analysis and giving me much feedback on the presentation of the thesis. The data flow machine project of Jack Dennis helped develop my interest in parallel computation.

There are many others that have contributed to my research by providing new insights or working together with me on some of the technical results. While I cannot begin to thank them all for the various levels of help they have provided I would like to single out those who have made the most outstanding contributions: A. Baratz, E. Davis, I. Greif, D. Harel, E. Jaffe, D. Kessler, A. LaPaugh, E. Lloyd, M. Loui, C. Papadimitriou, V. Pratt, M. Rabin, A. Shamir, and G. Stark.

Since no research can be done in a vaccuum, I must thank all of those friends and family whose kinship and support are necessary ingredients of a research effort. Most importantly I must express my eternal gratitude to my wife, Esther for all of her help. It is a rare blessing to be close to someone that provides not only the necessary love and moral support at home, but also a vast reservoir of technical assistance in my work.

This thesis was prepared with the support of a National Science Foundation graduate fellowship, and National Science Foundation grant no. MCS77-19754.

Table of Contents.

Abstract 2
Keywords 3
Acknowledgements 4
Table of Contents 5
Index of Theorems and informal description of content 7
Index of Lemmas 9
Index of Equations 10
Index of Figures and Tables 11

1. Introduction 12
    1.1 Major goals and accomplishments of thesis 12
    1.2 Reasons for parallel computation 15
    1.3 Some problems associated with parallel computation 19

2. Synchronization 26
    2.1 Introduction 26
        2.1.1 Background and motivation 26
        2.1.2 Unreliability assumptions and their motivation 30
    2.2 Basic definitions 34
        2.2.1 Process systems and the definition of simulation 34
        2.2.2 Properties of simulation 41
        2.2.3 Process systems with unreliable memory 45
        2.2.4 Managed system of processes and the main results of Chapter 2 52
    2.3 An Example 56
    2.4 Cooperating system of n simulators 59
        2.4.1 Overview of the system 59
        2.4.2 Memory of the cooperating system 64
        2.4.3 State transitions of the first n processes 71
        2.4.4 Simulation of the RM (state transitions of the simulators) 74
    2.5 Proof of Theorem 2.1 81
    2.6 Discussion of unreliability properties 104
    2.7 Open problems and further work 109

3. Scheduling Theory 111
    3.1 Introduction 111
    3.2 Scheduling tasks on processors of uniformly different speeds 119
        3.2.1 Basic definitions and models 119
        3.2.2 Nonpreemptive scheduling of tasks on uniform processors 123
        3.2.3 Preemptive scheduling of tasks on uniform processors 143
        3.2.4 Scheduling with limited information 159
    3.3 Nonpreemptive scheduling of independent tasks on unrelated processors 162
    3.4 Scheduling tasks on processors of different types 185

3.4.1 Basic definitions and models 185
3.4.2 Nonpreemptive scheduling of tasks on equally fast processors of different types 188
3.4.3 Nonpreemptive scheduling of tasks on processors of different types and different speeds 196
3.4.4 Maximal usage preemptive scheduling of tasks on processors of different types and different speeds 214
3.5 Open problems and further work 216

4. Schemes 219
4.1 Introduction 219
4.2 Syntax and semantics of schemes 222
4.3 Programming techniques 228
4.4 Simulating Turing Machine computations with restricted data flow schemes 235
4.5 Simulating arbitrary r.e. program schemes with data flow schemes 243
4.6 Conclusion and future work 255

References 256
Biographical note 264

Index of Theorems and informal description of content

Theorem 2.1	Cooperating systems simulate managed systems	54
Theorem 2.2	Every history for a cooperating system matches some history for the corresponding managed system	97
Theorem 2.3	Managed systems may be partially simulated by systems with n processes	101
Theorem 2.4	Fair mutual exclusion may be accomplished even with dying processes and unreliable memory	101
Theorem 2.5	Cooperating systems without errors simulate managed systems	102
Theorem 3.1	List schedules on the fastest i processors are at most $1 + 2\sqrt{m}$ times worse than optimal where m is the number of processors	130
Theorem 3.2	List schedules on the fastest i processors are at most $\sqrt{m} + O(m^{1/4})$ times worse than optimal	131
Theorem 3.3	Any preemptive schedule may be transformed into a maximal usage preemptive schedule in polynomial time	152
Theorem 3.4	A bound on maximal usage preemptive schedules related to Theorem 3.2	153
Theorem 3.5	Maximal usage preemptive schedules are at most $\sqrt{m} + (1/2)$ times worse than optimal	157
Theorem 3.6	An algorithm for independent tasks on unrelated processors is at most $2.5\sqrt{m}$ times worse than optimal	176
Theorem 3.7	An algorithm which is similar to the one analyzed in Theorem 3.6 is at most $2.41\sqrt{m}$ times worse than optimal	178
Theorem 3.8	An algorithm which is similar to but slower than the algorithm analyzed in Theorem 3.6 is at most $1.5\sqrt{m}$ times worse than optimal	180

Theorem 3.9	List schedules for typed task systems are at most $k+1$ times worse than optimal where k is the number of types	188
Theorem 3.10	List schedules for typed task systems on processors of different speeds are at most k plus the maximum ratio of speeds of processors of the same type worse than optimal	200
Theorem 3.11	A speed independent bound for list schedules on a subset of processors for typed task systems	203
Theorem 3.12	A different speed independent bound than that of Theorem 3.11	204
Theorem 3.13	A bound on maximal usage preemptive schedules for processors of different types related to Theorems 3.11 and 3.12	215
Theorem 4.1	Turing Machines may be simulated by restricted data flow schemes	237
Theorem 4.2	R.e. program schemes are equivalent to data flow schemes	247

Index of Lemmas

Lemma 2.1 41
Lemma 2.2 41
Lemma 2.3 42
Lemma 2.4 (progress lemma) 85
Lemma 2.5 92
Lemma 2.6 97
Lemma 3.1 125
Lemma 3.2 125
Lemma 3.3 154
Lemma 3.4 154
Lemma 3.5 168
Lemma 3.6 170
Lemma 3.7 171
Lemma 3.8 171
Lemma 3.9 172
Lemma 3.10 176
Lemma 3.11 189
Lemma 3.12 189
Lemma 3.13 198
Lemma 3.14 198
Lemma 4.1 (finite translation lemma) 230

Index of Equations

Equation (1) 127
Equation (2) 128
Equation (3) 128
Equation (4) 131
Equation (5) 131
Equation (6) 134
Equation (7) 135
Equation (8) 155
Equation (9) 156
Equation (10) 156
Equation (11) 156
Equation (12) 156
Equation (13) 173
Equation (14) 173
Equation (15) 173
Equation (16) 174
Equation (17) 174
Equation (18) 174
Equation (19) 174
Equation (20) 175
Equation (21) 175
Equation (22) 175
Equation (23) 176
Equation (24) 199
Equation (25) 199
Equation (26) 200
Equation (27) 201
Equation (28) 201
Equation (29) 202
Equation (30) 202
Equation (31) 202
Equation (32) 204
Equation (33) 204

Index of Figures and Tables

Table 2.1 66
Table 2.2 67
Table 3.1 136
Table 3.2 137

Figure 3.2.1 140
Figure 3.4.1 192
Figure 3.4.2 207
Figure 3.4.3 209
Figure 3.4.4 212
Figure 4.3.1 229
Figure 4.3.2 229
Figure 4.3.3 231
Figure 4.3.4 231
Figure 4.3.5 231
Figure 4.3.6 231
Figure 4.3.7 231
Figure 4.3.8 234
Figure 4.4.1 242
Figure 4.4.2 242
Figure 4.5.1 242
Figure 4.5.2a,b 250
Figure 4.5.2c 252
Figure 4.5.2d 254
Figure 4.5.3 254

1. Introduction

1.1. Major goals and accomplishments of thesis

There are three major topics covered in this thesis. They are resource allocation or synchronization of parallel processes, job scheduling, and schemes. In the introduction we give a very brief description of the main contributions of this thesis. We then give an overview of the usages of parallel compuation and describe some problems that designers of parallel systems are confronted with. More extensive introductions to the three main topics of the thesis may be found in Section 2.1 (synchronization), Section 3.1 (scheduling), and Section 4.1 (schemes).

1.1.1 Synchronization

There are two primary means of resource allocation in computer systems. There is the powerful mechanism of using a centralized resource manager that allocates the resources. Managed systems are usually easy to design and easy to prove to be correct. An apparently weaker mechanism is for the asynchronous processes of the system to allocate resources with some type of message passing between themselves. This method has the advantage of not needing a manager to "poll" the processes even at times that no resource is needed. This thesis provides a unifying treatment of these two methods. It is shown that a managed system may be simulated by the processes. As a corollary, a wide variety of synchronization algorithms may be easily designed without a manager.

The simulation works correctly even in an environment with unreliabilities. Processes may die in an undetectable manner and the memory of the system may be faulty. Thus our general simulation provides the first known algorithm for synchronizing dying processes in a faulty memory environment.

1.1.2 Scheduling

Scheduling jobs on processors of different speeds is studied. Algorithms are presented for two machine environments that have better performance than previously studied algorithms. These environments are processors of different speeds with partially ordered tasks and unrelated processors with independent tasks. These environments will be defined precisely in Chapter 3. In addition, the class of all preemptive schedules for uniform processor systems are studied. These schedules are shown to be more effective than previous analyses.

Scheduling jobs on functionally dedicated processors is introduced. Algorithms are presented for scheduling jobs on such processors, both in the case that the processors are equally fast and in the case that they are of different speeds.

1.1.3 Schemes

The expressive power of the data flow schemes of Dennis is evaluated. It is shown that data flow schemes have the power to express an arbitrary determinate functional. The proof involves a demonstration that "restricted data flow schemes" can simulate Turing Machines. This provides a new, simple basis for computability.

We proceed with an overview of parallel computation in general. Section 1.2 outlines the basic types of machine environments that we consider under the title of parallel computation. Section 1.3 discusses the three topics of the thesis in somewhat greater detail. We discuss the relevance of these issues in different parallel environments. We also explore the

relationships between the primary issues of synchronization, scheduling, and schemes, and secondary issues such as fault tolerant computing, distributed control, and heterogeneous systems.

1.2 Reasons for Parallel Computation

We identify three different sources of interest in parallel computation: "problem oriented", "machine oriented", and "community oriented" (distributed).

1.2.1 Problem Oriented Parallel Computation

Part of the original interest in parallel computation arose from time-consuming, "number crunching" programs that could not be executed within a reasonable amount of time on a single conventional machine. Many of these programs solve extremely important problems such as partial differential equations and linear programming. In order to solve these important problems one is often willing to buy a large amount of computing resources if this could only help obtain the solution. In certain applications (e.g. FFT) special purpose devices have been built to solve the problem. However, for other less pervasive applications, using more than one general purpose computer is often the cost-effective means of solution. By executing different portions of a program concurrently one substantially reduces the finishing time of a program.

This effort - to understand how to divide programs into different portions capable of being executed concurrently - promises to continue despite further advances in hardware technology. While hardware advances consistently extend the frontier of "which programs may be finished quickly", it seems unlikely that all important problems will ever be solvable quickly. As soon as improved technology and reduced hardware costs permit us to efficiently solve yesterday's intractable problems, a new set of intractable problems invariably appears on the horizon. Indeed, a large class of important decision problems are NP-complete. NP-complete problems are widely believed to require an

exponential amount of time for their solution [20] in terms of the size of the particular problem instance (i.e., for instances of size n, the time required is 2^n). In addition, there is a class of somewhat less important problems which provably require exponential time. For these problems, even moderately sized instances are beyond the reach of today's technology, and larger instances which arise in practice, will surely continue to be difficult.

It is interesting to note that the NP-complete problems lend themselves to parallel techniques; they can be solved quickly if one can explore many possible situations in parallel. While this use of parallel computation does not circumvent the apparent exponential growth in the time required for solving the NP-complete problems, it does permit one to solve larger instances than would be otherwise possible.

1.2.2 Machine Oriented Parallel Computation

Often the motivation for introducing parallelism into a system is the "dual" of problem oriented parallel computation. In a problem oriented environment no individual resource is sufficiently powerful to attack the problem at hand. The "dual" of this is that available resources are overly powerful for any single problem. It is then advantageous to divide the use of the powerful computer among many problems.

This situation has led to a successful series of multiprocessing and multiprogramming computer environments. Many users simultaneously execute different jobs, while a central controller allocates processing time and other machine resources. Typically, some of the users are in an interactive mode; i.e. the inputs to the program are inserted by the users during program execution (e.g., editing programs). This slow input process allows the system

to serve many users at once; each user's program needs only a small amount of CPU time per unit real time. Even programs that are not executed interactively do not utilize all of the machine resources simultaneously. Thus, if one program issues numerous Input/Output instructions, other programs may use the main processor and memory.

There is also some overlap between problem oriented and machine oriented environments. Consider a given problem oriented environment with a large cost overhead due to the purchase of extensive computing resources. It may be useful to amortize that cost by permitting low priority computing to take place in the "background" - to be executed if system resources would otherwise be unused.

1.2.3 Community Oriented Parallel Computation

Community oriented parallel computation is a relatively new development. In a computer network, a large number of computers are separated by physical distances and linked by communications lines. This interconnection scheme has become practical for a large number of applications due to the astonishing decrease in costs for small computers. A network provides a powerful, flexibile computing system to a large community of users. The dedication of a computer to a small number of local users provides them with computational power free from delays that are inherent with processor sharing in large multiprocessor systems. On the other hand, each user in the system still has access to a wide variety of system resources. In particular, each user has access to any resource located anywhere in the network of computers. Thus (for example), the communications network may enable the user to copy a "mailing list file" from one remote location, process it locally to removed an

undesirable portion of the list, and have the output produced at a different location; perhaps the single network location with a "mailing label" output device.

Another advantage of distribution is that most of the system remains operational if some of the local computers "crash" and have to be withdrawn from the system. This contrasts with the classical computer model that relies heavily on the reliability of the single central processor.

To recapitulate, the three motivating forces for parallel computation that are mentioned here have a number of common high-level goals. The first is to bring more processing power to bear on a particular environment. In a problem oriented environment, this occurs with the use of faster or specialized processors after one completes the programming task of parallelizing the programs. The multiprogrammed environment acquires more *real* computational power by insuring that existing power is not wasted. A distributed environment enables users to use dedicated machines without giving up the capability of a varied resource system.

A common goal of machine and community oriented computation is to attain a wider distribution of computer resources. Multiprocessor systems give each user relatively equal access. Distributed systems provide easier access to users located near the resources.

1.3 Some Problems Associated with Parallel Computation

1.3.1 Synchronization

One of the more challenging problems in the field of parallel computation is in trying to program moderately difficult algorithms in a distributed manner. That is, trying to establish protocols between different centers of computational power that enable the centers to cooperate on the solution to a particular task. A center of computational power may be a node in a network in a distributed system, or a process in a multiprocessor system. Decentralized algorithms are advantageous as they enable computational tasks to be finished quickly. Specifically, if the distribution is done properly, the completion of the algorithm may not be hampered by a center of computational power which is slow or busy doing other work. The large amount of service work done is one of the inefficient aspects of a central controller.

Chapter 2 of this thesis is devoted to such a "distributed control" program. What is more imortant than the fact that this is a distributed control problem is that it provides a new solution to a practical problem that has been widely studied (e.g., [15,17,37,53]). The problem is to synchronize resource assignment in a multiprocess system. Even in a uniprocessor machine it has long been felt that this should be solved without the central controller [15]. Since the central controller is busy with other time consuming tasks, it is helpful to remove some responsibilities from its domain. There is no reason for the controller to continually poll the users trying to establish if any resource requests have been made.

Some other solutions to this problem in distributed control may be found in [15,16,34,35]. The solution presented here takes hardware reliability into account. The solution works even if there is a failure of any one memory

unit. Also, slow processes do not degrade the efficiency of our solution. A slow process is modelled by a process that stops entirely. The solution continues to work even if some of the processes stop entirely and thus does not wait for a slow process (since a slow process will appear to have stopped entirely). Of course, there is no way of speeding up the system if a slow process is using the resource. In summary, there is a dual fault protection - memory may be unreliable, and processes may die (i.e. be slow) without any visible signs of death.

The main result is that we show that in the face of unreliability, the processes may simulate a manager of resources. Since it is a *general* simulation, it serves as a paradigm for other issues that arise in parallel computation. First, there is a general need to develop techniques of programming in a distributed manner. Our techniques should be generally applicable as they are not *ad hoc* mechanisms that solve only the specific problem of devising a fair synchronization protocol. Rather the techniques are mechanisms which enable the processes to cooperate to simulate the actions of a central controller, even if there is no controller, and memory and processes are unreliable. We note, however, that the solution itself is not quite usable to assign resources in distributed environments as it uses global read/write variables.

There is one final arena in which we believe that our protocols provide an excellent pedagogical model. This is the area of parallel program verification. To prove a complicated parallel program correct is quite difficult, as one must consider all possible interleavings of when which processes executed their instructions. There have been a number of suggested proof techniques in the literature.

The proof that our protocols simulate a central controller uses a "stable state" technique. At certain times, the current states of the processes and values in memory provide a clue as to what the system behavior will be in the immediate future. These times are defined to be the stable states. The constraints on the system at stabilization guarantee that the system will again become stable. Also, these constraints enable us to predict the behavior until the next stabilization.

1.3.2 Scheduling

One of the classical problems in the design of parallel systems is the scheduling of a large number of concurrently executable tasks on a fixed set of processors. The order of execution is an important consideration if one wants the jobs to be completed quickly [24]. A poor scheduling policy squanders the potential benefit of a parallel system. The scheduling problem arises in all three machine environments described in Section 1.1.

This problem has usually been attacked under the assumption that all of the machine's processors are identical [23]. This is a reasonable assumption in certain systems, and it is relatively easy to develop fast, near optimal algorithms in such a system.

In Chapter 3, scheduling algorithms are developed for complex machine environments with processors of different speeds. This problem has been considered in [28,30,43], but the known algorithms are not nearly as effective as in the identical processor case. The assumption that a system has processors of different speeds is widely applicable. For example, in problem oriented parallel computation a computer facility may have different machines with different capabilities. In a multiprocessor system, it is conceivable

(although perhaps not likely) that the computer has processors of different speeds. In a distributed system it is likely that the processors of the system be different. Also, the time required by a specific job on a specific processor (in a distributed system) must include some measure of communication cost. Thus identical processors are viewed as nonidentical from the system's point of view.

Sections 3.2.2 and 3.2.3 analyze the situation where the relative speed of any pair of processors is fixed (uniform) irrespective of the task to be executed. Both preemptive and nonpreemptive algorithms are presented for the scheduling of a partially ordered set of jobs on uniform processors. In both cases, approximation algorithms are developed whose worst case performance is better than known algorithms.

While the machine model with uniform processors is somewhat amenable to analytic, worst case analysis, the non-uniform (unrelated) case is much harder to analyze. The unrelated case models machines of specialized capabilities. In that environment the relative speeds of the processors depend on the task. This type of specialization may be used to model distributed computing, for example, even if the local processors are identical. As mentioned above, the time requirement of a task on a machine, M, depends on the relative distance in the network between the source of the task and M. Since this environment is more complex, the major results presented treat the situation that there is no partial order on the tasks. In that case, an algorithm is presented whose worst case performance is better than known algorithms.

The limiting case of a processor with specialized capabilities is a processor which is capable of executing certain tasks but not others. These types of processors are incorporated into the design philosophy of machines

such as the data flow machine [14]. This is a general purpose computer that contains many specialized processors. The simplicity of these processors permits one to install many of them at low cost. Scheduling problems on such sets of processors are studied in Section 3.4.

There is no active scheduler in current data flow computer architectures. Nevertheless, the scheduling results of Section 3.4 serve as a useful analysis tool. The scheduling algorithms analyzed include the passive scheduling that exists in the data flow architecture.

Whether or not a machine architecture permits the use of an active scheduler, one does not want to spend too much time on job scheduling, since the time spent on scheduling may be costly. Some of the results of Chapter 3 are applicable even to systems in which one wants to expend little effort in scheduling. These results suggest that if a processor is sufficiently slow, then a better worst case performance is obtainable if it is *never* used. Furthermore, techniques are provided for deciding which processors are sufficiently slow that they should never be used. This type of scheduling result is actually a machine design consideration - the machine should not be designed with such slow processors. Thus the use of these algorithms presents a one time cost which need not be repeated for every set of tasks.

1.3.3 Schemes

Synchronization and scheduling may be viewed as special cases of the fundamental question of how to develop and execute programs that are to be run in parallel. A basic issue in parallel program design is to develop a language that naturally exhibits all of the parallelism in a given program. Dennis [12] has proposed a language called data flow schemes. A program is essentially a

directed graph where nodes represent operations and arcs represent the fact that one operation may not be executed until the result of another operation is available. In this manner, a considerable amount of parallelism is expressed.

A subclass of data flow schemes, called well formed data flow schemes, is equivalent to if-then-while schemes [13]. These are proposed as constructs that are reasonable to program with. A natural extension of this would be to develop a data flow langauge that has more power than if-then-while schemes. It is shown in Chapter 4 that the full class of data flow schemes has as much expressive power as the class of r.e. program schemes. Thus, in particular, recursion is expressible in the language of data flow schemes. To prove this, it is shown that a very restricted version of data flow schemes (without explicit counters) can simulate Turing Machines.

To recapitulate, the three main areas of study are synchronization of parallel processes, scheduling of tasks on processors, and development of expressive parallel languages. In addition, a number of auxiliary issues arise due to the techniques that are used. The synchronization routines involve the following wide variety of issues: fault tolerant techniques, methods to permit parallel processes to cooperate without requiring fast processes to wait for slow ones, parallel program verification, and simulation of a managed system with a decentralized system.

The scheduling portion develops algorithms for scheduling jobs in heterogeneous systems. The results here have an impact on machine design as they indicate situations where the usage of a processor is rarely warranted. In addition, Section 3.2.4 discusses some limitations on scheduling with limited information, such as, without advance knowledge about the actual time

requirements of jobs. This is an important practical consideration since one often does not have advance knowledge about the actual time requirements of jobs.

Finally, in the scheme portion a new simple basis for computability is developed. Moreover, the techniques used to simulate r.e. program schemes (in particular the finite translation lemma of Section 4.3.2) quantitatively describe how data flow schemes differ from well formed data flow schemes. This difference suggests that perhaps certain additional constructs should be added to the constructs of well formed data flow schemes to achieve certain programming tasks.

2. Synchronization

2.1. Introduction

2.1.1. Background and motivation

2.1.1.1. Previous Work

The mutual exclusion problem is the problem of arranging for asynchronous parallel processes to take turns using a resource. Each process has a "critical section" during whose execution the resource is used. The critical section may be executed correctly only if no other process is simultaneously in its critical section. The first solution to the mutual exclusion problem, which uses read and write instructions as primitives, was devised by Dijkstra [15].

Often one desires protocols that satisfy additional properties, e.g. "fairness" properties such as "lockout free" and "linear wait". (A "lockout free" protocol ensures that every process that tries to enter its critical section will do so, sooner or later.) In [11,16,35], fairness properties of synchronization protocols are considered.

Recent work provides "robust" solutions to the mutual exclusion problem - solutions that work even if parts of the system are unreliable. For example, a process might "die" and set some special variable to a value "dead" [34,37,52,53].

While early solutions to the mutual exclusion problem use only reads and writes, recent work uses the more powerful test-and-set [4,5,17] (as defined in Section 2.1.1.2). Test-and-set instructions have been used either to obtain memory efficient solutions or to cope with unreliability in the processes. In [5], a test-and-set which operates on a single many-valued variable is used to synchronize reliable processes (i.e., the variable may take

on any of many possible values). Developing such synchronization routines is trivial; the contribution of [5] is in obtaining tight upper and lower bounds on the number of values that the single variable must take on.

Similarly, Burns [4] obtains upper and lower bounds on the number of *binary*-valued variables needed. Again, obtaining some synchronization protocol is easy, but *efficient* solutions are hard to find.

Undetectable process death (i.e. no special variable is set at death) is introduced in [17]. In [17], processes which may die are synchronized with a single many-valued variable. Once again difficulties arise only when one tries to obtain tight bounds.

The next generalization of the results of [4,17] is to develop resource allocation algorithms which work in spite of undetectable process death and use only bounded-valued variables. Here, it seems to be difficult to arrive at any solution. We solve this problem as a special case.

2.1.1.2 Focus of the chapter

In this chapter we present a general method for ensuring cooperation among asynchronous processes. This yields, among other things, a new solution to the mutual exclusion problem. We show that even with undetectable death in processes, a large class of synchronization problems is solvable if one uses a test-and-set on bounded-valued variables. In particular, we describe how asynchronous processes can simulate a powerful, "resource manager". Systems with a resource manager easily provide fairness properties (described in Sections 2.2.4 and 2.3). We have not attempted to be efficient; it appeared challenging to develop *any* protocol which simulates a resource manager in an unreliable environment.

This general simulation would not be interesting if we did not assume undetectable death. With totally reliable processes, one process may be chosen as the manager, spending part of its time as a manager and part as a process. Similarly, in a system in which processes announce their deaths, one process may be appointed as the manager, and if that process dies, a different process is chosen to be manager (note the work of [53] which synchronizes such dying processes even without a test-and-set). Here, no process can be chosen to be the manager, since it may die undetectably. Thus *all* of the processes must act as managers. They must coordinate their activities as managers, making sure that they do not interfere with each other's managing efforts. Even this problem would be easy to solve with many-valued variables. With one many-valued variable the manager's entire state may be encoded.

A test-and-set instruction in one atomic step reads a multiple-valued memory cell and updates the cell's value based on the value read. The algorithms in this chapter use test-and-sets on four-valued memory cells. An interesting open problem is to use binary variables - since common machine test-and-set instructions operate on binary variables. (We mention in Section 2.4, a technique which uses three-valued cells.)

2.1.1.3 Memory unreliability

Not only does our simulation work if processes die and bounded-valued variables are used, but it also tolerates limited memory failure. If only a single variable is used and this variable is unreliable then the mutual exclusion problem is not solvable. For every k, we may use many memory cells of fixed size to achieve fair synchronization even if any k cells fail.

Memory unreliability may be viewed as a generalization of the attempts of [4,5,17] to get tight bounds on memory requirements. One reason to use a minimum amount of memory is that a system that uses little memory is reliable if it uses reliable hardware. An alternative approach to providing reliability is to use inexpensive unreliable memory and protocols that tolerate memory errors. Other reasons for introducing memory unreliability are discussed in Section 2.1.2.

2.1.1.4 Outline of the chapter

In Section 2.2 we define models for systems of processes and for manager-directed synchronization protocols. Section 2.2 also defines what it means for one system of processes to *simulate* another system of processes, and shows that this definition preserves important fairness properties. The rest of Chapter 2 shows that a system of processes may simulate an undying manager, even when the simulating system has unreliable memory, processes that die, and no controlling manager. A corollary of this is that fair synchronization in an unreliable environment may be done without a manager.

Section 2.3 presents a sample synchronization protocol where a manager ensures typical fairness properties. Section 2.4 describes our unmanaged simulation of managed systems. Section 2.5 contains a proof of correctness.

2.1.2 Unreliability assumptions and their motivation

2.1.2.1 Process death

In this section we elaborate on our assumption of undetectable death. Real processes often fail without setting a variable to "dead". Thus we assume (as in [17]) that a dead process merely ceases to execute. Our solutions provide fair mutual exclusion as long as at least one process is alive. (Concepts such as fair mutual exclusion require reformulation if processes die undetectably.) We do not consider more extreme modes of failure in which processes deviate from their protocols.

A problem arises when a process dies while executing its critical section, thereby "locking out" all other processes. Our solution will recover from death in the critical section only if processes do announce their death. It is clearly impossible to do better if the system cannot recognize that the process has died.

If a mutual exclusion protocol is tolerant of unannounced death then extremely slow or dead processes cannot cause arbitrary system slowdowns. For example, assume that process j will be next to obtain access to a critical resource. In a previous solution [53], process j waits for signals from other processes before obtaining access to the resource. Thus, a slow process could unnecessarily hold up process j. In any solution tolerant of dying processes, (e.g. our solution) a process must correctly obtain access to the critical resource even if all other processes die. It therefore, cannot wait for acknowledgements from potentially slow or dead processes. This contrasts with solutions which require that other processes will eventually exhibit some activity.

Even if processes die in their critical section, a solution assuming

unannounced death can achieve the k-sharing property [17]. Namely, if there are enough resources for k processes to execute their critical section simultaneously, the deaths of fewer than k processes (even in their critical section) should not deadlock the rest of the system.

Finally unannounced death forces the simulation to be truly decentralized; we can not merely appoint one of the processes as a manager.

2.1.2.2 Memory faults

This section describes the extent to which our simulation methods tolerate memory unreliability. A memory cell is unreliable if after it is set to a specified value, the value may change even if the cell is not rewritten. If all of the memory is unreliable it is impossible for totally reliable coordination to take place. Our solutions are fault tolerant in the sense that they do not depend on the reliability of any single memory cell. Rather, they depend on assumptions about memory failure patterns. Memory is divided into blocks, and the assumptions are of the form "at most e cells in every block of s cells are ever in error." The precise dependence of s on e is described in Section 2.6. For any value of e, there is a value $s=f(e)$ such that the solution works as long as only e cells in each block of s cells fail. To tolerate the failure of any e cells, we make every block of cells tolerant of e failures.

Certain fault tolerant techniques are not applicable. For example, one might utilize primitives that test-and-set three cells at once. If at most one cell is faulty, the three cells still reliably represent a single reliable cell with majority voting. We outlaw this facility on the grounds that three cells that may be accessed at once presumably reside in the same memory "unit". We

desire that our solution tolerate the failure of any such "unit". This also prevents the use of more elaborate coding schemes.

In another fault tolerant technique each process maintains multiple copies of every cell (and each copy is updated at a different step). If at most one cell out of three is unreliable, the true value of a cell could be determined by keeping three copies of every cell and taking the majority. While this technique usually works if each variable is written by only one process, complications arise if different processes may write a variable in a potentially conflicting manner. Since we need to have more than one process write some variables, this multiple copy technique is not immediately usable.

In summary we assume that no single system unit, even a memory unit, is immune to failure. The correctness of the solutions depend on assumptions regarding the number of memory faults. Our protocols handle resource assignment unless all processes die, or as many die in their critical section as there are critical resources.

2.1.2.3 Motivating machine models

This section discusses the interpretation of our model in real systems. In a multiprogrammed environment with several processes running on a single processor, hardware failure would result in the death of all processes. Thus death in our model is best thought of as modeling the effect of an undetected infinite loop. In addition, death models a low priority process which rarely gets processing time. Such slow behavior must not become a system bottleneck.

Our model is also motivated by distributed environments. Here there are usually no global variables, but it is conceivable that some shared memory

might be distributed among the sites of a system. However, even in that case, our solution cannot handle "site death" because site death causes too much memory to be lost. Specifically, if the communication links to a site are unreliable, then all cells at that site are unreliable. If there are n sites in the system and one site dies (through unreliable communication links), then $1/n$ of the cells of the system are not reliably accessible. Our solution does not solve this problem, but does suggest some means of attacking it. As mentioned above, our solutions work if "at most e of s cells are unreliable". If such blocks of s cells are partitioned into s/e subblocks of size e, then the failure of any one subblock is tolerable. For the protocols of this chapter $s/e \geq 16n$. (Section 2.6 contains a variation with a factor of $14n$.) If s/e can be kept as small as n, then the death of one network site (containing one subblock) is tolerable.

2.2. Basic Definitions

Section 2.2.1 defines a *process system*, and several notions of simulation. Section 2.2.2 explores the properties of simulation and describes how simulation preserves fairness. Section 2.2.3 explains why and how the definitions of simulation must be relaxed to allow one to cope with unreliable memory. Section 2.2.4 introduces a special class of process systems, the *managed systems* which model resource allocation systems.

2.2.1 Process systems and the definition of simulation

2.2.1.1 Process systems

A *process system* captures the notion of a set of asynchronously executing processes each of which represents a program being executed in a computer system. Memory cells are used for message passing between processes. Any memory that is local to a process is incorporated into the process's state. The order in which processes appear in a *turns history* reflects the order in which they take turns executing instructions.

A *process system*, P, consists of a finite set of *processes*, denoted *processes*(P), a finite set of *memory cells*, denoted *cells*(P), and a *state transition function* δ. *Process* j (for $1 \leq j \leq |processes(P)|$) is a possibly infinite set, denoted PROC(P,j), of *process-states*. The j^{th} *memory cell* (for $1 \leq j \leq |cells(P)|$) is a finite set, denoted CELL(P,j), of *memory-states*. A designated element of PROC(P,j) is the *initial process-state* of process j and a designated element of CELL(P,j) is the *initial memory-state* of cell j.

The set of *system-process-states* of P, PROCSTATES(P), is the cartesian product PROC(P,1) × PROC(P,2) × . . . × PROC(P,n) where n=|*processes*(P)|. The set of *system-memory-states* of P, MEMSTATES(P), is the cartesian product

CELL(P,1) × CELL(P,2) × . . . × CELL(P,m) where m=|*cells*(P)|. A *system-state*, s, consists of a system-process-state, *procstate*(s), and a system-memory-state, *memstate*(s). The *state of process* j in a system-state s is the j^{th} coordinate of *procstate*(s). The *state of cell* j in s is the j^{th} coordinate of *memstate*(s). The set of all system-states is denoted SYSSTATES(P). The *initial system-process-state* is the system-process-state whose j^{th} coordinate is the initial state of process j. The *initial system-memory-state* and *initial system-state* are similarly defined.

The *state transition function*, δ, is a map

δ:SYSSTATES(P) × {1,...,n}→SYSSTATES(P) where n=|*processes*(P)|.

The function δ satisfies:

(a) For $k \neq j$, the state of process k in $\delta(s,j)$ is the same as that in s.

(b) The state of process j in $\delta(s,j)$ is not equal to that in s.

(c) If the states of process j in s_1 and s_2 are the same, and if *memstate*(s_1)=*memstate*(s_2), then the state of process j in $\delta(s_1,j)$ equals the state of process j in $\delta(s_2,j)$ and *memstate*($\delta(s_1,j)$)=*memstate*($\delta(s_2,j)$).

If $\delta(s_1,j)=s_2$ then s_2 *follows* s_1 *after a process* j *transition.*

Condition (a) asserts that at a process j transition, the state of no other process changes. Condition (b) insists that at a process j transition, the state of process j does change, a technical convenience to be explained later. Condition (c) indicates that the changes in the system-state at a process j transition depend only on the system-memory-state and the state of process j. Note that the processes are deterministic.

A *turns history* for P is an infinite sequence of integers from the set

$\{1,\ldots,|processes(P)|\}$. The *system-state-history* of a turns history $t_0t_1\cdots$ is the unique sequence of system-states $s_0s_1\cdots$ such that s_0 is the initial system-state and $\delta(s_i,t_i)=s_{i+1}$. Process j *dies* in a turns history T if j appears only finitely often in T.

2.2.1.2 One definition of simulation

We now discuss one definition of simulation of a process system P by a process system Q. This is a weak definition, involving only the memory.

Let $1\leq d\leq|cells(P)|$. Let $s_0s_1\cdots$ be a system-state-history for P and let m_i denote the d-tuple of the states of the first d cells in s_i. The *memory-history of the first d cells of* $s_0s_1\cdots$ is the sequence $m_0m_1\cdots$. We often refer to the memory-history of the first d cells as the "memory-history". One only looks at the first d cells in P, since the others may not be important; they may be used for bookkeeping and not for communication between processes. If all cells of P are important, then the memory history is only of interest for $d=|cells(P)|$.

The important part of such histories are the changes. Let $h_0h_1\cdots$ be an infinite sequence (a "history"). The i^{th} element of $h_0h_1\cdots$ is *altered* if $i=0$ or if $h_i\neq h_{i-1}$. The *altered-history* is the subsequence consisting of altered sequence elements. For example, the *altered-memory-history* is the altered history of the memory history.

We define the notion of Q simulating P. Let $1\leq d\leq|cells(P)|$. A *memory-simulator-function*, f, for Q into the first d cells of P, consists of d functions $f_1,\ldots f_d$, where f_j:MEMSTATES(Q)→CELL(P,j). The function f_j is called the *memory-simulator-function* for cell j of P. If $m\in$MEMSTATES(Q), then $f(m)=(f_1(m),\ldots f_d(m))$ is called the *simulated-memory-state* of m. For any

system-state-history, $s_0 s_1 \cdots$, of Q, the *simulated-memory-history* is the sequence $f(memstate(s_0))f(memstate(s_1))\cdots$. The simulated-memory-state and simulated-memory-history depend on d and f, but this dependence is often suppressed when obvious. (The simulated-memory-state is not a system-memory-state of P unless $d=|cells(P)|$.)

Q *simulates the memory* of P with respect to a memory-simulator-function, $f=f_1,\ldots,f_d$ (i.e. with respect to the first d cells) if the set of altered-memory-histories of P obtained from all turns histories for P and the set of altered-simulated-memory-histories of Q obtained from all turns histories for Q are the same.

One considers altered histories so that Q may spend many turns simulating one turn of P. Also, if P spends k turns without changing the first d cells, Q need not waste k turns.

2.2.1.3. Faithful simulation

The above notion of simulation does not require any correspondence between the processes of Q and P. When modelling entrance into the critical section, we need a correspondence between the processes of the two systems, e.g., the j^{th} process of P and the j^{th} process of Q.

Let P and Q be two process systems and let $1\leq r\leq \min(|processes(P)|,|processes(Q)|)$. A *process-simulator-function*, g, is a set of r functions $g_1,\ldots,g_r$ where g_j is a function g_j:PROC(Q,j)→PROC(P,j), the *process-simulator-function* for process j. The value $g_j(q)$ (for $q\in$PROC(Q,j)) is the *simulated-process-state* of q.

The *essential-state* of a system-state s with respect to r and d is the $r+d$-tuple listing the states of the first r processes in s, and the states of

the first d cells in s. The *essential-history-with-repetitions* of a system-state-history $s_0 s_1 \cdots$ is the sequence $e_0 e_1 \cdots$ where e_i is the essential-state of s_i. The *essential-history* is the subsequence of the essential-history-with-repetitions consisting of altered sequence elements.

One looks only at the first r processes since it may not be important to simulate the behavior of all of the processes of P. For example, in our simulation of a managed system, it is not important for any process of the simulating system to have its state correspond to the state of the manager.

Let Q and P be two process systems. Let $s \epsilon$SYSSTATES(Q), let $g=g_1,\ldots,g_r$ be a process-simulator-function, and let $f=f_1,\ldots f_d$ be a memory-simulator-function. The *essential-simulated-state* of s is the $r+d$-tuple listing the simulated-process-states of the first r processes and the simulated-memory-states of the first d cells. The *essential-simulated-history-with-repetitions* of a system-state-history for Q, $s_0 s_1 \cdots$, is $e_0 e_1 \cdots$, where e_i is the essential-simulated-state of s_i. The *essential-simulated-history* is the subsequence of the essential-simulated-history-with-repetitions consisting of altered sequence elements.

Let $\mathcal{T}$ be a set of turns histories of P and let $\mathcal{T}'$ be a set of turns histories of Q. Then Q *faithfully simulates* P *with respect to* $\mathcal{T}$ *and* $\mathcal{T}'$, with a memory-simulator-function $f=f_1,\ldots f_d$, and a process-simulator-function $g=g_1,\ldots,g_r$, if:

(1) The set of essential-histories of P obtained from turns histories in $\mathcal{T}$ and the set of essential-simulated-histories of Q obtained from turns histories in $\mathcal{T}'$ are the same.

(2) Let $T \epsilon \mathcal{T}'$, and let $q_0 q_1 \cdots$ be the sequence of process j states in the

system-state-history generated by T ($j \leq r$). If process j does not die in T then infinitely many elements of the sequence of simulated-process-states $g_j(q_0)g_j(q_1)\cdots$ are altered.

Thus, (2) prevents process j from having infinitely many turns in Q without simulating process j in P. The motivation is that if, in some turns history a process of Q does not die, then the corresponding simulated process of P does not die either. By condition (a) of the definition of the state transition function, the only turns in a turns history for Q at which the simulated-process-state of process j may change are at process j turns.

One reason that process j changes state at a process j transition (condition (b) of the definition of the state transition function), is to permit us to state condition (2). If a process in P could take infinitely many turns without changing state, then it would not make sense to insist that in Q, the corresponding process could not take infinitely many turns without changing its simulated state.

The second reason that process j changes its state is related to (1). If in some history of P process j has a number of turns in which it remains in the same state, there should be some method of noticing this in histories of Q. However, since (as explained in Section 2.2.1.2) it is convenient to define simulation on altered histories, the fact that process j does not change its state is not represented in any essential-history. Thus we force the state to change.

Q faithfully simulates P if it faithfully simulates P with respect to the set of all turns histories of P and the set of all turns histories of Q.

Q faithfully partially simulates P (with respect to $\mathcal{T}$ and $\mathcal{T}'$) if the set of essential-histories of P (obtained from turns histories in $\mathcal{T}$) contains the set of essential-simulated-histories of Q (obtained from turns histories in $\mathcal{T}'$) and (2) above holds (for $\mathcal{T}'$).

Intuitively, if Q faithfully partially simulates P, then no history of Q behaves incorrectly, since every history reflects a possible behavior of P. However, Q may not reflect all possible behaviors of P.

The usefulness of a faithful partial simulation depends on which features of P one wants to preserve with Q. For example, if P satisfies properties such as deadlock freedom or bounded waiting (as defined in Section 2.2.2), then so will Q, if Q faithfully partially simulates P. Since these fairness properties motivate this work, faithful partial simulation is important here.

2.2.2. Properties of simulation.

For this section suppose Q faithfully simulates P with respect to a memory-simulator-function $f=f_1,...,f_d$, and a process-simulator-function $g=g_1,...,g_r$. We discuss important properties of simulation which are trivial consequences of the definition.

Lemma 2.1. Q simulates the memory of P.

Proof. Obvious. □

Lemma 2.2 Let T be a turns history of P, T' a turns history of Q, such that the essential-history of T equals the essential-simulated-history of T'. Let $j \leq r$. Then process j dies in T iff process j dies in T'. If j does not occur in T', then j does not occur in T.

Proof. If process j dies in T, then there are only finitely many alterations in the "process j history" (i.e. the sequence of process j states), by condition (a) of the definition of the state transition function. Thus, the simulated-process-state of process j may change only finitely often in T'. By condition (2) on faithful simulation, if process j does not die in T', the simulated-process-state of process j is altered infinitely often. Thus process j dies in T'.

Conversely, assume that process j dies in T'. Since at most one change in simulated-process-state occurs for each occurrence of j in T', the simulated-process-state of process j changes only finitely often. Thus there are only finitely many changes in the state of process j in the

essential-history of T and so by condition (b) on state transition functions j occurs finitely often in T.

If j does not occur in T', then the simulated-process-state of process j never changes and j does not occur in T. □

Assume that each process of P has one designated *critical state*. Then P satisfies *mutual exclusion* for a set of turns histories, $\mathcal{T}$, if no two processes (i.e. of processes 1,...,r) are ever simultaneously in their designated critical states in any system-state-history that arises from $T \in \mathcal{T}$. We say that the simulating system, Q, satisfies mutual exclusion for the turns histories, $\mathcal{T}'$, if in no system-state which is an element of the system-state-history of a $T' \in \mathcal{T}'$, are the simulated-process-states of two processes (of processes 1,...,r) simultaneously critical. We think of a process in Q as executing its "critical section" in one of its states whose simulated-process-state is the critical state of P.

Lemma 2.3. If P satisfies mutual exclusion then Q satisfies mutual exclusion.

Proof. Assume that Q does not satisfy mutual exclusion. Then in some system-state-history of Q generated by some turns history T' there is a system-state in which the simulated-process-states of two different processes are both critical. Look at the first turn at which this happens. The essential-simulated-state of Q is altered at that turn, and thus appears in the essential-simulated-history of T'. Since the essential-simulated-history of T' equals the essential-history of some turns history T for P, it must be that P violates mutual exclusion. □

Similarly, P is *deadlock free* for $\mathcal{T}$ if in every turns history of $\mathcal{T}$ in which none of processes $1,\ldots,r$ dies, some process enters its critical state infinitely often, and P is *lockout free* for $\mathcal{T}$ if in every turns history of $\mathcal{T}$ in which none of processes $1,\ldots,r$ dies, all r processes enter their critical states infinitely often. With a similar definition of Q being deadlock free or lockout free, it follows that if P is deadlock free or lockout free, then Q is deadlock free or lockout free.

Assume that each process of P has a "trying" section; a set of "trying states" that it enters before entering its critical state. (Before entering the critical state, a process enters some trying state, and remains in the trying section until it enters its critical state.) Then P satisfies *bounded waiting* for $\mathcal{T}$ if for some k, there is no turns history of $\mathcal{T}$ in which none of processes $1,\ldots,r$ dies, in which one process enters its critical state more than k times while another process is in the trying section. With a similar definition of Q satisfying bounded waiting, if P satisfies bounded waiting then Q satisfies bounded waiting.

The proof of Lemma 2.3, and the fact that simulation preserves deadlock freedom, lockout freedom, and bounded waiting depend on every essential-simulated-history of Q being equal to an essential-history of P. Thus, all these properties are preserved if Q faithfully partially simulates P.

The property of being deadlock free only concerns turns histories in which no process dies. More generally, one may develop the notion of being "enabled" to enter the critical section [17]. Intuitively, this occurs when process j may enter its critical section after finitely many process j turns, irrespective of the other processes. A deadlock free system may then be

defined to be one that does not deadlock unless some process is dead in its critical section or is dead while enabled to enter its critical section. The exact definition to be used is not our objective, and we omit further discussion.

2.2.3. Process systems with unreliable memory

We discuss simulation of a system P with reliable memory by a system Q with unreliable memory. Due to the faulty memory, the definitions must become more complicated. In contrast to the definitions of Section 2.2.1 which we believe to be intuitively reasonable, the revised definitions are not clearly the best possible. However, they are useful in that they preserve fairness.

2.2.3.1. Error transitions

A *process system with errors*, Q, consists of sets of processes and cells (as with ordinary process systems), a set of *errors*, denoted *errors*(Q), and a state transition function δ. The set *errors*(Q) is a subset of $\{(j,k): 1 \leq j \leq |cells(Q)|)$ and $k \in CELL(Q,j)\}$. If $(j,k) \in errors(Q)$ for some $k \in CELL(Q,j)$ then cell j is called *faulty*. We will assume that if cell j is faulty, then for every $k \in CELL(Q,j)$, $(j,k) \in errors(Q)$.

The state transition function, δ, is a map δ:SYSSTATES(Q) $\times$ ($\{1,...,|processes(Q)|\} \cup errors(Q)$) $\rightarrow$ SYSSTATES(Q). The next state after a process j transition is the same as in ordinary process systems. If $e=(j,k) \in errors(Q)$, then $\delta(s_1,e)$ is defined to be the state s_2 where $procstate(s_2)=procstate(s_1)$, the memory-states of s_1 and s_2 are the same for all cells other than j, and the state of the j^{th} cell in s_2 is k. The state s_2 is called the state that *follows* s_1 *after the error transition* e.

