1995

Space sharing job scheduling policies for parallel computers

Ismail Mohamed Ismail
Iowa State University

Space sharing job scheduling policies for parallel computers

by

Ismail Mohamed Ismail

A Dissertation Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Department: Electrical and Computer Engineering
Major: Computer Engineering

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University

Ames, Iowa

1995

TABLE OF CONTENTS

ABBREVIATIONS

DEQP: dynamic equipartitioning

DFCFS: dynamic first come first served

DPROP: dynamic proportional

DPROP-SH: dynamic proportional/shorter jobs favored

DPROP-SM: dynamic proportional/smaller jobs favored

DSMJF: dynamic smallest job first

EPFP: even partitioning of free processors

FCFS: first come first served

FCFSUF: first come first served/unlimited folding

FF: first fit

FF+FIFO: first fit then first in first out

FFCFS: folding first come first served

FFDS: first fit decreasing size

FFDS+FIFO: first fit decreasing size then first in first out

FFF: folding first fit

FFIS: first fit increasing size

FFIS+FIFO: first fit increasing size then first in first out

FFITD: first fit increasing total demand

FIFO: first in-first out

FSJF: folding smallest job first

LOJFUF: longest job first/unlimited folding

MFFF: multifolding first fit

MFLOJF: multifolding longest job first

MFSJF: multifolding smallest job first

MFSHJF: multifolding shortest job first

MFSTDF: multifolding smallest total demand first

MISP: monotonically increasing speedup

PSCDF: preemptive smallest cumulative demand first

PSNPF: preemptive smallest number of processes first

RRjob: round-robin job

RRprocess: round-robin process

SCDF: smallest cumulative demand first

SHJFUF: shortest job first/unlimited folding

SNPF: smallest number of processes first

STDFUF: smallest total demand first/unlimited folding

ACKNOWLEDGMENTS

I warmly thank my advisor Jim Davis for his help, encouragement, cooperation, and kindness. His help was always forthcoming when needed.

I would also like to thank the other members of my committee, Albert Baker, John Gustafson, Doug Jacobson, and Charles Wright for their help and feedback.

I thank my mother and father. My father has always wanted me to seek the best. His love and trust in me have been unwavering. Although my mother does not know how to read or write, she has taught me that seeking knowledge requires serious effort and care. She is an example of honesty, generosity, dignity, and courage.

My special thanks to my wife. She has been supportive and patient. The nice environment she created at home made my Ph.D. program much easier.

Thanks to the Department of Electrical and Computer Engineering for financially supporting me through most of my program, and for providing a wonderful learning environment. The faculty has been helpful and supportive and the staff professional and kind. Chip Comstock has helped me learn how to teach and has been a wonderful friend. Dick Horton provided me with the opportunity to teach courses in addition to supervising lab sessions. I thank them. Special thanks to Prasant Mohapatra for his help and encouragement. Thanks to Patsy Wisecup, Linda Clifford, Shirley Calhoun, and Gloria Wierda for their graceful administrative help.

ABSTRACT

The distinguishing characteristic of space sharing parallel job scheduling policies is that applications are allocated non-overlapping processor subsets. The interference among jobs is reduced, the synchronization delays and message latencies can be predictable, and distinct processors may be allocated to cooperating processes so as to avoid the overhead of context switches associated with traditional time-multiplexing.

The processor allocation strategy, the job selection criteria, and workload characteristics are fundamental factors that influence system performance under space sharing. Allocation can be static or dynamic. The processor subset allocated to an application is fixed under static space sharing, whereas it can change during execution under dynamic space sharing. Static allocation can produce more predictable run times, permits a wide range of compiler optimizations (e.g., static data distribution and binding), and avoids the processor releases and reallocations associated with dynamic allocation. Its major problem is that it can induce high processor fragmentation.

In this dissertation, alternative static and dynamic space sharing policies that differ in the allocation discipline and the job selection criteria are studied and compared. The results show that significantly superior performance can be achieved under static space sharing if applications can be folded (i.e., allocated fewer processors than they requested). Folding typically increases program efficiency and can reduce processor fragmentation. Policies that increase folding with the system load are proposed and compared to schemes that use unconstrained folding, no folding, and fixed maximum folding factors. The adaptive policies produced higher and more stable system utilization, significantly shorter mean response times, and good fairness curves. However, unconstrained folding resulted in considerably more severe processor fragmentation than no folding. Its advantage is that it exploits the efficiency improvement that typically results when an application is allocated fewer processors. Conse-

quently, it can produce shorter mean response times than no folding under medium to heavy loads.

Also because of this efficiency improvement, dynamic policies that reduce waiting times by executing a large number of jobs simultaneously are more promising than schemes that limit the number of active jobs. However, limiting the number of active applications can be the superior approach when folding does not improve application efficiency.

1 INTRODUCTION

1.1 Extended Abstract

Alternative topology-independent space sharing policies for scheduling parallel applications on multiprogrammed homogeneous multiprocessors are studied and compared in this dissertation. The distinguishing characteristic of space sharing is that programs are allocated *distinct* subsets of processors. The interference among jobs is reduced, the synchronization delays and message latencies can be more predictable, and cooperating processes may be allocated to distinct processors so as to avoid the overhead of context switches associated with traditional time-multiplexing.

Distributed-memory multiprocessors have commonly been space shared [Seitz 90], and space sharing has been proposed as the processor allocation strategy in *two-level schedulers* [Tucker 89]. In these schedulers, the operating system kernel allocates processors to applications and user-level library routines schedule application threads on the allocated processors. To reduce their cost, load balancing and latency hiding are carried out at the user-level rather than by the kernel. Two-level schedulers have been proposed for shared and distributed-memory machines [Tucker 89][Beckerle 92].

In a multiprogrammed parallel computing system, where several jobs may compete for processing elements, the functions of the job scheduling algorithm are job selection and processor allocation. For example, the algorithm may give priority to the job that arrived first and implement the space sharing processor allocation strategy.

Space sharing can be static or dynamic. Under *static space sharing*, a program is allocated a fixed subset of processors. The subset's size can be determined in two basic ways: (1) it is fixed a priori (e.g., by the compiler or user), and the algorithm allocates that many processors; or (2) applications can be *folded*; that is, an application can be allocated fewer processors than it has requested. It is assumed that a program, upon arrival, requests a number of processors from the allocation algorithm.

Under *dynamic space sharing*, the processor subsets allocated to applications can change during execution. Coordination between the allocation algorithm and the runtime system is needed. The algorithm decides the changes, and the runtime system reconfigures the applications for execution on the new subsets.

There are several advantages to space sharing over traditional process-based scheduling. First, the context switches associated with time-multiplexing can be avoided. To achieve this, the loader or the runtime system, in coordination with the job scheduling algorithm, can adjust the job's process parallelism so that it is equal to the number of processors allocated and assign the processes to different processors. Second, when the processes comprising an application execute on distinct processors, co-scheduling (i.e., the simultaneous dispatching of cooperating processes) is guaranteed. Coscheduling is essential when the processes interact extensively [Ousterhout 82]. Excessive synchronization delays can result if some cooperating processes are not running.

Application folding and *processor fragmentation* are two factors that strongly influence performance under space sharing. Folding typically results in higher program efficiency. When an application executes on fewer processors, the inefficiency caused by the serial fraction (Amdahl's law) and similar load imbalances is reduced and less interprocess communication and synchronization is typically needed.

An algorithm produces processor fragmentation when it prevents processors from being allocated. Under space sharing, there can be *internal*, *external*, and *folding* fragmentation. There is internal fragmentation when the number of processors allocated to a job exceeds the number it requested. For example, the allocation algorithm in a mesh distributed-memory system may allocate a submesh to a job whose processor demand is less than the submesh's size. Topology-based allocation is often used in distributed-memory machines to reduce communication overhead and interconnection contention, but it can produce high fragmentation [Li 91][Liu 94]. A study of topology-dependent allocation is outside the scope of this research.

There is external fragmentation when free processors are not allocated to waiting applications. It exists, for example, when the number of free processors is less than the sizes of the waiting applications and folding is not supported. Folding may reduce the average processor fragmentation in static space sharing. If jobs can be folded arbitrarily, for example, a free processor need not remain idle if there is a waiting request. However, folding can produce a different type of fragmentation, folding fragmentation, which exists when there are idle processors and one or more folded jobs. It is defined as $(P_{da}-P_a)/P$, where P_{da} is the total processor demand of the running applications, P_a the number of allocated processors, and P the number of processors in the target computer.

Topology-independent dynamic space sharing is free from processor fragmentation. Folding fragmentation in static space sharing may be reduced if folding is *limited*; that is if applications wait until they can receive some fraction, not necessarily fixed, of the number of processors they have requested. Folding is *unlimited* or *unconstrained* if applications can be folded to any degree (i.e., they are allowed to execute on a single processor independently of their processor requests).

The role of the allocation algorithm is fundamental. The efficiency of the computing system depends on the *scheduling effectiveness* and the efficiency of programs. The scheduling effectiveness, S_e, is used to measure the ability of a static space sharing algorithm to avoid processor fragmentation. It is defined by the equation $S_e=P_a/\min(P,P_d)$, where P_d is the current total processor demand. Topology-independent dynamic space sharing can achieve $S_e=1$ because it is free from processor fragmentation.

The *system efficiency*, ξ_S, is defined by the equation:

$$\xi_s = \frac{\sum_j m_j * \xi_j(m_j)}{\sum_j m_j} * S_e$$

where $\xi_j(m_j)$ is the efficiency of program j when it is allocated m_j processors.

Folding has a strong influence on system efficiency as it can increase program efficiency and scheduling effectiveness. Both must be high for ξ_S to be high. Because it is difficult to write efficient

highly parallel programs, high system efficiency may be easier to obtain with multiprogramming and moderate application parallelism.

No space sharing approach is obviously the best. In static space sharing, runtime application reconfigurations are avoided and a broad range of compiler optimizations (including static data distribution) can be applied. However, there is processor fragmentation. Folding may reduce the average fragmentation and increase program efficiency, but it can increase cache misses and induce or increase swapping. The dynamic approach can achieve superior system utilization (it is free from processor fragmentation when it is topology-independent), but it induces several sources of overhead associated with application reconfigurations. Assuming a distributed-memory machine, for example, these include context switches, cache reloads, and data migration. They may also include code migration, and swapping as a result of folding.

The goal of this research was to study and compare the static and dynamic space sharing strategies when allocation is topology-independent and the execution times of jobs are not known a priori. Several algorithms that implement the two strategies are compared in this dissertation. The algorithms differ in the job selection criteria and the folding method, and the objective is achieving short average turnaround times and high system utilization.

The results show that static space sharing should support folding, and the maximum factor by which applications are folded should be limited and increase with the system load. Adaptive limited folding is substantially superior to no folding and unlimited folding. It can result in significantly lower processor fragmentation and mean response times. Traditional no folding policies (e.g., first-fit and first-come-first-served) suffer from high processor fragmentation. Unlimited folding variants of these policies produced much worse fragmentation under most system loads, but they can yield significantly shorter mean response times under high loads because of the significant efficiency improvement that typically results from folding.

As expected, dynamic space sharing is superior to static space sharing when the overhead of

application reconfigurations is low. However, static space sharing can, depending on the frequency and cost of application reconfigurations, be the better strategy. Because of the efficiency advantage of folding, dynamic schemes that reduce waiting times by folding a larger number of applications are more promising than others that reduce execution times by allowing only a small number of jobs to execute simultaneously.

1.2 Background

1.2.1 Classes of MIMD Machines

System architecture, especially the organization of the memory subsystem, has a strong influence on the design of scheduling algorithms for MIMD multiprocessors. Parallel architectures are commonly classified according to this subsystem's organization. A uniform memory access (UMA) multiprocessor has a common main memory whose cost of access is independent of the address of the requesting processor. Encore's Multimax and Sequent's Symmetry are examples of UMA systems.

In non-uniform memory access (NUMA) shared-memory systems, main memory is shared but hierarchical. Part of it is local (i.e., significantly less costly to access) to each processing element (PE), where a PE may consist of a single processor or a small bus-based cluster of processors. Examples of clustered NUMA systems are the Stanford Dash [Lenoski 92] and the Illinois Cedar [Eigenmann 91].

In distributed-memory multiprocessors, also called multicomputers, main memory is not shared, and processes executing on different processors must exchange messages in order to communicate and synchronize their activities. Examples of multicomputers are Intel's iPSC/860 and the NCUBEs.

In UMAs, processes are typically dispatched independently of the location of their code and data, and of where they executed previously. However, scheduling processes where they last executed can reduce the cache reload overhead and improve program performance [Squillante 93][Torrellas 93].

In NUMAs, executing processes or threads "close", in the memory hierarchy, to the shared objects they reference can significantly reduce the cost of memory operations and execution times [Chandra 93][Matkatos 92a][Markatos 92b]. As the speed of processors has been increasing faster than that of the memory subsystem, exploiting memory locality in recent bus-based systems, such as the Silicon Graphics Iris, can reduce memory access contention and also yield significant performance benefits [Markatos 92a][Markatos 92b].

In distributed-memory systems, applications are typically statically mapped and scheduled because the cost of data and code migration, and the overhead of managing the access to migrated data (e.g., access forwarding) are high. The mapping is often topology-based (i.e., processes that interact extensively are mapped onto adjacent processing nodes) so as to reduce message delays and contention. The fundamental problem with static topology-based allocation is that it can produce high processor fragmentation [Li 91][Liu 94]. Thus, there is a tradeoff between system utilization and exploiting locality.

With recent interconnection routing techniques (e.g., wormhole routing), message delays due to the number of hops between the communicating nodes are significantly reduced, and non-contiguous allocation schemes are receiving increasing interest [Naik 93b][Liu 94]. Experiments on a 208-processor Paragon, a distributed-memory machine that uses wormhole routing, indicate that the contention overhead in non-contiguous allocation may not be so severe so as to offset the benefits of reduced fragmentation [Liu 94]. An earlier parallel programming environment for hypercubes, the Cosmic Environment [Seitz 90], also supported non-contiguous allocation. Applications could request an arbitrary number of processing nodes.

1.2.2 Granularity of Process Interactions

The granularity of parallelism describes the amount of work a process accomplishes between consecutive interactions; that is, between consecutive communication or synchronization events. It is

fine if the amount of work is "small", *coarse* if it is "large". The granularity is described more precisely if the ratio R/C is used, where R is the computation time between two consecutive interactions, and C is the cost of the interaction [Stone 90]. It is fine if R/C is small, coarse if it is large.

A major goal of computer architecture and parallel programming language development has been the design of large parallel systems that can efficiently support fine-grained process interactions. For example, this is a goal of the Stanford Dash shared-memory machine [Lenoski 92], the *T distributed-memory system from Motorola and MIT [Beckerle 92], and wormhole routing [Seitz 90].

Fundamental properties of scheduling algorithms depend on parallelism granularity. A basic issue is whether scheduling cooperating processes independently, as in traditional process-based scheduling, is appropriate when parallelism is fine-grained. For example, the *T multithreaded distributed-memory system has machine instructions that support microthreading and split-phase transactions. Programs use them to hide the latency of requests for remote objects. When a computation needs remote data, one of its threads initiates the necessary network messages and terminates. The processor is then switched to a ready microthread. A data request message contains the address of the computation's continuation thread and is addressed to a program thread in the remote process that holds the requested item. The suspended thread becomes ready and joins the ready microthreads when the remote data is received [Beckerle 92].

Processes that make frequent use of split-phase transactions to synchronize their activities or communicate should be *active simultaneously* to achieve desired performance. Static space sharing that assigns cooperating processes to different processors, and (round-robin) coscheduling [Ousterhout 82] are appropriate candidates for this system. In coscheduling, cooperating processes are assigned to different processors, and they are dispatched together in a round-robin fashion. An advantage of static space sharing is that it avoids the context switches associated with time-multiplexing. Although dynamic space sharing algorithms can achieve the optimal scheduling effectiveness (i.e., S_e=1), it may not be a better solution because the cost of migration and access forwarding may be excessive.

Traditional process-based scheduling can lead to high synchronization delays and poor program turnaround times in UMA [Tucker 89] and NUMA systems [Eigenmann 91]. Tucker and Gupta [Tucker 89] determined experimentally on a 16-processor Encore Multimax, a UMA shared-memory machine, that performance can degrade when the number of processes exceeds the number of processors and regular preemptive priority-based scheduling is used in conjunction with busy-waiting synchronization. The severity of the degradation increased with the number of processes. In their experiments, several programs were used, the sizes of the programs were fixed, but the number of processes they spawned varied. Context switches are a major cause of performance degradation. For example, a process may be preempted in a critical section, blocking the entry of other processes to the section until the process is resumed. The higher the load, the longer a preempted process spends in the waiting queue and the more likely it is that cooperating processes are prevented from entering the critical section. In their solution, *process control*, the number of processes is controlled so that it does not exceed the number of processors, and the processes are assigned to different processors. In this way, preemptions are avoided. In addition to avoiding the preemptions associated with time-multiplexing, process control takes advantage of the increase in efficiency that typically results from folding.

When processes interact frequently, poor performance can result even if the kernel, upon request from the user, does not block a process in its critical section. When a process is preempted at the end of its time-slice, cooperating processes cannot, once they reach the next interaction point, make progress until the preempted process is resumed. If they block, they may cause yet other processes to block, etc.. This can lead to an excessive number of context switches.

However, multiprogramming at the processor level has been widely used in distributed-memory systems designed for coarse-grained parallelism. It provides a mechanism for hiding the high message latency in these systems. For example, the Cosmic Kernel node operating system supported a blocking receive system call. A process that issued this call was suspended if there were no messages for it, and a process that had one was dispatched. Time-slicing was used to enforce fairness by preventing proc-

esses from running uninterrupted for too long [Seitz 90].

This approach is inefficient when parallelism is fine-grained. A context switch entails a process state save and restore, and a system call typically entails a trap, saving the contents of machine registers, and restoring them upon exit from the call. Moreover, system calls are normally general in that they provide the sum of services required by users and typically include code that protects the kernel from user errors [Anderson 92]. For these reasons, two-level schedulers are a promising scheduling approach, and distributed-memory systems designed for finer-grained parallelism (e.g., *T) support microthreading and user access to the network interface. In two-level schedulers, user-level library routines are responsible for thread scheduling so as to reduce its cost. The high-level scheduler is part of the kernel, and is responsible for job selection and processor allocation.

1.2.3 Programming Languages and Models

Scheduling algorithms and programming models are not independent. For example, several parallel programming systems (languages and libraries) include statements for explicitly mapping processes onto specific processors. With these statements, a space sharing allocation algorithm may not run explicitly-mapped processes until the requested processors are free. This can increase or induce processor fragmentation. For example, topology-independent dynamic space sharing is not free from processor fragmentation in this case.

When the number of processors allocated to an application is less than its process parallelism, two scheduling techniques may be used. In the first, the number of processes is decreased so that it is equal to the new partition size and a processor is assigned a single process. A threads package that uses the shared task queue parallel execution model has been used for implementing this approach in shared-memory systems [Tucker 89][McCann 93]. Tasks are added to the queue when they are created and fetched for execution when processors need work. Parallelism is controlled when the task queue is accessed. The task queue model suffers from several sources of overhead, including the critical sections

that control access to the queue(s), and the extra instructions needed to read task descriptors.

In small systems, a single queue may be appropriate. However, multiple queues are needed in large systems to reduce access contention, especially when the tasks are fine-grained. Significant performance improvement can result if tasks execute close to the data objects they reference. These include cached objects, and objects residing in local memory in NUMA machines. To improve cache hit rates, the default thread scheduling policy in FastThreads uses a local threads list that is serviced in last-in-first-out order. A processor scans other lists when its list is empty [Anderson 92]. An issue is how to partition the work for load balancing and data locality. The tradeoff between these two factors is investigated in [Squillante 91][Markatos 92a][Markatos 92b][Markatos 93].

In the second scheduling technique, the number of processes is not changed, but their execution is interleaved. The sources of overhead include load imbalance, which can be much higher than when parallelism is controlled and the task queue model is used, and the process context switches associated with time-multiplexing.

In static space sharing policies that support folding, processor allocation is determined at load time. The compiler must generate object code that can be bound to any number of processors. MIT's Id compiler for the Monsoon, for example, generates code that can run on any number of processors [Hicks 93].

1.2.4 Relationship between Allocation and Efficiency

Speedup and efficiency are commonly used to measure the performance of a parallel application when it executes on a dedicated set of processors. When the application is allocated m processors, its efficiency, $\xi(m)$, and speedup, $Speedup(m)$, are defined as:

$$\xi(m) = \frac{t(1)}{m * t(m)}$$

and

$$Speedup(m) = \frac{t(1)}{t(m)}$$

where $t(j)$ is the execution time of the application on j processors. Speedup is linear if $t(1)/t(m)=m$, superlinear if $t(1)/t(m)>m$, and sublinear otherwise.

The efficiency of a parallel program typically increases when the number of processors it is (exclusively) allocated decreases. The allocation decrease reduces the effect of the serial fraction (Amdahl's law) and other load imbalances due to lack of parallelism, and it typically reduces communication and synchronization. Allocating more processors to a program can improve cache locality and reduce or avoid swapping. However, it is assumed that efficiency does not increase with m in this research because applications seldom have superlinear speedup.

An important issue is determining the number of processors an application should use, n. As m increases, the efficiency and the incremental reduction in execution time typically decrease. Often, the speedup curve is convex in m; that is, the speedup decreases if m increases beyond some value, M. Many applications and algorithms have this type of speedup curves. Examples abound in the literature. Obviously, n should not exceed M as additional processors increase the execution time.

Gustafson suggests that the size of the problem be increased with the number of processors [Gustafson 88]. Given m processors, increasing the size of the problem can increase $\xi(m)$. However, this method is not always useful as applications can have fixed sizes. Assuming that the execution times of the application as a function of m are known, several methods for determining the value of n have been proposed [Flatt 89][Ghosal 91][Gupta 93]. For example, Ghosal, *et al.*, [Ghosal 91] propose that n be the smallest value of m that maximizes the cost function $Speedup(m)*\xi(m)$ (choosing n that optimizes this function had been proposed by Kuck in 1976 [Flatt 89]). The value of $t(n)$ determined by these methods can typically be significantly reduced by allocating more processors. When the system load is moderate, a higher limit on the value of n can improve turnaround times. Methods based on the minimum, average, maximum, and variance of parallelism have also been proposed [Sevcik 89]. These methods are discussed in more detail later.

1.3 Space Sharing Scheduling Policies

Because of the presence of multiple processors in a parallel computer, space sharing and time sharing can be used. In space sharing, a job is allocated a distinct subset of processors; that is, no processor is concurrently assigned to more than one job. Space sharing may be static or dynamic. In static space sharing, the subset (partition) is fixed for the lifetime of the job. However, it can change in size and in the processors it contains in dynamic space sharing.

Process scheduling within program partitions is not modeled in this research. Several techniques, discussed earlier, may be used. In process control, for example, the number of processes is controlled so that is equal to the partition size, a processor is dedicated to each process, and load balancing and thread scheduling are carried out by the programming language runtime system in coordination with the operating system kernel. Traditional process-based scheduling may also be employed. The operating system kernel interleaves the execution of processes when their number exceeds the number of processors they are allocated.

1.3.1 Static Space Sharing

Two basic techniques have been used for implementing static space sharing in parallel systems. Under the *machine partitioning* technique, the system is subdivided into disjoint partitions independently of individual applications, and an application is commonly allocated a *single* distinct partition (when it is not stated otherwise, this one-to-one mapping is assumed). The partitioning may be fixed or adaptive. In *fixed partitioning*, the number and sizes of the partitions are constant, whereas they can vary in *adaptive partitioning*. An issue with preventing applications from running on more than one partition is that it can lead to poor system utilization under moderate loads, when idle partitions are unlikely to be allocated soon.

In the *program-based partitioning* technique, partitions are created for individual applications. When an application is selected for service, the allocation algorithm determines which free processors

to assign to it. The main advantage of this technique is that it can achieve superior system utilization.

1.3.1.1 Fixed Partitioning

A subclass of fixed partitioning policies, based on equipartitioning, was the subject of several recent studies [Ghosal 91][Naik 93b][Setia 93]. The machine is subdivided into partitions of equal size, an application is allocated a single partition, and several partition sizes were considered in these studies. Using the throughput to mean response time ratio as performance parameter, the results in [Ghosal 91] show that: (1) the performance of fixed equipartitioning depends on the size of the partitions, the processor demands of the applications, and the system load, (2) the best partition size generally decreases with the load, and (3) a policy based on the first-fit allocation discipline, FF+FIFO, is superior (under most load levels) to fixed equipartitioning, including when the best partition size (i.e., that which produced the best performance) considered in the study is used. In [Naik 93b][Setia 93], the mean response times decreased when the number of partitions increased with the system load.

No algorithm that determines the best number of partitions is given or used in the three studies, and the influence of job sizes on the performance of fixed equipartitioning was not adequately investigated. For example, the applications can use all processors in [Setia 93], and a small set of applications (five applications) was used in [Ghosal 91]. Another issue is how to dynamically change the number of partitions in this *static* allocation approach, while maintaining equipartitioning.

There are three general issues with equipartitioning. First, it suffers from internal fragmentation, which results when the maximum process parallelism of an active job is smaller than the size of the partitions. Second, only a subset of partition sizes can be used because the number of partitions must divide P. The third issue is that other partitioning strategies may be better when the distribution of the processor requests of applications is general.

1.3.1.2 Adaptive Partitioning

The partitions vary in their sizes and number during machine operation in this processor allocation strategy. The current workload characteristics and system load, for example, may be used in determining these parameters. That the performance of the fixed equipartitioning policy improves when the number of partitions increases with the load, as shown in [Ghosal 91][Naik 93b][Setia 93], implies that an adaptive equipartitioning policy in which the number of partitions is appropriately determined by the system load should be superior to fixed equipartitioning.

