1

**PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT**

PRECISIONSM RESOLUTION TARGETS

PIONEERS IN METHYLENE BLUE TESTING SINCE 1974

1600 COUNTY ROAD 6, BATHONVILLE, MISSISSIPPI 39027 USA
TEL: 601-426-7887 FAX: 601-426-7887 TLR: 6100000400

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

**PERFORMANCE EVALUATION OF A HIERARCHICAL TASK QUEUE
ORGANIZATION FOR MULTIPROCESSOR SYSTEMS**

by

Siu Pang Philip Cheng, B.C.S.

**A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of**

Master of Computer Science

**School of Computer Science
Carleton University
Ottawa, Ontario
April 30, 1992
© Copyright
1992, Philip Cheng**

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-75983-6

Canada

The undersigned recommend to the Faculty of Graduate Studies and Research
acceptance of the thesis

"Performance Evaluation of a Hierarchical Task Queue Organization for
Multiprocessor Systems"

submitted by

Siu Pang Philip Cheng, B.C.S.

in partial fulfillment of the requirements for
the degree of Master of Computer Science

Thesis Supervisor

Director, School of Computer Science

Carleton University

April 30, 1992

ABSTRACT

There are two basic ways in which waiting ready tasks in a shared-memory multiprocessor system can be organized: *centralized* organization or *distributed* organization. In a centralized organization, a single central task queue is maintained. In the latter case, each processor has its own private ready queue of tasks. Ideally, a central ready queue global to all processors is desired over the distributed organization because it allows perfect load sharing. However, the centralized organization is not suitable for large systems because the global task queue could become a bottleneck. The distributed organization, on the other hand, eliminates this task queue bottleneck disadvantage but there may be load imbalances in the sense that some local task queues may be empty while others have tasks to be scheduled.

Ideally, therefore, an organization that combines the desirable characteristics of both these organizations while avoiding the disadvantages associated with them is desired. That is, the goal is to have an organization that allows load sharing (as in the centralized organization) and that distributes task queues to eliminate the bottleneck problem (as in the distributed organization). A hierarchical organization has been proposed to incorporate the good features of these two organizations. This thesis provides a detailed performance evaluation of this hierarchical task queue organization.

ACKNOWLEDGEMENTS

I would take this opportunity to express my profound gratitude to Dr. S. Dandamudi, my thesis advisor, for introducing me to such a fascinating subject, and for guiding me through my research with such enthusiasm and expertise. In spite of his various preoccupations, he devotes time for stimulating discussions, whenever the need arose.

I would also like to thank my parents for the financial support of my studies, both undergraduate and graduate, in Canada. Without their assistance, I would not have been able to pursue the work reported in this thesis.

I am also grateful for Dr. G. White, University of Ottawa, and Dr. S. Majumdar, Department of Systems and Computer Engineering, Carleton University, in providing the comments on the final draft of this thesis.

Finally, I like to thank the school of computer science for financial support.

TABLE OF CONTENTS

ACCEPTANCE SHEET	ii
ABSTRACT	iii
ACKNOWLEDGEMENTS	iv
TABLE OF CONTENTS	v
LIST OF FIGURES	viii
LIST OF TABLES	xii
LIST OF SYMBOLS	xiii
 CHAPTER I:	
INTRODUCTION	1
1.1 Overview of the Thesis	7
 CHAPTER II:	
HIERARCHICAL ORGANIZATION	9
2.1 Previous Work	9
2.2 Motivation of Hierarchical Organization	14
2.3 Description of the Hierarchical Organization	15
 CHAPTER III:	
SYSTEM AND JOB MODELS	19
3.1 Multiprocessor System Model	19
3.2 Job Models	22
3.3 Queue Locking Model	26
 CHAPTER IV:	
PERFORMANCE WITH INDEPENDENT WORKLOAD	30
4.1 Workload Model	30
4.2 Performance Comparisons among the Three Organizations	31
4.2.1 Impact of Access Contention	32
4.2.2 Effect of Number of Tasks	36
4.2.3 Sensitivity to Service Time Variance	40

4.2.4	Performance in Large System	43
4.3	Sensitivity to Structural Parameters	45
4.3.1	Impact of Branching Factor	46
4.3.2	Impact of Task Transfer Factor	47
4.4	Discussion	48
4.5	Summary	50

CHAPTER V:

	IMPACT OF SCHEDULING POLICIES	51
5.1	Introduction	51
5.2	Performance of job scheduling policies	52
5.2.1	Job Scheduling Policies	53
5.2.2	Simulation Results	54
5.2.2.1	Performance Comparison	54
5.2.2.2	Sensitivity to Task Service Time Variance	55
5.2.2.3	Impact of Task Distribution Variance	56
5.2.3	Discussion	57
5.3	Performance of Task Scheduling Policies	58
5.3.1	Task Scheduling policies	58
5.3.2	Simulation Results	64
5.3.2.1	Performance Comparison	64
5.3.2.2	Performance Sensitivity to Task Service Time Variance	66
5.3.2.3	Impact of Variance in Task Distribution	67
5.3.2.4	Effect of Window Size	69
5.3.2.5	Sensitivity to Quantum Size and Context Switch Overhead	70
5.3.3	Discussion	71
5.4	Summary	72

CHAPTER VI:

	PERFORMANCE WITH SYNCHRONIZATION WORKLOADS	73
6.1	Introduction	73
6.2	Scheduling Policies	74
6.3	Lock Accessing Model	76
6.3.1	Workload Model	76

6.3.2	Simulation Results	77
6.3.2.1	Principle Comparison	78
6.3.2.2	Performance Sensitivity to Service Time Variance	79
6.3.2.3	Impact of Lock Holding Ratio	80
6.3.2.4	Impact of the Number of Iterations	81
6.3.2.5	Impact of the Queue Access Time	82
6.3.2.6	Sensitivity to the Spinning Time in the Hybrid Policy	83
6.3.3	Summary	84
6.4	Barrier Synchronization Model	85
6.4.1	Workload Model	85
6.4.2	Simulation Results	86
6.4.2.1	Impact of System Load	86
6.4.2.2	Performance Sensitivity to Service Time Variance	87
6.4.2.3	Impact of Task Granularity	88
6.4.2.4	Impact of the Queue Access Time	89
6.4.2.5	Performance Sensitivity of Hybrid Policy to the Spinning Time	90
6.4.3	Summary	91
6.5	Conclusion	91
CHAPTER VII:		
	CONCLUSIONS	93
7.1	Summary	93
7.2	Contributions of the Thesis	96
7.3	Directions for Further Research	96
REFERENCES		98

LIST OF FIGURES

<u>Figure_</u>	<u>Descriptions</u>	<u>Page</u>
1.1	Shared-memory system	2
1.2	Task queue organizations	5
2.1	Structure of the binary X-tree used in distributed hierarchical control system	10
2.2	Basic structure of server-based system	13
2.3	Hierarchical organization for $N = 8$ processors with a branching factor $B = 2$	16
3.1	System architecture	20
3.2	Fork-and-join job structure	23
3.3	Generic program structure for each task T_i in lock accessing job model	24
3.4	Generic program structure of each task T_i in barrier synchronization job model	25
3.5	Algorithm of the MCS list-based queuing lock	27
3.6	Pictorial version of MCS locking protocol	28
3.7	Extended version of queue lock applied to the root node	28
4.1	Performance of the three organizations as a function of the system utilization	33
4.2	Synchronization time in the distributed and hierarchical organizations as a function of the system utilization ($f = 4\%$)	35

4.3	Performance of the three organizations as a function of the average number of tasks per job with the fixed task size workload	38
4.4	Performance of the distributed and hierarchical organizations as a function of the average number of tasks per job with the fixed job size workload	40
4.5	Performance sensitivity of the three organizations to the task service time variance	41
4.6	Synchronization time in the distributed and hierarchical organizations as a function of service time variance	43
4.7	Performance of the distributed and hierarchical organizations as a function of system load when the task service demand is doubled	44
4.8	Performance of the distributed and hierarchical organizations as a function of system load when the average number of tasks per job is doubled	45
4.9	Impact of the branching factor on the performance of the hierarchical organization	46
4.10	Impact of the transfer factor on the performance of the hierarchical organization	47
4.11	Average number of queue access as a function of the system utilization	48
5.1	Performance of job scheduling policies as a function of system load	55
5.2	Performance sensitivity of scheduling policies to service time variance at the system load of 50% and 85%	56

5.3	Performance sensitivity of scheduling policies to task distribution variance at the system load of 50% and 85%	57
5.4	Behaviour of RR1 policy	61
5.5	Behaviour of RR2 policy	62
5.6	Behaviour of RR3 policy	63
5.7	Performance of task scheduling policies as a function of the system load ($C_s = 1$)	64
5.8	Performance of task scheduling policies as a function of the system load ($C_s = 7$)	66
5.9	Performance sensitivity to task service time variance ($\rho = 85\%$)	67
5.10	Performance sensitivity to task distribution variance ($\rho = 85\%$)	68
5.11	Performance sensitivity of RR policies to window size ($\rho = 85\%$)	69
5.12	Performance sensitivity of RR policies to quantum size (Q)	70
5.13	Performance sensitivity of RR policies to context switch overhead (O_c)	71
6.1	Performance of scheduling policies as a function of the system load (ρ) on the lock accessing workload	79
6.2	Performance sensitivity of scheduling policies to the service time variance	80
6.3	Impact of lock holding ratio	81

6.4	Performance of scheduling policies as a function of maximum number of iterations (Max_i)	82
6.5	Performance sensitivity of the scheduling policies on queue access time (γ)	83
6.6	Impact of spinning time (T_S) on the performance of the hybrid policy	84
6.7	Performance of scheduling policies as a function of system utilization	87
6.8	Impact of the service time variance (V_{Sc})	88
6.9	Performance sensitivity to the maximum number of iterations	89
6.10	Impact of the queue access time (γ)	90
6.11	Performance of the hybrid policy as a function of the spinning time (T_S)	91

LIST OF TABLES

<u>Figure</u>	<u>Descriptions</u>	<u>Page</u>
3.1	System parameters	22
4.1	Default parameter values used in the performance comparison	31
5.1	Default parameter values used in the task and job scheduling experiments	52
6.1	Parameter values used in the lock accessing workload experiments	77
6.2	The spinning time of suspended tasks vs the task service time variance in the spinning policy	80
6.3	Parameter values used in the barrier synchronization workload experiments	86

LIST OF SYMBOLS

<u>Symbol</u>	<u>Descriptions</u>
λ	Job arrival rate
μ	Task service rate
ρ	System utilization
B	Branching factor
C_T	Task distribution coefficient of variation
C_S	Service time coefficient of variation
Cd_{in}	Communication delay induced by the interconnection network in the system
Cd_{fe}	Communication delay induced by the interconnection network in front-end system
CV	Coefficient of variation
f	Queue access time
L	Number of levels in the hierarchical organization
O_C	Context switch overhead
Max_i	Maximum number of iterations in a task
N	System size
Q	Quantum size
S_C	Task service time in region 1 or local computation phase
S_l	Task lock holding time
T	Average number of tasks per job
T_S	Spinning time in the hybrid policy
Tr	Transfer factor
W_S	Window size
V_{Sc}	Variability of the local computation service time

CHAPTER I

INTRODUCTION

A parallel processing system consists of three main components: processors, memory modules and interconnection network. Processors executing a parallel solution to a problem usually communicate with the other processors to exchange intermediate results. The interconnection network is the component that facilitates such communication. Depending on the number of instruction streams and data streams, parallel systems can be divided into two classes [Flynn 1972]: SIMD (Single Instruction Multiple Data) system or MIMD (Multiple Instruction Multiple Data) system. In SIMD systems, processors execute the same set of instructions in parallel on different data. Connection Machine [Tucker & Robertson 1988] is an example of this class of parallel systems. In MIMD systems, processors execute instructions independently. Many parallel systems such as the Ceder system [Gajski et al. 1983], the Sequent Symmetry system [Lovett & Thakkar 1988] and the BBN Butterfly system belong to the MIMD group.

Depending on how the processors and memory modules are interconnected, MIMD systems are classified into two main categories: distributed-memory systems (also called multicomputer systems or loosely-coupled systems) and shared-memory systems (also called multiprocessor systems or tightly-coupled systems). This thesis is concerned with the shared-memory MIMD systems.

In a shared-memory system (Figure 1.1), all memory modules are shared by processors in the system and are used as a communication medium. This type of systems allows code and data sharing with a lower cost, and can support finer grain parallelism (compared to the distributed-memory systems). Although the shared-memory systems are architecturally complex, systems with hundreds of processors can be implemented with the existing technology (e.g. the BBN Butterfly system). Ideally, every processor in a shared-

memory system can access any memory module with an identical cost. Many systems such as the Ceder [Gajski et al. 1983] and the Sequent Symmetry system [Levett & Thakkar 1988] provide some additional local memory for each processor in order to reduce the memory access time. This local memory architecture provides better performance [Stenstorm 1988].

Figure 1.1 Shared-memory system

There are three main types of interconnection networks that are generally suitable for the shared-memory systems [Marson et al. 1986]: *bus network*, *crossbar network* and *multistage interconnection network*. Bus-based systems are limited by the available bandwidth (and hence these systems cannot be expanded to large system sizes). The cost of the network, however, is low. On the other hand, crossbar network provides an excellent performance by having distinct connection between any pair of components. However, its prohibitive cost is the major drawback of this network.

A multistage interconnection network (MIN) is constructed by a set of switches. The network is divided into a few stages and each stage has a set of switches. Each switch is capable of connecting an arbitrary input port to an arbitrary output port. The bandwidth of the network increases with system size N . In terms of cost and performance, MINs fall in between the bus and crossbar networks. Thus, MINs are considered to be suitable for the large-scale systems.

In order to exploit the organization of parallel systems, a job (program or application) is partitioned into several tasks, also called grains, that can be run in parallel. *Grain size* or *task size* is an important issue in considering the performance of parallel systems. Informally, grain size refers to the average task execution time. At one extreme, the entire job can be considered as a single, large task. At the other extreme, we can consider each executable instruction as a task. Ideally, fine grain size is preferred because it allows maximum exploitation of parallelism present in the job. For example, dataflow architectures (at least conceptually) support fine grain parallelism. In this architecture, each task can just be a single instruction. It should, however, be noted that associated overheads increase as the grain size decreases. Unless these overheads are well-controlled, exploiting such fine grain parallelism may cause implementation and performance problems. Thus, this thesis considers systems that can support medium grain parallelism efficiently. These systems include the shared-memory multiprocessor systems such as the Sequent Symmetry system [Lovett & Thakkar 1988] and the DEC Firefly [Thacker et al. 1988]. The average task size is large when compared to that in the dataflow systems. Also note that these systems can support coarse grain parallelism in which, for instance, a task could simply consist of an entire job (i.e., a job is not split into several tasks).

Typically, all executable tasks (i.e. executable tasks) wait in a ready queue to be scheduled. Conceptually, there are two operations associated with a ready queue: an *insert* operation to insert a task into the ready queue and a *remove* operation to delete a task from the queue. Obviously, mutually exclusive access should be provided to avoid conflicts and to insure proper operation of the queue. Mutually exclusive access to the ready queue can be implemented in several ways. Each ready queue could maintain a mutual exclusion lock as in Mach operating system [Black 1990]. Alternatively, a *monitor* [Hoare 1974] can be used to control exclusive access to the ready queue [Ni & Wu 1989]. In a lock-based implementation, for instance, an idle processor can repeatedly try to fetch a task from the

queue until it succeeds if the ready queue is empty or if the lock is not free. On the other hand, in a monitor-based implementation, idle processors will be placed in a condition queue. When a task is available in the ready queue, a suspended processor in the condition queue is assigned in a FCFS fashion by a signal operation. The underlying behaviour, however, remains the same (for the most part) in the sense that each task should get inserted into the ready task queue and then is removed from the queue.