A *turns history* for Q is any infinite sequence from the set $\{1,...,|processes(Q)| \cup errors(Q)$. Other concepts such as system-state-histories and memory-histories are defined similarly to the definitions of Section 2.2.1.

We will use *errors*(Q) as a parameter, i.e., we may consider a particular system Q with different sets of faulty cells.

2.2.3.2. Memory simulation

We illustrate the need to change the definition of memory simulation. Recall that our assumptions about memory unreliablity are of the form "at most e of s cells are faulty". Assume that we wanted to design a system Q with one out of five cells faulty and a single binary-valued cell of P is to be represented in Q as five cells. Assume that the memory-simulator-function is defined based on majority voting (i.e., the simulated memory-state equals 0 iff the majority of the five cells equal 0). Then when these cells are updated there is a potential oscillation in the simulated-memory-state. Specifically, assume that at some time the memory-state of exactly three of the five cells equal zero. If one of the cells containing zero is faulty and oscillates between 0 and 1, the simulated-memory-state will similarly oscillate. If P does not exhibit such oscillatory behavior, this unpreventable behavior in Q would prevent Q from simulating P. To handle such transients in the system we allow some flexibility in deciding when the simulated-memory-state changes. There is a transitional period, during which we say that the simulated-memory-state is in the process of being changed. We would say (above) that the memory-state is mapped by the memory-simulator-function to 0 if the state of zero, one, or two cells equals 1, to 1 if the state of four or five cells equals 1, and to *transitional* if the state of three cells equals 1.

For a *memory-simulator-function* f from the simulating system with errors, Q, into the first d cells of P, the component functions f_j are maps:

$$f_j: \text{MEMSTATES}(Q) \rightarrow \text{CELL}(P,j) \cup \{\textit{transitional}\} \text{ for } j=1,\ldots,d.$$

Assume $f_j(m) \neq transitional$ if m is the initial system-memory-state of Q.

The simulated-memory-state of a given system-memory-state of Q is defined with respect to a given history. Specfically, let $s_0 s_1 \cdots$ be a system-state-history for Q. If $f_j(memstate(s_i)) \in CELL(P,j)$, then the *simulated-memory-state* of cell j at step i is $f_j(memstate(s_i))$. If $f_j(memstate(s_i)) = transitional$, then the simulated-memory-state of cell j at step i is the same as the simulated-memory-state of cell j at step i-1.

Intuitively, if $f_j(m) = transitional$, then whenever the memory-state of Q equals m the system Q is in the middle of changing the simulated-memory-state of the j^{th} cell. In a system with errors, one develops simulation techniques that use the transitional memory-states to prevent the simulated-memory-states from oscillating.

The *simulated-memory-history* is defined as in a system without errors, i.e., the sequence of simulated-memory-states.

2.2.3.3. Process simulation

We discuss process simulation in a system with errors, starting with an example. Assume (as in Section 2.2.3.2) that one cell of P is represented as five cells in Q and when the cell in P is to be updated by a process from 0 to 1 all five cells in Q are updated by the corresponding process from 0 to 1. In P, the state of the cell changes at the same turn at which the state of the process changes. Thus if Q is to faithfully simulate P, the simulated-memory-state must change at the same turn at which the simulated-process-state of the process changes. Since the simulated-memory-state of Q changes at the first turn in which four cells equal 1, the simulated-process-state must change at the same turn. Unfortunately,

depending on whether there have been any error transitions, the process may be in any of a number of states and thus the simulated-process-state cannot depend only on the process-state. Thus the simulated-process-state of a process changes based partially on the simulated-memory-state. Intuitively, if a process of Q is trying to simulate a turn of a process of P, it completes this simulation when the simulated-memory-state of Q changes.

Let P and Q be two process systems as above. A process-simulator-function for process j, g_j, is a map g_j:PROC(Q,j)→PROC(P,j) ∪ {*transitional*}.

Since the simulated-process-state changes based on the simulated-memory-state, we need to relate the memory-states to the process-states.

A *process-correspondence* c, consists of r component functions $c_1,\ldots,c_r$. The function c_j is a pair of functions: the function g_j, and a *partition function* which partitions MEMSTATES(P) as follows. For each ordered pair of states (q_1,q_2) where $q_1,q_2 \in$PROC(Q,j), and neither $g_j(q_1)$ nor $g_j(q_2)$ are equal to *transitional*, MEMSTATES(P) is partitioned into two sets. These two sets are called MEMSTATES$_1(q_1,q_2)$ and MEMSTATES$_2(q_1,q_2)$. Intuitively, MEMSTATES$_1(q_1,q_2)$ consists of those memory-states that would occur in P while process j is still in $g_j(q_1)$, and MEMSTATES$_2(q_1,q_2)$ are those that occur by virtue of the process j turn into $g_j(q_2)$. The simulated-process-state changes as soon as the simulated-memory-state is in MEMSTATES$_2(q_1,q_2)$.

The following formalizes the notion that the simulated-process-state of a process depends on its last non-transitional state, the next non-transitional state it will enter, and the memory (as given by the process-correspondence). Let c_j as above be a process-correspondence for process j, let f be a memory-simulator-function, let $s_0s_1\cdots$ be a system-state-history of Q, and let

$q_0 q_1 \cdots$ be the process j history of $s_0 s_1 \cdots$. If $g_j(q_i) \in \text{PROC}(P,j)$, then the *simulated-process-state* at step i is $g_j(q_i)$. If $g_j(q_i)$=*transitional*, then let k, l be integers such that neither $g_j(q_k)$ nor $g_j(q_l)$ are *transitional* ($k<i<l$), and for $k<h<l$, $g_j(q_h)$=*transitional*. Let h be the smallest integer, if any, $k<h\leq l$ such that $f(\textit{memstate}(s_h)) \in \text{MEMSTATES}_2(q_k,q_l)$. Then if $k<i<h$ the simulated-process-state at step i is $g_j(q_k)$, and if $h\leq i<l$ the simulated-process-state at step i is $g_j(q_l)$. If no h satisfies $f(\textit{memstate}(s_h)) \in \text{MEMSTATES}_2(q_k,q_l)$ (for $k<h\leq l$) then the simulated-process-state at step i is $g_j(q_k)$ for all $k<i<l$.

To complete the definition of simulated-process-state, we discuss one additional case of relatively small importance. It is possible that $g_j(q_k) \neq$*transitional* and for all $l>k$ $g_j(q_l)$=*transitional*. By condition (2′) on faithful simulation (below), this will only happen in turns histories $t_0 t_1 \cdots$ in which process j dies. In that case, the simulated-process-state changes if it is clear which simulated-process-state process j was trying to enter. Specifically, assume $g_j(q_k) \neq$*transitional*, but for $l>k$ $g_j(q_l)$=*transitional*. Let l be the smallest integer such that for $i\geq l>k$ $q_i=q$, i.e., q is the last state that process j enters. Assume that there is a $q' \in \text{PROC}(Q,j)$ such that for all turns histories that start $t_0 t_1 \cdots t_l$ in which process j does not die, q' is the first state (after step l) that process j enters that is not mapped to *transitional*. (Intuitively, process j is "on the way" to q'.) Let h be the smallest integer (if any) greater than k such that $f(\textit{memstate}(s_h)) \in \text{MEMSTATES}_2(q_k,q')$. Then if $k<i<h$, the simulated-process-state at step i is $g_j(q_k)$. If $i\geq h$, the simulated-process-state at step i is $g_j(q')$. If there is no such q', or if there is a q' but no such h then the

simulated-process-state at step i ($i>k$) is $g_j(q_k)$.

Note that the simulated-process-state of process j need not change at a process j turn.

All the concepts that depend on the simulated-process-state and simulated-memory-state, such as the essential-simulated-state are defined in a manner similar to Section 2.2.1. The main difference is that the simulated-process-state and simulated-memory-state are only defined in the context of a history, and thus, the essential-simulated-state is only defined in the context of a history.

The notion of faithful simulation is similar to that of Section 2.2.1, with the following modified version of condition (2).

(2′) Let T be a turns history for Q and let $q_0q_1\cdots$ be the sequence of process j states in the system-state-history generated by T. Consider the subsequence, S, of $g_j(q_0)g_j(q_1)\cdots$ consisting of nontransitional elements. If j does not die in T, then the sequence S is infinite and infinitely many elements of S are altered.

By (2′), if process j does not die in Q, it enters infinitely many states which are not mapped to transitional. Furthermore, this sequence of infinitely many states is altered infinitely often and thus the process is acting like process j in P.

We discuss what it means for a simulating system with errors, Q to satisfy mutual exclusion. The way we have modelled the "critical section" of a process in Q is that it would be executed while the process is in one of the

states of Q whose simulated state is the critical state of the process in P. Since the simulated-process-state of a process in Q (with errors) depends not only on the process's state, we must say more about when the process in Q may execute its critical section.

One way to model the critical state of process j in P is to consider process j in Q to be in the critical state only if the state of process j is $q \in \text{PROC}(Q,j)$ where $g_j(q)$ equals the critical state of P. Q satisfies mutual exclusion for $\mathcal{T}'$ if in no system-state which is an element of a system-state-history of $T \in \mathcal{T}'$, are two processes in states that are mapped by their respective process-simulator-functions to a critical state. This definition of Q satisfying mutual exclusion is preserved by the definition of simulation.

2.2.4 A Managed System of Processes and the main results of the chapter

2.2.4.1 Definition of managed systems

This section defines a class of process systems, the *managed systems*. One process is designated as a manager, and intuitively, it informs processes that they may use resources by sending them messages. Communication between a process and the manager is done through a cell associated with the process. The process and manager alternate in the use of the cell.

A *Managed System of n processes* is a system of processes, P, with $|processes(P)|=n+1$ and $|cells(P)|=n$. Process $n+1$ is called the *resource manager* (RM). The state set of the RM is finite and is partitioned into n subsets $F_1,...,F_n$. Intuitively, if the state of the RM is in F_j then at its next turn it will look for a message from process j. The elements of CELL(P,j) ($1\leq j\leq n$) are pairs (L_j,M_j), where L_j may be either "REQUEST" or "RESPONSE" and M_j may be one of a set of integer values. The initial state of CELL(P,j) is (RESPONSE,0).

Intuitively, if L_j is REQUEST, then M_j is a request from process j to the RM. When L_j is RESPONSE, M_j is a response by the RM to the last process j request.

The state transition function δ, has the following properties. If $\delta(s_1,n+1)=s_2$, and the state of the RM in s_1 is in F_j then s_1 and s_2 must differ in the state of the RM and may differ in the state of the j^{th} cell. Furthermore, these two coordinates in s_2 depend only on these two coordinates in s_1 as follows. If L_j=REQUEST, then they depend both on the state of the RM and on the value of M_j in s_1. If L_j=RESPONSE, then the state of the RM in s_2 depends on the state of the RM in s_1 and the state of the j^{th} cell is unchanged.

For $j<n+1$, if $\delta(s_1,j)=s_2$ then s_1 and s_2 must differ in the state of process j and may differ in the state of the j^{th} cell. If L_j=REQUEST, then only the state of process j changes in s_2, and it depends on the state of process j in s_1. If L_j=RESPONSE, then the state of process j, and the value of M_j in s_2 both depend on the state of process j and the value of M_j in s_1. The value of L_j in s_2 is REQUEST.

2.2.4.2 Statement of the main result of Chapter 2

Given a managed system P, we define a system of $2n$ processes Q such that Q faithfully simulates P with respect to all of the memory cells and processes $1,...,n$. The last n processes are finite state processes, used to simulate the actions of the *RM*. Q may be thought of as an n process system where the n processes have a small amount of nondeterminism - they may choose to act either as a process or as a simulator of the *RM*.

Let $\mathcal{T}$ be the set of turns histories of P in which the *RM* does not die and in which at least one of the first n processes does not die. We only consider these histories as we think of *RM* systems operating only when the *RM* does not die, and at least one process does not die. In Q, a history is only of interest if at least one of the n "true" processes does not die (the j^{th} true process is thought of as both the j^{th} process and the j^{th} simulator). For a system of $2n$ processes, Q, define $\mathcal{T}'$ to be the set of turns histories in which at least one of processes $1,...,n$ does not die and for $j=1,...,n$, process j dies iff process $n+j$ dies. In every turns history in $\mathcal{T}'$ some process has infinitely many turns. Also, in $\mathcal{T}'$, there is a rough identification of the j^{th} and $n+j^{th}$ processes, namely, that if the j^{th} true process is not dead

it acts both as a process and as a simulator.

The main result is:

Theorem 2.1. For any managed system P and all $k>1$, there is a system of $2n$ processes Q that faithfully simulates P with respect to $\mathcal{T}$ and $\mathcal{T}'$ even if any k of the cells of Q are faulty.

From Theorem 2.1 we see that a simulating system Q can be built with k faulty cells for any k. Sections 2.4 and 2.5 prove Theorem 2.1 with $k=1$, and Section 2.6 discusses the generalization for arbitrary k.

2.2.4.3. Discussion of Theorem 2.1

It seems unlikely that one would be able to find a simulating system with n deterministic processes. If the processes are deterministic then the number of turns that these processes use to simulate the *RM* from a given system-state will be bounded. Such a system cannot faithfully simulate the *RM* system since in the *RM* system, the *RM* may have many consecutive turns, more than the number used by the processes to simulate the *RM*. Thus, the particular essential-history of the managed system would not equal any essential-simulated-history of any simulating system.

One might try to solve this problem by defining all processes to be nondeterministic and using a different definition of simulation. This, however, would obscure the issue that prevents n processes from simulating *RM* systems. In any real system, once it is determined how often each process acts as a simulator the n processes cannot simulate the *RM* system.

Using the process system Q, we devise a system of n processes Q' that

faithfully partially simulates P. Then Q' satisfies fairness properties such as lockout free and bounded waiting if P does. This indicates how n deterministic processes may accomplish fair mutual exclusion in the face of unreliability.

In Section 2.5 we explain how to simplify Q to obtain a system Q'' which faithfully simulates P with respect to $\mathcal{T}$ and $\mathcal{T}'$ where there are no error transitions in the elements of $\mathcal{T}'$. This simulation is correct even with the more restrictive definition of faithful simulation by a system without errors.

Certain properties of Q make it reasonable for use in practice. The next state after a process j transition depends only on the state of process j and the state of one cell. Furthermore, the only memory-state that may change at the transition is the state of that one cell. Finally, $|\mathrm{CELL}(Q,j)|$ is bounded for every j; it is at most four.

2.3. An Example

We illustrate a managed system, P, that provides fair mutual exclusion. PROC(P,j)=$\{q_1,q_2,t_1,t_2,c\}$ for j=1,...,n (the dependence on j in the elements of PROC(P,j) is suppressed). The cells consist only of the L_j portion. At a process j transition from either "quiessent" state (q_1 or q_2), process j enters the other quiessent state if L_j=REQUEST and t_1 if L_j=RESPONSE. If L_j=RESPONSE it is set to REQUEST. At a process j transition from a "trying state" (t_1 or t_2), if L_j=RESPONSE then process j enters the "critical state" (c), if L_j=REQUEST, process j enters the other trying state, and L_j is unaffected. At a process j transition from c, L_j is set to REQUEST and process j enters q_1. The state q_1 is the initial state of process j.

Intuitively, if process j is in a quiessent state, it requests a resource and enters a trying state. From the trying state, it waits for permission to enter the critical state (L_j=RESPONSE). When leaving the critical state, process j sets L_j to REQUEST to signal that it has left the critical state and enters a quiessent state. As soon as process j's release of the critical resource has been acknowledged (L_j=RESPONSE), process j may request a resource and enter a trying state.

Intuitively, the *RM* examines each process to determine whether the process is requesting access to the critical resource, or has left its critical state. If process j is being handled (i.e., the *RM* state is in F_j) and L_j is RESPONSE, then the *RM* proceeds to the next process. If L_j is REQUEST and the *RM*'s state indicates that process j is in the critical state the *RM* interprets the value of L_j to mean that process j has left the critical state. The *RM* acknowledges this by setting L_j to RESPONSE. If L_j=REQUEST and the RM's state indicates that process j is not in the critical state, then the *RM* places

process j on a list of waiting processes. If it is process j's turn to enter its critical state, then L_j is set to RESPONSE.

Formally, the *RM* state set is an n+2-tuple $(j,bit,i_1,...,i_n)$. The first coordinate specifies which process is being handled. The *bit* coordinate is 0 if no process is assigned to be in its critical state and 1 otherwise. The coordinates $i_1,...,i_n$ represent the queue of processes that are either in their critical state or waiting to enter. If the size of the queue is less than n, the last few entries are zero.

The next state function for the *RM* (as described above) is given as follows (j+1 means (j+1) (mod n)):

	Current state	L_j	Next state	L_j
1.	$(j,bit,i_1,...,i_n)$	RESPONSE	$(j+1,bit,i_1,...,i_n)$	RESPONSE
2.	$(j,1,j,i_2,...,i_n)$	REQUEST	$(j+1,0,i_2,...,i_n,0)$	RESPONSE
3.	$(j,bit,i_1,...,i_k,0,...,0)$ where $j \neq i_1,...,i_k$.	REQUEST	$(j+1,bit,i_1,...,i_k,j,0,...,0)$	REQUEST
4.	$(j,bit,i_1,...,i_k,0,...,0)$ where j equals one of $i_1,...,i_k$.	REQUEST	$(j+1,bit,i_1,...,i_k,0,...,0)$	REQUEST
5.	$(j,0,j,i_2,...,i_n)$	REQUEST	$(j+1,1,j,i_2,...,i_n)$	RESPONSE
6.	$(j,0,0,...,0)$	REQUEST	$(j+1,1,j,0,...,0)$	RESPONSE

Line 2 describes how the *RM* handles the relinquishing of resources by a process. Line 3 describes a process being added to the list of waiting processes. Lines 5 and 6 specify that if it is process j's turn to enter the critical section that the *RM* informs process j of that fact.

If the memory is reliable and processes do not die, then mutual

exclusion is satisfied. The *RM* keeps track of the "region" that each process is in, and permits only one to be in its critical state at a time. Due to the FIFO assignment procedure, deadlock freeness, lockout freeness, and bounded waiting are satisfied.

If some of the processes die but the *RM* does not die the fairness properties still apply if one uses the definitions of [17].

While this example is simple, managed systems can handle multiple resource systems and systems where some processes have high priority.

2.4. Cooperating system of n simulators.

2.4.1 Overview of the system

Section 2.4 describes the system Q, that is used to prove Theorem 2.1. For a given managed system P, a *cooperating system of n simulators* is defined; a system, of $2n$ processes, that faithfully simulates P. The first n processes of Q behave like the first n processes of P. The last n processes, called *simulators*, behave like the *RM*. They cooperate to simulate each *RM* turn with many simulator turns. (Each simulator consists of finitely many states.)

While each process has occasion to use many cells, at each process j transition ($1 \leq j \leq 2n$) the next state of process j depends only on its current state and the state of one memory cell. The state of that memory cell may be changed but the states of all other cells remain the same.

The memory cells are used for two purposes. One purpose is for communication between the first n processes and the simulators (similar to the communication between the processes and *RM*). The other purpose is to help the simulators keep track of bookkeeping information such as the current *RM* state.

The behavior of the simulators is quite complicated. In order for the simulation to be tolerant of the death of as many as $n-1$ of the "true" processes each simulator independently attempts to do the entire work of the *RM*.

We proceed with a broad overview of some of the components of Q and motivate their need. Each simulator tries to do a step-by-step simulation of the *RM*, reading messages from processes and sending responses to processes. In order to know which process to handle, the simulator first reads a shared bookkeeping area which includes the current *RM* state. The simulator reads a

message from a process, sends a response, and updates the bookkeeping area.

This naive approach suffers from many inadequacies and we categorize some of them as follows:

(1) Since bounded-sized memory cells are used, a simulator cannot update all cells at once. However, if each cell is updated separately, the bookkeeping area will at times be partially updated and partially old. Other simulators will then not be able to get a consistent picture of the bookkeeping area.

(2) Assume that some method was devised to permit other simulators to decode a "static", partially updated and partially old area. A second problem is that a simulator may slowly read the bookkeeping area while the other simulators are changing that area.

(3) Assume that some method was devised to permit each simulator to get a consistent picture of the bookkeeping area. A third problem is that the simulators, working at different speeds, may end up working on different *RM* turns. A possible result is a "race condition" in the updating of the cells.

(4) Assume that some method permits all simulators to work on the same *RM* turn. A fourth problem is that the simulators must coordinate their activity at the simulation of each *RM* turn. Specifically, each simulator tries to determine if a message is being sent by a process (i.e., whether process J should be handled as if L_J were RESPONSE or REQUEST). Since the determination for each simulator occurs at slightly different times, they may reach different conclusions as to whether L_J should be considered to be RESPONSE or REQUEST and thus they would update the bookkeeping area differently.

We now proceed with a sketch of the methods used to solve the above problems. We postpone the full details until Section 2.4.4.

To solve (1) there are three bookkeeping areas. At each instant only one of the three is considered current by the simulators. An area is current if certain "STATUS" cells equal 1. When the current area needs to be updated the new information is placed in a different area. When the update is completed, the new area becomes "current".

The old area cannot have its "STATUS" set to 0 at the same time that the new area becomes current (for reasons to become clear later) and thus at times two areas seem to be current at once. In that case, the current area is the one which precedes the other area in a circular ordering of the three areas. It is for this convention that three areas are needed.

To solve (2), before a simulator reads an area it sets certain "SIMULATOR-ACCESS" cells to 1 (each simulator has "personal" SIMULATOR-ACCESS cells in each area). After reading an area, the simulator checks that these cells are still 1, to determine if the area was changed (they are set to 0 when the area is changed). If it was changed, the simulator ignores what it read.

To solve (3), before each simulator j turn that is used to simulate the *RM*, there are other simulator j turns. At these turns, simulator j determines which *RM* turn the "fastest" simulator is up to again by using the STATUS and SIMULATOR-ACCESS cells. (The fastest simulator, is the one most advanced in the simulation of *RM* turns.) If simulator j discovers that it has been executing slower than the fastest simulator, the simulator tries to catch up to the fastest simulator. In this manner each slow simulator causes at most one cell to assume an incorrect state while the fastest simulator is working on each *RM* turn. This small amount of incorrect activity is handled by memory

redundancy.

To solve (4) all simulators participate in a protocol which coordinates their perceptions as to whether a message is being sent. The protocol will be described in Section 2.4.4.

In summary, the following basic techniques are used to coordinate the simulators.

(1) Three bookkeeping areas.

(2) Extra turns to ensure that the bookkeeping information which is read is consistent.

(3) Extra turns to ensure that each simulator does not get too much out-of-date.

(4) Memory redundancy for limited out-of-datedness.

(5) A protocol to achieve agreement on whether a message is being sent.

Of these, we have not given all of the details for (2)-(4) or any of the details for (5). The basic idea behind (2)-(4) should be clear although the details are fairly complex.

An outline of how a simulator simulates an *RM* turn is:

(1) Determine which bookkeeping area is current, set the SIMULATOR-ACCESS bits to 1 in that area, and read the bookkeeping information.

(2) Make sure that the bookkeeping information read is consistent (by checking STATUS and SIMULATOR-ACCESS).

In (3)-(6), between every pair of turns executed, make sure that the simulator is still up-to-date.

(3) Determine if the appropriate process is sending a message ("appropriate" depends on the bookkeeping information).

(4) Participate in a protocol with other simulators to determine if a message is being sent.

(5) Read the message from the process (if there is one) and send response.

(6) Update bookkeeping area (with redundant copies of all cells); set the STATUS of the next area to 1, set the STATUS and SIMULATOR-ACCESS (for each simulator) of this area to 0.

Minor details have been suppressed. For example, each process actually uses many message areas and the simulators maintain a MSG-PTR in the bookkeeping areas to point to the currently up-to-date message area. This and other details will be explained in the remainder of Section 2.4. Section 2.4.2 describes the cells that are used in our simulation and Section 2.4.3 describes the state sets of the first n processes. The techniques that the simulators use to keep track of the *RM* state, to remember which bookkeeping area is current, and to coordinate the determination of whether messages are being sent, are detailed in Section 2.4.4.

2.4.2. Memory of the cooperating system

2.4.2.1. Bookkeeping cells

This section gives the structure of $Q's$ memory. We first describe the bookkeeping areas. They are three equally sized collections of cells called COMMON1, COMMON2, and COMMON3 (abbreviated C1, C2, and C3). The simulation of each *RM* turn is associated with a different COMMON area (C area). Thus the first *RM* turn is associated with C1, the second with C2, the third with C3, and the fourth with C1. Each C area contains information such as an *RM* state, and whether the area is current. Each simulator reads and writes these cells as it tries to simulate an *RM* turn. The first n processes do not use these cells.

There are six message areas for process j which correspond to CELL(P,j). Three are used for messages from process j to the simulators, and three for messages from the simulators to process j. The message areas will be described after the description of the C areas.

For most values of j, |CELL(Q,j)|=2 (for some values of j, |CELL(Q,j)|=4). A cell that takes on two states is called a *bit*. A summary chart of the C area cells is given in Table 2.1.

Let h=|PROC(P,n+1)| (i.e., the size of the state set of the *RM* is h.) The *RM* portion of each C area is conceptually a set of $\lceil \log_2(h) \rceil$ bits which are used for encoding the state of the *RM*. Each of these $\lceil \log_2(h) \rceil$ "conceptual" bits is copied $2n+1$ times. Thus there are $(2n+1)\lceil \log_2(h) \rceil$ bits in the *RM* portion of each C area.

These $2n+1$ copies are needed for two reasons: to protect against one potential faulty bit and to protect against $n-1$ bits set in an out-of-date manner by $n-1$ slow simulators (as in Section 2.4.1). At any time, the majority of these bits is the "correct" value.

The STATUS portion of each C area is one "conceptual" bit, or $2n+1$ total bits. These bits specify whether the C area is currently in use. If the majority of the STATUS bits equal 1, then the C area is in use.

The SIMULATOR-ACCESS portion of each C area conceptually consists of n bits. The j^{th} bit is used by simulator j to determine if it is up-to-date as in Section 2.4.1. There are $2n+1$ copies of each of these n conceptual bits.

The MSG-PTR portion is a set of $2n$ conceptual bits (or n 3-valued cells) used by the simulators to remember which of three message areas is used by process j for its next request. The j^{th} pair of bits takes on three values, and is used with the message cells of process j. There are $2n+1$ copies of each of these $2n$ bits.

The MSG-SENT-VOTE portion consists of $(4n+1)(2n+1)$ pairs of bits which conceptually represent a single piece of information. There is a total of $2(4n+1)(2n+1)$ bits, but a pair of bits is utilized as one unit in this portion. These cells are used as $2n+1$ blocks of $4n+1$ cells. At each *RM* turn, the simulators use these cells during a vote which determines whether a process is sending a message.

The VOTE-TYPE portion consists of one bit (conceptually). This bit is complemented each time the C area is used. Thus for example, the first and fourth *RM* turns (which both use C1) have different values for their VOTE-TYPE bits. This bit is used with the MSG-SENT-VOTE bits as will be explained in Section 2.4.4. There are $2n+1$ copies of this bit.

Title	Conceptual Bits	Redundancy	Function
RM	$\log_2(h)$	$2n+1$	Code the *RM* state
STATUS	1	$2n+1$	Specifies if C area is in use
SIMULATOR-ACCESS	n	$2n+1$	Specifies if each simulator is uptodate
MSG-PTR	$2n$	$2n+1$	Specifies next message area to be used
MSG-SENT-VOTE	1	$O(n^2)$	Used by simulators to vote
VOTE-TYPE	1	$2n+1$	Used as a key for MSG-SENT-VOTE

(h=|PROC(P,n+1)|)

Table 2.1. Summary chart of each of three C area of cooperating system

2.4.2.2. Message areas

We give the details of the six mesasge areas for process j. They consist of sets of *response cells* R_{j0}, R_{j1}, and R_{j2} and sets of *request cells* S_{j0}, S_{j1}, and S_{j2} (j=1,...,n). A summary chart of these sets of cells is provided in Table 2.2.

Recall that the j^{th} cell of the managed system is a pair (L_j,M_j). If M_j takes on k values then there are (conceptually) $\lceil \log_2(k) \rceil$ bits in each of the six message areas for process j. The $\lceil \log_2(k) \rceil$ bits represent an encoding of the value of M_j. All of the bits of the request cells are triplicated for fault tolerance purposes (they are only set by one process). There are $2n+1$ copies of each bit in the response cells. All of the above bits are called *message* bits. There is a single additional conceptual bit for each message areas, called the *flag* bit. There are $2n+1$ copies of each flag bit. (The flag bits of the request areas may be set by the simulators, and thus $2n+1$ bits are

needed there too.) The MSG-PTR for process j specifies whether S_{j0}, S_{j1}, or S_{j2} will be used next for requests from process j.

The basic idea is that the first request from process j is sent in S_{j0}, a response arrives in R_{j1}, the next request is sent in S_{j1}, etc., until the third response arrives in R_{j0}, and the cycle starts again. After a message is sent in a message area, the flag bits are set to 1.

Title	Conceptual Bits	Redundancy	Function
request	$\log_2(k)$	3	Messages from process to RM
request flag	1	$2n+1$	Flag for request area
response	$\log_2(k)$	$2n+1$	Messages from RM to process
response flag	1	$2n+1$	Flag for response area

(The value k is the number of values taken on by M_j.)

Table 2.2. Summary chart of each of three sets of message areas for each process

2.4.2.3. Memory-simulator-function

This section defines f_j, the memory-simulator-function for process j. For a given system-memory-state m, the expression $val(R_{jk})$ is the integer obtained by viewing the states of the message bits of R_{jk} in m as the binary expansion of an integer. A similar definition applies to $val(S_{jk})$. The value of each conceptual bit in the expansion is the majority of the states of the $2n+1$ bits that represent the single bit. In general, "the value of a conceptual bit" is the majority of the states of the bits that represent the "conceptual" bit.

The memory-simulator-function f_j for the j^{th} cell only depends on the six message areas for process j. It turns out that the configuration of these six message areas in our simulation always satisfies:

(1) Between one and five of the message areas have at least $n+2$ of their flag bits equal to 1.

(2) Those areas that have at least $n+2$ flag bits equal to 1 are consecutively areas in the sequence R_{j0}, S_{j0}, R_{j1}, S_{j1}, R_{j2}, S_{j2}, R_{j0}, S_{j0}, etc. Thus for example, if R_{j1} and S_{j2} have at least $n+2$ of their flag bits equal to 1, then in order for the areas to be consecutively arranged, either S_{j1} and R_{j2} or R_{j0} and S_{j0} must have at least $n+2$ flag bits equal to 1.

The *current* message area is the one that has at least $n+2$ flag bits equal to 1, but is not followed by a message area with at least $n+2$ flag bits equal to 1. The current area is the one whose information is to be read next or was read last. Thus the simulated-memory-state depends primarily on the current area. For a memory-state m, if (1) and (2) above do not apply, then $f_j(m)$ may be defined arbitrarily. Otherwise, $f_j(m)$ is defined as follows:

(a) If the message area that follows the current area has exactly $n+1$ flag bits equal to 1, then $f_j(m)$=*transitional*.

(b) If the message area that follows the current area has at most n flag bits equal to 1 and the current message area is S_{jk} then $f_j(m)$=(REQUEST,$val(S_{jk})$); if R_{jk} then $f_j(m)$=(RESPONSE,$val(R_{jk})$).

The way that a new message area becomes current in the simulation is that its flag bits are set to 1. When $n+1$ become 1, the value of the memory-simulator-function is *transitional*, indicating that the simulated-memory-state is about to change. If there is an error transition,

there is no oscillation in the simulated-memory-state, since a *transitional* value for the memory-simulator-function does not change the simulated-memory-state. When n+2 flag bits become 1, the simulated-memory-state changes. Then if an error causes one of these n+2 bits to become 0, again the simulated-memory-state is not affected.

The initial states of all the cells in C2 and C3 are 0. The initial states of the cells of C1 are as follows. The *RM* section is the code of the initial *RM* state. Each bit is 0 in the SIMULATOR-ACCESS, MSG-SENT-VOTE, and MSG-PTR sections. Each bit in the STATUS and VOTE-TYPE sections is 1. This represents the fact that C1 is used first, and that the first *RM* turn is simulated first. Each of S_{j0}, S_{j1}, S_{j2}, R_{j1}, and R_{j2} have all bits initially 0. All of the message bits of R_{j0} are 0, and the flag bits are 1. Thus f_j(initial state)=(RESPONSE,0) for every j.

2.4.2.4. Error transitions

To prove Theorem 2.1 (with k=1) we only need to allow for one faulty cell. However, there is a larger class of faulty cells that are tolerable. This section describes the way that memory is divided into blocks, as mentioned in Section 2.1.2.2.

Each group of cells used for "conceptually" the same purpose may have one cell that has error transitions in a given turns history. For example, the $2n$+1 bits that represent a specific bit in an *RM* section of a C area may have one faulty bit. Similarly, one flag bit in each R_{jk} section may be faulty. The weakest assumption made is that only one of the $8n^2$+6n+1 cells in each MSG-SENT-VOTE area may fail.

Unreliability assumptions such as "one in $8n^2$" may seem unreasonable.

However, trivial modifications in the cooperating system permit stronger assumptions. For example, for MSG-SENT-VOTE, a modified system allows $2n$ failures out of $32n^2$. A more detailed discussion of these issues appears in Section 2.6. In addition, a system that requires a "one in $6n^2$" assumption which generalizes to $2n$ failures out of $28n^2$ is described.

Recall that $\mathcal{T}'$ is the set of turns histories of Q such that at least one of the processes $1,\ldots,n$ does not die, and for $j=1,\ldots,n$, process j dies iff process $n+j$ dies. One may define $\mathcal{T}_{sim}$ to be the set of $T\in\mathcal{T}'$ such that if e_1,e_2 are error transitions of T, then either e_1 and e_2 are error transitions for the same cell, or are error transitions for different conceptual cells. We prove a reformulated version of Theorem 2.1: Q faithfully simulates P with respect to $\mathcal{T}$ and $\mathcal{T}_{sim}$.

2.4.3 State transitions of the first n processes

This section describes process j ($1 \le j \le n$) in Q, and the process-simulator-function g_j. We will be slightly informal as we do not list every state of process j explicitly. The set PROC(Q,j) for j=1,...,n consists of three almost identical parts denoted Q_{j0}, Q_{j1}, and Q_{j2}. Intuitively, process j is in Q_{jk} when it expects a response in R_{jk}.

In Q_{jk}, process j tests the flag bits of R_{jk} to determine if a response has arrived. If not, process j remains in Q_{jk} and continues to vote on the flag bits. If a response has arrived, process j reads it and may send its next request in S_{jk}. After this request is sent, process j enters Q_{jk+1} and waits for a response in R_{jk+1}. In addition to this, process j handles a number of minor bookkeeping details. The details follow.

For each state $p \in$PROC(P,j) there are three states in PROC(Q,j) that are mapped (by g_j) to p, one in each of Q_{j0}, Q_{j1}, and Q_{j2}. The other states in Q_{j0}, Q_{j1}, and Q_{j2} are mapped to *transitional*. We discuss only Q_{j0} - Q_{j1} and Q_{j2} are similar.

There is a collection of states in Q_{j0} that for $p \in$PROC(P,j) are called the *voting states* of p. The "first" of these states, q, has $g_j(q)=p$. In these states successive flag bits in R_{j0} are voted on (i.e., each flag bit is read and process j determines if the majority are 0 or 1). If process j is in q, then at each of its next $2n+1$ turns process j enters $2n+1$ of these voting states, one to read each flag bit.

If process j determines that the majority of the flag bits are 0, then process j enters a state $q_1 \in Q_{j0}$. Process j stays in Q_{j0} since it is still looking for a message in R_{j0}. Let p_1 be the state that process j enters in P if its state is $p=g_j(q)$ and a process j turn occurs with L_j=REQUEST. Then

$g_j(q_1)=p_1$.

If the majority of the flag bits are determined to be 1, process j first sets the remaining flag bits of R_{j0} to 1 and then reads the message bits of R_{j0}. The value of the message bits that is read is denoted, v. The setting of the flag bits of R_{j0} is done to ensure that error transitions do not cause the simulated-memory-state to oscillate. Next, all flag bits of R_{j1} and S_{j1} are set to 0 in preparation for the next usage of those areas. Let k equal the number of values taken on by M_j. Process j next spends $3\lceil\log_2(k)\rceil+2n+1$ turns modifying S_{j0}. First (for $3\lceil\log_2(k)\rceil$ turns), the message bits of S_{j0} are set to what is written in M_j in P, if process j is in state p and the state of the j^{th} cell is (RESPONSE,v). In the last $2n+1$ turns, the flag bits of S_{j0} are set to 1. At the last of these $2n+1$ turns the process enters a state $q_2 \in Q_{j1}$. Let p_2 be the successor state of p when the state of cell j is (RESPONSE,v). Then $g_j(q_2)=p_2$. The function g_j maps all other states described above to *transitional*.

Starting in q_2, process j executes a similar procedure, looking for messages in R_{j1}, clearing flag bits in R_{j2} and S_{j2}, sending messages in S_{j1} and ending up in Q_{j2}.

It is trivial to verify that condition (2′) on faithful simulation is satisfied for any turns history in which process j does not die.

To finish the process-correspondence we must partition MEMSTATES(P) for each pair $q_1,q_2 \in$ PROC(Q,j) such that $g_j(q_1)$ and $g_j(q_2)$ are both not equal to *transitional*. We only give the partition if q_2 follows q_1 in the manner described above, otherwise the partition can be arbitrary.

The simulated-process-state of process j should change when the system can recognize that process j is attempting to simulate a turn of process j in

P. Thus, if process j is sending a request, the simulated-process-state changes as soon as the simulated-memory-state reflects this request. Formally, if q_2 follows q_1 as above and $q_1,q_2 \in Q_{j0}$ (i.e., q_2 follows q_1 after the flag bits were determined to be 0), then $\text{MEMSTATES}_1(q_1,q_2)=\emptyset$ and $\text{MEMSTATES}_2(q_1,q_2)=\text{MEMSTATES}(P)$. If $q_1 \in Q_{j0}$, $q_2 \in Q_{j1}$, then $\text{MEMSTATES}_1(q_1,q_2)=\{m \in \text{MEMSTATES}(P)$: the value of L_j in m is RESPONSE$\}$ and $\text{MEMSTATES}_2(q_1,q_2)=\{m \in \text{MEMSTATES}(P)$: the value of L_j in m is REQUEST$\}$. In the first case, the simulated-process-state changes when process j leaves q_1. In the second case, the simulated-process-state changes when the simulated-memory-state reflects the request.

Since process j first looks for a message in R_{j0}, its initial state is the state $q \in Q_{j0}$ such that $g_j(q)=p$, where p is the initial state of process j in P.

2.4.4 Simulation of the RM (state transitions of the simulators)

The state transitions of the n simulators (processes $n+1,...,2n$ of Q) will be presented algorithmically. It should be clear how to transform the algorithms into state sets with a transition function. Before giving the algorithm, we give a more detailed informal description of the protocols mentioned in Section 2.4.1. If the reader has forgotten the overview of Section 2.4.1, we suggest that he reread it at this point.

The simulation of each *RM* turn is associated with a C area. The C area in use is the one with the majority of its STATUS bits equal to 1. The simulators simulate an *RM* turn by updating the encoding of the *RM* state (in a different C area) and by sending responses.

At the simulation of an *RM* turn, the simulators must agree on whether a request is being sent. The problem with a redundancy technique where $2n+1$ cells protect against a faulty cell and $n-1$ slow simulators is as follows. The simulators that are not slow may have different perceptions as to whether a request is being sent. If n cells are set to reflect that a message is being sent, and n to reflect that a message is not being sent, and the $2n+1^{st}$ is faulty, the redundancy does not permit the simulators to agree. In contrast, when the *RM* part of a C area is updated (for example), all $2n+1$ bits representing a conceptual bit are set the same way by the up-to-date simulators.

To explain the basic way agreement is reached. We first use the (incorrect) hypothesis that no simulator is out-of-date. Assume that there are six three-valued cells initialized to -1. (These six cells are used in the same manner that the MSG-SENT-VOTE cells will be used.) Each simulator first votes on the flag bits of S_{jk} (where j depends on the *RM* state coded in the C

area and k depends on the MSG-PTR for process j). Each simulator then tries to set the first three of the six cells to 0 or 1 based on whether the simulator perceived the majority of the flag bits of S_{jk} to be 0 or 1. Each one of these cells is set (with a test-and-set) only if its current state is -1, but if its current state is 0 or 1, then its state is unchanged. After a simulator tries to set the first three cells it votes on them. If the majority are 0 (1), then the simulator tries to set the last three cells to 0 (1) as above. (If one of the first three cells is -1 due to an error and there is no majority, the simulator may use either 0 or 1.) Each cell is set only if its current state is -1. The majority of the last three cells is used as the decision as to whether a request is being sent. Note that each non-faulty cell is set only once; by the simulator that reaches it first.

Assume that at most one of these six cells is faulty. Then each simulator will discover the same majority for the last three cells. If the faulty cell is among the last three cells then all simulators will discover the same majority for the first three cells. Thus the last three cells are set to the same state and even with a faulty cell each simulator discovers the same majority. Assume that the faulty cell is among the first three. Then each of the last three cells are perceived to be the same by all simulators and hence the majority is the same. Note that if each simulator originally concluded that a request is (not) being sent, then the final decision is that a request is (not) being sent.

To initialize the cells to -1 the VOTE-TYPE is used and four-valued cells are used as pairs of bits using the following convention. Successive *RM* turns that use the same C area have a different value for VOTE-TYPE. The VOTE-TYPE value is used for one bit of each pair of bits in the MSG-SENT-VOTE

cells. The first time a C area is used is of "type" 0 and the MSG-SENT-VOTE cells are set either to 00 or 01; the second time is of "type" 1 and the MSG-SENT-VOTE cells are set to either 10 or 11, etc. In an odd numbered usage of a C area, if the state of a cell is 11 it is interpreted as -1 since odd usages set the first bit to 0. In the same step it is set (with a test-and-set) to 00 or 01.

Since a slow simulator may set one incorrect cell during the simulation of an *RM* turn (as in Section 2.4.1), the six cell solution does not work. If $2n+1$ blocks of $4n+1$ cells are used, the solution works as will be proved in Section 2.5.

To use three-valued cells one may adapt the above technique, reinitializing the cells to -1 between usages. We present the simpler four-valued cells solution. It is an open problem to design the protocol with two-valued cells.

The above protocol is the only place that a test-and-set is needed in our simulation. For uniformity, however, we use a test-and-set for all writing.

The following is the algorithm for each simulator. It is given in two parts, the synchronizing part and the managing part. The simulator executes the entire synchronizing part between every pair of turns in the managing part to verify that it is up-to-date. If the simulator determines that it is up-to-date it resumes its activity in the managing part. Otherwise, it remembers what it was up to in the managing part, and starts on a new *RM* turn. (It remembers what it was up to since it may return to it due to transients in the system.) In the managing part is the activity necessary for the simulation

of the *RM*.

The algorithm has read and write operations. A "read" is a change of process-state based on the state of a cell. All writes are with test-and-sets. Voting is a series of turns where a simulator determines the majority value of a set of cells. While all reading involves voting on multiple copies of a single "conceptual" cell, we only make the voting explicit when necessary. If i=3 then Ci+1=C1.

2.4.4.1 Synchronizing part for simulator j:

Step 1. COMMENT. Determine which C area is current.

Vote on the STATUS of each C area. If all three majorities are 0 or all are 1, go to step 1. If two majorities are 0 and one is 1, let i be the one that is 1. If two majorities are 1 and one is 0, let i be the one of the two which precedes the other (i.e. C1 precedes C2 which precedes C3 which precedes C1).

Step 2. COMMENT. If simulator j has worked on the current *RM* turn, then simulator j may execute a step in the managing part.

Vote on the $2n$+1 bits for simulator j in SIMULATOR-ACCESS of Ci. If the majority is 1 then resume the managing part (i.e., execute the next turn for simulator j in the managing part). If it is 0, write a 1 on all $2n$+1 SIMULATOR-ACCESS bits for simulator j. Also, remember the place in the managing part that simulator j was up to in Ci-1 (in case simulator j will have to return to that part of the simulation).

Step 3. COMMENT. Determine if C_i is still current. (If it is, and C_i is changed while simulator j is reading it, simulator j will be able to discover this, because its SIMULATOR-ACCESS bits will be set to 0.)

Vote on the STATUS of each C area (as in step 1). If C_i loses the vote, set the SIMULATOR-ACCESS bits for simulator j in C_i to 0 and go to step 1 of the synchronizing part.

Step 4. COMMENT. Determine the state of each cell in C_i.

Read C_i.

Step 5. COMMENT. Verify that C_i was not changed while simulator j was reading it. It suffices to check the SIMULATOR-ACCESS.

Vote on SIMULATOR-ACCESS for simulator j in C_i. If the majority is 1, go to step 1 of the managing part. Otherwise, set all $2n+1$ SIMULATOR-ACCESS bits for simulator j to 0 and go to step 1 of the synchronizing part.

2.4.4.2 Managing part:

Step 1. COMMENT. Prevent the usage of C_i-1.

Set all $2n+1$ STATUS bits of C_i-1 to 0.

Step 2. COMMENT. Clear the SIMULATOR-ACCESS bits for C_i-1. This prevents subsequent modification of C_i-1 from going unnoticed by a simulator that is reading it.

Set all $(2n+1)n$ SIMULATOR-ACCESS bits of C_i-1 to 0.

Step 3. COMMENT. Determine the process to be handled in this "*RM* turn" and whether the process is sending any messages.

Determine which process (process k) is to be handled on the basis of the *RM* state in Ci (as read in the synchronizing part). Determine on the basis of the MSG-PTR for process k of Ci (MSG-PTR(k)) if requests are to be expected in S_{k0}, S_{k1}, or S_{k2}. (If MSG-PTR(k)=h then requests are expected in S_{kh}.) Assume it is in S_{k0}, the modifications for S_{k1} and S_{k2} are obvious. Vote on the $2n+1$ flag bits of S_{k0}. The result of this vote is the *initial decision*.

Step 4. COMMENT. Participate in the protocol that permits the simulators to agree on whether process k is sending a request.

Write the first block of $4n+1$ MSG-SENT-VOTE cells in Ci. The first bit written for each such "pair of bits" is the majority value of the VOTE-TYPE bits of Ci (as read in the synchronizing part). The second bit is the initial decision. Write with a test-and-set that changes the cell only if its first bit disagrees with the VOTE-TYPE. Write the other $2n$ blocks with the following change. For a given block, the second bit in each pair is the majority value obtained in a vote on the second bits of the previous block. (This vote is taken while the simulator sets the bits.)

Step 5. COMMENT. If a request is being sent, read it and set the rest of the flag bits of S_{k0} to 1. This prevents errors from changing the simulated-memory-state.

Vote on the last block of MSG-SENT-VOTE. If the majority is 0 go to 7. If it is 1, set the $2n+1$ flag bits of S_{k0} to 1 and read S_{k0}.

Step 6. COMMENT. Simulate the setting of the k^{th} cell by the *RM*.

