Parallel Application Scheduling on Networks of Workstations

by

Stergios Anastasiadis

A thesis submitted in conformity with the requirements
for the Degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Parallel Application Scheduling
on Networks of Workstations

Stergios Anastasiadis
Master of Science
Department of Computer Science
University of Toronto 1996

Abstract

Parallel application support is one of the ways that have been recently proposed for exploiting the idle computing capacity of workstation networks. However, it has been unclear how to most effectively schedule the processors among different job requests. The distributed memory nature of such environments and the structure of the corresponding applications cause many solutions that were successful for shared-memory machines to be inadequate in the new environment.

In this thesis, we investigate how knowledge of system load and application characteristics can be used in scheduling decisions. We propose a new algorithm that, with proper exploitation of both the information types above, manages to improve the performance of non-preemptive scheduling relative to other rules. Thus, we show that with the appropriate support in the operating system, the application developers can be left free to use the programming model best suited to the individual application needs.

Acknowledgments

It is a great pleasure that I am given the opportunity to thank all those people, who each in his way have helped for the successful completion of this thesis. I am mostly grateful to my supervisor Prof. Ken Sevcik. During the months that passed, it was an exciting experience for me to realize his deep knowledge and intuition about the computer system modeling problems. He always made me feel free to investigate new approaches, and kept me on the right direction with his invaluable guidance. I am also indebted to my second reader Prof. Tarek Abdelrahman. He accepted with a big smile to read timely my thesis and offer his comments.

I would like also to express my gratitude to all my friends here in Toronto for making Canada such a warm place to live (despite the cold weather). Special thanks go to Nick Koudas for all the help and encouragement that he gave me during the months that passed. He motivated me to deal with computer systems research and we spent numerous hours laughing about unbelievable events at the University of Patras.

I am also grateful to Michalis and Petros Faloutsos. We shared lots of fun and beer drinking together, and they were always ready to offer their support in everything that went wrong. I used to enjoy my late night working on campus by listening to the perfect live sounds of Vassilis Tzerpos' guitar, although it was only recently that I heard about this favorite hobby of his and realized that his office was attached to one of the CSRI labs...

I was also lucky enough to find here Spiros Mancoridis before his recent graduation and Dimitra Vista. The great jokes of Spiros and the hospitality of Dimitra made a lot of days in Canada unique and unforgettable.

Dimitris Achlioptas was the person who found fun in answering my naive questions before and after my arrival. He was my officemate for one year, and we spent many hours talking about the big questions of life.

Thodoros Topaloglou used to offer expert and reliable advice that always worked. Theodoulos Garefalakis and Panagiotis Tsaparas, with their coming last September, gave additional joy to our gang. Later, Theodoulos became an excellent officemate of mine.

I am grateful to the graduate secretary of the department Kathy Yen for being so helpful, and the University of Toronto for offering generous financial support.

Above all, I feel the need to thank my family at home. Though at thousands of miles away, they never stopped to be close to me and supply their warmth and love.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Recent advances in software and hardware technology have improved considerably the performance of interconnected workstations. In most cases, each machine is owned by a single user, whose typical processing needs rarely require the full workstation capacity. Thus, valuable computing resources remain underutilized most of the time, yet each user is restricted to running applications within the boundaries of a single workstation.

Load balancing environments, installed on top of the existing workstation clusters and incorporating efficient remote execution facilities, allow fair exploitation of the nodes across the distributed system. Actually, commercial platforms already exist that improve significantly the system throughput and increase the system utilization.

There is still a need, however, to develop distributed parallel programming environments that will permit efficient execution of applications with multiple processes spawned on more than one machine. Usually, the communication and synchronization demands of the parallel applications conflict with the privileges of the workstation owners and degrade seriously the predictability of the service quality provided by the system.

Actually, it is not even clear which is the most effective way of scheduling the parallel application threads when the sequential workload is ignored. Many existing scheduling policies were inspired from scheduling results in uniprocessors, and were initially designed for shared-memory parallel systems.

However, distributed-memory machines are considerably different in both architectural design and performance behavior relative to shared-memory parallel systems. Therefore, it is not surprising that the effectiveness and the implementation effort of using shared-memory scheduling algorithms in distributed-memory environments may be different from those reported for shared-memory machines.

In particular, serious reservations can be expressed about the actual benefit of implementing a dynamic partitioning policy on a network of workstation-class nodes. Although dynamic partitioning is typically very effective in shared-memory, its advantages are expected to be restricted to a narrow range of workloads in a distributed-memory system. The difficulty mainly arises from the master-slave parallel programming model typically required by this scheduling policy, and the significant development effort involved for applications better suited to other paradigms.

Changing the number of processors assigned to an application during execution may require redistributing the data structures and, correspondingly, incurring increased runtime. It has been proven, both analytically and experimentally, that the dynamic reconfiguration overhead, which is dependent on the structure of the individual application, can more than cancel the related benefit in distributed-memory systems.

Until recently, it was considered unrealistic to assume knowledge of the execution behavior of an application, and to use this for performance improvement in scheduling. However, repetitive patterns in the runtime of an application at different numbers of processors, and use of advanced execution time models can facilitate the statistics gathering procedures. Although some policies based on application characteristics are outperformed by others based just on the system load, it is not clear if the best possible non-dynamic policies have yet been found.

Understanding how different levels of information about the workload attributes improve the processor allocation decisions may allow us to come up with enhanced rules that combine the advantageous properties of the best existing scheduling algorithms. Thus, it may be possible to improve the performance of non-preemptive scheduling by estimating and utilizing properly the useful parameters of the individual applications and the system.

1.1 Objective of the Study

The present study was motivated by the need to reduce the average response time provided by workstation clusters through running parallelized applications. Recognizing the restrictions imposed by the message-passing nature of such systems, we set as goal the answer to the following questions:

- Which are the inherent properties of previously proposed load-based adaptive algorithms that allow them to behave well under a wide range of arrival rates and application types ?
- What possibilities do we have for designing new load-based adaptive algorithms with even better performance ?
- What benefit can be obtained by incorporating knowledge of application characteristics in scheduling decisions ?
- What are the positive and negative properties of the Dynamic Policies, and how closely can their almost ideal performance assuming zero reconfiguration overhead be approached by non-preemptive algorithms ?

We do not place restrictions on the programming paradigm to which our results are applicable or even the architecture of the parallel machine. Message-passing, shared-memory and data-parallel programming models have all been found useful for particular applications on networks of workstations. In addition, our conclusions will be useful even on shared-memory machines, for parallel jobs that cannot easily adjust to the requirements of Dynamic Scheduling. However, the number of such applications is much larger in message-passing systems, with the Single Program Multiple Data model being the prevalent programming approach, and this is the reason why such architectures constitute our main target.

We have used simulation for evaluating the scheduling approaches under consideration. This facilitated the coverage of a wide range of workload assumptions and system load levels. In addition, it allowed us to ignore insignificant system architecture details, which helped us reach clearer conclusions about the properties of the scheduling policies and their effect in performance improvement.

1.2 Thesis Organization

In Chapter 2, an overview of the related literature is presented. Recent technological advances that justify our research are briefly explained, and workload characteristics of typical distributed environments are introduced. Then, the architecture of existing parallel distributed environments is presented. The remaining part of the chapter presents the significant results on parallel scheduling up to now.

In Chapter 3, the assumptions we make about the software and hardware structure of the system are explained. In addition, the workload used in the study is specified.

In Chapter 4, definitions are given for existing and new scheduling rules. Use of a unifying terminology highlights their similarities and differences and allows better understanding of the performance, as demonstrated in our experiments or reported from previous studies.

In Chapter 5, details are given about the simulation modeling of the system, the application characteristics, and the derivation of the parameters used. In addition, explanations are given for important implementation decisions.

In Chapter 6, the experimental results are displayed graphically, and reasons are given for the relative performance of the policies. Some previous results are validated, and our original conclusions are presented and justified.

In Chapter 7, the conclusions of our work are summarized.

Chapter 2

Literature Review

Today, massively parallel processors (MPPs) tend to be constructed as large collections of workstation-class nodes connected by dedicated low latency networks and exploiting commodity technologies (Anderson et al. 1995). However, they usually lag technologically one or two years behind workstations with comparable parts. An additional drawback is the increased engineering effort required for customizing their software and hardware components, and the relatively small volume of sales to cover the incurred cost. Although they are successful at delivering very high performance in certain application domains, they cannot provide high throughput in general purpose tasks, like file system service. However, MPPs provide low communication latency and overhead, and they offer a global system view to parallel programs.

In the domain of workstation technology, switched local area networks allow bandwidth to scale with the number of processors and low overhead protocols make fast communication possible (von Eicken et al. 1995). Also, workstations are improving in processing capacity, memory and disk storage, while the increasing memory-processor speed gap can be bridged, by taking advantage of the huge aggregate memory pool existing in workstation networks. Software-based fault isolation allows privileged operating system resource allocation code to execute in the application's address space with relatively low overhead. A user-level virtual operating system layer can be built using the underlying commercial system as a building block (Wahbe et al. 1993; Jones 1993; Vahdat et al. 1994).

2.1 Workload Characterization

Possible ways of effectively utilizing the unused computing capacity of workstation networks are being explored by several researchers. A fundamental step in this effort is the study of the resource usage patterns and the availability of workstations as places for remote execution.

At the University of Wisconsin, the Condor scheduling system was developed in order to manage efficiently the idle computing capacity of the local workstation network (Mutka and Livny 1987; Litzkow et al. 1988). In a study of the previously wasted processing resources (Mutka and Livny 1991), a workstation was considered non-available if the average processor utilization was larger than one-fourth of one percent at any time during a previous predefined period. It was observed that the relative frequency distributions of availability and non-availability intervals can be represented well by 2-stage and 3-stage hyper-exponential distributions, respectively. Furthermore, it is suggested that workstations with recent long available intervals should be considered for job placement before workstations with recent short available intervals.

Feitelson and Nitzberg (1995) study the statistics of a parallel workload on a 128-node hypercube iPSC/860 at the NASA Ames Center. While the majority of the jobs are sequential, most of the resources (node-seconds) are consumed by parallel jobs using 32, 64 or 128 nodes. The distribution of runtimes indicates that jobs with a higher degree of parallelism tend to run longer. Both job runtimes and inter-arrival times have a high coefficient of variation, suggesting hyper-exponential distributions. When multiple executions of the same application on the same number of nodes are examined, the coefficient of variation tends to be less than 1, which indicates that application resource requirements can be predicted.

A similar study is presented by Wang (1995), where measurements are conducted on workloads from workstation networks running under the LSF (Zhou et al. 1993; Platform Computing Corp. 1994). It is reported that job arrival patterns have similar profiles for both academic and industrial sites. Usually, in a 24-hour period there are two arrival peaks, and the job inter-arrival times conform to the Weibull distribution. Most of the jobs are submitted by a few hosts and there a few heavy users responsible for most of the system load. The majority of the jobs are small, but the large jobs dominate in the usage of processor resources. Small jobs spend relatively more of their time on I/O operations than large jobs.

Another related study is that of Leutenegger and Sun (1993). They study the model of a distributed system consisting of W homogeneous workstations and one owner per node. A parallel job consists of W tasks and has a single parallel phase. The main assumption is that a sequential process initiated by a workstation owner suspends the parallel task running on the owned node. The ratio of the parallel task service demand to the mean service demand of the non-parallel owner processes, called *task ratio*, is found to be the determining factor of the performance of parallel programs. It is shown that memory-bounded speedup jobs exhibit better performance than jobs having fixed-load speedup, since the task ratio of the former is fixed, while that of the latter decreases with an increase in the number of workstations (Sun and Ni 1993).

Issues arising when mixing sequential and parallel workloads are examined by Arpaci et al. (1994). It is proven that lack of coordination in the round-robin scheduling of the threads can significantly slow down the jobs, especially those with fine-grained parallelism. Also, skew in the clocks of the workstations can degrade the performance of gang scheduled applications, unless a large quantum is used. Daemon process interferences can even double the execution time of fine-grained parallel jobs. When the interactive load from the workstation owner is considered, the migration of parallel processes is necessary for applications without load imbalance, though migration delay itself can have negative effects on users. When users return to their workstations, they may have to wait for both the parallel programs to migrate and their previous contexts to be restored. The solution of restricting the number of interruptions per user is proposed.

2.2 Distributed Parallel Environments

The need for handling very large computational jobs, combined with the absence of a supercomputer in several organizations, led to the development of distributed parallel environments that exploit the unused processor time of workstation networks for execution of parallel applications.

One such batching system is presented by Silverman and Stuart (1989) from the MITRE Corp. The implementation involves no modifications to the Unix kernel, and batch daemons in the network machines are the master controlling entities. Maintenance of load information for the local machine and control of the active jobs are two responsibilities of the system. Parallel programming and process migration are possible through a set of C subroutines forming the remote command library, and include files with predefined parameters. An abstract data type, called rcmd (Remote Command Multiprogramming Data), is used for describing a process, or for the communication among and synchronization of multiple processes running on different hosts.

Amber (Chase et al. 1989) was designed to explore the use of a distributed object model as a tool for structuring parallel programs on networks of shared-memory multiprocessor workstations. Later, it was extended to allow the execution of parallel-distributed programs that adapt to three types of node reconfiguration: shrinkage in size, changes in membership, or growth (Feeley et al. 1991). At the system level, all these cases require support for program state transfer and communication forwarding. At the application level, the program must resolve the load imbalances that occur. The usual alternative approach is to use general-purpose facilities that migrate and schedule processes without knowledge of their internal state. Thus, efficiency is sacrificed due to the heavy-weight kernel abstractions involved and from placing multiple processes on the same physical node.

The feasibility of data-parallel programming in distributed computing environments is investigated by Nedeljkovic and Quinn (1993). The Dataparallel C programmer envisions an SIMD computer consisting of a front-end uniprocessor attached to a back-end parallel processor, with the number of processors activated being adjustable. The processors are called virtual, as their number is independent of the number of physical processors available. To avoid load imbalance, the compiler allows users to specify the relative speeds of individual workstations at compile time. In addition, the load on the workstations is controlled dynamically by properly adjusting the number of virtual processors emulated on each node.

A slightly different proposal, in the same context, is described by Carriero et al. (1995). The Piranha system is an adaptive version of master-worker parallelism. The application defines a worker function designated by the distinguished name *piranha()*. Whenever a node becomes idle, the system creates a new worker process on that node, using the *piranha()* function as a template. The piranha application must also define a function called *retreat()*, invoked automatically by the system on any node that stops executing *piranha()* and goes back to its owner's commands. Finally, a *feeder()* user-defined routine is included that runs on the submitting node and manages the ongoing computation.

2.3 Multiprogramming and Scheduling

Multiprogramming is the activity of executing multiple independent applications by time-sharing or space-sharing the system resources. It is usually introduced as a solution to the problem of low resource utilization, where an appropriate mix of jobs with complementary requirements can keep the utilization of system resources high. For many users, the increased capacity for interactive use is even more important than the improved utilization. *Scheduling* in parallel processing is the decision of how many and which applications to activate, and the number of processors that should be allocated to each of them.

Probably the most basic dichotomy in parallel scheduling is the distinction between *single-level* and *two-level scheduling*, where either the application creates a set of parallel threads, and the system is responsible for scheduling them, or the application just requests an allocation of processors and manages them on its own.

A common problem with single-level scheduling is that leaving all the scheduling to the operating system is too expensive and not optimized for the application needs. Instead, with two-level scheduling, the operating system just allocates computing resources, and the application itself or the runtime system does the actual fine-grain scheduling of threads in a way that satisfies the synchronization constraints. This level of internal scheduling provides high flexibility in resource allocation. For example, it is possible to create systems where the processor allocation changes during execution, and the application is expected to adjust accordingly. However, two-level scheduling is not universally accepted. Although perfect for shared memory machines or sufficiently large granularities, using the workpile of chores programming model is less suitable for distributed memory architectures. In that context, SPMD is the prevalent programming style (Feitelson 1994; Feitelson and Rudolph 1995).

Depending on the cost of processor rescheduling, three different scheduling approaches are possible. In *Dynamic Scheduling* there is no restriction on the frequency of processor rescheduling. The scheduler may take a processor away from an executing process and reallocate it to another process of a different job. In *Static Scheduling* a fixed set of processors is allocated to a job. These processors are held by the job throughout its entire execution time and are relinquished only when the job completes. Finally, *Semi-Static Scheduling* allows adjustments to the processors allocated to a running application only at specific events, like the arrival or departure of other applications. The scheduler must be aware of the cost associated with processor rescheduling and restrict such activities so that the improvement in performance due to careful scheduling is not offset by the incurred cost.

2.4 General Results

An early and fundamental result on the scheduling of parallel machines was presented by Ousterhout (1982). Fine-grain coordination between processes requires *Co-scheduling*, where all processes of a job are executing simultaneously on different processors[1]. In the *Matrix Algorithm*, the processes are organized as a matrix, with each column containing processes of one processor and each row containing co-scheduled processes. In *Continuous*, the processes are organized as a sequence. Scheduling is done by sliding across the sequence a window of length equal to the number of processors in the system. The *Undivided* method differs from the Continuous, by requiring the processes of a job to be contiguous in the linear process sequence. The Matrix and the Continuous methods suffer from internal and external fragmentation, respectively.

Eager et al. (1989) express formally lower and upper bounds on the speedup function using the average parallelism of the application and the number of allocated processors. They try improvements in the speedup bounds by using the job sequential fraction and the maximum parallelism of the application, and they investigate the impact on efficiency of allocating additional processors to an application. Also, the knee of the execution time-speedup curve is introduced, corresponding to the point where the benefit per unit cost is maximized. An estimate of the exact location of the knee is given along with bounds based on the average parallelism. Finally, several performance measures are estimated for processor allocations equal to the knee point and the average parallelism. It is shown that such allocations achieve a perfect balance between speedup and efficiency.

Ways of characterizing the parallelism in an application are discussed by Sevcik (1989). It is concluded that for static space sharing it is enough to know the minimum length (total execution time when the application has ample processors allocated) and the shape (indicating the proportions of time that the application would use various numbers of processors). Then it is examined how a few parameters that are derivable from the shape might capture a sufficient amount of information to do effective resource allocation. It is validated that the average parallelism alone

[1]Another term that is also mentioned is *Gang Scheduling* (Feitelson and Rudolph 1992a; Feitelson 1994). According to Feitelson, Co-sheduling was originally defined to describe a system attempting to schedule a set of threads simultaneously on distinct processors, but when it could not, resorting to scheduling simultaneously only a subset of the threads. In Gang Scheduling, threads interacting at fine granularity should be scheduled together and grouped into a gang. Thus, Gang Scheduling allows guarantees about the performance to be given.

is a good initial rule of thumb (Eager et al. 1989). Then is is shown that a more sophisticated rule would include the system load for applications with highly variable parallelism. Thus the appropriate allocation would be equal to one at very heavy load and to the total number of processors at very light load. A more advanced rule that uses information about individual applications would allocate a number of processors equal to the minimum and maximum parallelism of the application at heavy and light loads respectively.

Tighter bounds on the speedup function can be obtained when the execution time is known at a number of processors equal to the average parallelism of the application (Majumdar et al. 1991). Analytic models are used to show the importance of both the application characteristics and the load in determining the partition size of the jobs. The execution time-efficiency knee is found to yield almost optimal performance for a wide range of applications and loads. Also, semi-static policies are studied, that change the job partition size according to the system load or the execution behavior of the jobs. The significance of a low processor reallocation overhead is stressed for the success of such policies.