There are two basic ways in which the set of ready tasks can be organized: *centralized* organization or *distributed* organization. In the centralized organization, there is a single global queue of ready tasks that is accessible to all processors in the system. On the other hand, in the distributed organization, local ready queues are associated with the processors. Figure 1.2 shows these two organizations. In the absence of task queue access contention, the centralized organization provides better performance mainly due to its load sharing property. However, due to access contention, the centralized organization is not suitable for large parallel systems (i.e., with large number of processors). The distributed organization, on the other hand, avoids this problem but it may cause load imbalance resulting in poorer performance. Thus, this organization may not be suitable for large systems.

It is desirable to have an organization that combines the advantages of these two task queue organizations while avoiding the associated disadvantages. The goal is to have an organization that allows load sharing and that distributes the task queues to eliminate the access contention problem. A hierarchical organization has been proposed in [Dandamudi 1990] to achieve this goal. In the proposed hierarchical organization, a set of queues are organized as a tree. Processors in the system are linked to a particular queue at leaf level of the tree. When a processor requests a new task, it starts at the leaf level. If the queue is empty, the processor then moves up the tree for executable tasks. Chapter 2 provides a detailed description of this organization. The goal of this thesis is to evaluate the performance of the new task queue organization.

Figure 1.2: Task queue organizations
(SA = Synchronization Area)

Normally, tasks that run in parallel often interact with each other. These interactions arise in at least two situations: accessing a shared data structure and waiting for other processors to finish their tasks before proceeding onto the next phase of the computation. As Zahorjan et al. [1988] point out, the first situation involves waiting because of computation among tasks and the second situation involves waiting because of cooperation among tasks.

Locks are generally used to maintain consistency of shared resources by providing mutually exclusive access. Generally, a task requiring a lock does some local computation, and then requests the lock. After completing the operations inside the critical section, it releases the lock and initiates the next iteration. When a task requires a lock that is not free, the task cannot continue its execution. As discussed previously, ready queue, which stores the executable tasks, is an example of a shared resource.

The second situation is generally referred to as *barrier synchronization*. Barrier synchronization is adopted to coordinate the execution of tasks within a job. Tasks in jobs with the barrier synchronization alternate between a computation phase and a communication phase. The tasks do their computations independently, and then wait for the other tasks to complete their computation phase. That is, when a task reaches the barrier, it

suspends execution unless all the tasks had already reached the barrier. The Fast Fourier Transform (FFT) is an application that contains this type of synchronization. The computation of the next step may not commence unless the results from the present phase are obtained. Both synchronization behaviours cause performance degradation [Zahorjan et al. 1988].

There are two mechanisms that are commonly used to suspend processes: *spinning* and *blocking*. In the spinning mechanism, the tasks requiring locks sit in a tight loop continuously checking for the lock to become free. The time spent spinning is the overhead of this mechanism. If the blocking mechanism is employed, the processor suspends the current task and selects a new executable task. The context switch is the overhead involved in the blocking mechanism. Both mechanisms affect (i.e. degrade) the system performance. The choice between spinning and blocking depends on the relationship of the spinning time to the context switch time. It cannot be determined *a priori* due to uncertainties in multiprocessing environment. Zahorjan et al. [1989] discuss the trade-offs involved between these two mechanisms in a small multiprocessor system.

There are several alternatives to implement the spin-waiting mechanism [Mellor-Crummey & Scott 1990]. Anderson [1990] examined different spinning alternatives in the Sequent Symmetry system. This study demonstrates that the implementation of spin locks may cause serious memory contention. Both Anderson [1990] and Mellor-Crummey and Scott [1990] use distributed schemes to implement the spin-waiting mechanism in order to prevent memory contention. Mellor-Crummey and Scott [1990] present a performance comparison of these two distributed schemes. The results show that both schemes are feasible and avoid the contention problem in multiprocessor systems.

1.1 Overview of the Thesis

The remainder of the thesis is structured as follows.

Chapter 2 presents a detailed description of the hierarchical organization. The chapter starts with a survey of processor scheduling in shared-memory systems. This chapter also provides the motivation for the new organization along with a detailed description of the new organization.

Chapter 3 gives details on the system and job models used in conducting the performance evaluation of the new hierarchical task queue organization. The job models used in the study include an independent model (called fork-and-join model), a lock accessing model and a barrier synchronization model.

Chapter 4 presents the performance of the new organization using the independent job model. The performance advantages of the hierarchical task queue organization over the centralized and distributed organizations are demonstrated in this chapter. Also, the performance sensitivity of the new organization to the structural parameters is provided.

Chapter 5 presents the impact of various scheduling policies on the performance of the hierarchical organization under the independent job model workload. At job level scheduling (i.e., the order of jobs to be executed), the performance of two job-dependent policies is compared with the simple FCFS policy which is used in Chapter 4. At task level scheduling (i.e., the order of tasks to be executed regardless of which job the tasks are coming from), four round robin based preemptive policies are considered.

Chapter 6 analyzes performance of the hierarchical organization with the two job dependent models - lock accessing and barrier synchronization. The performance trade-off between spinning and blocking is studied. Since the suspended tasks can be added to any queue in the organization, two variations on the basic blocking policy are considered. In addition, a hybrid policy in which tasks spin for a while and then block is included in the performance comparison.

Finally, Chapter 7 provides a summary of the thesis. The contributions of the thesis and directions for further research are also given.

CHAPTER II

HIERARCHICAL ORGANIZATION

Processor scheduling is an important performance issue in the design of multiprocessor systems. This chapter starts with a brief summary of the previous work in the area of processor scheduling in multiprocessor systems. Then, the motivation for the new hierarchical organization is presented. Finally, a detailed description of the new organization is given.

2.1 Previous Work

Depending on how the resources are allocated, processor scheduling strategies can be classified into three main categories: *global scheduling*, *two-level scheduling* and *self-scheduling*. In *global scheduling approach* (or *dispatcher approach*), a centralized control unit (or subsystem) makes the decision for the allocation of processors. When a processor is idle or completes the quantum of service for a task (in time-sharing scheduling policies), the centralized dispatcher assigns a new task to that processor.

A strategy using *global scheduling approach* has been used in [Majumdar et al. 1988] to evaluate the scheduling policies in multiprocessor systems. One of the advantages of this strategy is to avoid the consistency problem of the ready queue(s) because the queues are accessed only by the centralized control unit. Besides, global coordination can be achieved by the centralized control unit. Gang scheduling policy and the round robin policies, which will be discussed in Chapter 5, can also be implemented using this strategy.

Unfortunately, this strategy has some disadvantages. Firstly, additional hardware is required to implement the control unit. Secondly, the design of such a system is very complicated because the control system requires direct connection to each processor. Also, the central control unit becomes the system bottleneck as the system size increases.

Furthermore, the failure of the control unit crashes the operation of the whole system. Thus, this scheduling approach is not suitable for large-scale systems.

To solve this problem, Feitelson and Rudolph [1990a] use the idea of distributed hierarchical control to implement the control system. A set of controllers are implemented in the X-tree structure as shown in Figure 2.1 where each node represents a controller. The controllers at each level are responsible for a different range of degrees of parallelism. The bottom ones are concerned only with a pair of processors and the top one is concerned with those activities that involve more than half the processors. Within a level, controllers are responsible for spatially distinct groups of processors. Thus, the controller system has multiple levels of control which allow concurrent scheduling of multiple group of tasks.

Figure 2.1: Structure of the binary X-tree used in distributed hierarchical control system [Feitelson & Rudolph 1990a]

The failure of a single controller in the system using the distributed hierarchical control alternative only affects part of the processing power rather than degrading the operation of the whole system. However, the distributed hierarchical control scheme requires a special architecture in which the control system is orthogonal to the existing system. Feitelson and Rudolph [1990a] avoid this type of architecture by considering a software implementation of the control system. They suggest that the processing units be

partitioned into two modes: *user* and *supervisor*. In the *supervisor* mode, the processors act as the controllers. They map the newly created jobs, determine the time slice for each application and schedule the tasks for the next quantum.

In order to determine the time slices of each application, each controller should maintain system information (i.e. number of threads and the number of applications mapped to the subtree of controller). This kind of distributed information gathering requires a large amount of communication overhead. These additional overheads are an important performance issue in the implementation of this strategy. Also, gang scheduling policy may increase the spin-time in lock access workload [Zahorjan et al. 1991]. Furthermore, the failure of a controller still affects the system performance seriously. The applications which are controlled by the faulty controller cannot be re-scheduled.

In the strategy using the *two-level* approach, the global control unit, which is in a higher level of the organization (operating system level), partitions the processors among the applications. Each application then locally schedules tasks to those processors. Depending on the processor allocation policy, this approach is further divided into two groups: *static* and *dynamic*. In the static alternative, the number of processors assigned to the applications is fixed during the life time of the applications. When the dynamic alternative is applied, the number of processors available for each application is changed according to the system workload and the parallelism of applications in the system.

Zahorjan and MaCann [1990] evaluated the scheduling policies using the two-level approach. In their study, applications may change their parallelism during their executions. The authors found that the dynamic policy has better performance than the static policies and the simple 'run-to-completion' policy is preferable to the time-sharing policy.

Leutenegger and Vernon [1990] study scheduling policies in multiprocessor systems using the dynamic two-level approach. The results indicate that the policy using the two-level approach has comparable performance with the round robin and coscheduling policies

under various workloads. In the lock access workload (considered in Chapter 6), the policy is the most efficient one because of the full knowledge about the applications. In this type of workload, the dynamic policy does not preempt the tasks which are holding locks and does not schedule tasks that are waiting for a lock.

Gupta and Tucker [1989] discuss the implementation of a dynamic two-level strategy. They find that the distributed scheme is difficult to be implemented because of the large amount of communication overheads. They selected a centralized scheme and constructed a centralized server on top of the operating system kernel (Figure 2.2). The server frequently changes the system status by acquiring information about the number of processors in the system, the number of executable tasks used by applications that cannot be controlled, and the number of applications that the system is controlling (i.e., applications which have processors allocated) from the system kernel. With this information, the server can decide the new partitioning of the processors. The applications then periodically check with the server to update the status. The applications may dynamically relinquish some processors or assign tasks to newly allocated processors according to the new system status. This implementation still requires a large amount of communication overhead, which causes performance problems.

As implied by its name, there is no control unit or sub-system that makes the global decision for allocation of processors in *self-scheduling approach*. The processors have to determine which tasks should be executed next by themselves. This approach is generally adopted in many studies [Dandamudi 1990, Nelson et al. 1988, Ni & Wu 1989, Zahorjan et al. 1991] due to its simplicity.

The strategies using this approach do not require any additional hardware. Strategies can be implemented in software. When a processor is idle, it directly accesses the ready queue to schedule the next task. The failure of a processor only reduces the processing

power by one. Furthermore, these strategies do not require any global information. There is no additional communication overhead.

Figure 2.2: Basic structure of server-based system
[Gupta & Tucker 1989]

However, the exclusive access to the ready queue may cause the software bottleneck in this approach. Many previous studies [Leutenegger & Vernon 1990, Zahorjan et al. 1991, Zahorjan & McCann 1990] generally ignored access contention for the ready queue in their performance evaluation. As addressed in [Ni & Wu 1989] and [Dandamudi 1990], ready queue access contention is a serious problem in this approach. When the overhead associated with accessing the ready queue is large, the ready queue becomes the system bottleneck.

In order to avoid the serious contention for the ready queue, Ni and Wu [1989] replace a single queue by a set of queues and organize them in distributed fashion. A subset of processors are linked to each ready queue. If a processor is idle, it only accesses the linked queue for executable processor. But 'load-imbancing' may exist in this distributed organization because there are several independent queues.

Because of the disadvantages associated with the existing organizations, a new organization is proposed with a hierarchical structure [Dandamudi 1990]. As shown in Chapter 4, the new organization incorporates the good features of both organizations.

2.2 Motivation of Hierarchical Organization

From the discussion in Chapter 1, the following design trade-offs can be identified. The centralized organization is preferred when the overhead associated with accessing the ready queue is small or negligible and/or for small system sizes. In this case, the load-sharing property of this organization is very attractive. Furthermore, the centralized organization is less sensitive to task service time variance. This centralized organization is widely used in several parallel systems including the Dynix operating system for the Sequent Balance [Lovett & Thakkar 1988] and the Mach Operating System [Black 1990]. However, the major drawback is that the central task queue will become a bottleneck for large system sizes and therefore it is not suitable for large parallel systems or fine granularity task scheduling.

The distributed organization, on the other hand, eliminates this task queue bottleneck disadvantage of the centralized organization by associating a local task queue with each processor. A chief disadvantage of this organization is that there may be load imbalance in the sense that some local task queues may be empty while others have tasks waiting to be scheduled. Furthermore, this organization is more sensitive to task service time variance. For large task service time variances, the performance deterioration is substantial [Dandamudi 1990]. For low variability in task service times, the distributed organization is

preferred. However, in a general-purpose system, such an environment cannot be guaranteed.

It is ideal to have an organization that combines the desirable characteristics of both these organizations while avoiding the disadvantages associated with them (or at least substantially reduce their effects on performance). The goal is to have an organization that allows load sharing (as in the centralized organization) and that distributes the task queues to eliminate the bottleneck problem (as in the distributed organization). The use of a hierarchical organization has been proposed to achieve this goal.

2.3 Description of the Hierarchical Organization

In this organization, a set of ready queues is organized as a tree with all the processors attached to the bottom level of the tree (i.e., as leaf nodes). Figure 2.3 shows an example hierarchical organization for $N = 8$ processors with a branching factor $B = 2$. Each ready queue can then be viewed as a ready queue in a centralized organization serving only the tree nodes (these nodes could themselves be ready queues or processors) directly below it. The centralized organization can be considered as a special case of this structure with just one level (just the root node) having a branching factor equal to the number of processors in the system. At the other extreme, the distributed organization can be considered as a tree (actually, a forest) with branching factor one. The scheme proposed by Ni and Wu [1989], where they consider adding multiple global ready queues in order to reduce the access contention, can also be considered as a special case of this hierarchical organization.

In such a hierarchical organization, all incoming tasks are added to the root task queue. Note that the tasks in the root node can be scheduled on any processor in the system. Let L be the leaf node level (i.e., processor level) in the tree. When a processor is looking for work, it first checks its associated task queue at level $(L-1)$. If that queue is empty, it checks the parent node of this node at level $(L-2)$ and the process is repeated up the tree until the processor finds a task to be scheduled (unless the system is empty).

However, in order to reduce access contention at higher levels, when a task queue is accessed, a set of tasks rather than just one as in the centralized organization, are moved one level down the tree. The size of the set decreases progressively as one goes down the tree (taking into account the availability of increasing number of task queues). At the bottom of the tree this set size is reduced to just one task (corresponding to scheduling that task on the associated processor). This type of organization avoids ready queue bottleneck because the branching factor can be adjusted. However, removing a set of tasks rather than just one task does increase ready queue access time but only marginally.