If the state of the *RM* and value of S_{k0} indicate that nothing would have been written into the k^{th} cell in the managed system go to 7. Otherwise write into R_{k1} based on the *RM* state and S_{k0}. The value written is the encoded version of the value that would have been written into M_k. Set the flag bits of R_{k1} to 1.

Step 7. COMMENT. Update the information needed for the handling of the next process in C*i*+1.

Update C*i*+1. Set the *RM* portion according to the *RM* state and the value of S_{k0} (if relevant). Set the VOTE-TYPE bits to the value of VOTE-TYPE in C*i* unless *i*=3, in which case VOTE-TYPE is complemented. If messages were sent to process *k*, increment MSG-PTR(*k*) (mod 3).

Step 8. COMMENT. Start preparing C*i*+1 to become current.

Set each STATUS bit in C*i*+1 to 1.

Step 9. COMMENT. Terminate the use of C*i*.

Set each STATUS bit in C*i* to 0.

2.5. Proof of Theorem 2.1.

It is easy to show that every essential-history of P equals an essential-simulated-history Q. In this section we prove that every essential-simulated-history of Q equals an essential-history of P. We only consider turns histories for Q from $\mathcal{T}_{sim}$ (defined in Section 2.4.2.4). Essential-simulated-histories from these turns histories must equal essential-histories for P from $\mathcal{T}$ in which the RM and at least one process has infinitely many turns. The following terminology is needed.

Definitions. C_i is *active* in a memory state of Q if at least $n+1$ of its STATUS bits are 1. Otherwise it is *passive*. If $t_0 t_1 \cdots \in \mathcal{T}_{sim}$, simulator j is *live in C_i after t_l* if after t_l at least n of simulator j's bits in SIMULATOR-ACCESS of C_i are 1 and none are faulty, or at least n are 1 and one is faulty and is equal to 0, or at least $n+1$ are 1 and one is faulty and is equal to 1.

A live simulator might vote on its SIMULATOR-ACCESS bits and determine a majority of 1, even if the simulator sets no additional bits to 1.

Before each turn that a simulator has in the managing part it executes the synchronizing part. During this execution, the simulator votes (in step 1) on which C area is to be considered current (called C_i in 2.4.4.1). If the current one (incorporated in the simulator's state) is C_i, then at the managing turn, the simulator *executes a turn in C_i*.

There are occasions in the managing part that a simulator's turn depends on the state of C_i that it read when last in the synchronizing part. We say that "the state of a cell used by the simulator equals its current

state" to mean that the state last read by the simulator in the synchronizing part equals the current state.

During the simulation of an *RM* turn, once one simulator finishes step 4 of its managing part it is determined whether any request area will be read at this *RM* turn. Thus, after the most advanced simulator completes step 4, we can determine how the system will "progress". Important conditions that exist at the turn at which step 4 is completed are incorporated into the next definition.

"The majority of a set of bits does not change" means that if the current majority is for example, 1, then at least half of the bits stay 1. (This is stronger than insisting that at each point the majority is 1.) "A C area does not become active" means that at least half of its STATUS bits stay 0.

We give an overview of the definition. Conditions (1)-(5) discuss the memory-state of the C areas and when certain bits are set. Conditions (1) and (2) specify that C1 is "current", conditions (3) and (4) specify that the "fastest" simulator has just finished step 4, and condition (5) specifies the contents of C1. Condition (6) ensures that all simulators live in C1 do roughly the same thing.

Condition (7) imposes constraints on the system's progress. Conditions (7)(a)-(d) discuss which C areas are used next, and the behavior of the memory-state of C1. Condition (7)(e) discusses some simulators not live in C1. Conditions (7)(f)-(h) are needed to prove that the simulators agree on whether a request is being sent.

Definition. Let $t_0 t_1 \cdots$ be a turns history for Q. Q is *stable in C1 at step i* if:

(1) C1 is active after t_i.

(2) C2 and C3 are passive after t_i.

(3) At t_i the last MSG-SENT-VOTE cell of C1 was written by a simulator live in C1.

(4) Let $k<i$ and assume that at t_k the last MSG-SENT-VOTE cell of C1 was written by a simulator live in C1. Let t_l be the turn at which C3 was most recently active. Then $k<l$.

(5) The *RM* section of C1 is the code of an *RM* state and each MSG-PTR equals 0, 1, or 2 after t_i.

(6) No simulator live in C1 is past the first turn of step 5 in its managing part after t_i. Every simulator live in C1 will use the current value of C1 (i.e., the current majority of the MSG-SENT-VOTE, VOTE-TYPE, MSG-PTR, and *RM* sections) at its next managing turn in C1 if the turn occurs before C3 becomes active.

(7) For every sequence $t_{i+1}' t_{i+2}' \cdots$ such that $t_0 \cdots t_i t_{i+1}' t_{i+2}' \cdots \in \mathcal{T}_{sim}$:

(a) C2 does not become active after t_i before one of its STATUS bits is set to 1 by a simulator live in C1.

(b) C3 does not become active after t_i before C2 becomes active and one of C3's STATUS bits is set to 1 by a simulator live in C2.

(c) C1 does not become passive after t_i before one of its STATUS bits is set to 0 by a simulator live in C1.

(d) The state of the *RM*, the VOTE-TYPE, and the MSG-PTR values represented in C1 do not change after t_i before a cell is set in MSG-SENT-VOTE of an active C2 by a simulator live in C2.

(e) If a simulator is live in C2 (C3) after t_i it is in the synchronizing part having just set its SIMULATOR-ACCESS bits to 1. Furthermore, if it has started voting on the STATUS of C2 (C3) in step 3, the sum of the number of bits it has determined to be 1 plus the number of bits it has not read which may become 1 before a STATUS bit is set to 1 in C2 by a simulator live in C1 (resp. before a STATUS bit is set to 1 in C3 by a simulator live in C2) is at most n.

(f) No more than n of the cells of MSG-SENT-VOTE of C1 have their first bit different from the value of VOTE-TYPE of C1 before a MSG-SENT-VOTE bit is changed to a different type by a simulator live in C1.

(g) No more than n of the cells of MSG-SENT-VOTE of C2 or C3 have their first bit equal to the value of VOTE-TYPE of C1 before a MSG-SENT-VOTE bit in C2 (C3) is changed to this type by a simulator live in C2 (C3).

(h) The majority of the last block of MSG-SENT-VOTE of C1 does not change before a cell is set in MSG-SENT-VOTE of an active C2 by a simulator live in C2.

Analogous definitions of being stable in C2 and C3 are left to the reader. If Q is stable at step i, then t_i is a *stabilization*.

The *decision* of the i^{th} stabilization is the majority of the second bits of the last block of MSG-SENT-VOTE cells at the i^{th} stabilization. If the decision is 1, the simulators have decided that there is a request to be read. If the state of the *RM* represented in C1 at the i^{th} stabilization is in F_j, the i^{th} stabilization *handles* process j.

The key to proving correctness is to describe how the system progresses

from one stable state to the next. The constraints on the system-state at stabilization are used to prove that the simulation is correct. The next two lemmas establish the existence of stable states.

In Section 2.5 we often make statements about C1, S_{j0}, R_{j0}, and MSG-PTR(j)=0 which apply without loss of generality to C2, S_{j1}, R_{j1}, and MSG-PTR(j)=1 and C3, S_{j2}, R_{j2}, and MSG-PTR(j)=2. The details are left to the reader.

We refer to the value of a response area as being "correct based on an *RM* state in a C area and a value in a request area". This means that if for example, the *RM* state in the C area is q and the value of the request area is v, then the response equals the response that the *RM* would have sent in P to M_j while in state q reading the value v. A similar definition applies to an "*RM* portion of a C area being correct based on an *RM* state and request".

Lemma 2.4 (progress lemma).

Let $T=t_0t_1\cdots\epsilon\mathcal{T}_{sim}$ be a turns history for Q. Assume that after t_l, Q is stable for the i^{th} time. Assume it is stable in C1 and that the i^{th} stabilization handles process j. If the decision of the i^{th} stabilization is 1 and MSG-PTR(j)=0 at the stabilization in C1, assume that for every sequence $t_{l+1}'t_{l+2}'\cdots$ such that $t_0\cdots t_lt_{l+1}'\cdots\epsilon\mathcal{T}_{sim}$ the value of every conceptual message bit of S_{j0} does not change before C2 becomes active and a simulator live in C2 reaches step 4 of its managing part.

Then after t_l, the system becomes stable in C2. The state represented in the *RM* portion of C2 is correct based on the state of the *RM* at stabilization (and value of S_{j0} at stabilizaiton if the decision is 1).

MSG-PTR(j) is incremented if the current *RM* state and value of S_{j0} indicate that a response is to be sent.

Proof. We show that most activity of the simulators helps achieve the target stabilized state. The small amount of activity that disagrees with the target is shown to be tolerable.

First observe the simulators that are live in C1 and execute managing turns in C1. Those simulators that first execute managing turns in C2 or C3, or are not live in C1 will be discussed later. By (6), all live simulators use the current information stored in C1. While they execute steps 1 and 2 in the managing part they only reinforce the passivity of C3. By (7)(h), steps 3 or 4 cannot affect the majority of the last block of MSG-SENT-VOTE.

Step 5 involves reading a request area iff the decision is 1. The simulators agree on the same decision by (7)(h) and the first part of (6) (i.e. since no simulator starts to vote on the decision before stabilization.) If the decision is 1, all simulators determine the same value of S_{j0} by the hypothesis of the lemma - the value of S_{j0} does not change. (Also, by the second half of (6) and by (7)(d) all of these simulators use the same MSG-PTR and *RM* information and thus read the same request area.) Step 6 is not relevant here. At step 7, C2 is updated. Since these simulators know the current *RM* state (by (6) and (7)(d)), and determine the same value of S_{j0} if the decision is 1, the *RM* and MSG-PTR sections are updated according to target. By (6), no live simulator is up to step 7 at the i^{th} stabilization. Thus some live simulator completes step 7 before any live simulator starts steps 8 and 9. By (7)(a)-(c), C1 does not become passive and C2 and C3 do not become active before some simulator reaches step 8. Thus each simulator live in C1

determines at each execution of the synchronizing part that it should execute its next managing turn in C1.

Next, C2 is activated (while C1 is still active) by the first simulator to get through step 8. Then C1 is made passive, but by (7)(c), this occurs after some live simulator starts step 9. In particular, C1 does not become passive until all STATUS bits of C2 are set to 1.

After C1 becomes passive, an error transition may reactivate C1. Thus, after a simulator becomes live and executes managing turns in C2, it may reenter the managing part for C1. Since the SIMULATOR-ACCESS bits of C1 for such simulators are not yet set to 0, any simulator that reenters C1, will resume its activity in the middle of the protocol for C1. For this purpose, when the simulator started C2 (after being in C1) it remembered where in C1 it was up to in step 2 of the synchronizing part. Ultimately, C1 stays passive since all STATUS bits are set to 0 in step 9 of C1, or step 1 of C2 and the simulators do not enter an infinite loop. Note that an error transition can occur for only one of the STATUS bits. After C2 becomes active, all of the managing turns in C1 due to C1's sporadic reactivations are a reenactment of the above discussion.

Now Q stabilizes in C2. C1 is passive since its STATUS bits are set to 0 by simulators live in C2 or simulators in C1 executing step 9. C3 is passive by (7)(b) of the i^{th} stabilization. (By (7)(e), no simulators live in C2 have set a STATUS bit of C3 to 1.) C2 is active by the action of processes in C1, before C1 became passive. Thus (1) and (2) are satisfied soon after C2 becomes active. By (7)(e), all simulators already live in C2 progress through their managing parts from step 1. Also, simulators who first become live in C2 start from step 1. The live simulators come sooner or later to step 4 where

the MSG-SENT-VOTE cells are set. When the last MSG-SENT-VOTE cell is reached for the first time Q stabilizes as we now show.

To verify (4), note that this is the first time that this last cell has been set by a simulator live in C2 since C1 was active since all live simulators begin the managing part of C2 from step 1. (If after a simulator started the managing part, errors made it appear that C1 was active and a simulator reverted to C1, this apparent reactivation must occur before step 1 is completed and thus no simulator is up to step 4. Once step 1 is finished, C1 cannot appear to become active again.)

Condition (5) follows from the activity of the simulators before C2 was activated. To verify (6) examine a simulator live in C2. If it became live after the C2 became active, (6) follows immediately by the above discussion. If it were live before C2 became active, by (7)(e) it does not read C2 in step 4 of the synchronizing part until a STATUS bit is set in C2 by a simulator live in C1, and thus (6) follows.

Condition (7) will be verified together with a discussion of those simulators that executed managing turns although they were not aware that C1 was active or were not live in C1. Each such simulator may write one bit that disagrees with the above discussion. However, from the i^{th} stabilization until the time conjectured to be the $i+1^{st}$, each simulator may not disagree with more than one bit. After the first disagreement the synchronizing part is executed and the simulator detects that C1 is active or that C2 is active (if C1 is already passive). This follows from (7)(a)-(c).

Due to cells that are set to values that disagree with the above discussion, C1 may become active n times after it first becomes passive. In addition to one error transition, $n-1$ of C1's STATUS bits may be set to 1 by

simulators not initially live in C1 (a simulator not live in C1 is called *outdated*). Similarly, C2 may not stay active until all $2n+1$ STATUS bits are set to 1. However, once all $2n+1$ STATUS bits of C2 are set to 1, at least $n+1$ will remain 1 until one is set to 0 by a simulator live in C2. It is to tolerate this outdatedness that all $2n+1$ STATUS bits of C1 are set to 0 either by simulators in C2 (which stays active after C1 first becomes passive), or by simulators in C1. At least $n+1$ remain 0 at least until one is set to 1 by a simulator live in C3 (as we discuss soon).

At the turn conjectured to be the stabilization (1), (2), and (5) cannot be harmed by outdated processes. There is enough protection for one faulty bit and $n-1$ bits that are set to states that differ from the target. Similarly, (3) and (4) are unaffected since once the "fastest" simulator finishes step 1 of the managing part for C2, C1 stays passive and C2 stays active. Condition (6) is unaffected since each cell in C2 has $2n+1$ copies and is tolerant of one faulty bit and $n-1$ bits sets outdatedly.

The fact that outdatedness is limited helps verify (7). Condition (7)(a) is inherited from (7)(b) of the i^{th} stabilization. To verify (7)(b), it must be shown that C1 may not become active before C3 becomes active and at least one of C1's STATUS bits are set to 1 by a simulator live in C3. Consider the STATUS bits of C1. All $2n+1$ bits are set to 0 in step 1 of the managing part of C2. After they are set to 0, at most n of them may become 1 due to faults or simulators that are out-of-date. Thus, C1 can become active only after an up-to-date simulator sets a STATUS bit to 1, i.e., after C3 becomes active and a STATUS bit of C1 is set by a simulator live in C3. Until then, the number of STATUS bits that equal 0 minus the number of out-of-date simulators whose next turn is to set a STATUS bit in C1 to 1 is at least $n+1$.

Conditions (7)(c) and (7)(d) are straightforward. For a simulator live in C3, (7)(e) is inherited from the i^{th} stabilization. To verify (7)(e) for a simulator live in C1 consider the time that such a simulator set its SIMULATOR-ACCESS bits. Since it is still live, some of the SIMULATOR-ACCESS bits must have been set to 1 after they were all set to 0 in step 2 of the managing part by simulators live in C2. But if they were set to 1 that recently, then the only bits that this live simulator may have determined to be 1 (in step 3) are the n-1 bits set by outdated simulators and the one faulty bit.

For (7)(f), all cells set in MSG-SENT-VOTE of C2 by live simulators before the turn conjectured to be the i+1st stabilization are set to the correct type. At most n cells may obtain different types by error transitions or out-of-date processes (some of these n may already be of a different type). Only after C3 and C1 become active does C2 become active with MSG-SENT-VOTE cells set to a different type. This can be proved by essentially repeating the proof of the progress lemma for two more stabilizations. To prove the entire progress lemma, we need a condition about certain message bits not changing. However, to use the portion of the proof involving the VOTE-TYPE this added hypothesis is not needed.

Condition (7)(g) for C1 follows from (7)(f) for the i^{th} stabilization. Condition (7)(g) for C3 follows from (7)(g) for C3 at the i^{th} stabilization.

To finish the proof, (7)(h) must be verified. It will be also shown that if the initial decision of every simulator in C2 is 0 (1), then the majority of the last block of $4n$+1 bits is also 0 (1).

Condition (7)(h) is proved in two parts. First we count how many cells

may have states that differ from all initial decisions. Then we analyze the protocol given the amount of unreliability.

By condition (7)(g) of the i^{th} stabilization, as many as n cells of MSG-SENT-VOTE might agree with the VOTE-TYPE before any simulator reaches step 4. These cells are not be reset in step 4, because they are already of the correct type. Thus they may disagree with every initial decision. Once a live simulator starts step 4, most of the MSG-SENT-VOTE cells are set by live simulators. At most n-1 simulators can set at most one cell each in an outdated manner. In addition, there is at most one error transition. Thus at most $2n$ cells of the $(4n+1)(2n+1)$ cells are potentially different from states correctly assigned to them. There will be not more than $2n$ of these before the simulators are up to step 4 of C3.

Now, (7)(h) can be verified. Consider the situation that all simulators have the same initial decision. Then that value appears on at least $2n+1$ of the cells of the first block, and each simulator uses it as the decision for the second block. Similarly, this decision wins all votes for all blocks.

If both initial decisions are represented it does not matter what the final decision is as long as it is agreed on. At least one of the $2n+1$ blocks does not have any of the $2n$ uncertain bits. Each simulator that emerges from that block emerges with the same decision. From that block on, it reduces to the situation that each simulator has the same initial decision. Condition (h) follows. □

The progress lemma shows that the simulators perform as expected, (if S_{j0} does not change when relevant). The next lemma uses the progress lemma

together with information about all the processes at stabilization, to prove that the stabilizations take place, and the processes behave as would be expected.

If process j is in Q_{j0} the *bits already cleared in* R_{j1} or S_{j1} refer to those flag bits of R_{j1} or S_{j1} that were set to 0 during process j turns in Q_{j0}.

In the next lemma "MSG-PTR(j)=0 at a stabilization", means that MSG-PTR(j)=0 in Ci where Ci is the C area that was stabilized.

The last 2n states of Q_{jk} are the collection of states in Q_{jk} from which the state of process j enters Q_{jk+1} after at most $2n$ process j turns. Recall that the voting portion of Q_{jk} is the set of states in which the flag bits of R_{jk} are voted on.

Lemma 2.5. Let $T \epsilon \mathcal{T}_{sim}$. For $i>0$.

(i) There is an i^{th} stabilization.

(ii) If the decision of the i^{th} stabilization is 1, and MSG-PTR(j)=0, then at least $n+1$ flag bits of S_{j0} were 1 for part of the time between the i-1^{st} and the i^{th} stabilizations.

(iii) For $j=1,...,n$, at the i^{th} stabilization, if MSG-PTR(j)=0 then process j is in the last $2n$ states of Q_{j2}, in Q_{j0}, or in the voting portion of Q_{j1}. If in addition at least $n+1$ flag bits of S_{j0} were 1 for part of the time between the i-1^{st} and the i^{th} stabilizations then process j is in the last $2n$ states of Q_{j0} or in the voting portion of Q_{j1}.

(iv) For $j=1,...,n$ at the i^{th} stabilization, assume MSG-PTR(j)=0. Let x = the number of simulators that are not live in the active C area and whose next managing turn is to write a flag bit R_{j1} (S_{j1}). Let y = the number of

bits cleared in R_{j1} (S_{j1}) that are equal to 1. If none of the bits cleared in R_{j1} (S_{j1}) are faulty then $x+y \leq n-1$. If one such bit is faulty and is equal to 1 then $x+y \leq n$. If one is faulty and is equal to 0 then $x+y \leq n-1$.

(v) For $j=1,\ldots,n$ at the i^{th} stabilization, assume that MSG-PTR(j)=0, and that process j is in Q_{j2} or in Q_{j0} (but not the last $2n$ states of Q_{j0}). Let x = the number of simulators that are not live in the active C area whose next managing turn is to write a flag bit in S_{j0}. Let y = the number of flag bits of S_{j0} that are equal to 1. If none of the flag bits in S_{j0} are faulty then $x+y \leq n-1$. If one is faulty and is equal to 1, then $x+y \leq n$. If one is faulty and is equal to 0, then $x+y \leq n-1$.

Proof. Use induction on i. The basis is left to the reader. Inductively assume (i)-(v) for i, and prove (i)-(v) for $i+1$.

To prove (i) and (ii) for $i+1$ it suffices to use the progress lemma if it can be shown that the message bits of S_{j0} do not change before the turn conjectured to be the $i+1^{st}$ stabilization (if the decision of the i^{th} stabilization is 1 and MSG-PTR(j)=0). Assume that the decision is 1 and process j is being handled with MSG-PTR(j)=0. By (ii), at least $n+1$ of the flag bits of S_{j0} were 1 between the $i-1^{st}$ and i^{th} stabilizations. By (iii), process j is in the last $2n$ states of Q_{j0}, or the voting portion of Q_{j1} at the i^{th} stabilization. While in Q_{j0} process j set all the message bits in S_{j0}, and each of the three bits that represent one conceptual bit were set to the same state. Thus, the only way that the message bits of S_{j0} may change after the i^{th} stabilization is if process j cycles through Q_{j1}, Q_{j2}, and back into Q_{j0}. However, even if process j gets past the voting portion of Q_{j1} it clears the flag bits of R_{j2} while in Q_{j1}. As long as the majority of the flag

bits of R_{j2} are 0, process j certainly cannot enter Q_{j0}. Since MSG-PTR(j)=0, the flag bits of R_{j2} are not set to 1 by live simulators before the time conjectured to be the i+1st stabilization. At most n bits set to 1 by outdated simulators or errors. Thus any vote taken on the flag bits of R_{j2} determines that the majority are 0.

To verify (iii) - (v) consider first a process (process j) not handled at the i^{th} stabilization. There are no bits set in the flag section of R_{j1} (except by outdated simulators) before the time conjectured to be the i+1st stabilization. By (iv) at the i^{th} stabilization process j cannot progress beyond the voting portion of Q_{j1} since any vote on R_{j1} will result in a majority of 0. Each outdated simulator executes at most one turn which modifies the flag bits of R_{j1} (as in the proof of the progress lemma). If process j started to vote prior to the i^{th} stabilization, no additional flag bits of R_{j1} could have been equal to 1 since process j is the only process that ever sets the flag bits of R_{j1} to 0. Since MSG-PTR(j) is 0 at the i+1st stabilization iff it is 0 at the i^{th}, the first part of (iii) is proved. The second part of (iii) follows from (v) at the i^{th} stabilization and the fact that each outdated simulator sets at most one bit. Conditions (iv) and (v) similarly follow for i+1.

Assume process j is handled at the i^{th} stabilization and MSG-PTR(j)=0 at the turn conjectured to be the i+1st. If MSG-PTR(j) was 0 at the i^{th} stabilization, then (iii)-(v) follow as in the case that process j is not handled. Although the simulators may set flag bits in S_{j0} between the i^{th} and i+1st stabilizations, this occurs only if at the i^{th} stabilization process j is "at least" in the last $2n$ states of Q_{j0} (by (ii) and (iii)). MSG-PTR(j) cannot equal 1 at the i^{th} stabilization since the i+1st stabilization follows

from the i^{th} as described in the progress lemma.

If MSG-PTR(j)=2 at the i^{th} stabilization and MSG-PTR(j)=0 at the time conjectured to be the i+1st, then the decision of the i^{th} stabilization is 1 (since the i+1st follows from the i^{th} as in the progress lemma). By induction, process j was "at least" in the last $2n$ states of Q_{j2} at the i^{th} stabilization. When and if process j finishes the voting portion of Q_{j0}, it clears the flag bits of R_{j1}. Thus process j cannot get past the voting portion of Q_{j1} before the i+1st, proving the first part of (iii). Also, by (iv) at the i^{th}, at most n flag bits of S_{j0} may become 1 until process j enters the last $2n$ states of Q_{j0}, and (iii) is verified. To verify (iv), note that once process j clears the flag bits of R_{j1} and S_{j1} only outdated turns and error transitions will set them. This is because no live simulators set bits in R_{j1} or S_{j1} since MSG-PTR(j)=2 at the i^{th} stabilization. Condition (v) follows from (iv) at the i^{th} stabilization. □

Note that (iii) - (v) impose the following structure on the communication between each process and the simulators. Process j begins by voting in Q_{j0}. Since the flag bits of R_{j0} are initially 1, process j may send a request and enter Q_{j1}. By (v) the majority of the flag bits of S_{j0} are not equal to 1 until process j enters the last $2n$ states of Q_{j0}. By (iv), process j votes in Q_{j1} until a response is written in R_{j1}, at which point MSG-PTR(j) is set to 1. The simulators then see no message in S_{j1} until process j sets a flag bit to 1 (by (v)). Process j next sends a request in S_{j1} and votes in Q_{j2} until the flag bits of R_{j2} are set to 1. At the stabilization at which the flag bits of R_{j2} are set to 1, MSG-PTR(j) is set to 2 and process j may send a request in S_{j2}. Finally, process j votes in Q_{j0} until the flag bits of R_{j0} are

set to 1, and the cycle is completed.

Thus, the set of message areas with at least $n+2$ flag bits equal to 1 is consecutively arranged. Once all $2n+1$ bits are set to 1, at least $2n$ remain 1 until process j clears them. Also, the simulated-memory-state does not oscillate due to error transitions. Once $n+2$ bits are set to 1 at least $n+1$ remain 1 until process j clears them, as at most one of these bits may be faulty. If $n+1$ of the bits are 1, the value of the memory-simulator-function is *transitional*, protecting the simulated-memory-state from oscillating.

Let $T' \epsilon \mathcal{T}_{sim}$ be a turns history of Q. A turns history, T, of P with essential-history equal to the essential-simulated-history of T' is built around the turns at which the simulated-process-states of the first n processes change. Each change in the simulated-process-state of process j is associated with a turn for process j in T. The ordering of these turns in T is given by the ordering of the turns of T' at which the simulated-process-states change. (The same turn in T' cannot cause both process j_1 and process j_2 to change their simulated-process-states ($j_1 \neq j_2$)). There are no other turns for processes $1,\ldots,n$ in T.

To finish the construction of T we include one *RM* turn associated with each stabilization as follows. The location of the i^{th} *RM* turn in T depends on what occurs near the i^{th} stabilization. Consider the case that at stabilization, the state of the *RM* represented in the C area and the value of S_{jk} (if relevant) indicate that a response should be sent. Then the position of the i^{th} *RM* turn in T is given by the position of the turn in T' at which the $n+2^{nd}$ flag bit of R_{jk+1} becomes 1 (i.e., as soon as the simulated-memory-state of the j^{th} cell reflects the response). It follows

from Lemma 2.5, that all $2n+1$ flag bits are set to 1 and at most one becomes 0 (by an error) before process j reads R_{jk+1}.

If no response is to be sent but the decision is 1, the *RM* turn in T is associated with the turn in T' at which the first live simulator sets the last of the $2n+1$ flag bits of S_{jk} to 1. If the decision is 0, the *RM* turn in T is associated with the first turn in T' (before the stabilization) at which a live simulator starts step 3 of the managing part.

T has infinitely many *RM* turns and process turns, i.e., $T \in \mathcal{T}$.

Theorem 2.2. Let $\mathcal{T}_{sim}$ and $\mathcal{T}$ as above and let $T' = t'_0 t'_1 \cdots \in \mathcal{T}_{sim}$. The essential-simulated-history of T' equals the essential-history of some $T \in \mathcal{T}$.

Proof. Let $T = t_0 t_1 \cdots$ be as above. Since $T \in \mathcal{T}$, it suffices to show that the essential-simulated-history of T' equals the essential-history of T.

Some of the turns t'_m in T' "correspond" to turns in T as described above. If t'_m occurs in T' no earlier than the turn of T' that corresponds to t_i (in T), but earlier than the turn that corresponds to t_{i+1}, then t'_m *is preceded by* t_i. To prove Theorem 2.2, the following lemma is used.

Lemma 2.6. (1) For every $m \in \mathbb{N}$ where t'_m is preceded by t_i:

(a) The simulated-process-state of process j $(1 \le j \le n)$ after t'_m is the same as the state of the process in P after t_i.

(b) The simulated-memory-state of cell j $(1 \le j \le n)$ after t'_m is the same as the state of the cell in P after t_i.

(2) The *RM* state after the l^{th} "$n+1$" in T is the same as the *RM* state represented in $C[l \pmod 3]+1$ at the $l+1^{st}$ stabilization.

Lemma 2.6 suffices to prove Theorem 2.2. From (1) the only turns at which the essential-simulated-state of T' changes correspond to turns of T. Furthermore the essential-simulated-state of T' changes in the same way as the essential-state of T.

Proof of Lemma 2.6.

The proof is by induction. To prove (1), we assume (1) for all turns before t'_m and (2) for stabilizations before t'_m. If t'_m is a stabilization, then (2) is proved for t'_m assuming (1) and (2) before t'_m.

By construction, the only turns at which the essential-simulated-state in T' changes correspond to turns in T (this follows from the discussion after Lemma 2.5). To prove Lemma 2.6, it suffices to analyze turns of T' at which the essential-simulated-state changes, and turns of T' with which an *RM* turn in T is associated. At all other turns, (1) follows from the correctness of (1) at the previous turn, since the essential-simulated-state does not change, and the notion of the "preceding turn of T" does not change.

If t'_m is an error transition, and the essential-simulated-state changes, let t_i be the turn in T associated with t'_m. If the cell that changed was a response cell, then t'_m is associated with a stabilization (i.e., t_i is an *RM* turn). By induction on (2) and the fact that there is one *RM* turn associated with each stabilization, the state of the *RM* used to determine the response equals the state of the *RM* before t_i in T. Furthermore, in T, the state of the j^{th} cell before t equals the simulated-memory-state of cell j before t'_m in T' (by induction on (1)(b)). The request value used by the simulators to determine their response must have been this simulated-memory-state. This follows from Lemma 2.5; the simulators use the

request area most recently used by process j. Since the simulators used the correct request and RM state, they send the response that the RM would have sent in T. Thus after t'_m, the simulated-memory-state of the j^{th} cell equals the state of the j^{th} cell in P after t_i, proving (1)(b). Condition (1)(a) is trivial.

If at t'_m an S_{jk} area obtains $n+2$ flag bits equal to 1, then it is associated with a process turn in T. In that case, (1)(a) follows from the correctness of the simulated-process-state before t_i and the use of the correct response by process j in determining how to change state (Lemma 2.5). Condition (1)(b) also follows from the correctness of the process-state and the use of the correct response. Note that each conceptual message bit of R_{jk} has at most n of $2n+1$ bits with the "wrong" value, for the same reason that the flag bits have the correct "majority".

If $t'_m \in \{1,...,n\}$ and the $n+2^{nd}$ flag bit of S_{jk} or R_{jk} is set to 1 at t'_m, the proof is similar to the error transition case. The only other case is when t'_m is the first of $2n+1$ turns in which the flag bits of R_{jk} are voted on and the majority is determined to be 0. Then, the simulated-process-state of process j changes and t'_m is associated with an element t_i, of T. Since the majority is determined to be 0, $n+2$ flag bits of the R_{jk} are not yet 1. Thus the RM turn at which the state of the j^{th} cell changes has not yet occurred in T. Therefore, in T, the state of process j changes at t_i based on L_j being equal to REQUEST and the lemma is satisfied.

If $t'_m \in \{n+1,...,2n\}$ and the $n+2^{nd}$ flag bit of R_{jk} or S_{jk} is set to 1 at t'_m, the proof is similar to the error transition case. Otherwise, the essential-simulated-state of Q does not change at t'_m. It must be shown that the essential-state of P is not altered if t'_m has an associated turn t in P.

If t_i arises from a stabilization that had decision 0, t'_m is the start of step 3 of the synchronizing part by the "fastest" simulator. Thus, t'_m precedes the turns in which simulators vote on flag bits of S_{jk}. Since the decision is 0, it is not yet the case that n+2 flag bits of S_{jk} equal 1. If that many flag bits were equal to 1, the decision would be 1. Thus before t_i, L_j=RESPONSE in P (by induction on (1)(b)) and the essential-state does not change at t_i.

If t'_m is associated with a stabilization that had decision 1 but no response then t'_m occurred after the turn in T' that is associated with a certain S_{jk} becoming current. The simulators used the correct RM state (by (2)) and value of S_{jk} to determine not to respond. Thus the "n+1" in T does not change the state of any cell.

Condition (2) is verified similarly; using induction, the progress lemma, and the fact that the simulators use the correct request area to determine the next RM state. □

To finish proving Theorem 2.1, it must be shown that every essential-history in P from $\mathcal{T}$ equals an essential-simulated-history from $\mathcal{T}_{sim}$. This is an easy exercise and is left for the reader. The basic idea is that if in a turns history T of P, process j has a turn, then in a turns history T' for Q, process j is given enough turns either to determine that the majority of the flag bits are 0, or to update the simulated-memory-state if the majority is 1. If a turn of T is an "n+1", enough simulator turns are given in T' until the system stabilizes and a response is sent (if relevant).

From this discussion and Theorem 2.2, Theorem 2.1 follows immediately. □

Next we define a system of n processes, Q', that faithfully partially simulates P. Let Q' have the same set of cells as Q. Assume that each process alternates between acting as a process and simulator, on a step-by-step basis. Then Q' faithfully partially simulates P. If process j has infinitely many turns as a process, then it has infinitely many turns as a simulator. Any history of Q' may be thought of as a history of Q and by Theorem 2.2 is mapped to a history of P. However, not all histories of P are represented. It follows from this:

Theorem 2.3. The system Q' faithfully partially simulates P. □

Using the definition of being enabled to enter the critical section [17] alluded to above, it follows from Theorem 2.3 that:

Theorem 2.4. There is a process system of n processes which satisfies mutual exclusion, is deadlock free, and satisfies bounded waiting, and tolerates the failure of any one memory cell and the death of any n-1 processes (as long as no dying process is enabled to enter the critical section). □

The system Q does not simulate P according to the more restricted way that simulated-memory-states and simulated-process-states change for systems without errors. In Q, a change in simulated-process-state of process j is not associated with a single state of PROC(Q,j).

The following minor changes to Q produce an error-free system, Q'', that faithfully simulates P. There is one flag bit for S_{jk} and n+1 flag bits for R_{jk}. The simulated-memory-state changes when *all* flag bits of a new area

become 1. The $n+1$ flag bits in R_{jk} are set to 1 by the simulators and process j does not set any bits in R_{jk} to 1. The single flag bit in S_{jk} is set only by process j.

Process j, and the process-simulator-function are as follows. Let $p \in \text{PROC}(P,j)$. There is a state (or actually three states as with Q), $q \in \text{PROC}(Q'',j)$ with $g_j(q)=p$. There are n additional states used to read successive flag bits of R_{jk}, and each of these states are mapped to p. In the above $n+1$ states if the flag bit read is 1, process j enters the next state and reads the next bit. If the bit read is 0, process j enters a state q_1, where $g_j(q_1)=p_1$ and p_1 is the next state process j enters from p in P if L_j=REQUEST. If all of the flag bits are 1 process j reads R_{jk}, sets the flag bits of R_{jk+1} and S_{jk+1} to 0, and writes in S_{jk}. The value written in S_{jk} depends on the state of process j and what was read in R_{jk}. At the turn when process j sets the flag bit in S_{jk}, it enters a state q_2, where $g_j(q_2)=p_2$ and p_2 is the state that follows p if the value of L_j is RESPONSE and the value of M_j is the value just read in R_{jk}. Using this construction:

Theorem 2.5 The system Q'' faithfully simulates (without errors) P with respect to the histories $\mathcal{T}$ and $\mathcal{T}'$ defined in Section 2.2.4. □

Finally, we count the time and space required by Q. Let $z_1=|\text{PROC}(P,n+1)|$ and let $z_2=|\text{CELL}(P,j)|$ (assume $|\text{CELL}(P,j)|$ is the same for all j). The number of cells in each C area is $O(n^2+n\log(z_1))$. Each of $3n$ R_{jk} areas use $O(n\log(z_2))$ cells. The total space is $O(n^2\log(z_2)+n\log(z_1))$.

It is not clear how to count time but we briefly discuss the time requirement. Error transitions are not counted. Process j requires $O(n)$ turns

to determine that a response has not been sent. Process j requires $O(n\log(z_2))$ turns to determine that a response has been sent and to send the next request. For a simulator to execute every turn in the managing part of a C area requires $O(n^2+n\log(z_2)+n\log(z_1))$ turns. Between each of these $O(n^2+n\log(z_1)+n\log(z_2))$ turns, there are $O(n)$ turns in the synchronizing part. In addition, the C area is read at least once in step 4 of the synchronizing part, but this is low order. The total number of turns, for one simulator to do one *RM* turn is thus $O(n^3+n^2\log(z_1)+n^2\log(z_2))$. If n simulators execute at about the same speed then $O(n^4+n^3\log(z_1)+n^3\log(z_2))$ are required for one *R* turn. This may be reduced by having the synchronizing part executed only between every two *writing* transitions, but this does not save very much.

2.6. Discussion of unreliability properties

This section describes types of unreliability and which types are tolerated by our simulation.

2.6.1 Process unreliability

Announced process death may be handled by our simulation techniques. The only constraint is that death in the *RM* model cannot be announced while it is the *RM*'s turn to respond. If the *RM* model is augmented with special "dead" bits for each process, this constraint is avoided. Thus fair mutual exclusion with announced death may be accomplished. If less general and more efficient solutions are desired, the works of [34,37,52,53] should be consulted.

If process death is undetectable, the process cannot be eliminated from its critical section as it is indistinguishable from a slow process. Our solution keeps the rest of the system operational if there are multiple resources.

Other failures may exist if a "clock process" is used. Synchronizing the speeds of clocks is studied in [38], and we do not elaborate.

A final type of unreliability is where a dead process modifies memory in a manner that is inconsistent with its protocol. There are no known solutions for this, and it seems hard to define what a solution would look like.

2.6.2 Memory unreliability

Before discussing different types of memory unreliability we give a detailed description of the properties of our simulation.

Consider the request cells, that are written only by process *j*. Our

solution adapts to tolerate m faults out of $2m+1$. Thus the proportion of faulty bits that may be tolerated can be arbitrarily close to fifty percent.

Consider all of the cells (other than the MSG-SENT-VOTE cells) that are written by the simulators. These consist of almost all of the C cells, the response cells and the request flag cells. Here, one conceptual bit is represented by $2n+1$ bits to handle $n-1$ processes that may be out-of-date and one faulty bit. To tolerate m errors, one needs $2(n+m)-1$ bits, for m faulty bits, and $n-1$ out of date processes. Again, the proportion of faulty bits may approach fifty percent, but the cost in memory size is higher.

The most stringent assumptions relate to the MSG-SENT-VOTE cells. Only one cell out of approximately $8n^2$ cells may fail. Extending this to tolerate greater unreliability is inefficient. To tolerate m errors, there may be $2n+m$ bits that disagree with the initial decisions of all simulators (see the proof of Lemma 2.5). In order for the simulators to agree on a decision, one needs at least $2n+m+1$ blocks of $4n+2m+1$ cells. Thus to tolerate m faults, one requires approximately $2(2n+m)^2$ cells.

Because of this quadratic growth, the proportion of failures tolerable does not approach fifty percent. The best proportion is at $m=2n$ where for $2n$ faults one needs only $32n^2$ cells. While it seems that slight improvements of our techniques may increase the constant, to improve to $o(1/n)$ apparently will require new techniques. We discuss one improvement in the constant factor at the end of this section.

One degree of fault tolerance is to tolerate m faults in a system (Theorem 2.1 as stated in Section 2.2.4.2). Our solution trivially satisfies this notion. If each "conceptual" bit is protected against m faults, then the

entire system is protected against m faults. Unfortunately, to tolerate m faults one needs much memory.

A second criterion for correctness is that only m percent of certain sets of cells fail. For $m<50$, all of the cells satisfy this except for the MSG-SENT-VOTE cells. The MSG-SENT-VOTE cells only satisfy this criterion if $(m/100)<(1/16n)$. Tolerating m percent of different groups of cells permits the partition of memory into $100/m$ memory "units", where any single unit may fail. Our solution tolerates the death of any one of $16n$ units. If our model can be applied to a distributed system, one can tolerate the death of all communications links to a site if $100/m$ is as small as n.

A third correctness criterion is that only m percent of all the bits in the system fail. Here, we do not succeed even for a moderately sized value of m. If there are r conceptual bits in the system, then even if $100/r$ percent of them fail, an entire conceptual bit might fail, ruining the system. Since in our solution r is large (it grows as a function of the size of the *RM* and message areas) our solution does not solve this problem. It is unlikely that any solution can be developed for a large value of m if the size of each cell is bounded. One needs many conceptual bits to code the information in the message areas.

Fault-tolerance is one means of producing a system that with high probability has *no* observable errors. If each memory cell has a probability x of failing per unit time then for each set of r bits, the expected number of failures is xr. If we use the solution that tolerates $2n$ MSG-SENT-VOTE failures out of $32n^2$ cells, the expected number of faulty MSG-SENT-VOTE bits is $32xn^2$, which is asymptotically greater than the $2n$ failures which are

tolerable. The fault-tolerance thus does not guarantee (probabilistically) reliability for a large number of processes, but we believe that it is a good solution for small values of n.

2.6.3. An improvement in the memory reliability

We discuss a modification in the algorithm which allows greater unreliability for the MSG-SENT-VOTE cells. In the voting procedure, there are $2n+1$ blocks of $4n+1$ MSG-SENT-VOTE cells that are used. Each simulator sets each cell in a block based on the perceived majority of the previous block. Each cell is set only if it is not of the correct "type". One needs $4n+1$ cells per block to tolerate $2n$ "incorrect" cells, at most n that started with the correct type and were not set at all and at most n that were reset by outdated simulators after they were set correctly.

We discuss these two classes of incorrect states of cells. Assume that a (nonfaulty) cell is not set by live simulators because it started with the correct type. Then this cell was set by an outdated simulator (simulator j) during or after the *last* usage of C_i. Simulator j therefore does not set any other cells incorrectly with the correct type. (Simulator j may set cells to the incorrect type if simulator j believes that the simulation is up to the turn that last used C_i.) Thus there are at most n incorrect cells of the correct type. (There are also at most n cells of the incorrect type.)

An improvement in the reliability may be obtained by ignoring the cells of the incorrect type in the voting and to use $2n+1$ blocks of $3n+1$ cells each. If all simulators have the same initial decision, then at most n cells of the correct type may have states that differ from all initial decisions, and there are at least $2n+1$ cells of the correct type. Thus if no initial decision is 0

(1), then 0 (1) does not win any vote. Note that $2n+1$ blocks are needed, as otherwise, there may be one incorrect cell in each block and the simulators may not agree at any block.

The revised algorithm tolerates one faulty cell out of approximately $6n^2$. If there are more faulty cells, this technique is less useful as each faulty MSG-SENT-VOTE cell may be of either type. To tolerate m faulty cells one needs $2n+m+1$ blocks with $3n+2m+1$ cells per block. If $m=n\sqrt{3}$, one needs $n^2\sqrt{3}$ $(2+\sqrt{3})^2$ cells for a ratio of $1/n(7+4\sqrt{3})$, or approximately $1/13.9n$.

2.7 Open Problems and Further Work

The simulation of an *arbitrary* manager suffers from the same drawbacks as other general results; it does not take advantage of the simplifications special to specific problems. There are important, identifiable, subproblems of a general *RM* simulation which are of inherent interest. One such problem is to develop fair mutual exclusion protocols with bounded-sized faulty cells and dying processes. Alternatively, one might prove that *any* such fair mutual exclusion protocol requires a large number of cells or a large amount of time. Similarly, a lower bound on the space or time of *RM* simulation would be of interest. Techniques which may be applicable are those of [4,5,17].

The next problem is to obtain an efficient *RM* simulation with an error free system. Our simulation technique permits processes to improperly modify a limited amount of memory. The resulting memory inconsistencies are handled as if they were faults. For that reason, our simulation by a system without errors is not much simpler than if the simulating system had errors.

Another class of open problems is to obtain mutual exclusion protocols without test-and-sets. With unannounced process death this has not been solved even with reliable memory and many-valued cells. (Note that our protocols use the test-and-set only in step 4 of the managing part, where the simulators agree on a decision.) If one cannot eliminate the need for test-and-sets, it would be interesting to develop a protocol where test-and-sets are applied only to binary variables.

A number of questions relate to the details of the model of Section 2.2, and the simulation of Section 2.4. The definition of simulation is a "global" definition. It would be easier to deal with a local characterization of one system simulating another; something to the effect that Q simulates P if

whenever a fixed sequence of turns occurs in Q it corresponds to a fixed sequence of turns in P.

A less important question also relates to the definition of simulation. If Q is to faithfully simulate P, where Q may have errors, then the determination of the simulated-process-state of a process is by a powerful partition function (described in Section 2.2.3.) A definition of simulation which uses less powerful methods to determine the simulated-process-states is desired.

The *RM* systems are one class of process systems that model resource allocation. One may prefer a different notion of a manager, e.g., a manager that uses one variable to communicate with all processes. Alternatively, one may permit a process to change a request before it is responded to. One question is to understand how sensitive the simulation is to the properties of the managed system. Is there some way of classifying process systems that can or cannot be simulated? A related question is to compare different notions of managed systems, to determine if some simulate others, or if some are more powerful than others in some sense.

The main technical question is to improve the reliability of the simulation. Even if many cells are used, the proportion of cells that can be allowed to fail in the MSG-SENT-VOTE section is at most $1/14n$. When n gets large, one expects to get $O(n^2)$ errors, but can only tolerate $O(n)$ of them.

The final comment is on the "stable" state proof technique. This is an ad-hoc technique which seems appropriate here. We need a better understanding of parallel programs to simplify proofs of this type. It would also be interesting if the "stable" state technique has wider applicability.

3. Scheduling

3.1 Introduction

Chapter 3 of this thesis discusses job scheduling. The basic problem in job scheduling is that one is given a set of jobs or tasks to be executed on a given set of processors. There are a wide variety of variations of such problems. Two important examples that we discuss in detail are that certain jobs may or may not have to be executed before other jobs, and the processors may be identical or have different capabilities. For current surveys on scheduling theory we refer the reader to [7,24].

The problem of job scheduling on processors of different speeds was introduced by Liu and Liu [42,43]. They studied the scheduling of a partially ordered set of jobs (i.e. some jobs precede others) on a uniform process system. In a uniform process system, the ratio of the speeds between any two processors is independent of the task to be executed.

They studied a class of schedules known as demand driven or list schedules. The characteristic property of these schedules is that at no time is there both an idle processor and an unexecuted executable task. They showed that any list schedule has a finishing time that is at most $1+(b_1/b_m)-(b_1/(b_1+...+b_m))$ times worse than optimal where b_i is the speed of the i^{th} fastest processor (the optimal schedule is the one with least finishing time). In addition, examples of list schedules were presented which do perform as poorly as the bound. This is a discouraging result since a large gap between the speeds of the fastest and slowest processors implies the ineffectiveness of list scheduling, independent of the speeds of the other processors or the number of processors. List scheduling has been a prototype of approximation algorithms since its introduction in the identical processor

case [23].