Sevcik (1994) considers workloads composed of computationally intensive applications where it is appropriate to dedicate each processor to one application. If the scheduling objective is to minimize the average response time, then the Least Work (or Smallest Cumulative Demand) First heuristic is proven useful. When only expected rather than exact total processing time requirements are known and processor allocation is done statically, then the Least Expected Work First is still the best rule. If priorities of applications differ, then the weighted average response time can be kept low by activating applications in increasing order of their ratio of expected work to priority weight. For heavily loaded systems, it is necessary to maximize the efficiency of use of the processors. At the extreme, this leads to allocating very few (even one) processor per application. At light loads, processors may be used at lower efficiency in order to complete the applications quickly.

2.5 Scheduling in UMA Shared-Memory Systems

The basic principles underlying the performance of scheduling strategies in multiprogrammed parallel systems are investigated by Majumdar et al. (1988). In *First Come First Served (FCFS)*, each process joins a FIFO queue and is serviced by the next released processor, while in *Round Robin (RR)* each process receives a quantum of service at a time. Also, policies based on job characteristics are defined. In *Smallest Number of Processes First (SNPF)*, a free processor is allocated to the job with the smallest number of waiting processes, while in *Smallest Cumulative Demand First (SCDF)* the cumulative demand of jobs is considered instead of the parallelism level. The policies are compared with respect to mean response time. Both FCFS and RR perform poorly, because the system is monopolized by large jobs, and they are outperformed by characteristics-based policies. In particular, the SCDF follows the optimality principle of favoring the jobs with the smaller demand and as expected outperforms SNPF. However, it is encouraging that the latter follows closely behind, because it allows a larger number of jobs to run simultaneously and tends to reduce the number of partially completed jobs in the system.

The work by Tucker and Gupta (1989) is motivated by the observation that parallel applications are degraded in performance, when the total number of processes exceeds the total number of available processors. Processor time may be wasted due to busy-waiting in order to obtain locks held by suspended processes, waiting for data from suspended producer processes, context switching overhead, and processor cache corruption. In the proposed solution, called *Process Control*, each application gets an equal fraction of the processors and dynamically controls the number of runnable processes, so that each runnable process will get its own processor. If an application is broken up into a number of tasks and processes select the tasks from a queue, a process can be suspended after it has finished executing a task and before it has selected another.

Zahorjan and McCann (1990) use simulation to compare scheduling policies with respect to response time. In *Run-To-Completion (RTC)*, if the job finds idle processors on arrival, it is allocated the minimum of the number of free processors and its maximum parallelism. Released processors are allocated to waiting jobs for which the expected improvement in elapsed execution time is greatest, with special care that as many jobs as possible are activated. The *Round-Robin (RR)* policy is based on Co-scheduling. A new process row is created only if this increases the average number of processors usefully busy. In *Dynamic*, at job arrival the processor request is satisfied by either idle processors or a single processor taken from a job with more than one. Upon release of processors, priority is given to jobs with no processors. It is shown that Dynamic is best for small context switching overheads and its advantage increases with increasing load or larger and more rapid changes in the workload parallelism. On the whole, RTC is preferable to RR for general purpose use.

Leutenegger and Vernon (1990) introduce a new scheduling policy, called *Round Robin Job* (as opposed to *Round Robin Process*), where a shared job queue is maintained and each entry contains a queue with the processes of a single job. Scheduling is done round robin on the jobs, and at each turn, each job gets a number of quanta equal to the number of processors in the system. It is compared to other policies with respect to mean response time. It is concluded that the Smallest Number of Processes First (Majumdar et al. 1988) does not perform well for workloads with coefficient of variation of service demand larger than 1 or 2. Also, Round Robin policies have better performance than the Co-scheduling policies. The minimization of spin-waiting by Co-scheduling appears to improve the performance at very high lock demands, but even then not as effectively as the handling of variance in job demand by Round Robin. The Round Robin Job and Process Control (Tucker and Gupta 1989) policies, which allocate an equal fraction of the processing power to each job, perform best under nearly all workload assumptions considered.

Black (1990) describes the parallel application scheduling support of the Mach OS. The scheduler accepts application-specific information from users about which virtual processors should or should not be running. Actually, a thread should avoid communicating or synchronizing with another thread that is not running, if the first knows the identity of the second thread. There are two classes of hints that can be used. The *Discouragement Hints* indicate that the current thread should not run. They are useful for synchronization in applications with process multiplexing at the system level. The *Handoff Hints* indicate that a specific thread should run instead of the current one. One use of this technique addresses the *priority inversion* problem, when a low priority thread holds a lock needed by a high-priority thread.

Gupta et al. (1991) examine how the synchronization primitives and the OS scheduling policies affect the system efficiency. The performance of a *Priority Scheduler* (single run-queue and the inverse of CPU usage as priority) is found to be very poor due to spin-waiting for processes suspended in the critical section. The use of a blocking synchronization primitive increases the performance significantly with the observation that the process must busy wait for a short time before being suspended. The processor utilization was even better with Gang Scheduling though some losses were reported due to cache flushes. The usefulness of Handoff Scheduling with respect to system throughput is found to be limited. Process control is the best policy, as the use of fewer processors decreases the performance lost due to load balancing, lowers the synchronization costs, and increases the spatial locality of the application. The performance is better than Gang Scheduling, as there are no cache flushes.

Another comparative study is presented by McCann et al. (1993). The *Dynamic* policy is introduced where processors are reallocated among jobs in response to changes in the parallelism of the jobs [2] (Zahorjan and McCann 1990). Also, the *Equipartition* policy is defined, which maintains an equal allocation of processors to all jobs. Reallocations take place only on job arrival and completion. The response time of Dynamic is compared with that of Round-Robin Job and Equipartition on a shared-memory multiprocessor. It is shown that space sharing is preferable to time sharing, because most applications have sublinear speedup functions and their efficiency decreases with the number of processors allocated to them. Dynamic

[2] In user-level thread packages, upon starting execution an application forks a kernel-schedulable thread per physical processor. This thread, called a virtual processor, alternates between running one of two kinds of user-level thread: scheduler threads, executing code provided by the thread package and initiating application threads, and application threads, executing code provided by the application and running until they either block or terminate.

allocation is proved preferable to static allocation and preempting processors in a coordinated way is critical. Furthermore, it is remarked that cache locality is not a deciding factor and is outweighed by performance improvements due to efficiency. The Dynamic Scheduling policy, characterized by space sharing, coordinated preemption and dynamic reallocation outperforms both the Equipartition and Round Robin Job policies.

2.6 Scheduling in NUMA Shared-Memory Systems

The impacts of context switching, preemption, processor affinity and processor sharing are explored on a NUMA multiprocessor by Crovella et al. (1991). A form of dynamic policy is introduced, called *Hardware Partitions*, that dynamically allocates processors equally among the jobs in the system and permits several threads from the same application to share a processor[3]. Uncoordinated Time Sharing is compared to both Co-scheduling and Hardware Partitions. It is shown that programs that synchronize infrequently or don't use barriers, are immune to the effects of time sharing. Co-scheduling performs significantly worse than Hardware Partitions, because the latter avoids cache corruption. In addition, it reduces remote references, contention and load imbalance by allocating fewer processors per job. These factors are more than enough to compensate for the costs of migration and the overhead of blocking required within a hardware partition, when the number of threads exceeds the number of available processors.

The *Processor-Pool Based Scheduling* for NUMA multiprocessors is introduced by Zhou and Brecht (1991). In contrast to hardware clusters of processors, processor pools are used as an operating system construct for application scheduling. A large number of processors are divided into groups, called pools. The parallel threads of a job are run in a single processor pool, unless there are performance benefits for a job to span multiple pools. Several jobs may share one pool. Simulation experiments show

[3]It was first introduced by Gupta et al. (1991). The name *Processor Sets* was used by Chandra et al. (1994).

that processor pool-based scheduling may effectively improve the average response time. The performance enhancement attained increases with the average parallelism of the jobs, the load level of the system, the differences in memory access costs and the likelihood of having system bottlenecks.

The performance benefit of Gang Scheduling is investigated by Feitelson and Rudolph (1992b). A thread is called *blocked*, when it is suspended from execution and the processor switches to another thread. With fine-grain jobs, Gang Scheduling with busy-waiting outperforms Uncoordinated Time-Shared Scheduling with blocking due to the overhead of blocking, which is only compensated for in unbalanced coarse-grained jobs. When Gang Scheduling is used and fine-grain applications need more threads than the available number of processors, it is suggested that threads should be grouped into gangs so that the majority of the interactions do not cross gang boundaries (*Interaction Locality*).

The relative performance of scheduling policies is reconsidered by Chandra et al. (1994) by running real applications on a CC-NUMA multiprocessor. First, it is reported that a combination of cache and cluster affinity along with automatic page migration can improve considerably the response time over a standard Unix Scheduler. Then Gang Scheduling is compared to the Process Control policy and *Processor Sets*, which is equivalent to Hardware Partitions (Crovella et al. 1991). Gang Scheduling, because of its co-scheduling property, provides the illusion of an exclusive machine and allows the programmer/compiler to successfully perform data locality optimizations. The dynamic nature of the other two policies make such optimizations difficult. Time multiplexing in Gang Scheduling and Processor Sets lead to cache interference. Also, a parallel application executes more efficiently with fewer processors by reducing the penalties of communication, synchronization and load imbalance. This, usually called the Operating Point Effect, is exploited by the Process Control policy. It is concluded that the performance of the different scheduling algorithms is a tradeoff among the relative importance of data distribution optimizations, cache interference and the operating point effect of the particular applications.

2.7 Scheduling in Distributed-Memory Systems

The *execution signature* of a parallel application is defined as the rate of execution with respect to the number of allocated processors, the system architecture, and the program implementation (Park and Dowdy 1989). It can be estimated by applying a least-squares approximation method on runtimes for different numbers of processors. An analytical expression is derived for the system throughput as a function of the number of processors, the execution signature parameters, the host speed, the interference overhead from shared-resources contention among different applications, and the reconfiguration overhead for changing the partition of the applications during execution. With low reconfiguration overhead, dynamic partitioning leads to maximum throughput, while with high reconfiguration overhead, dynamic partitioning should be avoided. The conclusions are validated using real applications on a distributed-memory system (Dussa et al. 1990).

Processor Working Set (pws) is defined as the minimum number of processors that maximizes the speedup per unit of a cost function incorporating the number of processors and the associated speedup. Several space sharing policies based on the *pws* are defined and compared, using characteristics of real applications. It is shown that a simple work conserving strategy that attempts to allocate the minimum of the number of free processors and the job *pws* is robust and offers good throughput-response time characteristics over a wide range of arrival rates (Ghosal et al. 1991).

Setia and Tripathi (1993) introduce the *Adaptive Static Partitioning (ASP)* policy. If a job finds idle processors on arrival, it is allocated the minimum of the idle number of processors and its maximum parallelism. When processors are released, they are divided equally among the waiting jobs, under the constraint that no job gets more than its maximum parallelism. Comparison to the Processor Working Set policy shows that, in general, ASP performs better and knowledge of program characteristics like the *pws* is useful only at moderate loads, where much simpler policies like ASP, perform similarly well.

In the study by Setia et al. (1993) it is observed that dynamic policies, entail considerable delays and cannot be applied to distributed memory systems due to both migration of data and application reconfiguration overhead. They introduce a new policy, called *Adaptive Multiprogrammed Partitioning (MP)*, which attempts to increase processor utilization by combining time sharing with Adaptive Static Partitioning. The policy is compared to pure Adaptive Static Partitioning. Both simulation and analytical modeling show that MP achieves better response time, particularly at heavier loads and larger partition sizes. This is attributed to the ability of the MP policy to overlap the communication and waiting time of a process with the computation time of other processes. The MP policy is further investigated by Setia et al. (1994).

Trends in supercomputing centers suggest environments in which jobs with very different resource requirements arrive at various intervals to systems with large numbers of processors (Naik et al. 1993). In *Fixed Partitioning*, processors are divided at system configuration time into a fixed number of disjoint sets. The *Fixed Partitioning with Job Priorities* is similar with the additional requirement that each partition is designated as belonging to a certain class of jobs. Larger job classes cannot acquire partitions belonging to smaller job classes, while small jobs have non-preemptive priority over medium jobs for large job partitions. In *Dynamic Partitioning*, applications are capable of reconfiguring themselves in response to requests from the OS, though the reconfiguration overhead is ignored in the study. This reconfiguration is realized in two phases. In the first phase, the entire data set representing the current state is transferred from all processors in the old partition to a subset of processors in the new partition. In the second phase, the data set is then redistributed from this subset to all processors in the new partition. The analysis demonstrates that the scheduling policy should reduce the number of processors allocated to each job with increasing load, up to a minimum number of processors, in order to provide good response time for the entire workload. Also, it must distinguish between jobs with large differences in execution times.

Rosti et al. (1994) investigate several adaptive policies that are particularly appropriate for implementation in distributed memory systems. In *Fixed Processors* *per Job*, the processor set is divided into a fixed number of equal sized partitions. In *Equal Partitioning with a Maximum*, the set of currently free processors is equally divided among the jobs in the waiting queue with a Maximum partition per job set as a configuration parameter. The *Insurance Policy*, instead of using maximum, always reserves a percentage of the available free processors for later job arrivals depending on the system load. In the *Adaptive Policies*, at each instant the system is in a certain state, which identifies the ideal partition size. The transition from one state to another is governed by the load on the system, as measured by the queue length. The policies are compared using the ratio of throughput to response time across several workload types and arrival rate levels.

Chiang et al. (1994) use simulation to demonstrate that policies based on the application average parallelism or the processor working set can be outperformed by *Adaptive Static Partitioning-Max*, where *Max* is a limit in the maximum job allocation. In addition, *Shortest Demand First-Max*, derived from Shortest Demand First (Majumdar et al. 1988; Sevcik 1994) with *Max* being defined as before, is shown to outperform Adaptive Static Partitioning-Max. Generally, a fixed maximum allocation limit that could be weakly dependent on the system load, though independent of the application characteristics, is claimed to be very important for run-to-completion policies. Also performance is degraded when too many processors are given to long jobs, and application characteristics, other than the total demand, are poor predictors of this situation.

The simplicity of current scheduling policies in message passing multicomputers and the possibility of improvement due to recent software and hardware advances is highlighted by McCann and Zahorjan (1994). In one proposed family of policies, called *Folding*, a newly loaded job is allocated a partition of processors obtained by dividing the largest currently allocated partition in half, with the threads running on those processors folded onto the remaining processors allocated to their job. To achieve equal long-term allocation of resources, the Folding policies periodically reallocate processors among the running jobs. The other family, called *Equipartition*, reallocates processors as equally as possible whenever a job arrives or departs, but makes no other reallocations. Folding emphasizes efficiency preservation over equality of resource allocation and Equipartition does the opposite. The Folding policy is proved to yield better response time, at little or no penalty in fairness.

2.8 Classification of Scheduling Policies

A basic classification of parallel scheduling schemes is based on the way computing resources are shared. The time sharing dimension is a direct extension of current practices on uniprocessors. It is divided into mechanisms that apply to each processor individually and mechanisms that handle a group of processors as a single unit. Mechanisms for independent processors are further divided into those that use Local Queues, requiring mapping of threads to processors, and those that use a shared Global Queue, combining mapping with scheduling. Mechanisms that perform time sharing on groups of processors actually implement Gang Scheduling and require coordinated context switching across the processors, which is harder to implement. A basic feature that results from time sharing is that, at job submission, the user can specify an arbitrary number of required processes (typically up to the maximum parallelism of the job), independently of the available processors in the system.

Independent Time Sharing:

Round Robin Process. When a job arrives, each of the processes is placed at the end of the shared process queue. A round-robin scheduling policy is invoked on the process queue (Majumdar et al. 1988; Leutenegger and Vernon 1990; Crovella et al. 1991).

Round Robin Job. A shared job queue is maintained and each entry is a queue itself containing the processes of an individual job. Each time a job comes to the front, it gets a number of quanta equal to the number of processors in the system. When the number of processes is less than the number of processors in the system, the service duration of each is proportionately extended. When the processes of a job exceed the number of processors in the system, each process gets service of one quantum at a time, and the processes are served in a round-robin fashion. Its advantage over Round Robin Process is that it prohibits jobs with more processes from getting extra service (Leutenegger and Vernon 1990; McCann et al. 1993).

Coordinated Time Sharing:

Matrix Co-Scheduling. The processes are organized as a matrix, with each column containing processes assigned to one processor and each row containing co-scheduled processes (Ousterhout 1982; Crovella et al. 1991; Gupta et al. 1991; Chandra et al. 1994).

Continuous Co-Scheduling. The processes are organized as a sequence. Scheduling is done by sliding across this sequence a window of length equal to the number of processors in the system (Ousterhout 1982).

Undivided Co-Scheduling. Its difference from the Continuous version is its additional requirement that the processes of each job be contiguous in the process sequence (Ousterhout 1982; Leutenegger and Vernon 1990).

Round-Robin Policy. It is based on Matrix Co-Scheduling with the additional requirement that a new process row is created only if this improves the average number of processors usefully busy (sum of speedup values divided by the number of rows). Otherwise a new job is fitted into the row with the most unallocated slots (McCann and Zahorjan 1989; Zahorjan and McCann 1990).

The space sharing dimension, which is more common in distributed memory machines, partitions the processors into disjoint sets allowing each job to execute in a distinct partition. In *Fixed Partitioning*, the partition sizes are set in advance by the system administrator. Typically, each job is informed about the number of processors it has been allocated and spawns an appropriate number of processes. In *Variable Partitioning*, the nodes are partitioned according to the number of processors each user requires, or according to some application characteristics, assumed known to the scheduler. With *Adaptive Partitioning*, the partition sizes are automatically set by the system according to the current load and probably some application characteristics, always including the job maximum parallelism. In *Dynamic Partitioning*, the size can change at runtime to reflect changes in requirements and load. The system is periodically informed about the current parallelism of the jobs and takes the appropriate actions, while the applications are informed about their current partition size and are expected to behave accordingly, as well. This constrasts with the other types of space-sharing, where the scheduler is activated only at job arrivals and departures.

Fixed Space Sharing:

Fixed Partitioning. At system configuration time, the processors are divided into a fixed number of disjoint sets of equal size. (Ghosal et al. 1991; Setia et al. 1993; Setia and Tripathi 1993; Naik et al. 1993; Rosti et al. 1994).

Fixed Partitioning with Job Priorities. It is based on Fixed Partitioning with the additional requirement that each partition is designated to a certain class of jobs. Large jobs cannot acquire partitions belonging to small jobs, while small jobs have non-preemptive priority over medium jobs for partitions of large jobs (Naik et al. 1993).

Variable Space Sharing:

First Come First Served. When a job arrives, it is allowed to spawn an arbritrary number of processes, which join a FIFO queue. Whenever a processor is released, the process at the head of the queue is taken up for service (Majumdar et al. 1988; Leutenegger and Vernon 1990).

Smallest Number of Processes First. A free processor in the system is allocated to a process which belongs to the job with the smallest number of processes not yet allocated any processor. Ties are broken in favor of the job that arrived earliest. A job can fork an arbitrary number of processes (Majumdar et al. 1988; Leutenegger and Vernon 1990).

Smallest Cumulative Demand First. A free processor in the system is allocated to a process which belongs to the job with the smallest total service time. A job spawns an arbitrary number of processes (Majumdar et al. 1988; Sevcik 1994; Chiang et al. 1994).