Figure 2.3: Hierarchical organization for $N = 8$ processors with a branching factor $B = 2$ (SA: synchronization area)

Memory contention is a major concern in large multiprocessor systems. In order to avoid memory contention in the hierarchical organization, multiple task queues should not be assigned to a single memory module. It can be seen that the set of task queues that form the tree can all be distributed to different memory modules so that concurrent access is permitted (provided the interconnection network to allow particular permutation). For a system with N processors, maximum number of task queues that will be required when the branching factor $B = 2$ is equal to $(N-1)$. Since there are N memory modules, these queues can be distributed without causing any contention problems.

The tree branching factor is a design issue in the hierarchical organization. Normally, it is preferred to keep the tree as "bushy" as possible (i.e., a tree with a large branching factor) because it allows better load sharing and results in better performance. Taken to the extreme (branching factor is equal to the system size), this represents the centralized organization. However, the access contention for the ready queue imposes a limit on the maximum branching factor of the tree. Furthermore, reducing branching factor increases the height of the tree introducing (in addition to the ready queue imbalance associated with the distributed organization) a different type of overhead in that more task queues will have to be accessed. Therefore, the value of the branching factor is an important parameter of this organization.

The number of tasks transferred from a parent task queue to its child task queue is also an important performance issue in the hierarchical organization. If the value is too large, it creates the problem of load imbalance as in the distributed organization. When the value is too small, the contention at higher levels may exist. The impact of these parameters is studied in Chapter 4.

In a sense, this hierarchical organization is similar in spirit to the combining trees used to reduce the adverse effects of hot-spot contention in interconnection networks [Yew et al. 1987]. It is felt that such an organization is attractive for building large parallel

systems. This has been recognized by several other researchers as well. Hierarchical schemes have been used in various aspects of large system design. Some example systems using a hierarchical organization are Cm* [Swan et al. 1977], Cedar [Gajski et al. 1983] and EMMA [Appiani et al. 1989] systems. Several interconnection networks have also been proposed that utilize hierarchical structure [Wu & Liu 1981, Hwang & Ghosh 1987, Ghosh & Desai 1989, Dandamudi & Eager 1990].

CHAPTER III

SYSTEM AND JOB MODELS

3.1 Multiprocessor System Model

Figure 3.1 shows the architecture of the parallel system under consideration. The system contains N identical processors. These N processors are connected to N memory modules using a multistage interconnection network. Furthermore, there is assumed to be a front-end sub-system that is responsible for generating system load by creating jobs, splitting jobs into tasks, inserting the tasks into appropriate task queues in the system memory and joining the tasks at the end of the completion. The front-end architecture is used in many parallel systems (Intel Hypercube Machine and Connection Machine [Tucker & Robertson 1988]). The front-end system is connected to memory modules through another communication network.

The system is modelled with an abstraction consisting of a set of queues and N servers. The communication between processors and memory modules is assumed to have an average delay of Cd_{in} . Similarly, the communication between front-end system and memory modules is assumed to have an average delay of Cd_{fe} . The actual values of these delays vary in real systems and the values depend on the system load, the job characteristics and the characteristics of the interconnection network. The determination of the actual delay costs are beyond the scope of the study. Both values of Cd_{fe} and Cd_{in} are assumed to be deterministic throughout the whole study and are expressed as a percentage of the average tasks service time. Since the interest in this study is in the performance of the actual system, the front-end system is assumed not to be a bottleneck in the system. Also, for the same reason, the front-end communication network is assumed to be faster than the interconnection network connecting the processors and memory modules. In addition, the front-end processor is assumed to do the memory allocation to the data and code of tasks

during the task creation. It is assumed that code and data sharing does not cause any memory and communication contention.

Figure 3.1: System architecture

The amount of time needed to access (not including the waiting time to get exclusive access to the ready queue) the ready queue is modelled as a fraction f of the average task execution time. In a lock-based implementation, this corresponds to the lock hold time. Each time an idle processor accesses a ready queue, it is assumed to spend a constant amount of time (i.e., f/μ where $1/\mu$ is the average task execution time) independent of the actual number of entries in the ready queue i.e., ready queue access time is deterministic. Deterministic access time has also been used in [Ni & Wu 1989] and [Leutenegger & Vernon 1990]. This is true for simple scheduling algorithms such as FCFS where it simply involves removing the task at the head of the queue. For more involved scheduling algorithms the variance in access time can be made very small by maintaining additional variables as is done in the Mach operating system [Black 1990].

The value of f is varied up to 4%. These values by no means represent the entire spectrum of parallel processing systems. For example, dataflow systems may require larger values of f ($> 4\%$) as they belong to fine grain parallel systems (where a task could simply be a single instruction). However, as the results show, the associated tradeoffs can be studied sufficiently by using this range of f values.

It should be noted that the values of f not only depend on the type of parallel system under consideration but also on other factors such as the scheduling algorithm used within a given system. For the same system, if a complex scheduling algorithm that requires a close scrutiny of the ready queue is used (for example, smallest task first type of scheduling algorithm), it may inflate the value of f substantially. Furthermore, preemption tends to reduce the task size and, thus, results in an increase in f value. An advantage of expressing the ready queue access time as a fraction of the average task execution time is that the results can be used to compare different types of systems and/or to compare different scheduling strategies for the same type of parallel system.

In the hierarchical organization, the number of tasks transferred from a parent task queue to its child task queue (i.e., one level down the tree) in each queue access is given by the parameter Tr , which is the ratio of the number of tasks moved one level down the tree to the number of processors below the child task queue. In addition, B is used to represent the branching factor of the tree node. Table 3.1 summarizes the parameters used to model the multiprocessor system.

Parameters	Description
N	System size
Cd_{in}	Communication delay induced by the interconnection network in the actual system
Cd_{fe}	Communication delay induced by the interconnection network in the front end system
f	Queue access time (fraction of mean task service time)
Tr	Transfer factor (ratio of the number of tasks transferred to number of processors linked to the child queue)
B	Branching factor

Table 3.1: System parameters

3.2 Job Models

The parallelism in a job can be characterized by a number of different levels of detail [Sevcik 1989]. At the highest level, a single parameter can be used to characterize a parallel program or job. For instance, the *average parallelism*, which is defined as the average number of busy processors during an execution of the job when unlimited number of processors is available, could be used [Eager et al. 1989]. It has been shown that even such simple characterizations are very useful [Eager et al. 1989]. At the lowest level, a detailed data dependency graph could be used. But, as pointed out by Sevcik [1989], the amount of information contained in such a characterization is too specific (depends, for instance, on the actual input values) and too detailed to be of practical use in performance analysis. For more details on various levels of characterization, see [Sevcik 1989]. Low-level detailed job characteristics are not considered in this thesis. Similar approach has also been used in other performance studies [Leutenegger & Vernon 1990, Majumdar et al. 1988, Nelson et al. 1989, Towsley et al. 1990 and Zahorjan et al. 1990].

Two classes of job structures, *independent* and *dependent*, are considered. In the independent job model, also called simple *fork and join*, a job is composed of a set of independent tasks that can be run concurrently (Figure 3.2). The job completes when all of its component tasks are completed. Tasks within a job do not communicate with each other. As stated by Towsley et al. [1990], the fork-join type of job structure is a reasonable model for a large number of parallel applications. For example, in the parametric investigation of numerical models, numerical computations are executed independently of each other for different sets of input parameters. The results of any one computation are not useful until the entire set completes.

Figure 3.2: Fork-and-join job structure

In the dependent job model, tasks within a job depend on the execution of the others. The dependence is caused by the synchronization as discussed in Chapter 1. In the *locking* type of job, a single lock is assumed to be shared by the tasks within a job [Nanda et al. 1991]. Figure 3.3 shows the basic structure of the generic program executed by each task in the locking type of job. In each iteration, tasks go through the following sequence: local computation, accessing the shared memory, requesting the lock, entering the critical section

and releasing the lock. The first two steps are grouped as region 1 and the last three steps are grouped as region 2. Region 1 represents the independent computation phase and region 2 represents the critical section phase. This type of jobs is reasonable for the class of problems with a solution that changes the application status after a phase of computation. The status affects the execution of the rest of the application.

Figure 3.3: Generic program structure for each task T_i in lock accessing job model [Nanda et al. 1991]

In the *barrier synchronization* type of job structure, tasks iterate through a computation phase and a communication phase. In the computation phase, tasks are executed independently. Tasks wait for the other tasks to complete their computation phase and may communicate their results. This is represented by the communication phase. Figure 3.4 shows the basic structure of the generic program executed by each task in the synchronization barrier type of job. In each iteration, tasks first perform the local computation and then communicate with others. This type of job structure is exemplified by the n-body simulations of stellar or planetary movements, in which the movement of each body is governed by the gravitational forces produced by the system as a whole [Jones & Murta 1989]. The next movement of each body cannot start unless the movement of all the bodies in the current state have been completed.

Figure 3.4: Generic program structure for each task T_i in barrier synchronization job model

3.3 Queue Locking Model

In this thesis, the spin-waiting lock is considered to implement the exclusive access for the ready task queues. As discussed in Chapter 1, the usage of the spin-waiting lock may cause serious memory contention. The MCS lock [Mellor-Crummey & Scott 1990] avoids the contention problem.

Figure 3.5 shows the pseudo code for the MCS lock [Mellor-Crummey & Scott 1990] and Figure 3.6 shows the pictorial version of MCS locking protocol. A record type, which contains a queue link (pointer to the record) and a boolean flag, is the basic structure required by each processor using the lock. Processors holding or waiting for the lock are linked into the chain with the lock holder at the front of the chain. Each waiting processor spins on its own local flag (flag in its own record). The lock itself is a record that contains a pointer to the record for the processor at the tail of the chain, or a nil if the lock is not held. When the processor releases the lock, it resumes the second processor in the chain (if there is one). It changes the pointer in the lock if the lock is empty. Figures 3.6a and 3.6b show the system status before and after processor 1 requires a free lock. Figure 3.6c presents the situation after processor 2 and 3 have requested the lock (the lock is currently held by processor 1). The condition of the lock after processor 1 has released the lock is shown in Figure 3.6d.

Since the front-end processor must insert the new tasks before any processor can remove those tasks from the queue, the queue lock in the root node requires additional *producer-and-consumer* features. Instead of using additional locks to implement this feature, the structure of the lock used in the root node is modified. The modified lock is shown in Figure 3.7. Two additional pointer fields and a flag field are added to the lock record. One of the pointer fields indicates the processor record in the front of the chain and the other pointer is used to link the chain of front-end processors. The flag field presents the status of the queue. If the front-end processor finds itself being the only one in the

front-end link, the processor checks the status flag. When the flag indicates a set of processors are waiting for tasks, the front-end processor directly accesses the root queue and puts the new tasks without acquiring the lock. After the insertion, the front-end processor then acknowledges the processor at the front of the chain to resume scheduling and execution.

```

type qlink = record
  next : ^qlink
  locked : Boolean

type lock = ^qlink

processor private I : ^qlink
  // initialized to point to a queue link record in
  // the local portion of shared memory

procedure acquire_lock (L : ^lock)
  var predecessor : ^qlink
  I -> next := nil
  predecessor := fetch_and_store (L, I)
  if predecessor != nil           // queue was non-empty
    I -> locked := true
    predecessor -> next := I
    repeat while I -> locked      // spin

procedure release_lock (L : ^lock)
  if I -> next = nil              // no known successor
    if compare_and_swap (L, I, nil)
      return // assuming compare_and_swap returns true iff it swapped
    repeat while I -> next = nil  // spin
  I -> next -> locked := false

```

Figure 3.5: Algorithm of the MCS list-based queueing lock
[Mellor-Crummey & Scott 1990]

Figure 3.6: Pictorial version of MCS locking protocol

Figure 3.7: Extended version of queue lock applied to the root node

In addition, when both the front-end processor and system processor simultaneously request the lock in the root queue, the front-end processor acquires the lock if the number of tasks in the queue is less than the number transferred from the root queue to its child

queue; otherwise, the processor in the system receives the lock and removes tasks from the queue.

The lock mechanism is modelled as follows:

- If a processor requires the lock, a delay of Cd_{in} is required to swap the pointer. After the delay time, the processor gets the permission to access the ready queue if the lock is free; otherwise, it is inserted at the end of the lock chain, spins in its local memory (no effect on the network traffic) and waits for the signal.
- When a processor releases the lock, it signals the second processor from the chain to indicate that the lock is free; otherwise, it swaps the pointer in the lock with nil pointer. After a delay of Cd_{in} , the second processor in the chain (if one exists) is acknowledged to acquire the lock.
- The processor then accesses the memory module into which the ready queue is mapped after obtaining the lock. Thus, it requires additional Cd_{in} time units before it can access the queue.
- The same mechanism is applied to the front-end processor when inserting executable task(s) into the root queue. The delay is Cd_{fe} time units in this case.

In summary, the queue accessing time is $(\frac{2Cd_{fe}+f}{\mu})$ time units by the front-end processor and $(\frac{2Cd_{in}+f}{\mu})$ time units by processors in the system. These two values are important in determining the upper bound of the system performance.

CHAPTER IV

PERFORMANCE WITH THE INDEPENDENT WORKLOAD

In this chapter, the performance of the three organizations - the centralized, distributed and hierarchical organizations - with the independent job model (described in Chapter 3) is studied. Section 4.1 describes the workload model used in the experiments. Section 4.2 presents the results that compare the performance of the three organizations. Section 4.3 studies the impact of the parameters of the hierarchical organization on performance. Section 4.4 discusses the simulation results and a chapter summary is given in Section 4.5

4.1 Workload Model

The independent job model is used in this chapter. It is assumed that the job arrivals form a Poisson process with parameter λ (i.e. job interarrival times are exponentially distributed). The jobs are partitioned into a set of tasks with a mean value of T and a coefficient of variation C_T . Tasks are characterized by their processor service demand (with mean $1/\mu$ and a coefficient of variation C_S). For the results presented in this chapter, exponential distribution is used for both the task distribution and the task service demand distribution (both C_T and C_S are equal to 1) except Section 4.2.3, which presents the impact of the variability in the task service demands. This model is identical to the workload B model in [Majumdar et al. 1988], and the high- C_n and uncorrelated workload model in [Leutenegger & Vernon 1990].

The primary performance measure obtained from the model is the average job response time. However, synchronization time, which is defined as the time between the completion of the first task and the last task of the same job, statistics are also collected.

Part of this chapter appears as [Cheng & Dandamudi 1992a].

While the synchronization time values are not presented explicitly (except in Section 4.2.1), these are used to explain the behaviour of the system. Also, the information about the average number of queue accesses per task is gathered. This value represents the additional network traffic induced by the hierarchical organization.

4.2 Performance Comparisons among the Three Organizations

In this section, the performance of the three organizations is studied. Section 4.2.1 presents the results that compare the mean job response times for different utilization. Section 4.2.2 shows the impact of the average number of tasks per job on the performance of these organizations. Section 4.2.3 presents the impact of service time coefficient of variation (CV). Section 4.2.4. presents the performance in a large system with 128 processors.