One way of avoiding this problem of unboundedly bad behavior is to use preemptive scheduling. When one is allowed to use preemptive scheduling, one is allowed to temporarily suspend the execution of an executing task, and restart it at a later time. Ordinarily, one assumes that preemption is a cost-free operation. Horvath, Lam, and Sethi studied a "level algorithm" for the preemptive scheduling of partially ordered tasks [28] which generalizes the algorithms of [47,48]. It is shown in [28], that the level algorithm has worst case performance between $\sqrt{1.5m}$ and $\sqrt{m/8}$.

The decision problem of determining whether a given set of tasks (even without a partial order) can be scheduled on processors of different speeds within a given finishing time is NP-complete [27]. As a result it seems unlikely that an algorithm can be found which runs in polynomial time and always produces the optimal schedule [20]. It is for this reason that we try to develop algorithms that approximate the optimal solution as we proceed to explain.

Sections 3.2.2 and 3.2.3 present algorithms for nonpreemptive and preemptive scheduling of a partially ordered set of tasks on uniform process systems. The performance bounds are better than those of [42,43] for the nonpreemptive case and better than those of [28] for the preemptive case. In fact, the bound proved for the algorithm of Section 3.2.2 (the nonpreemptive case) is better than the $\sqrt{1.5m}$ bound obtained by Horvath, Lam, and Sethi for the preemptive case (ordinarily, approximation algorithms are closer to optimal in the preemptive case, than in the nonpreemptive case [24]).

The focus of Section 3.2.2 is to provide a nonpreemptive algorithm which is guaranteed to be no worse than $O(\sqrt{m})$ times worse than optimal,

regardless of the speeds of the processors. While $\sqrt{m}$ and b_1/b_m (the leading term in the bound on list schedules) are incomparable in that either may be smaller for any particular set of processor speeds, the natural way that the algorithm is developed guarantees that the worst case performance for any fixed set of processor speeds is not worse than $1+b_1/b_m$.

The basic strategy of the heuristic is to use only the fastest i processors for an appropriately chosen value of i. In contrast to other algorithms that rule out the use of certain processors [21], this algorithm is unique in that the processors may be ruled out before the set of tasks is examined. In particular, the algorithm does not need any information about the time requirement of a task before scheduling the task.

Formal definitions are provided in Section 3.2.1. Section 3.2.2 describes which processors are to be used if $O(\sqrt{m})$ behavior is desired. A bound of $\sqrt{m} + O(m^{1/4})$ is obtained on the performance of the heuristic. Also, for small values of m the exact worst case performance of the algorithm is computed. This is significant as $O(m^{1/4})$ is potentially a dominating factor for small values of m. The time complexity of the algorithm is $O(m+n^2)$ where n is the number of tasks.

Section 3.2.3 analyzes preemptive scheduling. An improved upper bound is obtained on the performance of the level algorithm of Horvath, Lam, and Sethi [28]. This is accomplished by noticing that the $O(\sqrt{m})$ behavior of the level algorithm comes automatically with the use of preemption - and not from the particular way in which the level algorithm schedules tasks. Specifically, Section 3.2.3 analyzes a class of schedules, called the maximal usage schedules, that includes all "reasonable" preemptive schedules. Any preemptive

schedule may be transformed into a maximal usage schedule in polynomial time, where the new, maximal usage schedule has a finishing time at least as small as that of the original schedule. The main result of Section 3.2.3 is that any maximal usage schedule is at most $\sqrt{m} + (1/2)$ times worse than optimal.

Section 3.2.4 indicates that the nonpreemptive algorithm of Section 3.2.2 is essentially best possible among algorithms using a certain restricted class of heuristics. It is shown that any algorithm that does not look at the time requirement of a task before scheduling has worst case performance which is at least $\sqrt{m}$ times worse than optimal. Thus the algorithm of Section 3.2.2 is asymptotically best possible. This result also proves that one of the algorithms of Section 3.3 (which does not use the time requirement as we proceed to discuss) is best possible up to a constant factor.

While the uniform assumption is relevant for certain systems, it is not necessarily relevant for others. In Section 3.3 we study a number of algorithms for the scheduling of a set of n independent tasks on m *unrelated* processors. The processors are unrelated in the sense that there is no notion of a fast processor always requiring less time than a slow processor, irrespective of the task being executed. Rather, the time required for the execution of a task on a processor is a function of both the task and the processor. This models the situation that general purpose processors have specialized capabilities that permit them to execute certain tasks more efficiently than others. An example of this might be in a distributed system where the time requirement of a task on a processor may depend on communication costs.

Polynomial time approximation algorithms for such sets of tasks were first studied by Ibarra and Kim in [30]. Five algorithms are presented in [30], each of which is guaranteed to be at most m times worse than optimal in the worst case. In addition, four of the five are exactly m times worse than optimal in the worst case. The fifth algorithm was left as an open problem - its effectiveness was shown to be between 2 and m times worse than optimal. (In Section 3.3.6 it is shown that this algorithm is at least $1+\log_2(m)$ times worse than optimal in the worst case. Thus the gap left in [30] is somewhat tightened, but is still left open.)

The first new algorithm that we present is at most $2.5\sqrt{m}$ times worse than optimal in the worst case. Thus it may not be as good as the fifth algorithm of [30], but it is provably better than the other four, and may in fact be better than all five. The running time of this algorithm is $O(mn\log(n))$. We also show that there are examples for which the algorithm is as bad as $2\sqrt{m}$ times worse than optimal, indicating that the analysis is tight up to a factor of 1.25. This algorithm is somewhat similar to the one of Section 3.2.2 in that processors do not execute tasks for which they are very inefficient. This algorithm does not use the absolute time requirements of the jobs that are to be executed. It does however, use information about the realtive efficiencies of the processors on the tasks. This is discussed in greater detail in Section 3.3.2. The time complexity of this algorithm is $O(mn\log(n))$.

The second new algorithm is a modification of the first, which adds a largest processing time (LPT) heuristic. This algorithm is at most $(1+\sqrt{2})\sqrt{m}$ times worse than optimal and also runs in polynomial time. The worst example known for this heuristic is $\sqrt{m}$ times worse than optimal, and we believe that

this heuristic is actually more of an improvement over the original algorithm than the worst case bound suggests. The LPT heuristic has been a useful heuristic in situations where the processors are identical [23].

The third algorithm is a different modification of the first with a substantially longer running time. Whereas the first two require polynomial time in terms of both the number of tasks and the number of processors, this one requires exponential time in terms of the number of processors. The worst case behavior is at most $1.5\sqrt{m}$ times worse than optimal. When assigning tasks on a bounded number of processors, however, the algorithm runs in polynomial time. Horowitz and Sahni [27,54] devise algorithms of time complexity $O(n^{2m}/\epsilon)$ whose worst cases are within $1+\epsilon$ of optimal. Our algorithm requires time $O(m^m+mn\log(n))$. Thus its running time is less sensitive to large numbers of tasks executed on moderate numbers of processors.

All three of these algorithms may be varied in trivial ways to get performance bounds which are better than the above bounds by a small constant factor. While we are able to obtain slightly better performance bounds with these modifications, we do not believe that the resulting algorithms are actually any better.

One final result that we present is an additional algorithm which is m times worse than optimal in the worst case. This algorithm has the useful property that when extended to an algorithm for scheduling partially ordered tasks on unrelated processors, it is still at most m times worse than optimal.

Recently, the scheduling of systems with different types of processors - dedicated to different types of tasks has been studied [22,44]. Examples of systems where this is relevant include data flow models of computation [12,33]

where primitive operations are computed by different processors. Similarly, in machines such as the CDC6600, there are several specialized functional modules [59]. Also, in a system where I/O tasks and arithmetic tasks are handled by different processor units, such an assumption may be relevant. In Section 3.4 we analyze some of the properties of schedules for systems with different *types* of tasks.

The complexity of determining the optimal schedule is *NP*-complete in very simple cases. It is shown in [22], that the problem of determining whether a schedule exists for a given typed task system that requires fewer than a given number of steps is *NP*-complete even if there are only two processors, one of each type. Also, if the number of types of processors varies, the problem is *NP*-complete even if the precedence constraint is restricted to being a forest. The techniques used there are adaptations of those found in [7,60].

Section 3.4.2 discusses extensions to the results of Graham [18], which provide general bounds for non-preemptive list scheduling strategies which satisfy fundamental "no-waste" requirements. In ordinary task systems with identical processors, any list schedule is at most $2-(1/m)$ times worse than optimal where m is the number of processors. For typed task systems with equally fast processors a similar bound is obtained. Any list schedule is at most $k+1-(1/\max(m_1,\ldots,m_k))$ times worse than optimal, where k is the number of types of tasks and m_i is the number of processors of type i. This bound is achievable for any value of k and any values of $m_1,\ldots,m_k$. These same results were obtained independently in [44].

The results of [42,43] which provide general bounds for list schedules on machines with processors of different speeds are also extended. It is shown

in Section 3.4.3 that the bound for typed task systems is (approximately) $k+\max(b_{11}/b_{1m_1},\ldots,b_{k1}/b_{km_k})$ where b_{ij} is the speed of the j^{th} fastest processor of type i. Finally, the results of Sections 3.2.2 and 3.2.3 are extended to give an algorithm for the preemptive and nonpreemptive scheduling of typed task systems on processors of different speeds. The worst case performance of this algorithm is bounded by an expression which is in terms of the number of processors of each type and is independent of the speeds of the processors.

The results of Section 3.3 were obtained jointly with Ernest Davis [10].

3.2 Scheduling tasks on processors of uniformly different speeds

3.2.1 Basic Definitions and Models

A *task system* $(\mathcal{T},<,\mu)$ consists of:

(1) A set $\mathcal{T}$ of n *tasks*.

(2) A *partial ordering* $<$ on $\mathcal{T}$.

(3) A *time function* $\mu:\mathcal{T}\rightarrow\mathbb{R}^+$.

The set $\mathcal{T}$ represents the set of tasks or jobs that need to be executed. The partial ordering specifies which tasks must be executed before other tasks. The value $\mu(T)$ is the *time requirement* of the task T.

The total number of steps required by all the tasks of $\mathcal{T}$ will be denoted $\mu(\mathcal{T})$, i.e., $\mu(\mathcal{T})=\Sigma_{T\in\mathcal{T}}\ \mu(T)$.

Associated with a task system is a set of processors $\mathcal{P}=\{P_i:1\leq i\leq m\}$. There is a *rate* b_i associated with P_i $(b_1\geq b_2\geq\ldots\geq b_m>0)$. If a task T is assigned to a processor P with rate b, then $\mu(T)/b$ time units are required for the processing of T on P. (When discussing a generic processor "P", the associated rate is taken to be "b".)

A *nonpreemptive schedule* for $(\mathcal{T},<,\mu)$ *on a set of processors* $\mathcal{P}$ *with rates* $b_1,\ldots,b_m$ is a total function $S:\mathcal{T}\rightarrow\mathbb{R}\times\mathcal{P}$ satisfying conditions (1) and (2) below. If $S(T)=(t,P)$ then the *starting time* of task T is t, the *finishing time* of T is $t+(\mu(T)/b)$ and T is *being executed on* P for times x such that $t\leq x<t+(\mu(T)/b)$.

The function S satisfies:

(1) For all $t\in\mathbb{R}^+$, if two tasks are both being executed at time t, then they are being executed on different processors at time t.

(2) For $T,T'\in\mathcal{T}$, if $T<T'$ the starting time of T' is no less than the finishing time of T.

Condition (1) asserts that processor capabilities may not be exceeded. Condition (2) forces the obedience of precedence constraints.

The *finishing time* of a schedule is the maximum finishing time of the set of tasks. An *optimal schedule* is any schedule that minimizes the finishing time. For two schedules S and S', with finishing times w and w' the *performance ratio of S to S'* is w/w'.

A *preemptive schedule* for $(\mathcal{T},<,\mu)$ is a total function S that maps each task $T\in\mathcal{T}$ to a finite set of interval, processor pairs, i.e., the first element of each pair is an interval and the second is a processor. If $S(T)=\{([i_1,j_1],Q_1),([i_2,j_2],Q_2),\ldots,([i_l,j_l],Q_l)\}$ then

(a) $i_p,j_p\in\mathbb{R}$ for $p=1,\ldots,l$.

(b) $i_p\leq j_p$ for $p=1,\ldots,l$ and $j_p\leq i_{p+1}$ for $p=1,\ldots,l-1$

(c) $Q_p\in\mathcal{P}$ for $p=1,\ldots,l$.

For $i_p\leq t\leq j_p$ T is *being executed on processor* Q_p at time t. The time i_1 is the *starting time* of T, and the time j_l is the *finishing time* of T.

A *valid* preemptive schedule for $(\mathcal{T},<,\mu)$ *on a set of processors* $\mathcal{P}=\{P_i:1\leq i\leq m\}$ is a preemptive schedule for $(\mathcal{T},<,\mu)$ that satisfies conditions (1) and (2) above (for nonpreemptive schedules) and:

(3) For $T\in\mathcal{T}$ (with $S(T)$ as above), $\mu(T)=(j_1-i_1)rate(Q_1)+\ldots+(j_l-i_l)rate(Q_l)$ where if $Q_i=P_j$ then $rate(Q_i)=b_j$.

Condition (3) asserts that each task is processed exactly long enough to complete its time requirement.

The notions of *finishing time*, *optimal preemptive schedule*, and *performance ratio* are analogous to the same notions for nonpreemptive schedules.

A task T is *executable* at time t if all predecessors of T have been finished by time t, but T has not yet been completed.

To analyze some of the schedules considered in Section 3.2 it will be useful to have the following definitions. A *chain* C is a sequence of tasks $C=(T_1,\ldots,T_l)$ with $T_i\in\mathcal{T}$ such that for all j, $1\leq j<l$, $T_j<T_{j+1}$. C *starts* with task T_1. The *length* of C is $\Sigma_{j=1}^{l}\mu(T_j)$. The *height of a task* $T\in\mathcal{T}$ is the length of the longest chain starting at T. The *height of* $(\mathcal{T},<,\mu)$ is the length of the longest chain starting at any task $T\in\mathcal{T}$. The height of $(\mathcal{T},<,\mu)$ will be denoted h.

While the notion of the height of a task is a static notion which is a property of $(\mathcal{T},<,\mu)$, we also associate a dynamic notion of the height of a task with any schedule for $(\mathcal{T},<,\mu)$. Specifically, let S be a schedule for $(\mathcal{T},<,\mu)$, and let t be less than the finishing time of S. Then the *height of the task* T *at time* t is equal to the length of the longest chain starting at T, where the length of the chain considers only the unexecuted time requirements. Similarly, the *height of* $(\mathcal{T},<,\mu)$ *at time* t is the length of the longest chain starting at any task (not yet completed) $T\in\mathcal{T}$. Note that if a portion of a task has been finished at time t, then it contributes to the height the proportion of the time requirement not yet completed.

It is convenient to analyze schedules based on whether or not the height is decreasing during a given interval of time. One may plot the height of $(\mathcal{T},<,\mu)$ as a function of time for a given schedule S and make the following observation. The function is a nonincreasing function which starts at the original height of $(\mathcal{T},<,\mu)$ for t=0, and ends at height 0 for t=finishing time of S. If during an interval of time, the height was a monotonically decreasing function of time then that interval is called a

height reducing interval. If during an interval the height is constant the interval is called a *constant height interval.* Any schedule may be completely partitioned into portions executed during height reducing intervals, and portions executed during constant height intervals.

One more notation is needed. Define $B_i=\Sigma_{j=1}^{i} b_j$. Thus B_i is the *total processing power of the fastest i processors.* B_m is the total processing power of all the processors, and $B_1=b_1$.

Section 3.2.2 analyzes the nonpreemptive scheduling of tasks on processors of different speeds. Section 3.2.3 handles preemptive scheduling.

3.2.2 Nonpreemptive scheduling of tasks on uniform processors

3.2.2.1 List schedules on the fastest i processors

In Section 3.2.2, we devise an algorithm whose performance is at most $\sqrt{m} + O(m^{1/4})$ times worse than optimal, for the nonpreemptive scheduling of partially ordered tasks on processors of different speeds. The algorithm is related to a class of schedules that has attracted much interest, the list scheduling model [17]. List schedules are designed to avoid the apparently wasteful behavior of letting a processor be idle while there are executable tasks.

A list schedule uses a (priority) *list* L which is a permutation of the tasks of $\mathcal{T}$, i.e., $L=(T_1,...,T_n)$ ($T_i \in \mathcal{T}$ and for $i \neq j$, $T_i \neq T_j$). The *list schedule* for $(\mathcal{T},<,\mu)$ with the list L is defined as follows. At each point in time that at least one processor completes a task, each processor that is not still executing a task is assigned an unexecuted executable task. The tasks are chosen by giving higher priority to those unexecuted tasks with the lowest indices in L. If r processors are simultaneously available ($r>1$), then the r highest priority unexecuted, executable tasks are assigned to the available processors. The decision as to which processor gets which task is made arbitrarily (or one may choose to assign higher priority tasks to faster processors). Only if not enough unexecuted tasks are executable, do processors remain idle. Note that any schedule that is unwasteful in the sense that processors are never permitted to be idle unless no free tasks are available can be formulated as a list schedule.

The motivation for this heuristic comes from several sources. The primary motivation emanates from the optimality of some list schedule in the case that the processors are identical and the tasks are all unit execution

time tasks. It can be shown that when each task requires an equal amount of time at least one list schedule is an optimal schedule. Other sources of interest include the fact that it is simple to implement, and due to its simplicity, it is a good starting place for building other heuristics.

When the processors are of different speeds, list schedules may be as bad as $1+(b_1/b_m)-(b_1/(b_1+...+b_m))$ times worse than optimal [42,43]. The reason for this "unboundedly" bad behavior (as a function of the number of processors) is that an extremely slow processor may bottleneck the entire system by spending a large amount of time on a task. This motivates the following class of heuristics. A *list schedule on the fastest i processors* has a priority list as above. The difference in the execution strategy is that the slowest $m-i$ processors are never used, and tasks are scheduled as if the only processors available were the fastest i processors.

3.2.2.2 Performance bound for list schedules on the fastest i processors

The first portion of this section entails an analysis of the worst case performance of list schedules on the fastest i processors. Given a set of speeds $b_1,...,b_m$, and given i, a bound will be obtained in terms of the parameters b_j's and i. The second portion of this section analyzes this bound more carefully, and indicates that for each set of processor speeds, an easily determined value of i causes the performance ratio to be no worse than $1+2\sqrt{m}$ times worse than optimal. A more complicated analysis then shows that in fact some value of i causes a ratio of worse than $\sqrt{m} + O(m^{1/4})$. The final portion of this section provides examples which indicate that the performance bound is the correct order of magnitude. In particular, for certain sets of speeds, our heuristic and a class of related heuristics are as bad as $\sqrt{m-1}$ times

worse than optimal.

It is easy to get a speed independent bound for a simple scheduling algorithm, but this bound is not very good. It is not hard to see that if only the fastest processor is used, and it is always used, that the resulting schedule is no worse than m times worse than optimal. This possibility is actually alluded to in [42,43]. The bound of this section (which discusses a natural generalization of using only the fastest processor) is substantially better than this.

The approach to be used is to obtain two lower bounds, LB_1 and LB_2, on the finishing time of the optimal schedule for a given task system. Then an upper bound, UB, is obtained on the finishing time on any list schedule. The ratio $(UB/\max(LB_1,LB_2))$ is then an upper bound on the performance ratio of the schedule relative to optimal.

Lemma 3.1. Let $(\mathcal{T},<,\mu)$ be a task system to be scheduled on processors of different speeds. Let w_{opt} be the finishing time of an optimal schedule. Then $w_{opt} \geq \max(\mu(\mathcal{T})/B_m, h/b_1)$.

Proof. The most that any schedule can process in unit time is B_m units of the time requirement of the system. It is thus follows immediately that $w_{opt} \geq \mu(\mathcal{T})/B_m$.

A chain of length h requires at least time h/b_1 to be processed, even if the fastest processor is always used on the chain. It follows that $w_{opt} \geq h/b_1$. Combining these two bounds we obtain $w_{opt} \geq \max(\mu(\mathcal{T})/B_m, h/b_1)$ □

Lemma 3.2. Let $(\mathcal{T},<,\mu)$ be as in Lemma 3.1. Let w_i be the finishing time of a list schedule on the fastest i processors. Then $w_i \leq (\mu(\mathcal{T})/B_i)+(h/b_i)$.

Proof. To analyze the effectiveness of any list schedule on the fastest i processors, it is convenient to break up the schedule of interest into height reducing intervals and constant height intervals. The sum of the total duration of these intervals equals w_i.

Consider any constant height interval. Throughout the interval all of the fastest i processors are in use. We will prove this by contradiction. Let time t be a time within a constant height interval when fewer than i processors are in use. Consider the set of unfinished tasks that are at maximum height at time t. By definition, each of these tasks is executable (i.e. has no unfinished predecessors). Since not all of the fastest i processors are in use, and the schedule is a list schedule on the fastest i processors, it must be that all of these maximum height tasks are being executed. But then, it follows that the height of the task system in this interval is being reduced, contradicting the fact that this is a constant height interval.

By the above remarks, it follows that during each constant height interval, the processors are processing at least B_i units of the time requirement of the task system per unit time. Thus the total time spent on constant height intervals is at most $\mu(\mathcal{T})/B_i$.

Next, examine the height reducing intervals. At each point in time, some of the tasks being executed are at the maximum height. These as well as all other tasks being executed are being processed at the rate of at least b_i. Since the total height may be reduced by at most h throughout the schedule, it follows that the total amount of time spent on height reducing intervals is at most h/b_i. Together with the above bound on the amount of time in a constant height interval, one may conclude that $w_i \leq (\mu(\mathcal{T})/B_i)+(h/b_i)$.

Actually it is easily shown that $w_i \leq ((\mu(\mathcal{T})-h)/B_i)+(h/b_i)$, but this does not

substantially improve the performance bound. This improvement follows from the fact that at least h units of the time requirement are executed during height reducing intervals leaving only $\mu(\mathcal{T})$-h for constant height intervals. It will be important to remember this improved bound for some numerical results in Section 3.2.2.3. □

It follows from Lemmas 3.1 and 3.2 that:

$$(1) \qquad \frac{w_i}{w_{opt}} \leq \frac{(\mu(\mathcal{T})/B_i)+(h/b_i)}{\max(\mu(\mathcal{T})/B_m, h/b_1)}$$

Equation (1) presents us with an opportunity to formally state the schedule that will be used. Given a task system $(\mathcal{T},<,\mu)$, determine the total time requirement of all tasks $(\mu(\mathcal{T}))$, and the height of the system (h). Compute the right hand side of equation (1) for each value of $i=1,...,m$. The value of i that minimizes the expression is the number of processors that will be used. Devise any list schedule on the fastest i processors.

In Section 3.2.2.3 it will be shown that the performance ratio of the above schedule (relative to optimal) is at most $\sqrt{m} + O(m^{1/4})$. Before proceeding to the proof of that fact, a modification in the scheduling algorithm will be suggested. The strategy as stated involves doing a separate calculation for each task system in order to determine how many processors to use. In fact, to get the $\sqrt{m} + O(m^{1/4})$ behavior, it is possible to use the same number of processors independent of the task system (based only on the b_j's). Note that equation (1) also implies:

$$(2)\qquad \frac{w_i}{w_{opt}} \le \frac{(\mu(\mathcal{T})/B_i)}{(\mu(\mathcal{T})/B_m)} + \frac{(h/b_i)}{(h/b_1)} = \frac{B_m}{B_i} + \frac{b_1}{b_i}$$

The scheduling strategy is to use a list schedule on the fastest i processors where i minimizes this last performance bound. In Section 3.2.2.3 it is shown that any resulting schedule is not worse than $\sqrt{m} + O(m^{1/4})$ times worse than optimal irrespective of the value of the b_j's.

Note that if the bound used on w_i is $w_i \le ((\mu(\mathcal{T})-h)/B_i)+(h/b_i)$ then one may obtain a bound on the algorithm of:

$$(3)\qquad \frac{w_i}{w_{opt}} \le \frac{B_m}{B_i} + \frac{b_1}{b_i} - \frac{b_1}{B_i}$$

Thus, an alternative scheduling algorithm chooses i on the basis of equation (3). This improved algorithm does not have better asymptotic behavior, but does provide a better algorithm for small values of m. This will be discussed further when numerical results are discussed. For asymptotic bounds we use the algorithm generated by the simpler equation (2).

The right hand side of equation (2) will be denoted $E_i(b)$.
The right hand side of equation (3) will be denoted $E'_i(b)$.

Note that the bound of equation (3) with $i=m$ is the performance bound of Liu and Liu [42,43]. For this reason, the choice of the value of i as the one which *minimizes* $E'_i(b)$ provides a bound which is at least as good as their bound for any fixed set of processors. The derivation presented here is

similar to the derivation of their bound.

To analyze the running time of the algorithm, note that choosing the number of processors to use (on the basis of (2) or (3)) requires time $O(m)$; $O(m)$ time to compute $B_1, B_2, \ldots, B_m$, $O(m)$ to compute $E_i(b)$ for each value of i, and $O(m)$ to minimize $E_i(b)$.

The actual scheduling requires time $O(n^2)$ as follows. For each task T, a number $pred(T)$ is maintained which represents the number of tasks that preceed T that are not yet completed. The values $pred(T)$ for $T \in \mathcal{T}$ are initialized in $O(n^2)$ time using the adjacency matrix for $<$. When a task T is completed, then if $T < T'$ the value $pred(T')$ is decremented. It requires time $O(n)$ to update the *pred* function each time that a task is completed. If $pred(T')$ is set to 0 then T' is added to a list of executable tasks. The processor that had been executing the completed task T is then added to a list of idle processors. The actual assigning of the tasks given these two lists can be done in constant time. Thus the total time complexity is the $O(n^2)$ time required to do the initialization and to update the *pred* function after the execution of each task. This is a list schedule as no processors are idle while there are executable tasks.

Note that we have not described how to determine which of m currently executing tasks is finished next. This may be done in $\log_2(m)$ time if a list of "currently executing" tasks is maintained in order of their finish time. When a task is scheduled it is inserted into this list based on its finishing time. This must be done at most n times throughout the schedule. For $m < n$, $O(n\log(m))$ is a low order term. For $n < m$, the insertion requires only $\log(n)$ time, and $O(n\log(n))$ is a low order term.

3.2.2.3 A speed independent bound

This section analyzes equation (2) in three different ways. The first way provides an intuitive indication of which processors to use as a function of their relative speeds. Specifically, if the processors used are those whose rates are within a factor of $\sqrt{m}$ of the fastest processor, then list schedules on these processors are at most $1+2\sqrt{m}$ times worse than optimal. The second approach proves that one may always choose i such that the bound is $\sqrt{m} + O(m^{1/4})$. This complicated proof does not give any intuitive idea how to choose i in general, other than calculating $E_i(b)$ for each value of i and then minimizing. The third method of analysis is a calculation of the actual numerical bounds on the algorithm for small values of m when the correct choice of i is made. These bounds are better than $1+2\sqrt{m}$ and are more exact than a bound with an $O(m^{1/4})$ term.

Theorem 3.1. Consider a set of m processors of different speeds. Then some value of i $(1 \le i \le m)$ has the property that for any task system (with optimal finishing time w_{opt}), and any list schedule on the fastest i processors for that task system (with finishing time w_i) $w_i/w_{opt} \le 1+2\sqrt{m}$.

Proof. Recall that $w_j/w_{opt} \le (B_m/B_j)+(b_1/b_j)$. Choose i such that $\sqrt{m}\,b_i \ge b_1$ and $\sqrt{m}\,b_{i+1} < b_1$. Certainly some j satisfies $\sqrt{m}\,b_j \ge b_1$ (since b_1 satisfies it) and if no j satisfies $\sqrt{m}\,b_{j+1} < b_1$, then choose $i=m$. From equation (2) and the choice of i:

$$(4) \qquad \frac{w_i}{w_{opt}} \leq 1 + \frac{b_{i+1}+\ldots+b_m}{B_i} + \frac{\sqrt{m}\, b_i}{b_i}$$

This follows from breaking up B_m into $B_i+b_{i+1}+\ldots+b_m$ and using the upper bound on b_1.

Now clearly $B_i \geq b_1$. Also using $\sqrt{m}\, b_j < b_1$ for $j \geq i+1$ (4) may be modified to obtain:

$$(5) \qquad \frac{w_i}{w_{opt}} \leq 1 + \frac{(m-i)(b_1/\sqrt{m})}{b_1} + \sqrt{m}$$

Since $(m-i) \leq m$ one may conclude that $w_i/w_{opt} \leq 1+2\sqrt{m}$. □

This bound may be improved somewhat for the choice of i considered in the proof. We skip the details and proceed to give a more complicated proof which improves the bound of Theorem 3.1 by choosing i in a more intelligent manner.

Theorem 3.2. Consider a set of m processors of different speeds. Then some value of i $(1 \leq i \leq m)$ has the property that for any task system (with optimal finishing time w_{opt}), and any list schedule on the fastest i processors for that task system (with finishing time w_i) $w_i/w_{opt} \leq \sqrt{m} + O(m^{1/4})$.

Proof. The value of i that is chosen is the one that minimizes $E_i(b)$. Let $f(m)$ be the supremum of $\min_i E_i(b)$ where the sup is taken over $b_1 \geq b_2 \geq \ldots \geq b_m > 0$. It will be shown that $f(m)$ is actually achieved by a particular set of speeds $b_1,\ldots,b_m$. Also, these speeds have the property that for every i $f(m)=E_i(b)$. Using this fact one may conclude that $f(m) \leq \sqrt{m} + O(m^{1/4})$.

Define $B \subset \mathbb{R}^m$ by $B=\{(b_1,\ldots,b_m) \in \mathbb{R}^m : b_1 \geq b_2 \geq \ldots \geq b_m \geq 0 \text{ and } B_m=1\}$. Note that B consists of every legal set of processor speeds (normalized to sum to 1), and some illegal sets (e.g. $b_m=0$). For $b \in B$, with $b=(b_1,\ldots,b_m)$, define $g(b)=\min_i E_i(b)$. (Note that $g(b) \neq \infty$ since $E_1(b)=1+(1/b_1)<\infty$.) If $b_m \neq 0$, then $g(b)$ is a bound on the heuristic with processor speeds $b_1,\ldots,b_m$. Since B is compact and g is continuous, g attains a maximum at some particular point $b^*=(b^*_1,\ldots,b^*_m) \in B$. Note that $g(b^*) \geq f(m)$. Also, to show $g(b^*)=f(m)$ it suffices to show that $b^*_m \neq 0$.

It must be that $g(b^*)=E_m(b^*)$. To prove this, assume $g(b^*) \neq E_m(b^*)$ and define b' by $b'_i=b^*_i/(1+\epsilon)$ and $b'_m=(b^*_m+\epsilon)/(1+\epsilon)$. Note that $b' \in B$ (if $b^*_m \neq b^*_{m-1}$), $E_i(b')>E_i(b^*)$ for $i<m$, and $E_m(b')<E_m(b^*)$. Since $E_m(b^*)$ was not the smallest E_i value for b^*, for sufficiently small ϵ, $g(b^*)<E_m(b')$. Since $E_i(b')>g(b^*)$ for every i, $g(b')>g(b^*)$, contradicting the fact that b^* maximizes g. (If $b^*_m=b^*_{m-1}=\ldots=b^*_{i+1}<b^*_i$, then b' as defined is not in B. In that case, one defines $b'_j=(b^*_j+(\epsilon/(m-i)))/(1+\epsilon)$ for $j=i+1,\ldots,m$ in order to preserve the decreasing nature of the vector b'. A similar proof then follows.)

To prove that for every i $E_i(b^*)$ is the same, it suffices to consider the case that $g(b^*)=E_m(b^*)=\ldots=E_{k+1}(b^*)<E_k(b^*)$. We will define a sequence of vectors b',b'',b''' so that each successive vector has the various E_i values

at least as large as $g(b^*)$ but agreeing with g at one less value of i. That is $E_{k+1}(b')>g(b^*)$, $E_{k+2}(b'')>g(b^*)$, etc. Finally, we will obtain $E_m(b)>g(b^*)$ for some vector b with either $g(b)>g(b^*)$ or $g(b)=g(b^*)$. In the first case it contradicts the fact that b^* maximizes g. In the second case it contradicts $g(b^*)=E_m(b^*)$ since b then maximizes g and $g(b)\neq E_m(b)$.

The vector b' is defined by $b'_i=b^*_i$ for $i\neq k,k+1$, $b'_k=b^*_k+\epsilon$ and $b'_{k+1}=b^*_{k+1}-\epsilon$. It is easy to verify that both $E_{k+1}(b')$ and $E_k(b')$ exceed $g(b^*)$ for small ϵ. Also $E_i(b')=E_i(b^*)$ for $i\neq k,k+1$. We now show $b'\epsilon B$. From $E_{k+2}(b^*)=E_{k+1}(b^*)$ we have $b^*_{k+1}>b^*_{k+2}$. Thus $b'_{k+1}>b'_{k+2}$ for small ϵ. However, $b^*_k=b^*_{k-1}$ is possible which would imply $b'\notin B$. In that case, the ϵ is not added to b^*_k, but rather a total of ϵ is added to the processors whose rates equal b^*_k. The vectors b'', b''', ... are defined in a similar manner. The final vector has $E_m(b)>g(b^*)$. Since $g(b^*)=E_m(b^*)$, the vector b cannot maximize g. But $E_i(b)\geq g(b^*)$ for every $i=1,...,m$. This contradicts the fact that b^* maximizes g.

It follows from the fact that $E_i(b^*)=g(b^*)$ for every i that $b^*_m\neq 0$ and thus $g(b^*)=f(m)$.

The bound of $\sqrt{m}+O(m^{1/4})$ will be proved by using the fact that $f=f(m)=E_i(b^*)$ for each value of i. Using $f=E_i(b^*)$ and $f=E_1(b^*)$ one may conclude that $b^*_i=B^*_i/(f-1)(fB^*_i-1)$ since $b^*_i=b^*_1B^*_i/(fB^*_i-1)$ and $b^*_1=1/(f-1)$. This in turn proves that $b^*_i=(1/f(f-1))(1+(1/(fB^*_i-1)))$. Thus for $j>i$ $b^*_j\leq(1+(1/(fB^*_i-1)))/f(f-1)$. From this it follows that:

(6) $b^*_{i+1}+\ldots+b^*_m=(1-B^*_i)\leq(1+\epsilon)(m-i)/f(f-1)$ where $\epsilon=(1/(fB_i^*-1))$.

In order to use the above equation to get a bound on f, a few other facts are needed. Note that $f>\sqrt{m}$. This can be shown by considering the vector b' which is a normalized version of the vector $(\sqrt{m-1},1,1,\ldots,1)$ since $g(b')>\sqrt{m}$. Note also that $b^*_1\leq\sqrt{1/m}$. This follows from the fact that $E_1(b^*)=E_m(b^*)$ which implies that $b^{*2}_1=b^*_m$. Since $mb^*_m\leq 1$ the upper bound on b^*_1 follows. Thus for $j=1,\ldots,m$ $b^*_j\leq b^*_1<(1/\sqrt{m})$ and the successive sums $B^*_1,B^*_2,\ldots,B^*_m$ are spaced apart by a distance of at most $\sqrt{1/m}$. Thus, for some value of i $B^*_i=rm^{-1/4}$ for some r between $\sqrt{2}$ and $1+\sqrt{2}$ (for sufficiently large m).

Let $B^*_i=rm^{-1/4}$. Then $1+B^*_i\geq 1+\epsilon$ using the expression for ϵ in equation (6). This follows from the fact that $(B^*_i/\epsilon)=(fB^{*2}_i-B^*_i)\geq(\sqrt{m}\,r^2m^{-1/2}-B^*_i)=(r^2-B^*_i)$ using $f>\sqrt{m}$ and $B^*_i=rm^{-1/4}$. Since $r^2\geq 2$ and $B^*_i\leq 1$, $(B^*_i/\epsilon)\geq(r^2-B^*_i)\geq(2-1)=1$ and thus $B^*_i\geq\epsilon$. Using (6) and $1+B^*_i\geq 1+\epsilon$ we have $f(f-1)\leq(m-i)(1+B^*_i)/(1-B^*_i)\leq(m-i)(1+rm^{-1/4})(1+2rm^{-1/4})$ for sufficiently large values of m (the last inequality is obtained by expanding $1/(1-B^*_i)$). But then (by further increasing the right hand side) $(f-1)^2\leq m+4rm^{3/4}+4r^2\sqrt{m}$ which yields $f\leq\sqrt{m}+2rm^{1/4}+1$. □

While Theorem 3.2 provides a better asymptotic estimate of the performance of the algorithm than Theorem 3.1, it does not give a better bound for practical situations. In principle the $O(m^{1/4})$ term may be the dominating factor for the small values of m that typically arise in practice. For that reason, it is important to get a more meaningful bound for small values of m. A third way of evaluating the heuristic is thus presented, which

gives numerical bounds on the algorithm for small values of m. This also will give an intuitive idea as to the growth rate of $f(m)$.

Recall that the heuristic takes its worst value at the vector b^* with the property that $E_i(b^*)$ is the same for every i. From $E_1(b^*)=E_m(b^*)$ with $B^*_m=1$, we get $b^*_m=b^*_1{}^2$. From $E_i(b^*)=E_1(b^*)$ we get for $1<i<m$,

$$(7) \qquad b^*_i = \frac{b^*_1{}^2 B^*_i}{B^*_i(b^*_1+1)-b^*_1}.$$

Using $B^*_i=1-(b^*_{i+1}+\ldots+b^*_m)$, we note that the above equation is in terms of b^*_1 and $b^*_{i+1},\ldots,b^*_m$. Hence each b^*_i can be determined inductively as a function of b^*_1. Using $1=b^*_1+\ldots+b^*_m$, we can solve for b^*_1. Solving this equation and computing $1+(1/b^*_1)$ gives a bound on the algorithm. It is not hard to show that there is a unique solution to this equation subject to $b_1\geq b_2\geq\ldots\geq b_m>0$.

This calculation was done on the MACSYMA system which generated the expressions to solve and also solved them. The indication of this small sample of data is that $\sqrt{m}$ + O(log m) might in fact be an accurate bound. The value $f(m)$ for the range of values considered seems to be bounded by $\sqrt{m}$ + .21($\log_2 m$) + 1. In fact, not only is $f(m)$ bounded by this expression (in the range we considered), but it seems to grow slower. The results are given in Table 3.1, together with other key quantities for the sake of comparison.

m	$f(m)$	$\sqrt{m}$	$\sqrt{m} + m^{1/4}$	$1+2\sqrt{m}$	$(\sqrt{m} + .21\log_2 m + 1$
2	2.62	1.41	2.60	3.82	2.62
3	3.06	1.73	3.05	4.46	3.06
4	3.41	2	3.41	5	3.42
5	3.71	2.24	3.74	5.48	3.73
6	3.98	2.45	4.02	5.90	3.99
7	4.22	2.65	4.28	6.30	4.24
8	4.44	2.83	4.51	6.66	4.46
9	4.64	3	4.73	7	4.67
10	4.83	3.16	4.94	7.32	4.86
50	9.14	7.07	9.73	15.14	9.26
100	12.24	10	13.16	21	12.40
500	24.98	22.36	27.09	45.72	25.24
1000	34.41	31.62	37.25	64.25	34.71
5000	73.88	70.71	79.12	142.42	74.29
10000	103.33	100	110	201	103.79

Table 3.1.

Note that $f(m)$ does seem to be growing faster than $\sqrt{m} + O(1)$ although this can not be proven by such numerical studies.

The above results were obtained using the bound of equation (2). An important purpose of these results is to show how $f(m)$ behaves. An additional reason for this calculational exercise, though, is to get as good a bound as possible on the heuristic for small values of m. For the purpose of getting a tight bound on the algorithm for small values of m, a better bound is obtained if the algorithm uses the slightly more complicated bound given by equation

(3). In our analytic studies the b_1/B_i term was ignored since it does not improve the asymptotic results (in particular it is always less than 1). Nevertheless, for small values of m it is a significant portion of the bound. The next table gives a bound on the algorithm in terms of this better bound. (Incidentally, the same technique was used for generating Table 3.2 as was used for generating Table 3.1. To use this technique, it must first be shown that a bound on the algorithm is obtained by analyzing the vector $b^* \in B$ which has the property that $E'_i(b^*)$ is the same for each value of i. This is a simple exercise using the technique of the proof of Theorem 3.2.)

Notice that using this better bound gives a result which is about .7 or .8 better than the bound of equation (2) - a substantial saving for small m. For large values of m the improvement is slightly smaller, and less significant due to the large value of either bound.

m	bound on the algorithm
2	1.75
3	2.25
4	2.65
5	2.97
6	3.25
7	3.50
8	3.73
9	3.94
10	4.14

Table 3.2

Intuitively it seems quite wasteful *never* to use processors - no matter how slow they may be. It is an open question to determine how to use the slow processors in order to provide a quantitatively better performance ratio. There are certain simple safe techniques that one may use which do not harm the performance ratio. For example, one may first determine a list schedule on the fastest i processors. Then, if a slow processor is free at a particular time, and an executable task is not being executed, and furthermore the finishing time of the task will be later (with the current schedule) than the time that our slow processor could finish it, then it is safe to assign the task to the slow processor (and possibly to make other improvements based on the earlier finishing time of this task).

The fact that this procedure is not harmful may be easily seen. Since the finishing time of the chosen task is earlier in the new schedule than in the original schedule, no task needs to be finished later than in the original schedule. While it is easy to determine such safe uses for the slow processors, we have been unable to determine any methods that guarantee faster behavior.

3.2.2.4. Achievability of the performance bound

In this section it is demonstrated that the results of Section 3.2.2.3 are asymptotically correct. This is shown by demonstrating that for a certain set of processor speeds and a specific task system, the performance ratio of a list schedule on the fastest i processors (for *any* $i=1,\ldots,m$) may be as large as $\sqrt{m-1}$. The fact that this example shows that any choice of i has the potential of being $\sqrt{m-1}$ times worse than optimal is significant. It tells us that no sophisticated way of choosing i provides better than $\sqrt{m}$

behavior if once i is chosen a list schedule is the only added feature of the heuristic.

Consider the situation where $b_1=\sqrt{m-1}$ and $b_i=1$ for $i>1$. Consider the task system of $2n$ tasks as diagrammed in Figure 3.1. A node represents a task and an arrow represents a precedence dependence. The time requirement of each of the n tasks in the long chain is $\sqrt{m-1}$. The time requirement of the other n tasks is $m-1$. An asymptotically optimal schedule proceeds as follows. P_1 executes every task in the long chain. Each task in the long chain requires unit time on P_1. Meanwhile, $P_2,\ldots,P_m$ execute the tasks that are not in the long chain. Each of these processors requires time $m-1$ for one of the tasks. If $n=m-1$ then the long chain requires time $m-1$, but P_m will not finish its task until $2m-3$ units of time have passed since its task is not executable until $m-2$ units of time elapse. For any value of n the finishing time is similarly bounded by $n+m-2$.

To discuss the fact that this task system may be executed inefficiently, no matter how many processors are used, consider two situations. The first is the case that one attempts to schedule the system on the fastest processor alone. The second is the situation that the processing is done on i processors for any $i>1$.

If only the fastest processor is used, then there is not enough processing power to execute this task system efficiently. Specifically, the total amount of time requirement of this task system is $n(m-1+\sqrt{m-1})$. With only $\sqrt{m-1}$ processing power available, the finishing time must be at least $n(1+\sqrt{m-1})$. For large values of n, this provides a performance ratio of approximately $1+\sqrt{m-1}$ times worse than optimal.

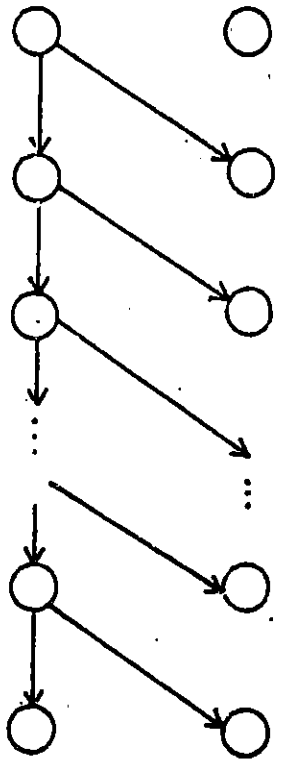


Figure 3.2.1

Consider the scheduling of this system on i processors for $i>1$. A "bad" list schedule uses P_1 on the "non-long chain" tasks, and P_2 on the chain tasks. After time $\sqrt{m-1}$, P_1 finishes the first non-chain element, and P_2 finishes the first chain element. Repeating this strategy for each pair of tasks requires time $\sqrt{m-1}$ for each pair. Thus the total time for the bad schedule is about $n\sqrt{m-1}$ and the ratio between the finishing times of the "bad" schedule and the optimal schedule approaches $\sqrt{m-1}$ for large n. Note that no matter how many processors one attempts to use, a bad list schedule only allows two processors to be used. □

The fact that $m-1$ of the m processors have the same speed in this example is important. In [21], it is shown that with independent tasks and almost identical processors (in the sense of $m-1$ of them being identical) improved algorithms may be obtained.

Recall that the upper bound on the algorithm was $\sqrt{m} + O(m^{1/4})$. It has not been shown that this is the best upper bound on the algorithm. However, whatever the numerical value of $f(m)=\max_b (g(b))$ is, a bound on the heuristic of Section 3.2.2.1 may be obtained in terms of $f(m)$. Consider a set of m unordered tasks, the i^{th} having time requirement b^*_i. The optimal schedule requires unit time. Using only the fastest processor (a valid choice with the algorithm as presented) requires time $1/b^*_1$. But $f(m)-1=1/b^*_1$. Thus $f(m)$ is almost achievable (whatever $f(m)$ is). This means that an exact bound on the algorithm may be obtained by solving the mathematical problem of determining $f(m)$, without any need to look at more task systems.

Consider the related heuristic of trying to minimize $E'_i(b^*)=(1/B^*_i)+(b^*_1/b^*_i)-(b^*_1/B^*_i)$. In that case $E'_1(b^*)=1/b^*_1$. Recall that at the vector b^* that maximizes $\min_i E'_i(b^*)$, $E'_i(b^*)$ is the same for every i, and using

only the fastest processor is a valid choice. In that case, $1/b_1^*$ for this vector b^* is an exact upper and lower bound.

To give a concrete example, consider m=2. $E'_1(b)=1/b_1$ and $E'_2(b)=1+(b_1/b_2)-b_1$. Solving $E'_1(b)=E'_2(b)$ provides b_1=.57 and b_2=.43. Let $\mu(T_1)$=.57, $\mu(T_2)$=.43 and <=empty. Then the optimal schedule requires unit time. The algorithm presented here may choose to use only the fastest processor and require time 1/.57=1.75. This agrees with the bound predicted by solving the maximization problem.

3.2.3 Preemptive Scheduling of Tasks on Uniform Processors

3.2.3.1 Maximal Usage Schedules

Section 3.2.3 analyzes preemptive scheduling of tasks on processors of different speeds. We study a large class of preemptive schedules, called the *maximal usage preemptive schedules*. As in Section 3.2.2, the goal is to obtain bounds on algorithms that improve known bounds. Whereas Section 3.2.2 described a new algorithm whose performance was better than known algorithms, that is not the case here. The worst case performance of the maximal usage schedules is no better than the worst case performance of known algorithms such as the level algorithm of [28]. However, our analysis is tighter than previously done analyses. Thus, for example, our analysis of maximal usage schedules provides a tighter bound on the level algorithm than was proved in [28].