Processor Working Set. The *processor working set* of a job is defined as the minimum number of processors that maximizes the speedup per unit of a cost function that incorporates the number of processors used and the associated speedup. The best *pws*-based variation maintains a FIFO job queue and allocates to each job the minimum of its processor working set and the number of processors currently available (Ghosal et al. 1991; Setia and Tripathi 1993; Chiang et al. 1994).

Average Parallelism. A FIFO job queue is maintained and a job is allocated a number of processors equal to the minimum of its average parallelism and the number of processors currently available (Eager et al. 1989; Sevcik 1989; Chiang et al. 1994).

Adaptive Space Sharing:

Adaptive Static Partitioning. If a job finds free processors on arrival, it is allocated the minimum of its maximum parallelism and the number of free processors. When processors are released, they are divided equally among the waiting jobs (Setia and Tripathi 1993; Naik et al. 1993). A job maximum allocation limit may be included as a configuration parameter (Rosti et al. 1994; Chiang et al. 1994).

Insurance Policy. It is similar to the previous policy. Although there is no maximum allocation limit, only a fraction of the currently available processors is equally divided among the waiting jobs, with the remaining processors being reserved for future arrivals (Rosti et al. 1994).

Adaptive Policies. The system is in a certain state, which defines the ideal partition size. The transition from one state to another is determined by the queue length of the waiting jobs. Actually, different transition rules lead to slightly different policies (Rosti et al. 1994).

Adaptive Multiprogrammed Partitioning. An increase in processor utilization is attempted by combining Adaptive Static Partitioning with interleaving multiple processes from the same job on each node (Setia et al. 1993).

Dynamic Space Sharing:

Process Control. Processors are dynamically allocated equally among the jobs in the system. No application is allocated more processors than its current parallelism. The applications dynamically control the number of runnable processes to match the number of processors available to them. Until the readjustment, the processes of the preempted processors are running round robin on the remaining processors of the application (Tucker and Gupta 1989; Leutenegger and Vernon 1990; Gupta et al. 1991; McCann et al. 1993; Chandra et al. 1994; Chiang et al. 1994).

Dynamic Scheduling. It is similar to the previous policy. However, when processors are preempted, it is the responsibility of the application to decide immediately which *single* threads will run on the remaining processors. Additionally, job priorities decrease with processor usage over time, and thus proper cooperation between the applications and the scheduler is ensured (Zahorjan and McCann 1990; McCann et al. 1993).

Equipartition. An equal allocation of processors to all jobs is maintained. Reallocations take place only on job arrivals and completions. An application does not change its processor partition immediately upon the scheduler request, but at the next "convenient point", which can appear arbitrarily later (McCann et al. 1993; McCann and Zahorjan 1994).

Dynamic Partitioning. This is an implementation of Process Control for distributed-memory systems. Applications must be capable of reconfiguring themselves in two phases. First, the entire data set representing the current state is transferred from all processors in the old partition to a subset of processors in the new partition. Then, the data set is redistributed from this subset to all processors in the new partition (Naik et al. 1993).

Processor Sets. The processors are allocated equally among the jobs in the system. Increased multiprogrammed workload leads to squeezing of applications on fewer processors and multiplexing of multiple processes on the same processor (Gupta et al. 1991; Crovella et al. 1991; Chandra et al. 1994).

Chapter 3

System and Workload Specification

In this chapter, the system framework that our study refers to is introduced. Assumptions about the hardware configuration and the main operating system features are presented, along with the expected workload requirements.

3.1 System Environment

As has already been mentioned, the main objective of this study is to clarify the impact that different approaches to the scheduling problem have on the average response time of a workstation network. It is important that our system assumptions are independent of the actual programming paradigm used. It is decided by the programmer according to the nature of the application whether a shared-memory, message-passing or data-parallel paradigm is used.

The concentration on workstation clusters is merely an effort to take into account the increased overhead that distributed-memory systems incur under dynamic processor reallocations among running jobs. In such an environment, the cost of a pure dynamic scheduling algorithm may include data transfers and is considerably more than just context switches and cache interference, as is usually assumed in shared-memory systems. This restricts considerably the applicability of dynamic scheduling (Park and Dowdy 1989; Dussa et al. 1990), and motivates the use of the system load and the application characteristics for improving the scheduling decisions, and hence the response time.

3.1.1 Hardware Structure

We consider a collection of nodes, each of which is an autonomous computer consisting of a processor and a local memory. The individual nodes are considered equivalent in processing capacity, therefore issues of hardware heterogeneity within the distributed system are ignored. This is a simplifying assumption that often does not hold in practice, but which permits concentration on the basic properties of the various scheduling policies.

The nodes are interconnected by a network. Different applications can run on separate partitions without degradation in performance due to contention in the communication medium. Experiments conducted on a 32-node *iPSC/2* justify this assumption (Leuze et al. 1989). Also, independence between different running jobs can be considered reasonable for high bandwidth communication media, like those based on ATM. Though the assumption may not hold in the case of a heavily loaded *Ethernet*-based installation, the present investigation concentrates on basic scheduling issues and does not aim to optimize the mixes of possibly interfering jobs.

In general, we assume that pairs of nodes can communicate with each other with negligible latency differences. In addition, the existence of dedicated file servers is presumed to guarantee that jobs can be executed on any node subset, without predictable discrepancies in code or data transfer delays. We ignore details of the networking technology and topology.

3.1.2 Software Structure

An important assumption about the software part of the environment we study is that the operating system can support two-level scheduling. We expect that the scheduler decides only the number of processors to allocate to each application. It is the responsibility of the application to determine how the allocated processors will be used and whether multiple threads will be interleaved on each individual processor or not. The programmer decides which choice fits better the computation, synchronization and communication needs of the problem (Setia et al. 1993).

The scheduling decisions are made at a single node, where a central queue of job requests is maintained. Centralized algorithms that can take advantage of the clustering properties in small or large scale distributed systems, have been proven successful in recent resource management environments (Zhou et al. 1993) without restricting the system scalability, as it was traditionally claimed.

Also, we expect that the scheduling system guarantees that the parallel applications are kept apart from the sequential workload of the system, whether they are daemons or applications initiated by the workstation owners. However, it is an open question in the distributed system literature, if this can be done efficiently, whether by migration of the running parallel threads, by remote execution of the sequential jobs or by properly coordinated execution of both in the same machine (Arpaci et al. 1994). Despite the importance of the issue, we will not examine it in the present study.

Nevertheless, we believe that it is reasonable to treat the sequential workload as separate problem. Its execution does not induce communication and synchronization requirements between different processors, and it demands at most one processor for each application.

3.2 Workload Description

This study aims to address the parallel scheduling requirements of general-purpose workstation clusters. We assume that each application is assigned a number of processors between one and a maximum number that can used appropriately. Time constraints, memory requirements or even debugging procedures may entail minimum allocation limits for some applications, which makes the above assumption non-realistic. However, the need for understanding the general scheduling problem led us to ignore such cases.

In addition, we make no distinction between interactive and batch workload. All requests for multiple processor execution are treated similarly, and the dispatching time is determined according to the specifications of the scheduling disciplines. The main objective of the scheduler is the minimization of the mean response time provided by the system, since the rationale behind the introduction of parallel processing is the need for decreasing the runtime of large time-consuming applications.

Also, we expect that all kinds of parallel jobs are acceptable by the system regardless of speedup quality, parallel programming paradigm, service demand or runtime behavior. It is the responsibility of the scheduler to recognize the useful characteristics of the applications and any other workload parameters, and to use such information in the processor allocation decisions, if this is necessary for improving the performance. In this study, we investigate how different disciplines can exploit such information.

Chapter 4

The Scheduling Policies

In the literature review, the scheduling policies that have been proposed previously were described, and information about their comparative performance was given. Based on the results of previous studies, we identified those algorithms that demonstrated the best performance.

In comparisons for distributed memory systems, the policies that have been proven most effective are the Adaptive Static Partitioning (Setia and Tripathi 1993), the Adaptive Policies (Rosti et al. 1994), and the Shortest Demand First (Sevcik 1994; Chiang et al. 1994).

Although Dynamic Partitioning (Naik et al. 1993) was proven to perform well, the result is workload-dependent when the repartitioning overhead is taken into account (Park and Dowdy 1989; Dussa et al. 1990). However, we include it here with the name *Dynamic*.

Finally, the idea of optimally distributing the free processors among the jobs waiting in the queue was proven to improve the performance (Park and Dowdy 1989; Zahorjan and McCann 1990; Wu 1993; Sevcik 1994). The name *Optimal*, will be used for this.

In the present chapter, these five scheduling algorithms are defined more formally using pseudo-code, and details are given for the workload information that the operation of each is based on. Also, a new adaptive policy is introduced, with the name *Adaptive Equipartition (AEP)*, along with several composite policies based on combinations of the previous ones.

4.1 Load and Application Parameters

Before defining the algorithms that will be studied, it is necessary to clarify the types of workload information that will be expected to be available to the scheduler.

Information about the system load that is available to the scheduling policy includes the number of the jobs that have arrived to the system, but have not been dispatched yet, and the number of jobs currently running. The values for these two parameters can be maintainted by the scheduler in a straightforward way. The names `waiting_jobs` and `running_jobs` respectively are used for them in the following sections.

With respect to the applications, we assume that the execution time at one processor *(Demand)* or at any number of processors is available to the scheduler. Also, a maximum processor requirement may be specified by the user as part of the job submission. Typically, this is assumed never to exceed the number of processors p_{max}, beyond which the application runtime increases due to the communication overhead.

In general, the runtime of a job cannot be known until its completion. However, Feitelson and Nitzberg (1995) conclude that statistics gathering for the runtime of jobs is feasible. Actually, it is shown that the majority of jobs that were run on the same partition size by the same user exhibited a runtime distribution with coefficient of variation less than one. Furthermore, the predictive power can be improved by using more sophisticated job behavior models than just the mean of previous runs (Devarakonda and Iyer 1989; Wu 1993). Still the feasibility of implementing a scheduling algorithm to make full use of the information available remains to be proven.

An execution time model that Wu (1993) used, was introduced by Sevcik (1994):

$$T(p) = \phi(p)\frac{W}{p} + \alpha + \beta p. \tag{4.1}$$

The parameter p is the number of processors allocated to the job and W the essential computational work. The parameter $\phi(p)$ represents the degree to which the work is not evenly spread across the p processors, and α stands for the increase of the work per processor due to parallelization. Finally, β represents the communication and congestion delays that increase with the number of processors.

It is proven by Wu (1993), that the execution time functions of real applications, with varying sizes and structures, can be represented accurately by the model above. A least-squares approximation method is applied, and the only requirement is the runtime of the application for different numbers of processors. A few points are usually enough for an acceptable formulation. Also, it is shown how the simpler model $T(p) = \frac{W}{P} + a$ (Park and Dowdy 1989) contains less information and fails to approximate adequately the execution time curve of several real applications.

From equation 4.1 it is possible to derive the point p_{max} (Sevcik 1994) that was mentioned above:

$$p_{max} = \begin{cases} \sqrt{\frac{W}{\beta}} & if\ \beta \neq 0 \\ \infty & if\ \beta = 0 \end{cases} \tag{4.2}$$

4.2 The Adaptive Static Partitioning Policy

The first algorithm that used system load only in order to achieve effective scheduling was introduced by Setia and Tripathi (1993) with the name Adaptive Static Partitioning. Under this policy, if a job arrives when there are idle processors, it is allocated the minimum of its maximum parallelism and the number of available processors. If no processors are available, the job waits. When a job completes, the released processors are divided equally (as possible) among the waiting jobs, with no job taking more than its maximum parallelism.

Thus the policy adapts to the load of the system. At low loads, the number of processors allocated to a job will tend to be close to its maximum parallelism, while at high loads the partition sizes will be smaller. A more formal description of this policy, which will be referred to here as ASP, is given in figure 4.1.

```
%Initially all allocations to waiting jobs are zero
%The jobs are maintained in the waiting queue in FIFO order

target ⟵ max(1, ⌊ free_processors / waiting_jobs ⌋)

if (target*waiting_jobs < free_processors)
  excess ⟵ free_processors - target*waiting_jobs
else
  excess ⟵ 0
while (waiting_jobs>0  and free_processors>0)  do
  remove J from the head of the queue of waiting jobs
 if (excess > 0)
    allocation of J ⟵ min(target+1,p_max of J)
    excess ⟵ excess - 1
 else
    allocation of J ⟵ min(target,p_max of J)
 fi
  free_processors  ⟵ free_processors - allocation of J
dispatch J
od
```

Figure 4.1: Definition of the Adaptive Static Partitioning (ASP).

4.3 The Adaptive Policy 1

A different idea of using the system load for partitioning processors among waiting jobs was introduced by Rosti et al. (1994). A system state i defines the current partition size in the system. When jobs arrive and depart such that all partitions are busy, and there is a steady queue of as many waiting jobs as partitions, each job will be scheduled on a partition of size i. If the queue length surpasses the number of partitions, it may be beneficial for the policy to *split* by passing to some new state j, $j < i$. Similarly, if several jobs depart the system without arrivals, leaving idle partitions, it may be beneficial for the policy to *merge* by going to some state k, $k > i$, so that the next arriving job will be allocated more processors. The allocator is called whenever a job arrives and there are free processors, or whenever a job departs and there are queued jobs. As previously, the processor allocation to a job does not change during the execution of the job.

This approach was proven to increase the robustness of the scheduler, that is, the ability to perform well over wide ranges of arrival rates and workload types. Several algorithms based on the idea above are introduced by Rosti et al. (1994). They differ in the way that the transitions from one state to another are defined.

The variation that is included in our study, was called AP1. Its pseudo-code description is given in figure 4.2. It is evident that whenever the queue length increases, the target size decreases (split), and whenever the queue length decreases, the target size increases (merge).

```
%Initially all allocations to waiting jobs are zero
%The jobs are maintained in the waiting queue in FIFO order

target_size ⟵ round(total_processors / waiting_jobs)

while (waiting_jobs>0  and free_processors>0)  do
  remove J from the head of the waiting queue
  temp ⟵ min(target_size,p_max of J)
  allocation of J ⟵ min(free_processors,temp)
  free_processors ⟵ free_processors - allocation of J
  dispatch J
od
```

Figure 4.2: The Adaptive Policy (AP1).

We found out that it is always to the benefit of the system to dispatch a job when $0 < free\_processors < target$. In the original definition, such a final allocation of the remaining processors was not done.

4.4 The Adaptive Equipartition Policy

In this section, we introduce a slightly different rule for adaptive partitioning. Intuitively, the ideal allocation is to divide the processors in the system equally among all the running and waiting jobs. By definition, this cannot be done in a non-preemptive policy.

However, we can use as a target the ratio of the total processors and the total

```
%Initially all allocations to waiting jobs are zero
%The jobs are maintained in the waiting queue in FIFO order

target_size ⟵ max(1, ⌊ total_processors / (waiting_jobs+running_jobs) ⌋)

while (waiting_jobs>0 and freeprocs>0) do
  remove J from the head of the waiting queue
  temp ⟵ min(target, p_max of J)
  allocation of J ⟵ min(free_processors,temp)
  free_processors ⟵ free_processors - allocation of J
  dispatch J
od
```

Figure 4.3: Definition of the Adaptive Equipartition (AEP).

number of jobs, waiting and running, in the system. The name that we will use is Adaptive Equipartition (AEP), inspired from the motivation above. The policy is defined in figure 4.3.

In order to clarify the difference between AEP and the previous policies, assume that all jobs in the system are of the same type and therefore can be described with the same execution time function, $T(n)$, where n is the number of allocated processors.

Then the processor occupancy for each job is $n\,T(n)$, if the parallelization overheads are taken into account. Thus the *actual load* in the system ρ_a would be :

$$\rho_a = \frac{\lambda\, n\, T(n)}{P}, \tag{4.3}$$

where λ is the arrival rate and P the total number of processors in the system. Then by applying Little's law we have:

$$N = \lambda\, R(n),$$

where N is the total number of jobs in the system and $R(n)$ the response time of the job. From these two last equations we get :

$$n\,\frac{T(n)}{R(n)} = \rho_a\,\frac{P}{N}$$

In addition, $R(n) = W(n) + T(n)$, that is, the response time $R(n)$ is the sum of the waiting time $W(n)$ and execution time $T(n)$. If we set $n = \frac{P}{N}$, which is equal to the *target* in AEP, we obtain:

$$W(n) = \frac{1-\rho_a}{\rho_a}\,T(n) \tag{4.4}$$

Alternatively, we may combine equation 4.3 with the following equation:

$$L = \lambda\,W(n)$$

where L is the queue length of the waiting jobs:

$$n\,\frac{T(n)}{W(n)} = \rho_a\,\frac{P}{L}$$

If we set $n = \frac{P}{L}$, which is equal to the *target* in AP1, we get:

$$W(n) = \frac{1}{\rho_a}T(n) \tag{4.5}$$

Thus, the main difference between rules AEP and AP1 is captured respectively from the relations $\frac{1-\rho_a}{\rho_a}$ and $\frac{1}{\rho_a}$ that describe how the waiting time $W(n)$ changes with respect to $T(n)$ at different load levels.

Actually, at high loads we expect that $\rho_a \longrightarrow 1$. Then, policy AP1 will keep $W(n)$ at a non-zero value $T(n)$, while the waiting time under AEP will tend toward zero, as

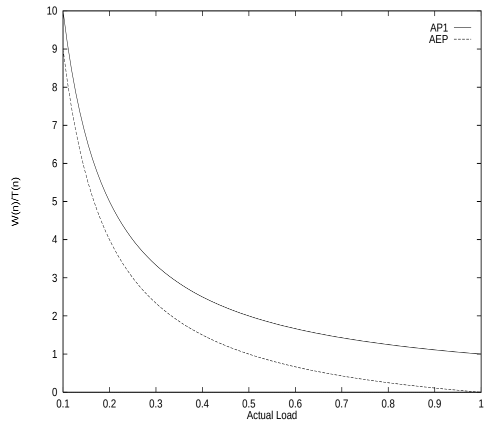

Figure 4.4: The ratio $\frac{W(n)}{T(n)}$ for AEP and AP1.

follows from equations 4.4 and 4.5, respectively. The exact difference in the behavior of these two equations is depicted in figure 4.4. The AEP policy will display a more conservative behavior than AP1 by giving fewer processors to the jobs at higher loads.

4.5 The Shortest Demand First Policy

A non-preemptive policy that exhibits good performance is the one called Shortest Demand First (SDF). The arriving jobs are kept in the queue ordered according to increasing total demand, that is, the execution time on one processor. When processors are released, jobs in the queue are allocated the minimum of their specified request (maximum parallelism) and the number of idle processors (Majumdar et al. 1988; Chiang et al. 1994). The definition of SDF in pseudo-code is shown in figure 4.5.

There are specific cases where the above policy demonstrates optimal performance. For instance, in the case of jobs with perfect speedups and in the absence of arrivals, the scheduling strategy that minimizes the average response time in the system is SDF with all processors being given to each application in turn (Sevcik 1994).

In addition to SDF as defined above, combinations of SDF with the previously defined adaptive policies can be introduced. For example, ASP can keep the waiting

```
%Initially zero allocation is assumed for the waiting jobs
%waiting jobs are kept ordered according to increasing total
%demand (execution time on one processor)
while (freeprocs>0) do
  remove J from the head of the waiting queue
  allocation of J ⟵ min(freeprocs,p_max of J)
  free_processors ⟵ free_processors - allocation of J
  dispatch J
od
```

Figure 4.5: The Shortest Demand First.

jobs in non-decreasing order of their total demand. This composite policy will be referred to as ASP-SDF. Similarly, we introduce the composite policies AP1-SDF and AEP-SDF, which are the AP1 and AEP respectively, with the waiting jobs queued in non-decreasing order of their demands.