Parameter	Description	Default values
N	System size (Number of processors)	64
Cd_m	Communication delay in IN	0.005
Cd_e	Communication delay in FE	0.0005
f	Queue access time	0.02
Tr	Transfer factor	1
B	Branching factor	4
λ	Arrival rate	0.75
μ	Task service rate	1
Cs	Service time variance	1
T	Average task per job	64
C_T	Task distribution variance	1

Table 4.1: Default parameter values used in the performance comparison

The default parameter values used in this performance comparison are summarized in Table 4.1. In the simulation experiments, λ is varied (and μ is fixed) to vary utilization ρ ,

representing an increase in the load of the same type of jobs. Since $\mu = 1$ time unit, f value can be interpreted as the ready queue access time. For example, $f = 1\%$ indicates that it takes 0.01 time units to remove/insert an entry from/into the ready queue (note here the time unit corresponds to the mean task service time). Similarly, $Cd_{in} = 0.5\%$ indicates that it takes 0.005 time unit for a processor to access a memory module.

To ensure small confidence intervals, the information is only collected after the measured throughput had reached 99.5% of the arrival rate parameter. Then, the batch strategy is used to compute the confidence intervals (at least 30 batch runs were used for the results reported here and each batch contains more than 30000 jobs). This strategy produced 95% confidence intervals that were less than 1% of the mean response times when the system utilization is low to moderate and less than 10% for high system utilization but below saturation.

In the following discussions, the round robin task routing strategy is used to assign tasks to the local task queues in the distributed organization. There are adaptive routing strategies [Nelson et al. 1988, Dandamudi 1991] that provide a better performance than the round robin strategy. These strategies, however, require additional work by the front-end processors. Therefore, the round-robin strategy is used. The two structural parameters used in the hierarchical organization, the branching factor (B) and the transfer factor (Tr), are set equal to 4 and 1 respectively. Section 4.3 presents the impact of these two structural parameters.

4.2.1. Impact of Access Contention

As noted before, ready queue access contention is expressed in the simulation model by parameter f . Figure 4.1 presents the impact of ready queue access time as a function of utilization for the three organizations. The results in Figure 4.1a show that the centralized organization is extremely sensitive to access contention. For example, when the ready queue access time is 2% of the average task execution time, the saturation utilization (and

hence the throughput) falls to less than about 0.5. In the centralized organization, since there is only one ready queue global to all processors, it becomes a 'hot-spot' leading to a behaviour similar to that observed in interconnection networks [Pfister & Norton 1985].

(a)
Centralized organization

Figure 4.1: Performance of the three organizations as a function of the system utilization

(b)

(c)

Figure 4.1: (continued)

Figures 4.1b and 4.1c present the performance of the distributed and hierarchical organizations for $f = 2\%$ and 4% , respectively. In contrast to the sensitivity of the centralized organization to f values, the distributed and hierarchical organizations exhibit a robust behavior. The job response time for the distributed organization increases dramatically as the system load increases. This is mainly due to the lack of load sharing in this organization. Figure 4.2 shows the synchronization time as a function of utilization for the distributed and hierarchical organizations for $f = 4\%$. The synchronization time in the distributed organization grows faster than that in the hierarchical organization as the system load increases. It indicates the problem of load imbalance in this organization. On the other hand, the synchronization time in the hierarchical organization is insensitive to the system load because tasks within a job can concurrently be executed. In summary, the hierarchical organization provides a substantially better performance than both the centralized and distributed organizations. This shows that the hierarchical organization achieves the load sharing of the centralized organization without creating a task queue bottleneck.

Figure 4.2: Synchronization time in the distributed and hierarchical organizations as a function of the system utilization ($f = 4\%$)

4.2.2. Effect of Number of Tasks

This section investigates the impact of the number of tasks in a job on the performance of the three organizations. The number of tasks in a job is a function of the system architecture and the actual application. For example, as mentioned earlier, dataflow systems support (at least conceptually) very fine grain parallelism (at the instruction level). On the other hand, multiprocessor systems support medium to coarse granularity of tasks. Even for a given system architecture, the scheduling policy and the type of application (among other factors) determine how an application is partitioned into tasks. For example, in a multiprocessor system, if the number of processors assigned to a particular application can be determined *a priori*, this knowledge can be used to partition the application to match the number of processors allocated to the job. While some applications may not lend themselves well to this scheme, applications involving operations such as matrix operations are well suited for this type of problem partitioning.

Two types of workloads are used to examine the impact of average number of tasks per job on system performance: *fixed task size* and *fixed job size*. In the *fixed task size* workload, the average task size (i.e., average task service demand) is fixed and the average number of tasks per job is varied to represent different job size. For example, in applications such as parallel mean value analysis [Zahorjan & McCann 1990] for product form queuing networks, the mean task service demand is approximately the same because all tasks perform an identical amount of work. However, as pointed out by Zahorjan and McCann [1990], there will be some variations caused by different cache hit ratios and contention for access to the shared memory. In other applications involving I/O, such as database applications, the variations might be much larger due to, for example, seek time variations. This type of applications are modelled by varying the task sizes exponentially. Larger jobs are modelled by increasing the average number of tasks (again, exponential distribution is used for the number of tasks per job).

In the *fixed job size* workload, applications can be partitioned into varying number of tasks. For example, in applications involving matrices such as matrix addition, the submatrix size can be determined by the number of processors allocated for that application. For instance, in static scheduling, where the number of processors allocated to a particular application remains constant throughout the life time of that application, the application can take advantage of all the processors allocated to the application during the phase involving these matrix operations. This type of application is modelled by keeping the average job cumulative demand fixed and varying the number of tasks per job. Thus, the larger the number of tasks, the smaller is the task size.

The results with the *fixed task size* workload are shown in Figure 4.3. The parameter values used are: $N = 64$, average task service time $(1/\mu) = 1$ time unit, and $\lambda = 0.75$. Figure 4.3a shows the response time values for the centralized organization while Figures 4.3b and 4.3c give the results comparing the distributed and hierarchical organizations for $f = 0\%$ and 3% , respectively.

It can be seen from Figure 4.3a that the centralized organization is sensitive to the number of tasks per job (T) and the ready queue access time (f). As discussed in Section 2.2, the hot-spot nature of the single global ready queue imposes a limit on the job size.

In contrast, the distributed and hierarchical organizations exhibit a robust behavior, as shown in Figures 4.3b and 4.3c, because of the distributed nature of the ready queues. The distributed organization performs worse than the hierarchical organization because of its lack of load sharing characteristic.

(a)
Centralized organization

(b)

Figure 4.3: Performance of the three organizations as a function of average number of tasks per job with fixed task size workload

(c)

Figure 4.3: (continued)

Figure 4.4 shows the response times of the distributed and hierarchical organizations with the *fixed job size* workload for $f = 4\%$ (0.04 time unit for the whole experiment). The average job service demand is fixed at 64 time units. The system size (N) is 64 and arrival rate (λ) is 0.75. Thus, the utilization (ρ) is fixed at 75%. Since the previous results in this chapter have shown that queue contention imposes a limit to the centralized organization on task granularity, only the distributed and hierarchical organizations are considered in this experiment. The results show that both the distributed and hierarchical organizations give better performance by increasing the parallelism within the applications. For all the task sizes, the performance of the hierarchical organization is better than that of the distributed organization (50% performance improvement).

Figure 4.4: Performance of the distributed and hierarchical organizations as a function of the average number of tasks per job with fixed job size workload ($f = 4\%$)

4.2.3 Sensitivity to Service Time Variance

This section studies the impact of service time coefficient of variation (C_s) on performance of the three organizations. Figure 4.5a shows the average response time versus the service time CV for the three organizations. The graphs are plotted for $f = 0\%$ and $\lambda = 0.75$. $f = 0\%$ is used basically to compare the performance of the hierarchical organization with that of the centralized organization in the absence of access contention. The results indicate that the hierarchical organization performs as well as the centralized organization. This shows that the hierarchical organization inherits the load sharing property of the centralized organization. The distributed organization is sensitive to the service time CV for reasons explained below.

Figure 4.5b compares the performance of the distributed and hierarchical organizations for $f = 4\%$ and $\lambda = 0.5$ and 0.75 . There are two major observations

concerning Figures 4.5a and 4.5b. First, the distributed organization is more sensitive to service time CV than the hierarchical organization. Second, this sensitivity of the distributed organization increases with the system load and the value of queue access overhead (f). In contrast, the hierarchical organization exhibits less sensitivity to service time CV and is independent of both the system loads and f values considered here.

Figure 4.5: Performance sensitivity of the three organizations to the task service time variance

(b)

Figure 4.5: (continued)

The sensitivity of the distributed organization to service time CV and system load can be attributed to its lack of load sharing that is present implicitly in the centralized and hierarchical organizations. A high CV implies that there is proportionately a large number of tasks with small service times and a comparatively small number of tasks with a very large service time. Such large service time tasks not only increase the response times of the jobs that they are part of but also affect the response times of other jobs. This is because other jobs may wait for their last tasks to be completed and this task might be queued after a large task of another job. This effectively increases the synchronization time. This observation is supported by the simulation results. As shown in Figure 4.6, the synchronization times of jobs increase substantially as the service time CV increases. It is straightforward to see that this effect becomes more pronounced with increasing system load. In contrast, in the hierarchical organization, even if a large task keeps a processor

busy for a long time, other processors can handle the remaining tasks in a load sharing fashion.

Figure 4.6: Synchronization time in the distributed and hierarchical organizations as a function of service time variance

4.2.4 Performance in Large System

This section studies the performance of the distributed and hierarchical organizations when the system size, N , is increased to 128. Since the results in previous sections have shown that the access contention in the global ready queue limits the system size in the centralized organization, only the distributed and hierarchical organizations are considered in these experiments. Two workloads are used to investigate the performance of the two organizations: one has twice the number of tasks per job and the other has twice the average task service demand. In both the workloads, the average service demand is 128 time units, i.e., twice the value used in the previous experiments of this section. The branching factor

in the hierarchical organization remains the same as before (i.e., $B = 4$) except that the root queue has a branching factor of 2.

Figure 4.7 shows average job response times of the two organizations as a function of system load when the task service time demand is doubled. The parameter values used are: $N = 128$, average tasks per job = 64, average task service time ($1/\mu$) = 2 time units, $f = 0.04$, $Cd_{in} = 0.005$ and $Cd_{fe} = 0.0005$. Comparing with the results shown in Figure 4.1c, both the distributed and hierarchical organizations exhibit similar performances. However, for all system loads, the hierarchical organization provides better performance with increased system size because of its load sharing property.

Figure 4.7: Performance of the distributed and hierarchical organizations as a function of system load when the task service demand is doubled

Figure 4.8 presents the performance of the two organizations when the average number of tasks per job is doubled ($T = 128$). The parameter values used are $N = 128$, average task service time ($1/\mu$) = 1 time unit, $f = 0.04$, $Cd_{in} = 0.005$ and $Cd_{fe} = 0.0005$.

Similar to the results presented in Figure 4.7, the performance of both the distributed and hierarchical organizations is almost identical to that shown in Figure 4.1c even though the scale is doubled.

Figure 4.8: Performance of the distributed and hierarchical organizations as a function of system load when the average number of tasks per job is doubled

4.3 Sensitivity to Structural Parameters

In Section 2.2, two important parameters were mentioned for the hierarchical organization: the branching factor (B) and the transfer factor (Tr). This section considers the impact of these two structural parameters on the performance of the organization. Section 4.2.1 presents the impact of the branching factor (B). Section 4.2.2 gives the sensitivity of the hierarchical organization to the task transfer factor (Tr). Other parameter values used in the experiment are the same as those shown in Table 4.1. The default values of these two parameters are 4 and 1, respectively, except in the sections which the parameters are discussed.

4.3.1 Impact of Branching Factor

Branching factor is a design issue in the hierarchical organization. For better load-sharing and performance, the tree should be kept as "bushy" as possible (i.e., keep the height of the tree as small as possible). However, the access contention for the ready queue imposes a limit on the maximum branching factor of the tree. Figure 4.9 presents the ratio of the average mean response time with different branching factors (2 and 8) to that with a branching factor of 4 for $f = 2\%$ and 4% respectively. The figure demonstrates that the system with a branching factor of 2 has the worst performance because of the additional queue accesses (due to increased height of the tree). The performance deteriorates with increasing f values. On the other hand, using a branching factor of 8 provides only marginal performance improvement over that obtained with a branching factor of 4. It should be noted that the performance of the system is fairly insensitive to the branching factor unless the system utilization is high.

Figure 4.9: Impact of the branching factor on the performance of the hierarchical organization

4.3.2 Impact of Task Transfer Factor

Transfer factor is another parameter used in the hierarchical organization. Recall that the number of tasks moved from a parent queue to a child queue is given by:

$$\begin{aligned} &\text{number of tasks moved one level down the tree} \\ &= Tr * \text{number of processors in the sub-tree rooted at the child queue} \end{aligned}$$

Figure 4.10: Impact of the transfer factor on the performance of the hierarchical organization

The number of queue accesses reduces as the transfer factor Tr increases but this will result in reduced load sharing capability. The ratio of average mean response time with different transfer factors ($Tr=2$ and $Tr=0.5$) to that with a transfer factor of $Tr=1$ for $f=2\%$ and 4% is shown on Figure 4.10. It can be seen from Figure 4.10 that using $Tr=2$ results in marginal performance deterioration unless the system load is high. As noted earlier, increasing Tr implies bringing more tasks down the tree, which diminishes the scope of load sharing. The associated reduction in the number of queue accesses (see

Figure 4.11) assume significance at high system loads. For similar reasons, performance deteriorates at high system loads when $Tr = 0.5$. It should be noted that unless the system load is high, the performance difference is less than 10%. But the number of queue accesses is reduced when a larger transfer factor is used (see Figure 4.10 for $f = 4\%$) and this is preferable in a system where the network traffic is critical.

Figure 4.11: Average number of queue access as a function of system utilization

4.4 Discussion

From the results presented in Section 4.2, the following observations can be made. The queue access contention has a serious effect on the performance of the centralized organization. As shown in Figures 4.1 and 4.3, the saturation values for the system size and the average parallelism of applications in the centralized organization is dramatically reduced when the overhead associated with accessing the ready queue increases. The distributed and hierarchical organizations, on the other hand, are proved to be not affected

by this contention. The performance of the distributed and hierarchical organizations are only bounded by the processor saturation.

As mentioned in [Majumdar et al. 1988], the variability in the task service demand of applications is the major parameter affecting the performance. This parameter demonstrates the problem of load imbalance in the distributed organization. The response time of applications increases dramatically as the system load increases. The centralized organization uses a single global ready queue that is shared by all the processors. The organization implicitly provides dynamic load sharing. The results in Figure 4.4a show that the hierarchical organization inherits the load sharing characteristic of the centralized organization.

The results shown in Figures 4.7 and 4.8 demonstrate that the performance of both the distributed and hierarchical organizations is independent of the system size. However, the distributed organization does not perform as well as the hierarchical organization because of its lack of load-sharing. In both the workloads used in Figures 4.7 and 4.8, the mean job response time in the hierarchical organization is found to be about 50% of that in the distributed organization. The results imply that the hierarchical organization is more suitable to organize the ready tasks in large-scale systems.

In Chapter 2, it has been conjectured that the two structural parameters might have a serious effect on the performance of the hierarchical organization. The results show that the performance of the hierarchical organization is fairly insensitive to both these parameters. As stated in Section 2.4, a system with a large branching factor provides a better performance because it allows for better load sharing. The results support this observation. The system with a branching factor of 8 has the best performance, and the system with a branching factor of 2 has the worst performance. When the system utilization is high, the highest branching factor is preferred because it eliminates the processor saturation.