1.3.1.3 Program-based Partitioning

Several program-based partitioning policies that differ in the folding method have been studied. The FF+FIFO allocation policy, proposed by Ghosal, *et al.*, [Ghosal 91] for shared-memory machines, uses a FIFO waiting queue and has two phases. In the first phase, the first-fit (FF) algorithm is run and the selected applications are allocated as many processors as they request. If there remains free processors, they are allocated to the head of the queue in the second phase. This policy outperformed several schemes, including FF and equipartitioning that uses the best number of partitions, under most system loads.

Folding is unlimited in FF+FIFO. An application may run on any number of processors that does not exceed its processor request. There are two problems with unlimited folding in static space sharing. First, when an application is allocated a very small number of processors, its execution time typically increases considerably. The application may complete much sooner if it waits for more processors to become available. Second, released processors are likely to remain idle for a long time, especially under moderate system loads, for two reasons: (1) the average length of the waiting queue is smaller than when folding is constrained or not supported, and (2) processors are not allocated to folded jobs in static space sharing. A job may be executing on fewer processors than it requested while processors are idle. The problems with unlimited folding are discussed in detail in the next Chapter.

Abraham and Padmanabhan [Abraham 92] studied a program-based partitioning policy that limits folding. An application is not serviced until it can receive at least $\lceil n/f_{max} \rceil$ processors, where n is the application's processor request and f_{max} a constant. When a program is serviced, it receives minimum(FP,n) processors, where FP is the number of free processors. An issue is determining the value of f_{max}. Using simulation, they determined that it should be 3 or 4. With these values, the limited folding policy outperformed first-fit and first-come-first-served (FCFS) significantly. One limitation of this study is that these f_{max} values were determined through experimentation under limited load and workload characteristics, and they may not be appropriate across a wider range of system loads and workload characteristics.

Naik, *et al.*, [Naik 93b] studied another policy that supports limited folding and compared it to fixed equipartitioning. In this policy, a request is not serviced until it can receive at least some fixed number, *min*, of processors. When a job completes, the free processors are divided evenly among the waiting programs under the above constraint. The policy produced better average turnaround times than fixed equipartitioning, including when the best number of partitions considered in the study is used. An issue is determining the value of *min*. For example, folding can be excessive if *min* is small, and many jobs are not folded if it is high. A second issue is whether the equal division of free processors among waiting jobs is a good policy. In their simulation experiments, the target machine is a 256-processor distributed-memory multiprocessor, and *min* is 32. The choice of this value is not explained. It presumably produced the best performance.

1.3.2 Multiprogrammed Static Partitioning

In this approach, allocation is static and the partitions are multiprogrammed. More than one job may be simultaneously assigned to the same partition. Multiprogramming at the processor level is commonly used to overlap communication and computation in distributed-memory systems that have high message latencies.

In a study by Setia, *et al.*, [Setia 93], the partitions are equal in size, and a job is assigned to a *single* partition. In the program model used, parallelism is coarse-grained, the program consists of one or more phases, and the processes synchronize at the end of each phase. Using simulation, and assuming that the processor requests of the jobs are equal to the machine size, they compared multiprogrammed and uniprogrammed equipartitioning using several partition sizes. Their results show that multiprogrammed equipartitioning can outperform uniprogrammed equipartitioning under all load levels when the load imbalance within programs (the variance of the execution times of processes) is high. However, when the imbalance is small, uniprogramming can be better under low to moderate system loads (< 0.6 in their study). Their results also show that the best number of partitions increases with the load in uniprogrammed and multiprogrammed equipartitioning.

There are several issues with this study. First, the variance of the execution times of processes is assumed to increase linearly when a job is folded. In practice, it may decrease or its increase may be sublinear. Second, the performance of multiprogramming is highly sensitive to the granularity of computation and the value of the time-slice. Third, it is not clear how equipartitioning, uniprogrammed or multiprogrammed, will perform when the processor requests of applications have a general distribution.

Multiprogrammed fixed equipartitioning was proposed by Ahmad, *et al.*, [Ahmad 94] for scheduling applications with dynamic structures on hypercube multicomputers. The hypercube is divided into spheres (partitions with a locality property and a median processor) of equal size. A host computer assigns an application, for its lifetime, to the sphere with the smallest number of main tasks. When a subtask is created, the median assigns it to the least loaded processor in its sphere. The processor load is the number of subtasks it is assigned. When compared with a neighborhood averaging distributed scheduling algorithm, this hierarchical two-level allocation scheme produced better mean response times. In the simulation experiments, a 256-node machine was subdivided into a fixed number (16) of spheres. A problem with this strategy is that it can lead to poor system utilization because applications can not use more than one sphere. Spheres are likely to idle for a long time under moderate

loads.

In the pool-based scheduling technique, proposed and studied by Zhou and Brecht [Zhou 91] for clustered NUMA systems, the machine is subdivided into partitions of equal size, the partitions are multiprogrammed, and a job may span *several* partitions. A job that runs on a given number of processors is assumed to suffer more overhead when it spans more partitions. Their simulation results show that this technique can result in significant reductions in mean response times, and spanning should be restricted or disallowed. However, limited spanning produced better results than no spanning. Although the best number of partitions decreased when the average job parallelism increased, two partitions produced good performance.

Using queuing theory to study job scheduling in distributed-memory machines, Setia, *et al.*, [Setia 94] show that applications should run on fewer processors when the load increases. In the workload model used, the applications have the same maximum process parallelism, N, and a job's processes synchronize once prior to terminating. The allocation policy considered is parameterized in an integer Z that divides N. A new job is split into Z components of size N/Z that are assigned to the Z processors that have the shortest local scheduler queues. The processing nodes scheduler supports multiprogramming and services the processes in its local queue using the FCFS discipline. The mean response times obtained with several parameter values show that Z should decrease with the load, and the rate of decrease should increase with parallelism overhead (e.g., communication and synchronization overhead).

1.3.3 Dynamic Space Sharing

The subset allocated to an application can change in size and in the processors it contains while the application is running. The main advantage of this strategy over static space sharing is that it can reduce processor fragmentation. A job that is allocated less than the number of processors it had requested can use processors released later, and processors need not remain idle if there is an allocation

request. This is important under moderate loads, when released processors are otherwise likely to remain idle for a relatively long time. The number of processors allocated to a job can also vary according to its parallelism. This can reduce load imbalances within applications, and it may, depending on the overhead of processor releases and reallocations, improve job performance and system efficiency. The overhead depends on the cost of the release/reallocation operations and on their frequency. If process-level parallelism is fine-grained and variable, the number of these operations can be excessive.

1.3.3.1 Process Control

The goal of this technique, proposed by Tucker and Gupta [Tucker 89] for shared-memory multiprocessors, is to reduce the number of context switches induced by traditional process-based time sharing. The number of processes is dynamically controlled so that it does not exceed the number of processors, a processor is dedicated to a single process, and time-multiplexing is avoided.

In the prototype implemented on a Multimax, the processors used by controllable jobs (i.e., jobs whose process parallelism can be varied) are divided evenly among them, however a job is not allocated more than the number of processors it requested. Controllable applications use a threads package that supports the shared task queue model and process control. In comparison to traditional priority-based scheduling, this policy reduces the number of context switches as processes are suspended only when their job's allocation is decreased. Significant reductions in turnaround times were obtained. For some applications, the improvement was by more than a factor of two. An issue with this equipartitioning implementation scheme is that smaller jobs receive a larger fraction of their processor request. Larger jobs are discriminated against and the efficiency advantage of folding is not exploited uniformly across job sizes.

McCann, *et al.*, [McCann 93] implemented and compared three scheduling policies on a Sequent Symmetry multiprocessor, a UMA shared-memory machine. They are called: round-robin job (RRjob), Equipartition, and Dynamic. In RRjob, originally proposed by Leutenegger and Vernon

[Leutenegger 90], a job is assigned n processors for a time interval $t=k/n$ when its turn arrives, where n is the maximum number of processors the job can use at any time during its execution (i.e., its maximum process parallelism) and k is a constant. The unassigned processors are given to the job whose turn is next. In the workload used, the values of n exceed $P/2$, where P is the number of processors in the machine. In Equipartition, the machine is subdivided evenly among the competing jobs and allocation is independent of instantaneous job parallelism. In Dynamic, the allocation depends on actual concurrency. Jobs request processors as they need them, and mark those they cannot currently use as "willing to yield". Free and willing to yield processors are allocated first, then equipartition is enforced by preempting processors from the job(s) with the largest allocation. Jobs schedule their threads using a low-level scheduler that manages a shared task queue and has support for process control.

RRjob resulted in the longest response times as it does not take advantage of the efficiency advantage of folding, and because progress is impeded when only a subset of a job's processes are active. Dynamic yielded better average turnaround times than Equipartition. The improvement was small (about 10%), but significant. The decrease in job idle times it produced was larger than the additional overhead it incurred because of its larger number of processor releases and reallocations.

1.3.3.2 Preemptive Policies

A problem with static space sharing is that waiting times can be excessive when long jobs are running. Assuming that the execution times can be estimated a priori, Naik, *et al.*, [Naik 93b] used simulation to study a preemptive space sharing policy that gives priority to short jobs. The target system is a distributed-memory multiprocessor, and jobs are classified a priori as short, medium, or long. When a job arrives, processors can be preempted from medium and long jobs, but not from short jobs. It is assumed that medium and long applications can be dynamically reconfigured. When processors are released, they are subdivided evenly among waiting applications under the constraint that they do not receive more than their processor request or less than some fixed number of processors.

When compared to their nonpreemptive limited folding static scheme, discussed earlier (see Section 1.3.1.3), this policy produced better average response times for short jobs, but the performance of medium and long jobs suffered under medium to high loads. In their experiments, the execution times of medium and long jobs are considerably longer than those of small jobs. Issues with this scheme are implementing application reconfigurability, the cost associated with preemptions and re-configurations, and the assumption that jobs can be classified in advance according to their execution times.

1.4 Other Related Scheduling Disciplines

1.4.1 Scheduling in the Xylem Operating System

This operating system runs on the Illinois Cedar computer, a clustered NUMA machine. A parallel program runs as a Xylem process, which contains one or more tasks assigned to Cedar clusters for their lifetimes. The tasks are scheduled independently. The Cedar Fortran runtime library has three variations, which are called Queued, Simple, and Static. The Queued and Simple variations support the shared queue subtask scheduling model. In the Queued version, a wait-then-block technique, proposed by Ousterhout [Ousterhout 82], is used to reduce the number of context switches. A task holds onto its processor for some time interval when it has to wait for an event. The hope is that the event will happen before the interval expires, and thus avoid the context switch. The task is blocked if the event does not occur within the interval [Eigenmann 91].

There are two problems with the wait-then-block technique. The first problem is that the best interval length is difficult to determine because it is application-dependent. Having the programmer specify this length is not a good solution because users should not be trusted in setting it. The second problem is the time that a task, once blocked, spends in the suspended state. Other tasks that interact with it may also be preempted, especially under heavy load when a blocked task is likely to spend a

long time in the suspended state. An excessive number of context switches can result, depending also on the granularity of process interactions. Notwithstanding the wait-then-block technique, Eigenmann, *et al.*, note that "Synchronization delays can be quite high because the task scheduler on each cluster and the microtask schedulers in each process work independently." [Eigenmann 91, page 5]. Space sharing that guarantees the simultaneous execution of cooperating tasks is a potential solution to this problem.

The Simple library version uses busy-waiting synchronization. Tasks do not surrender their processors when they must wait. Reductions in mean response times for individual applications were obtained under light conditions. A problem with busy-waiting is that it can lead to excessive spinning times for synchronization events. In the Static library version, loops are statically mapped and dispatched. For example, the iteration space of a parallel loop can be distributed among the tasks. The critical section that controls access to the shared subtask queue is avoided, but the load imbalances can be high.

1.4.2 Round-Robin Coscheduling

In this strategy, proposed by Ousterhout [Ousterhout 82] for multiprocessor systems that permit extensive (fine-grained) process interactions, an application's runnable processes are dispatched and preempted together. Coscheduling solves the problem of excessive synchronization and communication delays that can result when a proper subset of cooperating processes is not running. However, it suffers from several sources of inefficiency, including the corruption of cached code and data by interleaved applications, the potential need to swap groups of processes at the same time, and the overhead of context switches. Moreover, coscheduling, as specified by Ousterhout, does not take advantage of the increase in efficiency that typically results when an application executes on fewer processors. A job is assigned to as many processors as it requests. However, folding can be used with coscheduling.

1.4.3 Abstract Process-Based Scheduling Policies

Assuming that parallel jobs consist of *independent* processes, Majumdar, *et al.*, [Majumdar 88] studied several abstract process-based scheduling policies for shared-memory multiprocessors, including FCFS, round-robin process (RRprocess), preemptive smallest cumulative demand first (PSCDF), smallest number of processes first (SNPF), and preemptive SNPF (PSNPF). In PSCDF , for example, processes that belong to the job with the shortest remaining total demand are given preemptive priority. Processes belonging to the job with the smallest number of processes that have not yet completed are given preemptive priority in PSNPF. PSCDF produced the best average response times, however it requires that the remaining execution times be known.

Leutenegger and Vernon [Leutenegger 90] also evaluated these policies using simulation and assuming a shared-memory multiprocessor. They concluded that their general performance ordering, from best to worst, is: PSCDF, RRprocess, PSNPF, SNPF, FCFS. However, RRprocess can perform worse than SNPF when the total service demand of jobs is linearly dependent on the number of processes. In this case, a job with a small number of processes is likely to have a small total processing demand. A fundamental problem with these policies, with the possible exception of RRprocess, is the assumption that processes are independent. Moreover, experimental evidence shows that round-robin process-based policies can perform poorly in shared-memory multiprocessors when the number of processes exceeds the number of processors [Tucker 89][Markatos 93].

Leutenegger and Vernon [Leutenegger 90] also studied two non process-based policies that attempt to allocate an equal fraction of the processing power to each job: round-robin job (RRjob), and process control, proposed by Tucker and Gupta [Tucker 89]. These policies produced comparable mean response times. However, equipartition outperformed RRjob significantly in experiments by McCann, *et al.*, [McCann 93].

1.4.4 Job Scheduling with Detailed Characterization of Parallel Execution

The execution of a parallel program on dedicated processors is commonly characterized by its *execution curve*, which gives the execution times as a function of the number of processors used. Intuitively, superior space sharing policies can be designed if the execution curves are known in advance. Assuming that the influence of multiprogramming on the performance of applications is negligible, the on-line job scheduling problem can be considered as an instance of the on-line two-dimensional bin-packing problem when jobs are allocated contiguous linear subarrays of processors [Coffman 91]. One of the dimensions is space (i.e., the processors), and the second is time. When folding is supported, the rectangles to pack are malleable and superior packing is possible. The allocation algorithm can, for example, ensure that folding does not delay job completion because it can determine when the busy processors will be released. Alternatively, the algorithm may allocate more processors to short or efficient programs.

Zahorjan and McCann [Zahorjan 90] compared static space sharing, dynamic space sharing, and round-robin coscheduling using three algorithms, each representing one of these scheduling classes. The static space sharing algorithm bases allocation on the execution curves. The execution time of a waiting job is defined to be very large, and the job whose execution time will decrease the most following the allocation of an additional processor is given priority. Thus, an application is allocated one processor before any receives additional processors. There are several problems with this scheme. First, it supports eager unlimited folding, which, as it will be shown in Chapter 2, can result in excessive folding and poor mean response times. For example, when a job that is allocated two processors terminates, two waiting jobs may be serviced. The jobs may complete much sooner if they wait for more processors to become available. The second problem is that the algorithm will typically give priority to long jobs as their execution is likely to decrease the most when they are allocated additional processors. The coscheduling algorithm supports folding and process migration in order to exploit the efficiency advantage of folding and reduce processor fragmentation. In the dynamic policy, applications

request processors as they need them, but allocation does not depend on the execution curves. If there are no free processors when a new job arrives it is allocated a processor taken away from a job that is allocated multiple processors. If any part of a job's request is not satisfied it waits. When processors are released, waiting applications that are allocated no processors are given priority for the allocation of their first processor. Each is allocated one processor, if possible. The remaining processors are then allocated to requests for additional allocation on FCFS basis. A problem with this scheme is that some active jobs may be allocated a large number of processors while others that arrived later are allocated a few processors.

The three policies were compared using simulation. The workload consists of several program structures that exhibit variable parallelism, and the target multiprocessor is UMA. The results show that the dynamic algorithm can outperform the static algorithm depending on the overhead of processor releases and reallocations. The round-robin coscheduling scheme produced the worst mean response times [Zahorjan 90].

K. Sevcik [Sevcik 89] studied the influence of using several characteristics of program parallelism on the performance of static space sharing. The characteristics for application j are the average parallelism A_j, variance of parallelism, V_j, minimum parallelism, m_j, and maximum parallelism, M_j. He proposed that job j be allocated A_j processors across all system loads if V_j=0, otherwise it should be allocated fewer processors when the load is higher. The rate of allocation decrease should increase with V_j. When V_j is high, the job should be allocated M_j processors under very light load and m_j processors under very high load. The simulation results show that folding can produce significantly shorter mean response times when the variability of parallelism is greater than zero.

1.4.5 Hierarchical Process Allocation Algorithms

As the size of the target system increases, a centralized scheduling algorithm may become a bottleneck. To address this problem, Feitelson and Rudolph [Feitelson 90] proposed that the scheduling

function be carried out by dedicated hierarchical control processors. In their simulation study of hierarchical coscheduling, a binary tree of controllers is used. Processors at level i, in cooperation with their subordinates, are responsible for scheduling tasks that require a number of processors in the interval $[2^{i-1}+1,2^i]$. The tree leaves are at level 1. A parallel machine of size P requires P-1 control processors. An issue is whether such high cost is justifiable even if the control processors are, as proposed, less expensive than the processing nodes. Moreover, the bottleneck problem persists if many applications have large processor requirements as the processors in the higher levels of the tree, which handle large jobs, are smaller in number.

In the multiprogrammed fixed equipartitioning policy proposed by Ahmad, *et al.*, [Ahmad 94] and discussed earlier, a two-level scheduling hierarchy is used. The target (a hypercube multicomputer) is partitioned into a fixed number of spheres, the host assigns a main task to a single sphere, and the sphere's median assigns its subtasks to processors within the sphere.

1.4.6 Off-Line Scheduling

In off-line scheduling, the list of applications to be scheduled is fixed (i.e., the job arrival process is not dynamic). The policies discussed above are on-line. The problem of allocating processors to a list of applications can be viewed as an instance of the widely-studied off-line two dimensional bin packing problem [Baker 80][Coffman 80][Coffman 91] when the execution times on the number of processors requested are known, the jobs are allocated contiguous linear processor subarrays, and the goal is minimizing the overall completion time.

Assuming the execution curves are known, Krishnamurti and Ma [Krishnamurti 92] proposed an off-line heuristic for scheduling application lists on partitionable multiprocessors. The sizes of the partitions are a subset of $\{1,..,P\}$. In the heuristic, each application is allocated the smallest partition size initially. This step is followed by an iterative procedure that allocates the smallest number of additional processors to the application that would complete last if it executed on the number of processors

it is currently allocated. The goal is to produce a short overall completion time.

Off-line scheduling is not considered in this research. However, the same heuristic is often used in off-line and on-line scheduling (e.g. first-fit). A heuristic may perform well in both cases for the same reason.

1.5 System and Workload Models

The target multiprocessor for this work consists of P identical processing elements, an application is allocated its own subset of processors, and allocation is topology-independent. The algorithms studied in this dissertation can also be used when parallel programs are organized as tasks that can be scheduled independently, where a task is a collection of cooperating processes. In this case, processors are allocated to tasks instead of applications.

It is assumed that a new application requests a number, n, of processors, and that n does not exceed P. Unless it is specified otherwise, other job characteristics are not assumed to be known and the sole job characteristic used in making allocation decisions is n.

The algorithms are evaluated using simulation. The simulator was developed in the C programming language, and is event-driven. Jobs arrive from a Poisson source. The system load parameter, L, is computed using the equation $L=(\lambda*N*T_e)/P$, where N is the mean processor request, T_e the mean execution time, and λ the arrival rate of jobs.

To study the effects of the efficiency advantage of folding on the performance of the scheduling policies, it is assumed that a job consists of a serial and a parallelizeable components in some of the experiments. The corresponding speedup is sublinear and monotonically increasing in the number of processors allocated, and is denoted by the acronym MISP in this dissertation. The values of $\xi(n)$ are generated using a pseudo-random variable distributed uniformly over a subset of the interval [0,1] in these experiments. However, the efficiency values that correspond to a serial fraction exceeding 0.5 are discarded. As $\xi(m)$ increases when m decreases, L can exceed one without saturating the system when

the allocation policy supports folding.

The processor request n, execution time $t(n)$, and efficiency $\xi(n)$ generated by the simulator for a new job are used in computing the job's execution time when it is allocated fewer than n processors. The job's serial fraction, f_S, is computed by the equation:

$$f_s = \frac{1-\xi(n)}{\xi(n)(n-1)}$$

and the execution time on m processors, $t(m)$, is computed by:

$$t(m) = \frac{n[f_s(m-1)+1]}{m[f_s(n-1)+1]} t(n)$$

Linear speedup is assumed in other simulation experiments. The execution time $t(m)$ is computed then by the equation $t(m)=n*t(n)/m$.

The distributions of the execution times and the values of n are not well-known. However, they are often assumed to be exponential or uniform. The results of a Cray X-MP (a pipelined vector processor) workload characterization study over a two-month period show that about 28% of the private jobs were short, 44% were medium, and 29% were long. The number of jobs submitted over the period exceeded 60000 [Pasquale 91]. These results suggest that the exponential distribution, although commonly used, may not be appropriate for modeling the execution times of scientific applications.

The distribution of n depends on machine size, machine architecture, and program characteristics. Determining the number of processors an application should use is an active area of research, but is still mainly performed by application developers. As in other studies, the uniform and exponential distributions are used to model the values of n and $t(n)$.

The allocation algorithms are evaluated and compared using mean response times and scheduling effectiveness, and fairness curves. The fairness curves display the average turnaround times as functions of n under load levels of interest, and the scheduling effectiveness measures the ability of the algorithms to utilize the processors (avoid fragmentation).

1.6 Problem Summary

The folding method, the job selection criteria, the system load, and workload characteristics are fundamental factors that influence system performance under space sharing job scheduling policies. The goals of this research were to:

1) Investigate adaptive limited folding static space sharing.
2) Compare no folding, unlimited folding, and limited folding static space sharing.
3) Compare dynamic schemes that reduce job waiting times by executing many jobs simultaneously (including the promising previously proposed dynamic equipartitioning policy), dynamic schemes that limit the number of active jobs, and promising static schemes.
4) Study the effect of giving priority to shorter jobs and jobs with smaller processor requests on the performance of static and dynamic space sharing.

No folding and unlimited folding are compared in the next chapter. Adaptive static space sharing that supports limited folding is studied in Chapter 3. Dynamic space sharing policies are the subject of Chapter 4. Finally, Chapter 5 contains a summary of the conclusions and recommendations for future work.

2 TRADITIONAL AND UNLIMITED FOLDING STATIC SPACE SHARING POLICIES

In this chapter, several no folding and unlimited folding static space sharing policies that support topology-independent program-based allocation are studied and compared. The results of a detailed simulation study of their performance show that unlimited folding is superior to no folding under high system loads when the efficiency of applications increases significantly with a decrease in the number of processors allocated. However, when the applications are highly efficient, no folding is superior under most or all loads, depending on workload characteristics. The results also show that the job selection criteria has a significant effect on performance within both policy classes. Several algorithms based on the assumption that the execution times of applications on the number of processors they request can be estimated in advance are also studied. The results show that giving priority to shorter jobs can improve mean response times significantly.

2.1 Introduction

The two basic methods for implementing static space sharing in parallel systems are machine partitioning and program-based partitioning. In the machine partitioning method, the target system is subdivided into disjoint partitions independently of individual applications, and an application is commonly allocated a single distinct partition. In program-based partitioning, distinct partitions are created for applications as they are serviced.

Topology-independent program-based space sharing has two major advantages over machine partitioning that assigns an application to a single partition (the common mapping approach). First, it avoids internal processor fragmentation. Second, a job can be allocated any number of processors.

Significant performance improvement can result under moderate loads, when free machine partitions are not likely to be allocated soon.

Two classes of program-based space sharing policies are considered in this chapter. In the (traditional) no folding class of policies, a parallel program is always allocated the number of processors it requests. This number can be determined at compile-time, and several data access optimizations can be supported. For example, in the Rice University FORTRAN D compiler for distributed-memory machines, programs are compiled for a specific number of processors, and fundamental data distribution constructs and communication overhead reduction techniques that depend on this number are supported [Hiranandani 92].

In the unlimited folding class of policies, a job can be allocated any number of processors that does not exceed its processor request. The optimizations that are possible when the number of processors on which the program will run is fixed prior to execution (e.g., static data distribution) can still be supported. The compiler must generate object code that can be bound to the allocated processors. Such model is supported by MIT's Id compiler for their Monsoon dataflow system. The object code it generates can run on any number of processors [Hicks 93].

With unlimited folding, a processor does not remain idle if there is a pending allocation request, and the increase in efficiency that typically results when a job executes on fewer processors is exploited. However, a job may be allocated a very small fraction of the number of processors it requested, causing it to complete much later than if it waits until that many processors are available. Because a folded application can not be allocated additional processors (processor allocation is fixed in static space sharing), folding fragmentation results. This type of fragmentation exists when processors are free and there is one or more folded applications. As released processors are likely to remain unallocated for a long time under moderate loads, high folding fragmentation and poor system utilization can result.