4.6 The Optimal Allocation Policy

The issue of determining an optimal way for allocating the free processors among the waiting jobs in a system has been investigated in several studies (McCann and Zahorjan 1989; Park and Dowdy 1989; Wu 1993; Sevcik 1994).

Actually, use of the job execution time as function of the number of processors allows the formulation of a non-linear integer constrained optimization problem with the objective of minimizing the total response time of the jobs under consideration. In particular, the non-preemptive variation of this idea tries to minimize the objective function for those jobs that are currently waiting in the queue. If we assume m waiting jobs, with $T_j(n)$ being the execution time function of job j, $p_{j,max}$ the corresponding maximum parallelism, p_j the unknown processor allocation to job j, and N the number of available processors, then we have the following formulation:

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{m} T_j(p_j) \\ \text{subject to} \quad & \sum_{j=1}^{m} p_j \leq N \\ & p_j \in \{0, 1, 2, \ldots, p_{j,max}\}, \quad j = 1, 2, \ldots, m. \end{aligned} \tag{4.6}$$

Efficient solution of this problem in the general case is difficult. The specific property that makes it tractable in the present context is the assumption that the execution time function is convex.

We can prove the convexity of our chosen execution time function,

$$T(p) = \phi(p)\frac{W}{p} + \beta p + \alpha \tag{4.7}$$

with $\phi(p) = 1$, which is justified by the experiments of Wu (1993). We need to prove that:

$$T(\lambda\, x + (1-\lambda)\, y) \;\leq\; \lambda\, T(x) + (1-\lambda)\, T(y)$$

where $\lambda \in [0, 1]$. Equivalently we have that:

$$\begin{aligned} \frac{W}{\lambda\, x + (1-\lambda)\, y} + \beta\,(\lambda\, x + (1-\lambda)\, y) + \alpha \;\leq \\ \lambda \left(\frac{W}{x} + \beta\, x + \alpha\right) + (1-\lambda)\, \left(\frac{W}{x} + \beta\, y + \alpha\right) \end{aligned}$$

It is enough to show that:

$$\frac{1}{\lambda\, x + (1-\lambda)\, y} \leq \lambda\, \frac{1}{x} + (1-\lambda)\, \frac{1}{y}$$

which is equivalent to :

```
%Based on RTC (Zahorjan and McCann 1990)
%Initially zero allocation is assumed for the waiting jobs
%The jobs are maintained in the waiting queue in increasing
%order of T(allocation+1)-T(allocation) with T(0)=∞
%Ties are broken in favor of the jobs that arrived earlier
while (waiting_jobs>0  and free_processors>0)  do
  let J be the job at the head of the waiting queue
  allocation of J ⟵ allocation of J + 1
  free_processors ⟵ free_processors - 1
  if (allocation of J = p_max of J)
    dispatch J
  else
    move J in the waiting queue according to the
    new difference T(allocation+1)-T(allocation)
od
dispatch all jobs in waiting queue with allocation>0
```

Figure 4.6: Definition of the Optimal processor allocation

$$\left(\frac{x}{y} - 1\right)^2 \geq 0$$

It can be proven that when the convexity assumption is satisfied, the problem can be solved in polynomial time (Ibaraki and Katoh 1988). Another solution is described by Sevcik (1994). Here we will use a simpler algorithm which finds the optimal solution in exponential runtime of the input size. The correctness proof is given by Ibaraki and Katoh (1988), Park and Dowdy (1989) and also in the appendix of McCann and Zahorjan (1989). An estimation of the computational complexity is also given by Ibaraki and Katoh (1988).

An important problem that the above formulation cannot capture is the number of new jobs that should be activated at each release of processors, and the exact order with which they will be chosen. In one previous study, the waiting jobs were FIFO ordered and at each step as many jobs as possible were activated (Zahorjan and McCann 1990). Wu (1993) assumed that the jobs are SDF ordered and as many jobs as possible are activated at each step. Finally, Park and Dowdy (1989) assume that the processors are always allocated to jobs according to the difference $T(p+1)-T(p)$, including running jobs and jobs with zero allocation ($p = 0$).

In fact, the response time of a job that has received no processors can be considered infinite, due to the unknown waiting time and partition size of the jobs. It may be beneficial for the system performance (at low loads) to activate a few jobs with large partitions and leave the remaining jobs in the waiting queue, or, alternatively, to activate as many jobs as possible even with one processor each (at high loads). Thus, determination of the order and number of jobs to activate justifies special consideration.

Actually, the strategy of activating as many jobs as possible coincides with the adaptive policy ASP described previously. Thus, the approach of Zahorjan and McCann (1990) can be treated as a combination of ASP and OPT, and that of Wu (1993) as the triple combination ASP-SDF-OPT.

The SDF component of ASP-SDF-OPT refers to the waiting jobs being queued in non-decreasing order of their demand. The scheduler is called at job arrivals and departures, and the ASP component decides how many waiting jobs should be activated, as this follows from the number of processors that each job would obtain according to ASP. After the jobs for activation have been determined, the OPT component determines the actual distribution of the free processors among them. Since it is ensured (by ASP) that each of the activated jobs will take at least one processor, the $T(p+1) - T(p)$, for $p \geq 1$ is actually used by OPT.

We also introduce two other composite policies, namely, the AP1-SDF-OPT and AEP-SDF-OPT. In both of them, the order of the waiting jobs remains the same (SDF). However in AP1-SDF-OPT, the number of waiting jobs to be activated follows from the number of processors that each job would obtain according to AP1. Similarly in AEP-SDF-OPT, the number of waiting jobs to be activated is determined by the number of processors that would be allocated to them by AEP. The actual distribution of free processors among the jobs for activation is again decided based on the minimum differences $T(p+1) - T(p)$ for $p \geq 1$, since each such job will receive at least one processor.

4.7 The Dynamic Policy

Despite the variations in the name and the exact implementation with which this policy appeared in the past, it is one of the most common algorithms in the recent multiprocessing scheduling literature. The main reason for this is the excellent performance that it usually demonstrates in shared-memory environments (McCann and Zahorjan 1989; Tucker and Gupta 1989; Leutenegger and Vernon 1990; McCann et al. 1993; Chiang et al. 1994 etc.).

In one of the initial definitions presented by McCann and Zahorjan (1989), the processors are dynamically partitioned equally among the applications in the system. Special provisions are taken so that no application is given more processors than it can use. Processors are forcibly preempted when they have been reallocated to a different job, and immediately the application readjusts the number of running processes (McCann et al. 1993). A description of the policy as used in our experiments is given in figure 4.7.

The policy requires that the applications should be able to adapt to a change in the allocated number of processors during execution. This restricts the range of applications that can benefit from such a policy mainly to those written according to the workpile of chores programming paradigm. Such an application is structured as multiple independent chores, which are kept in a workpile. A number of worker threads are running on distinct processors, and they pick chores from the workpile in an undefined order and execute them. Additional workers may be added at any time, and existing workers may be removed whenever they finish one chore and before they start another. Such changes in the number of workers may change the order in which chores are computed and the rate at which they are completed, but this does not affect the outcome of the computation.

Adapting to a dynamically changing number of processors is possible in other programming models, as well, but it requires significant effort on the part of the application developer. For example, jobs written for distributed-memory machines can adjust to a changing number of processors by redistributing their data structures

```
% Procedure called at job arrivals and departures

% Running jobs are kept in a queue in decreasing order of
% the number of processors they possess.

if (waiting_jobs > 0 )
  remove J from the head of the waiting queue
  allocation of J ⟵ min(p_max of J, free_processors)

 if (running_jobs>0)
   while (p_max of J > allocation of J  and
       allocation of J < allocation of job at running queue head - 1 ) do
       K ⟵ job at running queue head
       allocation of K ⟵ allocation of K - 1
       move K in running queue according to new allocation
       allocation of J ⟵ allocation of J + 1
   od
 fi
  dispatch J if allocation of J > 0
else if (running_jobs>0)
  K ⟵ job at running queue tail
 while (free_processors>0  and there are running
     jobs to receive more processors)  do
   if (allocation of K < p_max of K)
       free_processors ⟵ free_processors - 1
       allocation of K ⟵ allocation of K + 1
       move K in running queue according to new allocation
       K ⟵ job at running queue tail
   else
       K ⟵ previous job in running queue
   fi
 od
fi
```

Figure 4.7: The Dynamic Policy, as we used it in the experiments.

Scheduling Policy	Load Parameters	Application Characteristics
ASP	waiting_jobs	p_{max}
AP1	waiting_jobs	p_{max}
AEP	waiting_jobs, running_jobs	p_{max}
SDF		p_{max}, $T(1)$
OPT	waiting_jobs	p_{max}, $T(p)$ for $p > 0$
DYN	waiting_jobs, running_jobs	p_{max}
ASP-SDF	waiting_jobs	p_{max}, $T(1)$
AP1-SDF	waiting_jobs	p_{max}, $T(1)$
AEP-SDF	waiting_jobs, running_jobs	p_{max}, $T(1)$
ASP-SDF-OPT	waiting_jobs	p_{max}, $T(p)$ for $p > 0$
AP1-SDF-OPT	waiting_jobs	p_{max}, $T(p)$ for $p > 0$
AEP-SDF-OPT	waiting_jobs, running_jobs	p_{max}, $T(p)$ for $p > 0$

Table 4.1: The policies and the information they need.

(Naik et al. 1993; Nedeljkovic and Quinn 1993; Carriero et al. 1995). This involves considerable overhead and the price of reconfiguration may outweigh the benefits of changing the allocation (Park and Dowdy 1989; Dussa et al. 1990).

In figure 4.1, a summary of the policies is presented, and the type of workload information each of them uses is shown. The *waiting_jobs* and *running_jobs* parameters correspond to the numbers of waiting and running jobs, respectively, while $T(p)$ corresponds to the execution time of the application with p allocated processors.

Chapter 5

Simulation Model

We have used simulation modeling to compare the performance of the scheduling policies presented in the previous chapter. Simulation gave us the flexibility of covering a wide range of application characteristics and arrival rates, and allowed us to abstract away unimportant details of the environment under study, which otherwise would complicate the evaluation procedure.

In the sections that follow, we describe the characteristics of our model, along with the method that was used for representing the workload.

5.1 System Model

In our experiments, we concentrate on a system with the general features presented in Chapter 3, and $P = 32$ independent nodes. In the effort to isolate the inherent properties of the different scheduling policies, we assume that the effect of the memory requirements is included in the shape of the job execution time functions. For the same reason, we assume that communication or synchronization latencies are implicitly represented in the execution time functions of the jobs.

Also, preemption and scheduling overheads are ignored. In this way, the performance of Dynamic policy is maximized and thus can serve as benchmark (unattainable in practice) for the remaining policies. The other algorithms are non-preemptive, and therefore the delays from processor reallocations are insignificant. The computational needs of the schedulers are also assumed to be negligible. This is valid for the number of nodes typically involved in workstation clusters.

5.2 Workload Model

A wide range of representative applications can be modeled by utilizing the execution time function introduced by Sevcik (1994):

$$T(p) = \phi(p)\frac{W}{p} + \alpha + \beta p. \tag{5.1}$$

It is evident that, by choosing different values for the parameters $\phi(), W, \beta$ and α, we obtain descriptions of jobs with different characteristics and inherent structures. The detailed structure description (for example, McCann and Zahorjan 1989) with all the associated difficulties is no longer necessary. The function allows generation of job pools with all the predefined properties, and gives precise control of the workload definition. Below, we clarify the relationship between the characteristics of the jobs and the few parameters of the execution time function.

5.2.1 The Work Imbalance

The parameter $\phi(p)$ has been taken equal to one, since the real measurements conducted by Wu (1993) indicate that it can be treated as constant and never exceeds the value 1.14.

5.2.2 The Essential Work

It could be claimed that the most important of all the parameters is the computational work W. Its value determines the minimum total service demand of the job (both β and α are typically smaller (Wu 1993)) and therefore the variation of service required among different jobs. We tried to make our simulated workload realistic by using the statistics gathered for a 128-node *iPSC/860* message-passing system at the NASA Ames Center by Feitelson and Nitzberg (1995) (as described in Chapter 2).

Job Size	Runtime (secs)	Number of Jobs	Total Demand	Average Demand Per Job
1	140	28800	0.14K	
2	714	1750	1.5K	
4	1116	3700	4.5K	1.3K
8	705	1800	5.6K	
16	569	1800	9K	
32	1305	3700	42K	
64	2350	1200	150K	101K
128	3280	500	420K	

Table 5.1: Service demands for a two-class representation of the data.

The first and second columns appear as presented by Feitelson and Nitzberg (1995). The third one contains the number of occurrences, as derived from a bar chart. The total demand is taken by multiplying the runtime by the number of allocated processors. The horizontal line separates the jobs into two separate classes. This decision follows from the observation (Feitelson and Nitzberg 1995) that jobs with 32, 64 and 128 nodes use more than 90% of the system resources (node-seconds), while their individual demands are almost one order of magnitude larger than those with sizes 1-16. Thus it is reasonable to treat these two subsets of jobs as separate classes.

The average demand for the small jobs is equal to $1.3K$, while that for the large ones $101K$. In addition, large jobs account for $\frac{1}{8}$ of the workload with small jobs composing the other $\frac{7}{8}$.

Feitelson and Nitzberg (1995) conclude that runtimes are hyper-exponentially distributed. Thus, the generation of the W values is done by using a 2-stage hyper-exponential distribution, as in other simulation studies (Chiang et al. 1994; Parsons and Sevcik 1995).

The exact cumulative distribution function used for W is :

$$F(W) = 0.125 \times (1 - e^{-W/101}) + 0.875 \times (1 - e^{-W/1.3})$$

The corresponding mean value is 13.76 and the coefficient of variation is 3.5.

5.2.3 The Maximum Parallelism

As noted previously, the maximum parallelism of a job is given by the following equation:

$$p_{max} = \begin{cases} \sqrt{\frac{W}{\beta}} & \text{if } \beta \neq 0 \\ \infty & \text{if } \beta = 0 \end{cases}$$

Thus, we can control the maximum parallelism of the job with proper choice of the value of β, since W has already been defined. If we require

$$p_{max} = \sqrt{\frac{W}{\beta}},$$

then β must satisfy the following relation:

$$\beta = \frac{W}{p_{max}^2}. \tag{5.2}$$

For the experiments, we used the above equality in order to keep the maximum processor allocation of the jobs equal to 4, 16 or 64, each with equal probability. These values correspond to the 12%, 50% and 100% of the processors in the system (32). The uniform distribution of jobs with different sizes, that is, the equiprobability assumption, was observed by Feitelson and Nitzberg (1995) in the real workload they studied.

5.2.4 The Job Speedup

The fundamental job characteristic that remains to be defined is the speedup function, $S(p)$, where p is the number of processors. This, when combined with the execution time of the job on one processor, $T(1)$, can define completely the execution time of the job, $T(p)$, for any other number of processors.

The speedup function can be derived from equation 4.1, as follows:

$$S(p) = \frac{T(1)}{T(p)} = \frac{W + \beta + \alpha}{\frac{W}{p} + \beta\, p + \alpha} \tag{5.3}$$

Dividing both the numerator and denominator by W, we get:

$$S(p) = \frac{1 + \frac{\beta}{W} + \frac{\alpha}{W}}{\frac{1}{p} + \frac{\beta}{W}p + \frac{\alpha}{W}}$$

or by replacing $\frac{\beta}{W}$ with $\frac{1}{p_{max}^2}$

$$S(p) = \frac{1 + \frac{1}{p_{max}^2} + \frac{\alpha}{W}}{\frac{1}{p} + \frac{1}{p_{max}^2}p + \frac{\alpha}{W}}.$$

Thus the speedup curve has been expressed as a function of the maximum parallelism p_{max} and the ratio $\frac{\alpha}{W}$. This means that for a specific value of the p_{max} the speedup curve will remain the same across jobs with different essential work W, provided the ratio $\frac{\alpha}{W}$ does not change value by proper choice of α.

We take one step further by setting, for some $\mu \in R \cup \{+\infty\}$ and assuming $p_{max} > 1$:

$$\alpha = W \left(\frac{1}{p_{max}^2}\right)^{\mu} \qquad \mu \in R \cup \{+\infty\}$$

or

$$\frac{\alpha}{W} = \left(\frac{1}{p_{max}^2}\right)^{\mu} \qquad \mu \in R \cup \{+\infty\}$$

where the $\mu = \infty$ corresponds to $\alpha = 0$. Thus, the $S(p)$ becomes:

$$S(p) = \frac{1 + \frac{1}{p_{max}^2} + \left(\frac{1}{p_{max}^2}\right)^{\mu}}{\frac{1}{p} + \frac{1}{p_{max}^2}p + \left(\frac{1}{p_{max}^2}\right)^{\mu}} \qquad \mu \in R \cup \{+\infty\} \tag{5.4}$$

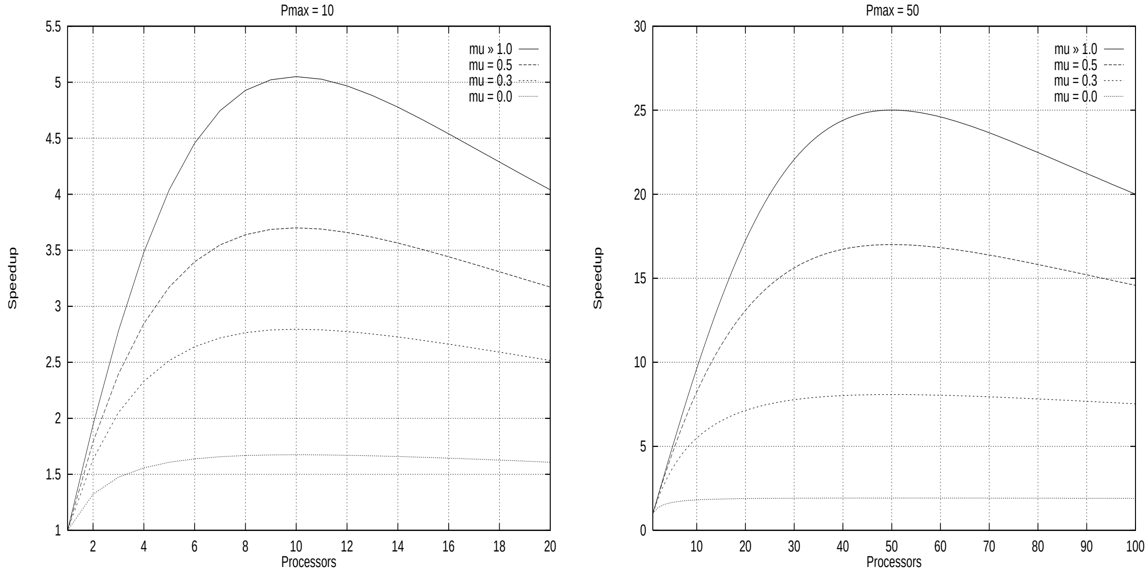

Figure 5.1: Speedup curves when $p_{max} \in \{10, 50\}$ and $\mu \in \{0.0, 0.3, 0.5, +\infty\}$.