Ideally, the transfer factor should be as small as possible to increase the scope of the load sharing. Acting against this is the increased number of queue accesses. The results presented in this chapter show that a transfer factor of 2 does not cause any major performance deterioration. Actually, it is preferable ($Tr = 2$) because it reduces the number of queue accesses and, thus, the network traffic.

4.5 Summary

In this chapter, it has been shown that the queue access contention and load imbalance have substantial impact on the performance of multiprocessor systems. The load-sharing property of the centralized organization and its insensitivity to task service time variance make the organization preferable in organizing ready tasks. However, the major drawback is that the central task queue will become a bottleneck for large system sizes. On the other hand, the distributed organization in which each processor is associated with a local task queue eliminates this disadvantage of the centralized organization. The chief disadvantage of this organization is the existence of load imbalance.

To incorporate the good features of the two task queue organizations, the hierarchical organization has been proposed. The results demonstrate that the hierarchical organization is suitable to organize the ready tasks in large-scale systems. The organization inherits the load sharing property of the centralized organization and avoids queue access contention. The results also show that the hierarchical organization is fairly insensitive to the structural parameters.

CHAPTER V

IMPACT OF SCHEDULING POLICIES

5.1 Introduction

In the last chapter, the hierarchical organization has been shown to be more effective in organizing ready tasks in large-scale systems. In this chapter, the effect of job and task scheduling policies on the performance of the hierarchical organization is studied.

Previous studies [Majumdar et al. 1988, Leutenegger & Vernon 1990] have shown that scheduling policies have significant impact on the system performance. The results of these studies can be summarized as follows:

- The strategies based on job characteristics provide a much superior performance relative to a simple *FCFS* policy.
- The variability in the service demand of jobs and in the number of tasks per job are the major parameters affecting the system performance.
- Round robin policy has better performance than the *FCFS* policy because the round robin policy prevents the total monopolization of processing resources by large jobs.
- Preemptive policies perform well when the variance in the parameters is high.
- In lock accessing synchronization workload, non-preemptive (i.e., run-to-completion) policies are seriously affected because the tasks cannot be preempted.

Scheduling can be considered at two levels: *job* level and *task* level. Job scheduling policies consider the order of jobs to be executed and the scheduling policies applied at the task level consider how the tasks are to be executed regardless of which job the task is coming from. Section 5.2 discusses the impact of job scheduling policies on the

Part of this chapter appears as [Cheng & Dandamudi 1992b].

performance of the hierarchical organization and Section 5.3 provides the performance of the hierarchical organization for different task scheduling policies.

The independent workload that was used in the last chapter is considered. The default parameter values are summarized in Table 5.1. The additional parameters used in these experiments are: window size (W_s), quantum size (Q) and context switch overhead (O_c). A detailed description of these parameters is given in Section 5.3

Parameters	Description	Default Values
N	System size (Number of processors)	64
Cd_{in}	Communication delay in IN	0.005
Cd_{fe}	Communication delay in FE	0.0005
f	Queue access time	0.02
Tr	Transfer factor	1
B	Branching factor	4
μ	Task service rate	1
C_s	Service time variance	1
T	Average number of tasks per job	64
C_T	Task distribution variance	1
W_s	Window size	2
O_c	Context switch overhead	0.01
Q	Quantum size of Round Robin policies	0.5

Table 5.1: Default parameter values used in the task and job scheduling experiments

5.2. Performance of Job Scheduling Policies

In this section, the impact of job scheduling policies on the performance of the hierarchical task queue organization is considered. Note that the job scheduling policies affect the task queuing only at the root node of the hierarchical structure.

5.2.1 Job Scheduling Policies

Three job scheduling policies are considered here. The first policy is a job-independent policy. The remaining two policies are job-dependent in the sense that they require information on job characteristics to make their scheduling decisions. In order to isolate the effect of task scheduling on the performance, FCFS task scheduling policy is used in this section. The next section discusses the impact of task scheduling policies on the performance of the hierarchical organization.

(a) First-Come-First-Served (FCFS): In this policy, jobs (i.e., all the tasks that constitute a job) are queued in the order of their arrival. These tasks are removed from the root queue depending on the task scheduling policy in effect (which, in this section, is assumed to be the FCFS policy).

(b) Smallest Number of Task Job First (SNJ): [Majumdar et al. 1988] This policy uses the number of tasks into which a job is partitioned as an indicator of the job size. In this policy, the root node orders the jobs by their job size as indicated by the number of tasks. Note that this policy requires knowledge about jobs.

(c) Smallest Service Demand Job First (SDJ): [Majumdar et al. 1988] This policy assumes that *a priori* knowledge about jobs is available in the form of cumulative service demand of all its tasks. When such knowledge is available, the root node orders the jobs in an increasing order of their service demand. However, it should be noted that *a priori* information is not often available. The performance of this policy can be used to compare the performance of other less demanding policies such as the SNJ policy.

Note that the last two policies are not fair in the sense that a job with a large service demand or with a large number of tasks could potentially wait in the queue for a long time depending on the system load.

5.2.2 Simulation Results

This section presents the results of the simulation experiments that compare the performance of the three job scheduling policies. In these experiments, the independent job model is considered. The default parameter values used in the experiments are shown in Table 5.1. FCFS task scheduling is assumed in this section. The impact of task scheduling policies is presented in the next section.

5.2.2.1 Performance Comparison

Figure 5.1 shows the job response time as a function of system load. The following observations can be made from the data presented in this figure.

- Both SDJ and SNJ policies have a better performance than the FCFS policy. The relative performance deterioration of the FCFS policy increases with increasing system load.
- Interestingly, both the job-dependent policies provide similar performance. It should be noted that the number of tasks that a job is partitioned into and the service times of these tasks are independent, which could lead to some potential differences in the system behaviour. For example, it is possible for a job to have fewer tasks than another job but have a cumulative service demand that exceeds that of the other job. Earlier studies using the centralized task queue organization have shown that SDJ performs better than the SNJ policy [Majumdar et al. 1988, Leutenegger & Vernon 1990]. However, in the hierarchical organization, there is no performance advantage associated with the SDJ policy despite the fact that this policy requires a detailed *a priori* information in the form of task service times.

Figure 5.1: Performance of job scheduling policies as a function of system load

5.2.2.2 Sensitivity to Task Service Time Variance

The data presented in Figure 5.1 assumed an exponential task service time distribution (i.e., service time CV = 1). This section discusses the impact of the task service time variance on the job response time. Figure 5.2 shows job response time as a function of task service time variance. All the other parameters are set as in Figure 5.1. This figure shows job response times at two system utilization levels: 50% and 85%. It can be seen from this data that task service time CV affects all three policies in the same way (slope of the curves is similar) and is more or less independent of the system load. A consequence of this is that the percentage of the performance deterioration of the FCFS policy decreases with increasing task service time variance. Also note that the SDJ and SNJ policies provide similar performance and the task service time variance has no noticeable impact on their performance difference.

Figure 5.2: Performance sensitivity of scheduling policies to the service time variance at the system load of 50% and 85%

5.2.2.3 Impact of Task Distribution Variance

In Section 5.2.2.1, the number of tasks/job is assumed to follow the truncated exponential distribution. The performance sensitivity to this parameter is discussed in this section. Figure 5.3 shows the job response time as a function of variance in task distribution. All the other parameters are set as in Figure 5.1. As in Section 5.2.2.2, job response times are shown at two system utilization levels: 50% and 85%. This data shows that the FCFS policy is more sensitive to the variance in task distribution than the other two policies. Furthermore, this sensitivity increases with system load. As with task service time variance, the task distribution variance has no impact on the performance difference between the SDJ and SNJ policies. The job response time associated with the SDJ and SNJ policies decreases marginally with increasing variance in task distribution. This is due to the fact that, with increasing variance, there will be more jobs with small number of

tasks/small cumulative service demand; therefore, it allows more jobs to be executed concurrently and thereby reduces the job response time.

Figure 5.3: Performance sensitivity of scheduling policies to task distribution variance at the system load of 50% and 85%

5.2.3 Discussion

This section has investigated the impact of three job scheduling policies on the performance of the hierarchical task queue organization. As can be expected, the job-independent FCFS policy provides the worst performance among the three policies. While this policy is no more sensitive to the task service time variance than the other two policies, it exhibits more sensitivity (than the other two) to the variance in task distribution and this sensitivity increases with system load.

Interestingly, the two job-dependent policies provide similar performance despite the fact that the SDJ policy requires a detailed *a priori* task service time information, which is often difficult to get in practice. The study indicates that the same performance can be

obtained by simply using the number of tasks/job. It is also shown that the performance similarity is not sensitive either to the task service time variance or to the variance in task distribution.

5.3 Performance of Task Scheduling Policies

In the last section, the FCFS policy is used for task-level scheduling. This section studies the impact of task scheduling policies on the performance of the hierarchical task queue organization. In particular, the impact of preemption on the performance at this level of scheduling is studied. As a consequence, performance implications of four round robin-based preemptive task scheduling policies are investigated and their performance is compared to that of the FCFS task scheduling policy.

5.3.1 Task Scheduling Policies

Five task scheduling policies are considered in this section. The first policy is a non-preemptive FCFS policy used in the previous section. For completeness a brief description of this policy is included. The remaining four policies are preemptive policies based on the round robin discipline.

(a) First-Come-First-Served (FCFS): As described in Chapter 3, the number of tasks transferred from a parent queue to a child queue is equal to the transfer factor (Tr) times the number of processors below the child queue. At the bottom of the tree the number of tasks transferred is reduced to just one task (corresponding to scheduling that task on the associated processor). In this scheduling policy, tasks are run to completion without preemption.

(b) Round Robin policies: Implementation of round robin discipline in the hierarchical organization is difficult because the system does not have the global coordination unlike in the centralized organization. Since contention is a serious problem, round robin discipline

cannot be applied at the global level (as is done in previous studies, which neglected the task queue access contention). Also, since each task queue in the hierarchical structure is mapped to a memory module, implementing at the intermediate queue levels would create network contention (even if there is no contention for the task queue itself). In addition, this process would introduce substantial context switch overhead. For these reasons, the implementation of the round robin discipline is considered at the local queue level. Note that this local queue resides in the local memory of the processors and no interconnection network access is required to implement preemption. This substantially reduces the context switch overhead and thus may make the round robin policy attractive.

In a local round robin policy, a processor brings a set of tasks (equal to window size W_S) into its local memory and applies the standard round robin policy on this set. That is, each task is given the processor for a quantum Q and if not finished within that quantum, the task is preempted and returned to the tail of the associated local queue and the task at the head of the local queue is given the processor for the next quantum. Such preemption involves overhead - context switch overhead (O_C). Once all the tasks in the set are finished, a new set of tasks will be brought into the local memory to start another cycle of round robin scheduling. There are several ways to implement the local round robin policy depending on how the set of tasks are brought into the local memory from the root queue.

It has been shown that, in the centralized task queue organization, round robin policy outperforms the FCFS policy in most cases [Majumdar et al. 1988, Leutenegger & Vernon 1990]. These studies, however, assumed that there is no contention for the task queue and thus can be applied only to small systems (i.e., system with small number of processors). For large systems with hundreds of processors, round robin policy increases task queue access contention substantially due to frequent preemption of tasks. In order to make the round robin policy effective, the window size W_S has to increase so that a few large tasks do not monopolize the processors - the chief advantage of the round robin policy.

However, in the hierarchical organization, transferring too many tasks from the higher levels to lower levels diminishes the load sharing ability of the hierarchical organization. Keeping as many tasks as possible in the higher level queues closer to the root queue is preferred. This implies that the window size is reduced. In order to study this inherent trade-off, four variations to the basic local round robin discipline are considered. Since the conclusion is that the FCFS task scheduling policy is better than the local round robin policy, it is also important that we consider different variations to show the performance superiority of the FCFS policy.

(i) Round Robin 1 (RR1): This is an extension of the FCFS task scheduling policy discussed above. In each visit to a parent queue, the number of tasks moved from the parent to a child queue is equal to $Tr \times$ number of processors below the child queue (as in the FCFS policy). However, instead of moving one task from the leaf queue to the associated processor local queue, each processor brings W_s tasks to execute them in local round robin fashion. This process is illustrated in Figure 5.4 for a system with 64 processors. The hierarchical structure is assumed to have a branching factor of 4. When processor 1 is looking for work (assuming that both Q5 and Q1 are empty and Q0 has 100 tasks) 16 tasks are transferred from the root queue to Q1 assuming Tr to be 1. Of the 16 tasks, 4 tasks are transferred to Q5 leaving 12 tasks in Q1. Assuming that the window size W_s is 2, processor 1 moves 2 tasks into its local queue. Note that W_s cannot be arbitrarily large and is limited by the branching factor B and the transfer factor Tr .

(ii) Round Robin 2 (RR2): This is a dynamic policy in the sense that the number of tasks transferred to a child queue from its parent queue depends on the number of tasks queued at the parent queue. Thus, this policy responds to the system load by transferring more tasks when the system load is high. In order to reduce task transfer overhead, however, the number of tasks transferred in each access should not be too small. Here, the constraint that

each transfer should move (where possible) at least ($Tr * \text{number of processors below the child queue}$) is used. The number of tasks transferred from a leaf queue to the associated processor local queue is W_s . This approach differs from RR1 when the system load is high. The following figure (Figure 5.5) illustrates the behavior of the RR2 policy. The initial conditions of the queues are assumed to be the same as in the previous example. When processor 1 is looking for work (assuming that both Q5 and Q1 are empty and Q0 has 100 tasks) $1/4 * 100 = 25$ tasks are transferred from Q0 to Q1. One fourth of these 25 tasks are transferred to Q5 leaving 19 tasks in Q1. Assuming that the window size W_s is 2, processor 1 moves 2 tasks into its local queue. Note that larger W_s values can be accommodated in this scheme at high system loads.

Figure 5.4: Behaviour of the RR1 policy

Figure 5.5: Behaviour of RR2 policy

(iii) Round Robin 3 (RR3): A potential problem with the above policy (RR2) is that it may increase memory access overhead as the number of tasks moved is proportional to the number of tasks waiting in the parent queue. For example, with the system state as in Figure 5.5, if another processor (say processor 14 in Figure 5.6) accesses Q1, it would bring down only 5 ($19/4$) tasks leaving 14 tasks at Q1. The next time, only 4 ($14/4$) tasks would be moved. But, at the same time, RR2 policy increases scope for load sharing. One way to reduce this overhead is to transfer a number of tasks that is proportional to the number of tasks transferred to the parent queue (rather than the number of tasks currently at the parent node as is done in RR2). This policy is applicable to local queue transfers and therefore this policy does not need the parameter W_s . In order to implement this policy, an additional variable (say i) is needed. The i value of a queue is set to B when tasks are transferred to it from its parent queue, where B is the branching factor of the hierarchical organization. Each time a processor accesses a queue (except for the root queue) it transfers $1/i$ fraction of the tasks at that queue and decrements i . The process is illustrated in Figure 5.6. The initial conditions of the queues are assumed to be the same as those used in the

previous example. When processor 1 is looking for work (assuming that both Q5 and Q1 are empty and Q0 has 100 tasks) $1/4 * 100 = 25$ tasks are transferred from Q0 to Q1 and sets i value of Q1 to 4. One fourth of these 25 tasks are transferred to Q5 leaving 19 tasks at Q1 (and decrements i value to 3). This process is repeated at Q5. Now suppose that processor 14 is looking for work and Q8 is empty. Processor 14 accesses Q1 and transfers 6 ($19/3$) tasks to Q8 and decrements the i value of Q1 by one. This is in contrast to RR2 policy which would have transferred 5 ($19/4$) tasks to Q8. Thus RR3 policy decreases number of queue accesses at the cost of diminished load sharing ability.