Several policies that support unlimited folding and no folding were the subject of recent stud-

ies. The unlimited folding policy FF+FIFO, described in more detail in the preceding chapter, is compared to FF, equipartitioning, and other less promising disciplines in [Ghosal 91]. It produced the best performance, including when equipartitioning uses the best number of partitions considered in the study. The performance parameter is the throughput to mean response time ratio, and the workload is a mix of a small set of five applications. An application is allocated a single partition in the equipartitioning policy, but it is allocated at most the smallest number of processors that maximizes the speedup*efficiency parallel execution cost function in FF and FF+FIFO.

There are several issues with this study. First, a job is not allocated the number of processors that minimizes its execution time under FF and FF+FIFO, including when the load is moderate and free processors are likely to remain unallocated for a long time. Second, the speedup curves are difficult to estimate for many applications. The execution time of a program can strongly depend on the input, and numerous applications have dynamic structures. Third, it is intuitive that superior scheduling algorithms can be designed if the execution curves are known. For example, a job may wait until more processors are available when waiting leads to better performance.

In the static space sharing policy proposed by Zahorjan and McCann [Zahorjan 90], an application is allocated at most the smallest number of processors that maximizes its speedup, and folding is unlimited. When a job terminates, a released processor is assigned to the waiting program whose execution time will be reduced the most if it receives an additional processor. The execution time of an application that is allocated no processors is set to a very large value. Consequently, waiting applications are allocated one processor each first. Zahorjan and McCann note that this property had a critical influence on the performance of the scheme. It is assumed that the execution times on one processor and the speedup curves of applications are known.

There are two issues with this policy: (1) applications that have long execution times are given priority, and (2) servicing is eager (the maximum number of waiting applications are serviced when processors are released). Many applications are expected to be allocated a small number of processors,

and released processors are likely to idle for a long time because the waiting queue is expected to be short. An advantage of this policy over FF+FIFO as implemented by Ghosal, *et al.*, [Ghosal 91] is that a program can be allocated the number of processors that minimizes its execution time. An eager policy is compared to other disciplines that support unlimited folding in this study. It produces the worst scheduling effectiveness and mean response times. The results also show that giving priority to longer jobs can degrade performance significantly.

The traditional FF and FCFS policies were compared in a simulation study by Abraham and Padmanabhan [Abraham 92], where FF produced significantly shorter mean response times. The advantage of FF is that it can allocate processors to any waiting request, whereas the head of the FIFO waiting queue must be serviced first in FCFS. FF can yield superior system utilization. These policies do not support application folding.

The study reported in this chapter has two main goals. The first is to compare the no folding and unlimited folding approaches to determining the number of processors to allocate to applications. Under both approaches, the role of the job selection criteria is investigated. The second goal is to determine how allocation should depend on the execution times; that is, should short or long jobs be given priority?.

2.2 Allocation Policies

The target parallel computer consists of P identical processors, and allocation is topology-independent. It is assumed that an application requests, upon arrival, a number of processors, denoted by n, from the allocation algorithm.

2.2.1 No Folding Allocation Policies

An application is always allocated the number of processors it requested. The following policies are studied:

First Come First Served (FCFS): Allocation requests are serviced in their order of arrival. The advantage of this policy is its fairness. However, it is expected to produce poor scheduling effectiveness and mean response times. Processors remain idle if the application at the head of the waiting queue requests more than the number of free processors, even though there may be a waiting request that can be satisfied. Algorithms based on the first-fit allocation strategy solve this problem by allowing waiting requests to be serviced out of arrival order.

First Fit (FF): The waiting queue is FIFO. A new application waits at the tail of the queue if the number of free processors, *FP*, is less than its processor demand. When processors are released, the waiting queue is scanned and the first job whose processor requirement does not exceed *FP* is serviced. Scanning continues until *FP* is zero or until all jobs in the queue have been examined. Processors do not idle if their number is at least equal to the processor demand of a waiting application.

First Fit Decreasing Size (FFDS): The waiting jobs are sorted in the non-increasing order of their processor requests, and allocation is as in FF. The goal is to increase the number of busy processors by giving priority to the largest job that can fit.

First Fit Increasing Size (FFIS): The waiting jobs are sorted in the non-decreasing order of their processor requests, and allocation is as in FF. The goal is to reduce the mean response times by increasing the number of jobs running simultaneously. Intuitively, FFIS discriminates against large jobs more than the previous algorithms.

Policies based on the best-fit strategy are not considered as best-fit appears to offer only a slight advantage [Tuomenoksa 85] or some disadvantage [Ghosal 91] over first-fit.

2.2.2 Unlimited Folding Policies

A job may be allocated *any* number of processors that does not exceed its request. The four algorithms that result from the following modification to the no folding algorithms are included. An arriving job is allocated $\min(n,FP)$ processors if $FP>0$, otherwise it waits. At a job completion, the cor-

responding no folding algorithm is run. If processors are still free they are allocated to the head of the waiting queue. The resulting policies are named FCFSUF, FF+FIFO [Ghosal 91], FFDS+FIFO, and FFIS+FIFO respectively.

In addition, the following policy, denoted by EPFP (from Even Partitioning of Free Processors), is studied. The waiting queue is FIFO. As in the other unlimited folding policies, a new application is allocated $\min(n,FP)$ processors if $FP \neq 0$, otherwise it waits. Released processors are divided as evenly as possible among the waiting applications. When the number of jobs does not divide FP, the excess processors are allocated to the earliest arrivals, one to each. Applications are serviced eagerly; that is, the maximum number of waiting applications is serviced when a job terminates.

2.3. Results

The algorithms are compared using mean response times and scheduling effectiveness, and fairness curves.

2.3.1 Simulation Parameters

In the simulation experiments, the machine consists of $P=64$ identical processors. A new job is characterized by three independently-generated parameters: the requested number of processors n, the execution time $t(n)$, and the efficiency $\xi(n)$. Two distributions are used to model the values of n: the uniform over the interval $[2,P]$, and the truncated exponential with a mean of 15 and $2 \leq n \leq P$. This mean was chosen because it produced worse mean response times than other values in several simulation experiments. The values of $t(n)$ are distributed uniformly over the interval [10,200]. A pseudo-random variable distributed uniformly over [0.4,0.9] produces the values of $\xi(n)$ under MISP speedup. The efficiency values that correspond to a serial fraction greater than 0.5 are discarded. The characteristics n, $t(n)$ and $\xi(n)$ are used in computing the execution time when the job is folded, as seen in Chapter 1.

During each run, the simulator generates 8500 jobs. To ignore startup effects, the performance data obtained for the first 500 jobs is discarded.

2.3.2. Scheduling Effectiveness

This performance parameter measures the ability of the algorithms to avoid processor fragmentation. Utilizing a large percentage of the PEs is essential to the performance of parallel computing systems as both program efficiency and scheduling effectiveness must be high for the system efficiency to be high. The definitions of system efficiency and scheduling effectiveness are given in Chapter 1. The effectiveness values shown below have relative errors below 1% (they are typically much less than 1%) when the confidence interval is 95%.

2.3.2.1 The no folding policies

A comparison of the mean scheduling effectiveness of the no folding policies is shown in Figures 2.1 and 2.2. Their general ordering, from best to worst, is: FFDS, FF, FCFS, and FFIS. The mean

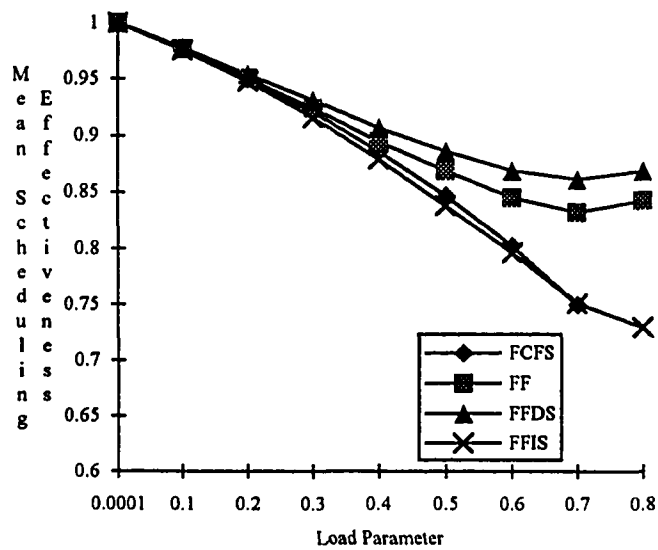

Figure 2.1: Mean scheduling effectiveness as a function of the load parameter for the no folding policies, uniform size and execution time distributions.

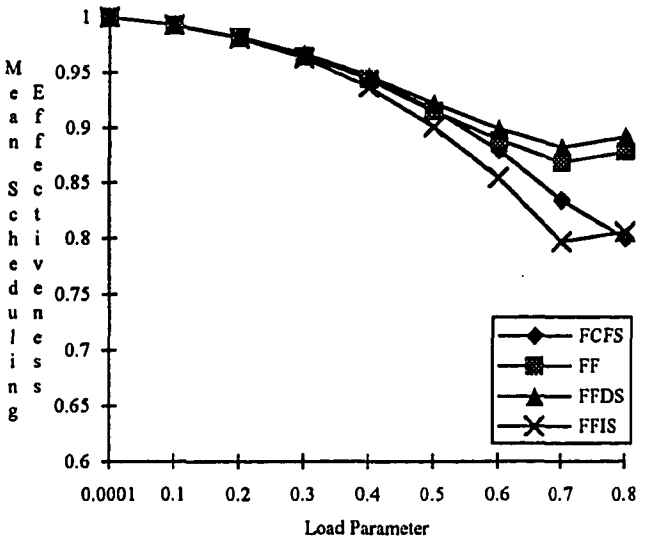

Figure 2.2: Mean scheduling effectiveness as a function of the load parameter for the no folding policies, exponential size distribution, uniform execution time distribution.

effectiveness is high under light loads because it is highly likely that a job is serviced as soon as it arrives. It drops as L increases because of processor fragmentation, which occurs when the number of free processors is less than the processor request of the head of the queue under FCFS, and when it is smaller than the request of each waiting job under the first-fit policies. When the requests are exponentially distributed, fragmentation is less severe and the mean S_e is higher because most jobs are small.

The effectiveness curves of FFDS and FF increase under heavy loads because the number of waiting jobs and the probability of finding a job that fits are higher. This also explains why the effectiveness of FFIS declines slowly (Figure 2.1) or increases (Figure 2.2) under heavy loads.

2.3.2.2 The unlimited folding policies

These policies do not differ much in their scheduling effectiveness (approximately 5% at most), as can be seen in Figures 2.3-2.5, and they are less effective than the no folding policies, except under very high loads (e.g., Figures 2.6 and 2.7). Their effectiveness curves drop more rapidly as the system

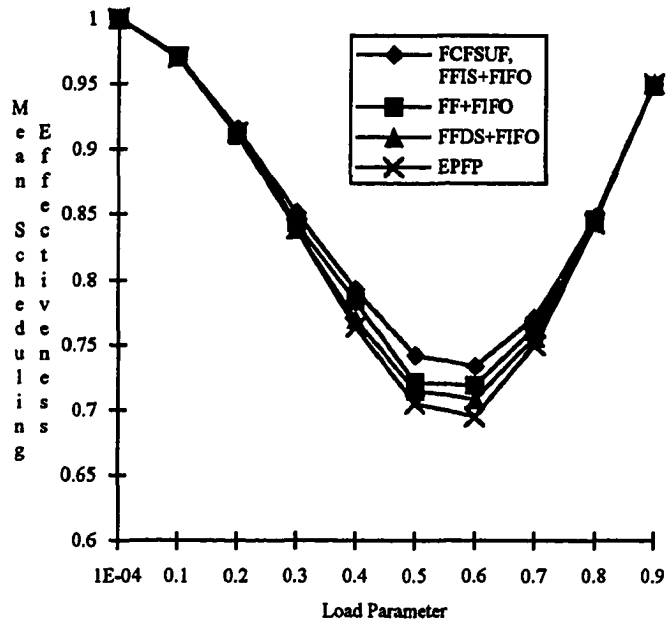

Figure 2.3: Mean scheduling effectiveness as a function of the load parameter for the unlimited folding policies, exponential size distribution, uniform execution time distribution, linear speedup

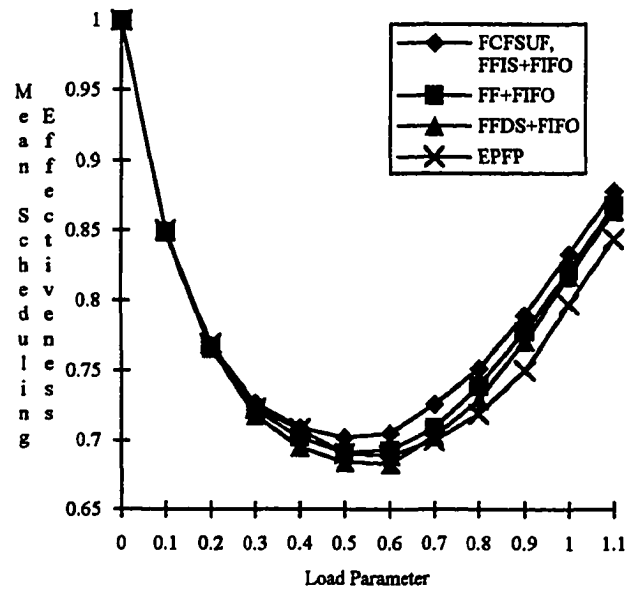

Figure 2.4: Mean scheduling effectiveness as a function of the load parameter for the unlimited folding algorithms, uniform size and execution time distributions, MISP speedup

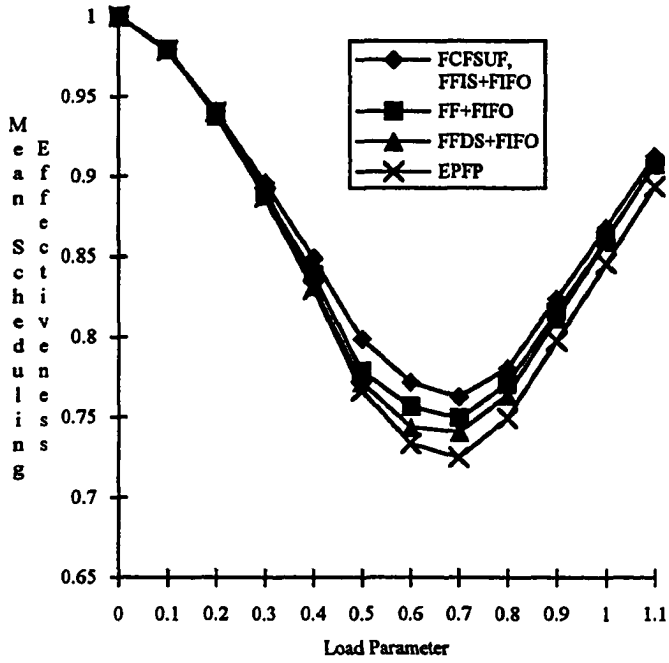

Figure 2.5: Mean scheduling effectiveness as a function of the load parameter for the unlimited folding algorithms, exponential size distribution, uniform execution time distribution, MISP speedup

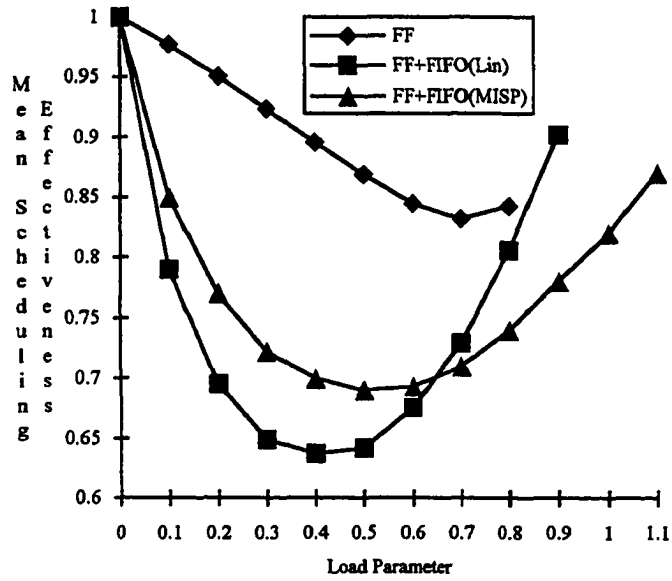

Figure 2.6: Mean scheduling effectiveness as a function of the load parameter for the FF policies, uniform size and execution time distributions.

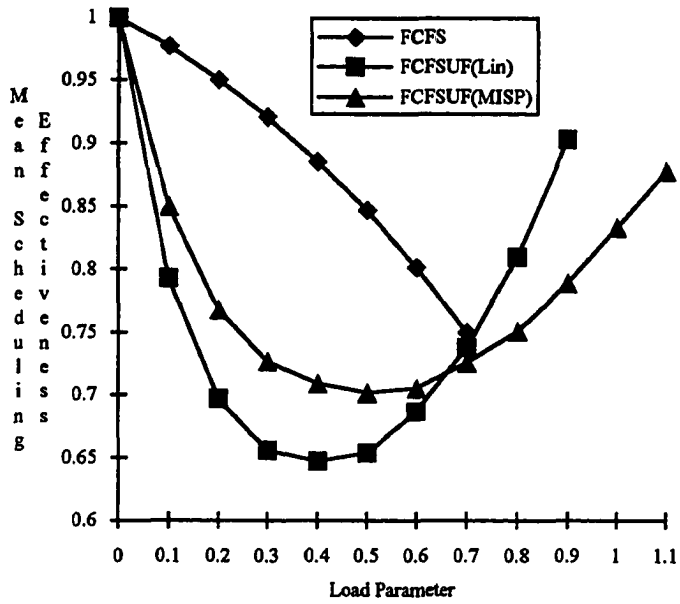

Figure 2.7: Mean scheduling effectiveness as a function of the load parameter for the FCFS policies, uniform size and execution time distributions.

is loaded, reach minimum values at medium load levels, then rise rapidly as the load increases further. When the load is very light, the effectiveness is high because most applications are not folded and the factor by which an application is folded is small, on average. As the load increases, the percentage of folded applications and the factor by which an application is expected to be folded also increase; that is, the mean actual folding factor increases with the load (e.g., Figure 2.8). This factor is the average, over all jobs, of the ratio n/P_{al}, where n is a job's size and P_{al} the number of processors it is allocated.

Under medium system loads, the probability that there are no pending requests when processors are released is high, released processors are likely to remain idle for a long time, and high folding fragmentation and poor scheduling effectiveness result. Note that the mean length of the waiting queue is shorter under unlimited folding than under no folding. Under heavy loads, the probability that there is a pending request is higher, released processors are likely to be allocated sooner, and the scheduling effectiveness increases with the load.

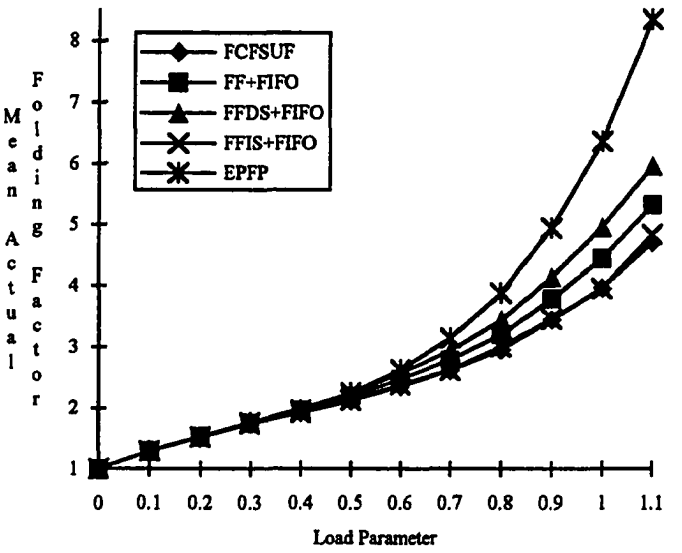

Figure 2.8: Mean actual folding factors as a function of the load parameter for the unlimited folding policies, uniform size and execution time distributions, MISP speedup

In FCFSUF, the application at the head of the waiting queue does not, unlike in FCFS, wait until the exact number of processors it requested is available, and the scheduling effectiveness is not much different from that of the unlimited folding FF policies. Because of the absence of the first-fit phase, FCFSUF folds jobs less than FF+FIFO, as can be seen in Figure 2.8. Unlike FFIS, which suffers significantly more fragmentation than FF and FFDS, FFIS+FIFO is slightly more effective than the other two unlimited folding FF policies. Also, its mean folding factors are smaller. FFDS+FIFO produces the largest mean folding factors, and is the least effective FF variant. When compared to the other unlimited folding policies, EPFP, which services the maximum number of waiting jobs, results in much higher mean actual folding factors and lower scheduling effectiveness under high loads.

2.3.3 Mean Response Times

The typical user of a general purpose parallel machine is unlikely to be directly interested in the scheduling effectiveness or system throughput. To this user, the expected response time is far more

important. In this section, the mean response times of the policies are compared. The values shown have relative errors that do not exceed 5% when the confidence interval is 95%.

2.3.3.1 The no folding policies

FFDS and FF produced shorter mean response times than FCFS and FFIS (Figures 2.9 and 2.10) because of their ability to better avoid processor fragmentation (Figures 2.1 and 2.2). Although the goal of FFIS is to reduce the mean response times by simultaneously executing more applications, it produced longer mean response times than FF and FFDS, including when most jobs are small (the sizes are exponentially distributed), because its effectiveness is poorer. As expected, FCFS has the worst performance due to the high fragmentation it induces. The performance of the policies is practically the same under low loads ($L \leq 0.3$), when applications seldom have to wait.

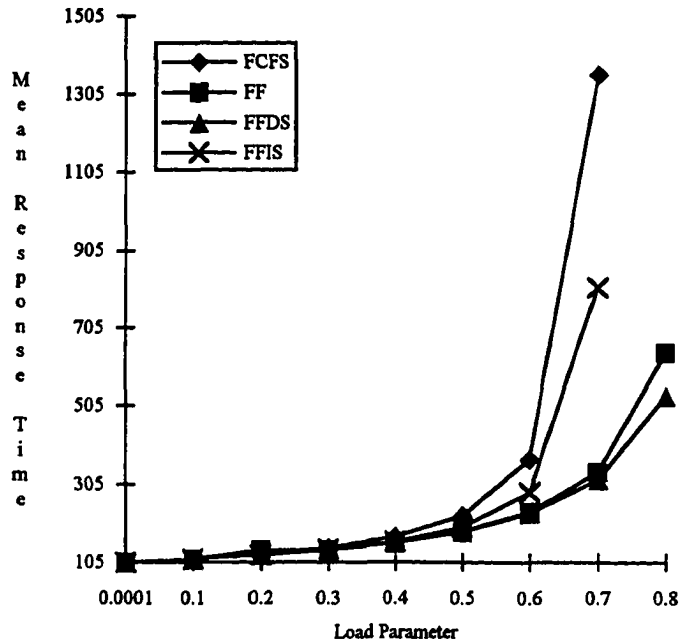

Figure 2.9: Mean response time as a function of the load parameter for the no folding algorithms, uniform size and execution time distributions

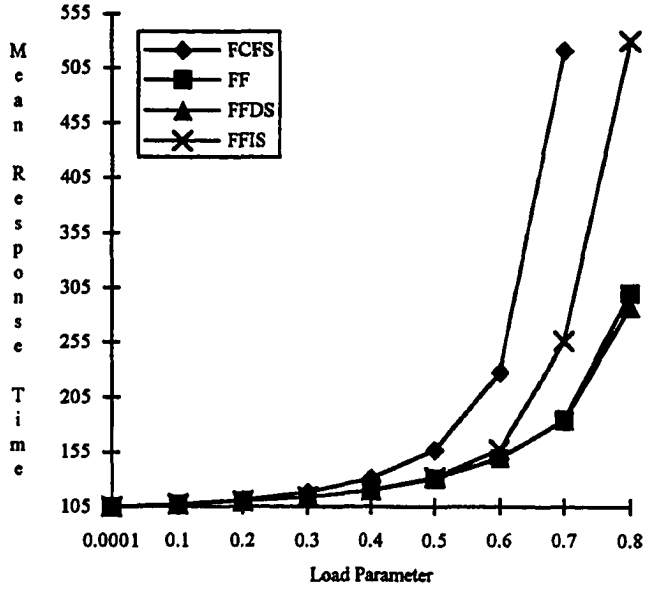

Figure 2.10: Mean response time as a function of the load parameter for the no folding algorithms, exponential size distribution, uniform execution time distribution

2.3.3.2 The unlimited folding policies

A study of the mean response times of the unlimited folding policies (Figures 2.11-2.13) leads to the following observations:

1) The unlimited folding variants of the no folding policies differ much less in the mean response times they produced. Folding reduced the importance of the job selection criteria, and it has a strong influence on performance (see also Figures 2.6 and 2.7).

2) As the performance of EPFP is poor, the eager servicing of applications, recommended in [Zahorjan 90], is a poor job selection strategy.

3) There is positive correlation between the mean actual folding factors and the mean response times. Less folding is better.

4) Although FFDS is the best no folding first-fit policy, FFDS+FIFO is the worst unlimited folding first-fit policy because it folds jobs more than FF+FIFO and FFIS+FIFO (e.g., Figure 2.8).