From equation 5.4, we see that the speedup curve is completely defined by the values of p_{max} and μ. Examples for $p_{max} \in \{10, 50\}$ and $\mu \in \{0.0, 0.3, 0.5, +\infty\}$ are shown in figure 5.1, with the curve $\mu >> 1$ corresponding to $\mu = \infty$.

A few important conclusions from the figure are:

1. The speedup with $p = p_{max}$ hardly exceeds $\frac{p_{max}}{2}$, if $\beta \neq 0$. Actually, this last value is reached when $\mu \to +\infty$.

2. The speedup is better for large μ, with the best quality reached when $\mu \to +\infty$.

3. The same values of μ seem to have similar effects on the slopes of the speedup curves for different values of p_{max}.

The first remark can be proven if we let $\mu \to +\infty$ and $p = p_{max}$ in equation 5.4, assuming that $\beta \neq 0$:

$$S(p) = \frac{1 + \frac{1}{p_{max}^2}}{\frac{1}{p_{max}} + \frac{1}{p_{max}^2} p_{max}}$$

or equivalently,

$$S(p) = \frac{1 + p_{max}^2}{2p_{max}} \approx \frac{p_{max}}{2} \tag{5.5}$$

This means that the behavior of function 4.1 makes it impossible to keep the speedup linear at number of processors close to p_{max}, even for $\alpha = 0$ if $\beta \neq 0$

The third conclusion is a strong indication that we may generate jobs with similar speedup curve characteristics for jobs with different W and p_{max} values, if we use the same value for μ.

Equations 5.2 and 5.4 give us control of the distributions of the maximum parallelism and speedup curve shape of the represented applications. Of course, absolute control is not possible due to the interdependencies of these properties. However, we have managed to avoid random combinations of the parameters W, β and α. That would lead to workloads with arbitrary distributions of job characteristics and, therefore, unclear effects.

In the experiments we used four different workloads:

1. The workload WK1 consists of curves with relatively good speedup, to the degree that this is permitted by the value of p_{max}. They correspond to $\mu \to +\infty$.

2. The workload WK2 consists of jobs with $\mu = 0.4$, and speedup not as good as in WK1.

3. The workload WK3 consists of jobs with poor speedup, corresponding to $\mu = 0.2$.

4. The workload WK4 contains jobs with all three speedup types, appearing with approximately equal frequency.

Each of the values of $\mu \in \{0.2, 0.4, +\infty\}$ and the mixed case have been combined with values of $p_{max} \in \{4, 16, 64\}$ in order to generate 12 separate job pools. Each workload incorporates three of these job pools, as they are defined by the respective values of μ. The results for the pure speedup types are shown in figure 5.2. These curves are intentionally similar to the speedup curves that were used in the experiments of Rosti et al. (1994) and were derived on a transputer multicomputer using actual applications for different input string lengths. However, in our study, we have three different maximum parallelism values $\{4, 16, 64\}$, instead of the one that was used there (16).

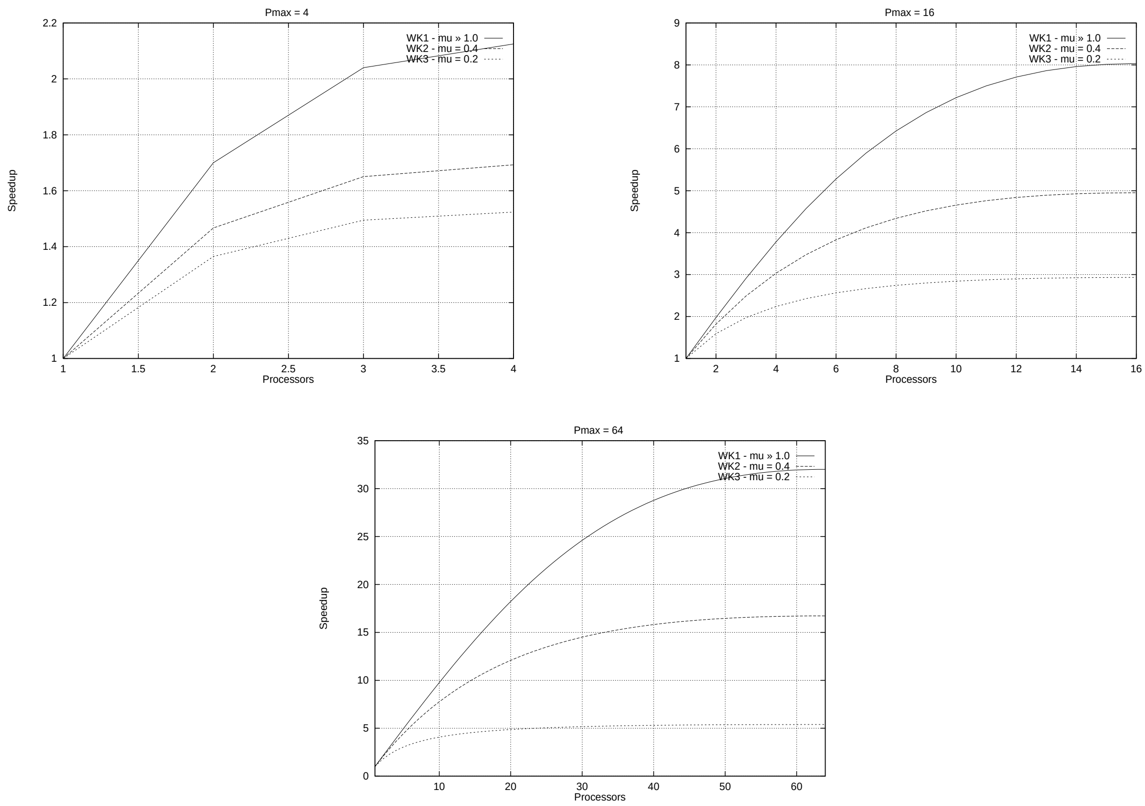

Figure 5.2: Speedup curves for $\mu \in \{0.2, 0.4, +\infty\}$ and $p_{max} \in \{4, 16, 64\}$.

5.2.5 The Arrival Process

In general, the *traffic intensity* or *offered load* of a queuing system with P servers is defined as follows :

$$Load = \frac{Mean\ Service\ Time}{P \times Mean\ Interarrival\ Time}$$

In studies of multiprocessor systems, the $Mean\ Service\ Time$ is set equal to the mean total execution time of the jobs at one processor (Sevcik 1989):

$$Load = \frac{T(1)}{P \times Mean\ Interarrival\ Time}$$

Intuitively, the use of $T(1)$ is reasonable since it is the minimum total execution time of a job, due to the overhead that arises with larger numbers of processors.

In the case of the jobs that have been used in our study, the mean execution time at one processor can be defined as follows:

$$E(T(1)) = E(W) + E(\beta) + E(\alpha)$$

where $E()$ is the expectation operator. We already know that $E(W) = 13.76$. By using the theorem of total expectation (Trivedi 1982) :

$$\begin{aligned} E(\beta) =& E(\beta \mid p_{max} = 4)P(p_{max} = 4) + \\ & E(\beta \mid p_{max} = 16)P(p_{max} = 16) + \\ & E(\beta \mid p_{max} = 64)P(p_{max} = 64). \end{aligned}$$

we finally get $E(\beta) = 0.30$.

Similarly, from:

$$
\begin{aligned}
E(\alpha) = & E(\alpha \mid p_{max} = 4)P(p_{max} = 4) + \\
& E(\alpha \mid p_{max} = 16)P(p_{max} = 16) + \\
& E(\alpha \mid p_{max} = 64)P(p_{max} = 64)
\end{aligned}
$$

it follows that :

$$
E(\alpha) = \begin{cases} 0.0 & if\ \mu = +\infty, \\ 2.0 & if\ \mu = 0.4, \\ 5.0 & if\ \mu = 0.2, \end{cases}
$$

In the case of mixed speedup :

$$
\begin{aligned}
E(\alpha) = & E(\alpha \mid \mu = +\infty)P(\mu = +\infty) \\
& E(\alpha \mid \mu = 0.4)P(\mu = 0.4) \\
& E(\alpha \mid \mu = 0.2)P(\mu = 0.2)
\end{aligned}
$$

and finally $E(\alpha) = 2.34$.

Thus the computation of the system mean interarrival time for a specific *Load* can be calculated as follows:

$$
Mean\ Interarrival\ Time = \frac{E(T(1))}{P \times Load}
$$

An exponential distribution with the above Mean Interarrival Time was used for the generation of the interarrival times in the system. Table 5.2 summarizes the name and value range for all the parameters that were used in our simulation model.

$Parameter$	Description of Values
Processors P	32 of equivalent computing capacity
Execution Time $T(p)$	$\phi(p)\frac{W}{p} + \alpha + \beta p$
Essential work W	$F(W) = 0.125 \times (1 - e^{-W/101}) + 0.875 \times (1 - e^{-W/1.3})$ with $E(W) = 13.76$, and $CoV = 3.5$
Max Parallelism p_{max}	$\in \{4, 16, 64\}$ with $P(4) = P(16) = P(64) = \frac{1}{3}$
Parameter β	$\frac{W}{p_{max}^2}$ and $E(\beta) = 0.30$
Parameter α	$W\left(\frac{1}{p_{max}^2}\right)^{\mu}$
Speedup $S(p)$	$\left(1 + \frac{1}{p_{max}^2} + \left(\frac{1}{p_{max}^2}\right)^{\mu}\right) / \left(\frac{1}{p} + \frac{1}{p_{max}^2}p + \left(\frac{1}{p_{max}^2}\right)^{\mu}\right)$
WK1	$\mu = +\infty$, $E(\alpha) = 0.0$, and $E(T(1)) = 14.06$
WK2	$\mu = 0.4$, $E(\alpha) = 2.0$, and $E(T(1)) = 16.06$
WK3	$\mu = 0.2$, $E(\alpha) = 5.0$, and $E(T(1)) = 19.06$
WK4	$\mu \in \{0.2,\ 0.4,\ +\infty\}$ with $P(0.2) = P(0.4) = P(+\infty) = \frac{1}{3}$, $E(\alpha) = 2.34$, and $E(T(1)) = 16.40$
Interarrival Time t	$f(t) = \lambda\, e^{-t\lambda}$ with $\frac{1}{\lambda} = \frac{E(T(1))}{Processors \times Load}$

Table 5.2: All the system parameters as used in the experiments.

5.3 The CSIM Simulation Package

CSIM is a process-oriented discrete-event simulation package for use with C or C++ programs. Mainly it is a library of routines, which implement all the necessary operations. A modeled system is represented as a collection of CSIM structures and CSIM processes which interact by visiting the structures. Insight into the dynamic behavior of the modeled system is given through a simulated clock. Actually, simulated time passes when processes execute *hold* statements.

The structures provided in CSIM are as follows:

Facilities - consisting of servers reserved or used by processes,

Storages - resources which can be partially allocated to processes,

Events - used to synchronize process activities,

Mailboxes - used for interprocess communications.

Tables - used to collect data during execution,

Process Classes - used to segregate statistics.

```
sim() {
  initialize data structures
  declare CSIM processes and tables
  create("sim")
  call scheduler process
  while (termination condition not met) {
    generate essential computation (W)
    generate maximum parallelism p_max (β)
    generate speedup (α)
    insert job in waiting queue
    signal scheduler
    generate interarrival time int_arv
    hold(int_arv)
  }
}
```

Figure 5.3: Simplified CSIM process that we used for generation of job arrivals.

```
scheduler() {
  create("scheduler")
  while (1) {
    wait for signal
    allocate processors among jobs
    remove jobs from appropriate queues
    call job() for each dispatched job
  }
}
```

Figure 5.4: Simplified CSIM process that we used for the scheduler.

All these structures are used by CSIM processes, which represent the active entities in the CSIM model. There can be several simultaneously active instances of the same process, and each of them appears to be executing in parallel (in simulated time), even though they are in fact executing concurrently on a single processor.

The main process in our CSIM model is called *sim*() (fig. 5.3). It is where the characteristics of the simulated jobs are generated and the arrival process is determined. At every job generation, it signals the scheduler from where the new job will be dispatched, provided free processors are (or can be made) available. The scheduler is implemented as a single separate process called *scheduler*() (fig. 5.4), which is called initially by *sim*() and is activated each time a new job is generated or a running job terminates. Each job is modeled

```
job() {
  create("job")
  put job in running queue
  runtime = T(processor allocation)
  reserve allocated processors
  hold(runtime);
  release allocated processors
  gather statistics
  remove job from running queue
  signal scheduler
}
```

Figure 5.5: Simplified CSIM process that we used for the jobs.

as a separate process called *job*() (fig. 5.5) and is created by *scheduler*() at the moment of dispatching.

Communication among the *sim*() and the *scheduler*() is accomplished through the waiting queue, where the newly arrived jobs waiting to receive some processor partition are stored. External implementation of this basic data structure gave us absolute control in the implementation of the different scheduling policies. Otherwise, we would have to customize the structures and the scheduling disciplines offered by the CSIM package, which would probably require intervention into the source code of the library.

The waiting queue has been implemented as a doubly-linked priority list. The same structure has supported both the FCFS discipline of the adaptive algorithms and the Shortest Demand First discipline. It has been also used for the construction of the running queue, where information for the running jobs of the Dynamic Policy, was stored.

5.4 The Job Behavior Among Different Policies

We attempted to assure consistency in the characteristics of the jobs, across the different forms that they take, in order to drive the different scheduling policies.

Special care was necessary to ensure that the reconfigurable version of a job which would permit change in the number of allocated processors during execution should not require modification in the representation of the applications. In other words, use of the execution time function for the job modeling should not place limits on the search space of our investigation.

The main issue that was raised by dynamic scheduling was the way of accounting for the part of completed simulated computation, when the policy required that the job released processors or reserved more during the simulated execution. The problem is due to the fact that the total execution time is different for different numbers of processors. The approach that we followed was to accumulate the elapsed time of the current partition size as fraction of the total execution time for this partition size.

For example, if a job with execution time function $T()$, ran for t_1 seconds of simulated time using p_1 processors, before it was required to change into p_2 processors, we claim that the fraction of the total work F_{t_1} completed, until this point is equal to:

$$F_{t_1} = \frac{t_1}{T(p_1)}$$

and the remaining runtime R_{t_1} is equal to:

$$R_{t_1} = (1 - F_{t_1})\ T(p_2)$$

In general, if a job was completed in n phases with respective partial runtimes $t_i,\ i \in \{1,\ldots,n\}$ and numbers of processors $p_i,\ ,i \in \{1,\ldots,n\}$, we would have that:

$$\sum_{1 \le i \le n} \frac{t_i}{T(p_i)} = 1.$$

Thus, we can keep the rate of job execution consistent with the respective execution time representation regardless of the scheduling policy examined.

5.5 Generation of Independent Replications

Simulation of any system or process with inherently random components requires a method of generating *random numbers*. This is the name for random variates from the uniform distribution on the interval $[0, 1]$, since variates from all other distributions can be obtained by transforming random numbers.

Due to the significance of the random-number generator for the validity of the simulation model, we decided to use an external random-number generator and not that provided by the CSIM package (which is that of the C language). Thus, we used a C implementation of the algorithm from Marse and Roberts (1983). It is considered to be a well-tested and acceptable generator that should work properly and consistently on virtually any computer apt to be used for serious simulation (Law and Kelton 1991).

Chapter 6

Experimental Results

In this chapter, details of our experiments are presented and explanations are given for the comparative performance among the policies under study. Most of the experimental configurations are examined using the four different workloads defined in Chapter 5. The performance of each policy is given at five separate load levels, that is, 10%, 30%, 50%, 70% and 90%.

Since the main reason for the parallelization of applications is the need to decrease their response time, the *Mean Response Time* will be the main performance measure in our comparisons. However, other parameters are also displayed when it is helpful for better justification of the results.

To summarize the procedure we used, at each replication the first 500 jobs are used to warm up the system and their performance is ignored. The reported statistics correspond to the subsequent 19,500 jobs. The performance of the jobs that arrive after the first 20,000 jobs is omitted. The policy is considered as saturated when the generation of 10,000 additional jobs is reported without the termination of the 19,500 intervening jobs. In this case, the waiting time of the non-terminated jobs has increased too much already, and the mean response time is considered infinite. The number of replications is taken large enough to give a 95% confidence interval with half-length within 5% of the mean response time. All the results correspond to the simulation model appearing in Chapter 5 of the general system introduced in Chapter 3.

6.1 The Inadequacy of Fixed Allocation Limits

In this first section, we study the performance of the Shortest Demand First policy. It is examined both as it was defined in Chapter 4, and combined with partition size limits (Max=1, 2, 6) for performance improvement, as was suggested by Chiang et al. (1994).

In figure 6.1, the mean response time for the four workloads is shown.

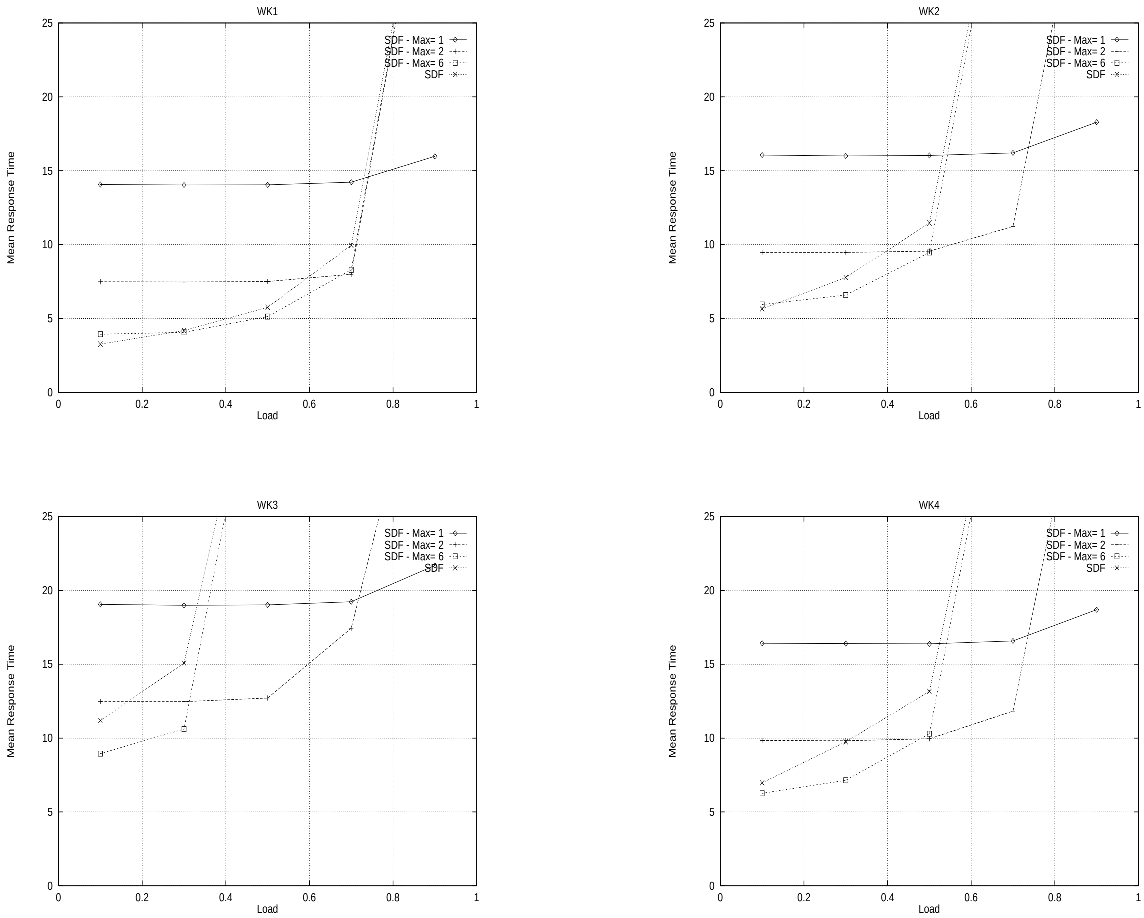

Figure 6.1: Mean Response Time versus System Load for the policy SDF with/without Maximum Allocation Limit.