Figure 5.6: Behaviour of RR3 policy

(vi) Round Robin 4 (RR4): This is a minor variation of the previous policy (RR3). This policy differs from RR3 policy in the number of tasks transferred to the processor local queue. RR4 policy transfers W_j tasks into the local queue as in the first two round robin policies.

5.3.2 Simulation Results

The results of the simulation experiments are presented in this section. The parameter values used in the experiments is identical to the values given in Table 5.1. Instead of presenting the average job response times, the response time ratios relative to the response times of the *FCFS* policy will be used.

5.3.2.1 Principal Comparison

Figure 5.7: Performance of task scheduling policies as a function of the system load ($C_S = 1$)

The performance of the five task scheduling policies as a function of system load ρ is shown in Figure 5.7. Note that the performance of the FCFS policy is better if the response time ratios are greater than 1 and the performance of a Round Robin policy is better if the curve lies below the unity line. It can be observed from this figure that the FCFS policy outperforms all four Round Robin policies. The performance advantage associated with the FCFS policy decreases with increasing system load for all RR policies (except for RR3

policy). All four RR policies show performance advantages only when the system load is greater than about 0.9.

The following observations can be made on the performance of the four RR policies.

- RR2 policy performs worse than the other three RR policies. Recall that RR2 policy transfers a number of tasks that is proportional to the number of tasks waiting in the parent queue at the time task transfer is attempted. Since there is a minimum number of tasks that it should transfer (and this minimum is the same as that used in RR1 policy) RR2 policy transfers more tasks only when the system load is high. A disadvantage with such a scheme is that it diminishes scope of load sharing as tasks are removed from the top levels of the hierarchical structure. On the other hand, such a policy reduces task transfer overhead. The results in Figure 5.7 show that the performance of RR1 and RR2 policies is the same for system loads below 0.4 but at higher system loads RR2 performs worse than RR1. This indicates that the decrease in task transfer overhead does not compensate well for the associated reduction in the scope of load sharing.
- RR3 policy provides better performance than RR4 for system loads below about 0.7. For higher system loads, performance of RR3 policy is worse than RR4 policy. Note that RR3 policy does not use a fixed window size (unlike the other three RR policies). The window size dynamically varies depending on the system load. For low system loads, the window size of RR3 may be smaller than 2 (the window size used in the other three policies); for high system loads, the window size of RR3 may be larger than 2. Since window size is the only difference between RR3 and RR4 policies, the difference in performance can be attributed to this parameter. This is also supported by the results presented in Section 5.3.2.4, which discusses the impact of the window size parameter.
- At very high system loads (i.e., at about greater than 0.9), reducing task transfer overhead is important because this reduces the amount of time processors spend in self-scheduling tasks. Thus policies that transfer larger number of tasks between levels of the

hierarchical organization will perform better. This is supported by the results presented in Figure 5.7. The results show that the performance of RR1 policy is worse than the other three RR policies. Since the other three policies respond to the system load variations, the performance difference among these three policies is marginal.

5.3.2.2 Performance Sensitivity to Task Service Time Variance

Figure 5.8: Performance of task scheduling policies as a function of the system load ($C_s = 7$)

The results discussed in Section 5.3.2.1 assumed that the task service times are exponentially distributed (i.e., task service time $CV = 1$). This section discusses the impact of this parameter on the performance of the task scheduling policies. Figure 5.8 presents the performance of the five policies as a function of the system load when the task service time CV is 7. Comparing Figures 5.7 and 5.8 it can be seen that increasing service time CV not only decreases performance difference among the RR policies but also decreases performance difference among the five policies. The four RR policies perform marginally

worse than FCFS policy for system loads less than about 0.8. At higher system loads RR policies perform better. However, RR policies provide substantial performance improvements only when the system load is higher than about 0.9 as shown in Figure 5.8. The performance difference is further demonstrated in Figure 5.9, which gives response time ratios as a function of task service time CV when the system load (ρ) is fixed at 0.85. These results show that the policies that transfer tasks depending on system load perform better. In Figure 5.9, RR1 policy performs worse than the other three policies for higher service time CVs.

Figure 5.9: Performance sensitivity to task service time variance ($\rho = 85\%$)

5.3.2.3 Impact of Variance in Task Distribution

The previous results in Section 5.3 assumed that the number of tasks/job follows (truncated) exponential distribution ($C_T = 1$). This section presents the results (Figure 5.10 for system load = 0.85) that show the impact of variance in task distribution. It has been established, in the context of the centralized organization, that the FCFS policy is sensitive

to task distribution CV. The RR policies are not as sensitive to this parameter. This is the reason for their improvement in performance. Since RR3 policy uses dynamic window size, the window size is dependent on the number of tasks present at the time task transfer is attempted. Thus, if a job just arrives with large number of tasks, the window size is automatically increased. As discussed in Section 5.3.2.1 (and also in Section 5.3.2.4) increased window size increases the job response time. This possibly explains the behaviour of RR3 in Figure 5.10. The results in this figure also suggest that transferring a number of tasks that is dependent on the system load is beneficial when there is large task distribution variance. For example, when task distribution CV is 4, both RR2 and RR4 show better performance than RR1. However, the performance difference is within $\pm 5\%$ (relative to that of the FCFS policy).

Figure 5.10: Performance sensitivity to task distribution variance ($\rho = 85\%$)

5.3.2.4 Effect of Window Size

The effect of the window size parameter is shown in Figure 5.11 (system load is 0.85). Since RR3 does not use window size parameter (W_s), the sensitivity of the remaining three RR policies is compared. It can be observed that increasing the window size increases job response time. This also explains the behavior of RR3 policy at different system loads. At high system loads, the RR3 policy performs worse because it uses a larger window size compared to the other three policies. The results in this figure also show that for large window sizes, transferring large number of tasks is beneficial because it reduces task transfer overhead. For example, in Figure 5.11, when the window size is 4, RR4 performs better than the RR1 and RR2 policies. Recall that the RR4 policy transfers larger number of tasks (proportional to the number of tasks present at the root node) than RR1; at intermediate levels, it transfers larger number of tasks than RR2 policy. Thus RR4 outperforms both these (RR1 and RR2) policies.

Figure 5.11: Performance sensitivity of RR policies to window size ($\rho = 85\%$)

5.3.2.5 Sensitivity to Quantum Size and Context Switch Overhead

The impact of the quantum size and context switch overhead on performance of the four RR policies is shown in Figures 5.12 and 5.13, respectively, for a system load of 0.85. All other parameter values are set to the default values. Previous results assumed a quantum size of 0.5 and a context switch overhead of 0.01 (i.e., 1% of the average task service time).

The results in Figure 5.12 show that decreasing quantum size increases job response time. These results are obtained with the assumption that context switch overhead is 0.01. The RR3 policy exhibits increased sensitivity to the quantum size than the other three RR policies due to its dynamic window size variation.

Figure 5.12: Performance sensitivity of RR policies to quantum size (Q)

It should be noted that quantum size (Q) and context switch overhead (O_c) are interrelated in the sense that for larger context switch overhead the quantum size should be increased in order to reduce the impact frequent context switching. Performance of the four

RR policies as a function of the context switch overhead is shown in Figure 5.13 (for a quantum size of 0.5). It is interesting to note that even when the context switch overhead is zero (i.e., no penalty in preempting a task), the performance of RR policies is only marginally better than that of FCFS policy (less than 4%). Of course, performance of RR policies can be improved further by decreasing the window size when there is no context switch overhead. As shown in Figure 5.13, RR3 policy exhibits increased sensitivity to context switch overhead than the other three RR policies due to its dynamic window size as explained before.

Figure 5.13: Performance sensitivity of RR policies to context switch overhead (Oc)

5.3.3 Discussion

This section has provided a performance comparison of five task scheduling policies - one non-preemptive and four variations of preemptive round robin policies. In the hierarchical organization, round robin policy could be applied only in a local context. This diminishes the impact of these preemptive policies. The results presented in this section indicate that the round robin policies do not offer any performance advantages unless the

system load is very high (more than 90% system utilization). The round robin policies are sensitive to quantum size and context switch overhead. It has been shown that even when there is no context switch overhead, round robin policies offer only marginal improvement in job response time (less than 4%) when compared that of the FCFS policy.

5.4 Summary

In this chapter, several task and job scheduling policies for the hierarchical task queue organization have been studied. Three job scheduling policies have been considered: job-independent FCFS policy, job-dependent smallest number of tasks job first (SNJ) and smallest service demand job first (SDJ). Expectedly, both the job-dependent policies, SDJ and SNJ, have been shown to perform well for all workloads including the ones with a high variability in both task service times and the task distribution. The results indicate that the SNJ policy provides similar performance as the SDJ policy, which requires *a priori* knowledge about the job service demands.

A performance comparison of five task scheduling policies - one non-preemptive and four variations of preemptive round robin policies - has also been provided. In the hierarchical organization, round robin policy could be applied only on processor local task queues because of the queue access contention. In contrast to the previous work, which used a centralized organization and considered round robin globally, the results in this study show that those preemptive local ready queue policies cannot offer any performance advantage over the simple FCFS policy. This is due to the fact that the number of tasks participating in round robin (the window size) is not large. Increasing the window size diminishes the scope of load-sharing. These policies represent a trade-off between increasing the window size and the scope of load-sharing.

CHAPTER VI

PERFORMANCE WITH SYNCHRONIZATION WORKLOADS

6.1 Introduction

The performance of the hierarchical organization with the independent job model has been studied in the last two chapters. In this chapter, two dependent job models - *lock accessing* model and *barrier synchronization* model (discussed in Section 3.2) - are considered. Both job models have been used in previous studies that used the centralized organization [Karlin et al.1991, Leutenegger & Vernon 1990, Zahorjan et al. 1989 and 1991]. Some of the conclusion of these studies are the following:

- Spinning policy performs well in the lock accessing model because the spinning time is expected to be very small.
- Spinning policy is not affected by the number of tasks in the system and job characteristics (in the lock accessing model); its performance is sensitive to the system size.
- Performance improvement can be obtained by using scheduling policies that do not swap out tasks that are holding locks.
- System performance can be improved if the tasks that are waiting for a lock are not scheduled until the lock is free.
- Spinning policy is extremely sensitive to the task service time variance in the barrier synchronization job model.
- Hybrid policies that combine both spinning and blocking perform well.

The trade-off between spinning and blocking is considered in this chapter. The remainder of the chapter is organized as follows. Section 6.2 describes the scheduling policies used in the dependent job models. Sections 6.3 and 6.4 present the results of the

simulation experiments using the lock accessing job model and barrier synchronization job model respectively. Section 6.5 concludes the chapter with a summary of the results for the dependent job models.

6.2 Scheduling Policies

Four task scheduling policies are considered in this chapter: a spinning policy, two blocking policies and a hybrid policy. The lock mechanism proposed in [Mellor-Crummey & Scott 1990] is applied to all task scheduling policies; thus, *FCFS* order in the lock acquisition is considered. The *FCFS* job scheduling is assumed in this chapter.

(a) Spinning policy: When a task requires a lock or waits at a barrier, the processor executing the task sits in a tight-loop and waits. Theoretically, deadlock may exist in the barrier synchronization workload if the number of tasks within a job inserted into a queue is greater than the number of processors linked to the queue (i.e., a task is placed in a queue and all the processor linked to the queue are spinning and waiting for the completion of the task). In order to prevent the deadlock, the number of tasks within a job moved down to a child queue is restricted to the number of processors linked to the queue. This restriction is a trade-off between deadlock prevention and load-balancing property and is discussed in Section 6.4.

(b) Blocking policies: Blocking policies can be implemented in several ways. The simplest one treats the blocked tasks as new tasks when they are activated. The tasks are then inserted into the root queue. Similar to the global round robin policies, queue access contention for the root queue could cause performance problems. Furthermore, these tasks in the root queue will have to wait for the processors to complete all the tasks in the child queue. The blocking time (the time interval between when the tasks are blocked and when they are re-scheduled) will be considerably large. Preliminary results have shown that this approach is not feasible.

In the other approach, each node in the hierarchical structure maintains two task queues: a new ready task queue and a blocked task queue. All the new tasks transferred from the parent queue are inserted into the first queue. The latter queue contains all the tasks that have been suspended. When a processor accesses a node for a ready task, it places its suspended task (if there is one) into the blocked task queue of the node. The control block of each task is assumed to contain a status flag which indicates the status of the task, *ready* or *suspended*. When a suspended task is activated, the processor, which previously executed the task, changes the status flag in the task control block. The next available processor accessing the node scans through the blocked task queue first and then removes the task to process. Since the suspended tasks are inserted into the blocked queue during the acquisition of the ready task queue, a single lock can be shared by both task queues in the node. The approach is simpler than the first one, and is found to be feasible. This approach is considered in this chapter.

Theoretically, processors may wait in the queue at higher levels of the tree when the suspended tasks are activated and are placed in the queue at lower levels. This situation frequently occurs at low system utilization and deteriorates system performance. In order to improve the performance, an interrupt is assumed to be sent to the processor that is holding the queue lock when the flag is changed. If the processor, which receives the interrupt, is waiting in the queue at a higher level the processor then releases all the queue locks it held (a communication delay overhead is induced for each released lock) and processes the task. Also, if the suspended task has been activated before the processor that previously processed the task can access a queue, the processor removes itself from the lock chain of the queue (a communication delay is induced by this operation) and processes the task.

(i) Blocking policy 1 (Static): In this policy, all the blocked tasks are only inserted back into the leaf node to which the processor is linked.

(ii) Blocking policy 2 (Dynamic): In contrast to the static policy, blocked tasks are inserted into a task queue at a higher level from which the processor gets ready tasks. This policy utilizes the load sharing property of the hierarchical organization.

(c) Hybrid scheduling policy (mix of spinning and blocking scheduling policy): In this policy, tasks that cannot obtain a lock or go through the barrier first spin for a while and then block. In the lock accessing model, the lock is assumed to be assigned to a task that is currently spinning. If none of the tasks is spinning, the lock is assigned to the first blocked task. The static blocking policy (blocking policy 1) is applied to the blocking part of the policy and the spinning interval is modelled by an additional parameter T_s .

6.3 Lock Accessing Model

6.3.1 Workload Model

The generic structure of a task in the lock accessing job model is shown in Figure 3.3. It is assumed that the job arrivals form a Poisson process with parameter λ (i.e. job interarrival times are exponentially distributed). The jobs are partitioned into a set of tasks T , which is uniformly distributed between 1 and the system size (N). Tasks are characterized by the number of iterations i , which is uniformly distributed between 1 and Max_i , the region 1 service demand (S_C) and the lock holding time (S_l) in each iteration. Tasks within a job are assumed to have similar behaviour, i.e., identical number of iterations and service demand in each iteration. In the model, the number of iterations and the mean service demand (S_C) in each iteration are determined when a job is created. The mean service demand (S_C) in each iteration is generated by an exponential distribution. The region 1 service time in each iteration is uniformly distributed between $(1 - V_{SC})$ and $(1 + V_{SC})$ of S_C . This kind of variability in the service demand has also been used in [Zahorjan et al. 1989, Zahorjan & McCann 1990]. The variance represents the variation in accessing the shared memory (i.e. second step in Figure 3.3).