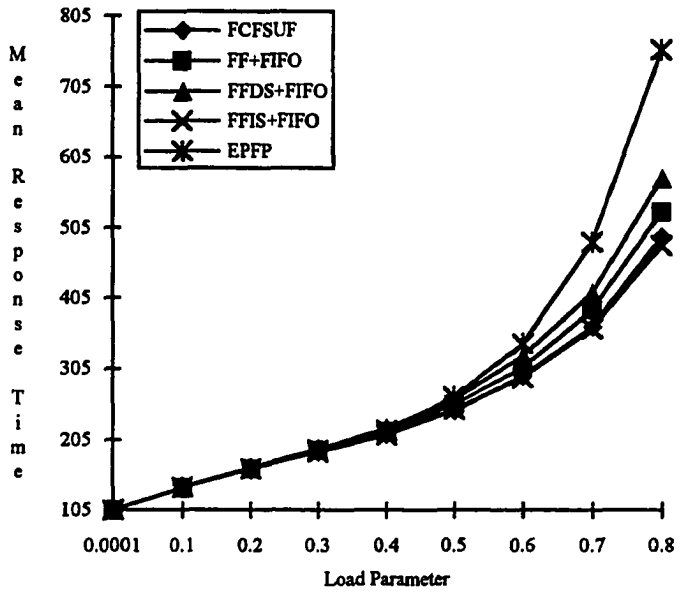

Figure 2.11: Mean response time as a function of the load parameter for the unlimited folding policies, uniform size and execution time distributions, linear speedup

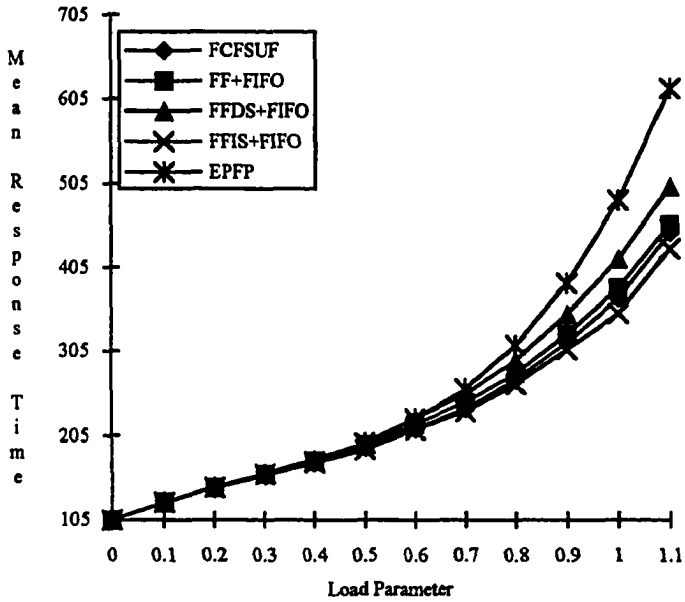

Figure 2.12: Mean response time as a function of the load parameter for the unlimited folding policies, uniform size and execution time distributions, MISP speedup

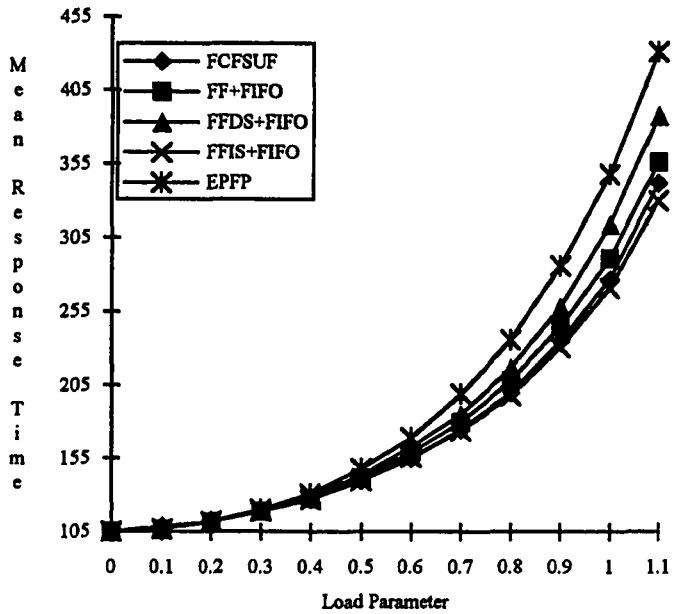

Figure 2.13: Mean response time as a function of the load parameter for the unlimited folding policies, exponential size distribution, uniform execution time distribution, MISP speedup

5) FCFSUF is slightly superior to FF+FIFO. FCFSUF is more effective, and its mean actual folding factors are smaller. This is unlike FCFS, which is much worse than FF.

6) Although FFIS is worse than FF and FFDS, FFIS+FIFO is superior to FF+FIFO and FFDS+FIFO because it folds jobs less, on average, while achieving comparable scheduling effectiveness.

7) When the efficiency of applications increases significantly with a decrease in the number of processors they are allocated (e.g., MISP speedup model), no folding is better than unlimited folding under low to moderate loads but unlimited folding is superior under high to very high loads, as can be seen in Figures 2.14 and 2.15. However, when linear speedup is assumed, no folding is superior to unlimited folding under most or all system loads, depending on the distribution of the processor requests, as can be seen in the same figures.

For example, FF starts outperforming FF+FIFO at $L \approx 0.75$ under linear speedup in Figure

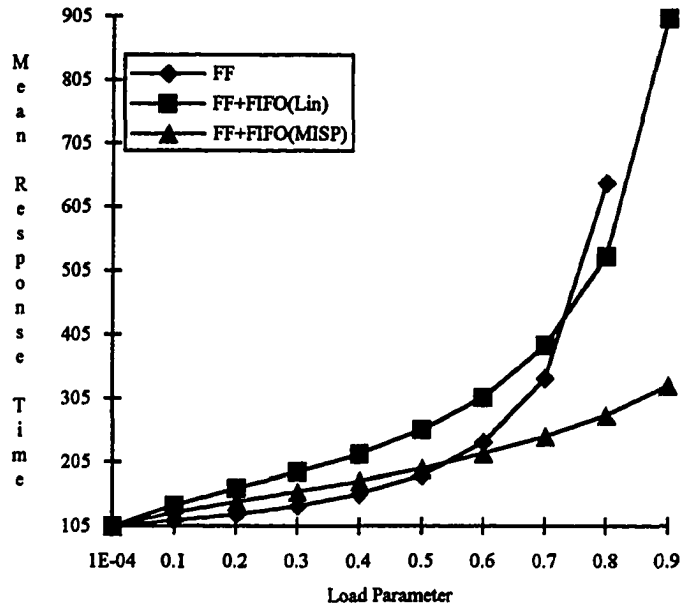

Figure 2.14: Mean response time as a function of the load parameter for the FF policies, uniform size and execution time distributions

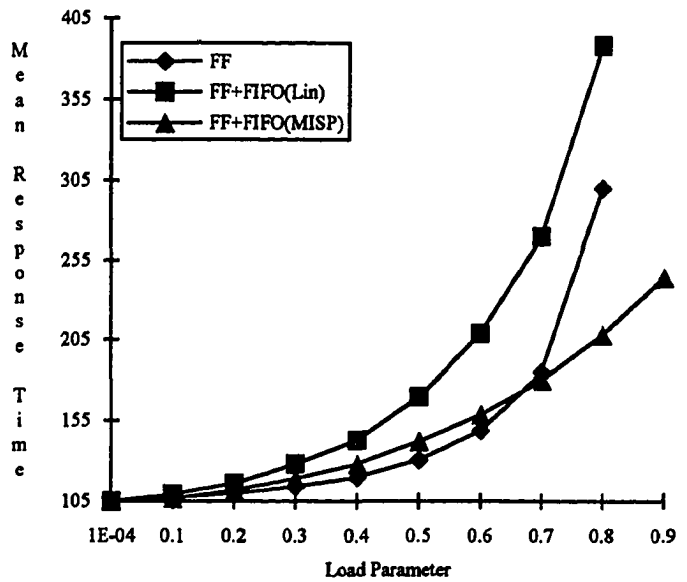

Figure 2.15: Mean response time as a function of the load parameter for the FF policies, exponential size distribution, uniform execution time distribution

2.14, whereas the crossover load level in the same figure is approximately 0.55 under MISP speedup. The dependence of the crossover load levels on the distribution of job sizes is also illustrated in Figures 2.14 and 2.15. Under linear speedup, for example, FF outperforms FF+FIFO across all loads in Figure 2.15, but FF+FIFO is better than FF under heavy loads in Figure 2.14.

As the efficiency of parallel jobs normally increases when the number of processors they are allocated decreases, unlimited folding is expected to outperform no folding significantly under high to very high loads, but no folding is expected to be superior under light to medium loads.

Although a folded job begins execution sooner, it may complete later than if it waits for more processors to become available, depending on its starting time and the number of processors it is allocated. Superior scheduling algorithms can be designed if the execution times as functions of processor allocation are known a priori. For example, the algorithm can ensure that a folded application does not complete later than it would if it waits for the number processors it requested to become available.

2.3.4 Fairness

A fairness curve gives the average response times as a function of the job size (i.e., processor request) at a load level of interest. The job size is chosen because it is the sole job characteristic used in making the allocation decisions. Different criteria should be used if other characteristics are utilized.

There is a tradeoff between fairness and performance. FCFS is relatively fair. The mean turnaround times of large jobs are not much longer than those of small jobs (see Figures 2.16 and 2.17). However, its mean response time performance is poor under medium to high loads. The performance of FF and FFDS is significantly better than that of FCFS, but they are not as fair. For example, when the sizes are exponentially distributed, the turnaround times of large jobs are considerably longer than those of small jobs under the three no folding FF variants (Figure 2.17). Based on the three performance parameters, FFIS and FCFS are poor no folding policies.

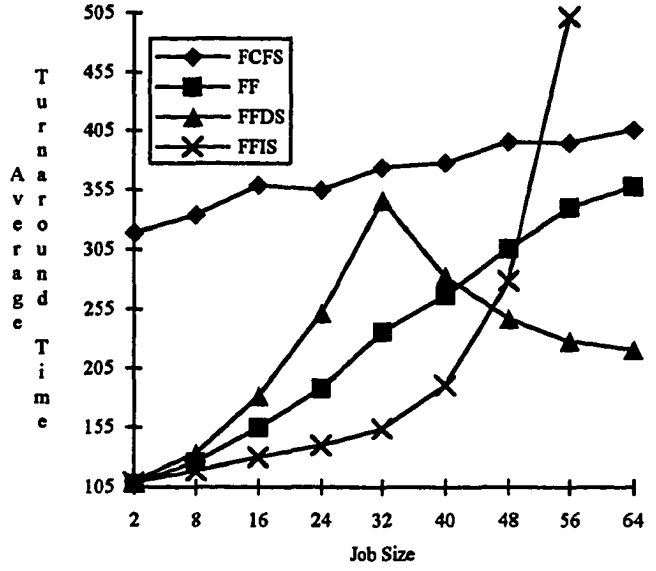

Figure 2.16: Average turnaround time as a function of the job size for the no folding policies, uniform size and execution time distributions, L=0.6

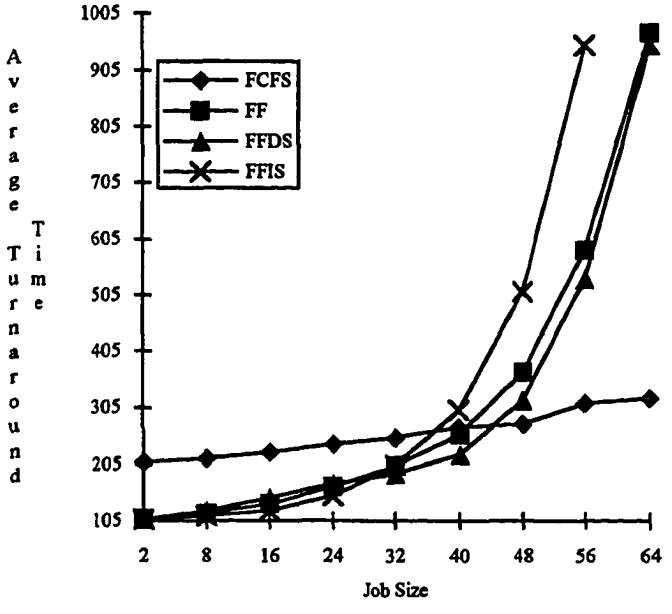

Figure 2.17: Average turnaround time as a function of the job size for the no folding policies, exponential size, uniform execution time, L=0.6

In general, the mean response times under the unlimited folding policies increase significantly with the job size (Figures 2.18-2.20) because larger applications are typically allocated a smaller fraction of the number of processors they requested. Even FFDS+FIFO, which gives priority to larger jobs, and FCFSUF discriminate against large jobs considerably. Under FCFSUF, large jobs have significantly shorter turnaround times than under FFIS+FIFO when the sizes are distributed uniformly and the load is high (Figure 2.19), even though FFIS+FIFO produces the shortest mean response times. However, the fairness curves of FFIS+FIFO and FCFSUF do not differ significantly when the sizes are exponentially distributed (Figure 2.20). Overall, FCFSUF has the best fairness characteristics.

In the fairness figures (Figures 2.16-2.20), the confidence interval is 95% and the maximum relative error is 5% when the uniform job size distribution is used, but they are 90% and 10%, respectively, under the truncated exponential job size distribution.

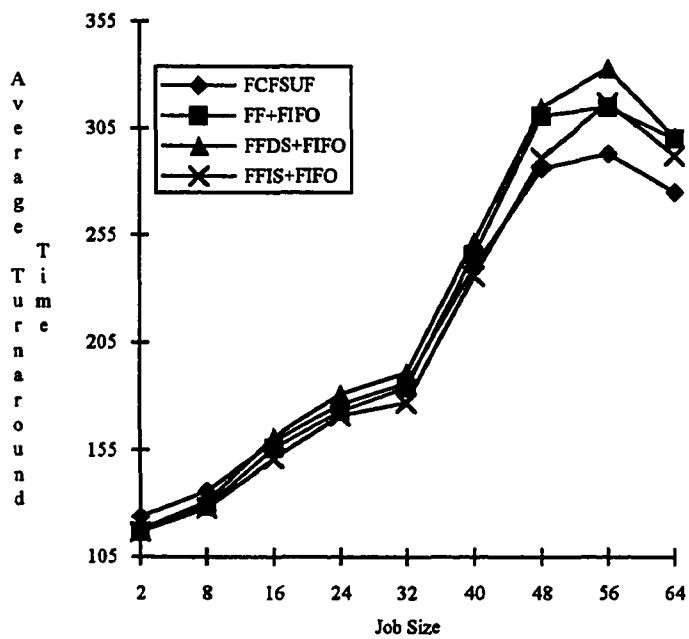

Figure 2.18: Average turnaround time as a function of the job size for the unlimited folding policies, uniform size and execution time distributions, MISP speedup, L=0.6

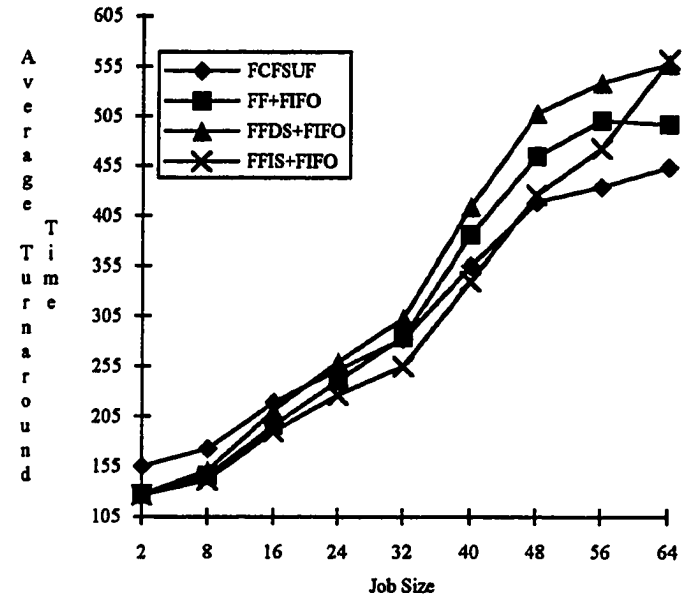

Figure 2.19: Average turnaround time as a function of the job size for the unlimited folding policies, uniform size and execution time distributions, MISP speedup, L=0.9

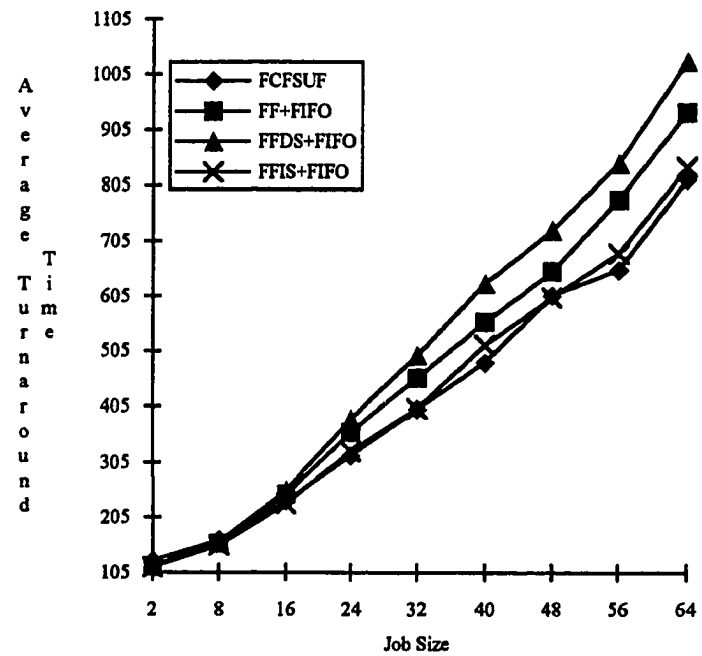

Figure 2.20: Average turnaround time as a function of the job size for the unlimited folding policies, exponential size, uniform execution time, MISP speedup, L=0.9

2.4. Other Policies

The goal of studying these policies is to estimate the improvement in average response times that could be expected when $t(n)$ is known a priori.

First Fit Increasing Total Demand (FFITD): Waiting jobs are sorted in the non-decreasing order of their total processing demand, defined as $n*t(n)$. Job selection and allocation are as in FF. This no folding algorithm produced shorter mean response times than FFIS but performed worse than FF and FFDS, as can be seen in Figure 2.21.

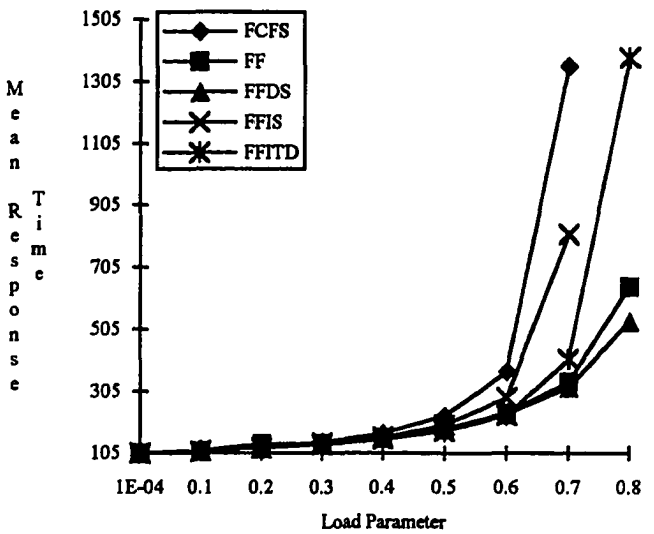

Figure 2.21: Mean response time as a function of the load parameter, uniform size and execution time distributions

Unlimited folding Smallest Total Demand First (STDFUF): Waiting applications are sorted in the non-decreasing order of their total processing demand. The head of the queue waits until there is one or more free processor. When serviced, it is allocated min(n,FP) processors. A new job is allocated min(n,FP) processors if FP>0. It waits if FP=0.

Unlimited folding Shortest Job First (SHJFUF): The waiting jobs are sorted in the non-decreasing

order of the values of $t(n)$, and allocation is as in STDFUF.

Unlimited folding Longest Job First (LOJFUF): Waiting jobs are sorted in the non-increasing order of the values of $t(n)$, and allocation is as in STDFUF.

The mean response times of STDFUF, SHJFUF, and LOJFUF are compared to those of FCFSUF in Figures 2.22-2.24. In Figure 2.22, the execution times are modeled by a truncated exponential distribution with a mean of 10 and an interval of [1,100]. LOJFUF performed poorly under heavy loads and MISP speedup, but SHJFUF and STDFUF outperformed FCFSUF. These results show that: (1) a reduction in mean response times can be achieved under heavy loads when shorter jobs are given priority, and (2) giving priority to longer jobs can lengthen the mean response times significantly. Note that the reduction in mean response times associated with SHJFUF and STDFUF is smaller than their degradation associated with LOJFUF in Figures 2.22-2.24. If the execution times are not estimated adequately performance can degrade, as suggested by the poor mean response times of

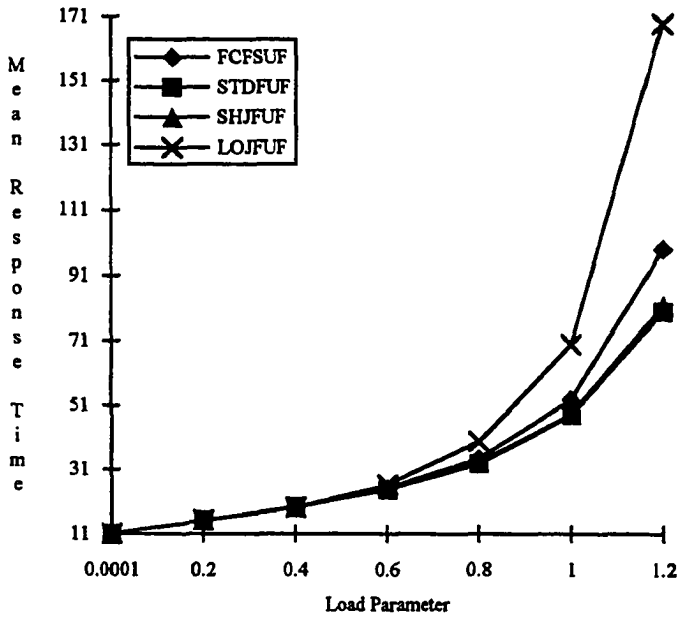

Figure 2.22 : Mean response time as a function of the load parameter, MISP speedup, uniform size distribution, exponential execution time distribution

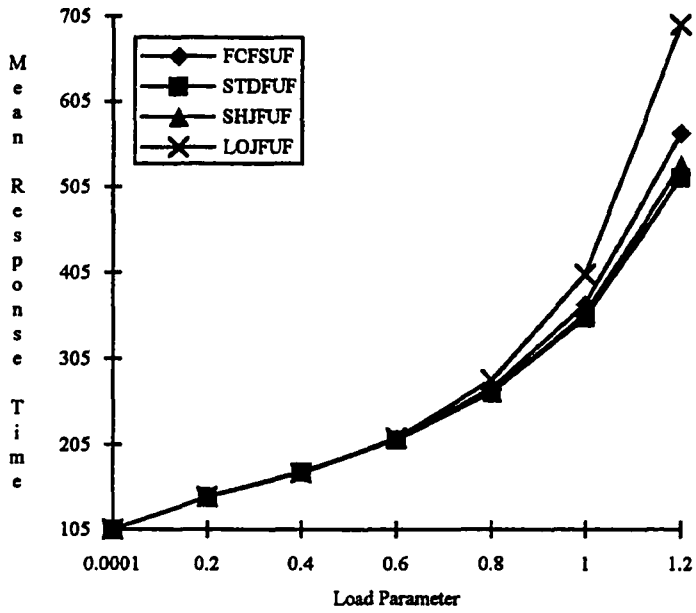

Figure 2.23 : Mean response time as a function of the load parameter, MISP speedup, uniform size and execution time distributions

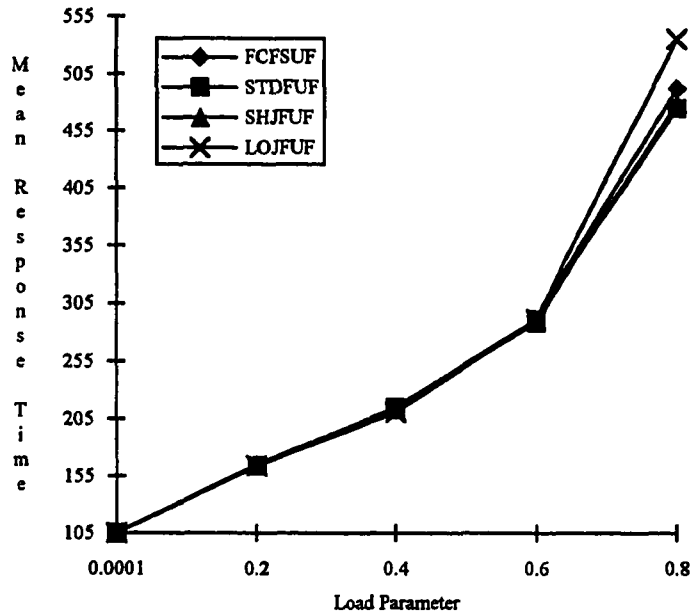

Figure 2.24 : Mean response time as a function of the load parameter, linear speedup, uniform size and execution time distributions

LOJFUF. Moreover, the degradation can be severe under very heavy loads, as indicated by the results displayed in Figure 2.22.

2.5 Conclusions

The efficiency of a parallel application typically increases significantly when the number of processors it is allocated decreases, and it is often worse than that predicted by speedup curves of the MISP type when n is high. Consequently, unlimited folding is expected to be significantly superior to no folding under high system loads. However, it is expected to be inferior to no folding under moderate loads because of the high processor fragmentation it can produce.