Those parts of the diagrams that reach the upper border line imply saturation of the policy for the respective load levels. Our main observation is that only the SDF with maximum allocation 1 survives at all loads. However, SDF-Max=1 performs very badly, when compared to other variations of SDF, that are not saturated for the particular load level.

Larger allocation limits cannot provide the necessary adaptability to the policy to keep the response time bounded; this holds even for Max=2. In particular, for good speedup

(WK1), the policies SDF-Max=2, 6, 32 saturate at load 90%. For poor speedup (WK3), even a load of 50% makes the mean response time infinite. The excellent performance reported for SDF by a previous study (Chiang et al. 1994) is illustrated there only for good speedup jobs (WK1) and load 70%. Our experiments do not contradict this specific result.

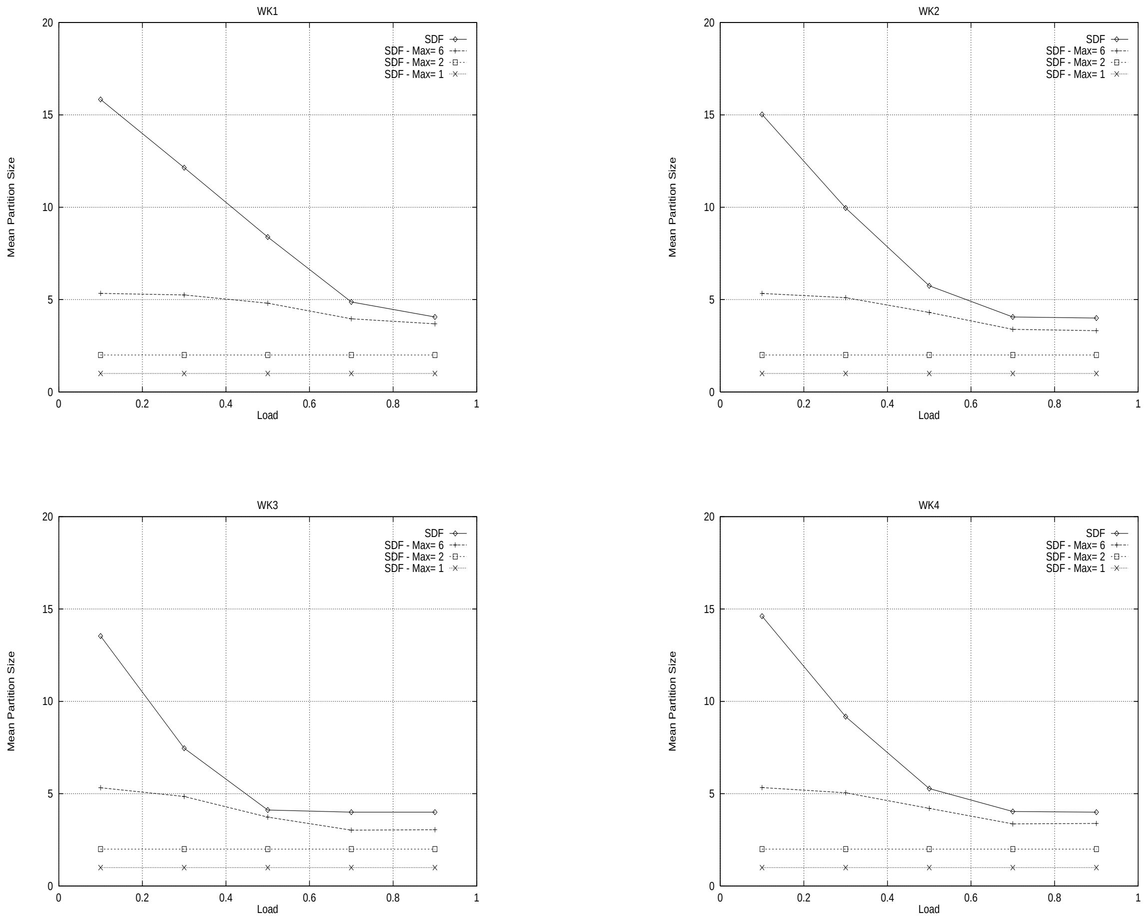

Figure 6.2: Mean Partition Size versus System Load for the policy SDF with/without Maximum Allocation Limit.

A reasonable explanation for these observations is that hard limits at the maximum partition sizes inherently optimize the performance of the policy at a particular workload. However, the optimal points are different for the individual maximum allocations. This means that tuning the parameter Max to the special characteristics of the workload is always necessary. Of course, this cannot provide a solution to the scheduling problem of general purpose machines.

In figure 6.2, we can see the mean partition size for all the reported response times. While the SDF-Max=1 and 2 keep the partition size at constant values of 1 and 2, respectively, the other policies fail to lower their mean partition size at levels significantly less than four. The value four is the minimum p_{max} value of the jobs in the workload (the other two being 16 and 64). This verifies our previous conclusion that the behavior of SDF with any fixed maximum allocation is not robust across all levels of load. It also proves that giving a job the minimum of its maximum parallelism and the current number of free processors, although this provides some flexibility to the policy, is not enough adaptability for a wide range of workload conditions.

6.2 Comparison of the Adaptive Policies

In this section, we will study the comparative performance of the three different adaptive policies, namely ASP, AP1 and AEP, as they were described in Chapter 4.

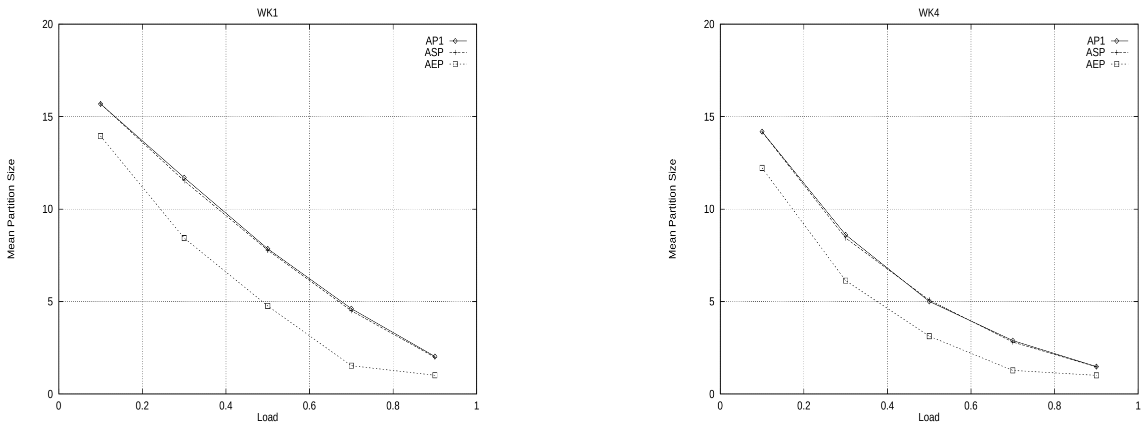

Figure 6.3: Mean Partition Size versus System Load of ASP, AP1 and AEP for WK1 and WK4.

In figure 6.3, we have drawn the mean partition sizes as a function of load for workloads WK1 and WK4. The respective partition sizes of ASP and AP1 are the same. At low loads this could be expected, since for small queue lengths both policies tend to give all the free processors to the next arriving job. Also, it seems that at higher loads the queue length does not become large enough to make AP1 different from ASP.

In figure 6.4 the mean response time of the three policies can be observed. An important conclusion is that the policies ASP and AP1 have almost the same response time for all workload types. These two rules have not been compared previously with respect to mean response time. In the article that introduces AP1, *power*, which is defined as the ratio of throughput to response time, is used as performance measure (Rosti et al. 1994).

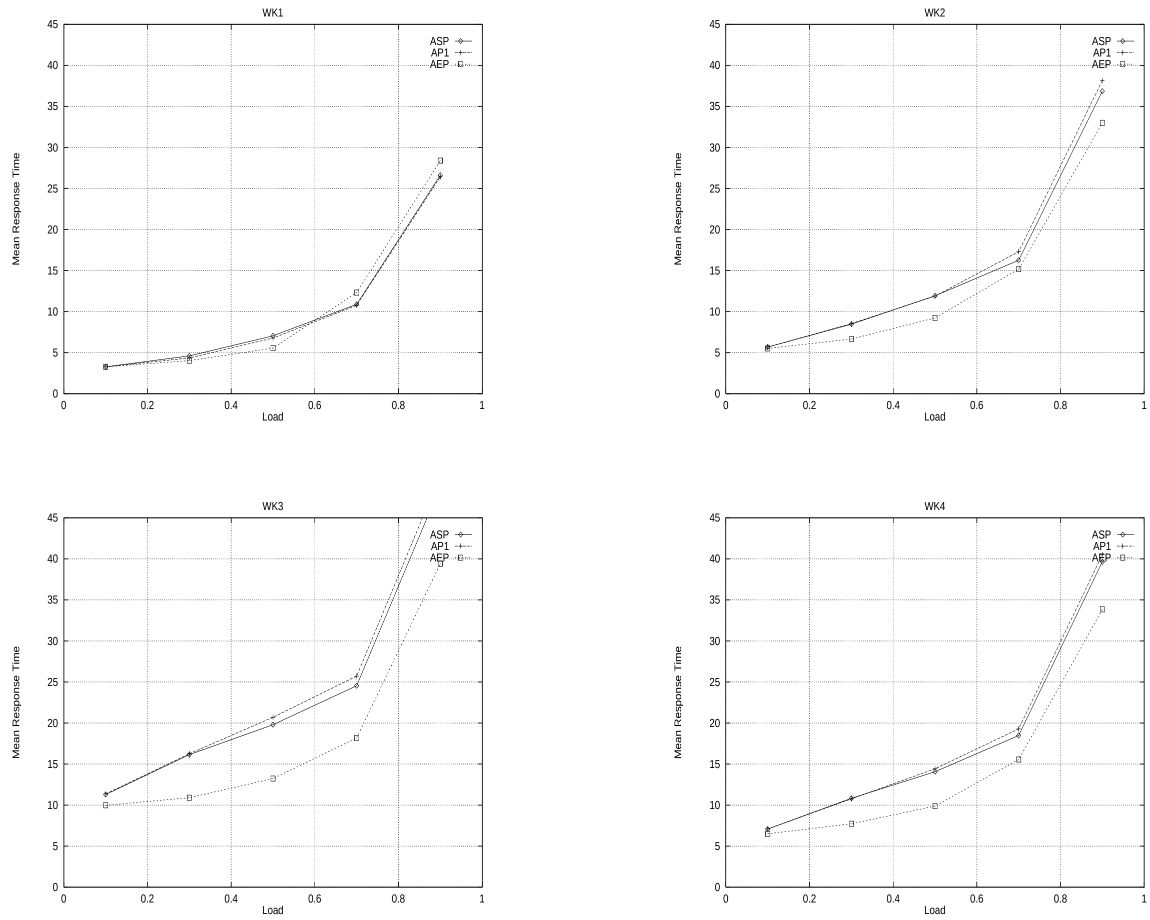

Figure 6.4: Mean Response Time versus System Load for ASP, AP1, AEP.

Another important observation from figure 6.4 is the discrepancy between AEP and the other two policies. Figure 6.5 depicts the waiting time of all three of them with respect to load for workloads WK1 and WK4. This figure verifies that waiting time tends to be smaller in AEP, as was predicted from the analysis in Chapter 4. The only exception is at load 90% and good speedup (WK1). In this particular case, AEP has waiting time equal to that of ASP and AP1, since the jobs are efficient enough to exploit the additional processors given by the other two policies.

However, in the general case as this is captured by workload WK4, AEP correctly decides to provide reduced partition size relative to ASP and AP1, since this improves the mean response time of the system.

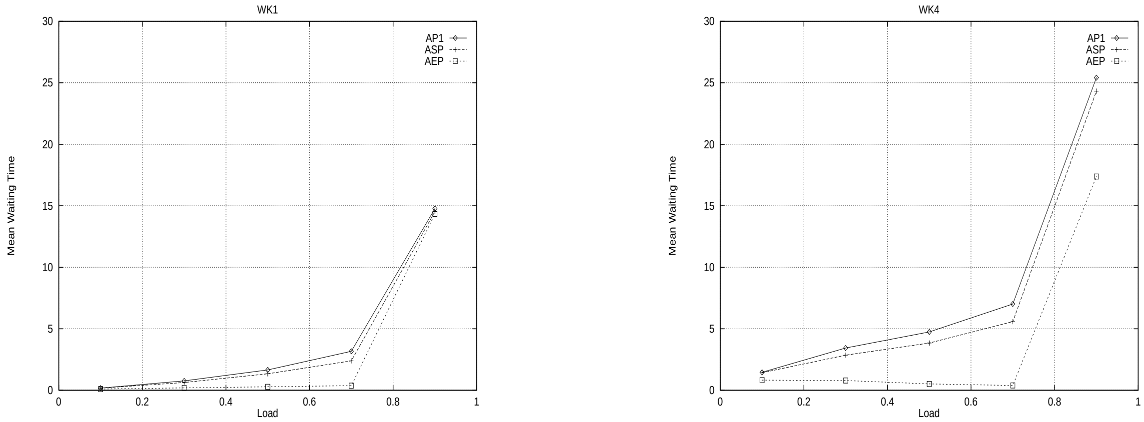

Figure 6.5: Mean Waiting Time versus System Load of ASP, AP1 and AEP for WK1 and WK4.

6.3 Combining the Adaptive Policies with SDF

Taking into account the very good performance that Shortest Demand First demonstrates at the different load levels with the appropriate choice of the Maximum Allocation parameter, we decided to examine SDF combined with the Adaptive Policies. It is already known that the main property of the latter is to automatically adjust the partition sizes using the load information, as determined by the length of the queue formed by the waiting jobs or all the jobs in the system (fig. 6.3).

In figure 6.6, we can see the improvement in the system performance that this combination achieves. At high loads, we get a decreased response time, which in some cases is less than half the response time of the pure Adaptive Algorithms. At very low loads, there is no improvement and the reason is the almost empty machine that the newly arriving jobs find. Of course, it is necessary that a nonzero queue length exists for the queue discipline to have a significant impact on the response time.

In figure 6.7, we can see the mean partition size as function of the load in the system for the workloads WK1 and WK4. Its comparison with that of the previous section (fig. 6.3) proves that SDF incurs no interference to the adaptability of the policies ASP, AP1 and AEP. On the other hand, figure 6.8 proves that there is a remarkable decrease in the waiting time of the jobs at high loads. Thus it is shown that the desired properties of the adaptive policies are orthogonal to those of SDF. Therefore, we have a clear advantage by combining these different features, and not treating them as incompatible strategies.

An important observation from figure 6.6 is the change in the relative performance of

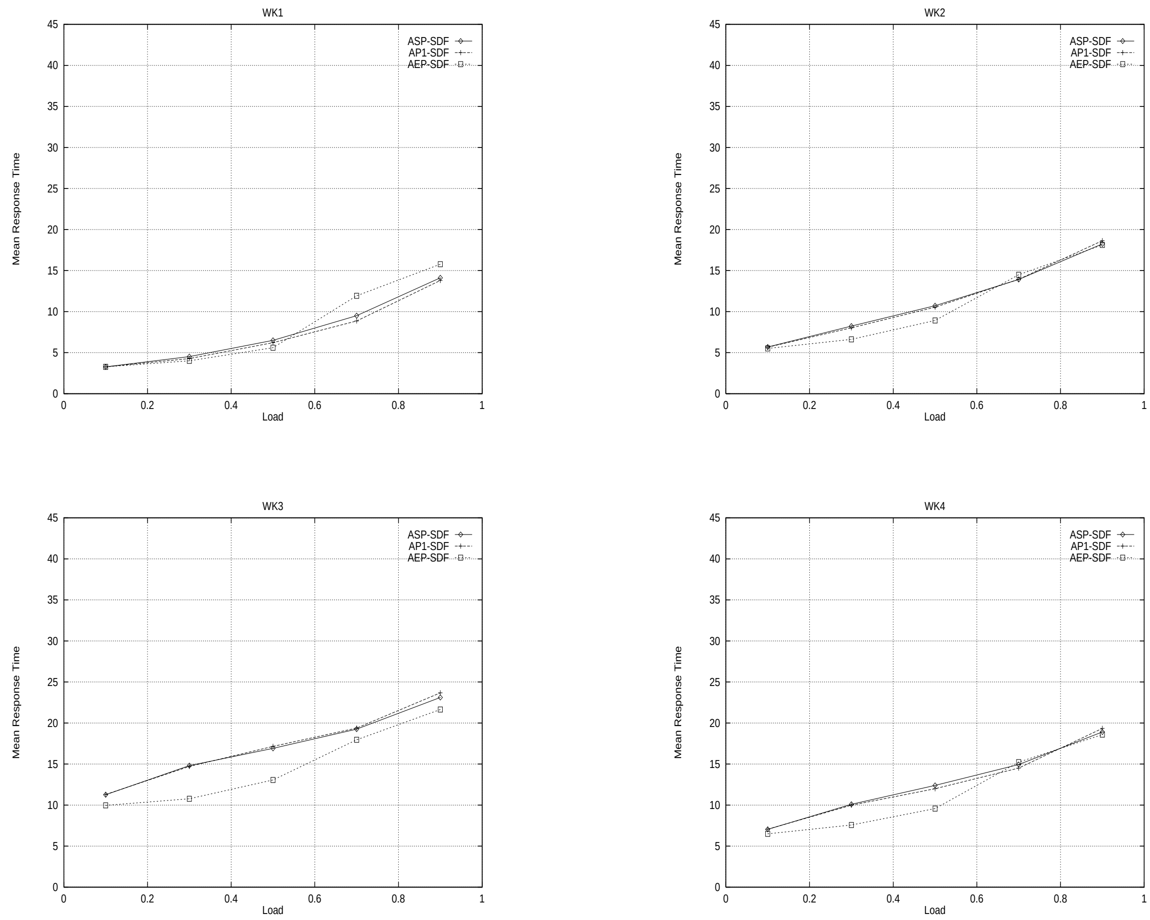

Figure 6.6: The Mean Response Time of the Adaptive Policies combined with SDF.

AEP and ASP or AP1. Although the response times of the last two policies remain identical to each other, it seems that AEP-SDF performs worse at high loads. From figure 6.8 it can be concluded that, due to the reduction in the waiting time from SDF, AEP-SDF loses the advantage, that previously held, of minimizing the waiting time by granting fewer processors than the other two policies.