The lock holding time is assumed to be deterministic and is a fraction of the average task service demand in a single iteration ($S_C + S_I$). The deterministic lock holding time has also been used in [Zahorjan et al. 1989] and [Leutenegger & Vernier 1990] except that the lock holding time considered in those studies is dependent on the number of tasks in the job.

6.3.2 Simulation Results

Table 6.1 summarizes the default parameter values used in the lock accessing workload experiments. With these values, the actual system load (percentage of the time spent by the processor executing tasks, excluding the queue accessing time and waiting time) is close to the arrival rate (λ). The lock holding ratio is given by $\frac{S_I}{S_C + S_I}$. In the experiments, the values of S_C and S_I are set equal to 0.24 and 0.01, respectively. Thus, the locking holding ratio is 4%. The spinning time (T_s) in the hybrid policy is equal to the queue access time (f). Section 6.3.2.6 studies the impact of this parameter on the performance of the hybrid policy.

Parameters	Description	Default Values
λ	Arrival rate	0.7
Max_i	Maximum number of iterations	15
C_{dn}	Communication delay in IN	0.005
C_{dfc}	Communication delay in FE	0.0005
f	Queue access time	0.02
Tr	Transfer factor	1
B	Branching factor	4
S_C	Mean service demand in region 1 per iteration	0.24
S_I	Lock holding time	0.01
V_{S_C}	Variability to mean service demand	0.5
T_s	Spinning time in hybrid policy	0.02

Table 6.1: Parameter values used in the lock accessing workload experiments

6.3.2.1 Principal Comparison

The performance of the four task scheduling policies as a function of system load (ρ) is shown in Figure 6.1. Ideally, the spinning policy performs well if the spinning time is less than the time required to obtain a ready task. In this model, the minimum time required by a processor to access a task queue is $(2Cd_{in} + f)$ (given in Section 3.3 - this value is 0.03 time unit in the experiments). In the experiments, the spinning time is always less than the time to obtain the lock. Thus, the spinning policy performs superior to the blocking policy except when the system load (ρ) is over 70% .

Interestingly, the job response time with the blocking policies increases dramatically with decreasing system load at low utilization ($< 20\%$). At such low system utilization (as discussed in Section 6.2), the processors frequently wait in the queues for a ready task. A system interrupt is usually required to resume a suspended task. And this overhead is large (at least one communication delay is induced by each operation). Thus, the performance of the policies deteriorates as the system load reduces close to zero.

The dynamic blocking policy (blocking 2) has a better performance than the static blocking policy (blocking 1) because the dynamic policy provides better load-sharing. The results imply that the suspended tasks should be inserted into the queue at the highest level. Unfortunately, the queue access contention may exist at the root queue if all the suspended tasks are to be inserted back in the root queue. Again, this demonstrates the trade-off between load-sharing and queue access contention.

Figure 6.1: Performance of scheduling policies as a function of system load (ρ) on the lock accessing workload

6.3.2.2 Performance Sensitivity to Service Time Variance

As defined before, the variability of the region 1 task service time (V_{Sc}) represents the variation in accessing the shared memory. When the value is close to 1, it reflects uncontrollable environment influences such as page faults and cache hits in I/O buffers. Figure 6.2 shows the performance sensitivity of the four task scheduling policies to the service time variance (V_{Sc}). The two blocking policies and the hybrid policy are not affected by the variance in task service demands. However, the job response time with the spinning policy decreases marginally with increasing variability. This is due to the fact that the lock contention is reduced by the high variability. Table 6.2 shows the average spinning time of suspended tasks with different variability.

Figure 6.2: Performance sensitivity of scheduling policies to the service time variance

Variability (VSc)	0.0	0.25	0.5	0.75	1.0
Spinning Time	0.105826	0.102956	0.101573	0.098484	0.098111

Table 6.2: The spinning time of suspended tasks vs the task service time variance in the spinning policy

6.3.2.3 Impact of Lock Holding Ratio

As discussed in Section 6.3.2, the lock holding ratio is defined as $\frac{S_l}{S_c + S_l}$. Figure 6.3 shows the performance of the policies as a function of the lock holding ratio ($\rho = 70\%$ and $S_c + S_l = 0.25$). The spinning policy is extremely sensitive to the lock holding ratio because the spinning time increases dramatically with increasing locking holding time. Since processors can execute other ready tasks, the blocking policies are less sensitive to

the lock holding time. Interestingly, the hybrid policy has better performance than the corresponding blocking policy over the range of values considered in the experiment.

Figure 6.3: Impact of lock holding ratio

6.3.2.4 Impact of the Number of Iterations

The previous results in Section 6.3.2 assumed that the maximum number of iterations is 15. This section presents the impact of the variance in the maximum number of iterations per task (the mean value is $\frac{1 + \text{Max}_i}{2}$) on the performance of the policies for $\rho = 70\%$. The task service demand is fixed at 2 time units. The larger the value of the maximum number of iterations is, the smaller is the service demand per iteration (i.e. the system moves toward finer granularity).

Figure 6.4 demonstrates that the maximum number of iterations has a negative impact on the performance of all scheduling policies. This implies that reducing the number of lock accesses should provide a better performance. The same conclusion was arrived at in [Polychronopoulos & Kuck 1987] in the context of scheduling parallel loops. In blocking

policies, the suspended task acquires a lock and is then re-scheduled. There exists a time delay between when a task acquires a lock and when it is re-scheduled that is implicitly included in the lock holding time. Therefore, the blocking policies are more sensitive than the spinning policy to the parameter.

Figure 6.4: Performance of scheduling policies as a function of maximum number of iterations (Max_i)

6.3.2.5 Impact of the Queue Access Time

This section investigates the impact of the queue access time (f) on the performance of the four task scheduling policies. Figure 6.5 presents the performance sensitivity of the scheduling policies to the queue access time for $\rho = 70\%$. As discussed in Section 6.3.2.4, there exists a time delay that is implicitly included in the lock holding time. This time delay increases with increasing queue access time. Therefore, the blocking policies exhibit increased sensitivity to the queue access time.

2

of/de

2

PM-1 3 1/4" x 4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

PIONEERS IN METHYLENE BLUE TESTING SINCE 1974

MicroD

1080 COUNTY ROAD 5, BURLINGAME, NJ 08007 USA
TEL: 612 440 7887 FAX: 612 440 7887 TLE: 8100000000

In addition, suspended tasks in the dynamic blocking policy may be inserted into a queue at a higher level requiring additional queue accesses. Thus, the dynamic policy is more sensitive (than the static policy) to queue access time. Also, tasks in the system with spinning policy only require a single queue access; therefore, the queue access time has marginal effect on the performance of this policy.

Figure 6.5: Performance sensitivity of the scheduling policies to queue access time (f)

6.3.2.6 Sensitivity to the Spinning Time in the Hybrid Policy

The spinning time (T_s) is an additional parameter used in the hybrid policy. When the value is zero, the policy is exactly identical to the blocking policy. And, the hybrid policy behaves like the spinning policy if the value is very large. Figure 6.6 presents the performance sensitivity of the hybrid policy to the spinning time (T_s) for $\rho = 70\%$. Since the spinning policy has better performance than the blocking policy in this job model, increasing the spinning time (T_s) improves the performance of the hybrid policy.

Figure 6.6: Impact of spinning time (T_s) on the performance of the hybrid policy

6.3.3 Summary

This section has provided a performance analysis of four task scheduling policies - a spinning policy, two blocking policies and a hybrid policy. As in the previous studies [Karlin et al. 1991, Zahorjan et al. 1989 & 1991], the spinning policy performs better than the blocking policies except when the lock holding time is large. This is due to the fact that the spinning time is smaller than the overhead of obtaining a ready task.

Also, the dynamic blocking policy (blocking 2) outperforms the static policy (blocking 1) except when the queue access time is large. This is because of its dynamic load-sharing property. Interestingly, the hybrid policy, in which the suspended tasks spin for some time before being blocked, has better performance than the blocking policy including the situation in which the blocking policy performs better than the spinning policy.

6.4 Barrier Synchronization Model

6.4.1 Workload Model

The generic structure of tasks in the barrier synchronization job model is shown in Figure 3.4. Similar to the lock accessing job model, it is assumed that the job arrivals form a Poisson process with parameter λ (i.e. job interarrival times are exponentially distributed). The jobs are partitioned into a set of tasks (T), which is uniformly distributed between 1 and the system size (N). Tasks are characterized by the number of iterations (i), which is also uniformly distributed between 1 and Max_i , the computation phase service demand in each iteration (S_C). Again, tasks within a job are assumed to have identical behaviour, i.e., identical number of iterations and computation phase service demand in the same iteration. In the model, the number of iterations and the mean computation phase service demand (S_C) in each iteration are determined when a job is created. The computation phase service demand of tasks in i^{th} iteration is uniformly distributed between $(1 - V_{S_C})$ and $(1 + V_{S_C})$ of S_{C_i} , which is generated by an exponential distribution. The variance represents the variation in accessing the shared memory during the local computation phase.

The system is assumed to use the combining-tree barrier mechanism discussed in [Mellor-Crummey & Scott 1990] to implement the barrier. Thus, the barrier synchronization time (S_b) is modelled as $Cd_{in} * \lceil \log T \rceil$ where T is the number of tasks in the job. In the spinning policy, the next computation phase of the application starts at S_b time units after the completion of the last task in the current computation phase. Similarly, in blocking policies the suspended tasks are activated after a delay of S_b time units after the completion of the last task in the previous computation phase.

6.4.2 Simulation Results

Table 6.3 summarizes the default parameter values used in the experiments. The task service demand is fixed at 2 time units (excluding the time for executing the barrier mechanism). Section 6.4.2.1 presents impact of the system load. Section 6.4.2.2 shows the performance sensitivity to the variability in service time in computation phase. Section 6.4.2.3 investigates the effect of the task granularity. Section 6.4.2.4 discusses the impact of the queue access time (f) and Section 6.4.2.5 gives the performance sensitivity of the hybrid policy to the spinning time (T_s).

Parameters	Description	Default Values
λ	Arrival rate	0.5
Max_i	Maximum number of iterations	15
Cd_{in}	Communication delay in IN	0.005
Cd_{fe}	Communication delay in FE	0.0005
f	Queue access time	0.02
Tr	Transfer factor	1
B	Branching factor	4
S_C	Mean service demand in computation phase	0.25
V_{Sc}	Variability to mean service demand	0.25
T_s	Spinning time in hybrid policy	0.2

Table 6.3: Parameter values used in the barrier synchronization workload experiments

6.4.2.1 Impact of System Load

In order to avoid the deadlock associated with the spinning policy, the number of tasks within a job moved down to a child queue is limited to the number of processors linked to that queue. Figure 6.7 presents the performance of the spinning policy and the other task scheduling policies as a function of system load (ρ). As can be expected, the

spinning scheduling policy is very sensitive to the system load because the policy may not effectively utilize the resources in the system. For example, assume that three consecutive applications with a size of 60, 49 and 47 are inserted in the root queue. The deadlock restriction with the *FCFS* job scheduling policy forces the applications to be executed in serial order. Therefore, the system becomes a single user system and saturates at a low system load (around 50%). In contrast to the lock accessing model, the blocking policies are less sensitive to the system load.

Figure 6.7: Performance of scheduling policies as a function of system utilization

6.4.2.2 Performance Sensitivity to Service Time Variance

This section investigates the impact of the computation phase service demand variance on performance of the task scheduling policies ($\rho = 50\%$). In the spinning policy, the total task spinning time in an application in each communication phase is $T * \text{Max}(S_{C_{ij}}) - \sum_{i=1}^T S_{C_{ij}}$, where $S_{C_{ij}}$ is the service demand of the i^{th} task of the job in the j^{th} computation phase. When the variability becomes large, the total spinning time dramatically increases. Thus, this

policy is very sensitive to the service time variance. Since processors do not waste any time in spinning, the service time variance (V_{Sc}), on the other hand, has no impact on the performance of the blocking policies.

Figure 6.8: Impact of the service time variance (V_{Sc})

6.4.2.3 Impact of Task Granularity

Figure 6.9 shows the performance of the policies as a function of the maximum number of iterations in a task (Max_i) for system load (ρ) = 50%. The task service demand is fixed at 2 time units. In contrast to the lock accessing workload, the blocking policies are marginally affected by the granularity. However, the job response time of the spinning policy increases dramatically with increasing number of iterations. This is due to the fact that, with increasing number of iterations, the spinning time of processors increases. As stated in Section 6.4.1, each communication phase requires an additional S_b time delay before the beginning of the next computation phase for barrier mechanism, and all but one processor are spinning during this time period.

Figure 6.9: Performance sensitivity to the maximum number of iterations

6.4.2.4 Impact of the Queue Access Time

The queue access contention is an important issue. This section presents the impact of the queue access time (f) on performance of the policies with the barrier synchronization workload. Generally, there is a delay - introduced by the mutually exclusive access to the task queue - between when the first and last tasks of the same job are scheduled. The delay, which depends on the value of the queue access time (f), has a direct impact on performance of the spinning policy. The blocking policies, on the other hand, are not seriously affected by the queue access time because processors can process the other ready tasks during the waiting time.

Figure 6.10: Impact of the queue access time (f)

6.4.2.5 Performance Sensitivity of the Hybrid Policy to the Spinning Time

Figure 6.11 presents the performance of the hybrid policy as a function of the spinning time (T_s) at 50% system load. The results in Section 6.4.2.1 show that the blocking policy has better performance than the spinning policy at this system load ($\rho = 50\%$). Therefore, job response time associated with the hybrid policy increases slightly with increasing the spinning time. The results imply that additional spinning time deteriorates the system performance of the hybrid policy.

Figure 6.11: Performance of hybrid policy as a function of the spinning time (T_s)

6.4.3 Summary

This section has discussed the performance of the task scheduling policies on the barrier synchronization workload. Since processors do nothing but spinning and waiting, the spinning policy performs poorly in this type of job model. On the other hand, the blocking policies are marginally affected by the system and synchronization parameters. Similar to the results in Section 6.3 (for the lock accessing model), the dynamic blocking (blocking 2) policy mostly outperforms the static blocking (blocking 1) policy because of its dynamic load-sharing property. Also, the hybrid policy is affected by the additional spinning time and performs worse than the blocking policy in this workload.

6.5 Conclusion

In this chapter, the performance of two scheduling policies for parallel systems - *spinning* and *blocking* - has been examined. The results show that these two policies perform well in one particular job model but are very sensitive to the system workload and

job characteristics in the other job model. The spinning policy performs better than the blocking policies except when the locking holding ratio is large ($> 5\%$) in the lock accessing job model. The blocking policies, on the other hand, are not affected by the system load and job characteristics in the barrier synchronization job model. The hybrid policy, which is a combination of both the spinning and the blocking policies, performs well in both job models.

CHAPTER VII

CONCLUSIONS

7.1 Summary

As stated in Chapter 1, the goal of this thesis is to design a processor scheduling scheme for large-scale parallel systems (> 100 processors). In order to achieve this goal, a set of performance experiments have been done through the use of simulation models. The results of these experiments are summarized in this section.

In Chapter 4, a detailed performance comparison of two existing organizations and a new organization has been provided. The following observations can be made on the two existing task queue organizations:

- Centralized organization, in which a single central task queue contains all the runnable tasks, has the load-sharing property. This avoids performance deterioration caused by variance in the service time and in the task distribution. Unfortunately, the queue access contention, which is neglected in most of the previous work, limits the system size and the task granularity of the system.
- Distributed organization, in which each processor has its own private queue for ready tasks, is not affected by the access contention. However, this organization lacks load sharing resulting performance deterioration.