When the execution times of applications are not known a priori, FCFSUF is the best policy considered in this study under heavy loads, but FFDS and FF are superior under moderate loads. When compared to FCFSUF, FFIS+FIFO can produce slightly shorter average response times, but it can discriminate against large jobs significantly more than FCFSUF under heavy loads. If the values of $t(n)$ are known, giving priority to shorter jobs can improve performance. However, If these values are estimated poorly performance can degrade significantly, as indicated by the poor mean response times of LOJFUF.

FF+FIFO outperformed FF in [Ghosal 91]. However, the results of this study show that FF (and FFDS) can outperform FF+FIFO. FF+FIFO suffers high processor fragmentation under most load levels, but it can outperform FF under high loads for two reasons: (1) it exploits the increase in efficiency that commonly results from folding, and (2) fragmentation is less severe when the load is high.

When a job completes, the unlimited folding EPFP algorithm services the maximum number of waiting applications. This eager servicing of requests in static space sharing, recommended by Zahorjan and McCann [Zahorjan 90], can lead to poor performance. EPFP is the worst unlimited folding policy studied in this chapter.

The results of this study indicate that increasing the degree to which a job is folded with the system load should be superior to no folding and unconstrained folding. Such a strategy should produce superior system utilization, while exploiting the efficiency advantage of folding.

3 ADAPTIVE STATIC SPACE SHARING POLICIES

No folding static space sharing job scheduling disciplines do not take advantage of the increase in efficiency that typically results when a job is allocated fewer processors, and they can induce high processor fragmentation because applications wait until they can be allocated the number of processors they requested. Their mean response times start increasing sharply at load levels significantly smaller than one. The level is approximately 0.6 for FCFS and FFIS and 0.7 for FF and FFDS in Figures 2.9 and 2.10. The unlimited folding policies exploit the efficiency advantage of folding, but they can suffer considerably more overall fragmentation under most load levels because of folding fragmentation. The fragmentation problem is not severe under very heavy loads when released processors are likely to be allocated immediately or soon after their release.

As low processor fragmentation is achieved without folding under moderate loads but with unlimited folding under very high loads, superior performance should result if the maximum degree to which jobs are folded increases with the load. In this chapter, adaptive folding is investigated with the goal of reducing fragmentation and mean response times in topology-independent static space sharing. Several program-based algorithms that implement this strategy but differ in the folding method and the job selection criteria are studied. Detailed simulation is used to compare them to first-fit and FF+FIFO. The results show that adaptive folding offers substantial performance advantage over no folding and unlimited folding. It can produce higher and more stable system utilization and significantly shorter mean turnaround times. Moreover, the performance of the best algorithm proposed (Multifolding First Fit) is only slightly sensitive to the order in which applications are serviced. Its performance did not improve significantly when the shortest job, the job with the smallest processor request, and the job with the smallest total processing demand were given priority.

3.1 Introduction

The method used in determining the number of processors to allocate to parallel applications strongly influences system performance under static space sharing, as seen in Chapter 2. It has a strong effect on fragmentation and application efficiency.

The target system is subdivided into partitions of equal size and an application is permanently assigned to a single partition in several recent studies of static space sharing policies [Ghosal 91][Naik 93b][Setia 93]. The results show that performance improves significantly if the size of the partitions decreases when the system load increases under both uniprogrammed [Ghosal 91][Naik 93b] and multiprogrammed equipartitioning [Setia 93]. Several partition sizes were considered. However, no algorithm that determines the size as function of the system load is used or proposed in these studies. Moreover, the issue of how to dynamically change it, while maintaining equipartitioning, in response to load changes in these *static* allocation strategies is not dealt with.

Decreasing the size of the partitions when the load increases has several benefits. By allocating a large number of processors to applications under moderate loads, reasonable system utilization and mean response times can be achieved. Applications that request a large number of processors are allocated a high fraction of their request, and they do not take too long to complete. However, because small applications are also allocated large partitions under moderate loads internal processor fragmentation can be high. When the partitions are smaller, under heavier loads, system utilization is not likely to suffer significantly because there are more jobs in the system (i.e., more partitions can be allocated). Moreover, reducing the partitions' size reduces internal fragmentation and improves the execution efficiency of more applications. System efficiency may increase monotonically and significantly with the system load when the size of the partitions increases concurrently. Folding is assumed in adaptive partitioning. An application executes on the number of processors in the partition it is allocated, which can be less than the application's processor request.

High external fragmentation can result when a program is not allocated more than one parti-

tion. For example, in a machine with P processors a job that requests P-1 processors is allocated only $P/2$ processors when there are two machine partitions, even when the second partition is free and there are no other pending allocation requests. Internal and external fragmentation depend on the distribution of processor requests. However, all applications request P processors in [Setia 93], and a small set of five applications that request 1,2,4,8, and P=16 processors is used in [Ghosal 91].

Topology-independent program-based partitioning avoids the internal fragmentation problem and can reduce external fragmentation because applications may be allocated the exact number of processors they request. It is again assumed that a new application requests a number of processors, n, from the allocation algorithm.

In addition to the no folding and unlimited folding methods for determining the number of processors to allocate to applications in program-based static space sharing, limited folding can be used. Folding is limited if an application waits until it can receive at least some number, not necessarily fixed, of processors. Recall that folding is unlimited when the lower limit is one processor.

In [Sevcik 89], it is shown that limited folding can improve performance significantly under heavy loads. In the workload model used, detailed knowledge of application parallelism is assumed and the overhead of parallel execution is due to load imbalances within applications. The speedup of a job is linear when the variance of its parallelism is zero. Otherwise, the speedup is sublinear. An application is not folded when its parallelism variance is zero, but it is folded by a factor that is an increasing function of the variance and system load when the variance is greater than zero. An issue with this study is the assumption that parallelism characteristics are known in detail. The results in this dissertation are different in that they show that limited folding is also superior to no folding when the applications have linear speedup.

FF, FCFS, and a limited folding policy that uses a fixed maximum folding factor were compared in [Abraham 92]. A parallel task waits until it can be allocated at least 1/3 or 1/4 of the number of processors it requested in the limited folding policy. These values produced good mean response

times. FF outperformed FCFS, but the limited folding policy yielded the shortest mean response times.

In the limited folding scheme proposed and compared to equipartitioning in [Naik 93b], a job waits until it can receive at least the minimum of its processor request and a fixed number of processors. When a job terminates, released processors are divided as evenly as possible among waiting applications under this constraint. The fixed number of processors used (32 processors) is not explained. It presumably gave good mean response times under the system and workload models used. The scheme outperformed fixed equipartitioning across a broad range of system loads, including when the best partition size considered in the study is used.

An issue with the last two limited folding policies is whether the maximum folding factor, FF_{max}, should be fixed. Assuming linear speedup, for example, and a policy that does not support folding, an application that must wait is not expected to start execution before $r*T_e$ time units. Therefore, it can be folded by $1+r$ without changing its expected completion time. The quantities T_e and $r*T_e$ are the expected execution time and residual lifetime of an application that is allocated the number of processors it requested. The application can be folded by $1+r*f_m$ if it must wait for a job that is folded by f_m. Thus, the folding factor should increase with the load and depend on the distribution of the execution times. For example, $r=1$ for the exponential distribution, $r\approx 2/3$ for the uniform distribution over [Tmin,Tmax] when Tmin/Tmax is small, and $r=1/2$ for constant execution times. In general, $1/2 \le r \le 1$. Folding therefore should be greater when the execution times are distributed exponentially than when they are distributed uniformly, for example. As the series $f_{m+1}=1+r*f_m$ converges to $1/(1-r)$ when r is less than one, a fixed maximum folding factor can be expected to produce good performance under high loads and a specific execution times distribution.

The expected residual lifetime of a job folded by the same factor is shorter under MISP speedup than under linear speedup because of the increase in efficiency that results from folding. However, to estimate the factor by which an application can be folded without changing its expected completion time the application's execution times as function of the number of allocated processors should

be known. Nonetheless, the best folding factor depends on the load level and the efficiency of applications under MISP speedup. The improvement in the performance of equipartitioning when the number of partitions increases with the system load [Ghosal 91][Naik 93b][Setia 93] demonstrates that folding should increase with this parameter. Moreover, the comparison of no folding and unlimited folding in Chapter 2 and the results in [Sevcik 89] show that folding should increase with the workload inefficiency and the load.

3.2 Allocation Policies

When job execution times as function of processor allocation are not known, as it is assumed in this research, the best folding factor can not be determined. Consequently, two methods, based on first-order statistics, are used to compute the maximum folding factor, FF_{max}. The first FF_{max}, $f1_{max}$, is determined as follows. A worst-case estimate of the time needed to finish the jobs currently in the system is $\lceil P_d/(S_e*P) \rceil * T_e$ time units (P_d is the total processor demand of the jobs). Assuming high scheduling effectiveness ($S_e \approx 1$), an application is expected to complete within this time if it is folded by at most $f1_{max}=\lceil P_d/P \rceil$. The assumption that $S_e \approx 1$ is validated by the simulation results. In this method, the completed fraction of the executing jobs is ignored.

The second maximum folding factor, $f2_{max}$, is equal to $f1_{max}-(1-r)N_r/P_u$, where N_r is the total number of processors requested by the executing applications and P_u is the number of busy processors. The ratio N_r/P_u is the mean current folding factor, and $(1-r)N_r/P_u$ corresponds to an estimate of the completed fraction of the active applications.

The following limited folding policies are studied:

Folding FCFS (FFCFS): The head of the FIFO waiting queue or a job that arrives while the queue is empty is allocated $\min(FP,n)$ free processors if their number $FP \geq \lceil n/FF_{max} \rceil$. When $FF_{max}=1$, the traditional FCFS results. The unlimited folding variant is obtained when an application can be allocated a single processor.

Folding First Fit (FFF): A new job waits in a FIFO queue until it can receive at least $\lceil n/FF_{max} \rceil$ processors. When selected for service, a job is allocated $\min(FP,n)$ processors.

Folding Smallest Job First (FSJF): This is a variant of the last algorithm. It differs in that waiting jobs are sorted in the non-decreasing order of their processor demand. The goal is to improve the mean response times by executing more simultaneously.

Multifolding First Fit (MFFF): The waiting queue is FIFO. During an allocation scan of the queue, a job is selected for service if $(x+n)/FP \leq FF_{max}$, where x is the total processor requirement of the jobs selected so far. At the end of a scan, the selected jobs are folded by $FF_{act}=x/FP$. Each is allocated approximately (due to arithmetic errors) n/FF_{act} processors. Multifolding algorithms differ from the remaining algorithms in that they may fold multiple applications per allocation scan.

Multifolding Smallest Job First (MFSJF): This is a variant of the last algorithm where waiting requests are sorted in the non-decreasing order of their processor requirement. The goal is again to improve the mean response times by executing more jobs simultaneously.

When FF_{max} is load-dependent, there is an allocation scan per job arrival or departure.

3.3 Results

The allocation policies are compared using mean response times and scheduling effectiveness, and fairness curves.

3.3.1 Simulation Parameters

The target machine consists of P=64 identical processors, and $2 \leq n \leq P$. The uniform and the truncated exponential distributions are used to model the job processor requests (i.e., the values of n) and the execution times on n processors. When the requests are modeled by the truncated exponential distribution, a mean of 15 is used. The interval of the execution times on the requested number of processors is [10,200] when they are distributed uniformly. It is [1,100], the mean is 10, and the resulting

true mean is approximately 11 under the truncated exponential execution times distribution. A pseudo-random variable distributed uniformly over [0.4,0.9] models the initial job efficiency values under MISP speedup. However, the values that correspond to a serial fraction greater than 0.5 are ignored.

The simulator generates 8500 applications per run. To ignore startup effects, the performance data of the first 500 jobs is discarded. The number of runs is such that the mean response times obtained have relative errors not exceeding 5% under the 95% confidence interval. The scheduling effectiveness values have relative errors below 1% (they are typically much less than 1%) under the same confidence interval.

The results shown are for $FF_{max}=f_{1max}$. The second folding factor f_{2max} did not change the mean response times and scheduling effectiveness values significantly, and is less practical because it depends on the distribution of the execution times through r.

3.3.2 Scheduling Effectiveness and Mean Response Times

3.3.2.1 No folding, unlimited folding, and limited folding

Adaptive limited folding is superior to no folding and unlimited folding. It produced higher system utilization (scheduling effectiveness) and shorter mean response times under the processor requests, execution times, and speedup assumptions considered in this study. For example, the effectiveness curves of the three first-fit variants FF, FF+FIFO, and FFF are compared in Figures 3.1 and 3.2 under the uniform execution time distribution and MISP speedup. FFF yielded more stable and higher effectiveness. The performance advantage of FFF is less significant in Figure 3.2 because the requests are distributed exponentially. Most jobs are small then, and their number, under the same system load, is higher than under the uniform distribution. Both factors reduce processor fragmentation in the three policies, but the reduction is higher under FF+FIFO and FF.

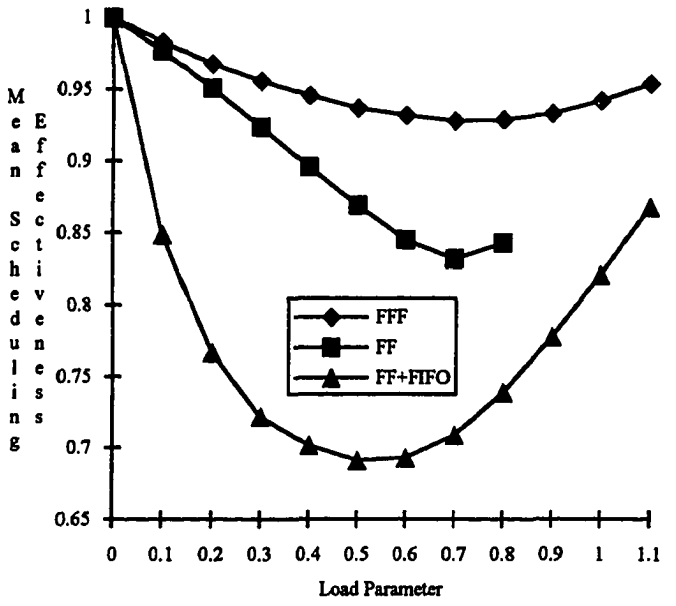

Figure 3.1: Scheduling effectiveness as a function of the load parameter, MISP speedup, uniform size and execution time distributions

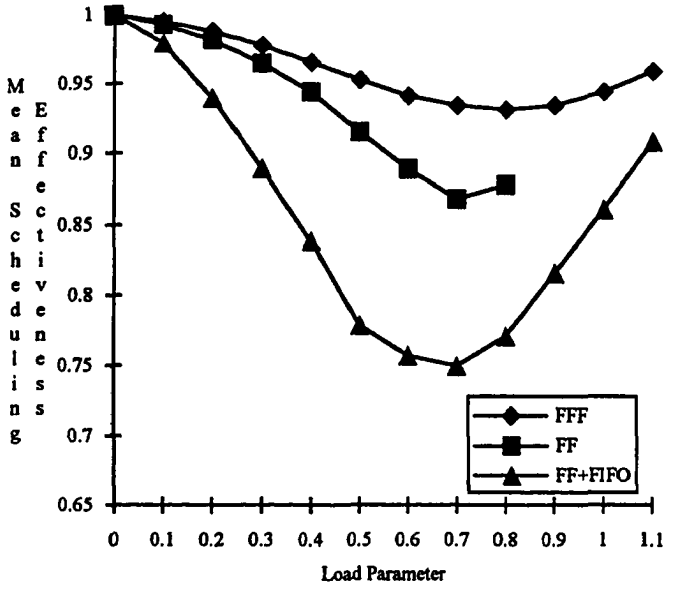

Figure 3.2: Scheduling effectiveness as a function of the load parameter, MISP speedup, exponential size distribution, uniform execution time distribution

The mean response times of the three first-fit variants when the job sizes and execution times are uniformly distributed are compared in Figures 3.3 and 3.4. FFF yielded shorter mean response times than FF and FF+FIFO across all load levels. In Figure 3.3, where MISP speedup is assumed, the mean response times are 30-40% longer under medium to high load levels when FF+FIFO is compared to FFF. The degradation is approximately 30% under L=0.4 and L=1, 40% under L=0.6 and L=0.8, and 20% under L=0.2 and L=1.2. The advantage of limited folding over unlimited folding is less significant under very high load levels because processors are unlikely to remain idle then, even if folding is unconstrained. In Figure 3.4, where linear speedup is assumed, the degradation is more significant. It is approximately 40% under L=0.2, 60% under L=0.5, 70% under L=0.7, and 50% under L=0.9. FFF outperformed FF and FF+FIFO under both linear and sublinear speedup curves.

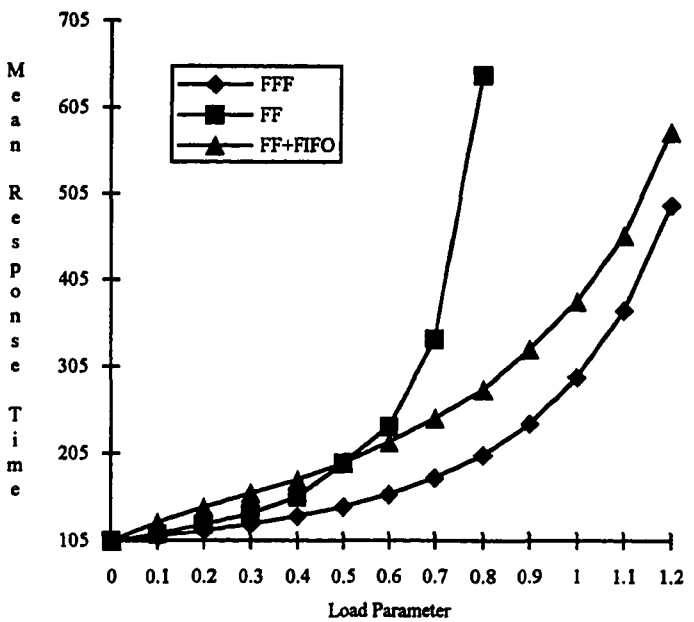

Figure 3.3: Mean response time as a function of the load parameter, MISP speedup, uniform size and execution time distributions

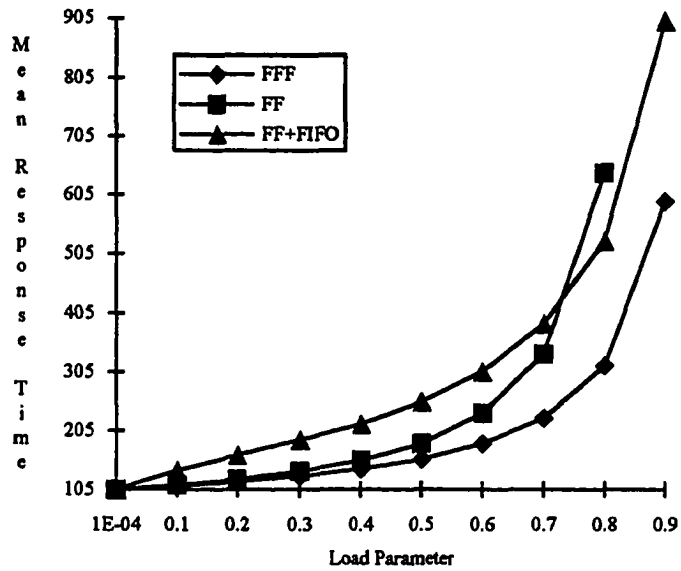

Figure 3.4: Mean response time as a function of the load parameter, linear speedup, uniform size and execution time distributions

The advantage of FFF over FF+FIFO is more moderate in Figure 3.5 than in Figure 3.3 because most jobs are smaller and their number is higher under the same load level (the processor requests are modeled by the truncated exponential distribution). The degradation in performance produced by FF+FIFO in Figure 3.5 is approximately 10% under L=0.4, 20% under L=0.6, 30% under L=0.8, and 15% under L=1.2.

Because FFF is more effective than FF, it saturates under higher loads, including when linear speedup is assumed (e.g., Figures 3.3-3.5). The saturation loads are much higher under MISP speedup because of the increase in efficiency that results from folding. The mean response time performance advantage of FFF over FF is very high under heavy loads. For example, the performance of FF under L=0.8 is worse than that of FFF under L=1.2 in Figure 3.3, and the mean response time of FF is approximately double that of FFF under L=0.7 in Figure 3.3 and under L=0.8 in Figure 3.4.

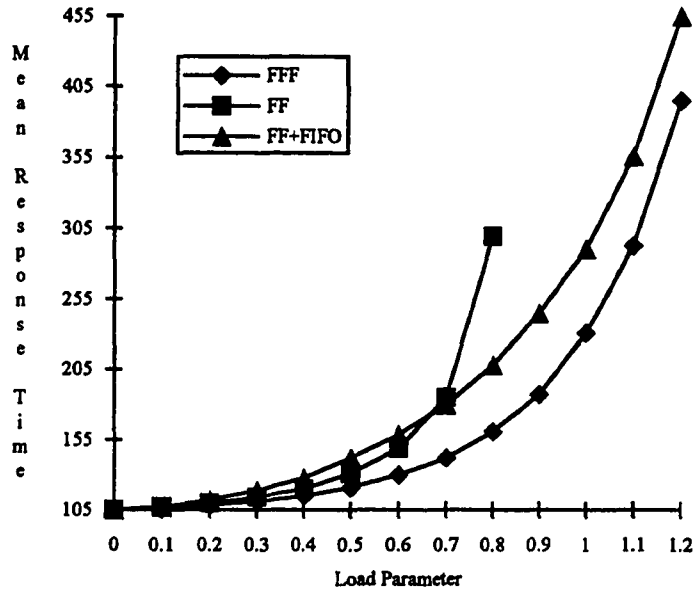

Figure 3.5: Mean response time as a function of the load parameter, MISP speedup, exponential size distribution, uniform execution time distribution

Even though FF does not exploit the folding efficiency improvement factor, it is superior to FF+FIFO under low to medium system loads and MISP speedup because of its ability to utilize more processors. When linear speedup is assumed, FF is better than FF+FIFO under most or all system loads, as seen in the preceding chapter.

3.3.2.2 Limited folding policies

The mean response times of the five limited-folding policies are compared in Figures 3.6-3.9 under MISP speedup and the four combinations of the execution time and size distributions considered. These results show that:

1) FFF and FFCFS are inferior to MFFF, FSJF, and MFSJF.

2) The mean response times produced by MFFF, FSJF, and MFSJF do not differ significantly.

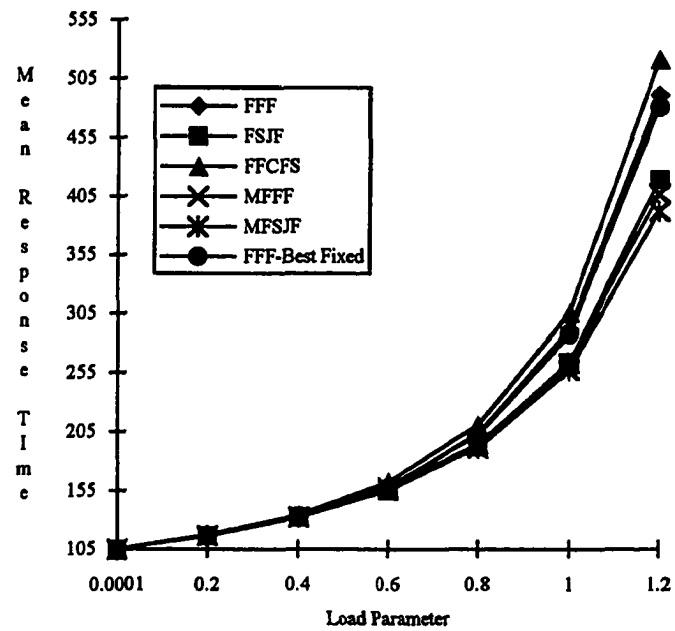

Figure 3.6: Mean response time as a function of the load parameter, MISP speedup, uniform execution time and size distributions

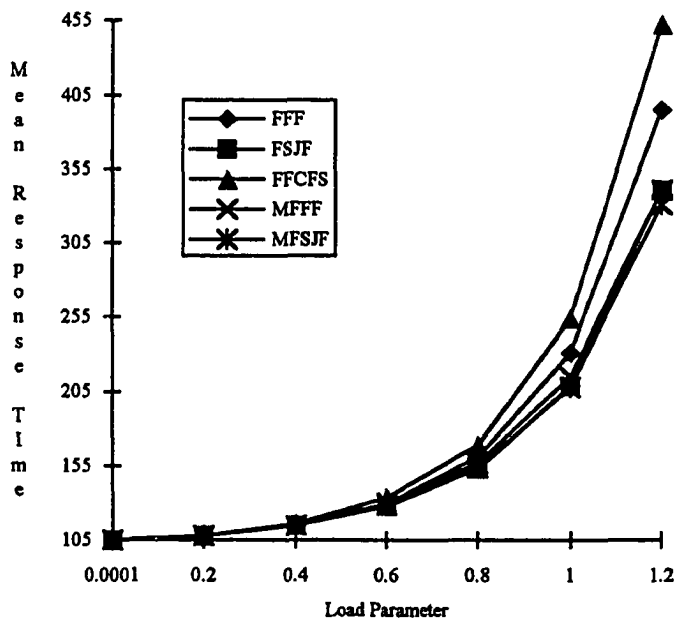

Figure 3.7: Mean response time as a function of the load parameter, MISP speedup, uniform execution time distribution, exponential size distribution

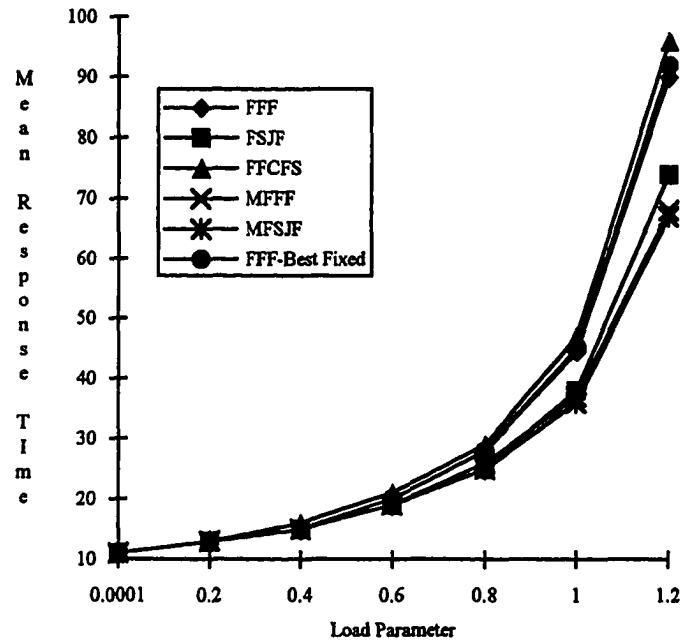

Figure 3.8: Mean response time as a function of the load parameter, MISP speedup, exponential time distribution, uniform size distribution

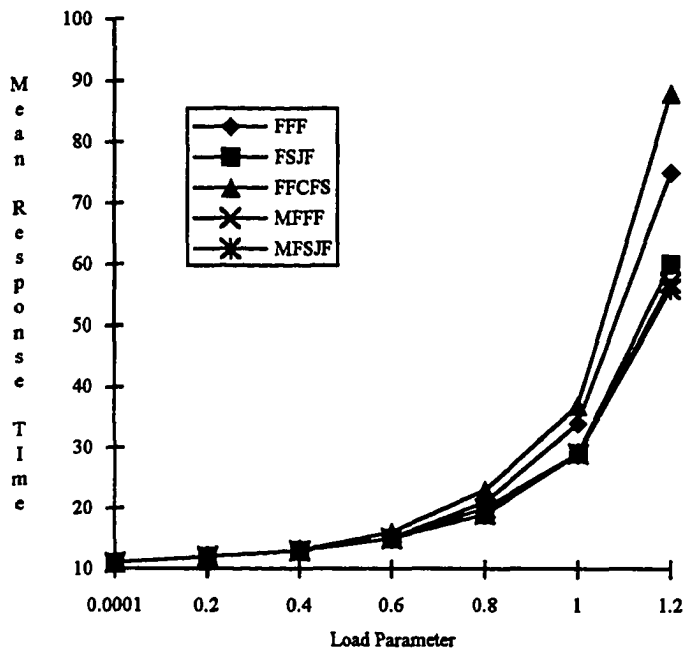

Figure 3.9: Mean response time as a function of the load parameter, MISP speedup, exponential execution time and size distributions

3) FFCFS performs only slightly worse than FFF because the application at the head of the waiting queue no longer waits until the exact number of processors it requested is available.