6.4 Combining the Adaptive Policies with SDF and OPT

The last enhancement that we add to the adaptive policies is the optimal distribution of released processors among waiting jobs. As we already remarked in Chapter 4, the greedy approach of dispatching as many waiting jobs as possible, is not intuitively the

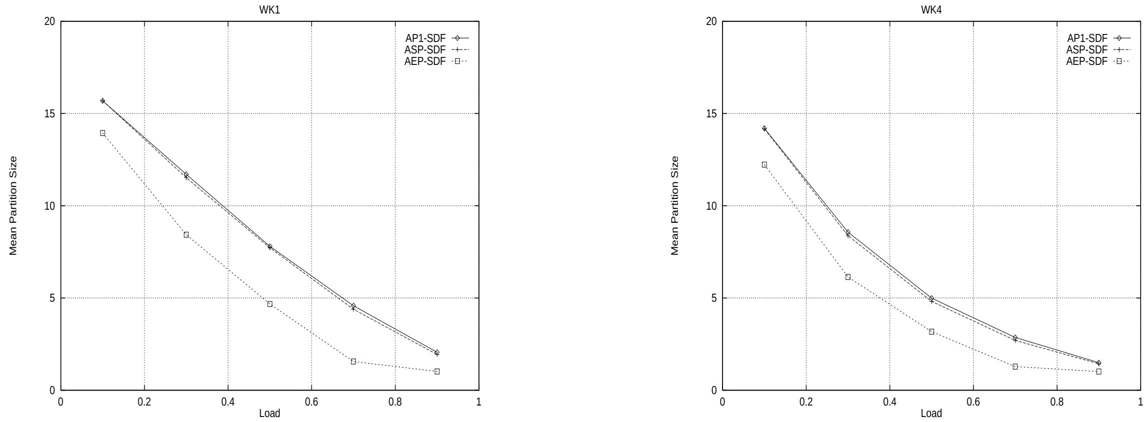

Figure 6.7: The Mean Partition Size of the Adaptive Policies combined with SDF for WK1 and WK4.

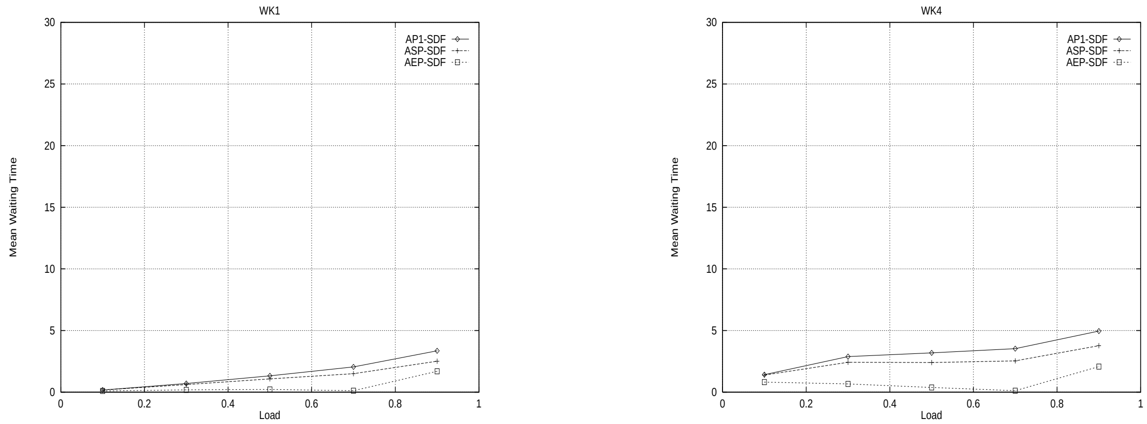

Figure 6.8: The Mean Waiting Time of the Adaptive Policies combined with SDF for WK1 and WK4.

best (McCann and Zahorjan 1989; Wu 1993). Here this greedy approach appears with the name ASP-SDF-OPT. As will be shown below, this decomposition allows a considerable performance improvement by replacing ASP with AEP.

In figure 6.9, we have depicted the Mean Response Time of all three triple combinations, ASP-SDF-OPT, AP1-SDF-OPT and AEP-SDF-OPT. The most important conclusion is that the enhanced variation of AEP always performs better than the other two policies. There is a small exception at high load of good speedup workloads, where statistically insignificant differences in response time are attained. This is due to the potential of the jobs to exploit the more processors they are given. Another important conclusion is that the enhancement that OPT adds does not affect the performance of ASP-SDF and AP1-SDF.

The main reason for the second conclusion is the SDF policy itself. It seems that the allocation optimization, as described in Chapter 4, is most effective when it can discriminate

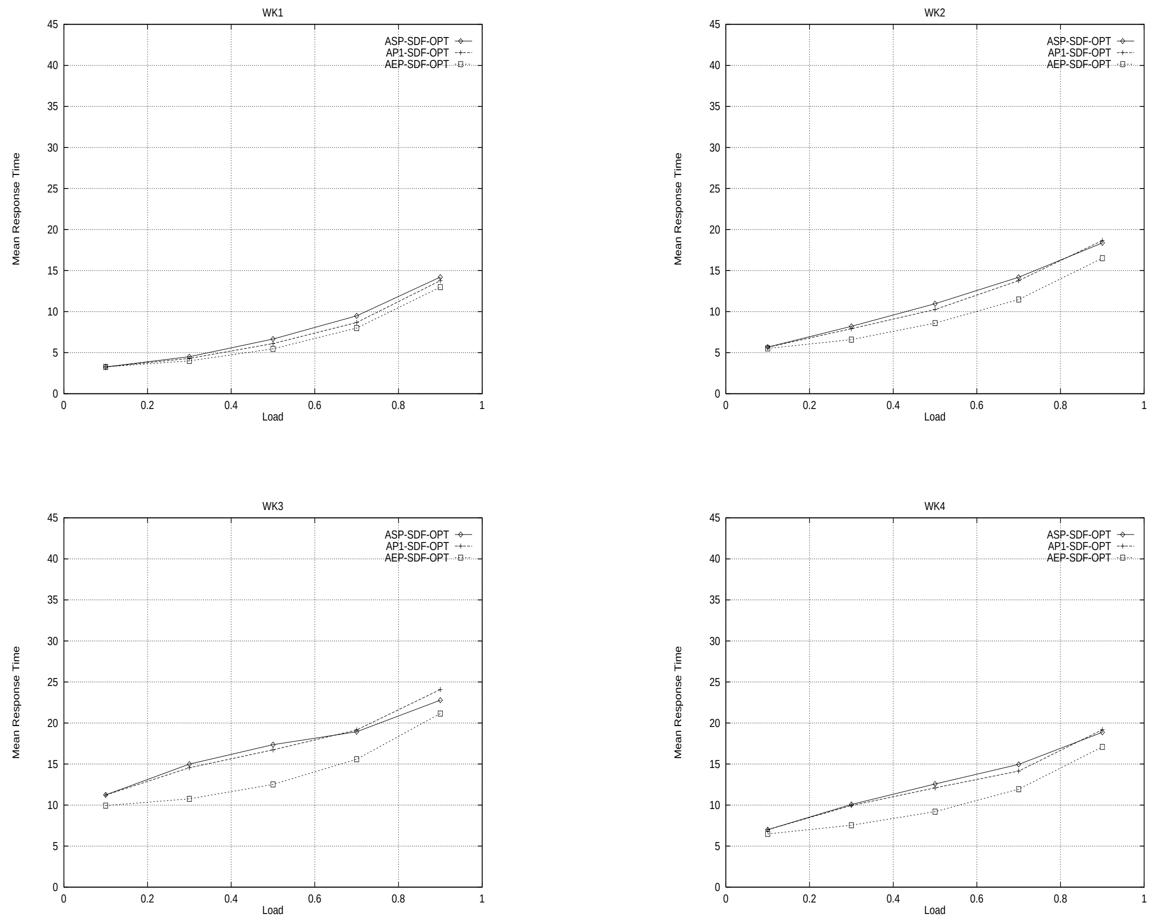

Figure 6.9: The Mean Response Time of the Adaptive Policies combined with SDF and OPT.

between jobs with small and large essential computation. Actually, the essential computation W determines the potential of a job to reduce its response time when given additional processors. The optimization procedure is not affected by the value of α, since α does not participate in the derivative of the execution time function (Sevcik 1994). Furthermore β is important but in typical workloads is much less than W (Wu 1993). Thus the ordering of the jobs according to the runtime on a single processor separates the jobs into groups with similar potential for processor exploitation.

OPT is effective in the case of AEP, due to the inherent tendency of the latter to give few processors to each job, and therefore to be sensitive to the final processor distribution. Of course, slightly changing the allocation size to a job with ten processors, is much less effective than with a job that has obtained only one. In the second case, an additional processor may even halve the execution time of the job. This argument does not hold in the

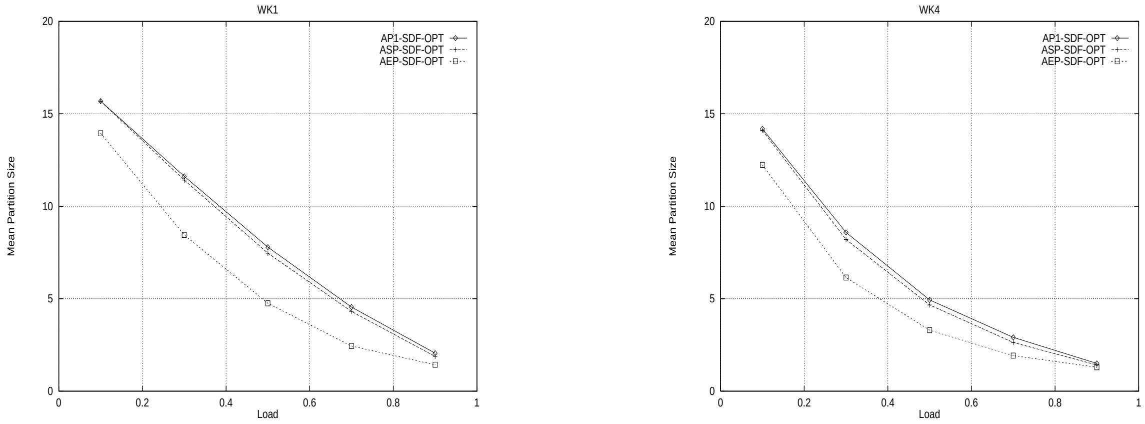

Figure 6.10: The Mean Partition Size of the Adaptive Policies combined with SDF and OPT for WK1 and WK4.

case of ASP and AP1, due to their tendency to keep the mean partition size significantly larger than that of AEP (Figure 6.10).

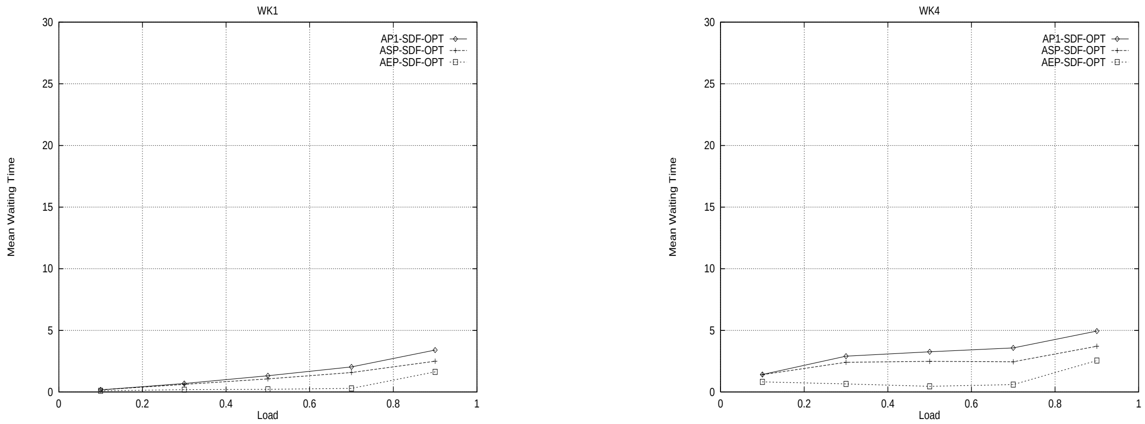

Figure 6.11: The Mean Waiting Time of the Adaptive Policies combined with SDF and OPT for WK1 and WK4.

From figures 6.10 and 6.11, where the mean partition size and the mean waiting time are depicted, it is obvious that both the partition size and the waiting time remain the same in the case of AEP, when compared to that without OPT. Therefore, the OPT clearly affects only the execution time as was expected.

6.5 Comparison with the Dynamic Policy

Finally, we will examine how our best run-to-completion algorithm, AEP-SDF-OPT, compares with the idealized version of the Dynamic Algorithm, where the reconfiguration overhead is assumed to be zero. From figure 6.12 it is easy to find out, how close to DYN, the algorithm AEP-SDF-OPT manages to keep, regardless of the speedup or the arrival rate.

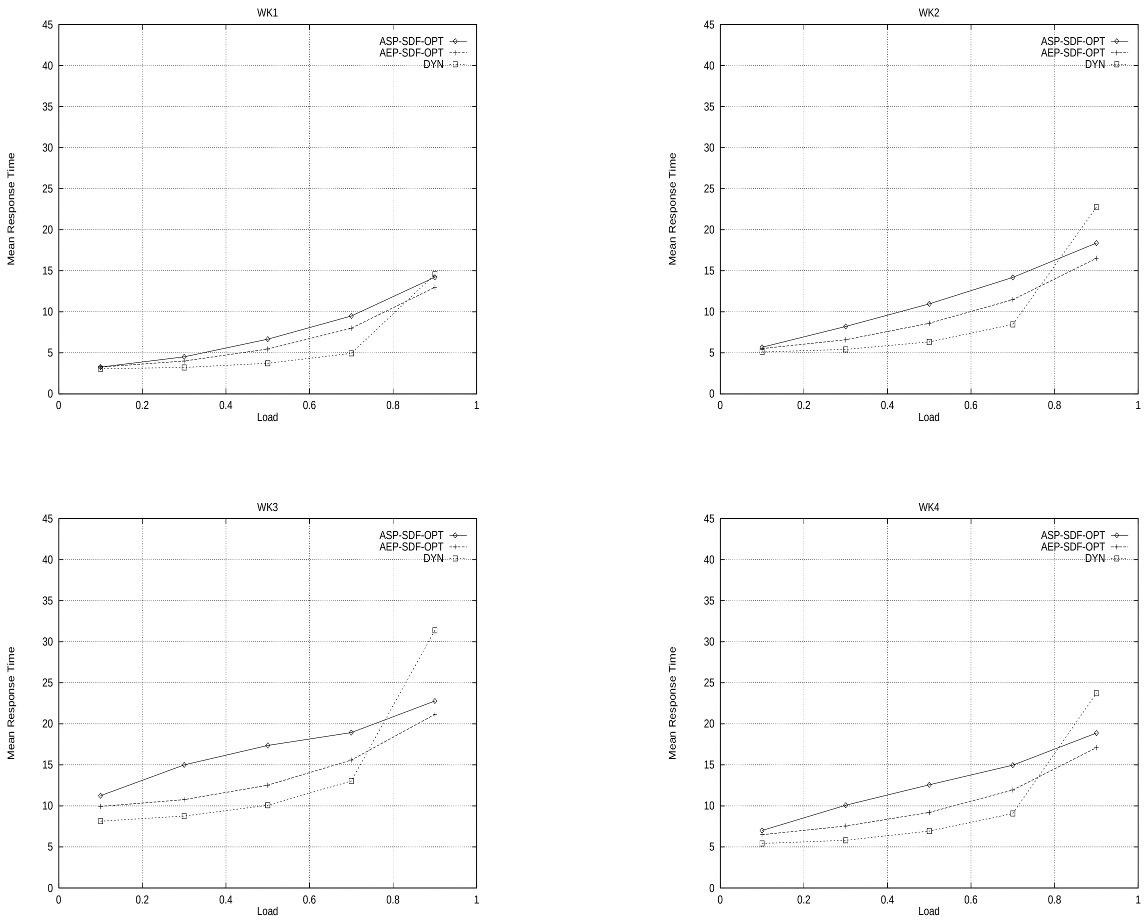

Figure 6.12: The Mean Response Time of AEP-SDF-OPT, ASP-SDF-OPT and Dynamic.

First, at high loads AEP-SDF-OPT (and ASP-SDF-OPT) performs better than DYN. In order to explain this result, it is necessary to observe the waiting time for the two types of policies as a function of load (fig. 6.13). We realize that at high loads the preemptive behavior of DYN fails to make the waiting time zero. On the other hand, the mean partition size of the adaptive policy is very close to that of DYN (fig. 6.14). In addition, the former has the advantage of the Shortest Demand First policy, which minimizes the expected waiting time. Again, in workload WK1, the difference is negligible because the jobs are efficient enough to complete quickly and keep the queue length low. This can be verified from the waiting time figure.

It is noteworthy, that by adding SDF to DYN we could improve its performance further

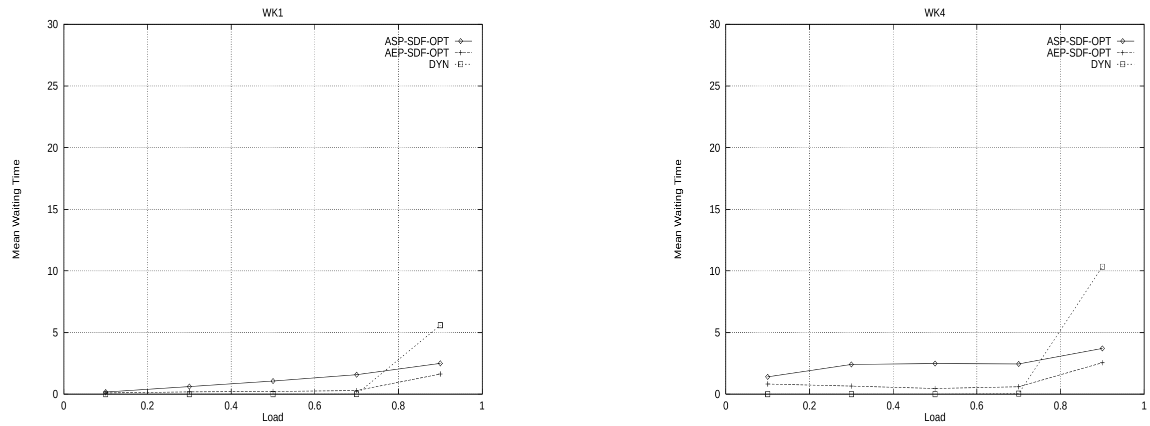

Figure 6.13: The Mean Waiting Time of AEP-SDF-OPT, ASP-SDF-OPT and Dynamic for WK1 and WK4.

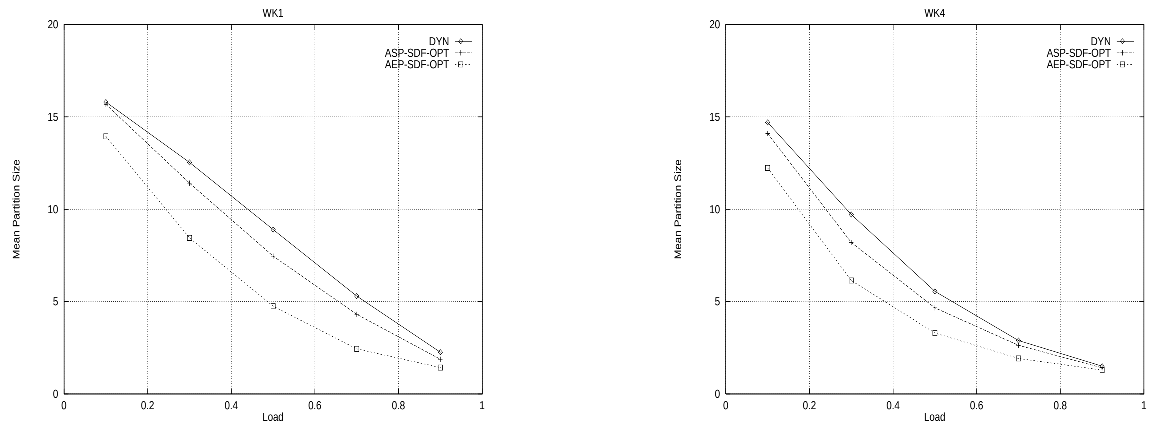

Figure 6.14: The Mean Partition Size of AEP-SDF-OPT, ASP-SDF-OPT and Dynamic for WK1 and WK4.

and probably by use of OPT (Wu 1993). However, it seems that estimation of the execution time function for a Dynamic policy is not as straightforward as in the case of static partitioning, due to the continuous change in the number of processors. But, both SDF and OPT need the execution time representation in order to be realized.