Because of the disadvantages of the centralized and distributed organizations, a new organization called the hierarchical organization in which a set of ready task queues are organized in a tree structure, has been proposed. The new organization has been shown to have the following properties:

- **Load-sharing:** A few processors share a leaf queue in the hierarchical structure, and the processors transfer tasks from the parent queue to its child queue if and only if the child queue is empty. This mechanism implicitly contributes to dynamic load-sharing as in the centralized organization.
- **Contention-free access to the ready queues:** Since the number of the processors allowed to simultaneously access a queue is bounded by the parameter B , the queue access contention can seldom exist with the appropriate selection of the parameter value.

The results of the simulation experiments shows that the hierarchical organization outperforms the other two organizations. Besides the performance advantage, the hierarchical organization is more tolerant to processor failures than the distributed organization. In a distributed organization, failure of a single processor may affect the whole system because the tasks placed in the local task queue of the faulty processor are never processed. Processor failures in the hierarchical organization, on the other hand, do not seriously affect the system performance unless the processors linked to a particular queue have failed.

In Chapter 5, the performance of various job scheduling policies and task scheduling policies (with the hierarchical task queue organization using the independent job model) has been studied. The results of this chapter can be summarized as follows:

- **Job-dependence policies** (the smallest number of tasks job first policy, *SNJ*, and the smallest service demand job first policy, *SDJ*) perform better than the simple *FCFS* job scheduling policy.
- Although the smallest service demand job first (*SDJ*) policy requires detailed *a priori* task service time information, the policy offers only negligible improvement in the job response time compared to that of the simple *SNJ* policy.

- Local round robin task scheduling policies (i.e. the processors remove a set of tasks from the ready task queue to its local memory, and process the tasks locally in the round robin fashion) do not offer any performance advantages over the simple *FCFS* task policy. This is due to the trade-off between increasing the number of tasks participating in round robin scheduling and the scope of the load-sharing.

In Chapter 6, the performance sensitivity of task scheduling policies (with the hierarchical task queue organization) to the system and synchronization parameters of two dependent job models - *barrier synchronization* model and *lock accessing* model - has been investigated. Four task scheduling policies were considered: a spinning policy, two blocking policies and a hybrid policy. The results are summarized below.

- Spinning policy performs well on the lock accessing model except when the lock holding time is large ($> 5\%$ of the task service demand).
- Blocking policies perform worse than the spinning policy in the lock accessing model because the overhead cost to obtain a ready task is larger than the lock waiting time.
- Queue access contention results in additional spinning time in the barrier synchronization workload.
- Blocking policy that provides additional load-sharing performs better than the other blocking policy in all cases except when the queue access time is large.
- The variability of task service demand has substantial impact on the performance of the spinning policy in the barrier synchronization workload.
- Hybrid policy that spins for a while and then blocks performs well in both the workloads. From the results shown of this chapter, it may be conjectured that this hybrid policy may provide the best performance if a mixed job model is considered.

7.2 Contributions of the Thesis

This section summarizes the contributions of this thesis to the area of processor scheduling in the shared-memory systems.

Most of the previous studies have concentrated on the performance of the centralized organization and neglected the task queue access contention. Thus, their results are applicable only to small systems (say less than 100 processors). This thesis has identified the problems associated with the centralized and distributed organizations and studied the performance of a hierarchical task queue organization. The hierarchical organization is shown to inherit the good properties of the centralized and distributed organizations. A very detailed performance evaluation of this hierarchical organization has been done. The material presented in Chapters 4, 5 and 6 are the contributions of this thesis. The results presented in Chapters 4 and 5 appear as [Cheng & Dandamudi 1992a, Cheng & Dandamudi 1992b].

7.3 Directions for Further Research

As a result of these performance experiments, a number of suggestions for further research are given in this section.

In Chapter 4, it was observed that the performance of the hierarchical organization can be improved by increasing the transfer factor Tr with the system load (Figure 4.10). Thus, a dynamic scheme in which the number of tasks moved one level down the tree is a function of the system load (the idea used in the dynamic local round robin policies RR2 and RR4 in Section 5.3) may provide performance improvements by adapting to the varying system conditions.

Lock accessing and barrier synchronization are commonly found in parallel applications. These two workloads were considered separately in Chapter 6. In practice, applications may contain both lock access and barrier synchronization. Since the results in

Chapter 6 have shown that the blocking and the spinning policies perform differently in the two jobs models, a study with mixed type of workload is necessary.

The distributed organization with adaptive task routing strategies is found to be competitive to the hierarchical organization under the independent job model. Since the previous work on the distributed organization has not considered the dependent job models, further work is needed that provides a performance comparison between the distributed and the hierarchical organizations with dependent job models.

In this study, tasks are assumed to run on any processor, and that the task execution time is not affected by the processor selection. This assumption ignores some possible second-order effects such as the cache-related benefits of running tasks of an application on the same processors. In most commercial systems, the cache memory architecture actually improves the system performance. Therefore, a more detailed study is needed to understand the effects of these issues.

REFERENCES

[Anderson 1990]

T. Anderson, "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors", *IEEE Trans. on Parallel and Distributed Systems*, Vol. 1, No.1, January 1990, pp. 6-16.

[Appiani et al. 1989]

E. Appiani, B. Conterno, V. Luperini and L. Roncarolo, "EMMA2, A High Performance Hierarchical Multiprocessor", *IEEE Micro*, February 1989, pp. 42-56.

[Black 1990]

D.L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System", *IEEE Computer*, May 1990, pp. 35-43.

[Cheng & Dandamudi 1992a]

S.P. Cheng and S. Dandamudi, "Performance Analysis of a Hierarchical Task Queue Organization for Parallel Systems", *Proceedings of the 12th International Conference on Distributed Computer System*, June 1992 (to appear).

[Cheng & Dandamudi 1992b]

S.P. Cheng and S. Dandamudi, "Scheduling in Parallel Systems with a Hierarchical Organization of Tasks", *Proceedings of International Conference on Supercomputing*, July 1992 (to appear).

[Dandamudi 1990]

S.P. Dandamudi, "The Impact of Ready Queue Access Contention on Performance of Parallel Systems", Research Note, School of Computer Science, Carleton University, Ottawa, 1990.

[Dandamudi 1991]

S.P. Dandamudi, "A Comparison of Task Scheduling Strategies for Multiprocessor Systems", *Proceedings of 3rd IEEE Symp. on Parallel and Distributed Processing*, 1991, Dallas, Texas, pp. 423-426.

[Dandamudi & Eager 1990]

S.P. Dandamudi and D.L. Eager, "Hierarchical Interconnection Networks for Multicomputer Systems", *IEEE Trans. on Computers*, Vol. C-39, No. 6, June 1990, pp. 786-797.

[Feitelson & Rudolph 1990a]

D.G. Feitelson and L. Rudolph, "Distributed Hierarchical Control for Parallel Processing", *IEEE Computer*, May 1990, pp. 65-77.

[Feitelson & Rudolph 1990b]

D.G. Feitelson and L. Rudolph, "Mapping and Scheduling in a Shared Parallel Environment Using Distributed Hierarchical Control", *Proc. of 1990 Int. Conf. on Parallel Processing*, Aug. 1990, Vol. I, pp. 11-18.

[Flynn 1972]

M.J. Flynn, "Some Computer Organizations and Their Effectiveness", *IEEE Trans. on Computers*, Vol. C-21, No.9, Sept. 1972, pp.948-960.

[Gajski et al. 1983]

D. Gajski, D. Kuck, D. Lawire and A. Samch, "Ceder", Department of Computer Science, University of Illinois, Urbana, 1983.

[Ghose & Desai 1989]

K. Ghose and K.R. Desai, "The HCN: A Versatile Interconnection Network Based on Cubes", *Proc. of Supercomputing Conference*, 1989, pp. 426-435.

[George 1990]

Lal George, "A Scheduling Strategy for Shared Memory Multiprocessors", *Proc. of 1990 Int. Conf. on Parallel Processing*, Aug. 1990, Vol. I, pp. 167-171.

[Gupta & Tucker 1989]

A. Gupta and A. Tucker, "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors", *Proc. of 12th ACM Symp. on Operating System Principle*, Dec 1989, pp. 159-166.

[Gupta et al. 1991]

A. Gupta, A. Tucker and S. Urushibara, "The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Application", *Proc. of 1991 ACM Sigmetric Conference on Meas. and Mod. of Computing Systems*, May 1991, pp. 120-132.

[Hoare 1974]

C.A.R. Hoare, "Monitors: An Operating System Structuring Concept", *Comm. ACM*, Vol. 17, No. 10, Oct. 1974, pp. 549-557.

[Hwang & Ghosh 1987]

K. Hwang and J. Ghosh, "Hypernet: A Communication-Efficient Architecture for Constructing Massively Parallel Computers", *IEEE Trans. on Computer*, Vol. C-36, No. 12, December 1987, pp 1450-1466.

[Jiang et al. 1991]

H. Jiang, L.N. Bhuyan and J.K. Muppala, "MVAMIN: Mean Value Analysis Algorithms for Multistage Interconnection Networks", *Journal of Parallel and Distributed Computing*, Vol 12, 1991, pp. 189-201.

[Jones & Murta 1989]

P. Jones and A. Murta, "Practical Experience of Run-Time Link Reconfiguration in a Multi-Transputer Machine", *Concurrency - Practice and Experience*, Vol. 1, No. 2, December 1989, pp. 171-189.

[Karlin et al. 1991]

A.R. Karlin, K. Li, M.S. Manasse, S.Owicki, "Empirical Studies of Competitive Spinning for A Shared-Memory Multiprocessor", *Proc. of ACM Thirteen Symp. on Operating System Principle*, 1991, pp. 41-55.

[Kruskal & Snir 1983]

C.P. Kruskal and M. Snir "The Performance of Multistage Interconnection Networks for Multiprocessors", *IEEE Trans. on Computers*, Vol. C-32, No. 12, Dec. 1983, pp. 1091-1098.

[Leutenegger 1990]

S.T. Leutenegger, *Issues in Multiprogrammed Multiprocessor Scheduling*, PhD Thesis, University of Wisconsin-Madison, Computer Sciences Department, 1990.

[Leutenegger & Vernon 1990]

S.T. Leutenegger and M.K. Vernon, "The Performance of Multiprogrammed Multiprocessor Scheduling Policies", *Proc. of ACM Sigmetrics Conference*, Boulder, Colorado, 1990, pp. 226-236.

[Lovett & Thakkar 1988]

T. Lovett and S. Thakkar, "The Symmetry Multiprocessor System", *Proc. of 1988 Int. Conf. on Parallel Processing*, 1988, Vol. I, pp. 303-310.

[Majumdar et al. 1988]

S. Majumdar, D.L. Eager and R.B. Bunt, "Scheduling in Multiprogrammed Parallel Systems", *Proc. of ACM Sigmetrics Conf.*, Santa Fe, NM, 1988, pp. 104-113.

[Marson et al. 1986]

M.A. Marson, G. Balbo and G. Conte, *Performance Models of Multiprocessor Systems*, MIT Press, 1986.

[Mellor-Crummey & Scott 1990]

J.M. Mellor-Crummey and M.L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", University of Rochester, Computer Science, Technical Report 342, April 1990.

[Nanda et al. 1991]

A.K. Nanda, H. Shing, T.-H. Tzen and L.M. Ni, "Resource Contention in Shared-Memory Multiprocessors: A Parameterized Performance Degradation Model", *Journal of Parallel and Distributed Computing*, Vol. 12, Aug. 1991, pp.313-328.

[Nelson et al. 1988]

R. Nelson, D. Towsley and A.N. Tantawi, "Performance Analysis of Parallel Processing Systems", *IEEE Trans. on Soft. Eng.*, Vol SE-14, No. 4, April 1988, pp532-540.

[Ni & Wu 1989]

L.M. Ni and C.E. Wu, "Design Tradeoffs for Process Scheduling in Shared Memory Multiprocessor Systems", *IEEE Trans. on Soft. Eng.*, Vol. SE-15, No. 3, March 1989, pp. 327-334.

[Ousterhout 1982]

J.K. Ousterhout, "Scheduling techniques for concurrent systems", *Proc. of 3rd Int. Conf. on Distributed Computing Systems*, Oct. 1982, pp. 22-30.

[Pfister & Norton 1985]

G.F. Pfister and V.A. Norton, "'Hot Spot' Contention and combining in Multistage Interconnection Networks", *IEEE Trans. on Computers*, Vol. C-34, No. 10, Oct. 1985, pp. 943-948.

[Polychronopoulos & Kuck 1987]

C.D. Polychronopoulos and D.J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers", *IEEE Trans. on Computers*, Vol. C-36, No. 12, Dec. 1987, pp. 1425-1439.

[Sevcik 1989]

K.C. Sevcik, "Characterization of Parallelism in Applications and Their Use in Scheduling", *Proc. of ACM Sigmetrics Conf.*, Berkeley, 1989, pp. 171-180.

[Stenstrom 1988]

P. Stenstrom, "Reducing Contention in Shared-Memory Multiprocessor", *IEEE Computer*, Nov. 1988, pp. 26-37.

[Swan et al. 1977]

R.J. Swan, S.H. Filler and D. Siewiorek, "C.m* -- A Modular, Multi-Microprocessor", *Proc. of the National Computer Conference*, 1977, pp. 39-46.

[Thacker et al. 1988]

C.P. Thacker, L.C. Stewart and E.H. Satterthwaite, Jr., "Firefly: A Multiprocessor Workstation", *IEEE Trans. on Computers*, Vol. C-37, No. 8, Aug. 1988, pp. 909-920.

[Towsley et al. 1990]

D. Towsley, C.G. Rommel and J.A. Stankovic, "Analysis of Fork-Join Program Response Times on Multiprocessors", *IEEE Trans. on Parallel and Distributed Systems*, Vol. 1, No. 3, July 1990, pp. 286-303.

[Tucker & Robertson 1988]

L.W. Tucker and G.G. Robertson, "Architecture and Applications of the Connection Machine", *IEEE Computers*, Vol. 21, No. 8, August 1988, pp. 26-37.

[Wu & Liu 1981]

S.W. Wu and M.T. Liu, "A cluster structure as an Interconnection Network for Large Multicomputer System", *IEEE Trans. on Computers*, Vol. C-36, No.4, April 1987, pp. 388-395.

[Yew et al. 1987]

P.-C. Yew, N.-F. Tzeng and D.H. Lawrie, "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors", *IEEE Trans. on Computers*, Vol. C-36, No.4 April 1987, pp. 388-395.

[Zahorjan & McCann 1990]

J. Zahorjan and C. McCann, "Processor Scheduling in Shared Memory Multiprocessors", *Proc. of ACM Sigmetric Conference*, Boulder, Colorado, 1990, pp. 214-225.

[Zahorjan et al. 1988]

J. Zahorjan, E.D. Lazowska and D. L. Eager, "Spinning versus Blocking in Parallel Systems with Uncertainty", *Proc. of Int. Symp. on Performance of Distributed and Parallel System*, Kyoto Japan, Dec. 1988, pp.445-462.

[Zahorjan et al. 1991]

J. Zahorjan, E.D. Lazowska and D. L. Eager, "The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems", *IEEE Trans. on Parallel and Distributed Systems*, Vol. 2, No. 2, April 1991.

END

2|3|1 2|92|

FIN