As FFF is not the best limited folding policy, the advantage of adaptive limited folding over no folding and unlimited folding is higher than shown in Figures 3.3 and 3.5.

When linear speedup is assumed, FFF can produce shorter mean response times than MFFF and MFSJF (e.g., Figure 3.10). However, the speedup of parallel applications is typically sublinear, and it is often worse than that predicted by an MISP speedup curve, especially when n is high. As a result, MFFF is expected to outperform FFF significantly in practice.

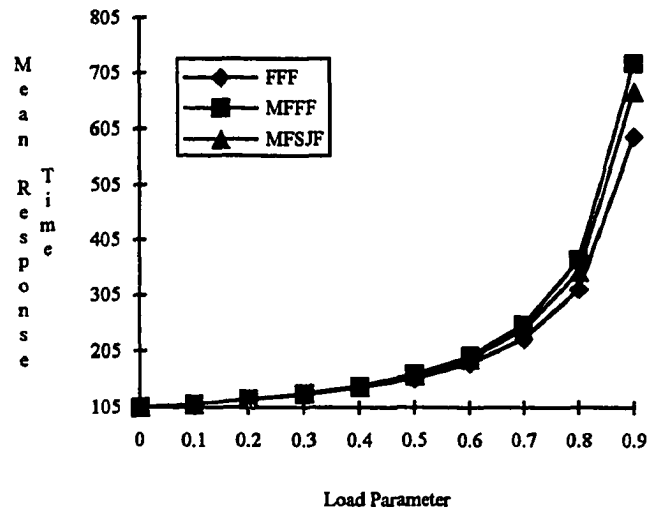

Figure 3.10: Mean response time as a function of the load parameter, linear speedup, uniform size and execution time distributions

MFFF produced lower scheduling effectiveness values than FFF (e.g., Figures 3.11 and 3.12), however it resulted in shorter response times under MISP speedup because it takes more advantage of the increase in efficiency that results from folding. The mean actual folding factors of MFFF are larger than those of FFF, as can be seen in Figures 3.13 and 3.14, because MFFF can fold multiple applica-

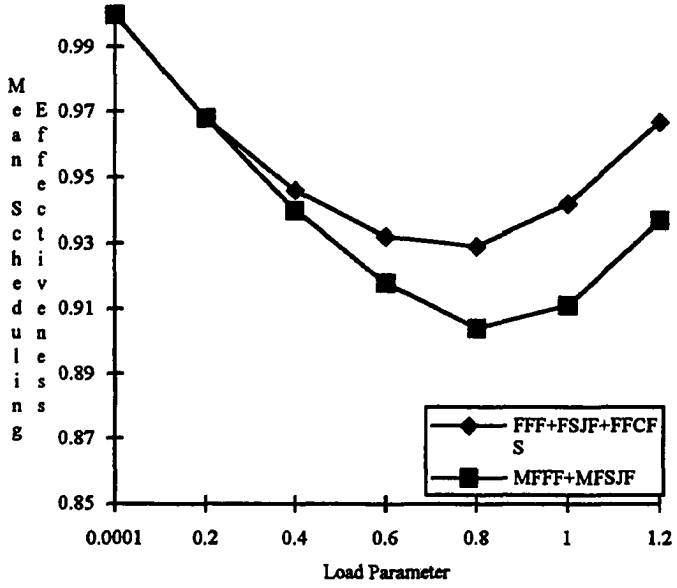

Figure 3.11: Mean scheduling effectiveness as a function of the load parameter, MISP speedup, uniform execution time and size distributions

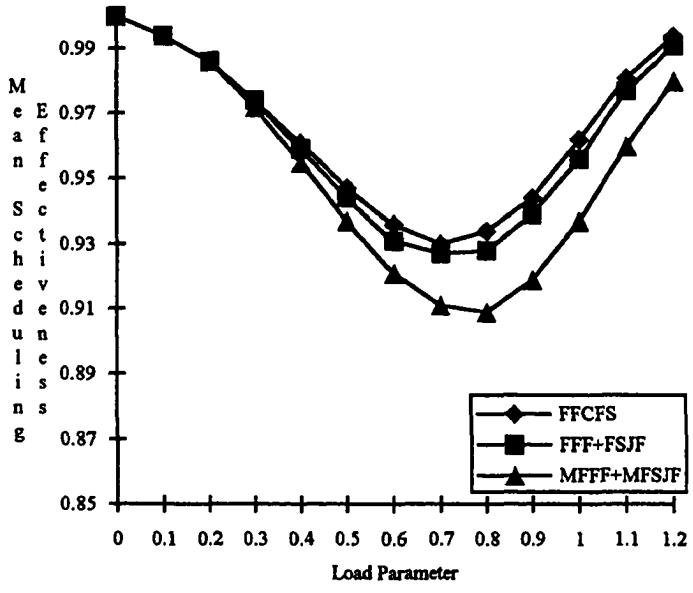

Figure 3.12: Mean scheduling effectiveness as a function of the load parameter, MISP speedup, exponential execution time and size distributions

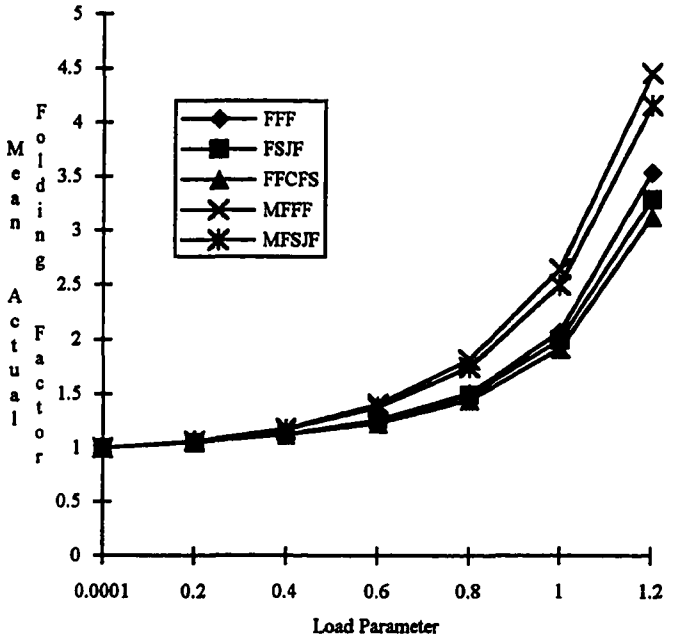

Figure 3.13: Mean actual folding factor as a function of the load parameter, MISP speedup, uniform size and execution time distributions

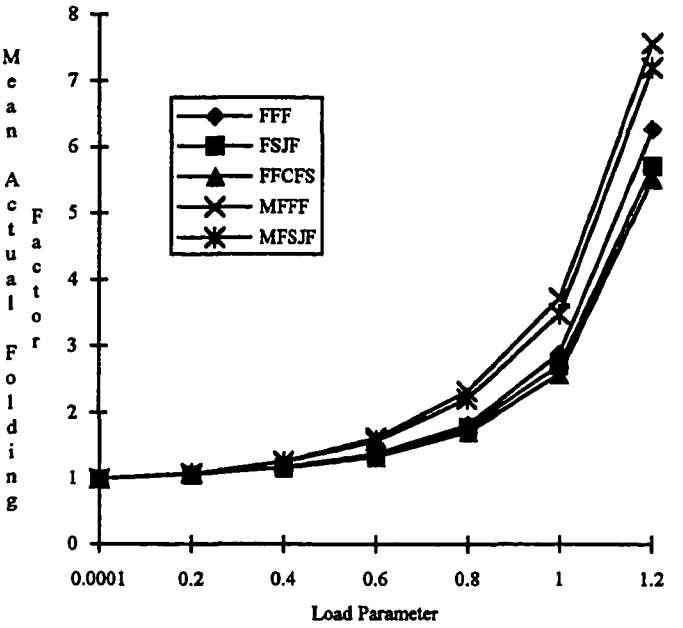

Figure 3.14: Mean actual folding factor as a function of the load parameter, MISP speedup, exponential time distribution, uniform size distribution

tions per allocation scan. These figures also show that the mean actual folding factors are larger when the execution times are distributed exponentially. FFF outperformed MFFF under linear speedup because it is more effective.

The mean scheduling effectiveness values are suboptimal because of processor fragmentation. Fragmentation is low under light loads because applications seldom have to wait, most applications are not folded, and the factor by which an application is folded is small on average. It is also low under high loads because released processors are likely to be allocated immediately or soon after their release. The number of waiting applications and the probability that a job will arrive soon increase with the system load.

FFF was also simulated under fixed maximum folding factors. The results of these experiments (e.g., Figures 3.15 and 3.16) also show that folding has a very strong influence on performance, and the best fixed FF_{max} increases with the load and depends on workload characteristics. Excessive mean response times can result under high to very high loads if FF_{max} is too small, and the best fixed maximum folding factor is larger when the execution times are distributed exponentially than when their distribution is uniform.

These results confirm that the advantage of limited folding over unlimited folding depends on the load and the distribution of the execution times. The mean response times increase relatively more under L=0.8 than under L=1.2 when FF_{max} increases from its best value to P in Figures 3.15 and 3.16. As folding should be lower under the uniform execution times distribution, the relative increase is higher in Figure 3.15 than in Figure 3.16.

A comparison of the mean response times of the adaptive FFF and FFF that uses the best fixed FF_{max} (FFF-Best Fixed) under MISP speedup and the uniform job sizes distribution shows that they do not differ significantly under this workload model (Figures 3.6 and 3.8).

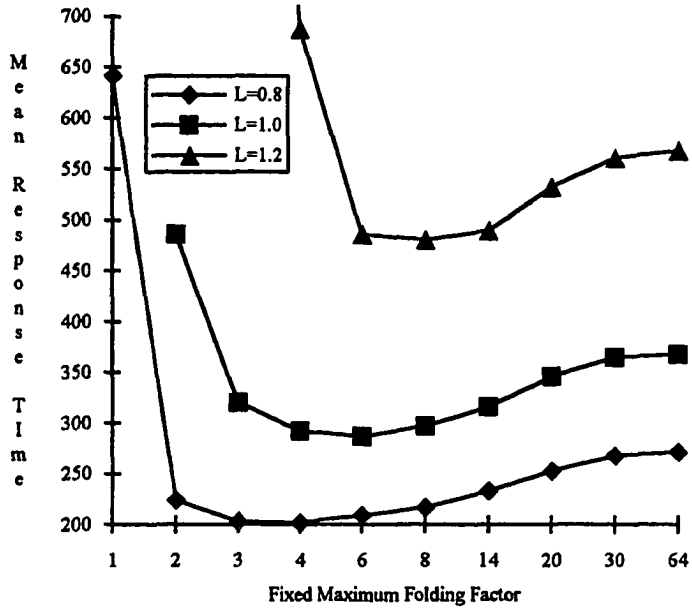

Figure 3.15: Mean response time as a function of the fixed maximum folding factor for FFF, MISP speedup, uniform size and execution time distributions

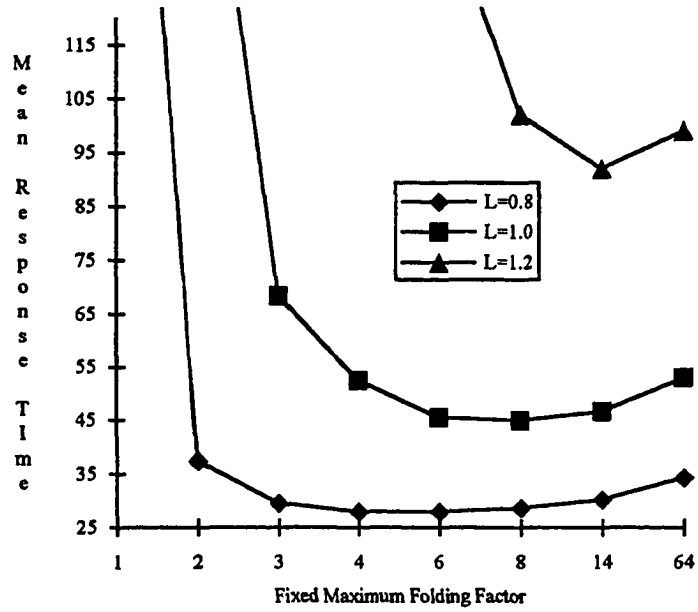

Figure 3.16: Mean response time as a function of the fixed maximum folding factor for FFF, MISP speedup, uniform size distribution, exponential execution time distribution

3.3.2 Fairness

Sets of curves are used to characterize the fairness of the scheduling policies. Each contains the average turnaround times as a function of the job processor demand under a load level of interest (the processor demand is chosen because it is the sole job characteristic used by the algorithms). FSJF causes large jobs to have excessive response times under moderate to high loads, as can be seen in Figures 3.17 and 3.18. Moreover, its mean response time performance under MISP speedup is slightly worse than that of MFFF. MFSJF offers insignificant mean response time performance advantage over MFFF, and it has worse fairness characteristics. FFCFS discriminates against large jobs less than MFFF, but its mean response times are longer for all job sizes. Based on the three performance parameters considered, MFFF is more promising than FFF, FFCFS, FSJF, and MFSJF.

Note that the policies favor small jobs because they are likely to have shorter waiting and execution times. It is easier to find enough processors for allocation to these jobs, and they are likely to be

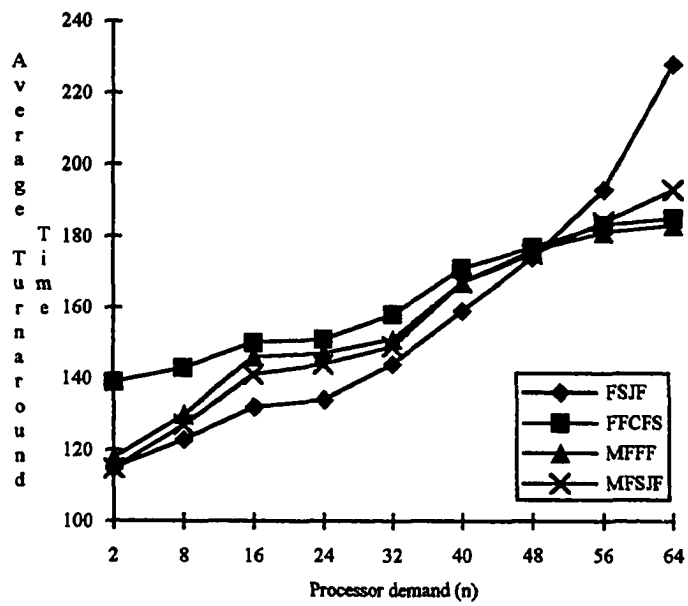

Figure 3.17: Mean turnaround time as a function of the processor demand, L=0.6, MISP speedup, uniform size and execution time distributions

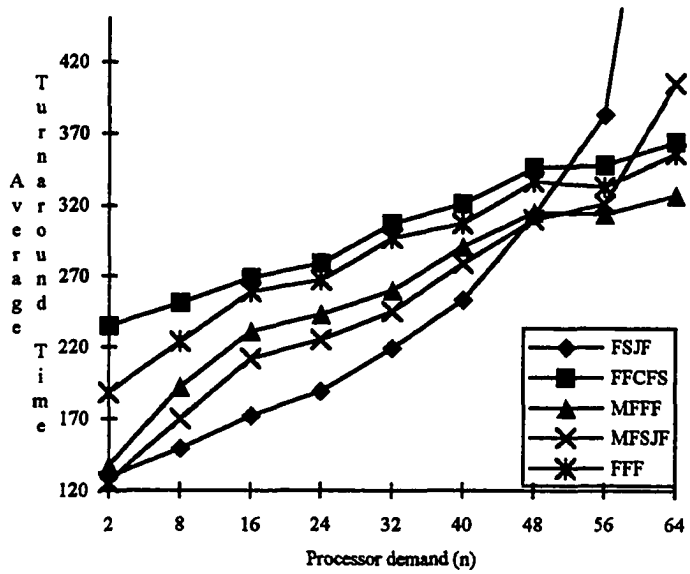

Figure 3.18: Mean turnaround time as a function of the processor demand, L=1, MISP speedup, uniform size and execution time distributions

allocated a larger fraction of their processor demand. A comparison of the fairness curves of MFFF to those of the no folding and unlimited folding policies studied in Chapter 2 shows that they have better fairness characteristics.

3.4 Other Policies

The goal of studying the following additional multifolding policies is to estimate the improvement in mean response times that may be expected when $t(n)$ is known a priori. They differ from MFFF in the sorting order of waiting applications.

Multifolding Smallest Total Demand First (MFSTDF): Waiting applications are sorted in the non-decreasing order of their total processing demand, defined as $n*t(n)$, and allocation is as in MFFF.

Multifolding Shortest Job First (MFSHJF): The waiting jobs are sorted in the non-decreasing order of the values of $t(n)$, and allocation is as in MFFF.

Multifolding Longest Job First (MFLOJF): The waiting jobs are sorted in the non-increasing order of the values of $t(n)$, and allocation is as in MFFF.

The mean response times of these policies were compared to those of MFFF under MISP speedup. MFSTDF and MFSHJF outperformed MFFF, but MFLOJF produced longer mean response times. However, the performance differences were very small, as can be seen in Figures 3.19 and 3.20, and they were statistically insignificant. Increasing the upper bound of the execution times interval increased the performance advantage of MFSTDF and MFSHJF, but the improvement remained statistically insignificant when the bound was increased to 1000. MFFF is a robust policy in that its performance is not expected to change significantly if smaller, shorter, longer, or jobs with smaller total processing demands are given priority.

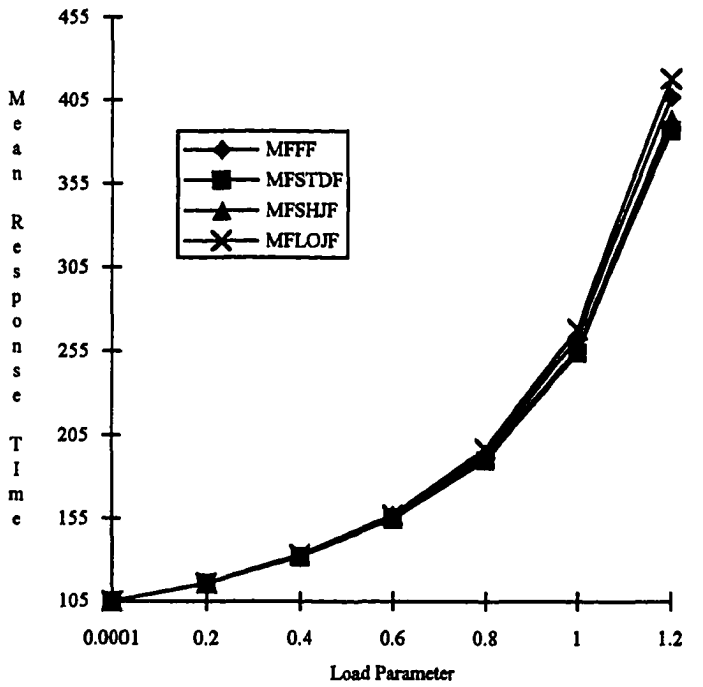

Figure 3.19: Mean response time as a function of the load parameter, MISP speedup, uniform execution time and size distributions

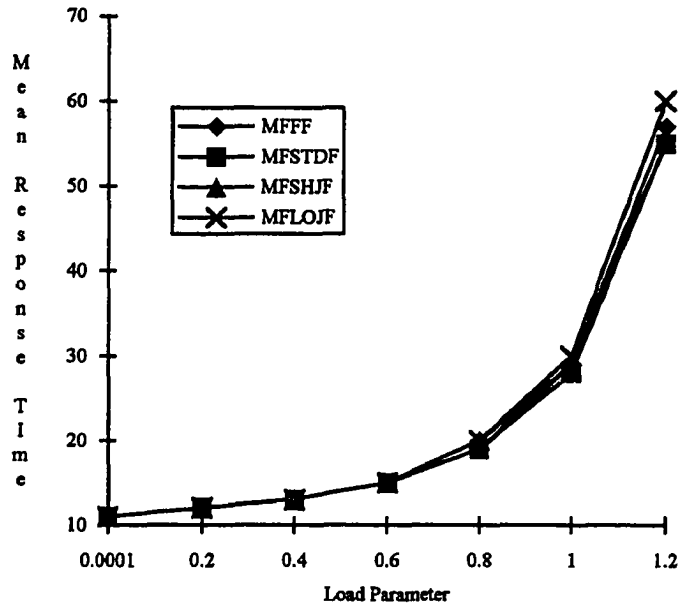

Figure 3.20: Mean response time as a function of the load parameter, MISP speedup, exponential execution time and size distributions

3.5 Conclusions

Adaptive folding of parallel jobs can substantially improve the performance of parallel systems. The efficiency of a parallel application is normally a decreasing function of the number of processors allocated, and it is often worse than that predicted by an MISP speedup curve, especially when the physical parallelism exploited is high. For typical applications, and based on the fairness curves and mean response times, MFFF, when $FF_{max}=f1_{max}$, is the best policy studied in this chapter.

A fixed maximum folding factor can produce good performance under specific system loads and workload characteristics. However, it can degrade performance if it is too small or too large. MFFF is a robust policy. It resulted in good performance across the system loads and workload characteristics considered in this study, and its performance did not improve significantly when shorter applications and applications with smaller total processing demands were given priority.

MFFF is superior to FF, FF+FIFO, and the remaining policies studied in Chapter 2, including when linear speedup is assumed. This conclusion contradicts the assertion in [Sevcik 89] that no folding is optimal under linear speedup (constant degree of parallelism in Sevcik's study). The benefits of folding can be substantial under linear speedup (e.g., Figures 3.4) because it can reduce processor fragmentation considerably.

A fundamental problem with static space sharing is folding fragmentation, which exists because released processors *are not* allocated to folded jobs. Dynamic space sharing solves this problem. However, it induces application reconfiguration overhead. MFFF is compared to dynamic space sharing policies in the next chapter.

4 DYNAMIC SPACE SHARING POLICIES

In this chapter, several program-based topology-independent dynamic space sharing policies are studied and compared. The results of a detailed simulation study of their performance show that the policies that reduce waiting times by allowing more applications to execute simultaneously are superior to those that reduce execution times by restricting the number of active applications. This outcome is due to the significant increase in efficiency that typically results when programs execute on fewer processors. When the applications have linear speedup, the policies that restrict the number of active applications are superior. Also, the results show that there is a tradeoff between mean response times and fairness. Giving priority to jobs with small processor requests can reduce the overall mean response times, but it increases the expected response times of large jobs.

4.1 Introduction

Dynamic space sharing differs from static space sharing in that the number of processors allocated to applications can vary during their execution. However, applications are allocated distinct processor subsets, as in static space sharing. The dynamic strategy avoids the folding fragmentation problem that exists under static space sharing, and it can reduce program idle times due to insufficient parallelism. Folding fragmentation is avoided because released processors can be allocated to applications executing on less than the number of processors requested. To reduce its idle times an application may, for example, request processors as it needs them and release those it no longer uses. A major disadvantage of dynamic space sharing is the overhead induced by processor releases and reallocations, specifically context switches, cache reloads, and data migration in distributed-memory and NUMA systems. This overhead can offset the benefits of dynamic space sharing [Zahorjan 90].