A *maximal usage preemptive schedule* is a valid preemptive schedule which satisfies the following two additional requirements.

(1) Whenever i tasks are executable, then $\min(m,i)$ tasks are being executed.

(2) Whenever i processors are being used, the fastest i processors are in use.

Maximal usage schedules encompass all preemptive schedules in a very strong sense. Given any task system, at least one optimal preemptive schedule is a maximal usage schedule. Furthermore, given any preemptive schedule S (with finishing time w), one may easily transform S into a maximal usage schedule S' (with finishing time w') with the property $w' \leq w$. We proceed to describe this transformation algorithm.

Assume that S is given as a listing $U_1,\ldots,U_r$ where each

listing element U_k=([$start_k$,end_k],T_k,Q_k). The interpretation of U_k is that task T_k is executed on Q_k between times $start_k$ and end_k.

The translation from an arbitrary schedule into a maximal usage schedule will proceed in stages. It starts with the schedule S=S_0, and produces schedules S_1,...,S_q where S_q is the resulting maximal usage schedule. The basic idea is that if S_i satisfies conditions (1) and (2) above before t, but does not at t, then S_i is modified at t to produce S_{i+1}. Specifically, let t' be the smallest time greater than t that equals $start_k$ for some k. Then, if S_i does not use all m processors at t and there are executable tasks that are not being executed at t, then an executable task is assigned to an idle processor for the interval $[t,t']$. Assignment of tasks to processors for this interval continues until there are either no more executable tasks or no more idle processors. Similarly, if the processors being used at t are not the fastest processors, then the tasks that are currently being executed on the slower processors are reassigned to the faster processors.

Next, S_i is further modified to insure that the total time requirement of each task T executed in S_{i+1} is $\mu(T)$. This is necessary to insure that S_{i+1} is in fact a valid schedule. For example, if a listing element $([t,t'],T,P)$ is added to S_{i+1} and the speed of P is b, then $(t'-t)b$ units of the time requirement of T are removed from later assignments. Similarly, if T is rescheduled onto a faster processor at t, then the added time requirement which is completed is removed from a later interval of T. If executing T on P during $[t,t']$ causes the total time requirement of T that is assigned to processors before t' to exceed $\mu(T)$, then the interval $[t,t']$ is shortened somewhat.

After the reassignments are made, the translation algorithm determines

the smallest time t'' greater than t and equal to end_k for some k. Note that $t'' \leq t'$. For if any tasks are added in the above transformation, then they are scheduled to end at or before t'. On the other hand, if no tasks are added, then it must be that some other task ends between t and t'. For assume that no other task ended in this interval. Then, consider a task, T, that starts at t'. Since there are no new tasks completed between t and t', and T is executable at t', T is executable at t. Since no new processors became free between t and t', the fact that T is started at t' implies that there is a free processor at t. Thus T would have been rescheduled to start at t in the above translation contrary to the hypothesis.

Note that S_{i+1}, attains maximal usage before t'' and that portion of the schedule will not be changed again. Starting at t'', the procedure modifies S_{i+1} to produce S_{i+2} in the manner specified above.

The resulting finishing time of each task in S_q is at least as small as in S. From this it follows that the finishing time of S_q is at least as small as that of S.

While the above translation idea is relatively straightforward, the bookkeeping required for the transformation is rather detailed. The following data structures are helpful in the modification of S to the maximal usage schedule S_q.

We assume that the list S is sorted by the $start_k$ coordinate (i.e. $start_1 \leq start_2 \leq \ldots \leq start_r$) and stored as a linked list. If the transformation algorithm is handling time t it maintains an array, *remain*, of n numbers which represents the portion of the time requirement of each task not completed by the time t. The value $pred(T)$ represents the number of predecessors of T that are not yet completed (at time t). For each task $T \epsilon \mathcal{T}$, the sublist of the

current version of the schedule (i.e. one of the S_i) which has T as the task component is kept separately. This is kept sorted by the end_k value and is linked (with pointers) to the list S_i. A linked list, *ex*, of currently executable tasks is maintained.

The *remain* array is initialized to μ. The *pred* array and the *ex* array may be initialized from the adjacency matrix of $<$. This requires time $O(n^2)$. The n sublists of S (for each task) may be constructed by one pass through S. (Note that each sublist is sorted simultaneously by the $start_k$ and end_k coordinates.)

The algorithm will modify S_i by adding listing elements to S_i, and removing listing elements from S_i. The resulting listing is what we have been calling S_{i+1}.

Transformation Algorithm

1. t=0.

2. Let t'=min$\{x:x=start_k$ and $x>t\}$.

3. Let c = the number of tasks that are being executed at time t in S_i (i.e., the current version of the schedule).

4. If $c=m$ go to 11.

5. If $c<m$, let d = min(m,the size of the *ex* list).

6. If $c=d$ and the fastest d processors are in use go to 11.

7. If $c=d$ but not all of the fastest d processors are in use go to 10.

8. If $c<d$, choose the first $d-c$ tasks from the list *ex* that are currently not being executed to be rescheduled.

a. Assign each of these tasks (T) to one of the fastest d processors that are not already in use (P) as follows.

i. If $remain(T)/b \geq t'-t$, then insert a listing element to S_i, namely, $([t,t'],T,P)$. Also add this to the list of intervals for the task T.

ii. If $remain(T)/b < t'-t$, then insert a listing element to S_i, namely, $([t,t+(remain(T)/b)],T,P)$. Also add this to the list of intervals for the task T.

b. If T is assigned to P by 8.a.i, then eliminate $(t'-t)b$ units of time from existing intervals for T. Specifically, remove entire intervals of T that finished latest in S_i if possible, and/or reduce the ending time of the latest interval not removed, until $(t'-t)b$ units of the time requirement of T are removed. If T is assigned to P by 8.a.ii, then remove all remaining intervals associated with T from S_i.

9. If the fastest d processors are in use at t with the current schedule go to 11.

10. For each task, T, being executed on one of the slowest $m-d$ processors, P reassign it to one of the unassigned fastest d processors, P', for the interval $[t,t']$. If T was originally assigned to an interval with ending time $t_f > t'$, then divide this interval into two portions $[t,t']$ and $[t',t_f]$. In the first interval T is assigned to P' and in the second interval T is assigned to P. To compensate for the additional time requirement finished in $[t,t']$, remove later intervals of T as in 8.b. The exceptions to this are if as a result of assigning this task to P', we must finish the entire task before t_f or before t'. Then this interval is handled as those of 8.a.ii, i.e., all other remaining intervals associated with it are removed from S_i and the intervals defined here are shortened.

11. Let $t'' = \min\{x : x = end_k \text{ and } x > t\}$.

12. For each task T executed in the interval $[t,t'']$, reduce $remain(T)$ by $(t''-t)b$ if T is executed on processor P. Also, change the list element $([t,t_f],T,P)$ to two listings: $([t,t''],T,P)$ and $([t'',t_f],T,P)$ if $t_f>t''$.

13. For each task T, for which $remain(T)$ became 0 in Step 12, update the $pred$ values of all the successors of T. Add each task whose $pred$ value becomes 0 to the ex list. Remove from the ex list all tasks that are completed at t''.

14. If the ex list is empty then end. Otherwise let $t=t''$ and go to 2.

There are a number of facts that must be verified about the transformation algorithm. It must be shown that the algorithm terminates. It must be shown that the final schedule, S_q, is a schedule and is in fact a maximal usage schedule. It must be shown that the finishing time of S_q is no larger than the finishing time of S. Also, we will analyze the run-time of the algorithm.

To show termination, it will be shown that the main loop (2-14) is executed at most $n+2r$ times. In order to count the number of iterations, we explore a number of properties of the ending times and starting times of the intervals (i.e. the listing elements) of S_q.

There are at most r distinct starting times in the original schedule S. In addition, a small number of different starting times may appear in intervals of S_q. Fix an iteration that analyzes intervals that start at t. Then at this iteration, even if t is not one of the original starting times, t will become a starting time when and if tasks are assigned to start at t. Specifically, if any intervals are assigned in Steps 8, they are assigned to start at t. Also, in Step 10, intervals may be assigned to start either at t or at t' (the next

starting time). Note by definition t' is not a new starting time. Finally, in Step 12, intervals are assigned to start at the time that will be used as the t value in the next iteration. Thus at a given iteration, the only possible new starting times are the "t value" of that iteration and the "t value" of the next iteration.

Using this we will show that there are at most $2r+n$ possible ending times for tasks in the schedule S_q. There are at most r ending times in the original schedule S. Consider a new interval introduced into S_q. An interval introduced in 8.a.i. ends at t' where t' is the starting time of some interval. In fact, it must be the starting time of one of the original r intervals. This follows from the above analysis of starting times: The "new" starting times that arise while t is analyzed are only t itself (note that $t<t'$) or a starting time introduced in Step 12 (i.e., after the interval in 8.a.i. was already introduced). Thus, the new ending time t' must be the starting time of one of the original intervals. Similarly, intervals that end at t' in Step 10 arise from one of the r original starting times. Also, intervals that end at t_f in Step 10 or t'' in Step 12 do not produce new ending times.

It suffices to analyze intervals introduced in 8.a.ii, and 10, which complete the time requirement of a task. Clearly, there may be at most n of these in S_q. Thus S_q has at most $2r+n$ distinct ending times (i.e., the r original ones, the r original starting times, and the n finishing times of the tasks). It follows from this that there are at most $2r+n$ iterations of the main loop since each iteration completes at least one of these $2r+n$ ending times.

To verify that S_q is a schedule it must be shown that the total time assigned to each task completes its time requirement, that no processor is

assigned to more than one task at once, and no task is assigned to more than one processor at once. Note that a task is only assigned to a processor in an interval that the processor did not already have a task assigned to it and the task was also unassigned. Also, for each new interval added in S_{i+1}, the total time requirement "used up" during the new interval is removed from other intervals of the schedule S_i. Finally, it is easy to see that the precedence constraints are obeyed.

The fact that S_q is a maximal usage schedule follows directly from the algorithm. Assume that time t is the earliest time at which maximal usage is not attained in S_q. The time t must be the end of some interval. Thus time t is the start of some iteration, and if there were any unexecuted executable tasks at t, they would be assigned to processors in Step 8. Similarly, if the "wrong" processors were in use at t, the tasks would be reassigned in Step 10. This contradicts the assumption that maximal usage was not attained at t.

The fact that S_q has at most as large a finishing time as S follows from the fact that after each iteration the finishing time of each task is not increased.

Finally, the run time of the algorithm is analyzed. By the above count of the number of iterations and the fact that $n \leq r$, there are at most $O(r)$ iterations. Assume that at the iteration that analyzes intervals starting at time t, there is a pointer to the tasks that start at t in the current version of the schedule. Then at each iteration, Step 2 requires time $O(m)$ to determine the value of t' and save the pointer to the intervals starting at t' for later. Steps 3-7 similarly require at most time $O(m)$. (A bit array of size m is useful to determine which of the fastest d processors are in use in $O(m)$ time.)

In Step 8 of the algorithm, choosing the d-c tasks may be done in $O(m)$ time as follows. For each of the first d tasks on the *ex* list, one may determine if it is already being executed by checking the first unexecuted interval on the list for that task (assume that a pointer is maintained to that location). The rest of step 8.a may similarly be done in $O(m)$ time.

Step 8.b is slightly harder to analyze as potentially, it may require time $O(r)$ just for one iteration. Note however, that each of the $O(r)$ iterations introduces at most $O(m)$ new intervals each. Thus the total number of *removals* of intervals, during all iterations, for all tasks, is at most the total number of intervals that ever appear which is $O(mr)$. In addition, $O(m)$ intervals may be *reduced* in size at each iteration.

Step 9 takes time $O(m)$. In step 10, the reassignments require time $O(m)$, and the total number of removals is bounded by $O(mr)$ as with Step 8.b. Using the pointer in the schedule list at t', inserting the intervals that start at t' may similarly be done in time $O(m)$.

Step 11 requires time $O(m)$ as the only intervals that need be considered in the minimization are those that start at t. For assume that an interval starts at $start_k > t$, but finishes earlier than any of the intervals that start at t. This violates maximal usage, which we have shown above to be a property S_q. Similarly Step 12 requires time at most $O(m)$.

The algorithm may spend $O(n)$ time in Step 13 for each task completed. Thus for all n tasks, the algorithm requires time $O(n^2)$ for Step 13. Step 14 requires time $O(m)$ if there is a pointer from each task to its position in the *ex* list. Finally, to reset the pointer in the current listing of intervals to t'' requires time $O(m)$.

Using the above calculation, the total time requirement of the

algorithm is $O(n^2+mr)$. Step 13 and some initialization requires a total of $O(n^2)$ time. Removing intervals for all iterations requires time $O(mr)$. The rest of the algorithm requires at most $O(m)$ time per $O(r)$ iterations. Note that n^2+r+m is essentially the size of the input to the algorithm since the size of the partial order is $O(n^2)$.

We summarize the above discussion with the following:

Theorem 3.3. There is an algorithm that takes as input a valid preemptive schedule S for a task system $(\mathcal{T},<,\mu)$ on a set of processors $\mathcal{P}$ and produces as output a maximal usage schedule, S' for $(\mathcal{T},<,\mu)$ on $\mathcal{P}$. The finishing time of S' is less than or equal to the finishing time of S. Furthermore, the running time of the algorithm is $O(n^2+mr)$ where n is the number of tasks in $\mathcal{T}$, m the number of processors in $\mathcal{P}$, and r the number of intervals in S.

3.2.3.2 An upper bound related to the nonpreemptive bound

This section provides a bound on the maximal usage schedules by using the analysis techniques of Section 3.2.2. In Section 3.2.3.3 we use additional techniques to get a slightly better bound. Recall that B_i is the sum of the rates of the fastest i processors.

Recall equation (2) of Section 3.2.2. The equation states that if w_i is the finishing time of a list schedule on the fastest i processors and w_{opt} is the finishing time of the optimal nonpreemptive schedule, then $w_i/w_{opt} \leq (B_m/B_i) + (b_1/b_m)$. We will show that if w is the finishing time of an arbitrary maximal usage schedule and w_{opt} is the finishing time of the optimal preemptive schedule, then for every i $(1 \leq i \leq m)$, $w/w_{opt} \leq (B_m/B_i) + (b_1/b_m)$. From this, one may conclude that for any task system and any maximal usage

schedule for the task system $w/w_{opt} \leq \sqrt{m} + O(m^{1/4})$. This proof follows by applying the bound on w/w_{opt} for the value of i that minimizes $(B_m/B_i) + (b_1/b_m)$. For this value of i $(B_m/B_i) + (b_1/b_m) \leq \sqrt{m} + O(m^{1/4})$ as proved in Section 3.2.2.

It suffices to show that w_{opt} satisfies the lower bound of Lemma 3.1 (i.e., $\max(\mu(\mathcal{T})/B_m, h/b_1)$) and w satisfies the upper bound of Lemma 3.2 (i.e., $(\mu(\mathcal{T})/B_i)+(h/b_i)$) for every value of i. The former is immediate, since the lower bound did not consider the fact that nonpreemptive schedules were used.

To get the upper bound on w given in Lemma 3.2, break up all intervals of any maximal usage schedule into two types of intervals. One type of interval is when at least i processors are being used, and the second type is when fewer than i processors are in use. Clearly, one may use at least i processors for at most a total of $\mu(\mathcal{T})/B_i$ units of time. Also, the intervals during which fewer than i processors are in use are height reducing intervals. These height reducing intervals decrease the height by a rate of at least b_i per unit time. Lemma 3.2 follows and thus one may conclude:

Theorem 3.4. Let $(\mathcal{T},<,\mu)$ as above. Let w be the finishing time of any preemptive maximal usage schedule, and let w_{opt} be the finishing time of an optimal preemptive schedule. Then $w/w_{opt} \leq \sqrt{m} + O(m^{1/4})$.

3.2.3.3 An Improved Performance Bound

In this section an improved bound on the performance of maximal usage schedules is obtained in terms of the number of processors. The techniques here are different than those of Section 3.2.2 as we proceed to explain. We find two different bounds in terms of the speeds of the processors. The first

bound is obtained by using the fact that in a maximal usage schedule the fastest processor is always in use. This bound essentially proves that maximal usage schedules are somewhat effective even if the slower processors are considerably slower than the fastest processor. The second bound uses techniques similar to those of [56], generalized to the case where the processors are of different speeds. In [56], a $2-(1/m)$ bound is obtained in the identical processor case, for the performance of list schedules (as compared to the optimal preemptive schedule). It turns out that for any set of processor speeds, at least one of the two upper bounds is smaller than $\sqrt{m}+(1/2)$.

Lemma 3.3. Let $(\mathcal{T},<,\mu)$ be a task system. Let w be the finishing time of a maximal usage schedule S and let w_{opt} be the finishing time of an optimal schedule. Then $w/w_{opt}\leq B_m/B_1$.

Proof. As in the proof of Theorem 3.4, $w_{opt}\geq\mu(\mathcal{T})/B_m$. Also note that $w\leq\mu(\mathcal{T})/B_1$. This follows from the fact that the maximal usage heuristic forces the fastest processor to be in use at every moment before the finishing time. Thus the ratio between the finishing time of an arbitrary maximal usage schedule and the optimal schedule is bounded by $(\mu(\mathcal{T})/B_1)/(\mu(\mathcal{T})/B_m)=B_m/B_1$. □

The second bound is smaller than the first in the situation that the speeds of the processors are close to equal (in that case the bound of Lemma 3.3 reduces to approximately m).

Lemma 3.4. Let $(\mathcal{T},<,\mu)$ be a task system. Let w and w_{opt} be as in Lemma 3.3. Then $w/w_{opt}\leq 1+((m-1)(B_1)/B_m)$.

Proof. First note that as in the proof of Theorem 3.4, $w_{opt} \geq \max(\mu(\mathcal{T})/B_m, h/B_1)$.

To determine the value of w, let p_i denote the amount of time during which exactly i processors are used in the schedule S. By definition $w=p_1+\ldots+p_m$. Due to the maximal usage discipline, whenever i processors are in use B_i units of the time requirement are finished per unit time. Thus a total of p_iB_i units of the time requirement of the task system are completed during the p_i units of time that exactly i processors are used. Thus $p_1B_1+\ldots+p_mB_m=\mu(\mathcal{T})$. Solving for p_m in this equation and substituting for p_m in the equation $w=p_1+\ldots+p_m$ yields

$$(8) \qquad w=(p_1+\ldots+p_{m-1})+((\mu(\mathcal{T})-(p_1B_1+\ldots+p_{m-1}B_{m-1}))/B_m).$$

Fix an interval of time during which exactly i processors are being used ($i<m$). Assume that during this interval, an unfinished task T is at the greatest height of the system (i.e., for every t in the interval, the height of T at time t equals the height of $(\mathcal{T},<,\mu)$ at t). Then T is being executed throughout this interval. This follows from the fact that the maximal usage discipline forces the execution of all executable tasks, providing there are enough processors. Since not all of the m processors are being used, all executable tasks are being executed. Furthermore, since T is at the greatest height, it must be executable. It is therefore easy to conclude that when $i<m$ processors are being used, the height of $(\mathcal{T},<,\mu)$ is reduced at a rate of at least b_i, the speed of the i^{th} fastest processor. Thus $b_1p_1+\ldots+b_{m-1}p_{m-1} \leq h$ since the total amount that the "greatest height" can be reduced during all of the times that fewer than m processors are used is at most h.

Rewriting equation (8) and using the lower bounds on w_{opt} produces an upper bound on the performance of arbitrary maximal usage schedules of:

$$(9)\qquad \frac{w}{w_{opt}} \le \frac{(\mu(\mathcal{T})/B_m)+p_1(1-(B_1/B_m))+\ldots+p_{m-1}(1-(B_{m-1}/B_m))}{\max\ (\mu(\mathcal{T})/B_m, h/B_1)}$$

The first lower bound on the optimal schedule in equation 9 is used only as a comparison to the $\mu(\mathcal{T})/B_m$ term in the numerator. Thus:

$$(10)\qquad \frac{w}{w_{opt}} \le 1 + \frac{p_1(1-(B_1/B_m))+\ldots+p_{m-1}(1-(B_{m-1}/B_m))}{(h/B_1)}$$

Now, multiply numerator and denominator of the fractional part of equation (10) by B_mB_1 to obtain:

$$(11)\qquad \frac{w}{w_{opt}} \le 1 + \frac{B_1[(p_1)(B_m-B_1)+\ldots+(p_{m-1})(B_m-B_{m-1})]}{hB_m}$$

Now $B_m-B_i=b_{i+1}+\ldots+b_m\le(m-i)b_i$ since $b_i\ge b_j$ for $j>i$. Thus

$$(12)\qquad \frac{w}{w_{opt}} \le 1 + \frac{(B_1)[(p_1)(b_1)(m-1)+\ldots+(p_{m-1})(b_{m-1})(1)]}{hB_m}$$

Using $p_1b_1+\ldots+p_{m-1}b_{m-1}\leq h$ and further increasing the numerator of equation (12), one obtains $w/w_{opt}\leq 1+((m-1)B_1/B_m)$. □

If all processor speeds are identical then $B_1/B_m=1/m$. In that case, the bound of Lemma 3.4 is $1+((m-1)/m)=2-(1/m)$. Thus it follows from Lemma 3.4 that when the processors are identical, maximal usage schedules are no worse than $2-(1/m)$ times worse than optimal.

Theorem 3.5. Let $(\mathcal{T},<,\mu)$ be a task system. Let w be the finishing time of a maximal usage schedule and let w_{opt} be the finishing time of an optimal schedule. Then $w/w_{opt}\leq\sqrt{m}+(1/2)$.

Proof. By Lemmas 3.3 and 3.4 $w/w_{opt}\leq B_m/B_1$ and $w/w_{opt}\leq 1+((m-1)B_1/B_m)$. Let $r=B_m/B_1$. Then $w/w_{opt}\leq r$ and $w/w_{opt}\leq 1+(m-1)/r$. To maximize $\min(r,1+((m-1)/r))$, solve $r=1+((m-1)/r)$ and obtain $r=(1/2)+(\sqrt{(1+4(m-1))})/2$. This value of r maximizes $\min(r,1+(m-1)/r)$ since r and $1+(m-1)/r$ are inversely related. Thus $w/w_{opt}\leq (1/2)+(\sqrt{(1+4(m-1))})/2 < \sqrt{m}+(1/2)$. □

3.2.3.4. Achievability of the performance bound

This section proves that the bound of $\sqrt{m}+(1/2)$ is almost achievable, even for machines with one fast processor and $m-1$ identical slow processors. The example task system used to prove this is the same as the one used in Section 3.2.2 to show that the algorithm of Section 3.2.2 is at least $\sqrt{m-1}$ times worse than optimal. The optimal schedule for the task system of Figure 3.2.1 is also an optimal preemptive schedule. The "bad" list schedule on the fastest i processors ($i>1$) is also a maximal usage preemptive schedule. The ratio of

the finishing time of the list schedule on the fastest i processors to the finishing time of the optimal schedule is shown in Section 3.2.2.4 to be $\sqrt{m-1}$. □

3.2.4 Scheduling with limited information

While the algorithms of Sections 3.2.2, 3.2.3, (and the ones we will present in Section 3.3) are all better than known algorithms, it is still somewhat disturbing that the performance of these algorithms is as large as $\sqrt{m}$ times worse than optimal. For simpler scheduling problems, there are algorithms that are at most a constant times worse than optimal [23]. One such machine environment was briefly mentioned in Section 3.2.3.3. It was proved in Section 3.2.3.3 that any maximal usage schedule for a machine with identical processors is at most $2-(1/m)$ times worse than optimal.

One might wonder if there are polynomial time algorithms that have worst case behavior that is better than $O(\sqrt{m})$ times worse than optimal for the machine environments considered in this paper. We give a partial answer to that question in this section. We consider a limited class of scheduling algorithms and show that no algorithm in that class performs better than $\sqrt{m}$ times worse than optimal in the worst case even if the tasks are independent (i.e. no partial order). In particular, no algorithm is better than $\sqrt{m}$ times worse than optimal irrespective of the amount of time used by the algorithm.

Specifically, assume that the scheduler knew everything about the task system and the processors except for the time requirements of the individual tasks. Such a circumstance is imaginable if for example there are loops of undetermined number of iterations in the tasks.

In this environment any algorithm must be at least $\sqrt{m}$ times worse than optimal in the worst case. This is true even if before scheduling a task to start at a given time, the scheduler knows the absolute time requirement of all tasks finished by that given time (as one might expect that the scheduler would at least have access to that information). This indicates that the algorithm

of Section 3.2.2 is asymptotic to the best possible heuristic that is restricted to this type of scheduling. It will turn out in a similar way that one of the algorithms of Section 3.3 is best possible to within a constant (multiplicative) factor.

To prove this result consider a task system consisting of m independent tasks to be scheduled on m processors. The rate of the fastest processor is $\sqrt{m}$ and the rate of the other processors is 1. If an algorithm schedules all of the m tasks on the fastest processor then in the worst case, the schedule is $\sqrt{m}$ times worse than optimal. This occurs when each task requires unit time. Then assigning all tasks to the fastest processor requires time $\sqrt{m}$, whereas assigning the j^{th} task to the j^{th} processor requires time 1.

On the other hand, if any of the tasks are not assigned to the fastest processor, the performance ratio is still $\sqrt{m}$ times worse than optimal in the worst case. Assume that the j^{th} task is the first one not assigned to the fastest processor. Assume that the time requirement of each task is ϵ ($\epsilon << 1$), except for the j^{th} task whose time requirement is 1. Then an asymptotically optimal schedule, schedules the j^{th} task on the fast processor, and the k^{th} task ($k \neq j$) on the k^{th} processor. This requires time $(1/\sqrt{m})+\epsilon$, whereas the heuristic requires at least time 1 by not assigning the j^{th} task to the fastest processor. Thus the heuristic is $\sqrt{m}$ times worse than optimal. □

Even in this restricted environment, where the time requirements are not known, the above result is not a strong result for the following reason. A very weak form of preemption avoids the difficulties illustrated in the example. Specifically, if at "run-time", a task which has been begun on one processor may be reassigned to be rerun from the beginning on another, then the

above task system may be executed relatively efficiently even if the absolute time requirements are not known a priori.

3.3 Nonpreemptive Scheduling of Independent Tasks on Unrelated Processors

3.3.1 Basic Definitions and Models

As mentioned in Section 3.1, the algorithms presented in Section 3.3 schedule a set of independent tasks on unrelated processors. A variety of polynomial time algorithms are presented (in Sections 3.3.2, 3.3.5, and 3.3.6). One of these algorithms is as good as $2\sqrt{m}$ times worse than optimal in the worst case (Section 3.3.5). In addition, an algorithm which requires exponential time in the number of processors is at most $1.5\sqrt{m}$ times worse than optimal. All of the bounds on these new algorithms are achievable to within a constant factor.

Due to the fact that there is no precedence constraint on the tasks, the style of definition here is slightly different from that of Sections 3.2 and 3.5, to take advantage of this conciseness. Specifically (as we will proceed to describe), a schedule actually only assigns jobs to processors, without specifying a starting time function explicitly. Since all jobs are initially executable, all the jobs assigned to each processor may be executed in any order, and the finishing time will be the same (assuming no idle time for processors).

For notational conciseness we also name the n tasks by the integers $1,\ldots,n$, and the m processors by the integers $1,\ldots,m$. This was not done where tasks were partially ordered so that one would not confuse the ordering on the tasks with the ordering on the integers.

A *task system of n tasks and m unrelated processors* is an $n \times m$ matrix μ with entries in $\mathbb{R}^+ \cup \{\infty\}$ for $n,m \geq 1$ such that for every t there is a p such that $\mu(t,p) \neq \infty$ $(1 \leq t \leq n)$.

The value $\mu(t,p)$ is the *time requirement* of the t^{th} task on the p^{th} processor ($1 \leq t \leq n$, $1 \leq p \leq m$).

The execution of jobs by processors is modelled by the notion of an assignment function for a task system. An *assignment function* A is a map $A:\{1,...,n\} \rightarrow \{1,...,m\}$ such that $\mu(t,A(t)) \neq \infty$ for $t=1,...,n$. If $A(t)=p$ we will often say that *task t is assigned to processor p*. In general we will often refer to the index t as the t^{th} task, and the index p as the p^{th} processor. The *finishing time of an assignment*, A, (denoted $w(A)$) is defined by:

$$w(A) = \max_{1 \leq p \leq m} \sum_{t:A(t)=p} \mu(t,p)$$

An *optimal assignment* is any assignment that minimizes the finishing time. For two assignments A and B, the *performance ratio of A to B* is $w(A)/w(B)$.

Intuitively, the assignment function assigns jobs to processors. The finishing time intuitively represents the largest amount of time needed by any of the processors to execute the tasks assigned to it.

While the starting time of each task is not an intrinsic portion of the schedule, it is still sometimes useful to associate a starting time function s, with a given assignment function A. Intuitively, for $1 \leq t \leq n$ the value $s(t)$ represents the time at which processor $A(t)$ begins to execute task t. Formally, a *starting time function* s, is a map $s:\{1,...,n\} \rightarrow \mathbb{R}$ satisfying conditions (a) and (b) below. The value $s(t)$ is called the *starting time* of the task t. Task t is *being executed on processor p at time x* providing that $p=A(t)$ and $s(t) \leq x < s(t)+\mu(t,p)$.

A starting time function satisfies:

(a) For $p=1,...,m$, at most one task is being executed at any time on processor p.

(b) For $p=1,...,m$, if $0 \le x < \Sigma_{t:A(t)=p} \mu(t,p)$ then at least one task is being executed at time x on processor p.

Intuitively, the first condition forces all tasks assigned to a processor to be executed sequentially, and the second condition prevents any idle periods on the processor.

By abuse of notation $w(t)=s(t)+\mu(t,A(t))$ is called the *finishing time* of the task t. Similarly, if $T \subset \{1,...,n\}$, $w(T)=\max_{t \in T} w(t)$.

The matrix μ associates m possible time requirements with each task. The *best time* of the t^{th} task, (denoted $b(t)$), is the smallest of these m values (i.e. $b(t)=\min_p \{\mu(t,p)\}$). The *efficiency* of the p^{th} processor on the t^{th} task (denoted $ef(t,p)$) is $b(t)/\mu(t,p)$. Note that the maximum efficiency is one.

The following algorithm devises an assignment A and a starting time function s for a given task system. The basic idea is that the processor which will finish first according to the tasks currently assigned is assigned the unassigned task for which it is most efficient. If, however, there are no tasks for which the processor is somewhat efficient, then the algorithm stops assigning tasks to that processor.

3.3.2 A Scheduling Algorithm

Algorithm 1.

1. For $1 \le p \le m$ sort the n values $ef(t,p)$, $1 \le t \le n$. Set $sum_p=0$ for $p=1,...,m$. Designate all processors as being "active" and all tasks as being "unassigned".

2. Find any value of p such that sum_p is minimal among active processors. (Note that there must always be such a p.)

3. Find the task, t with largest value of $ef(t,p)$ among unassigned tasks. If there are no unassigned tasks then HALT. If $ef(t,p) \geq 1/\sqrt{m}$ go to Step 4. Otherwise designate p as being inactive and go to Step 2.

4. Define $A(t)=p$. Designate t as being assigned. Define $s(t)=sum_p$. Set $sum_p=sum_p+\mu(t,p)$. Go to Step 2.

It may be noted that A is an assignment function. The algorithm terminates when there are no unassigned tasks remaining. Note that at termination some processor is active. For, at any point during the assignment of tasks, the processor that has the best time on any unassigned task could not have yet been deactivated.

Since each iteration of the main loop (steps 2-4) either assigns a task or deactivates a processor, the algorithm terminates after $n+m$ iterations.

It may be noted that this algorithm can be applied as a run time scheduler even if the absolute time requirements of each of the jobs are not known in advance, as long as the efficiencies are known. The only place that the time requirements were needed was in determining which processor is the next to be assigned a task. If this decision is made at run time (i.e., after a processor completes all tasks already assigned to it), then the time requirements are not needed while the assignment is being made.

We recall at his point the results of Section 3.2.4. It was shown that if the time requirements are not known then any algorithm is at least $\sqrt{m}$ times worse than optimal. This algorithm is in the class of schedules considered, although the efficiencies must be known for Algorithm 1. It is conceivable

that a scheduler would know the efficiencies but not the actual time requirements. For example, if each task had known characteristics which indicated which machine would handle it well, but the length of the jobs were unknown, this assumption would be relevant.

The assignment A and starting time function s determined by Algorithm 1 satisfy the following three conditions:

(1) If $A(t)=p$ then $ef(t,p)\geq 1/\sqrt{m}$.

(2) If $s(t)>s(u)$, then $ef(t,A(u))\leq ef(u,A(u))$ $(1\leq t,u\leq n)$.

(3) If $\Sigma_{t:A(t)=p}\,\mu(t,p)<s(u)$, then $ef(u,p)<1/\sqrt{m}$.

Intuitively, condition one indicates that a task is executed on a processor only if the processor is "somewhat" efficient for the task. The second condition ensures that if a task u is assigned at an earlier time than t, it must be that processor $A(u)$ is more efficient on u than t. The third condition prevents a processor from stopping as long as it is still somewhat efficient for some unstarted task.

The fact that Algorithm 1 satisfies condition (1) is immediate from the way tasks are assigned. Condition (2) follows from the fact that if $s(t)>s(u)$, then the fact that u (and not t) is assigned to $A(u)$ implies that u must have at least as high an efficiency on $A(u)$ as t. Condition (3) also follows immediately from the way tasks are assigned.

In the sequel, the only facts used about Algorithm 1 are conditions (1), (2), and (3) above. Thus the analysis of Algorithm 1 applies to any algorithm that follows these principles. In Section 3.3.5 we combine Algorithm 1 with a different heuristic which preserves (1), (2), and (3), and thus the entire analysis applies tc the modified algorithm.

To analyze the running time note that the presorting requires time

$O(n\log(n))$ for each value of p (if sorting is done with comparisons). Consider the total time spent on iterations in which the value p is chosen in Step 2. Determining if a given task is unassigned requires time $O(\log(n))$ if that information is stored as a bit array. Thus, steps 3 and 4 of all iterations in which a particular p is chosen require time at most $O(n\log(n))$ (assuming that the list of tasks sorted by $ef(t,p)$ is maintained with a pointer to the task that will be looked at next). If a data structure is maintained which keeps sum_p sorted, the $m+n$ iterations of step 2 require at most time $O((m+n)(\log(m)))$. If $m>n$, a slightly different data structure permits the execution of step 2 in time $O(m\log(n))$. Thus the total running time is $O(mn\log(n))$.

3.3.3. Analysis of Algorithm 1

To analyze Algorithm 1 for a given task system, we fix a particular assignment A and starting function s consistent with (1), (2), and (3). Certain subsets of $\{1,...,n\}$ will be defined based on s,A, and an optimal assignment function B. Let z be the last (or one of the last if there are ties) task, to be completed, i.e., $w(z)=w(A)$. Consider a value p such that $\mu(z,p)=b(z)$, i.e., processor p is one of the processors that is most efficient on z. Without loss of generality assume that $p=1$, i.e., processor 1 is most efficient on z.

Definition. Let K consist of those tasks executed on the first processor with starting time earlier than $s(z)$, i.e. $K=\{t:\ A(t)=1 \text{ and } s(t)<s(z)\}$.

By condition (3) the first processor may not finish earlier than $s(z)$. Furthermore, all tasks of K must have efficiency one on the first processor or else condition (2) is violated.

The set K will be partitioned into two sets. The assignment of each task in B determines the set to which it belongs.

Definition. Let $L=\{t: t\in K \text{ and } ef(t,B(t))\geq 1/\sqrt{m}\}$.

Thus L consist of those tasks executed on processor 1 before time $s(z)$, that are assigned somewhat efficiently in B. The second subset of K will be denoted $K-L$.

Let $A^{-1}(\{2,...,m\})$ be the set of tasks that A does not assign to the first processor, i.e. $A^{-1}(\{2,...,m\})=\{t:A(t)\neq 1\}$.

To proceed with the proof, it is convenient to define a few more quantities. For a set of tasks $T\subset\{1,...,n\}$ and an assignment function ASG, *the actual workload of T under ASG* (denoted $E(ASG,T)$) is $\Sigma_{t\in T}\mu(t,ASG(t))$. The *minimal workload for T* (denoted $E_0(T)$) is $\Sigma_{t\in T}b(t)$.

In order to evaluate $w(A)/w(B)$, it is convenient to consider a related task system, μ'. This task system has almost identical entries as μ, differing only in the entries for tasks in $K-L$. In addition, there is a starting time function, s', associated with A such that s' is consistent with (1), (2), and (3), and s' starts all of L, before it starts any of $K-L$.

Lemma 3.5. Let μ be an $n \times m$ task system, A an assignment function for μ obtained from Algorithm 1, and s the associated starting time function. Let K and L as above. Let B be an optimal assignment for μ. Then there is an $n \times m$

task system μ', and a starting time function s' (with finishing time w') associated with A such that:

(i) The starting time function s' with the assignment A is consistent with (1), (2), and (3) for μ'.

(ii) $w(A)=w'(A)$ and $w(B)=w'(B)$.

(iii) The sets K and L are identical when defined in terms of μ, s, and A, as when they are defined in terms of μ', s', and A.

(iv) For every $t_1 \in L$ and $t_2 \in K-L$, $s'(t_1)<s'(t_2)$.

(v) $E_0(L)=w'(L)$

Proof. Define μ' as follows. For $t \notin K-L$ $\mu'(t,p)=\mu(t,p)$. For $t \in K-L$, $\mu'(t,A(t))=\mu(t,A(t))$, $\mu'(t,B(t))=\mu(t,B(t))$, and $\mu'(t,p)=\infty$ for $p \neq A(t)$ and $p \neq B(t)$. Note that $w(A)=w'(A)$ and $w(B)=w'(B)$. (In fact B is still an optimal assignment). The starting time function s' is defined to be only slightly different from s. For $t \notin K$, $s'(t)=s(t)$. For $t \in L$:

$$s'(t)= \sum_{u \in L : s(u)<s(t)} \mu'(u,1)$$

For $t \in K-L$:

$$s'(t)=E_0(L) + \sum_{u \in K-L : s(u)<s(t)} \mu'(u,1)$$

Thus, all of the tasks in L are scheduled before all of the tasks in $K-L$, and within each set (L and $K-L$), the order that the tasks are scheduled is the same in s. Also $E_0(L)=w'(L)$ and the sets K and L are unchanged.

Note that A with s' is consistent with (1), (2), and (3) for μ'. Condition (1) follows immediately since tasks are executed on the same processors, with the same efficiency as in s. Condition (2) needs to be verified only for tasks in K. If $u \in K$ and $s'(t) > s'(u)$ then since $ef(u,1)=1$, $ef(t,1) \leq ef(u,1)$. If $t \in L$, $s'(t) > s'(u)$ and $A(u) \neq 1$, then since $s(t) \geq s'(t) > s'(u) = s(u)$, $ef(t,A(u)) \leq ef(u,A(u))$ (using the fact that (2) was true for s). If, $t \in K-L$ and $s'(t) > s'(u)$, condition (2) follows from the fact that in the primed system all tasks $t \in K-L$ have efficiency less than $1/\sqrt{m}$ for all but the first processor. Condition (3) follows in a similar manner. □

This modification of μ to μ' is an *ad hoc* modification needed to make the technical assumption (v).

To obtain a bound on $w(A)/w(B)$ we compute a bound on the ratio using the starting function s' on the task system μ'. Note that $w(K) \geq s(z)$ by condition (3) on A and s. Also, $w(A) = s(z) + E(A,\{z\})$. Therefore, $w(A) = s(z) + E(A,\{z\}) \leq E(A,\{z\}) + w(K) = E(A,\{z\}) + E(A,K-L) + E(A,L)$. A bound will be obtained on the three summands in terms of $w(B)$. This will be used to get a bound on $w(A)/w(B)$. In the sequel, we will drop the primes of μ, s, and w, for notational convenience. The proof however refers to this modified task system and starting time function.

Lemma 3.6. Let A,B as above and $1 \leq t \leq n$. Then $E(A,\{t\}) \leq \sqrt{m}\, w(B)$.

Proof. This follows from the fact that A assigns tasks to processors that are somewhat efficient for the task. That is,

$w(B) \geq E(B,\{t\}) = \mu(t,B(t)) \geq b(t) = ef(t,A(t))\mu(t,A(t))$. By condition (1), $ef(t,A(t)) \geq 1/\sqrt{m}$. Thus $ef(t,A(t))\mu(t,A(t)) \geq \mu(t,A(t))/\sqrt{m} = E(A,\{t\})/\sqrt{m}$. □

Lemmas 3.7, 3.8, and 3.9 are technical facts that are needed for the proof of Lemma 3.10.

Lemma 3.7. Let A,B,K,L as above. Then $\sqrt{m}\, E(A,K-L) \leq E(B,K-L)$.

Proof. $E(B,K-L) = \Sigma_{t \in K-L} \mu(t,B(t)) \geq \Sigma_{t \in K-L} \sqrt{m}\, \mu(t,A(t))$ by the definition of $K-L$. Thus $E(B,K-L) \geq E(A,K-L)\sqrt{m}$. □

Let $t_i \in L$ be the i^{th} task of L (ordered by starting time). Assume $|L| = q$.

Lemma 3.8. Let A,L as above. Then
$\Sigma_{t \in L} b(t)w(t) = \Sigma_{i=1}^{q} b(t_i)\Sigma_{j=1}^{i} b(t_j) \geq (\Sigma_{i=1}^{q} b(t_i))^2/2 = E(A,L)^2/2$.

Proof. The first equality follows from the fact that for $t \in L$, $A(t) = 1$, and the fact that all tasks in L are executed sequentially with no intervening tasks. The last equality follows from the definition of $E(A,L)$. To get the inequality $\Sigma_{i=1}^{q} b(t_i)\Sigma_{j=1}^{i} b(t_j) \geq (\Sigma_{i=1}^{q} b(t_i))^2/2$ note that on the left hand side of the inequality the term $b(t_i)b(t_j)$ appears exactly once. This term appears twice on the right hand side for $i \neq j$ and once for $i = j$ in the expansion for $E(A,L)^2$. Dividing by two proves the result. □

Lemma 3.9. Let $A,B,A^{-1}(\{2,...,m\}),L$ as above. Then

$$E_0(L\cup A^{-1}(\{2,...,m\}))\geq(E(A,L)^2/2w(B))-E(A,L).$$

Proof. The main intuitive idea will be to show that if the set L is large, then $E_0(A^{-1}(\{2,...,m\}))$ must be large. The reason is as follows. Consider a task $t\in L$. Since processor $B(t)$ is somewhat efficient on t by the definition of L, processor $B(t)$ is a candidate processor for t to be assigned to in A. The only reason that t would not be assigned to processor $B(t)$ is because the workload in A of tasks assigned to processor $B(t)$ exceeded $s(t)$ (by condition (3)). Thus if L is a large set, it must be that the total workload of $A^{-1}(\{2,...,m\})$ is large, or else tasks of L would be assigned earlier to different processors. Thus a relation is derived relating the size of L to the value of $E_0(A^{-1}(\{2,...,m\}))$.

To make the above intuitive ideas precise, it is convenient to divide L into m portions. $L_p=L\cap B^{-1}(p)$ is the subset of L that is assigned in B to the p^{th} processor, and in particular is executable efficiently on the p^{th} processor. The set $A^{-1}(p)$ (the tasks assigned to the p^{th} processor by A) is the set that we will show is large, based on the size of L_p. Note that $A^{-1}(\{2,...,m\})=\cup_{p=2}^{m}A^{-1}(p)$.

For the rest of the proof, fix a value of p and define the following parameters (which have an implicit dependence on p). Let e be the number of tasks in L_p, and let these tasks be denoted $t_1,...,t_e$ with $s(t_1)<s(t_2)<...<s(t_e)$. For $i=1,...,e$, define $U_i=\{t\in A^{-1}(p) : s(t)<s(t_i)\}$. The set U_i is the subset of $A^{-1}(p)$ started before task t_i.

Note that $ef(t_i,p)\geq 1/\sqrt{m}$ by the definition of L. Thus, by condition (3) $w(A^{-1}(p))\geq s(t_i)$ for every i. Thus some task in U_i finishes at $s(t_i)$ or later

and thus $w(U_i)\geq s(t_i)$ where w refers to the finishing time of U_i in the schedule A.

For $i=1,\ldots,e$, define ef_i to be the smallest efficiency of any of the tasks in U_i on p, i.e., $ef_i=\min_{t\in U_i} ef(t,p)$. Note that if $k>i$, then $ef_k\leq ef_i$ since $U_i\subset U_k$. Also, $ef(t_i,p)\leq ef_i$ by condition (2) since otherwise t_i would have been assigned to p before the last task in U_i was assigned to p. Note that

(13) $$w(B)\geq \Sigma_{i=1}^{e}\mu(t_i,p).$$

This follows because $w(B)$ exceeds the actual workload assigned to any one processor in B. Also note that

(14) $$\mu(t_i,1)=b(t_i)=ef(t_i,p)\mu(t_i,p)\leq ef_i\mu(t_i,p).$$

Using the above definitions and inequalities, we may now show that the minimal workload of the set $A^{-1}(p)$ must be large if L_p is large.

For convenience define $U_0=\emptyset$, $w(\emptyset)=0$, and $ef_{e+1}=0$. From the definitions we have

(15) $$E_0(A^{-1}(p))\geq E_0(U_e)=\Sigma_{t\in U_e} b(t)=\Sigma_{i=1}^{e}\Sigma_{t\in U_i-U_{i-1}} b(t)=\Sigma_{i=1}^{e}\Sigma_{t\in U_i-U_{i-1}} ef(t,p)\mu(t,p).$$

Using (15) and $ef(t,p) \geq ef_i$ for $t \in U_i - U_{i-1}$ produces

(16) $$E_0(A^{-1}(p)) \geq \sum_{i=1}^{e} \sum_{t \in U_i - U_{i-1}} ef_i \mu(t,p).$$

Combining (16) with the fact that $\sum_{t \in U_i - U_{i-1}} \mu(t,p) = w(U_i) - w(U_{i-1})$ (where again w refers to A) produces

(17) $$E_0(A^{-1}(p)) \geq \sum_{i=1}^{e} ef_i (w(U_i) - w(U_{i-1})) = \sum_{i=1}^{e} w(U_i)(ef_i - ef_{i+1}).$$

The last equality is obtained by rearranging indices. Now, using $w(U_i) \geq s(t_i) = (w(t_i) - b(t_i))$ we have:

(18) $$E_0(A^{-1}(p)) \geq \sum_{i=1}^{e} (w(t_i) - b(t_i))(ef_i - ef_{i+1})$$

To get (18) from (17) note $ef_i \geq ef_{i+1}$.