In addition, by comparing our algorithm AEP-SDF-OPT with ASP-SDF-OPT (Wu 1993), we realize that, in the general case (WK4), the former manages to be $20-30\%$ worse than Dynamic, while the latter is correspondingly $55-80\%$ worse. The relative difference is smaller in the specific case of WK1, but even there the advantage of the AEP-based policy remains statistically significant.

A final issue is that, as was reported in the previous section, OPT is ineffective and it could even be omitted in the case of ASP (and AP1). However, since even the support of the SDF rule practically requires knowledge of the execution time function, the OPT in the case of AEP is taken almost for free, and cannot be considered to be significant additional overhead required by the algorithm for the reported improvement to be achieved.

Chapter 7

Conclusions

The central goal of this thesis was the understanding and the improvement, if possible, of the most promising scheduling policies that could serve the general-purpose parallel processing requirements of workstation cluster users. It was necessary to identify the impact that different levels of system load and application characteristics knowledge have on the ability of scheduling algorithms to yield good performance. Usually, different types of information are used by separate policies, and it is unclear how they interact and what benefit each yields under different system and workload conditions.

A major step in our work was the definition of simple scheduling rules and the clarification of the type and level of information that each of them needs. With respect to system load, three different policies were described. Two of them, known in the literature as Adaptive Static Partitioning (ASP) and Adaptive Policy 1 (AP1), are based primarily on the queue length of the waiting jobs for the processor allocation decisions. A third policy was introduced, which we called Adaptive Equipartition (AEP). It is intended to improve the robustness of the scheduler by using the total number of jobs in the system and not just the waiting ones.

With respect to application characteristics, we investigated two different approaches. The simplest one, usually called Shortest Demand First (SDF), is based on knowledge of the application total service demand. A more complex policy, called Optimal (OPT), is also introduced, which assumes knowledge of the application execution time function.

Finally, a variation of Dynamic partitioning (DYN), was defined. This is based on the total number of jobs in the system, and assumes that the applications are capable of adapting to changing numbers of processors during execution. In order to unify our results with previous ones, we assumed that all the policies have information about the number of allocated processors beyond which the runtime of the application increases instead of decreasing. Actually, this allocation upper bound is only known by those policies that have knowledge of the job execution time function.

We represented the characteristics of a wide range of applications with proper choices of parameters in Sevcik's execution time model. This was preferred to Dowdy's model due to the proven increased accuracy of the former in approximating real applications. In particular, three separate workloads are formed with applications having different speedup characteristics. A fourth workload that incorporates all the other three in equal proportions is also included.

Our first major conclusion from the simulation experiments is that, at high loads and with applications having sublinear speedup, SDF fails to complete the jobs in finite time. This is not surprising, since SDF allocates processors up to the maximum requirement of the jobs, regardless of the load in the system. Thus at high loads, the reduction in the runtime of the jobs allocated many processors, is not large enough to compensate for the increased waiting time experienced by jobs still in the queue with no processors. Another important conclusion from the same experiment, is that fixed maximum allocation limits cannot provide general improvement in SDF in constrast to previous claims about this. They just improve the performance for some particular workload or arrival rate.

Comparison of the three adaptive policies verifies that AEP tends to allocate fewer processors and to decrease the waiting time at higher loads, as was shown analytically. This behavior makes AEP perform better in the general case. However, its comparative performance is slightly worse when the jobs have nearly linear speedup. This is expected since applications that exploit well the additional allocated processors are capable of keeping the waiting time of the queued ones acceptably low. Finally, no differences are found between the performance of ASP and AP1. It is the first time that these two policies are compared with respect to mean response time.

Our next step is to compare the adaptive policies when combined with SDF, called respectively ASP-SDF, AP1-SDF and AEP-SDF. It is shown that the response time of all three is considerably decreased, due to the expected reduction in the waiting time from SDF. For the same reason, the advantage of AEP of keeping waiting time low is much less important than before. Therefore, the differences in performance among the three policies appear significantly reduced. However, it is important that the properties of SDF are orthogonal to those of the adaptive policies, and performance improvement is only obtained from the above combination.

The final step is to add OPT to the policies of the last paragraph and thus obtaining ASP-SDF-OPT, AP1-SDF-OPT and AEP-SDF-OPT, with ASP-SDF-OPT already known in literature and the the other two introduced for first time. The gain for the policies based on ASP-SDF and AP1-SDF is negligible. A reason for this is the fact that due to SDF the jobs dispatched together and on which OPT applies have similar total demand and therefore potential for decreasing their response time with additional processors. However, the gain of AEP-SDF-OPT is considerable at high loads due to its tendency to allocate fewer processors than the other two and the significance therefore of the exact processor distribution among the jobs.

Comparison among AEP-SDF-OPT, ASP-SDF-OPT and Dynamic shows that the former two always perform better at very high loads. The reason is that at very high loads the job partition sizes of both policies are very close to one. With the additional advantages of SDF and OPT, it is expected that AEP-SDF-OPT and ASP-SDF-OPT will be better. At the other loads, Dynamic performs better, due to its preemptive property. The difference for AEP-SDF-OPT and the mixed workload (WK4) is typically between $20 - 30\%$ of the response time relative to Dynamic. The corresponding differences between ASP-SDF-OPT and Dynamic are about $55 - 80\%$.

It seems that by exploiting properly the load and application characteristics information, we have managed to improve the performance of the non-preemptive policies and to come very close to that of zero-overhead Dynamic. Thus, we have proven that it is possible to design efficient schedulers for network clusters where Dynamic partitioning cannot be a general solution. In addition, with adequate support in the operating system, the program developer is free to choose the parallel programming model that best fits each application, without compromises in the scheduling efficiency. In this way, the operating system remains a collection of services that offers options to the users and minimizes the difficulties of utilizing the virtues of parallelism.

7.1 Future Work

In this thesis, we have described an approach for improving the performance of static space-sharing scheduling of parallel systems.

- An implementation of the proposed algorithm on an actual workstation cluster is a necessary next step for verifying the simulation predictions, and revealing any problems not captured by a simulation study.

- Only experimentation in a real environment can reveal the potential and accuracy of approximating the execution time function of parallel applications with the existing models.

- Heterogeneity, a main feature of most distributed system, is itself a challenging problem to be handled by the parallel application scheduler. Machines with different uniprocessor or multiprocessor configurations must be considered, along with applications with specific hardware requirements.

- Interaction with the sequential workload is a very serious problem, peculiar to workstation installations, that has to be handled. The issue becomes more interesting when studied within load balancing environments.

- It is necessary to investigate the impact of the applications with minimum processor allocation demands. Their existence in the workload can complicate further the scheduling decisions.

We believe that the limit of non-preemptive scheduling performance has not been reached yet. It is still an open question how application characteristics and system load parameters, whether employed in our approach or not, can be used for further enhancement in the response time of parallel processing.

Bibliography

Agrawal, R. and A. K. Ezzat (1985, November). Processor Sharing in NEST: A Network of Computer Workstations. In *1st Int'l Conference on Computer Workstations*, pp. 198–208.

Anderson, T. E., D. E. Culler, D. A. Patterson, and the NOW team (1995, February). A Case for NOW (Networks of Workstations). *IEEE Micro 15*(2), 54–64.

Arpaci, R. H., A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson (1994). The Interaction of Parallel and Sequential Workloads on a Network of Workstations. Technical Report CS-94-838, Computer Science Division, University of California, Berkeley.

Black, D. L. (1990, May). Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *Computer 23*(15), 35–43.

Carriero, N., E. Freeman, D. Gelenter, and D. Kaminsky (1995, January). Adaptive Parallelism and Piranha. *Computer 28*(1), 40–49.

Chandra, R., S. Devine, B. Verghese, A. Gupta, and M. Rosenblum (1994, November). Scheduling and Page Migration for Multiprocessor Computer Servers. In *6th Int'l Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS VI), San Jose, CA*, pp. 12–24.

Chase, J. S., F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield (1989, December). The Amber System: Parallel Programming on a Network of Multiprocessors. In *12th ACM Symposium Operating Systems Principles*, pp. 147–158.

Chiang, S.-H., R. K. Mansharamani, and M. K. Vernon (1994, May). Use of Application Characteristics and Limited Preemption for Run-To-Completion Parallel Processor Scheduling Policies. In *ACM SIGMETRICS Conf. Measurement and Modeling of*

Computer Systems, pp. 33–44.

Crovella, M., P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos (1991, December). Multiprogramming on Multiprocessors. In *3rd IEEE Symposium Parallel and Distributed Processing*, pp. 590–597.

Devarakonda, M. V. and R. K. Iyer (1989, December). Predictability of Process Resource Usage: A Measurement-Based Study on Unix. *IEEE Transactions on Software Engineering 15*(12), 1579–1586.

Dussa, K., B. Carlson, L. Dowdy, and K.-H. Park (1990, May). Dynamic Partitioning in a Transputer Environment. In *ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp. 203–213.

Eager, D. L., J. Zahorjan, and E. D. Lazowska (1989, March). Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers 38*(3), 408–433.

Efe, K. and M. A. Schaar (1993). Performance of Co-Scheduling on a Network of Workstations. In *13th Int'l Conference on Distributed Computing Systems*, pp. 525–531.

Feeley, M. J., B. N. Bershad, J. S. Chase, and H. M. Levy (1991). Dynamic Node Reconfiguration in a Parallel-Distributed Environment. In *3rd ACM Symposium Principles and Practice of Parallel Programming*, pp. 114–121.

Feitelson, D. G. (1994, October). A Survey of Scheduling in Multiprogrammed Parallel Systems. Research Report RC-19790 (87657), IBM T. J. Watson Research Center.

Feitelson, D. G. and B. Nitzberg (1995, April). Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860. In *IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 215–227.

Feitelson, D. G. and L. Rudolph (1992a, October). Coscheduling based on Run-Time Identification of Activity Working Sets. Research Report RC-18416 (80519), IBM T. J. Watson Research Center.

Feitelson, D. G. and L. Rudolph (1992b, December). Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *Journal of Parallel and Distributed Computing 16*(4), 306–318.

Feitelson, D. G. and L. Rudolph (1995, April). Parallel Job Scheduling: Issues and Approaches. In *IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 1–8.

Ghosal, D., G. Serazzi, and S. K. Tripathi (1991, May). The Processor Working Set and Its Use in Scheduling Multiprocessor Systems. *IEEE Transactions on Software Engineering 17*(5), 443–453.

Graham, R. L., E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan (1979). Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey. *Annals of Discrete Mathematics 5*, 287–326.

Gupta, A., A. Tucker, and S. Urushibara (1991, May). The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Application. In *ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp. 120–132.

Hagmann, R. (1986, May). Process Server: Sharing Processing Power in a Workstation Environment. In *6th IEEE Distributed Computing Conference*, pp. 260–267.

Ibaraki, T. and N. Katoh (1988). *Resource Allocation Problems: Algorithmic Approaches*. Series in the Foundations of Computing. MIT Press.

Jones, M. B. (1993, December). Interposition Agents: Transparently Interposing User Code at the System Interface. In *14th ACM Symposium on Operating Systems Principles*, pp. 80–93.

Krueger, P. and R. Chawla (1991, May). The Stealth Distributed Scheduler. In *11th Int'l Conference Distributed Computing Systems*, pp. 336–343.

Law, A. M. and W. D. Kelton (1991). *Simulation Modeling and Analysis* (Second ed.). McGraw-Hill.

Leutenegger, S. T. and X.-H. Sun (1993, April). Distributed Computing Feasibility in a Non-Dedicated Homogeneous Distributed System. In *Int'l Conference Supercomputing*, pp. 143–152.

Leutenegger, S. T. and M. K. Vernon (1990, May). The Performance of Multiprogrammed Multiprocessor Scheduling Policies. In *ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp. 226–236.

Leuze, M. R., L. W. Dowdy, and K.-H. Park (1989, September). Multiprogramming a Distributed-Memory Multiprocessor. *Concurrency: Practice and Experience 1*(1), 19–33.

Litzkow, M. J., M. Livny, and M. W. Mutka (1988, June). Condor - A Hunter of Idle Workstations. In *8th IEEE Distributed Computing Conference*, pp. 104–111.

Majumdar, S., D. L. Eager, and R. B. Bunt (1988, May). Scheduling in Multiprogrammed Parallel Systems. In *ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp. 104–113.

Majumdar, S., D. L. Eager, and R. B. Bunt (1991, October). Characterization of programs for scheduling in multiprogrammed parallel systems. *Performance Evaluation 13*(2), 109–130.

Marse, K. and S. D. Roberts (1983). Implementing a Portable FORTRAN Uniform (0,1) Generator. *Simulation 41*, 135–139.

McCann, C., R. Vaswani, and J. Zahorjan (1993, May). A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems 11*(2), 146–178.

McCann, C. and J. Zahorjan (1989). Processor Scheduling in Shared Memory Multiprocessors. Technical Report 89-09-17, Computer Science Department, University of Washington at Seattle.

McCann, C. and J. Zahorjan (1994, May). Processor Allocation Policies for Message-Passing Parallel Computers. In *ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp. 19–32.

McCann, C. and J. Zahorjan (1995). Scheduling Memory Constrained Jobs on Distributed Memory Parallel Computers. In *ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp. 208–219.

Mutka, M. W. and M. Livny (1987). Scheduling Remote Processing Capacity In a Workstation-Processor Bank Network. In *7th Int'l Conference on Distributed Computing Systems*, pp. 2–9.

Mutka, M. W. and M. Livny (1991). The Available Capacity of a Privately Owned Workstation Environment. *Performance Evaluation 12*(4), 269–284.

Naik, V. K., S. K. Setia, and M. S. Squillante (1993, November). Performance Analysis of Job Scheduling Policies in Parallel Supercomputing Environments. In *Int'l Conference*

Supercomputing, pp. 824–833.

Nedeljkovic, N. and M. J. Quinn (1993, June). Data Parallel programming on a Network of Heterogeneous Workstations. *Concurrency: Practice and Experience 5*(4), 257–268.

Ousterhout, J. K. (1982, October). Scheduling Techniques for Concurrent Systems. In *3rd Int'l Conference on Distributed Computing Systems*, pp. 22–30.

Park, K.-H. and L. W. Dowdy (1989). Dynamic Partitioning of Multiprocessor Systems. *International Journal of Parallel Programming 18*(2), 91–120.

Parsons, E. W. and K. C. Sevcik (1995, April). Multiprocessor Scheduling for High-Variability Service Time Distributions. In *IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 76–88.

Platform Computing Corp. (1994, December). *LSF Administrator's Guide.* Toronto, ON: Platform Computing Corp.

Prouty, R., S. Otto, and J. Walpole (1994, March). Adaptive Execution of Data Parallel Computations on Networks of Heterogeneous Workstations. Technical Report CSE-94-013, DCSE, Oregon Graduate Institute of Science and Technology.

Rosti, E., E. Smirni, L. Dowdy, G. Serazzi, and B. M. Carlson (1994). Robust Partitioning Policies of Multiprocessor Systems. *Performance Evaluation 19*(2–3), 141–165.

Schwetman, H. (1992a, June). *CSIM Reference Manual.* Microelectronics and Computer Technology Corp. Revision 16.

Schwetman, H. (1992b, June). *CSIM Users' Guide.* Microelectronics and Computer Technology Corp. Revision 16.

Setia, S. and S. Tripathi (1993, January). A Comparative Analysis of Static Processor Partitioning Policies for Parallel Computers. In *Int'l Workshop on Modeling and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 283–286.

Setia, S. K., M. S. Squillante, and S. K. Tripathi (1993). Processor Scheduling on Multiprogrammed Distributed Memory Parallel Computers. In *ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp. 158–170.

Setia, S. K., M. S. Squillante, and S. K. Tripathi (1994, May). Analysis of Processor Allocation in Multiprogrammed Distributed-Memory Parallel Processing Systems.

IEEE Trans. on Parallel and Distributed Systems 5(4), 401–420.

Sevcik, K. C. (1989, May). Characterizations of Parallelism in Applications and Their Use in Scheduling. *Performance Evaluation Review 17*(1), 171–180.

Sevcik, K. C. (1994). Application Scheduling and Processor Allocation in Multiprogrammed Parallel Processing Systems. *Performance Evaluation 19*(2–3), 107–140.

Silverman, R. D. and S. J. Stuart (1989, December). A Distributed Batching System for Parallel Processing. *Software-Practice and Experience 19*(12), 1163–1174.

Sun, X.-H. and L. M. Ni (1993, September). Scalable Problems and Memory-Bounded Speedup. *Journal of Parallel and Distributed Computing 19*(1), 27–37.

Trivedi, K. S. (1982). *Probability & Statistics with Reliability, Queuing, and Computer Science Applications.* Prentice-Hall Inc., Englewood Cliffs, N.J.

Tucker, A. and A. Gupta (1989, December). Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. In *12th ACM Symp. Operating System Principles*, pp. 159–166.

Vahdat, A., D. Ghormley, and T. Anderson (1994, December). Efficient, Portable and Robust Extension of Operating System Functionality. Technical Report CS-94-842, Computer Science Division, University of California, Berkeley.

von Eicken, T., A. Basu, and V. Buch (1995, February). Low-Latency Communication Over ATM Networks Using Active Messages. *IEEE Micro 15*(2), 46–53.

Wahbe, R., S. Lucco, T. E. Anderson, and S. L. Graham (1993, December). Efficient software-based fault isolation. In *14th ACM Symposium on Operating Systems Principles*, pp. 203–216.

Wang, C.-J., P. Krueger, and M. T. Liu (1993, May). Intelligent Job Selection for Distributed Scheduling. In *13th Int'l Conference on Distributed Computing Systems*, pp. 517–524.

Wang, Z. (1995). Trace Data Analysis and Job Scheduling Simulation for Large-Scale Distributed Systems. Master's thesis, Department of Computer Science, University of Toronto.

Welch, P. D. (1983). The Statistical Analysis of Simulation Results. In S. S. Lavenberg (Ed.), *The Computer Performance Modeling Handbook*, pp. 268–328. Academic Press, New York.

Wu, C.-S. (1993). Processor Scheduling in Multiprogrammed Shared Memory NUMA Multiprocessors. Master's thesis, Department of Computer Science, Technical Report CSRI-341, University of Toronto.

Zahorjan, J. and C. McCann (1990, May). Processor Scheduling in Shared Memory Multiprocessors. In *ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp. 214–225.

Zhou, S. and T. Brecht (1991, May). Processor Pool-Based Scheduling for Large-Scale NUMA Multiprocessors. In *ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp. 133–142.

Zhou, S., X. Zheng, J. Wang, and P. Delisle (1993, December). Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. *Software-Practice and Experience 23*(12), 1305–1336.