Dynamic space sharing policies produced shorter mean response times than traditional process-

based time-multiplexing schemes in several experimental and simulation studies of job scheduling in UMA [Tucker 89][McCann 93] and NUMA systems [Markatos 93]. The context switches associated with time-multiplexing are avoided, and significant reductions in cache reloads and synchronization delays can result. For example, a process may spend a long time in the suspended state, especially under high system loads, and impede the progress of processes that interact with it under traditional time-multiplexing schemes, which schedule processes independently of their interactions.

To address this synchronization problem, Ousterhout [Ousterhout 82] proposed round-robin coscheduling. Under this scheme, cooperating processes are assigned to distinct processors, and they are dispatched and preempted together so as to avoid waiting for suspended processes. However, co-scheduling incurs the overhead associated with time-multiplexing, and system utilization can be low because a subset of processors idles when it can not execute a complete set of cooperating processes during a time-slice. Dynamic space sharing outperformed round-robin coscheduling in simulation [Zahorjan 90][Leutenegger 90] and experimental [Markatos 93] studies.

The fundamental issue in space sharing is determining the number of processors to allocate to competing jobs. Allocating a small number of processors typically increases program efficiency and decreases the waiting times, but it normally increases the execution times. Consequently, there may be a tradeoff between reducing the running times of individual applications and reducing the overall mean response time.

A few dynamic space sharing schemes have been proposed and evaluated. The number of processes is dynamically controlled so that it does not exceed the number of processors and a processor is dedicated to the execution of a single process in the process control technique proposed for shared-memory machines [Tucker 89]. Time-multiplexing is avoided. In the prototype implemented on Encore Multimax shared-memory multiprocessor, the processors are divided evenly among the jobs under the constraint that no application is allocated more than its processor demand. In comparison to performance under traditional process-based round-robin scheduling, significant reductions in response times

were obtained. For some applications, the improvement was by more than a factor of two.

McCann, *et al.*, [McCann 93] compared three scheduling policies on a Sequent Symmetry shared-memory multiprocessor. They are called Round-Robin job (RRjob), Equipartition, and Dynamic. In RRjob, originally proposed in [Leutenegger 90], a job is assigned n processors for a time interval $t=k/n$ when its turn arrives, where n is the job's maximum process parallelism and k a constant. The unassigned processors are given to the job whose turn is next (the maximum process parallelism of the applications exceeds $P/2$ in the study). In Equipartition, the target system is subdivided evenly among the jobs present in the system, allocation is independent of instantaneous concurrency, and a job's processor demand equals its maximum process parallelism. Dynamic differs from Equipartition in that allocation depends on the actual degree of concurrency. Applications request processors as their process parallelism increases, and release those they no longer need. Consequently, Dynamic typically induces more processor releases and reallocations than Equipartition. An issue with equipartitioning, used in Dynamic and Equipartition, is that small jobs are favored. Moreover, as smaller jobs typically receive a relatively larger fraction of their processor request, their execution efficiency may be low.

RRjob produced the longest response times as it does not take advantage of the increase in efficiency that typically results from folding, and because progress may be impeded when only a proper subset of a job's processes are active. Dynamic outperformed Equipartition by about 10% for the applications considered. The decrease in job idle times it produced was greater than the additional overhead associated with the larger number of processor releases and reallocations it normally induces.

In another dynamic policy, proposed by Zahorjan and McCann [Zahorjan 90], allocation varies with job parallelism. If an allocation request cannot be satisfied, it waits in a FIFO queue. Pending requests are serviced on FCFS basis. However, a new job is given priority for the allocation of its first processor. If there are no free processors, one is preempted from a job that is allocated two or more processors. An application may receive an unfair share of processors under this scheme. For example, a new job may be allocated a single processor while an earlier arrival is allocated many more.

4.1.1 Problem Statement and Goals

Equipartitioning has been widely evaluated as a dynamic space sharing implementation technique [Tucker 89][Leutenegger 90][McCann 93][Markatos 93]. However, there are two issues with it. First, it favors smaller jobs because they can receive a larger fraction of the number of processors they requested. Second, the execution efficiency of these jobs may be low because they are not folded enough. The overall goal of this study of dynamic space sharing is to evaluate and compare a wide range of dynamic policies that differ in the folding method and in the criteria used in selecting which applications to service. The specific goals are to:

- Investigate the tradeoff between reducing the execution times of individual applications and reducing the overall mean response time. The effects on performance of giving priority to smaller and shorter jobs are also investigated.
- Study the influence of the increase in efficiency that results from folding on the design of dynamic space sharing policies.
- Compare dynamic space sharing policies to the Multifolding First Fit (MFFF) static space sharing job scheduling scheme. MFFF outperformed several other static policies, as can be seen in Chapter 3.

4.2 Workload Model

The results presented in Chapters 2 and 3 show that the increase in efficiency that typically results from folding has a strong influence on the performance of static space sharing policies. To evaluate the influence of this factor under dynamic space sharing, three workload classes are used. In the first, the jobs have linear speedup. In the second, they have speedup curves that correspond to a large sequence of consecutive sequential and parallel phases (Figure 4.1). The fraction of sequential code is assumed constant for the lifetime of the job. The efficiency values when the jobs are allocated the maximum number of processors they can use, n, is assumed to be distributed uniformly over

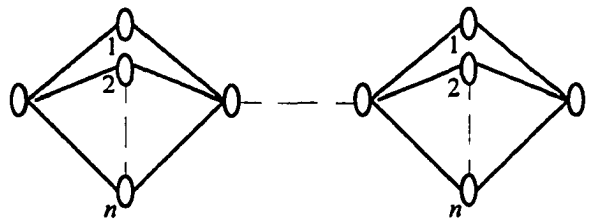

Figure 4.1: sequence of parallel and sequential phases

[0.4,0.9]. The corresponding speedup curve is monotonically increasing, and is denoted by MISP.

Many applications have a structure similar to that shown in Figure 4.1. For example, many parallel jobs consist of processes that iterate over a subset of the input data between successive serial phases. Moreover, MISP is a good approximation to the speedup of many jobs when the number of processors used is not too large, including the jobs specified below.

The third class of jobs consists of thirty applications whose execution times, efficiency curves, and processor demands are specified or can be derived easily from Table 4.1. The first ten applications use a maximum of 16 processors and have the characteristics defined in the table. The column $\xi(p)$ contains the efficiency of the applications when they execute on p processors. Twenty additional job characteristics are derived from Table 4.1 by doubling and quadrupling the problem sizes and number of processors, and assuming the applications scale up. For example, the applications 11 and 21 in Table 4.2 are derived from application 1 in Table 4.1. The job characteristics in Table 4.1 were derived from performance data in [Naik 93a] and [Bailey 92]. Where the performance data is not provided for a value of p, linear interpolation of the efficiency curve is used to derive the missing execution efficiency.

It is assumed that programs are capable of reconfiguring themselves to run on the processors they are allocated. Threads packages augmented with functions that support application reconfiguration were used in implementing dynamic space sharing policies in shared-memory multiprocessors [Tucker 89][McCann 93]. Schemes that support explicit data migration were proposed for NUMAs

Table 4.1: Execution profiles of specific applications

Application	$t(1)$	$\xi(2)$	$\xi(4)$	$\xi(8)$	$\xi(16)$
1	158	0.967	0.897	0.789	0.559
2	185	0.977	0.928	0.913	0.884
3	357	0.977	0.928	0.882	0.786
4	1916	0.984	0.943	0.877	0.768
5	1553	0.982	0.948	0.903	0.844
6	657	0.949	0.842	0.720	0.604
7	2532	0.952	0.892	0.787	0.665
8	6141	0.986	0.966	0.929	0.882
9	9740	0.960	0.915	0.853	0.753
10	28794	0.979	0.935	0.880	0.820

Table 4.2: Sample of derived execution profiles

Application	$t(1)$	$\xi(4)$	$\xi(8)$	$\xi(16)$	$\xi(32)$
11	315	0.967	0.897	0.789	0.559
Application	$t(1)$	$\xi(8)$	$\xi(16)$	$\xi(32)$	$\xi(64)$
21	630	0.967	0.897	0.789	0.559

[Markatos 93] and distributed-memory systems [Naik 93a].

4.3 Allocation Policies

The dynamic allocation schemes defined below and variants of the second policy are studied. It is assumed that a new job requests a number, n, of processors upon arrival. The target system consists of P identical processors, and allocation is topology-independent.

Dynamic Equipartitioning (DEQP): The processors are divided evenly among the applications in the system. They each receive $\min(n,\text{int}(P/M))$ processors, where M is their number. Load levels that result in more than P applications in the system are not considered. The applications are sorted in a queue in the non-decreasing order of their processor demands, and any remaining processors are allocated as follows: The queue is scanned, and a job that is allocated fewer than the number of processors it requested is allocated one more. This procedure is repeated if there are free processors and unsatisfied allocation requests at the end of the scan.

Dynamic Proportional Allocation (DPROP): Each application is allocated $\max(1,\text{int}(n/ff))$ processors. The folding factor ff is equal to $\max(1,T/P)$, where T is the current total processor demand. The remaining processors are allocated to the earliest arrivals, one to each, under the constraint that an application is not allocated more than its processor demand.

Dynamic FCFS (DFCFS): A new application is allocated $\min(n,FP)$ processors, where FP is the number of free processors. It waits if FP=0. When processors are released, the earliest arrival that is allocated less than its processor demand receives $\min(n-p,FP)$ additional processors, where p is that job's current allocation. This procedure is repeated until all requests are satisfied or FP=0.

Dynamic Smallest Job First (DSMJF): The jobs in the system are sorted in the non-decreasing order of their processor demands. A job waits if FP=0, otherwise it is allocated $\min(n,FP)$ processors. When an application terminates, the smallest job that is allocated less than its processor demand receives $\min(n-p,FP)$ additional processors. This procedure is repeated until all requests are satisfied or FP=0.

Applications are allocated more processors on average under DFCFS and DSMJF. DPROP and DEQP reduce waiting times by executing more applications simultaneously, and they take more advantage of the efficiency improvement associated with folding. For example, as many as P jobs can be executing simultaneously under DEQP. The goal of DPROP is to improve the fairness characteristics of DEQP by folding applications evenly. Note that applications that have small processor requests may not be folded under DEQP.

4.4 Results

The scheduling effectiveness of the dynamic policies is equal to one because they avoid processor fragmentation. Their performance is compared using mean response times and fairness curves.

4.4.1 Simulation Parameters

Load levels that result in more than P jobs in the system are not considered. During each run, the simulator generates 5500 jobs. The performance data of the first 500 jobs is discarded so as to ignore startup effects. The number of runs is such that the mean response times obtained are within 5% of the true mean with 95% confidence.

In some experiments, the applications are selected randomly from the set of thirty applications specified in Section 4.2. In these experiments, the mean processor demand N is 32, and the mean execution time when the applications are allocated their processor demand, T_e, is 407.5 time units. In the remaining experiments, the processor demands are uniformly distributed over [2,P=64], and two execution time distributions are used. The distributions are the uniform over [1,360], and the truncated exponential with a mean of 60 and the values outside the interval [1,1000] discarded. As most applications reported in the literature take more than a few seconds to complete, an appropriate time unit is 5 to 20 seconds.

When the number of processors allocated to an application changes as a result of a job arrival

or departure, its execution time on the new number of processors is increased by C time units. The sources of overhead associated with allocation changes include context switches, extra cache misses, and data migration in NUMA multiprocessors. As this overhead is application and system-dependent, a range of overhead values are considered, as in [Zahorjan 90]. Data in [Zahorjan 90] and [Markatos 93] indicate that values of C that do not exceed 1 are of most interest. Note that C=1 represents a high overhead of a few seconds.

4.4.2 Mean Response Times and Fairness

Based on mean response times, the ordering of the policies from best to worst under the uniform execution times distribution, $C \leq 1$, and linear speedup is: DSMJF, DFCFS, DEQP, and DPROP (e.g., Figure 4.2). The ordering under the exponential execution times distribution and linear speedup is: DSMJF, DEQP, DFCFS, and DPROP (e.g., Figure 4.3). DPROP outperformed DFCFS slightly under this workload model and C=0, but it performed slightly worse than DFCFS when C=1 because of the larger number of allocation changes it produces.

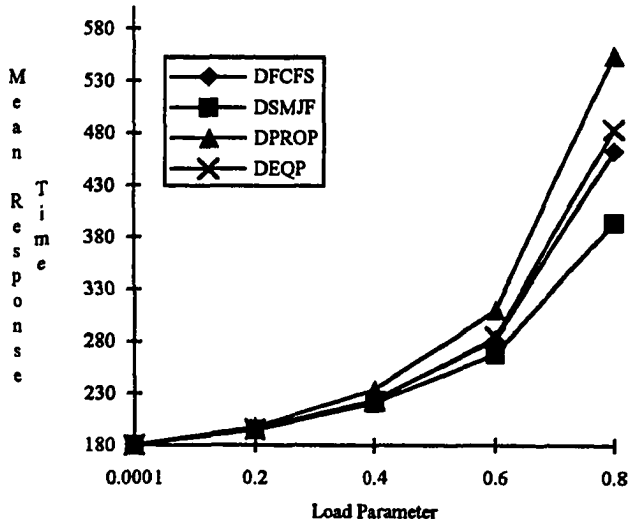

Figure 4.2: Linear speedup, C=0, uniform size and execution time distributions

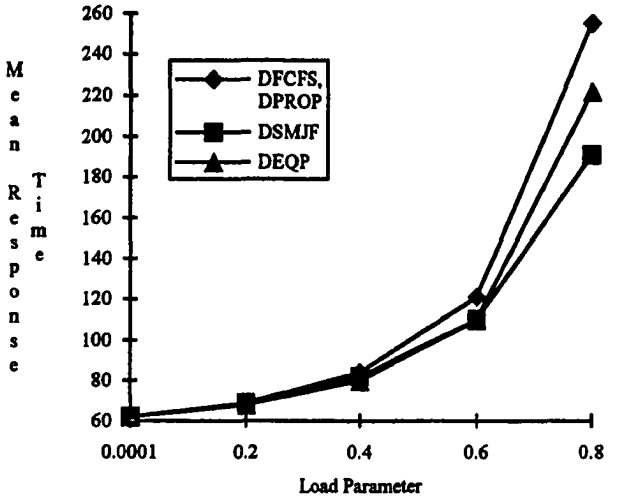

Figure 4.3: Linear speedup, C=1, uniform size distribution, exponential execution time distribution

The performance advantage of DSMJF over DEQP under linear speedup increases with C due to the significantly higher number of allocation changes DEQP results in (e.g., Table 4.3). For example, this performance advantage is less than 5% under the exponential execution times distribution, C=0, and L=0.8, but it is approximately 15% under the same workload model and L when C=1 (Figure 4.3). However, the increase is less significant under the uniform execution times distribution. Under this distribution, the advantage of DSMJF over DEQP rises from 22% to 27% when C increases from 0 to 1 and L=0.8.

Table 4.3: Relative number of application allocation changes, linear speedup, uniform size and execution time distribution

Dynamic Policy	L=0.4	L=0.6	L=0.8
DFCFS	1.00	1.00	1.00
DSMJF	1.00	1.03	1.02
DPROP	1.50	1.51	1.28
DEQP	1.64	1.68	1.44

Under the remaining workload models, where speedup is sublinear, the ordering of the mean response times of the dynamic policies from best to worst is: DEQP, DPROP, DSMJF, and DFCFS (e.g., Figures 4.4-4.8). This different ordering results from the efficiency advantage of folding. The ratio of the mean response times of DPROP and DEQP, *RT*(DPROP)/*RT*(DEQP), increases with *L*, but it remains small. For example, it is 106 to 108% under L=1 in Figures 4.4-4.7 and 112% under L=0.8 in Figure 4.8. Because DEQP produces more allocation changes than DPROP, its relative advantage decreases slightly when *C* increased.

DFCFS and DSMJF performed much worse than DEQP. For example, when L=0.8, the ratio *RT*(DSMJF)/*RT*(DEQP) is about 128% under the uniform execution times distribution, 150% under the exponential execution times distribution, and 215% when the applications are those defined in Tables 4.1 and 4.2. These ratios decreased slightly when *C* increased from 0 to 1.

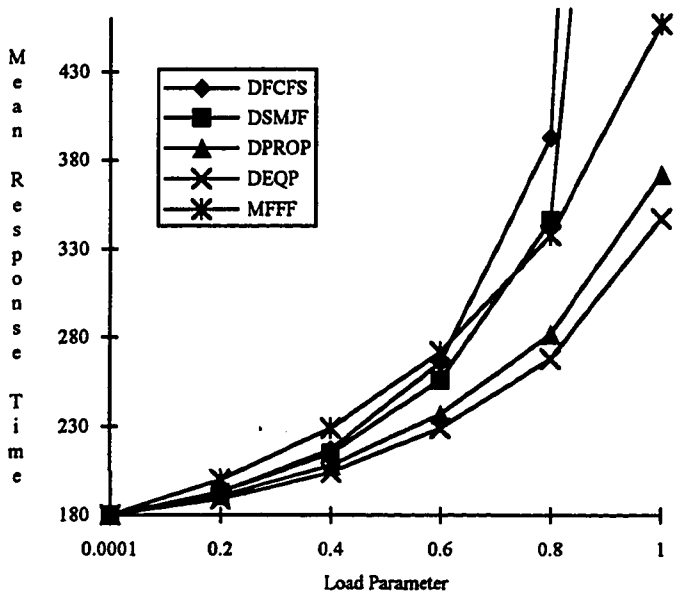

Figure 4.4: MISP speedup, C=0, uniform execution time and size distributions

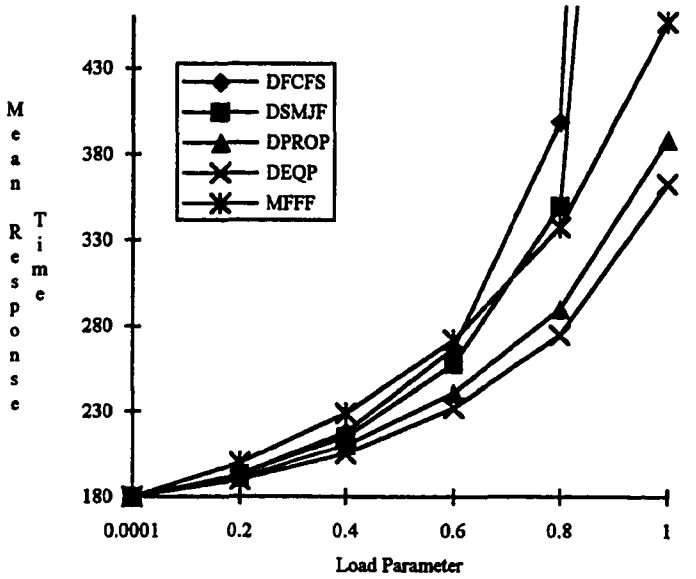

Figure 4.5: MISP speedup, C=1, uniform execution time and size distributions

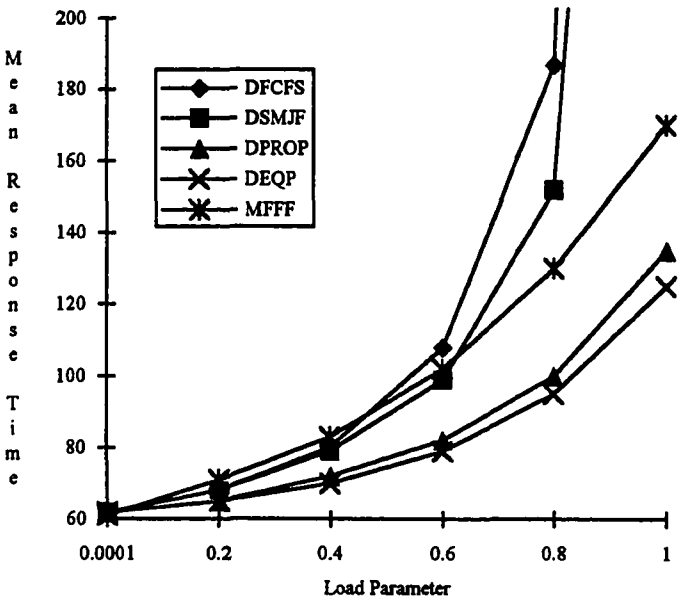

Figure 4.6: MISP speedup, C=0, uniform size distribution, exponential execution time distribution

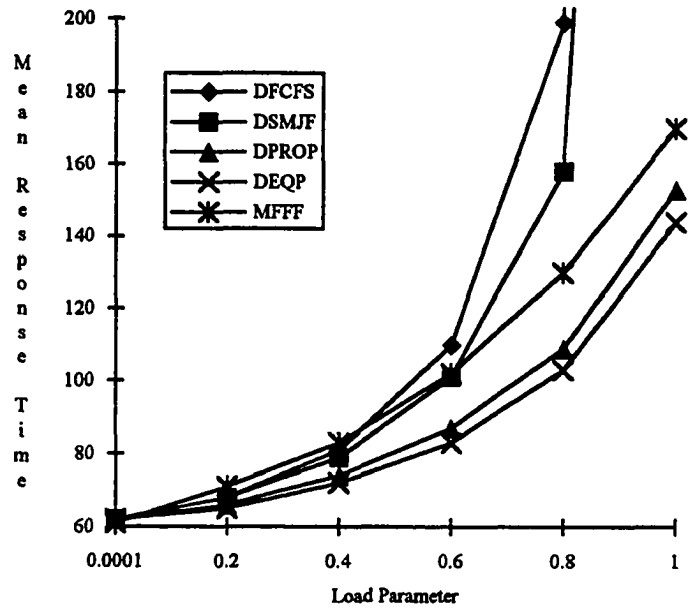

Figure 4.7: MISP speedup, C=1, uniform size distribution, exponential execution time distribution

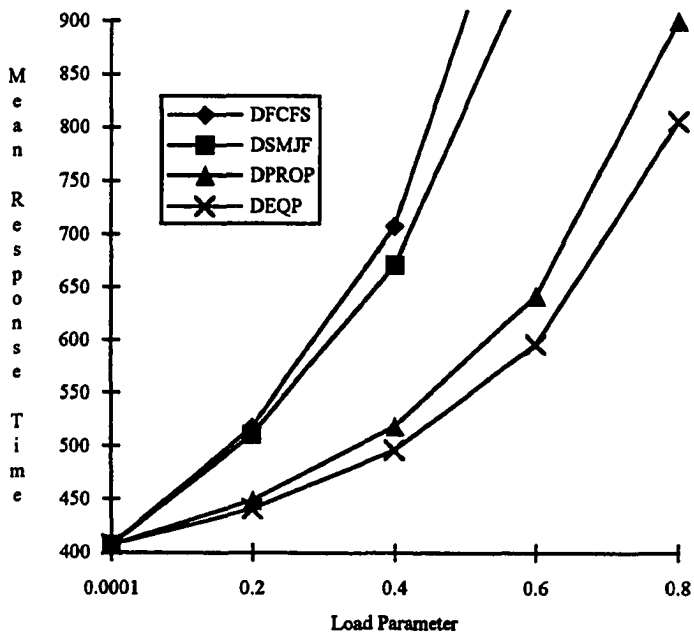

Figure 4.8: Workload defined in Tables 4.1 and 4.2, C=0

As speedup is seldom linear and the efficiency of parallel applications typically increases significantly when they are allocated fewer processors, the policies that fold more applications and reduce the waiting times, DEQP and DPROP, are expected to be superior to DFCFS and DSMJF in practice.

DPROP and DEQP outperformed the static space sharing policy MFFF under the uniform and exponential execution times distributions defined earlier, including when C=1 (see Figures 4.4-4.7). Under these distributions, T_e is much larger than C, and application reconfigurations occur infrequently enough for DPROP and DEQP to be superior to MFFF. However, MFFF can outperform these policies under higher reconfiguration rates and relative overhead costs, as can be seen in Figure 4.9. Here, the execution times are distributed uniformly over the interval [1,10]. The job arrival rates are approximately 35 times larger than those in Figures 4.4 and 4.5, and the rates of application reconfigurations are, accordingly, considerably higher.

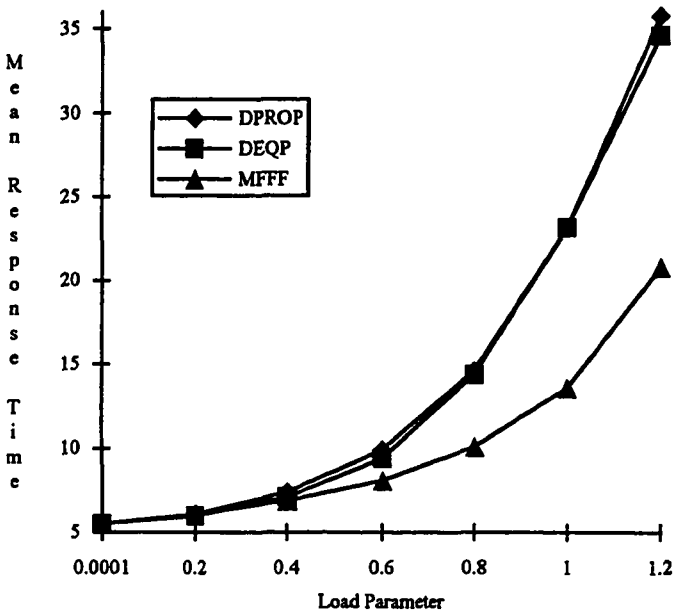

Figure 4.9: MISP speedup, C=0.5, uniform execution times distribution over [1,10], uniform sizes distribution

A disadvantage of DEQP is that it can discriminate against jobs with large processor demands much more than DPROP under high system loads, as can be seen in Figures 4.10 and 4.11. These figures display the fairness curves of DEQP and DPROP under L=0.6 and L=1.0. The mean response times of large jobs are longer under DEQP, however jobs with small to medium processor requests perform better than under DPROP. As allocation in DEQP is based on the even division of processors among jobs, smaller applications receive, on average, a larger fraction of the number of processors they requested. In choosing between DPROP and DEQP, there is tradeoff between overall mean response times and fairness. DPROP has better fairness characteristics, but its overall mean response times are longer.