Let $X = \sum_{i=1}^{e} w(t_i)(ef_i - ef_{i+1})$ and $Y = \sum_{i=1}^{e} (ef_i - ef_{i+1}) b(t_i)$. Then $E_0(A^{-1}(p)) \geq X - Y$. Note that $ef_i \leq 1$ for every i. Thus $Y \leq \sum_{i=1}^{e} b(T_i)$.

Also, for each k we have:

(19) $$ef_k w(t_k) \leq (\sum_{i=1}^{k-1} (ef_i - ef_{i+1}) w(t_i)) + ef_k f(t_k) + (\sum_{i=k+1}^{e} ef_i (w(t_i) - w(t_{i-1}))) = X.$$

Equation (19) follows from $ef_i \geq ef_{i+1}$ for every i and $w(t_i) > w(t_{i-1})$ for every i. Using (14), (19), and (13) (successively) provides:

(20) $\Sigma_{i=1}^{e} b(t_i)w(t_i) \leq \Sigma_{i=1}^{e} \mu(t_i,p)ef_i\, w(t_i) \leq \Sigma_{i=1}^{e} \mu(t_i,p)X \leq w(B)X.$

Thus $X \geq \Sigma_{t\in L_p} w(t)b(t)/w(B)$. This together with the bound on Y provides us with

(21) $E_0(A^{-1}(p)) \geq \Sigma_{t\in L_p}(w(t)b(t)/w(B)) - \Sigma_{t\in L_p} b(t).$

A special computation is done for $E_0(L)$. We claim that

(22) $E_0(L) \geq \Sigma_{t\in L_1}((w(t)b(t)/w(B)) - b(t)).$

This follows from $\Sigma_{t\in L_1}((w(t)b(t)/w(B) - b(t)) \leq \Sigma_{t\in L_1} w(t)b(t)/w(B)$ $\leq (\max_{t\in L_1} w(t))\Sigma_{t\in L_1} b(t)/w(B) \leq \max_{t\in L_1} w(t)$. The last inequality follows as in (13) since $w(B)$ exceeds the finishing time of any one processor. Finally, by Lemma 3.5, $\max_{t\in L_1} w(t) = E_0(L)$ and thus (22) is verified. This is the step for which we had to transform the original task system into μ'.

Finally, we compute $E_0(L \cup A^{-1}(\{2,\ldots,m\}))$. $E_0(L \cup A^{-1}(\{2,\ldots,m\})) = (\Sigma_{p=2}^{m} E_0(A^{-1}(p))) + E_0(L)$ by definition. By (21) and (22), $E_0(L \cup A^{-1}(\{2,\ldots,m\})) \geq (\Sigma_{p=2}^{m} \Sigma_{t\in L_p}((w(t)b(t)/w(B)) - b(t))) + (\Sigma_{t\in L_1}((w(t)b(t)/w(B)) - b(t)))$. Thus

(23) $E_0(L \cup A^{-1}(\{2,\ldots,m\})) \geq \Sigma_{t \in L}\, w(t)b(t)/w(B)-b(t)$.

By Lemma 3.8 and equation (23), $E_0(L \cup A^{-1}(\{2,\ldots,m\})) \geq (E(A,L)^2/2w(B))-E(A,L)$. □

Lemma 3.10. Let A,B,K,L as above. Then $E(A,K-L)+E(A,L) \leq (1.5\sqrt{m}+1+(1/2\sqrt{m}))w(B)$.

Proof. Combining Lemmas 3.7 and 3.9 we derive that $E(B,\{1,\ldots,n\}) \geq E_0(L \cup A^{-1}(\{2,\ldots,m\}))+E(B,K-L) \geq (E(A,L)^2/2w(B))-E(A,L)+\sqrt{m}\,E(A,K-L)$. Note that $mw(B) \geq E(B,\{1,\ldots,n\})$, since B can do no better than divide the workload of all the tasks equally among the m processors. Thus $mw(B) \geq (E(A,L)^2/2w(B))-E(A,L)+\sqrt{m}\,E(A,K-L)$. Let $a=E(A,K-L)/w(B)$ and $b=E(A,L)/w(B)$. Then, $2m \geq b^2-2b+2a\sqrt{m}$. To prove the lemma, we determine the maximum value for $(E(A,L)+E(A,K-L))/w(B)=a+b$ subject to $2m \geq b^2-2b+2a\sqrt{m}$.

Note that the maximum value of $a+b$ occurs at $a=(2m-b^2+2b)/2\sqrt{m}$ (for any fixed value of b). Now, to maximize $b+((2m-b^2+2b)/2\sqrt{m})$, differentiate with respect to b, and set the derivative to 0. Solving $1+(2-2b)/2\sqrt{m}=0$ produces $b=1+\sqrt{m}$. For that value of b, the maximum value for a is $(\sqrt{m}/2)+(1/2\sqrt{m})$. Thus the maximum value of $a+b$ is $1.5\sqrt{m}+1+(1/2\sqrt{m})$ and the lemma is proved. □

Theorem 3.6. Let A be an assignment function for a task system consistent with (1), (2), and (3). Let B be an optimal assignment function. Then $w(A)/w(B) \leq 2.5\sqrt{m}+1+(1/2\sqrt{m})$.

Proof. Reduce μ to μ' as above, and analyze $w(A)/w(B)$, by analyzing the assignment function A together with the starting function s'. Note that by

Lemma 3.6 $E(A,\{z\}) \leq \sqrt{m}\, w(B)$. Use $w(A) \leq E(A,\{z\}) + E(A,K-L) + E(A,L)$ together with $E(A,\{z\}) \leq \sqrt{m}\, w(B)$ and $E(A,K-L)+E(A,L) \leq (1.5\sqrt{m}+1+(1/2\sqrt{m}))w(B)$ to get the theorem. □

3.3.4. Achieving the bound to a constant factor

Consider the following $m+1 \times m$ task system. For $t=1,\ldots,m-2$, $\mu(t,1)=1$, $\mu(t,t+2)=\epsilon+\sqrt{m}$, and $\mu(t,p)=\infty$ for $p \neq 1,t+2$. The value $\epsilon>0$ is a parameter which will be sent to zero to get as tight a bound as possible. In addition, $\mu(m-1,2)=\epsilon$ and $\mu(m-1,p)=\infty$ for $p \neq 2$; $\mu(m,3)=m-2-\epsilon$, $\mu(m,2)=\sqrt{m}$, and $\mu(m,p)=\infty$ for $p \neq 2,3$; $\mu(m+1,1)=\sqrt{m}$, $\mu(m+1,3)=m$, and $\mu(m+1,p)=\infty$ for $p \neq 1,3$.

An optimal assignment is as follows. For $t=1,\ldots,m-2$, $B(t)=t+2$, requiring time $\epsilon+\sqrt{m}$. $B(m-1)=B(m)=2$, and thus the actual workload of the tasks assigned to processor 2 is $\epsilon+\sqrt{m}$. Finally, $B(m+1)=1$, requiring time $\sqrt{m}$.

An assignment consistent with (1), (2), and (3) may proceed as follows. First note that no tasks may be assigned to processors $4,\ldots,m$ since no task is sufficiently efficient on those processors. $A(t)=1$ for $t=1,\ldots,m-2$ with $s(t)=t-1$. $A(m-1)=2$ with $s(m-1)=0$. $A(m)=A(m+1)=3$ with $s(m)=0$ and $s(m+1)=m-2-\epsilon$. It is straightforward to verify that this satisfies (1), (2), and (3).

Due to the tasks assigned to the third processor, $w(A)=2m-2-\epsilon$. The ratio between finishing times is $(2m-2-\epsilon)/(\epsilon+\sqrt{m})$. As ϵ gets smaller the ratio approaches $2\sqrt{m}-(2/\sqrt{m})$. □

3.3.5 Modifications of the Basic Algorithm

In this section we consider a number of minor modifications to Algorithm 1 which provide slightly better performance bounds:

Algorithm 2.

Step 1. Same as Step 1 in Algorithm 1, except that if $ef(t,p)=ef(u,p)$, then t is ordered before u if $\mu(t,p)\geq\mu(u,p)$.

Steps 2-4. Same as Algorithm 1.

This algorithm satisfies the following stronger form of condition (2).

(2′) If $s(t)>s(u)$ then either $ef(t,A(u))<ef(u,A(u))$ or $ef(t,A(u))=ef(u,A(u))$ and $\mu(t,A(u))\leq\mu(u,A(u))$.

We obtain a bound on this algorithm of approximately $(1+\sqrt{2})\sqrt{m}$ times worse than optimal.

Using the notation of Section 3.3.3, we wish to place a bound on $(E(A,K-L)+E(A,\{z\}))/w(B)$. If $K-L$ is empty then $E(A,K-L)+E(A,\{z\})/w(B)=E(A,\{z\})/w(B)\leq\sqrt{m}$ by Lemma 3.6. If $K-L$ is not empty, choose $t\in K-L$. Then $w(B)\geq\mu(t,B(t))\geq\sqrt{m}\,\mu(t,1)\geq\sqrt{m}\,\mu(z,1)\geq\mu(z,A(z))=E(A,\{z\})$. Thus $E(A,\{z\})/w(B)\leq 1$. Furthermore, by Lemma 3.7, $\sqrt{m}\,E(A,K-L)\leq E(B,K-L)$. Since $mw(B)\geq E(B,K-L)$, we conclude $E(A,K-L)/w(B)\leq\sqrt{m}$. Hence in either case $(E(A,K-L)+E(A,\{z\}))/w(B)\leq\sqrt{m}+1$.

Applying the inequality $E_0(LUA^{-1}(\{2,\ldots,m\}))\leq mw(B)$ to Lemma 3.9, we obtain $mw(B)\geq(E(A,L)^2/2w(B))-E(A,L)$. Let $x=E(A,L)/w(B)$. Then $m\geq(x^2/2)-x$. Solving this for the maximum possible value for x yields $x=(1+\sqrt{2m+1})$. Thus $E(A,L)/w(B)\leq 1+\sqrt{2}\sqrt{m}+(1/2\sqrt{2}\sqrt{m})$. Using the above inequalities proves:

Theorem 3.7. Let A be an assignment given by Algorithm 2 and let B be an optimal assignment. Then $w(A)/w(B)\leq(1+\sqrt{2})\sqrt{m}+2+(1/\sqrt{8}\sqrt{m})$.

Next we present a different modification of Algorithm 1, which has a worst case performance ratio of $1.5\sqrt{m}$ time worse than optimal. This algorithm has running time $O(m^m+mn\log(n))$, and is therefore quite useless if the number of processors is an input to the problem. However, in the situation that the scheduler needs to deal with a bounded number of processors, this is still an $O(n\log(n))$ algorithm.

Algorithm 3.

1. Devise an assignment A_1 and starting function s, using Algorithm 1.

2. Let u be the task that has latest starting time in s, and let $U=\{t:w(t)>s(u)\}$.

3. Determine an optimal assignment for the set U (considered as entire task system) by trying all possible assignments. Assign each task in U to the processor that it is assigned to in this optimal schedule (the tasks in U are assigned to start after the other tasks already assigned to the relevant processors).

First note that $|U|\leq m$. This follows from the fact that if $t_1,t_2\in U$ then $A_1(t_1)\neq A_1(t_2)$. For if two tasks are assigned to the same processor, the later one must have starting time which is later than $s(u)$. Note that the running time of Algorithm 3 is $O(m^m+mn\log(n))$. Step 1 requires time $O(mn\log(n))$. Step 3 requires time $O(m^m)$ since each possible assignment must be considered. Once these m^m schedules have been tried, determining the best assignment also requires no more time than $O(m^m)$. We proceed as in Section 3.3.3, letting A be an assignment resulting from Algorithm 3, and letting B be an optimal assignment for μ.

The sets K and L are defined on the basis of u. At time $s(u)$, if $ef(u,p)=1$, then (informally speaking) processor p is still executing tasks that have efficiency one. Assume again that $p=1$. The set K consists of all tasks assigned to processor one with starting time earlier than $s(u)$. As in Section 3.3.3, $L=\{t\in K: ef(t,B(t))\geq 1/\sqrt{m}\}$. The proof that $E(A,L)+E(A,K-L)\leq(1.5\sqrt{m}+1+(1/2\sqrt{m}))w(B)$ still applies, even though the sets L and $K-L$ are not the same as those defined in Section 3.3.3.

Let $w^*(U)$ be the "optimal finishing time" of the set of tasks U when considered as a separate task system (i.e., not as part of μ). Then it follows from the design of the algorithm that $w(A)\leq E(A,K-L)+E(A,L)+w^*(U)$. Thus to obtain a bound on $w(A)/w(B)$ it suffices to get a bound on $w^*(U)$ in terms of $w(B)$. But clearly $w^*(U)\leq w(B)$. Using this fact together with Lemma 3.10 provides:

Theorem 3.8. Let A be a schedule obtained with Algorithm 3, and let B be an optimal schedule. Then $w(A)/w(B)\leq(1.5\sqrt{m}+2+(1/2\sqrt{m}))$.

To show that the bounds on Algorithms 2 and 3 are achievable (to within a constant factor), consider the $m \times m$ task system given by $\mu(t,1)=1$, $\mu(t,t)=\epsilon+\sqrt{m}$, and $\mu(t,p)=\infty$ for $p\neq 1,t$. The heuristics assign all tasks to the first processor and perform $\sqrt{m}$ times worse than optimal. □

The ϵ approximation algorithms of Horowitz and Sahni [27,54] require time n^{2m}/ϵ. Even for a relatively small value of m this algorithm requires too much time, as n^{10} or n^{20} algorithms are actually not feasible. Algorithm 3 requires time $O(n\log(n))$ for any fixed value of m and thus for most values of m it is more feasible than the algorithms of [27,54].

Next we consider even more minor modifications of Algorithm 1 that also provide slightly better performance bounds. Specifically, the threshhold of efficiency chosen by Algorithms 1, 2, and 3 is that if $A(t)=p$ then $ef(t,p)\geq 1/\sqrt{m}$. While we feel that this is the best threshhold to choose, we can actually obtain a better bound using a different threshhold.

Consider the heuristic which is identical to Algorithm 1, except that it uses a threshhold of $1/c\sqrt{m}$. Most of the analysis of the algorithm is the same with the following easily verifiable changes. (Note that the choice of whether tasks go into L or $K-L$ depends on the threshhold of $1/c\sqrt{m}$.) It is curious to note that Lemmas 3.8 and 3.9 do not depend at all on the threshhold.

(1) $E(A,\{z\})\leq c\sqrt{m}\, w(B)$.

(2) $c\sqrt{m}\, E(A,K-L)\leq E(B,K-L)$

(3) In the proof of Lemma 3.10, when trying to maximize $a+b$, the relevant inequality that constrains the maximization is $a\leq(2m-b^2+2b)/2c\sqrt{m}$. This occurs at $b=1+c\sqrt{m}$. For that value of b, the maximum value of a is $(\sqrt{m}(2-c^2)/2c)+(1/2c\sqrt{m})$.

(4) Putting together (1) and (3) provides a bound on the algorithm of $\sqrt{m}((3c/2)+(1/c))$, neglecting lower order terms. The smallest value of this occurs at $c=\sqrt{2/3}$, where the bound is $\sqrt{6m}$ times worse than optimal. This is marginally better than the original bound of $\sqrt{6.25m}$ times worse than optimal. □

Using this technique in conjunction with Algorithm 2 provides slightly better results. If $K-L$ is empty then $(E(A,\{z\})+E(A,K-L))/w(B)\leq c\sqrt{m}$ and $E(A,L)/w(B)\leq\sqrt{2}\sqrt{m}$. If $K-L$ is nonempty then $E(A,A^{-1}(\{2,\ldots,m\}))/w(B)\leq 1$, and $E(A,K-L)+E(A,L)\leq((c/2)+(1/c))\sqrt{m}$. The best value of c to choose is the one that

minimizes $\max(c+\sqrt{2},(c/2)+(1/c))$. Choosing $c=2-\sqrt{2}$ provides a bound of $2\sqrt{m}$. □

Using this technique in conjunction with Algorithm 3 provides marginally better results. In that case, $E(A,\{z\})$ does not effect the bound, and $E(A,K-L)+E(A,L)\leq\sqrt{m}\,((c/2)+(1/c))$ plus lower order terms. For this case, choosing $c=\sqrt{2}$ gives a bound on Algorithm 3 of being at most $\sqrt{2}\sqrt{m}$ times worse than optimal. □

3.3.6 Two Other Algorithms

The following is "Algorithm D" from [30]. This is the algorithm that was shown to be between 2 and m times worse than optimal in the original paper of Ibarra and Kim, and we will show that in the worst case it is at least $1+\log_2(m)$ times worse than optimal.

Algorithm D:

Step 1. $sum_p=0$ for $1\leq p\leq m$, $S=\{1,\ldots,n\}$.

Step 2. If $S=\emptyset$ then end.

Step 3. Find an index $t\in S$ such that $\min_p\{sum_p+\mu(t,p)\}\leq\min_p\{sum_p+\mu(t',p)\}$ for all $t'\in S$. Let p be such that $sum_p+\mu(t,p)$ is minimum. Define $A(t)=p$. Set $sum_p=sum_p+\mu(t,p)$ and $S=S-\{t\}$. Go to step 2.

The basic idea behind "Algorithm D" is the following. After i tasks have been scheduled, the scheduler sets up a temporary goal of trying to schedule one more task and minimize the total finishing time of the system of $i+1$ tasks. The scheduler chooses the task to use as the $i+1^{st}$, and which processor to assign it to. After iterating this procedure n times, all tasks have been assigned.

The following is an $m \times m$ task system for which algorithm D performs poorly. The entries are $\mu(t,p)=1$ if $m-t+1 \geq p$ and $\mu(t,p)=\infty$ if $m-t+1<p$. An optimal schedule has $B(t)=m-t+1$ with a finishing time of 1. However, the following schedule is consistent with Algorithm D.

Assume that m is a power of 2. The first $m/2$ tasks to be assigned are tasks $1,\ldots,m/2$ with $A(t)=t$. After these are assigned, the workload assigned to each of the first $m/2$ processors is 1. Next, tasks $(m/2)+1,\ldots,(3m/4)$ are assigned. Task $(m/2)+p$ is assigned to processor p. Continuing in this manner, tasks $1,(m/2)+1,(3m/4)+1,\ldots$ are all assigned to processor 1. Thus the total workload assigned to processor 1 is $1+\log_2(m)$. The finishing time of A is $1+\log_2(m)$. □

The exact worst case performance of this algorithm is left as an open problem. Note, that while it may be a better algorithm than the algorithms of this paper in terms of worst case performance, it has a longer running time. Algorithm 1 requires time $O(mn\log(n))$ whereas "Algorithm D" requires time $O(mn^2)$.

We mention one final trivial algorithm which is quite simple, and in fact generalizes to the situation where there is a precedence relation. If there is a precedence relation between tasks, then the starting time function is needed in order to determine the finishing time of an assignment function. This is due to the fact that as in Section 3.2 if $t<u$, the restriction $s(u) \geq w(t)$ is imposed.

This naive algorithm is to assign each task to a processor which has efficiency one for the task. Since there is a precedence relation in this

general case, processors may be temporarily idle and reactivated. However, we insist that at no time (before the finishing time) are all processors idle.

Let A be a schedule consistent with the heuristic and B an optimal schedule for such an ordered set of tasks. Clearly $w(A) \leq E_0(\{1,...,n\})$, since $E(A,\{1,...,n\}) = E_0(\{1,...,n\})$, and at least one unit of $E(A,\{1,...,n\})$ is executed per unit time (before the finishing time). However, $w(B) \geq E_0(\{1,...,n\})/m$. This follows from the fact that B can certainly do no better than assign all tasks to their best processor, and furthermore do them m at a time. □

This bound of m times worse than optimal is achievable even without any precedence constraint. Consider the $m \times m$ task system with $\mu(t,1)=1$ (for $t>1$), $\mu(t,t)=1+\epsilon$, and $\mu(t,p)=\infty$ for $p \neq 1,t$. The heuristic assigns all m tasks to processor 1 and requires time m. Optimal scheduling assigns task t to processor t and requires time $1+\epsilon$. As ϵ goes to 0, the ratio approaches m. □

This algorithm was worth mentioning only because precedence constraints have a tendency to make scheduling heuristics perform quite poorly. For example, it can be shown that the natural extension of Algorithm 1 to the case that there are precedence constraints provides an algorithm which is as bad as $m\sqrt{m}$ times worse than optimal in the worst case.

3.4 Scheduling Tasks on Processors of Different Types

3.4.1 Basic Definitions and Models

Section 3.4 generalizes the scheduling models of Section 3.2 to the situation that tasks and processors are all of a given type, and a task of a given type may only be executed by a processor of the same type. This is actually a special case of the unrelated processor model of Section 3.3, except that in this section we do consider systems with precedence constraints. We remark that essentially all scheduling results about untyped task systems with no partial order directly generalize to the situation that there are different types. This follows from the fact that a k type task system with no partial order may be thought of as k separate one type task systems. We thus restrict attention in this section to task systems with partial orders.

Most of the definitions for typed task systems are quite similar to the definitions in the untyped case. These definitions are briefly discussed here, assuming familiarity with Section 3.2.1.

A *k type task system* $(\mathcal{T},<,\mu,\nu)$ is a task system $(\mathcal{T},<,\mu)$ together with a *type function* $\nu:\mathcal{T}\rightarrow\{1,\ldots,k\}$. Intuitively, if $\nu(T)=i$ then T must be executed by a processor of type i.

A set of *processors* of k different types is a set $\mathcal{P}=\{P_{ij} : 1\leq i\leq k$ and $1\leq j\leq m_i\}$. There are m_i processors of type i. There is a *rate* b_{ij} associated with P_{ij} and for $i=1,\ldots,k$, $b_{i1}\geq b_{i2}\geq\ldots\geq b_{im_i}>0$. $\mathcal{P}$ is a set of equally fast processors if $b_{ij}=1$ for every i and j.

A nonpreemptive schedule for a k type task system $(\mathcal{T},<,\mu,\nu)$ is a nonpreemptive schedule, S, for $(\mathcal{T},<,\mu)$ in the sense of Section 3.2.1 satisfying the additional property that if $S(T)=(t,P_{ij})$, then $\nu(T)=i$.

List schedules generalize in a straightforward manner. A (priority)

list $L=(T_1,\ldots,T_n)$ $(T_i \in \mathcal{T})$ consists of a permutation of all the tasks of $\mathcal{T}$. The *list schedule* for $(\mathcal{T},<,\mu,\nu)$ with the list L is defined in the same way as for untyped systems with the following modification. When choosing a task for a potentially idle processor, one chooses the highest priority unexecuted task of the same type. If there are no such tasks, the processor becomes idle, even if there are executable tasks of other types available.

The notion of a list schedule on the t_j fastest processors of type j (for $j=1,\ldots,k$) generalizes directly from the notion of a list schedule on the fastest t (untyped) processors. The basic idea is that one schedules as if the only available processors of type j are the fastest t_j processors of type j.

The total number of steps required by all the type i tasks is denoted by μ_i. *The total processing power of the fastest j processors of type i*, denoted B_{ij} is defined by: $B_{ij}=\Sigma_{l=1}^{j} b_{il}$.

The height of a task is defined in the same way as for untyped systems. That is, the types are ignored in this definition.

The notion of a preemptive schedule for a typed task system is also similar to the notion for an untyped system. The only difference in the definition of a valid schedule is that if T is being executed on P_{ij}, then $\nu(T)=i$.

Maximal usage schedules also generalize in a straightforward manner. the defining characteristics are:

(1) Whenever t_j tasks of type j are executable, then $\min(m_j,t_j)$ tasks of type j are being executed.

(2) Whenever i_j processors of type j are being used, the fastest i_j processors of type j are in use.

The translation algorithm of Section 3.2.3.1 which transforms a given preemptive schedule into a maximal usage schedule applies here too with little additional modifications.

3.4.2 Nonpreemptive Scheduling of Tasks on Equally Fast Processors of Different Types

3.4.2.1 Performance Bounds on List Schedules

In this section a bound is obtained on the performance of any list schedule for a typed task system which is to be scheduled on a set of equally fast processors. It is shown that the performance ratio is at most $k+1-(1/\max(m_1,\ldots,m_k))$. A naive bound on the performance ratio of any list schedule relative to an optimal schedule is given by $m_1+\ldots+m_k$. This follows from the fact that an optimal schedule may use at most $m_1+\ldots+m_k$ processors at each point in time, and the fact that any list schedule uses at least one processor at every point in time. The result of this section is that list schedules are far better than the naive bound. Note that the comparison of list schedules to optimal schedules is applicable even to the situations that no optimal schedule is a list schedule.

When $k=1$, this bound reduces to the bound of $2-(1/m)$ of [23] on the performance of list schedules on untyped task systems to be scheduled on identical processors. The techniques used here are somewhat similar to those, although slightly different.

Some of the results of Section 3.4.2 have been obtained independently by Liu and Liu [44] using somewhat different techniques.

Theorem 3.9. Let $(\mathcal{T},<,\mu,\nu)$ be a k type task system to be scheduled on equally fast processors. Then the performance ratio of any list schedule for $(\mathcal{T},<,\mu,\nu)$ to an optimal schedule for $(\mathcal{T},<,\mu,\nu)$ is at most $k+1-(1/\max(m_1,\ldots,m_k))$.

The proof will closely parallel that of Section 3.2.2.2 for list schedules on general purpose processors of different speeds. First, a series of lower bounds are derived on the finishing time of an optimal schedule for $\mathcal{T}$. Then an upper bound on the finishing time of any list schedule is obtained. The ratio between these bounds is an upper bound on the performance ratio of any list schedule to an optimal schedule.

Lemma 3.11. Let $(\mathcal{T},<,\mu,\nu)$ as above. Let w_{opt} be the finishing time, of an optimal schedule. Then $w_{opt} \geq \max(h,\mu_1/m_1,\ldots,\mu_k/m_k)$.

Proof. Clearly at most m_i units of time requirement of type i tasks may be executed during each time unit (for every i). Thus at least $\lceil \mu_i/m_i \rceil$ units of time must be spent on the execution of $\mathcal{T}$ for every i. A conservative lower bound is thus $\max(\mu_1/m_1,\ldots,\mu_k/m_k)$.

Also, by the way height is defined, the height of $(\mathcal{T},<)$ may decrease at a rate of at most one per unit time. Thus h is a lower bound on the finishing time and $w_{opt} \geq \max(h,\mu_1/m_1,\ldots,\mu_k/m_k)$. □

Lemma 3.12. Let $(\mathcal{T},<,\mu,\nu)$ as above. Let w be the finishing time of a list schedule. Then

$$w \leq (\mu_1/m_1)+(\mu_2/m_2)+\ldots+(\mu_k/m_k)+h(1-(1/\max(m_1,\ldots,m_k))).$$

Proof. Given a list schedule S with finishing time w, divide the interval $[0,w]$ into constant height intervals and height reducing intervals. During height reducing intervals, the height is decreasing by a rate of one per unit time. Thus the total length of height reducing intervals is equal to h. The

goal is now to show that the length of the constant height intervals is at most $(\mu_1/m_1)+\ldots+(\mu_k/m_k)-(h/\max(m_1,\ldots,m_k))$.

Note that at any time unit at which the height is not being reduced, m_i tasks of type i must be executed for some i. The reasoning is as follows. Consider any task at greatest height at time t (among unexecuted tasks). Note that if a task is greatest height, then all predecessors of the task have been finished. Thus every task of greatest height that has not yet executed, is executable at time t. Assume that at t the schedule does not use all m_i processors for any i. That is, for each i, at least one processor is unused at t. Such a "potential waste" of processors may occur only if *all* executable tasks are being executed since otherwise one of the unused processors would be used. In particular all tasks at greatest height are being executed and the height is being reduced. Thus time t is in a height reducing interval contradicting the assumption.

Let h_i denote the total time requirement of type i tasks executed during height reducing intervals. Clearly $\Sigma_{i=1}^{k} h_i \geq h$ since during a height reducing interval of length g, the height of $(\mathcal{T},<)$ is reduced by g, and at least g units of time are completed. Now an upper bound on the length of constant height intervals is obtained. Note that for type i, the total length of time at which m_i tasks of type i are being simultaneously executed is at most $\lfloor(\mu_i-h_i)/m_i\rfloor$. Thus, executing m_i tasks of type i at one time for some i may occur for a total duration of at most $\lfloor(\mu_1-h_1)/m_1\rfloor+\ldots+\lfloor(\mu_k-h_k)/m_k\rfloor$ units of time. That is, the total length of all constant height intervals is at most $(\mu_1/m_1)+\ldots+(\mu_k/m_k)-((h_1/m_1)+\ldots+(h_k/m_k))\leq(\mu_1/m_1)+\ldots+(\mu_k/m_k)-(h/\max(m_1,\ldots,m_k))$. A bound on w is thus given by: $w\leq(\mu_1/m_1)+\ldots+(\mu_k/m_k)+h(1-(1/\max(m_1,\ldots,m_k)))$.□

We may now put together the upper bound on list schedules and the lower bound on optimal schedules. The two results combine to show that the worst possible performance ratio is $k+1-(1/\max(m_1,...,m_k))$.

Proof of Theorem. Fix a k type task system $(\mathcal{T},<,\mu,\nu)$, and let $p=\max(\mu_1/m_1,...,\mu_k/m_k,h)$. Then a lower bound on the optimal schedule is p. A conservative upper bound on any list schedule is $(k+1-(1/\max(m_1,...,m_k)))p$. Thus the performance ratio of the list schedule relative to the optimal schedule is bounded by $k+1-(1/\max(m_1,...,m_k))$. □

3.4.2.2 Achieving the Performance Bound

In this section it is shown that the bound of Section 3.4.2.1 is achievable. Specifically, for any k and any values of $m_1,...,m_k$ there is are k type task system and list schedules for the systems with the property that the finishing time of the schedules approach $k+1-(1/\max(m_1,...,m_k))$ times worse than optimal. In fact, the task systems are unit execution time task systems, i.e., $\mu(T)=1$ for every $T\in\mathcal{T}$. This is significant as for unit execution time task systems at least one optimal schedule is a list schedule.

The set of task systems used for this proof are sketched below (Figure 3.4.1). Each node in the graph represents one task. Arrows specify the partial ordering and the labels of the nodes represent the type of the tasks. As mentioned above, each task has unit execution time. (Assume without loss of generality that $m_k=\max(m_1,...,m_k)$.)

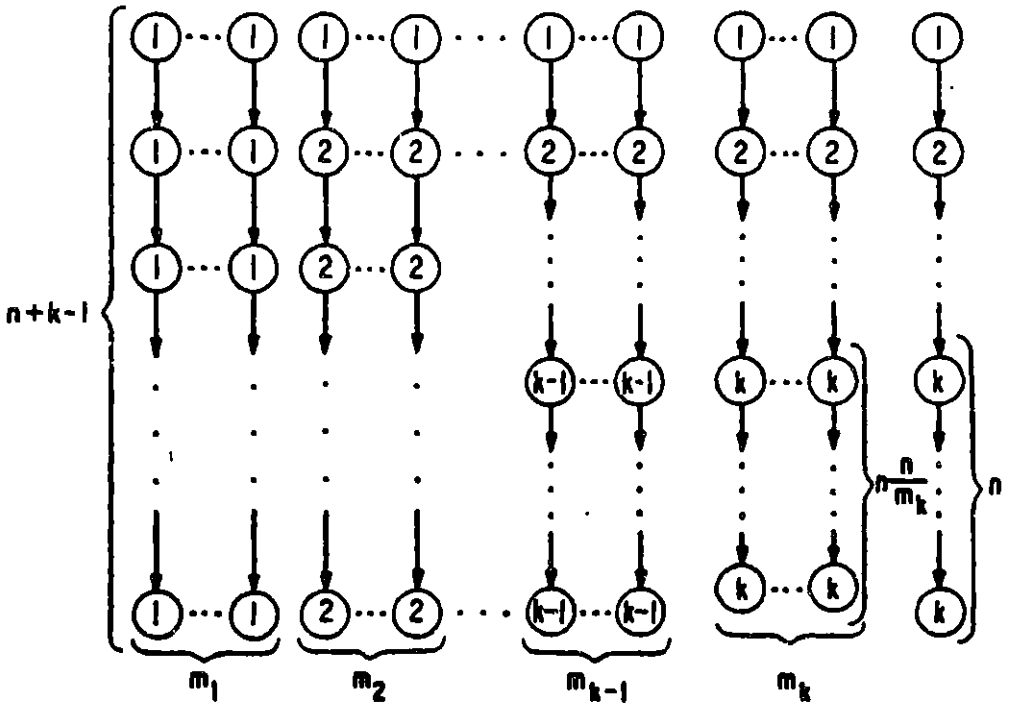


Figure 34.1

In the task systems, there are m_i columns of tasks that consist primarily of type i tasks, which we informally refer to as "corresponding to type i" ($1 \le i \le k-1$). Each of these m_i columns contains a chain of $n+k-1$ tasks (n arbitrary). The j^{th} task in each of these columns has $\nu(T)=j$ (for $j \le i-1$) and $\nu(T)=i$ (for $i \le j$).

There are m_k+1 columns that "correspond to type k". In each of these columns the j^{th} task (for $j \le k-1$) has $\nu(T)=j$. For the first m_k of these m_k+1 columns there is a chain of $n-(n/m_k)$ additional tasks, with $\nu(T)=k$ for each task in the chain. For the $(m_k+1)^{st}$ column there is a chain of n additional tasks, with $\nu(T)=k$ for each task in the chain.

The following is an asymptotically optimal strategy. The first $k-1$ tasks of each column are executed using an arbitrary list schedule. For fixed values of k and the m_i's this may be done in constant time. Now, only n units of time are required to complete the entire system. It is clear that only n units of time are required to finish the columns corresponding to each of the first $k-1$ types of processors. During these same n units of time the columns corresponding to the k^{th} type of processor may be completed as follows: During each of the n units of time, one of the m_k processors of type k is used on the $(m_k+1)^{st}$ of these columns finishing this column in n units of time. The other m_k-1 processors are used on the other m_k columns in rotation. Thus during the first unit of time, no task is executed from the first column, during the second unit of time, no task is executed from the second column, etc. Thus, the total amount of time for this procedure is $n+O(1)$.

An ineffective list schedule is now presented. The schedule first handles all type 1 tasks, then all type 2 tasks, etc. For the first $n+k-1$ units of time only tasks from columns that correspond to type 1 are executed.

At the next $n+k-1$ units of time all tasks from columns that correspond to type 2 are executed, stripping off type 1 tasks from the tops of the rest of the columns in the process. In this manner, $(k-1)n + O(1)$ units of time are required to finish all of the columns that correspond to the first $k-1$ types of processors.

Now the last m_k+1 columns of the program are executed. Using a list schedule, only the first m_k of them are processed for the next $n-(n/m_k)$ units of time, completing these columns in their entirety. Another n units of time are required just to process the last of these m_k+1 columns. The total amount of time with this schedule is thus $n(k+1-(1/m_k)) + O(1)$ and the performance ratio between this and the optimal schedule is $(n(k+1-(1/m_k))+O(1))/(n+O(1))$. As n goes to infinity, the ratio approaches $k+1-(1/m_k)$. □

There are a number of other results for schedules on equally fast typed processors which directly generalize existing results for identical processors. As mentioned in Section 3.4.1 results involving systems with no partial order (e.g. [18]) for the most part generalize directly.

We mention two other generalizations which apply even with a partial order. In [8], Coffman and Graham define a label algorithm which always produces an optimal schedule for scheduling a partially ordered set of unit execution time tasks on two identical processors. This algorithm is generalized in [36] for m identical processors, and the generalized algorithm is at most $2-(2/m)$ times worse than optimal. If $\max(m_1,...,m_k)\geq 2$, then the natural generalization of this algorithm to typed task systems is at most $k+1-(2/\max(m_1,...,m_k))$ times worse than optimal. This is hardly an improvement in performance, but it is an interesting technical result.

The proof of this result is a straightforward generalization of the proof of [36] and thus is omitted. There are only slight modifications necessary to treat the types with only one processor. This bound is achievable [45].

The second generalization is related to the level algorithm of Muntz and Coffman [47,48] for preemptive scheduling. In [36], Lam and Sethi analyze this algorithm for m identical processors, and obtain a performance bound of $2-(2/m)$. Using similar techniques one may generalize this to typed task systems. If $\max(m_1,\ldots,m_k)\geq 2$, then the level algorithm is at most $k+1-(2/\max(m_1,\ldots,m_k)$ times worse than optimal.

3.4.3 Nonpreemptive Scheduling of Tasks on Processors of Different Types and Different Speeds

3.4.3.1 Performance Bounds for List Schedules

This section continues the discussion of scheduling tasks on processors of different types. We consider the situation that within each type, the processors are of uniformly different speeds as defined in Section 3.4.1. In Section 3.4.3.1 a bound is obtained on list schedules on the fastest i_j processors of type j (j=1,...,m_j). Thus the results of this section are an extension of the results of Section 3.2.2 for the untyped case. The results are not as tight, however. There is no single bound (such as the $\sqrt{m}$ + $O(m^{1/4})$ bound of Theorem 3.2) which is developed as the best speed independent bound. Rather a number of bounds may be obtained based on the number of processors of each type and the number of types. Two such bound are discussed in Section 3.4.3.3, either of which may be smaller based on the values of $k,m_1,...,m_k$.

A special case of list schedules on the fastest i_j processors is the case of i_j=m_j (j=1,...,k). This case is the ordinary list schedule case, and the analysis for this case is done slightly more carefully in Section 3.4.3.1. While list schedules may be disasterously worse than optimal in cases that have a wide variation in speeds between the processors, the list scheduling model is nevertheless an important model to be considered separately. For one thing, the analysis informs us about the worst case behavior for systems in which no filtering out of relatively useless processors is done. Also, once it is decided that certain processors should not be used, it is the list scheduling bound which is the relevant bound for the remaining processors.

The approach used in this section closely parallels the approach of Section 3.2.2. The one major difference here is that in order to obtain the

proper bound, the concept of height of a task must be modified to take into account both the type of the task and the speeds of the processors of that type.

To define the height of a task, it is convenient to use a slightly different definition of height for each set of k numbers $i_1,...,i_k$. The following is the definition of height if the fastest i_j processors of type j are used ($j=1,...,k$). (While reading the definition consider the following motivation. We would like to be able to say that the total amount of time spent on height reducing intervals is at most the height of the graph. Thus it is convenient to have the height reduced at least one unit of height per unit time during height reducing intervals.)

The *height length* of a task T (of type j) is given by $\mu(T)/b_{ji_j}$. (Thus if P_{ji_j} (the slowest processor of type j that the algorithm will use) processes T, it executes one unit of the height length of T per unit time.) The length of a chain C, the height of a task T, and the height of $(\mathcal{T},<,\mu,\nu)$ are defined as usual, except that summations are taken with respect to the height lengths of the tasks instead of the time requirements of tasks.

The rest of this discussion assumes a fixed task system $(\mathcal{T},<,\mu,\nu)$ executed on a fixed set of processors $\mathcal{P}$. Also, the discussion fixes which processors are to be used, and thus fixes a notion of the height of a task or of $(\mathcal{T},<,\mu,\nu)$. As before h will denote the height of the system. If h is the height then there is some chain of tasks whose height equals h. Let c_i denote the sum of the time requirements of all type i tasks along this chain. Then $h=(c_1/b_{1i_1})+...+(c_k/b_{ki_k})$.

Lemma 3.13. Let $(\mathcal{T},<,\mu,\nu)$ as above. Let w_{opt} be the finishing time of an optimal schedule. Then

$$w_{opt} \geq \max((\mu_1/B_{1m_1}),(\mu_2/B_{2m_2}),\ldots,(\mu_k/B_{km_k}),((c_1/b_{11})+\ldots+(c_k/b_{k1}))).$$

Proof. The first k bounds follow from the fact that at most B_{jm_j} units of the time requirement of type j tasks can be executed in unit time. To get the last bound, consider a chain of height h as above. Then all the type j tasks in the chain require a total of at least c_j/b_{j1} units of time to be processed, and all tasks must be processed separately. The bound follows immediately. □

Lemma 3.14. Let $(\mathcal{T},<,\mu,\nu)$ as above. Let $w(\{i_j\})$ be the finishing time of a list schedule on the fastest i_j processors of type j (j=1,...,k). Then $w(\{i_j\}) \leq (\mu_1/B_{1i_1})+\ldots+(\mu_k/B_{ki_k})+((c_1/b_{1i_1})+\ldots+(c_k/b_{ki_k}))$.

Proof. As in Section 3.2.2 consider height reducing intervals and constant height intervals. Any constant height interval must have all of the fastest i_j processors of type j in use for some j. The reason is that otherwise the unexecuted task with the greatest height is being executed, reducing the height of the task system. The total time spent in a constant height interval when all i_j processors of type j are in use is at most μ_j/B_{ji_j}. Thus the total time spent on constant height intervals may be at most $(\mu_1/B_{1i_1})+\ldots+(\mu_k/B_{ki_k})$.

Consider height reducing intervals. The greatest height task being executed is being executed at the rate of (at least) 1 unit of height per unit time. Thus the time spent on height reducing levels can be at most $h=(c_1/b_{1i_1})+\ldots+(c_k/b_{ki_k})$. □

Lemmas 3.13 and 3.14 imply:

$$(24)\qquad \frac{w(\{i_j\})}{w_{opt}} \le \frac{(\mu_1/B_{1i_1})+\ldots+(\mu_k/B_{ki_k})+(c_1/b_{1i_1})+\ldots+(c_k/b_{ki_k})}{\max((\mu_1/B_{1m_1}),\ldots,(\mu_k/B_{km_k}),((c_1/b_{11})+\ldots+(c_k/b_{k1})))}$$

One way to choose the set $\{i_j\}$ is to compute the right hand side of equation (24) for each possible choice of processor speeds. As in Section 3.2.2, since this depends on the task system, this can be quite tedious. Note that this expression depends on which notion of height is used, since the c_i's refer to a maximal chain, but maximal chains may be different with the different definitions of height. It is thus even more desirable in the typed case to obtain a task system independent choice of which processors to use. Using techniques similar to those in Section 3.2.2 equation (25) follows from equation (24).

$$(25)\qquad \frac{w(\{i_j\})}{w_{opt}} \le \frac{B_{1m_1}}{B_{1i_1}}+\ldots+\frac{B_{km_k}}{B_{ki_k}}+\frac{(c_1/b_{1i_1})+\ldots+(c_k/b_{ki_k})}{(c_1/b_{11})+\ldots+(c_k/b_{k1})}$$

Let $q=\max((b_{11}/b_{1i_1}),(b_{21}/b_{2i_2}),\ldots,(b_{k1}/b_{ki_k}))$. The value q is the analog of the b_1/b_i term in the ordinary task system case. Note that $q \ge (b_{j1}/b_{ji_j})$ for $j=1,\ldots,k$. Thus $(c_j/b_{ji_j}) \le (q(c_j)/b_{j1})$. Substituting the right hand side of this inequality for c_j/b_{ji_j} in the last of the $k+1$ summands gives a bound on the last term of q. Thus a task system independent way of choosing which processors to use would be to minimize:

$$(26) \qquad \frac{B_{1m_1}}{B_{1i_1}} + \ldots + \frac{B_{km_k}}{B_{ki_k}} + \max(b_{11}/b_{1i_1},\ldots,b_{k1}/b_{ki_k})$$

The heuristic is then to compute equation (26) for each possible set of processor speeds to determine a set of indices i_j (still a somewhat lengthy procedure, but something that needs be done only once), and then use only the fastest i_j processors of type j. The results of Section 3.4.3.3 will indicate that such a choice guarantees performance that is no worse than $k+2\sqrt{\max_j (km_j)}$ times worse than optimal. In fact, the proof technique is such that it suggests one simple way of choosing the i_j so that the bound is reached, and as such even one calculation of equation (26) for each possible choice of $\{i_j\}$ is unnecessary.

As in Section 3.2.2, the bound of Lemma 3.14 may be tightened somewhat by noting that any portion of the time requirement executed during height reducing intervals may not be executed during constant height intervals. The marginally better bound that results is always less than one, and does not contribute to the asymptotic speed independent bound of Section 3.4.3.3. We introduce this tightening only for the case of list schedules on all the processors (i.e. $i_j=m_j$) - a case that we indicated above is of importance in its own right.

Theorem 3.10. Let $(\mathcal{T},<,\mu,\nu)$ be a k type task system to be scheduled on a machine with processors of different speeds as above. Let w be the finishing time of a list schedule, and let w_{opt} be the finishing time of and optimal schedule. Then $w/w_{opt} \leq k + (\max_j (b_{j1}/b_{jm_j}))(1-\min_j (b_{jm_j}/B_{jm_j}))$.

Proof. To get the tighter bound, we take a closer look at the total time spent on constant height intervals. As in Section 3.2.2, this is done by recognizing that any portion of the time requirement that is handled during a height reducing interval cannot be handled during a constant height interval.

Let p_j denote the total time requirement of type j tasks that is completed during height reducing intervals. Note that $(p_1/b_{1m_1})+\ldots+(p_k/b_{km_k})\geq h$. For consider a height reducing interval during which one task (of type j) was at the greatest height throughout the interval (all height reducing intervals may be partitioned into such intervals). Assume that the height is reduced by a total of g in that interval. In that case, at least g_{jm_j} units of the time requirement of that task are completed during that interval. Using this argument for all intervals, the above inequality follows.

Thus, the total time spent on constant height intervals is actually bounded by $\Sigma_{j=1}^{k}(\mu_j-p_j)/B_{jm_j}$. Thus if w is the finishing time of a list schedule, and w_{opt} the finishing time of an optimal schedule we have:

$$\frac{w}{w_{opt}} \leq \frac{(\Sigma_{j=1}^{k}(\mu_j-p_j)/B_{jm_j}) + h}{\max((\mu_1/B_{1m_1}),\ldots,(\mu_k/B_{km_k}),((c_1/b_{11})+\ldots+(c_k/b_{k1})))} \tag{27}$$

Separating out the p_i terms from the summation, and using the first k lower bounds on w_{opt} yields:

$$\frac{w}{w_{opt}} \leq k + \frac{h-((p_1/B_{1m_1})+\ldots+(p_k/B_{km_k}))}{((c_1/b_{11})+\ldots+(c_k/b_{k1}))} \tag{28}$$

Now let $d=\min_j b_{jm_j}/B_{jm_j}$. Then for every value of j, $B_{jm_j} \leq (b_{jm_j}/d)$. By increasing the value of the numerator of the right hand side of (28), one thus obtains:

$$\frac{w}{w_{opt}} \leq k + \frac{h-(d((p_1/b_{1m_1})+\ldots+(p_k/b_{km_k})))}{((c_1/b_{11})+\ldots+(c_k/b_{k1}))} \quad (29)$$

Now use $h \leq ((p_1/b_{1m_1})+\ldots+(p_k/b_{km_k}))$ together with $h=((c_1/b_{1m_1})+\ldots+(c_k/b_{km_k}))$ to obtain:

$$\frac{w}{w_{opt}} \leq k + \frac{(1-d)((c_1/b_{1m_1})+\ldots+(c_k/b_{km_k}))}{((c_1/b_{11})+\ldots+(c_k/b_{k1}))} \quad (30)$$

Let $q=\max(b_{11}/b_{1m_1},\ldots,b_{k1}/b_{km_k})$. Using $b_{jm_j} \geq (b_{j1}/q)$ for every j (in the numerator of the right hand side of (30)) yields:

$$\frac{w}{w_{opt}} \leq k + \frac{q(1-d)((c_1/b_{11})+\ldots+(c_k/b_{k1})))}{((c_1/b_{11})+\ldots+(c_k/b_{k1}))} \quad (31)$$

From equation (31), it follows immediately that $w/w_{opt} \leq k+q(1-d)$, i.e., $w/w_{opt} \leq k + (\max_j (b_{j1}/b_{jm_j}))(1-\min_j (b_{jm_j}/B_{jm_j}))$. □

Note that this improvement is really quite small. It differs from the earlier bound by at most 1, as $(\max_j (b_{j1}/b_{jm_j}))(\min_j (b_{jm_j}/B_{jm_j}))$

is at most 1. For if j is the index that maximizes the first term in the product, the total product is at most $b_{j1}/B_{jm_j} \leq 1$.