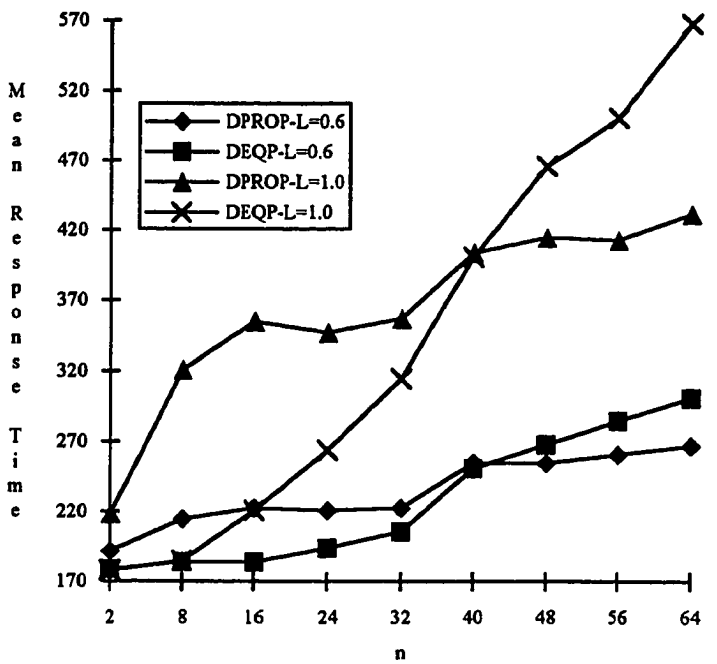

Figure 4.10: Mean response time as a function of job processor demand, C=0, L=0.6 and L=1.0, uniform size and execution time distributions

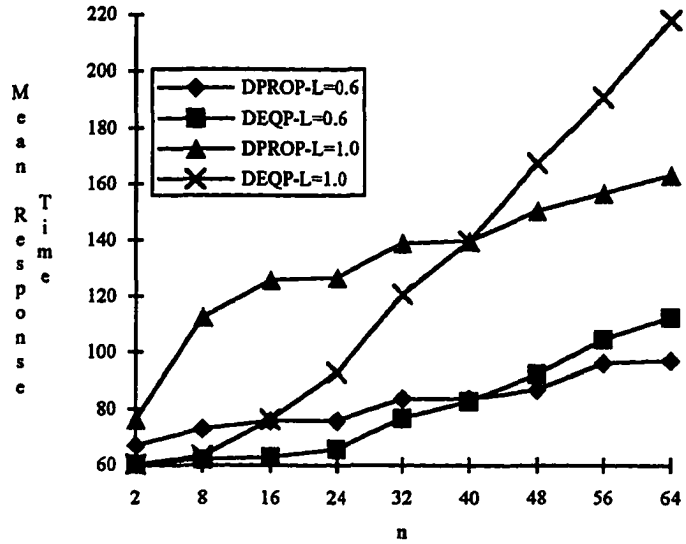

Figure 4.11: Mean response time as a function of job processor demand, C=0, L=0.6 and L=1.0, uniform size distribution, exponential execution time distribution

4.5 DPROP Variants

To study the influence of giving priority to smaller and shorter jobs, two variants of DPROP are considered. In the first variant, denoted by DPROP-SM, smaller jobs are given priority. Let n_j denote the number of processors requested by job j, the job's allocation under DPROP-SM is based on a new processor demand, nd_j, computed by $nd_j=n_j/(1+x*n_j/P)$, where x is a priority parameter greater than zero. When x increases, smaller jobs receive a larger fraction of their processor demands. Allocation is as follows:

1) Compute nd_j for all jobs in the system
2) Compute $S=\sum_j nd_j$
3) Calculate the folding factor $ff=S/P$
4) If $ff < 1$ then $ff=1$
5) For all jobs: allocate $\max(1,nd_j/ff)$ processors to job j
6) Any remaining processors are allocated as in DPROP

In the second DPROP variant, denoted by DPROP-SH, priority is given to shorter jobs. It is

assumed that applications can be classified a priori as normal or long, and allocation is as follows:

1) Allocate 1 processor to each job in the system
2) For all jobs do: If job j is long then $nd_j=(n_j-1)/x$ else $nd_j=n_j-1$ (x is a priority parameter greater than zero)
3) Compute $S=\sum_j nd_j$
4) Calculate the folding factor $ff=S/FP$ (FP is the number of free processors)
5) If $ff<1$ then $ff=1$
6) For all jobs: allocate int(nd_j/ff) additional processors to job j
7) Any remaining processors are allocated as in DPROP

DPROP-SM/m and DPROP-SH/m refer to DPROP-SM and DPROP-SH when x=m. A wide range of priority parameters (i.e., values of x) were considered. DPROP-SM/2 outperformed DPROP, but it produced slightly longer mean response times than DEQP (e.g., Figures 4.12-14). An advantage of DPROP-SM/2 over DEQP is that it discriminates against large jobs less, as can be seen in Figure

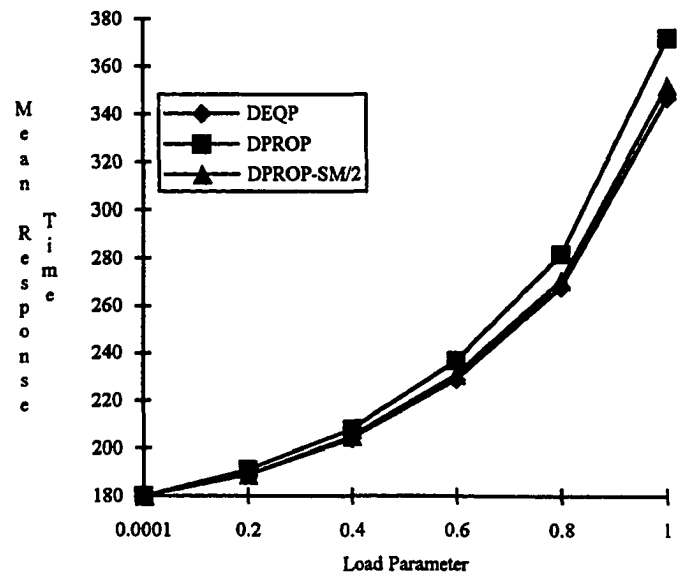

Figure 4.12: MISP speedup, uniform size and execution time distributions

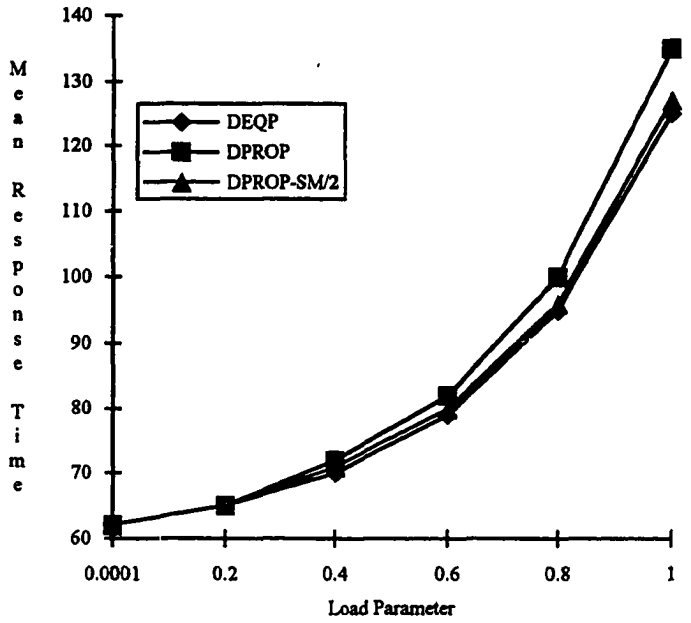

Figure 4.13: MISP speedup, uniform size distribution, exponential execution time distribution

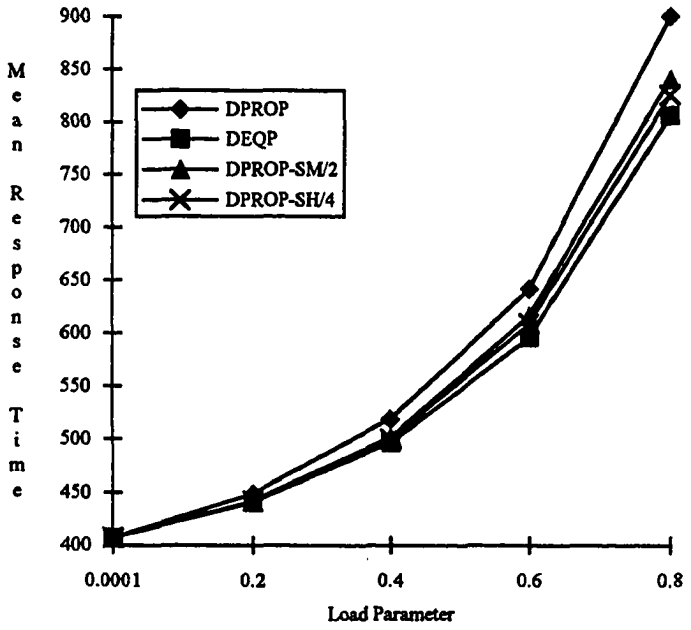

Figure 4.14: Workload defined in Tables 4.1 and 4.2

4.15. Increasing x beyond 2 decreased the overall mean response times slightly, but it also resulted in a small lengthening of the mean turnaround times of large jobs (e.g., Figure 4.15).

In the simulation experiments of DPROP-SH under the third workload class, applications 9, 10, 19, 20, 29, and 30 are considered long. A job is considered long if its execution time on the number of processors it requested is greater than the mean $t(n)$ (i.e., T_e) in the experiments that use the uniform and exponential execution times distributions.

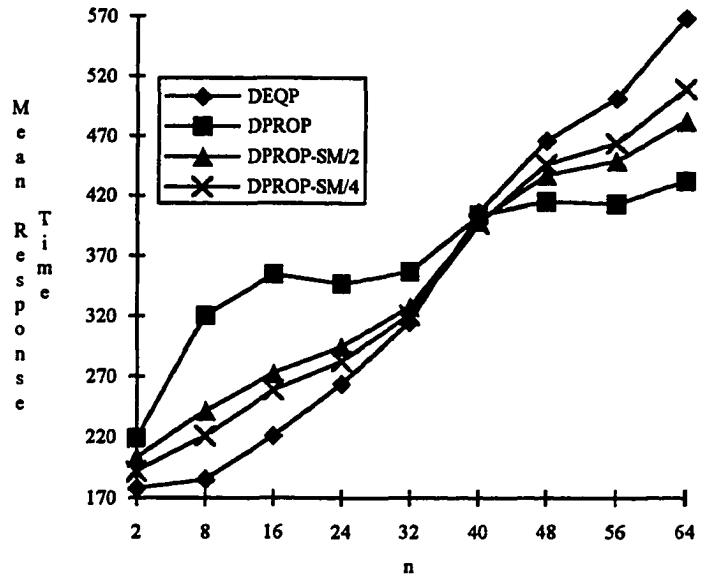

Figure 4.15: Mean response time as a function of job processor demand, L=1.0, uniform size and execution time distributions

The biggest improvement in performance obtained with DPROP-SH was achieved under the third workload class. Under this workload, the best DPROP-SH mean response times are almost identical to those of DEQP, and they are slightly better than those of DPROP-SM/2. The best results were obtained with large values of x (x>16). As x was increased beyond 1, the response times decreased rapidly initially, then the rate of decrease slowed down significantly. A problem with large values of x is that they degrade the performance of long jobs. The performance of DPROP-SH under an interme-

diate value of x (x=4) is compared to that of DEQP and DPROP-SM/2 in Figure 4.14, where it is shown that DPROP-SH/4 is slightly better than DPROP-SM/2, but worse than DEQP. A problem with DPROP-SH is that it is less practical than the other policies as it is difficult to determine or estimate the execution times of jobs in advance of their execution. Moreover, DPROP-SH does not offer significant advantage over DEQP and DPROP-SM/2.

4.6 Conclusions

When the overhead of application reconfigurations is small, dynamic space sharing is, as expected, superior to static space sharing. However, when the overhead is high, static space sharing can outperform dynamic space sharing significantly, as can be seen in Figure 4.9. Dynamic space sharing is expected to be a poor strategy when parallelism is fine-grained and allocation changes occur frequently (i.e., C/T_e is large).

As the efficiency of parallel applications typically increases significantly when the number of allocated processors decreases, DEQP, and DPROP and its variants are expected to outperform DFCFS and DSMJF in practice. Overall, DEQP and DPROP-SM/2 are the best dynamic policies considered in this study. DEQP produced slightly shorter mean response times than DPROP-SM/2, but it discriminated against large jobs significantly more. Based on the mean response time performance parameter alone, DEQP, which has commonly been used in implementing dynamic space sharing, is the best dynamic policy considered in this study.

Although DPROP-SH can improve on the performance of DPROP, the improvement is not such that DPROP-SH is significantly better than DEQP or DPROP-SM/2. Moreover, DPROP-SH is less practical as it requires that the execution times of applications be estimated or known a priori.

A reason why DEQP, DPROP might not be superior to DFCFS and DSMJF is that they can result in a large number of applications executing simultaneously. Depending on memory access patterns and the bandwidth of the memory subsystem, the efficiency of jobs may suffer, and a policy that

limits their number may be superior. A natural extension to this work would be to study of the effect of interconnection contention on the performance of dynamic space sharing policies.

5 CONCLUSIONS AND FUTURE WORK

Alternative topology-independent program-based space sharing policies that differ in the folding method and the job selection criteria are compared using extensive simulation in Chapters 2, 3, and 4. The results of this comparison lead to the following main conclusions:

- Traditional allocation algorithms (e.g., first-come-first-served and first-fit), which do not support application folding, can produce high processor fragmentation under medium to high system loads even when allocation is topology-independent. As a result, their mean response times under general workload models start increasing sharply under load levels significantly smaller than one, as seen in Chapter 2. Free processors remain idle if their number is smaller than the processor demands of the waiting applications, and the efficiency improvement that typically results when applications execute on fewer processors is not exploited.
- In implementing the static space sharing processor allocation strategy, adaptive folding is superior to no folding and unconstrained folding. The major disadvantage of unconstrained folding is the high folding fragmentation it can produce under most system loads. Because released processors are not allocated to folded applications in static space sharing, a large number of processors may remain idle while parallel jobs are executing on fewer than their processor demands. This problem is especially severe under medium loads when many applications are folded and the waiting queue is short. The advantage of unconstrained folding over no folding is that it exploits the efficiency benefit of allocation reduction. By increasing the maximum factor by which applications are allowed to be folded with the load, the adaptive approach prevents them from running on too few processors under moderate loads, when released processors are likely to remain idle for a long time. Adaptive folding produced higher and more stable scheduling effectiveness and shorter mean response times.

- The adaptive multifolding static policy MFFF, which can fold multiple applications when a job terminates, is robust and superior to policies that fold at most one application upon a job completion. It is robust in that no significant mean response time improvement was obtained when priority was given to applications with small processor demands or short execution times. Its advantage is that it exploits the efficiency benefit of folding more than the other policies.
- As dynamic space sharing avoids folding fragmentation, it is superior to static space sharing provided the overhead of application reconfigurations does not offset the improvement in performance that results from the fragmentation reduction. Experimental data obtained on UMA and NUMA systems indicates that this overhead is not excessive when parallelism is coarse-grained and allocation changes are infrequent [Zahorjan 90][McCann 93][Markatos 93].
- Dynamic space sharing policies that reduce waiting times by executing a large number of applications simultaneously (DEQP, DPROP, and DPROP-SM) are superior to policies that limit the number of active applications and reduce execution times (DFCFS and DSMJF). Dynamic schemes based on the assumption that applications are classified a priori as short or long did not result in significant improvement over DEQP and DPROP-SM.

An obvious extension to this work is to implement the most promising policies and compare their performance. As topology-based allocation can improve interconnection performance significantly in distributed-memory systems, the tradeoff between processor fragmentation and exploiting locality in these systems needs investigation. In particular, folding topology-based schemes should be studied and compared to folding topology-independent and no folding topology-based schemes.

An issue with the static and dynamic policies that resulted in the best performance in this study is that they can execute a large number of applications concurrently. Depending on parallelism granularity and interconnection bandwidth, the efficiency of jobs may suffer because of increased contention for the interconnection subsystem. Another possible extension to this work is to study the effect

of memory subsystem contention on the performance of policies that execute many applications simultaneously.

REFERENCES

[Abraham 92] Abraham, S., and Padmanabhan, K. "Effect of data access delays and system partitionability on the dynamic performance of a shared memory multiprocessor". Proceedings of the Supercomputing '92 Conference, pp. 674-682, November 1992.

[Ahmad 94] Ahmad, I., Ghafoor, A., and Fox G. C. "Hierarchical scheduling of dynamic parallel computations on hypercube multicomputers". *Journal of Parallel and Distributed Computing*, vol. 20, pp. 317-329, March 1994.

[Anderson 92] Anderson, T., Bershad, B., Lazowska, E., and Levy, H. "Scheduler activations: effective kernel support for user-level management of parallelism". *ACM Transactions on Computer Systems*, vol. 10, pp. 53-79, February 1992.

[Bailey 92] Bailey, D., Barszcz, E., Dagun, L., and Simon, H. "NAS parallel benchmark results", Proceedings of the Supercomputing '92 Conference, pp. 386-393, November 1992.

[Baker 80] Baker, B., Coffman, E., and Rivest, R. "Orthogonal packings in two dimensions". *SIAM J. Comput.*, vol. 9, pp. 846-855, November 1980.

[Beckerle 92] Beckerle, M. "An overview of the START(*T) computer system". Motorola technical report MCRC-TR-30, revision 2, Motorola Cambridge Research Center, Cambridge, MA, October 1992.

[Chandra 93] Chandra, R., Gupta, A., and Hennessy, J. "Data locality and load balancing in COOL". Proceedings of 4th ACM Symposium on PPoPP, pp. 249-259, May 1993.

[Coffman 80] Coffman, E., Garey, M., Johnson, D., and Tarjan, R. "Performance bounds for level-oriented two-dimensional packing algorithms". *SIAM J. Comput.* , vol. 9, pp. 808-826, November 1980.

[Coffman 91] Coffman, E., and Lueker, G. *Probabilistic Analysis of Packing and Partitioning Algorithms*. John Wiley and Sons, New York, 1991.

[Eigenmann 91] Eigenmann, R., Hoeflinger, J., Jaxon, G., and Padua, D. "Cedar Fortran and its restructuring compiler". *Advances in Languages and Compilers for Parallel Processing*, editors A. Nicolau *et al.*, The MIT Press, Cambridge, MA, pp. 1-23, 1991.

[Feitelson 90] Feitelson, D., and Rudolph, L. "Distributed hierarchical control for parallel processing". *IEEE Computer*, vol. 23, pp. 65-77, May 1990.

[Flatt 89] Flatt, H., and Kennedy, K. "Performance of parallel processors". *Parallel Computing*, vol. 12, pp. 1-20, 1989.

[Ghosal 91] Ghosal, D., Serazzi, G., Tripathi, S. "The processor working set and its use in scheduling multiprocessor systems". *IEEE Transactions on Software Engineering*, vol. 17, pp. 443-453, May 1991.

[Gupta 91] Gupta, A., Tucker, A., and Urushibara, S. "The impact of operating system scheduling policies and synchronization methods on the performance of computer systems". Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp. 120-132, May 1991.

[Gupta 93] Gupta, A., and Kumar, V. "Performance properties of large scale parallel systems". *Journal of Parallel and Distributed Computing*, vol. 19, pp. 234-244, 1993.

[Gustafson 88] Gustafson, J. "Reevaluating Amdahl's law". *Communications of the ACM*, vol. 31, pp. 532-533, May 1988.

[Hicks 93] Hicks, J., Chiou, D., Seong, B., and Arvind "Performance studies of Id on the Monsoon dataflow system". *Journal of Parallel and Distributed Computing*, vol. 18, pp. 273-300, 1993.

[Hirnandani 92] Hirnandani, S., Kennedy, K., and Tseng, C.-W. "Compiling Fortran D for MIMD distribute-memory machines". *Communications of the ACM*, vol. 35, pp. 66-80, August 1992.

[Krishnamurti 92] Krishnamurti, R., and Ma, E. "An approximation algorithm for scheduling tasks on varying partition sizes in partitionable multiprocessor systems". *IEEE Transactions on Computers*, vol. 41, pp. 1572-1579, December 1992.

[Lenoski 92] Lenoski, D., Laudon, J., Gharachorolo, K., Weber, W.-D., Gupta, A., Hennessy, J., Horowitz, M., and Lam, M. "The Stanford DASH multiprocessor". *IEEE Computer*, pp. 62-79, March 1992.

[Leutenegger 90] Leutenegger, S., and Vernon, M. "The performance of multiprogrammed multiprocessor scheduling policies". Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp. 226-236, May 1990.

[Li 91] Li, K., and Cheng, K.-H. "A two-dimensional buddy system for dynamic resource allocation in a partitionable mesh connected system". *Journal of Parallel and Distributed Computing*, vol. 12, pp. 79-83, 1991.

[Liu 94] Liu, W., Lo, V., Windisch, K., and Nitzberg, B. "Non-contiguous processor allocation algorithms for distributed memory multicomputers". Proceedings of the Supercomputing '94 Conference, pp. 227-236, November 1994.

[Majumdar 88] Majumdar, S., Eager, D., and Bunt, R. "Scheduling in multiprogrammed parallel systems". Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp. 104-114, May 1988.

[Markatos 92a] Markatos, E. and LeBlanc, T. "Load balancing vs. locality management in shared-memory multiprocessors". Proceedings of the 1992 International Conference on Parallel Processing, pp. I-258 to I-267, 1992.

[Markatos 92b] Markatos, E. and LeBlanc, T. "Using processor affinity in loop scheduling on shared-memory multiprocessors". Proceedings of the Supercomputing '92 Conference, pp. 104-113, November 1992.

[Markatos 93] Markatos, E. "Scheduling for locality in shared-memory multiprocessors". Ph.D. dissertation, University of Rochester, Rochester, New York, 1993.

[McCann 93] McCann, C., Vaswani, R., and Zahorjan, J. "A Dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors". *ACM Transactions on Computer Systems*, vol. 11, pp. 146-178, May 1993.

[Naik 93a] Naik, V., Setia, S., and Squillante, M. "Performance analysis of job scheduling policies in parallel supercomputing environments". Proceedings of the Supercomputing '93 Conference, pp. 824-833, November 1993.

[Naik 93b] Naik, V., Setia, S., and Squillante, M. "Scheduling of large scientific applications on distributed memory multiprocessor systems". Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing, pp. 913-922, 1993.

[Ousterhout 82] Ousterhout, J. "Scheduling techniques for concurrent systems". 3rd International Conference on Distributed Computing Systems, pp. 22-30, October 1982.

[Pasquale 91] Pasquale, J., Bittel, B., and Kraiman, D. "A static and dynamic workload characterization study of the San Diego supercomputer center Cray X-MP". Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp. 218-219, May 1991.

[Seitz 90] Seitz, C. "Multicomputers". *Developments in Concurrency and Communication*. Addison-Wesley, Reading, MA, pp. 131-200, 1990.

[Setia 93] Setia, S., Squillante, M., and Tripathi, S. "Processor scheduling on multiprogrammed, distributed-memory parallel computers". Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp. 158-170, May 1993.

[Setia 94] Setia, S., Squillante, M., and Tripathi, S. "Analysis of processor allocation in multiprogrammed, distributed-memory parallel processing systems". *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 401-420, April 1994.

[Sevcik 89] Sevcik, K. C. "Characterizations of parallelism in applications and their use in scheduling". *Performance Evaluation Review*, vol. 17, pp. 171-180, May 1989.

[Squillante 91] Squillante, M. and Randolph, D. "Analysis of task migration in shared-memory multiprocessor scheduling". Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp. 143-155, May 1991.

[Squillante 93] Squillante, M., and Lazowska, E. "Using processor-cache affinity information in shared-memory multiprocessor scheduling". *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 131-143, February 1993.

[Stone 90] Stone, H. *High Performance Computer Architecture*, second edition, Addison-Wesley, Reading, MA, pp. 309-311, 1990.

[Torrellas 93] Torrellas, J., Tucker, A., and Gupta A. "Benefits of cache-affinity scheduling in shared-memory multiprocessors: A summary". Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp. 272-274, May 1993.

[Tucker 89] Tucker, A., and Gupta, A. "Process control and scheduling issues for multiprogrammed shared-memory multiprocessors". Proceedings of the 12th ACM Symposium on Operating Systems Principles, pp. 159-166, December 1989.

[Tuomenoska 85] Tuomenoska, D. L. and Siegel, H.J. "Task scheduling on the PASM parallel processing system". *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 145-157, February 1985.

[Zahorjan 90] Zahorjan, J., and McCann, C. "Processor scheduling in shared memory multiprocessors". Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp. 214-225, May 1990.

[Zhou 91] Zhou, S., and Brecht, T. "Processor pool-based scheduling for large-scale NUMA multiprocessors". Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp. 133-142, May 1991.