3.4.3.3 Speed independent bound

We now return to the more general consideration of analyzing list schedules on the fastest i_j processors of type j (j=1,...,k).

Theorem 3.11. Consider a set of m processors of different speeds with m_j of type j (j=1,...,k). Then some set of indices $\{i_j : 1 \leq j \leq m\}$ have the property that for any task system $(\mathcal{T},<,\mu,\nu)$ (with optimal finishing time w_{opt}), and any list schedule on the fastest i_j processors of type j for that task system (with finishing time $w(\{i_j\})$), that $w(\{i_j\})/w_{opt} \leq k+2\sqrt{\max_j (km_j)}$.

Proof. Let $r=\sqrt{\max_j (km_j)}$. Choose i_j such that $rb_{ji_j} \geq b_{j1}$ and $rb_{ji_{j+1}} < b_{j1}$. (Let $i_j=m_j$ if the second inequality fails for each value of b_{ji}.) Then $(B_{jm_j}/B_{ji_j})=1+((b_{ji_{j+1}}+...+b_{jm_j})/B_{ji_j})$. Each of the m_j-i_j terms in the numerator of the fraction is at most b_{j1}/r. The denominator is at least b_{j1}. Thus B_{jm_j}/B_{ji_j} is at most $1+(m_j/r)$. Now $r \geq \sqrt{km_j}$ for each j. Thus $B_{jm_j}/B_{ji_j} \leq 1+(\sqrt{m_j/k})$.

By the choice of i_j, $r \geq b_{j1}/b_{ji_j}$ for each value of j. Thus $r \geq \max_j (b_{j1}/b_{ji_j})$. Using this, the bound on B_{jm_j}/B_{ji_j} and equation (26) provides:

(32) $w(\{i_j\})/w_{opt} \leq k+\sqrt{1/k}\,(\sqrt{m_1}+\ldots+\sqrt{m_k})+r.$

Increasing $\sqrt{m_i}$ to $\sqrt{\max_j (m_j)}$ in the above expression yields

(33) $w(\{i_j\})/w_{opt} \leq k+k\sqrt{\max_j (m_j/k)}+r.$

From (33) it is immediate that $w(\{i_j\})/w_{opt} \leq k+2r$ for this choice of i_j irrespective of the task systems used or list schedule used. □

The bound of Theorem 3.11 is the bound that one would use if each of the m_j were almost equal. For different situations, though, it might be beneficial to make a different choice of which processors to use. A sample of this is contained in the following theorem.

Theorem 3.12. Consider a set of m processors of different speeds with m_j of type j ($j=1,\ldots,k$). Then some set of indices $\{i_j | 1\leq j\leq m\}$ have the property that for any task system $(\mathcal{T},<,\mu,\nu)$ (with optimal finishing time w_{opt}), and any list schedule on the fastest i_j processors of type j for that task system (with finishing time $w(\{i_j\})$), that $w(\{i_j\})/w_{opt} \leq k+\sqrt{m_1}+\sqrt{m_2}+\ldots+\sqrt{m_k}+\max_j \sqrt{m_j}$.

Proof. Let $r_j=\sqrt{m_j}$. Choose i_j such that $r_j b_{ji_j} \geq b_{j1}$ and $r_j b_{ji_j+1} < b_{j1}$. Then $(B_{jm_j}/B_{ji_j}) \leq 1+\sqrt{m_j}$. Also, $\max_j r_j = \max_j \sqrt{m_j}$ exceeds the $k+1^{st}$ summand in the bound of equation (26). The theorem follows immediately. □

The choice of which bound is better depends on the values of the m_j. If all are equal, the first bound is better by a factor of about $(1/2)\sqrt{k}$. If all

the m_j equal 1 except for one which is large, then the latter bound beats the former by a factor of $\sqrt{k}$.

3.4.3.4 Achievability of the performance bound

Three achievability results will be presented for each of Theorems 3.11 and 3.12, and one for Theorem 3.10. The first two results for Theorem 3.11 and 3.12 discuss the situation where k is quite small relative to the number of processors, and thus the first summand in the bounds of Theorems 3.11 and 3.12 may be ignored.

If k is fixed, and $m_1,...,m_k$ vary, it is easy to see that for some set of processor speeds the bounds of Section 3.4.3.2 are achievable to within a constant factor. For let j be the type that has the most processors (i.e., m_j is largest). Then the same construction used in Section 3.2.2, using only processors of type j gives an achievability result of $\sqrt{m_j-1}$. While this is a factor of at most $2\sqrt{k}$ times worse than the bound of Theorem 3.11 and about $k+1$ times worse than the bound of Theorem 3.12, for a fixed value of k, it is only a constant factor worse than the bound. □

A different achievability result is to show that as k is varied and as the values $m_1,...,m_k$ are varied, there is a set of processors speeds, a task system and a list schedule such that the algorithm is as bad as the bound of Theorem 3.11. This is in fact not true in general. Neither Theorem 3.11 nor Theorem 3.12 is tight for all values of the m_j since either may be as much as a factor of $O(\sqrt{k})$ times more than the true bound (as discussed above). Instead it will be shown that for any values of m and k where $m=\Sigma_{j=1}^{k} m_j$, there are values for m_j and speeds for the processors such that the bound of Theorem 3.11

is achieved and values for m_j and speeds for the processors such that the bound of Theorem 3.12 is achieved (assuming that k is small but varying permits us to ignore additive terms of k but not multiplicative terms in the performance bounds).

To achieve Theorem 3.12 is easy. Use $m_1=m-k+1$ and $m_i=1$ for $i>1$. Then the construction used in Section 3.2.2 provides an achievability result of a constant factor. □

To achieve Theorem 3.11, consider distributions of processors such that $m_j=m/k$ for each value of j (if $m\not\equiv 0 \pmod k$ then some of the m_j's are appropriately rounded off).

The precedence structure of the graph is the same as the graph of Figure 3.2.1 (see Figure 3.4.2). A node labelled with the integer j indicates that the task represented by the node is of type j. In this case, there are n blocks, each made up of $m-k$ pairs of tasks. The tasks of the first m_1-1 pairs are of type 1; the tasks of the next m_2-1 pairs are of type 2, etc. The time requirement of each task in the long chain is $\sqrt{m-k}$. The time requirement of the other $n(m-k)$ tasks is $m-k$. Type j has one processor with rate $\sqrt{m-k}$ and m_j-1 with rate 1.

An asymptotically optimal schedule proceeds as follows. At each point in time, the fastest processor of some type is executing some task on the long chain. Thus the execution of each task on the long chain requires unit time. To finish all tasks on the long chain requires time $n(m-k)$. Meanwhile, the rest of the processors execute the tasks that are not on the long chain. Once a processor begins executing one of these tasks, it takes $m-k$ units of time until it is completed. At that time, the processor begins executing the task that is in the same position in the next block. Thus, after the chain is

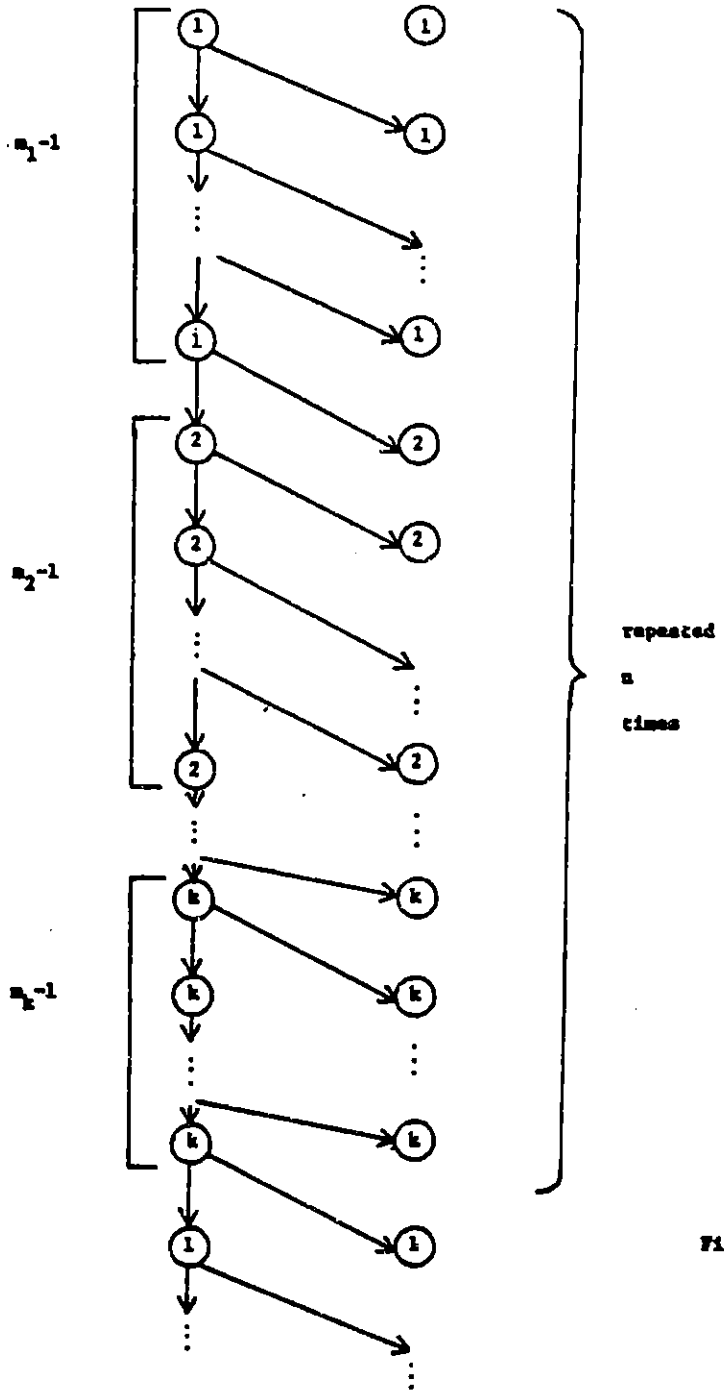


Figure 3.4.2

completed, at most an additional $m-k$ units of time are needed, for a finishing time of at most $(n+1)(m-k)$.

A bad list schedule on the fastest i_j processors of type j ($j=1,\ldots,k$) might proceed as follows. In fact, only the two fastest processors of each type would be used (unless $i_j=1$ for some j in which case only the fastest processor of that type would be used). While executing type j tasks, the schedule assigns the chain task to processor P_{j2} and the non chain task to processor P_{j1}. It takes times $\sqrt{m-k}$ to finish both of them (simultaneously). If only one processor of type j is to be used, then it first processes the non chain task, and then the chain task, again requiring at least $\sqrt{m-k}$ time units to finish each pair of tasks. Thus the total time required is at least $n(m-k)(\sqrt{m-k})$. This is $\sqrt{m-k}=\sqrt{k(m_j-1)}$ times worse than optimal. Since m is substantially larger than k, Theorem 3.11 is essentially achieved to within a factor of 2. □

There is a large spectrum of similar results that may be proved. It can be shown that Theorem 3.12 is achieved for a class of processor distributions that have $m_1=cm$ for some $c<1$. Similarly, Theorem 3.11 is achieved by a class of processor distributions that have $\max_j m_j=cm/k$ for a fixed $c>1$. These added constructions are trivial extensions of the above and are omitted.

The third type of result involves the situation that the m_i are relatively small (compared to k). In this case we will show that k times worse than optimal is achievable.

Consider Figure 3.4.3. There are m_i columns that informally speaking

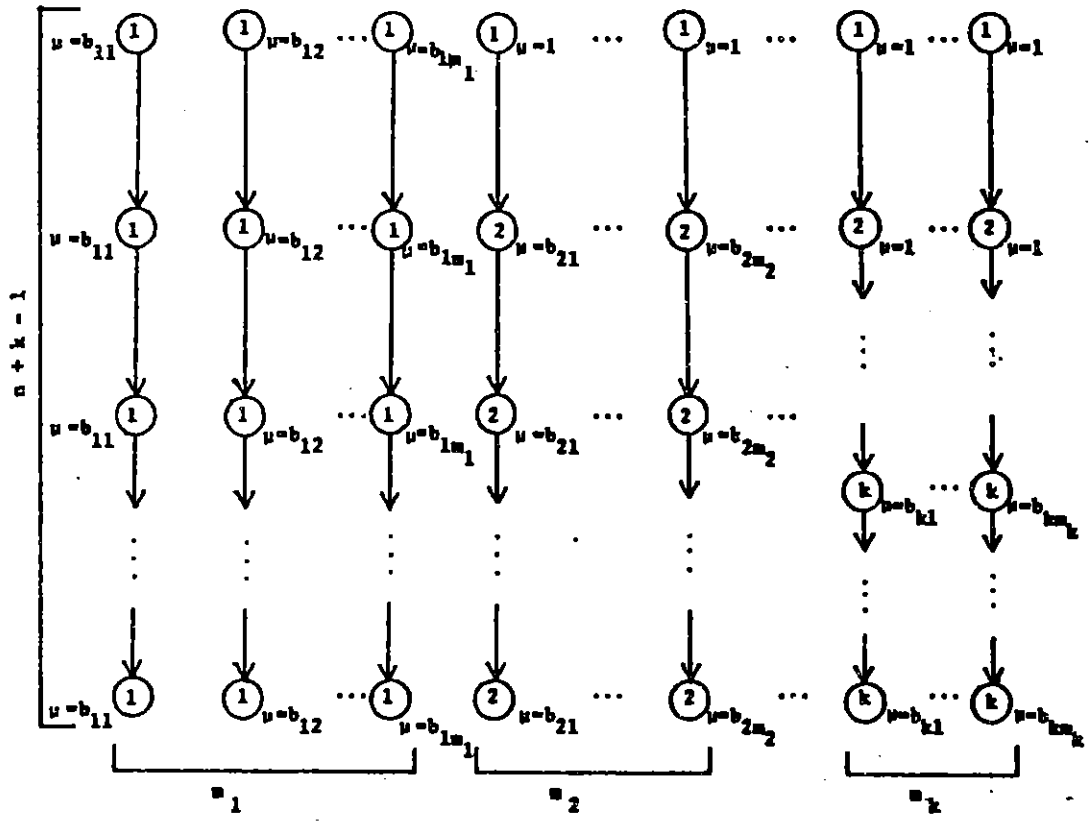


Figure 3.4.3

"correspond to type i". Each of these m_i columns contains a chain of $n+k-1$ tasks. The j^{th} task in each of these columns has $\nu(T)=j$ (for $j\leq i-1$) and $\nu(T)=i$ (for $i\leq j$). The notation $\mu=i$ means that the time required by the indicated task is the value i (note that the only values that appear are the integer 1 or the rate of a processor). An asymptotically optimal schedule first executes the first $k-1$ tasks of each column using an arbitrary schedule. Then only n units of time are required. The i^{th} remaining task in every column is executed i units of time later.

A bad schedule first executes only those tasks in the first m_1 columns. The best that could be done in that case (assuming all processors of type 1 are in use) is that these columns will be finished in time n. In a similar manner, it takes a minimum of time kn to finish the entire tasks system. Thus this schedule is at least k times worse than optimal. Thus for the class of processors considered, the example illustrates achievability up to a constant factor. □

The last achievability result is related to the case of list schedules (i.e. $i_j=m_j$). We have been treating this case separately, and we now present a bound that achieves Theorem 3.10 up to a constant additive factor (in fact, the difference between the theorem and what can be achieved is at most one).

To obtain this lower bound we combine the construction of Section 3.4.2.2 with a construction used in [42,43]. The result used from [42,43] is as follows. Fix a set of uniform non-identical processors $\mathcal{P}$, of one type. Then there are a set of ordinary task systems for $\mathcal{P}$ (with empty precedence relation) with the property that the performance ratio of list schedules relative to optimal schedules over this set of task systems is arbitrarily

close to $1+(b_1/b_m)-(b_1/B_m)$ where these quantities are as defined in Section 3.2.1.

Consider the task system of Figure 3.4.4. Diagramming conventions are as in Section 3.4.2.2. The notation $\mu=r(P_{ij})$ means that the time required for the task equals the rate of the processor P_{ij} (i.e., b_{ij}). A node labelled with B denotes a copy of one of the task systems in the set used to obtain the lower bound in [42,43] with the type of each task in this task system being m_k. The interpretation of an arrow between two nodes labelled with B indicates a precedence dependence of each task at the destination of the arrow on each task at the source of the arrow. The class of task systems described in the figure is parameterized by the variable n and the class of task systems described in [42,43]. Let n' denote the time required to execute B using an optimal schedule. Assume without loss of generality that $\max(\{(b_{j1}/b_{jm_j})-(b_{j1}/B_{jm_j}) | j=1,..,k\})$ is achieved by processors of type k.

An asymptotically optimal schedule first executes the first $k-1$ tasks of each column using an arbitrary list schedule. Then only n more units of time are required. It is clear how to finish the columns that correspond to the first $k-1$ types of tasks in n units of time. By using the optimal schedule for each occurrence of B each occurrence of B requires only n' units of time. Since there are n/n' copies of B only n units of time are required.

A bad list schedule spends $(k-1)n$ units of time completing the tasks that correspond to the first $k-1$ types of processors. It then spends arbitrarily close to $(n/n')(n')(1+(b_{k1}/b_{km_k})-(b_{k1}/B_{km_k}))$ units of time to complete the columns that correspond to the k^{th} type of processor using the

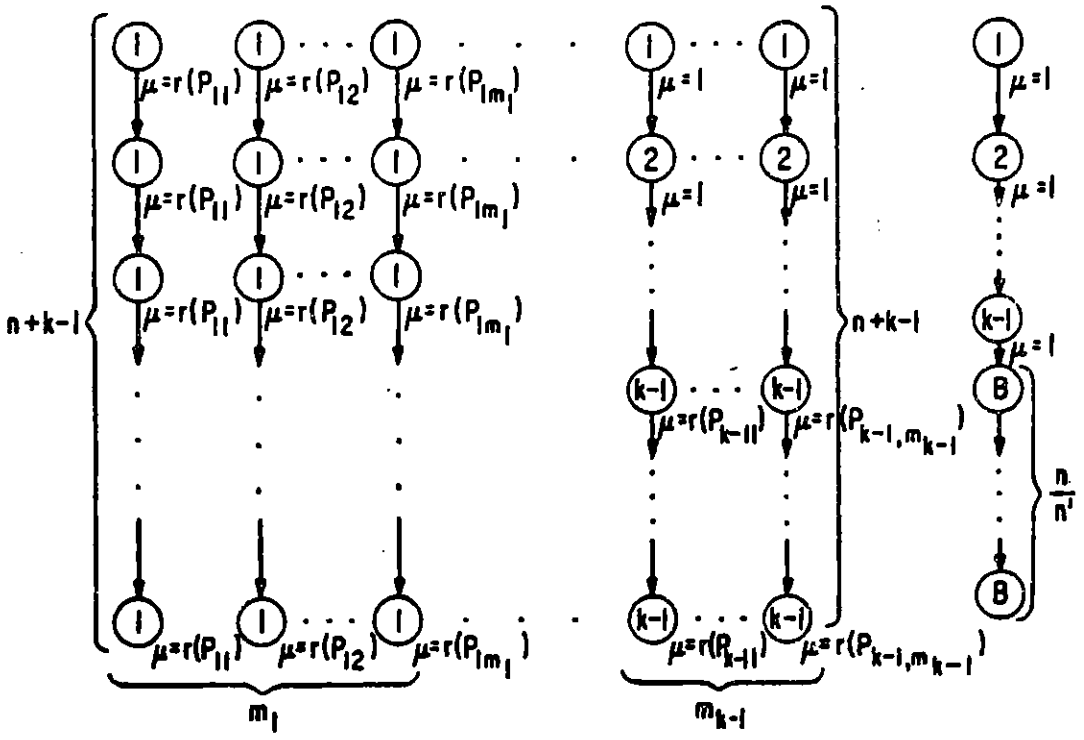


Figure 3.4.4

bad list schedule from [42,43]. The exact amount of time depends on which task system is used for the nodes labelled with B in the task system sketched in Figure 3.4.4. The ratio thus approaches $k+(b_{k1}/b_{km_k})-(b_{k1}/B_{km_k})$ for large n and B's whose variation in execution speed approaches a ratio of $1+(b_{k1}/b_{km_k})-(b_{k1}/B_{km_k})$. □

The gap between our upper and lower bounds on performance ratios is not very large. The gap is between $k+\max(\{b_{j1}/b_{jm_j}:j=1,..,k\})(1-\min(\{b_{jm_j}/B_{jm_j}:j=1,..,k\}))$ and $k+\max(\{(b_{j1}/b_{jm_j})-(b_{j1}/B_{jm_j}):j=1,..,k\})$. Recall that $q=\max(\{b_{j1}/b_{jm_j}:j=1,..,k\})$. Since both the upper bound and lower bound are between $k+q$ and $k+q-1$, for all practical purposes the result is tight.

3.4.4. Maximal usage preemptive scheduling of tasks on processors of different types and different speeds

This section discusses preemptive scheduling of typed task systems. The results are somewhat analogous to those of Section 3.2.3.2 in that we use the bound on list schedules on a subset of the processors. A better bound based on techniques such as those in Section 3.2.3.3 has not been devised. The preemptive schedules analyzed are the maximal usage schedules defined in Section 3.4.3.1 which encompass all schedules as in the untyped case.

A performance bound on maximal usage schedules is obtained by appealing directly to the results of Section 3.4.3. Fix a set of processors with speeds b_{ij}. Consider equation (33) in Section 3.4.3. It suffices to show that equation (33) (when interpreted as a bound on the performance of any maximal usage schedule) applies to any task system, any maximal usage schedule for the task system, and any set of indices $\{i_j\}$. From this, one may conclude that for any task system and any maximal usage schedule for the task system (with finishing time w), $w/w_{opt} \leq k+2\sqrt{\max_j (km_j)}$. Similarly, one may conclude that for any task system and any maximal usage schedule $w/w_{opt} \leq k + \sqrt{m_1} + \ldots + \sqrt{m_k} + \max_j \sqrt{m_j}$. This proof follows by applying the bound of equation (33) for the set of i_j's that minimize equation (33). For this set of i_j's, equation (33) is bounded by the above quantities (as shown in Section 3.4.3.3). Note that in this context w_{opt} represents the finishing time of the optimal *preemptive* schedule.

It suffices to show that w_{opt} satisfies the lower bound of Lemma 3.13 and w satisfies the upper bound of Lemma 3.14 (using the definition of height relevant to the set of chosen i_j's). The former is immediate, since the lower bound did not consider the fact that nonpreemptive schedules

were used.

To get the upper bound on w given in Lemma 3.14, break up all intervals of any maximal usage schedule into two types of intervals. One type of interval is when i_j processors of type j are being used for some j, and the second type is when i_j processors of type j are not being used for any j. Clearly, one may use at least i_j processors of type j for at most a total of μ/B_{ji_j} units of time. Also, the intervals during which i_j processors of type j are not used for any j must be height reducing intervals. These height reducing intervals decrease the height by a rate of at least one per unit time. Lemma 3.14 follows and thus one may conclude:

Theorem 3.13. Let $(\mathcal{T},<,\mu,\nu)$ as above. Let w be the finishing time of any preemptive maximal usage schedule, and let w_{opt} be the finishing time of an optimal preemptive schedule. Then

$$w/w_{opt} \leq \min(k+2\sqrt{\max_j (km_j)}, k+\sqrt{m_1} + \ldots + \sqrt{m_k} + \max_j \sqrt{m_j}).$$

Achievability may similarly be obtained by appealing to the constructions of Section 3.4.3.3. The "bad" list schedules on the i_j fastest processors of type j are also bad maximal usage preemptive schedules.

3.5 Open Problems and Further Work

We close Chapter 3 with a discussion of some open problems related to the scheduling results of this chapter. For surveys on scheduling theory, we refer the reader to [7,24].

It seems that once one considers processors of different speeds, it is very difficult to get very close to the optimal schedule. This is something that this author believes to be inherent. Thus for example, it seems highly unlikely that there are any polynomial time approximation algorithms that schedule partially ordered tasks on unrelated processors, which are always within a constant of being optimal. The most interesting open problem that we leave is to prove such a result.

As mentioned in Section 3.1, this and a number of the other scheduling problems considered are NP-complete problems [27]. A characteristic of these problems is that although it is widely believed that they are not solvable in polynomial time, this conjecture has not been proved. Indeed, if one could determine whether one of the NP-complete problems was or was not solvable in polynomial time, then one could similarly determine the complexity of all these problems [20]. Thus to prove that no polynomial approximation algorithm is within a constant of optimal, one would also have to prove that no polynomial time algorithm always produced the optimal schedule. This is widely believed to be quite a difficult problem [20].

However, even if one could not decide the complexity of the NP-complete problems, one could still give compelling evidence that there are no constant factor polynomial time approximations. For example, one could show that unless the NP-complete problems were indeed solvable in polynomial time, that no polynomial time algorithm could produce a solution that is within a constant

times the optimal. Similar results have appeared in different areas of scheduling theory. For example, unless the *NP*-complete problems are all solvable in polynomial time, no polynomial algorithm for scheduling partially ordered unit execution time tasks on identical processors can be better than 4/3 times optimal [40]. The techniques used for that proof do not generalize to a result such as "no algorithm is within a constant times optimal".

Section 3.2.4 tries to get at such a result from a different direction. It shows that for limited classes of scheduling algorithms, no algorithm, even one that does use exponential time, can be within $\sqrt{m}$ times worse than optimal. It would be interesting to explore other classes of scheduling algorithms, for example the class of schedules with limited preemption as described in Section 3.2.4.

Of course, there are many open problems of the form "find a better approximation algorithm for a particular problem". We feel that it would be quite difficult to improve on $O(\sqrt{m})$ times worse than optimal for preemptive or nonpreemptive scheduling of partially ordered tasks on uniform processor systems. On the other hand, it is quite likely that some algorithm (e.g. the level algorithm of [28]) improves the algorithms presented here by a constant factor.

If there is no partial order, there seems to be some hope of improving on $O(\sqrt{m})$, even in the unrelated processor case. One of the algorithms of [30] has not been fully analyzed (Algorithm D) and seems to be a likely candidate for better than $O(\sqrt{m})$ times worse than optimal behavior. Even if this algorithm fails, one might expect to find some other algorithm that does better than the algorithms considered in Section 3.3.

For scheduling partially ordered tasks on unrelated processors, one

might be able to adapt some of the techniques for uniform processors to the unrelated case, and to thereby obtain $O(\sqrt{m})$ approximations.

There are a number of technical questions that we leave open, related to tightening some of the bounds on the algorithms. First, there is the algorithm of Section 3.2.2, of using only a subset of the processors. The performance of the algorithm is asymptotic to $\sqrt{m}$, and thus the analysis is quite tight. Nevertheless, the fact that the exact performance may be determined for any fixed value of m leads one to hope that the low order terms in the performance of this algorithm may be determined.

All of the algorithms of Section 3.3 may have their analyses improved by a constant factor. The proof of Lemma 3.10 seems to be the cause for the lack of tightness. The complicated counting argument that is used necessarily throws away some information. Improving this counting argument is the most likely mechanism for improving the bound.

Finally, the results of Section 3.4.3, while tight (to a constant factor) are somewhat unwieldy. Recall that either the bound of Theorem 3.11 or the bound of Theorem 3.12 may be tighter, depending on the relative number of processors of each type. A more uniform treatment of the speed independent bound is needed.

4. Schemes

4.1 Introduction

In this chapter, we discuss a number of theoretical results related to the data flow programming models. There is a class of schemes, introduced as a mechanism for expressing parallelism, called data flow schemes. The main result is that these schemes, are as powerful as r.e. program schemes. We begin with some background on the motivating principles behind schematology.

Early researchers investigating the relative "power" or "expressiveness" of different programming constructs quickly determined that a comparison of the set of partial recursive functions computed did not adequately capture the differences between the different programming styles. Any reasonable set of constructs computes all partial recursive functions, and thus all constructs are equivalent.

A different technique has evolved for comparing programming constructs. The notion of a scheme has been introduced [29,46,49], which enables one to discern differences between the classes of functionals computable by different constructs. Essentially, in a scheme there are no defined operations and thus (for example) variables can not be used as counters. Rather, all function and predicate symbols are uninterpreted, and a "scheme" is a functional over the symbols. This approach turned out to be quite successful, as it provided a rigorous interpretation to intuitive ideas such as "recursion is more powerful than iteration" [50,58].

A hierarchy of program constructs has been developed [1,9]. It turns out that just as there is a notion of "all primitive recursive functions", there is an analogous thesis about a construct being equivalent to "all effective determinate functionals" or "all recursively enumerable (r.e.)

program schemes" [58].

Data flow schemes were introduced by Dennis [12] to serve as a model of programming constructs to be used for highly parallel, data driven computer architectures. The power of this basic construct has never been fully explored. Early researchers in this area felt that it would be worthwhile to sacrifice the full power of this construct in order to enforce programming disciplines that are similar to those found in conventional languages. To this end, well formed data flow schemes were introduced [13]. They are a class of data flow schemes that satisfy certain structural requirements. These requirements force subprograms of any given program to behave like "if-then-while" statements. Indeed it has been shown [41] that the "expressive power" of well formed data flow schemes is equivalent to the expressive power of flowchart schemes.

Although data flow programs are often written using the well formed constraint, this constraint does not reflect the capabilities of the data flow computer architecture [14]. It is thus worthwhile to evaluate the expressive power of the entire class of data flow schemes which more closely approximate the potential of such data driven, asynchronous architectures. The restrictions which give rise to well formed data flow schemes are a structure imposed from the outside, having little to do with machine architecture.

The main result of this chapter is that the class of data flow schemes is equivalent to the class of effective determinate functionals. One direction is immediate using usual programming techniques in the language of r.e. program schemes. The proof of the other half is greatly simplified by two insights which are of interest in their own right.

The first is a programming technique using the language of data flow

schemes. A general translation lemma is proved (using quite simple techniques) which characterizes a class of boolean functions computable by data flow schemes but not computable by well formed data flow schemes. This lemma provides much of the machinery to prove later results.

The second insight is a theoretical result of interest. A very restrictive version of data flow schemes can simulate Turing Machines. This result implies various undecidability results about simple data flow schemes. The simulation of Turing Machines (TM's) follows immediately from the translation lemma using certain coding techniques.

In Section 4.2 r.e. program schemes, data flow schemes, and restricted data flow schemes are defined. Section 4.3 illustrates the programming techniques alluded to above. Using these techniques Section 4.4 discusses the simulation of TM's with restricted data flow schemes. Section 4.5 contains the proof that any r.e. program scheme can be simulated by a data flow scheme. Combining the programming techniques of Section 4.3 and the Turing Machine simulation of Section 4.4, this final simulation is not too hard to develop since r.e. program schemes obtain much of their power from a finite state TM control. Once restricted data flow schemes are shown to simulate TM's, they provide the necessary TM control.

4.2. Syntax and semantics of schemes

4.2.1 R.e. program schemes

To define r.e. program schemes it is helpful to first define the components.

Any particular r.e. program scheme may have infinitely many *variable symbols* $x,y,u,v,\ldots$, finitely many *function symbols* $f_1,\ldots,f_r$ (from an infinite function symbol alphabet), and finitely many *predicate symbols* $p_1,\ldots,p_s$ (from an infinite predicate symbol alphabet). Associated with a function symbol f (or predicate symbol p) is a number $arity(f)\in N$ which specifies the number of *arguments* needed by f. In the alphabet there is also a symbol *HALT*. Uninterpreted constants are thought of as 0-*ary* function symbols.

A *term* is:

(1) a variable x.

(2) a 0-*ary* function symbol f.

(3) the sequence $f(x_1,\ldots,x_n)$ where f is a function symbol of *arity* n, and $x_1,\ldots,x_n$ are variable symbols.

(4) the sequence $p(x_1,\ldots,x_n)$ where p is a predicate symbol of *arity* n, and $x_1,\ldots,x_n$ are variable symbols.

Terms of types (1), (2), and (3), are called *functional terms*. Terms of type (4) are called *predicate terms*.

A *statement* is:

(1) a *simple assignment* $x\leftarrow t$ where t is a functional term and x is a variable.

(2) a predicate term.

(3) the symbol *HALT*.

An *r.e. program scheme*, P, is an infinite list of statements (indexed by

the integers), a finite set of variable symbols $\{x_1,...,x_n\}$ called *input variables*, a finite set of variable symbols $\{y_1,...,y_k\}$ called *output variables* and two recursive functions. The first recursive function takes as input a statement number and produces the statement as output. The second function $r : \mathbb{N} \times \{0,1\} \rightarrow \mathbb{N}$, intuitively, specifies the successor statement of a given statement as explained in Section 4.2.2.

4.2.2 Semantics of r.e. program schemes.

For brevity the presentation of this section will be a bit informal. For a more formal treatment for the semantics of schemes one may consult [25].

An *interpretation*, I, for a r.e. program scheme P consists of a domain D, an assignment of a total function from D^n to D to each *n-ary* function symbol, an assignment of a total predicate on D^n to each *n-ary* predicate symbol, and an assignment of an element of D to each input variable. A *program* is a pair (P,I).

The *computation* of a program (P,I) proceeds as follows. The first statement executed is statement 0 (the meaning of "executing a statement" should be understood - it involves updating variables or evaluating predicates). Assume that statement i has just been executed. The next statement to be executed is given by the function r. Specifically, if statement i was a *HALT* statement, then the program is said to *terminate* and the *outputs* of the program are the current values of the output variables. If statement i was a predicate and the result of the predicate was $j \in \{0,1\}$, then the next statement to be executed is $r(i,j)$. If statement i was not a predicate then by convention, the next statement is $r(i,0)$. By convention, if

the execution of a statement involves evaluating an undefined variable, the program "loops", i.e., it never halts.

4.2.3 Data Flow Schemes

A *data flow scheme* is a labelled (finite) directed graph (with self-loops and multiple arcs). The labels of the nodes of any particular data flow scheme come from the following alphabets:

(1) An alphabet of function symbols $f_1, f_2, \ldots$

(2) An alphabet of predicate symbols $p_1, p_2, \ldots$

(3) The *gate* symbols "T", "F", "T F". (The gate labelled with "T F" is called a *Merge* gate.)

If a function symbol of arity n labels a node then there are n incoming arcs to the node. (Also, arity("T")=arity("F")=2 and arity("T F")=3).

Each arc in the graph is either a *boolean* arc or a *value* arc. (During execution each boolean arc has associated with it a word $w \in \{0,1\}^*$ and each value arc has associated with it a word $w \in D^*$ where D is a domain supplied by the interpretation). Each arc that leaves a node labelled with a predicate is a boolean arc. Each gate has a designated "control" boolean arc. (In the various figures, the control arc is labelled with a "c".) All arcs entering or leaving a node labelled with a function symbol are value arcs and all incoming arcs to a node labelled with a predicate symbol are value arcs. All of the noncontrol arcs that enter or leave a gate must be of the same type (but may be either boolean or value arcs). (Examples of various data flow schemes may be found in Figures 4.3.1-4.3.8. Their semantics are defined in Section 4.2.4.)

An *initialized data flow scheme* consists of a data flow scheme with an

assignment of a word $w \in \{0,1\}^*$ to each boolean arc and an assignment of the empty word of D^* to each value arc.

A *restricted data flow scheme* is an initialized data flow scheme whose nodes are only labeled with "T", "F", and "T F".

Certain uninitialized arcs are designated *input arcs*. Certain arcs are designated as *output arcs*. It is sometimes convenient to allow two additional labels for nodes. *INPUT* nodes have no incoming arcs, and *OUTPUT* nodes have no outgoing arcs.

4.2.4 Semantics of data flow schemes

An *interpretation* supplies a domain D, assigns functions to function symbols, and predicates to predicate symbols as in Section 4.2.2. Also, each input arc is assigned an input word from its value domain. A *program* is a scheme with an interpretation.

A node is said to be *enabled* if each of its incoming arcs has a non empty word associated with it. For *Merge* gates, the definition is slightly different: a *Merge* gate is enabled if the first symbol of the data word associated with its control input is T and its "T" arc is non empty or if its control input is F and its "F" arc is non empty. Also, *INPUT* and *OUTPUT* nodes are never enabled.

When an enabled node *executes*, the first symbol (value) of the word associated with each incoming arc is removed and the function or predicate labelling the node is applied to the values represented by these symbols. Next the result of the function or predicate is concatenated to the end of the data word associated with each outgoing arc. For T gates, if the control arc is T, then the gate is the identity on the other incoming arc, and if the

control arc is *F* then the first symbols are removed and nothing is placed on outgoing arcs. *F* nodes are the same as *T* nodes with the role of the control arc complemented. For *Merge* gates, if the control arc is *T*, the gate is the identity on the "*T*" arc and the "*F*" arc is unaffected. Analogously, if the control arc is *F*.

At any step in execution there may be many enabled nodes. There is no notion of what "must" happen in one step, as the model does not completely specify this. There is a notion of what may happen in one step. The work of [51] implies that this incomplete specification does not change the output of computation (we will momentarily define the output). At one step the data flow scheme executes any number (greater than or equal to one) of its currently enabled nodes. The execution updates some or all of the arcs, by removing old values from the data words associated with arcs that lead to executing nodes, concatenating result values to the associated data words of arcs that emanate from executing nodes, doing both operations to arcs that connect two executing nodes, and leaving the rest of the arcs unchanged.

A data flow program *terminates* when no node is enabled. At that time the words associated with the output arcs are the *outputs* of the data flow program. (It is convenient to assume that formally, all constant functions have one incoming arc, i.e., take one argument for otherwise a data flow program would never terminate.)

Note that as defined above, data flow schemes do not have a fixed number of inputs or outputs. To compare data flow schemes to r.e. program schemes, we extend the definition of data flow schemes to include a set of integers which specify how many input symbols are to be supplied on each input arc and how many outputs are to be produced on each output arc. We only

consider computations where the proper number of inputs is supplied. If at termination an output arc is supposed to supply k outputs, then the first k values are the outputs. If there are fewer than k outputs then the remaining outputs are undefined.

It is important to note at this point that we rely heavily on the fact that the queues are infinite. It follows from the pebbling argument of [50] that no class of schemes that does not have the potential to store an unbounded amount of information can possibly simulate recursive schemes, and certainly not the entire class of r.e. program schemes. In the simulation, it will be pointed out where the assumption of infinite queues is used.

With this in mind, it is interesting to refer back to the results of [41] which analyze the expressive power of the well formed data flow schemes. In that case, the expressive power of well formed data flow schemes does not depend on the size of the queues. That is, even infinite queues do not increase the expressive power of well formed data flow schemes. Even in the well formed case, however, there are differences that arise with different sized queues; but these are not differences in expressive power. For example, if one assumes that in one step all enabled nodes are executed, it can be shown [31] that well formed data flow schemes with large queues may operate considerably faster than with small queues.

4.3. Programming techniques

There are two goals of this section. The first is to show that restricted data flow schemes have enough power to define boolean functions (NOT and OR). This will be needed, as these functions play a role in subsequent results.

The second task is to prove the lemma alluded to in Section 4.1. This is a more substantial indication of the programming power of restricted data flow schemes. It is a useful tool in shortcircuiting the detailed programming needed to simulate TM's by data flow schemes.

4.3.1 Boolean operators

To define NOT and OR, consider the initialized restricted data flow schemes of Figures 4.3.1 and 4.3.2. In Figure 4.3.1, a True value on arc *A* chooses out the "*T*" arc of the *Merge* (i.e. the value False). A similar arrangement for *A* having False implies *B*=NOT(*A*). (The diagramming convention for arc *A* is an arc emanating from a node labelled with an *A*.)

In Figure 4.3.2, when *A* is true, then *A* (i.e. True) is output, and when *A* is false then *B* is output. Thus *C*=*A* OR *B*. Note that in both programs the initial configuration is restored after one usage. Thus values may be pieplined through these circuits, and the NOT circuit will always complement its input, and the OR circuit will always "OR" a pair of inputs.

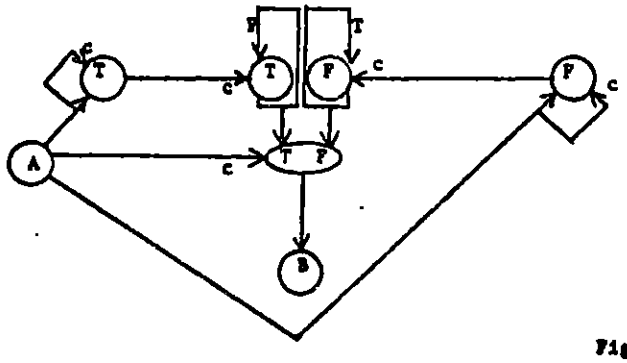


Figure 4.3.1

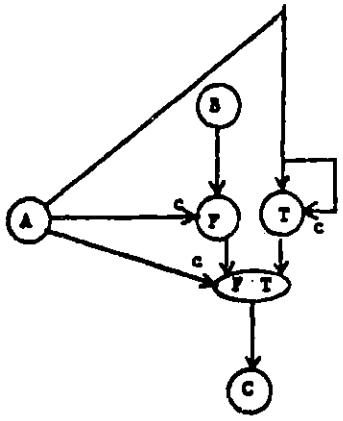
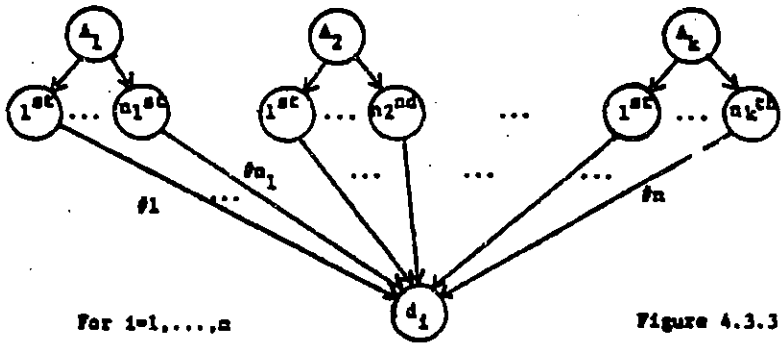


Figure 4.3.2



For i=1,...,n

Figure 4.3.3

4.3.2. Finite translation Lemma.

Lemma 4.1. Let $g:\{0,1\}^n \to \{0,1\}^m$, let $n_1+\ldots+n_k=n$ and $m_1+\ldots+m_l=m$. Then g is computable by a restricted data flow scheme with k input arcs and l output arcs where the input/output conventions are as follows. The n inputs are presented in the following way: the first n_1 of them appear on the first input arc, the next n_2 on the second input arc ..., and the final n_k on the k^{th} input arc. The outputs occur in the following configuration: the first m_1 of them appear on the first output arc,..., the last m_l of them appear on the l^{th} output arc. Also, for any such input to g, at termination the configuration of the data words on the arcs of the data flow scheme matches the initial configuration of the data words on the arcs.

Proof The following explains certain abbreviations used in the data flow schemes described in the proof (see Figures 4.3.3 and 4.3.4). The expansion of these abbreviations is discussed in Section 4.3.3.

1. At times a few nodes are used together labelled 1^{st}, 2^{nd},..., i^{th}. This abbreviates a program where each node outputs one value for every i inputs. The node labelled j^{th} prints out the j^{th}, $i+j^{th}$, $2i+j^{th}$ (etc.) inputs.

2. A labelled arc that is drawn as a self-loop (and does not emanate from a node) denotes that the labels (initial values) of the arc constantly circulate around. Thus there is an infinite supply of those values. While it may seem that a data flow program with such self-loops never terminates, the expansion of this abbreviation discussed in Section 4.3.3 does in fact terminate.

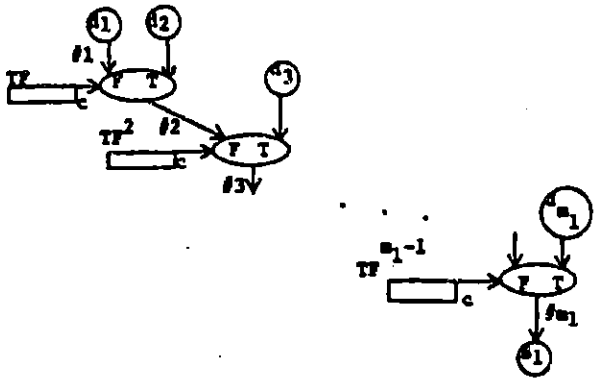


Figure 4.3.4

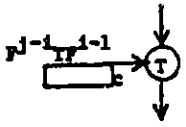


Figure 4.3.5

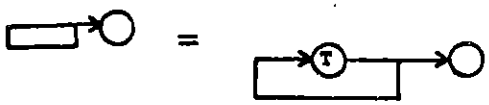


Figure 4.3.6

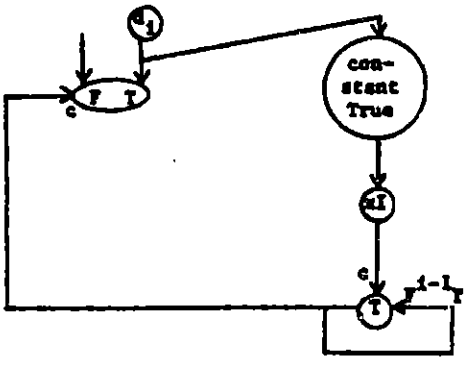


Figure 4.3.7

Now, consider Figure 4.3.3 which gives the first part of the desired restricted data flow scheme. For each set of n inputs, arc #j obtains the j^{th} input. Thus the n inputs are "parallelized" onto n different arcs. The node labelled d_i is an abbreviation for an acyclic graph of boolean operators which produces the i^{th} digit of the m digit result. Note that all arcs are restored to their initial data words when the computation is completed.

For each of the l output arcs there is a program segment like the one in Figure 4.3.4. The figure denotes the program segment for the first output arc. The arc labelled d_i is 1 if the i^{th} of the m_1 results should be a 1. Arc #1 thus obtains the first of the m_1 results that will be output on arc B_1 (the first output arc), arc #2 obtains the first two of the m_1 results, ..., and arc #m_1 obtains all m_1 results. Note that every arc has the same data word associated with it before and after each execution of the program. The self-loops are discussed in the next section. □

It is important that the initial configuration matches the final configuration. The program is thus reusable in the sense that if g must be applied repeatedly on a sequence of inputs, the same program may be used for each set of n inputs. Note also that the lemma is clearly false for well formed data flow schemes due to their well behaved characteristic [51].

4.3.3. Programs for the abbreviations

Figure 4.3.5 is the expanded version of the nodes labelled with i^{th}. If a node desires to choose the i^{th} of j values, it merely absorbs all but those of the form $nj+i$. This is accomplished by circulating around a control of $F^{i-1}TF^{j-i}$ to a T gate as in the figure.

Figure 4.3.6 indicates the definition of an arc that is drawn as a

self-loop. The arc actually represents two arcs leaving a T gate. One arc leads to the destination of the arc in the original program, and the other leads back to the input of the T gate. To control this T gate the program generates as many True values as needed. In particular, for the control inputs to the *Merges* of Figure 4.3.4, one would like i control values to the T gate for each value on d_i.

Figure 4.3.7 indicates how to accomplish this. The node labelled xl expands one value on the input to l values on the output. Figure 4.3.8 is the $x13$ program and it is easy to generalize this to the construction of the program for xl for any l.

The self-loops of Figure 4.3.5 are easier to handle and are left to the reader.

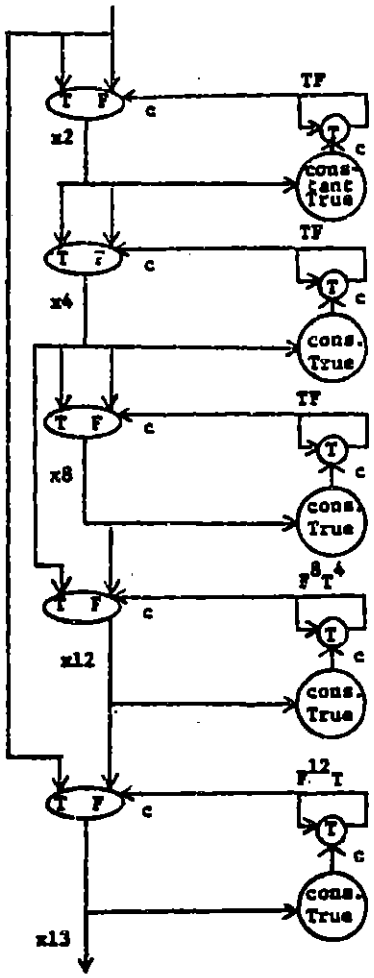

TF
T F
c
x2
T
c
constant True
TF
T F
c
x4
T
c
cons. True
TF
T F
c
x8
T
c
cons. True
F8T4
T F
c
x12
T
c
cons. True
F12T
T F
c
x13
T
c
cons. True

Figure 4.3.8

4.4. Simulating Turing Machine computations with restricted data flow schemes.

The object of this section is to present a simulation of one tape Turing Machines by restricted data flow schemes. (Simulation in this chapter is related to the usual notion of TM simulation and is not related to the notion of simulation defined in Chapter 2). The input to any given one tape TM is given on the single tape, the read head initially scans the leftmost symbol of the input, and the rest of the tape is blank. The data flow scheme will simulate the TM by keeping on one of its arcs, a representation of the TM tape. In Figure 4.4.1 (at the end of Section 4.4) this data arc is labelled arc *C*. Note that the arc will not always have a representation of the TM tape. However, if one observes the sequence of symbols that pass through the arc, the sequence will be a coded form of successive instantaneous descriptions (i.d.'s) of the TM computation. We assume familiarity with TM computations [26].

4.4.1. Representation

The data flow simulator represents an i.d. of the Turing Machine computation by coding it into binary. Let Σ be the tape alphabet of a given TM and Q be the set of states of the TM ($\Sigma \cap Q = \emptyset$). Let $\$ \notin \Sigma \cup Q$ be a "delimiter". Let $r = \lceil \log_2(|\Sigma|+|Q|+1) \rceil$. Then the data flow simulator represents an i.d. with respect to a fixed coding of the symbols of $\Sigma \cup Q \cup \{\$\}$ as r-tuples.

Let $f: \Sigma \cup Q \cup \{\$\} \rightarrow \{0,1\}^r$ be a 1-1 map. Assume that during a particular TM computation, after a number of steps the active portion of the TM's tape is $w_1 \cdots w_k$ with $w_i \in \Sigma$ (the active portion of the tape consists of those squares already traversed), and that the machine is in state q reading the symbol w_j. Then the *f-code* of the TM i.d. at that step is given by

f(i.d.)=$f(w_1)\cdot f(w_2)\cdots f(w_{j-1})\cdot f(q)\cdot f(w_j)\cdots f(w_k)\cdot f(\$)$. Note that the r symbols in the f-code that appear before the coded version of the symbol currently being scanned, is the coded form of the state and the i.d. is delimited with $f(\$)$ at the end.

The f-code of the initial i.d. is f(start-state) followed by the coded versions of the input to the TM, followed by $f(\$)$. If the entire tape is initially blank then the f-code of the initial i.d. is f(start-state)$f(\flat)f(\$)$ where $\flat$ is the blank symbol. The f-code of the computation of a given TM on an input, is the concatenation of the f-codes of the individual i.d.'s (starting with the initial i.d.). For convenience, assume that any TM has infinitely many i.d.'s. The successor i.d. of a halt i.d. is the same i.d.

Assume without loss of generality that in one step a TM will either move left, move right, print $\sigma\in\Sigma$, read the symbol being scanned and branch to a TM state based on what was read, or halt.

If the TM is reading the leftmost active square and moves left, or the rightmost active square and moves right, then the simulator must know to expand the size of its representation of the TM's tape. (If the TM head moves right and ends up scanning a square not yet activated, the data flow simulator does not activate the scanned square until that square is written on or until the read head moves to the right of it.) Otherwise, to update an i.d., only a few symbols must be changed, with none added.

4.4.2. Description of simulation

Definition. A restricted data flow scheme D with input arc C, *simulates* (with respect to f) a TM M if for any input to M, the infinite sequence of symbols passing through C is the f-code of the computation of M on that input,

where the input to D is the assignment of the f-code of the initial i.d. of M to the arc C.

Theorem 4.1. For any TM M, there is a restricted data flow scheme D that simulates M.

Proof. Consider Figure 4.4.1 which schematically illustrates the data flow simulator for a given TM. Assume that the simulator has the f-code of an i.d. of the TM on arc C and all other arcs in Figure 4.4.1 are blank. It suffices to show that the next sequence of symbols to be added to C by the program is the f-code of the successor i.d. of the current i.d. and that furthermore the data words on all other arcs are restored to their current values. Then, to prove the theorem, the program represented by Figure 4.4.1 will be used, with input arc C. Note that the nodes of the data flow graph will not necessarily execute at the times that are assigned to them in this discussion, since the models does not specify when nodes must execute. Due to the determinacy of data flow programs [51], this does not effect the sequence of symbols that pass throught arc C. Our program will have the property that it is impossible to indefinitely prevent the execution of a node without causing the program to terminate while there are still executable nodes.

Arc C feeds into different arcs denoted $C_{-2}, C_{-1}, C_0, C_1, C_2$. Initially, all five arcs have the same associated word, except that C_{-2} has an extra $2r$ symbols ($f(\$)f(\$)$), C_{-1} has an extra r symbols ($f(\$)$), C_1 is missing the first r symbols of the i.d. and C_2 is missing the first $2r$ symbols of the i.d.

Technically, to fit the definition of simulation, the input may only

appear on a single arc. The way that we provide input to the other arcs is as follows. Arcs C_{-2} and C_{-1} are initialized to $f(\$)^2$ and $f(\$)$. There are "identity boolean function" nodes which separate arc C from each of arcs C_{-1} and C_{-2}. This provides arcs C_{-1} and C_{-2} with the desired input. Similarly, it is left as an exercise to remove the first r symbols to get the desired value on arc C_1. It is not hard to do if one uses a T gate with an initial control of r 0's. Care must be taken to insure that no further symbols are deleted.

We now describe how an i.d. is updated to get the next i.d. We will describe the update for each symbol on the active portion of the TM tape, as well as the two other symbols of the i.d. Box #1 of Figure 4.4.1 updates the current i.d. by processing r symbols from each arc at once. If the first r symbols on arc C_0 are the representation of the contents of a TM square, and the square is "far" from the square currently being scanned by the head, then these r symbols are copied to the next i.d. Similarly, if these r symbols are $f(\$)$ and the TM head is far from the extreme left or extreme right of the active portion of the tape, then $f(\$)$ is copied.

One might wonder how it is possible for the data flow scheme to know whether the head scans a nearby square. This information is contained on arcs $C_{-2},\ldots,C_2$. The way that arc C_i is initialized and the way that the arcs are processed guarantees that at all times the first block of r symbols on arc C_i is the block that will appear i blocks later in the i.d. then the current block of C_0. (If i is negative, then it is the block that appeared $-i$ blocks earlier.) As will presently be discussed, the changes in the i.d. may be determined from these five blocks of r symbols.

To finish the discussion of the updating, the following is done if the

read head is near the TM symbol coded by the r bits on arc C_0. First consider the case that the i.d. representation need not be expanded (due to the exploration of new squares on the extreme right or left of the tape). If the square coded by the block of r symbols on C_0 is the predecessor of the square being scanned (i.e., the TM state is coded on C_1), and the TM is in a left move state, then the r symbols output by box #1 are the r symbols of the representation of the next TM state. If the r symbols on arc C_0 are the code of a left move state, then the output is the first r symbols on arc C_{-1}. Similarly, it is easy to see how to update the i.d. for any type of TM state. For example, if arc C_0 contains the code of a read and branch state, then the output is the code of the new state based on the first block of r symbols on C_1 (i.e., the TM symbol being scanned). If arc C_0 is the code of a print state then the output is the code of the next state. If arc C_{-1} is the code of a "print σ" state, then the output is $f(\sigma)$. A right move is similar to a left move except that the location of the state symbol is interchanged with the *next* symbol of the f-code of the i.d.

The final cases to consider are the cases of a left move onto a new square, a right move onto a new square, and a print onto a new square. It is to handle these cases that infinite queues are needed on the arcs. For to simulate the TM on various inputs may require exploring an unbounded number of TM squares, and thus unbounded storage is needed in the data flow scheme. A left move onto a new square is recognized when C_{-2} has $f(\$)$ and C_{-1} has the code of a left move state. In that case, the output is $f(\not\flat)\cdot g$ where g is the input on arc C_0 ($\not\flat$ is the new leftmost symbol). If C_0 has $f(\$)$ and C_1 has the code of a left move state, then the output is $f(\$)$, and if C_0 has a left move state and C_{-1} has $f(\$)$ the output is the code of the next state. In the

right move case, if arc C_0 has the TM state and C_1 has $f(\$)$, the output is $f(\$)$ followed by the code of the next TM state. In the print σ case (where the square to be printed is not yet in the active portion of the i.d.), the block $f(\$)$ is updated to $f(\sigma)\cdot f(\$)$. Note that in these cases $2r$ symbols are output, whereas in the earlier cases only r symbols need to be output.

For conformity to the hypothesis of the finite translation lemma, it is convenient to assume that box #1 always gives the same number of results on each arc for each set of 5 input blocks. Thus, box #1 always prints out $2r$ symbols on arc A. Arc B consists of 1^{2r} when all $2r$ symbols on arc A are desired, and $0^r\cdot 1^r$ if only the last r are desired. In the latter case, the first r on arc A are arbitrary (included for convenience), and the last r are the desired r symbols. The irrelevant symbols are deleted in box #2.

As far as the actual program is concerned, there is not much to add. Careful inspection of the specifications of box #1 indicates that it fits the hypothesis of the finite translation lemma, and thus a program exists for it. The program for box #2 is trivial, and is given in Figure 4.4.2. □

4.4.3. Other results about restricted data flow schemes

We discuss a number of corollaries of Theorem 4.1. The above discussion has not considered the results of computation in the case that the "TM" has a notion of output. This problem has been ignored since the main motivation for our simulation has been to help prove that data flow schemes are as powerful as arbitrary r.e. program schemes. For that purpose, showing that a data flow scheme can simulate the control structure of a TM is sufficient.

A brief description of a possible convention to make the data flow scheme halt will be discussed. Whenever C_0 has f(halt-state), box #1 outputs

f(halt-state). However, if C_{-1} has f(halt-state) then the r symbols that appear on arc C_0 are deleted (i.e. arc B consists of 0^{2r}). Ultimately, one of the C_i becomes void, causing the data flow program to terminate.

Now assume that box #1 also contains two additional outgoing arcs (arcs D and E) where D ordinarily produces 0^r and instructs a T gate to delete r symbols output on arc E. When the halt i.d. is reached, at each step that the data flow simulator deletes r symbols, arc D becomes 1^r, permitting the r deleted symbols (that are now sent to arc E) to be preserved. Thus the tape contents to the right of the read head are preserved during this process.

Now, assume that the desired output convention is that the sequence of symbols until the first ₿, is the output. The symbols that have just been deleted from the i.d. representation are the input to another data flow program which outputs its input until it sees f(₿) or f(\$). From that time on, no output is produced. Such a program may be constructed and is left to the reader.

There are a number of undecidability results for restricted data flow schemes that follow from Theorem 4.1. We mention one as a typical example:

Corollary The problem of deciding for a given arc of a given initialized restricted data flow scheme and a given finite sequence of 0's and 1's, whether the sequence ever appears on the arc is undecidable.

Proof. If we could decide the above problem, we could decide the halting problem as follows. Code the halt state of each TM as r 1's, and code all other states with a string in $0(0+1)^{r-2}0$. For such restricted data flow schemes, 1^r occurs iff the associated TM halts.

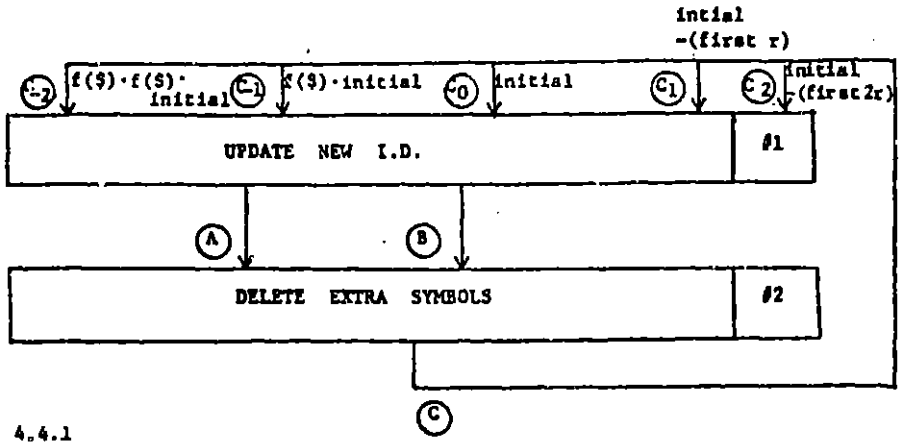


Figure 4.4.1

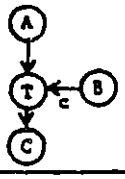


Figure 4.4.2

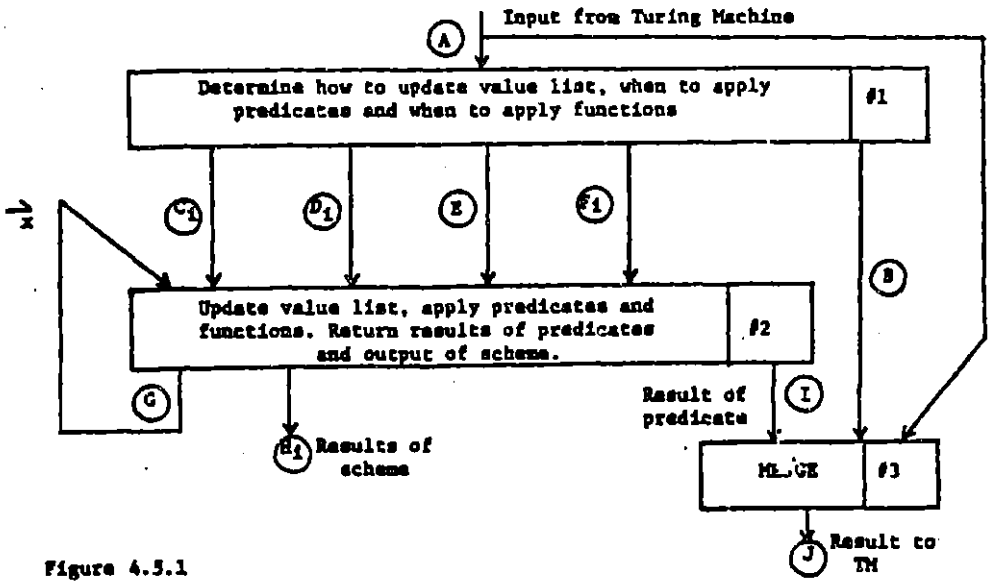


Figure 4.5.1

4.5. Simulation of an arbitrary scheme with a data flow scheme

The data flow scheme that we use to simulate a given r.e. program scheme will use a Turing Machine (as simulated in Section 4.4) as a subroutine. The TM that is chosen is one that is related to the r.e. program scheme in the following way. Given the current statement number of the program scheme and (where relevant) the results of predicates the TM has the capability to generate the next statement number as well as the function or predicate that the r.e. program scheme applies next.

The TM interfaces with a data flow program called "the scheme simulator". This program contains the current values of the variables of the r.e. program scheme (that have already been defined) on a "value list" (one of its arcs), and also has nodes labeled with the uninterpreted function and predicate symbols. The TM instructs the scheme simulator when to use these nodes and how to update the value list. The value list is maintained by the scheme simulator as a circular buffer, i.e., when a particular value is needed the other values are circulated around until the desired value reaches the beginning of the buffer (which we refer to as the "top of the value list"). In order to know how to instruct the data flow scheme in its circulation of the value list, the TM maintains an "association table" which keeps track of which variables occupy which positions in the scheme simulator's value list at any particular time.

4.5.1. Operation of the Turing Machine.

Initially, the Turing Machine has a blank tape. The first task of the TM is to initialize the association table which is stored as data on the TM tape. If the n input variables to the scheme are $x_1,...,x_n$, the table contains

information that "for i=1,...,n, the i^{th} value on the scheme's value list is x_i".

Assume that the TM has the number of the last statement executed (initially 1) written on its tape (referred to as i). Assume also that if statement i was a predicate, that the result of the predicate is currently in the finite state memory of the TM. The TM then computes the number of the successor of statement i in the program scheme using the recursive function that generates the next statement number. (Note that the association table must be left intact in the process.) The Turing Machine then computes the type of operation required by $s(i)$, the successor of instruction i (i.e., the TM computes the "contents" of the statement numbered by $s(i)$). Assume that all variables required as input to the operation have already been defined (i.e. appear in the table), otherwise, the TM loops.

Let the operation required by statement $s(i)$ be $Z(y_1,...,y_k)$. The TM now instructs the scheme simulator where to get $y_1,...,y_k$ from, which function or predicate is referred to by Z, and where to place the result. First the TM sends a message that tells the scheme simulator what to do with the top of the value list. Specifically, if it equals y_i for some i, the message is "Let the top value be used as the i^{th} input to the operation Z and circulate the top value to the end of the value list". If it equals none of the y_i then the message is "Circulate the top value to the end of the list". At this time the TM updates its association table so it knows which value on the list refers to which variable.

The way that messages are sent is by causing specific "message symbols" to appear on the TM tape for exactly one i.d. There are finitely many possible messages, and each is coded as a different symbol of the TM's tape alphabet.

These symbols are reserved symbols used only for this purpose and are only printed by the TM when messages are to be sent. The step following the printing of a reserved symbol, is always a print σ for some nonreserved symbol σ. Thus the same "message" does not exist in two successive i.d.'s. The motivation for this will become clear when the operation of the scheme simulator is discussed. Basically, the idea is that the scheme simulator will look at each symbol of each i.d. exactly once and thus will see the message symbol exactly once.

The TM continues sending messages to the scheme simulator, until all the inputs to Z have been defined. After Z has received all inputs, subsequent messages take on one of five forms.

(1) If Z is a function symbol, and statement $s(i)$ is of the form $x_n \leftarrow Z(y_1,\ldots,y_k)$ where x_n has not previously been defined, then the message sent is "Add the result of Z to the list of values".

(2) If the statement is a function application $x_n \leftarrow Z(y_1,\ldots,y_k)$ and x_n has already been defined, then if x_n is the top value on the list, the message says "Replace the top value with the result of Z".

(3) In the case of a function application to x_n where x_n has been defined but is not the top of the list, the message is "Circulate the top of the value list to the end of the list". In this case, the TM then proceeds to determine if the new top of the value list is x_n.

In all of the above cases, the TM updates its association table.

(4) If Z is a predicate, then two messages are sent. The first message is "Obtain the result of the predicate Z" which instructs the scheme simulator which predicate is to be evaluated. Then, a second message called a "return message" is sent that instructs the scheme simulator to "Return the result of a

predicate to the TM". The return is accomplished as follows. The "return message" sent by the TM is a specific symbol $\sigma \in \Sigma$ like all other messages. Let $\sigma' \in \Sigma$ be a reserved symbol of Σ whose code differs from that of σ in exactly one bit. (Assume that σ has a 0 in that place, and σ' has a 1 in that place.) Then the scheme simulator leaves σ unaffected if the predicate was false, and changes the code of the message from σ to σ' if the predicate was true. This is the only time that the scheme simulator modifies the TM i.d., and the TM remembers the modification in its finite state control.

(5) If Z is a *HALT* instruction then getting the values for $y_1,...,y_k$ (in the above discussion) is all that is necessary, and the scheme simulator outputs those values.

After operation $s(i)$ is completed, the TM continues to $s(s(i))$. If $s(i)$ is a *HALT* operation, then after $s(i)$ is completed, the TM halts. Using techniques similar to those of Section 4.4.3, it is easy to see how to make the data flow version of the TM halt.

4.5.2. Operation of the scheme simulator

The outline for the program of the scheme simulator is given in Figure 4.5.1 (above). The input to the scheme simulator consists of a queue of n values corresponding to the r.e. program scheme being simulated (on arc G in Figure 4.5.1). The scheme simulator uses the TM described in Section 4.5.1 as follows. Consider Figure 4.4.1. Arc C (for the TM) is interrupted and passed as input to the scheme simulator. The scheme simulator returns an output to the TM which is identical to the input unless the result of a predicate is to be returned to the TM. If a predicate is to be returned, exactly one bit is changed as described above. In this way, the scheme simulator sees every

symbol of every i.d. exactly once.

If the scheme simulator sees a reserved symbol, then the simulator knows that some action must be taken. Otherwise the simulator does nothing.

The possible actions are:

(1) Use the top value of the value list as input to a certain function or predicate.

(2) Update the top value without using it.

(3) Change the top value according to the result of a function.

(4) Add the result of a function to the value list without deleting the top value.

(5) Evaluate the result of some predicate.

(6) Return the result of the last predicate evaluated.

Once it is shown how to write the program for the scheme simulator it is easy to prove the main theorem:

Definition. Let P be an r.e. program scheme, and D be a data flow scheme with input arc G. Then P and D are said to be *equivalent* if for all interpretations of all the function and predicate symbols of P and D and all inputs to P, P halts iff D halts, and the output of P equals the output of D, where the input to D is the ordered set of inputs to P placed on arc G.

Theorem 4.2. Let P be a r.e. program scheme. Then there is a data flow scheme D that is equivalent to P.

Proof. Use the TM described in Section 4.5.1 appealing to the construction of Section 4.4. It suffices to show how to implement the above description for

the scheme simulator. The following is a detailed description of the program for the scheme simulator.

Box #1 of Figure 4.5.1:

Arc A of the flowchart of Figure 4.5.1, is the input from the data flow program which simulates the TM described in Section 4.5.1. The input is used in the following way. The scheme simulator searches through the i.d.'s for reserved message symbols. The function of box #1 is to determine which operations must be performed by the scheme simulator.

Inputs to box #1 are processed r symbols at a time where r is the number of bits needed to code each TM i.d. by f. When arc A has $f(\sigma)$ where σ is a reserved message symbol, box #1 decides which action needs to be performed for this i.d. Each possible arc in the scheme simulator (such as input arcs to function and predicate nodes) that could use a value from the value list is assigned an integer position. The possible actions are as follows. If the top value of the value list is to be used at the i^{th} "position" then arc C_i is true. Otherwise arc C_i is false. If the result of function f_i is to be added to the value list then arc D_i is true, otherwise arc D_i is false. When one of the D_i's is true, then arc E is true if the result of the function is to replace the top of the value list and is false if the result is a new value to be added to the value list. In general, arc E is true if the top of the value list is updated. In particular, if anything but a reserved message appears on A, arc E is false. Arc F_i is true if the desired action is to obtain the result of predicate i, and is false otherwise.

Arc B is a set of r symbols which specify whether a previously evaluated predicate result is to be returned. Arc B produces 1^r, unless

$f(\sigma)$ appears on arc A (where σ is the "return predicate message") in which case exactly one of the r symbols is a 0.

Careful inspection of the function of box #1 indicates that it fits the hypothesis of the finite translation lemma, and thus a program exists for it.

Box #2 of Figure 4.5.1:

The purpose of box #2 is to apply the functions and predicates. If a *HALT* is being processed, and the message says to output the top value of the value list on the i^{th} output arc, then box #2 outputs the top value of the value list on arc H_i. If the top value is to be updated, then arc G is modified according to whether a new value is added to the list, the top value is changed, or the top value is recirculated. If the top value is to be sent to a function or predicate as an input, then that is controlled here. If the result of some predicate is to be sent to the TM, it is sent on arc I.

The program for box #2 is given in Figure 4.5.2. For each possible function or predicate there is a subprogram described in Figure 4.5.2a. For the *HALT* action there is a subprogram described in Figure 4.5.2b. The subprogram for each possible function or predicate is as follows. Assume that arc G contains the value list and assume that Z is a function (or predicate). For each input of Z (if any) that needs the top value, the C values corresponding to that position will be "True", permitting the top value to be placed as an input to the function or predicate. The gating accomplished with

For each FCN or PRED Z of j VBLS, at locations $i_1,\ldots,i_j$

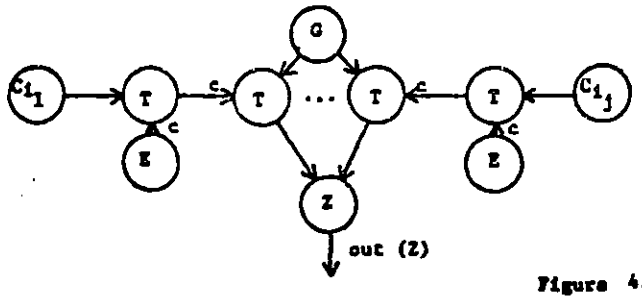


Figure 4.5.2a

For "HALT" with locations $i_1,\ldots,i_k$

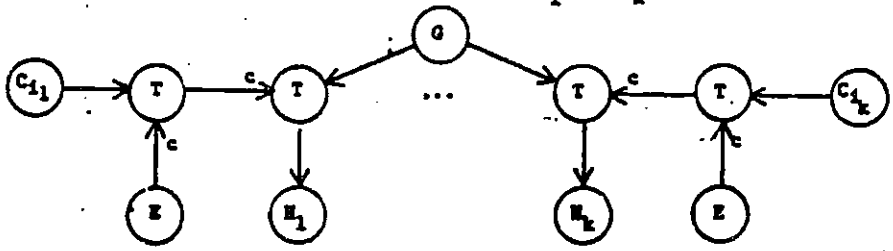


Figure 4.5.2b

arc E merely insures that the top value is unaffected if it is not supposed to be looked at during this step. The arc labelled out(Z) obtains the result of the operation after all inputs to Z have arrived.

Figure 4.5.2b is similar to 4.5.2a. If a *HALT* is the desired operation, then the meaning of "needing" the top value as the i^{th} input to *HALT*, is that arc H_i should get the top value (i.e., the top value is the i^{th} output value).

Figure 4.5.2c describes the updating of the value list. If a function has been applied, then arc #1 is the value of the function if the operation was an f_1 or f_2. Similarly, arc #2 is the result if the operation was f_1, f_2, or f_3, and arc #3 is the result irrespective of which function was applied. If the top value is to be recirculated, then it appears on arc #4. If it is to be replaced then it is absorbed by the F gate that leads into arc #4. If it is to be ignored at this step, then it remains at the input to the F gate. After an operation arc #5 contains the new value on the list if a new one was added, the new value of x_i if the old value of x_i was replaced, and the old top value, if the top value was to be recirculated. Arc #5 then becomes the end of the value list.

Figure 4.5.2d is similar to 4.5.2c in that it merges the predicate results. Arc #1 is the value of the predicate if it was the first or second predicate that applied, arc #2 is the value of the predicate if any of the

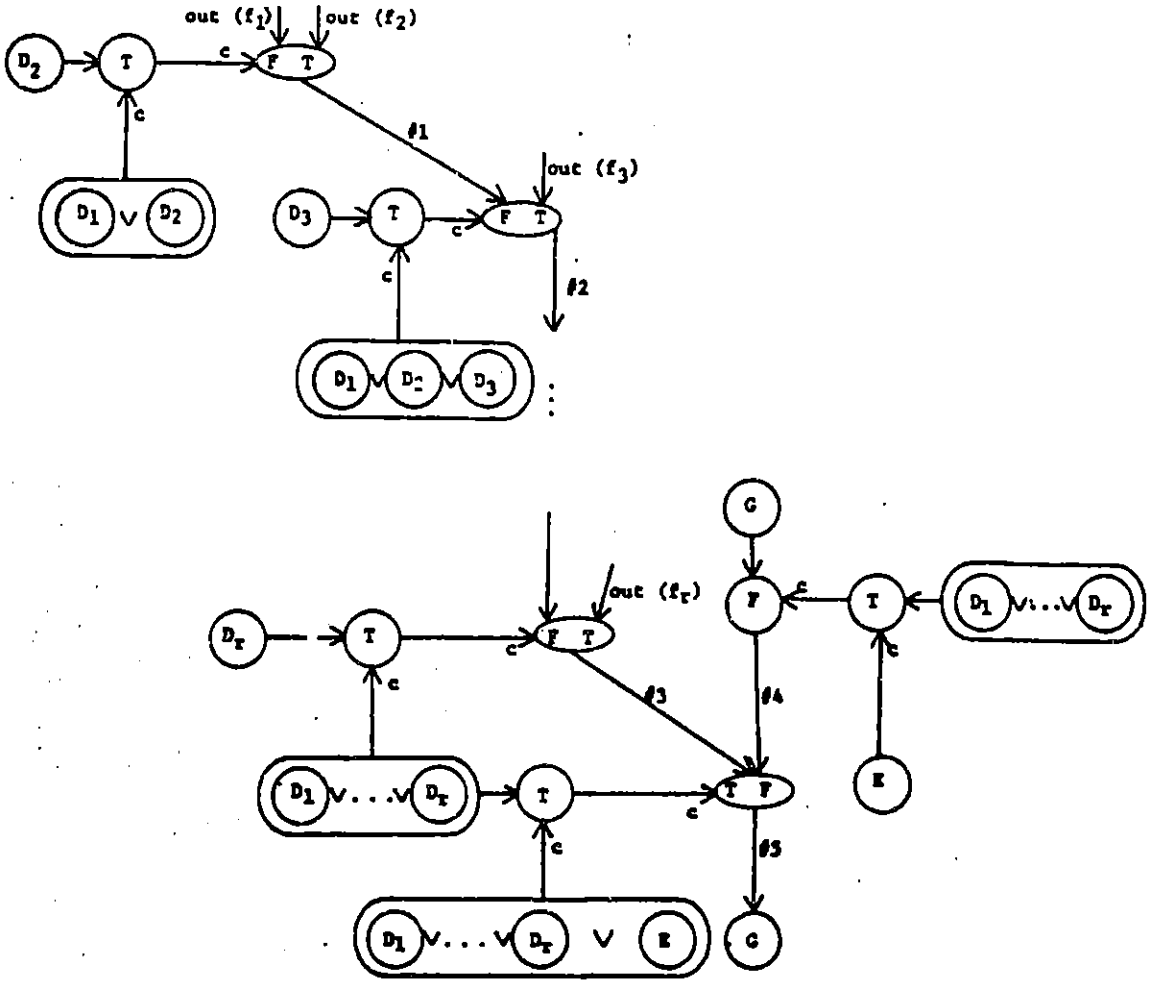

out (f_1)
out (f_2)
D_2
T
c
F T
c
#1
out (f_3)
D_1 ∨ D_2
D_3
T
c
F T
c
#2
D_1 ∨ D_2 ∨ D_3
G
out (f_r)
D_r
T
c
F T
c
#3
F
c
T
D_1 ∨ . . . ∨ D_r
c
#4
D_1 ∨ . . . ∨ D_r
T
c
T F
E
c
#5
D_1 ∨ . . . ∨ D_r ∨ E
G

Figure 4.3.2c

first three predicates applied, and arc #3 is the value of the predicate if any predicate applied. If no predicate applied, this portion of box #2 does not do anything.

Box #3 of Figure 4.5.1:

The purpose of box #3 is to merge the old TM i.d. with the result of the predicate if applicable. If arc B has 1^r then the block from arc A is output by the scheme simulator. If arc B has one 0, then arc I is substituted for one of the result bits. The program for box #3 is given in Figure 4.5.3.

If the set of arcs $\{H_i\}$ are the output arcs of this data flow program, then the above simulation successfully simulates a given r.e. program scheme P. □

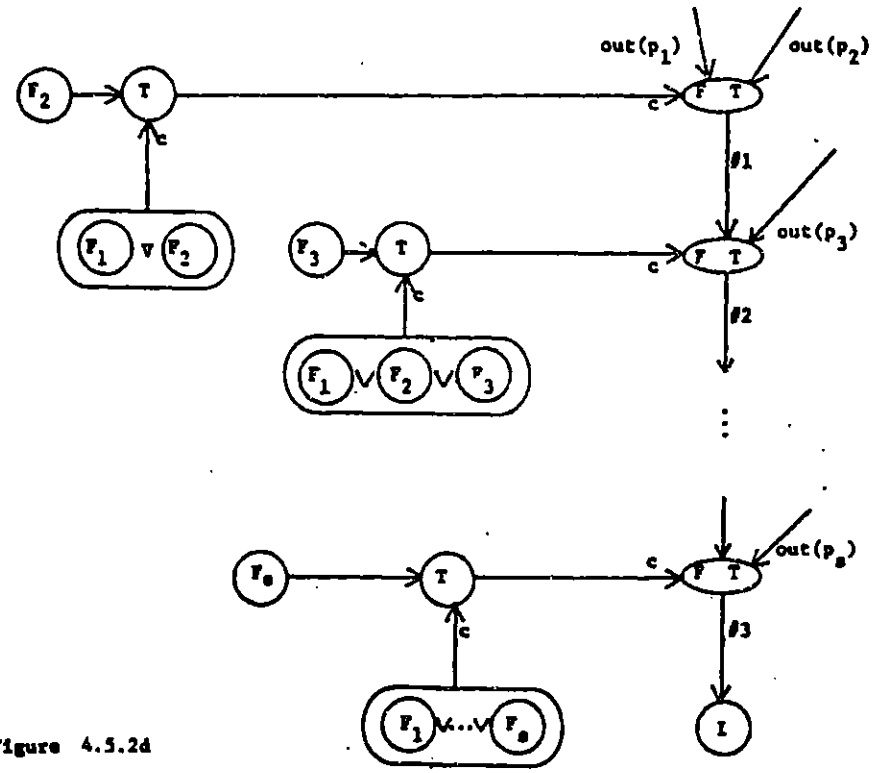


Figure 4.5.2d

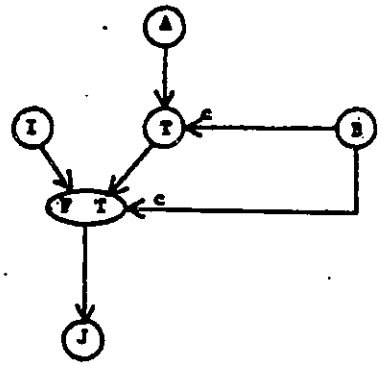


Figure 4.5.3

4.6. Further Work and Conclusion

The power of two versions of data flow schemes have now been analyzed. There is a wide gap between well formed data flow schemes which are almost a direct translation of "if-then-while" programs [41], and data flow schemes which fully express the architectural constructs of data driven architectures [14]. It would be interesting to define natural restrictions on data flow schemes which make a subclass of data flow schemes equivalent to other models in the scheme hierarchies [1,9].

There have been some attempts at defining recursive data flow schemes [12], but in order to define them a slightly different execution rule in needed. Specifically, if a node is labelled with a recursive function symbol, then it is interpreted as a "macro" denoting another copy of the recursive procedure. It follows from Theorem 4.2 that anything expressible with recursion is expressible with data flow schemes with queues. It would be interesting to find a natural translation between recursive schemes and data flow schemes, perhaps one that does not rely on the general simulation theorem.

References

1. S. Brown, D. Gries, and T. Szymanski, Program Schemes with Pushdown Stores, *SIAM Journal on Computing, 1*, 3, Sept. 1972, pp. 242-268.

2. J. Bruno, E. G. Coffman Jr., and R. Sethi, Scheduling independent tasks to reduce mean finishing time, *CACM 17*, 7 (July 1974), 382-387.

3. J. Bruno, E. G. Coffman Jr., and R. Sethi, Algorithms for minimizing mean flow time, *IFIP 74*, North-Holland, Amsterdam, pp. 504-510.

4. J. E. Burns, Mutual Exclusion with Linear Waiting using Binary Shared Variables, *Sigact News*, Whole Number 39, Vol. *10*, No. 2, Summer 1978.

5. J. E. Burns, M. J. Fischer, P. Jackson, N. A. Lynch, and G. L. Peterson, Shared Data Requirements for Implementation of Mutual Exclusion Using a Test-and-Set Primitive, University of Washington TR No. 78-08-03.

6. Y. Cho and S. Sahni, Bounds for list schedules on uniform processors, University of Minnesota TR78-13, June 1978.

7. E. G. Coffman, *Computer and Job Shop Scheduling Theory*, J. Wiley and Sons, NY 1976.

8. E. G. Coffman and R. L. Graham, Optimal scheduling for two-processor systems. *Acta Informatica, 1*, 200-213.

9. R. L. Constable and D. Gries, On classes of Program Schemata, *SIAM Journal on Computing*, *1*, 1, March 1972, pp 66-118.

10. E. Davis and J. M. Jaffe, Algorithms for scheduling tasks on unrelated processors, MIT, Laboratory for Computer Science Technical Memo No. ???, June 1979. Also, submitted to *JACM*.

11. N. G. de Bruijn, Additional Comments on a Problem in Concurrent Control, *CACM*, 10, 1967, pp 137-138.

12. J. B. Dennis, First Version of a Data Flow Procedure Language, *Lecture Notes in Computer Science 19* (G. Goos and J. Hartmanis eds.) pp 362-376, Also *Symposium on Programming*, Institut de Programmation, Univ. of Paris, Paris, France, April 1974, pp 241-271.

13. J. B. Dennis, J. B. Fosseen, and J. E. Linderman, Data Flow Schemas, *International Symposium on Theoretical Programming, Lecture Notes in Computer Science 5*, Springer Verlag, Berlin 1974, pp 187-216.

14. J. B. Dennis and D. P. Misunas, A Preliminary Architecture for a Basic Data-Flow Processor, *Proceedings of the Second Annual Symposium on Computer Architecture*, Jan. 1975, pp 126-132.

15. E. Dijkstra, Solution of a Problem in Concurrent Programming Control, *CACM*, 9, 9, 1965, page 569.

16. M. A. Eisenberg and M. R. McGuire, Further Comments on Dijkstra's Concurrent Programming Control Problem, *CACM*, *15*, 11, 1972, page 999.

17. M. J. Fischer, Private Communication.

18. M. R. Garey and R. L. Graham, Bounds for Multiprocessing Scheduling with Resource Constraints, *SIAM J. Comput. 4*, 2, June 1975, pp 187-200.

19. M. R. Garey and D. S. Johnson, Complexity Results for Multiprocessor Scheduling under Resource Constraints, *Proceedings of the Eighth Annual Princeton Conference on Information Sciences and Systems*, 1974.

20. M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco (1979).

21. T. Gonzalez, O. H. Ibarra, and S. Sahni, Bounds for LPT schedules on uniform processors, *SIAM J. Comput.*, *6*, 1, (1977) pp 155-166.

22. D. K. Goyal, Scheduling Processor Bound Systems, *Proceedings of the Sixth Texas Conference on Computing Systems* 1977.

23. R. L. Graham, Bounds on multiprocessing timing anomalies, *SIAM J. of Appl. Math.*, *17*, (1969) 263-269.

24. R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan, Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey, *Discrete Optimization, 1977.*

25. S. A. Greibach, *Theory of Program Structures: Schemes, Semantics, Verification. Lecture Notes in Computer Science 36*, (G. Goos and J. Hartmanis Eds.) Springer Verlag, Berlin 1975.

26. J. E. Hopcroft and J. D. Ullman, *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.

27. E. Horowitz and S. Sahni, Exact and approximate algorithms for scheduling nonidentical processors, *JACM, 23*, 2 (April 1976), 317-327.

28. E. C. Horvath, S. Lam, and R. Sethi, A Level Algorithm for Preemptive Scheduling, *JACM 24*, 1, (1977) 32-43.

29. I. Ianov, The Logical Schemes of Algorithms in *Problems of Cybernetics I*, pp. 82-140, Perganon, NY 1960.

30. O. H. Ibarra and C. E. Kim, Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors, *JACM 24, 2*, (April 1977), 280-289.

31. J. M. Jaffe, The Use of Queues in the Parallel Data Flow Evaluation of 'If-Then-While' Programs, proceedings of the *12th Annual Johns Hopkins*

Conference on Information Systems and Sciences, March, 1978, Baltimore, MD, pp 451-456. MIT Laboratory for Computer Science Technical Memo 104, May, 1978.

32. D. G. Kafura and V. Y. Shen, Task scheduling on a multiprocessor system with independent memories, *SIAM J. Comput.* 6, (March 1977), 167-187.

33. R. M. Karp and R. E. Miller, Properties of a model for parallel computations: determinacy, termination, queueing, *SIAM J. of Applied Math.*, *14*, 6, Nov. 1966, pp 1390-1411.

34. H. P. Katseff, A new solution to the Critical Section Problem, *Proc 10th Annual Symposium on Theory of Computing*, 1978, pp. 86-88.

35. D. E. Knuth, Additional Comments on a Problem in Concurrent Control, *CACM*, 9, 1966, pp 321-322.

36. S. Lam and R. Sethi, Worst case analysis of two scheduling algorithms, *SIAM Journal on Computing, 6*, pp 518-536, 1977.

37. L. Lamport, A new solution of Dijkstra's Concurrent Programming Problem, *CACM*, *17*, 8, 1974, page 453.

38. L. Lamport, *Time, Clocks and the Ordering of Events in a Distributed System*, *CACM*, *21*, 7, 1978, pp 558-565.

39. E. L. Lawler and J. Labetoulie, On preemptive scheduling of unrelated parallel processors by linear programming, *JACM*, 25, 4, 1978 612-619.

40. J. K. Lenstra and A. H. G. Rinnooy Kan, Complexity of scheduling under precedence constraints, *Operations Research*, 25, to appear.

41. C. K. Leung, Formal Properties of Well-Formed Data Flow Schemas, MIT, LCS, TM 66, Cambridge, MA., June 1972.

42. J. W. S. Liu and C. L. Liu, Bounds on Scheduling Algorithms for Heterogeneous Computing Systems, TR No. UIUCDCS-R-74-632 Dept. of Comp. Sci., Univ. of Illinois, June 1974.

43. J. W. S. Liu and C. L. Liu, Bounds on Scheduling Algorithms for Heterogeneous Computing Systems, *IFIP74*, (North Holland Pub. Co.), 349-353.

44. J. W. S. Liu and C. L. Liu, Performance Analysis of Multiprocessor Systems Containing Functionally Dedicated Processors, *Acta Informatica*, *10*, 1, (1978) 95-104.

45. E. L. Lloyd, Private Communication

46. J. McCarthy, Towards a Mathematical Science of Computation pp 21-28, *Proceedings of IFIP Congress*, Munich 1962.

47. R. R. Muntz and E. G. Coffman Jr., Optimal preemptive scheduling on two-processor systems, *IEEE Trans. Comptrs., C-18*, 11 (1969) 1014-1020.

48. R. R. Muntz and E. G. Coffman Jr., Preemptive scheduling of real time tasks on multiprocessor systems, *JACM 17*, 2 (1970) 324-338.

49. M. Paterson, Equivalence Problems in a Model of Computation, Ph.D. Thesis, Univ. of Cambridge.

50. M. Paterson and C. Hewitt, Comparative Schematology, *Record of the Project MAC Conference on Systems and Parallel Computations*, ACM, New York, 1970, pp 119-128.

51. S. S. Patil, Closure Properties of interconnections of determinate systems, *Record of the Project MAC Conference on Systems and Parallel Computations*, ACM, New York, 1970, pp 107-116.

52. G. Peterson and M. Fischer, Economical Solutions for the Critical Section Problem in a Distributed System, *Proc 9th Annual Symposium on Theory of Computing*, 1977, pp. 91-97.

53. R. Rivest and V. Pratt, The Mutual Exclusion Problem for Unreliable Processes: Preliminary Report, *Proc 17th Annual Symposium on Foundations of Computer Science*, 1976, pp 1-8.

54. S. Sahni, Algorithms for scheduling independent tasks, *JACM, 23*, 1, (Nov. 1976), pp 116-127.

55. S. Sahni and T. Gonzalez, Preemptive scheduling of two unrelated machines, Tech. Rep. 76-16 Comptr. Sci. Dept., U. of Minnesota, Minneapolis, MN (November 1976).

56. Sethi, R. "Algorithms for Minimal-Length Schedules", in *Computer and Job Shop Scheduling Theory*, (E. G. Coffman, ed.) J. Wiley and Sons, NY 1976.

57. H. R. Strong, High level languages of Maximum Power, *Proceedings of Twelfth IEEE Conference on Switching and Automata Theory*, 1971, pp 1-4.

58. H. R. Strong, Translating Recursion Equations into Flowcharts, *JCSS*, *5*, (1971), pp 254-285.

59. J. E. Thornton, *Design of a Computer - The Control Data 6600*, Scott, Foresman College Division, (1971).

60. J. D. Ullman, *NP*-complete scheduling problems, *JCSS 3*, June 1975, pp 384-393.

Biographical Note

Jeffrey Jaffe was born in New York, NY on July 21, 1954. He soon moved to Brooklyn, NY where he lived until graduating the Yeshiva University High School of Brooklyn in June 1972.

Mr. Jaffe attended the Massachusetts Institute of Technology as an undergraduate where he received a B.S. in Mathematics in June 1976. While an undergraduate, he was elected to the Xi chapter of Phi Beta Kappa in fall 1975.

Mr. Jaffe continued his MIT education with the Department of Electrical Engineering and Computer Science in September 1976. He received his Master's degree from EECS in June 1977, and his Ph.D. in August 1979. During this time he was a National Science Foundation fellow and was associated with the Theory of Computation group. Also during this time (6/19/1977) he married the former Esther Klipper '79.

Mr. Jaffe will be joining the Decentralized Network Control Group at the IBM - TJ Watson Research Center in September 1979. He will also resume his tour of the five boroughs by moving to the Riverdale section of The Bronx.