# A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors

CATHY MCCANN, RAJ VASWANI, and JOHN ZAHORJAN
University of Washington, Seattle

We propose and evaluate empirically the performance of a dynamic processor-scheduling policy for multiprogrammed shared-memory multiprocessors The policy is dynamic in that it reallocates processors from one parallel job to another based on the currently realized parallelism of those jobs. The policy is suitable for implementation in production systems in that:

—It interacts well with very efficient user-level thread packages, leaving to them many low-level thread operations that do not require kernel intervention.
—It deals with thread blocking due to user I/O and page faults
—It ensures fairness in delivering resources to jobs.
—Its performance, measured in terms of average job response time, is superior to that of previously proposed schedulers, including those implemented in existing systems.
—It provides good performance to very short, sequential (e.g., interactive) requests.

We have evaluated our scheduler and compared it to alternatives using a set of prototype implementations running on a Sequent Symmetry multiprocessor. Using a number of parallel applications with distinct qualitative behaviors, we have both evaluated the policies according to the major criterion of overall performance and examined a number of more general policy issues, including the advantage of "space sharing" over "time sharing" the processors of a multiprocessor, and the importance of cooperation between the kernel and the application in reallocating processors between jobs. We have also compared the policies according to other criteria important in real implementations, in particular, fairness and response time to short, sequential requests. We conclude that a combination of performance and implementation considerations makes a compelling case for our dynamic scheduling policy.

Categories and Subject Descriptors: D.4 1 [**Operating Systems**]: Process Management—*multiprocessing/multiprogramming, scheduling*

General Terms Measurement, Performance

Additional Key Words and Phrases: Shared memory parallel processors, threads, two-level scheduling

This material is based on work supported by the National Science Foundation (grants CCR-8619663, CDA-9123308, and CCR-9200832), the Washington Technology Center, and Digital Equipment Corporation (the External Research Program and the Systems Research Center).
Authors' address: University of Washington, Department of Computer Science and Engineering, FR-35, Seattle, WA 98195; email: mccann@ raj@ zahorjan@cs.washington.edu.

## 1. INTRODUCTION

We consider alternative strategies for scheduling parallel jobs on multiprogrammed shared-memory parallel computers. Such machines typically have a modest number of processors (say, between two and 32) connected to main memory by a shared bus. This architecture is in use for machines ranging in power from single-user workstations (e.g., the Digital Equipment Corporation Firefly experimental workstation [23]) to medium-scale general-purpose machines (e.g., the Sequent Symmetry [16]) to supercomputers (e.g., the Cray MP's). While systems of this type have been in operation for a number of years, the available approaches for scheduling them have been considered in detail only quite recently. Work by Ousterhout [18], Zahorjan et al. [26, 27], and Gupta et al. [11] provides convincing evidence that *single-level* schedulers, those in which the kernel schedules all threads of all applications (usually from a single queue of ready threads), are not appropriate choices. Such schedulers can provide very poor performance to individual jobs, whose threads often need to synchronize or execute critical sections, and they do not provide fair service when one application contains many more threads than others.

Based on the evidence given in these earlier works, we restrict our attention to *two-level* schedulers, those in which (1) the kernel allocates processors to applications and (2) the applications themselves schedule their threads on those processors. There have been a number of recent proposals for schedulers of this type. Much of this work is based on modeling (e.g., [14], [17], [19], [21], and [25]) and of necessity made simplifying assumptions about the system and its workload. Tucker and Gupta [22] do provide an experimental evaluation of an aspect of multiprocessor scheduling, and Gupta et al. [11] a trace-driven simulation, but even here the policy aspects of the study are simplified by ignoring, for example, the effects of I/O and page faults.

The purpose of our study was to evaluate multiprocessor scheduling when all aspects of the problem are included. We have built schedulers for a number of policies and run them on a Sequent Symmetry. We have used existing parallel applications as our workload. Also, we have considered both the raw performance afforded by the policies, measured in terms of average response time, and other less easily quantified measures, such as fairness.

We characterize scheduling strategies for multiprogrammed shared-memory multiprocessors along three dimensions. The first, and perhaps most basic, is the manner in which concurrency is provided. Under *time sharing*, the processors are quickly rotated from one job to another. Thus, each job sees alternating periods when it holds many (possibly all) processors followed by few (possibly no) processors. Under *space sharing*, the processors of the machine are partitioned among the jobs. Space sharing tends to provide each job a more constant allocation of a fewer number of processors than does time sharing.

The second dimension concerns the frequency with which allocation decisions are made. Under a *static* policy, a single allocation decision is made per job at the start of execution. Whenever the job has any processors, it has the number of processors decided on initially. Under a *dynamic* policy, the number of processors allocated each job may change at arbitrary times. While it may be easier to program applications to run under a static policy (since these applications can count on having a fixed number of processors once they start execution), the dynamic policies have greater potential to efficiently execute applications whose parallelism changes during their lifetimes. On the other hand, dynamic policies incur more system overhead, which may lead to a degradation of performance.

The final dimension distinguishes *uncoordinated* reallocations, those in which the kernel moves the processor without interaction with the application, from *coordinated* processor reallocations, those that are performed in concert with the application. Previous work using modeling [26, 27] has shown that uncoordinated preemption can lead to very poor performance. As a simple example, imagine that the preempted processor has been running a thread that holds a spin lock. In this case, preemption leads quickly to spinning by the active threads. This waste of processors continues until the suspended thread is restarted and can seriously affect both application and overall system performance. While this effect can be alleviated somewhat through modified lock acquisition protocols [7] or by the use of "spin then block" locks [12, 15], these can neither completely prevent the degradation, nor are they applicable to the more general problem of preemption of threads critical to the progress of the application (but which may not be holding locks).

We examine three particular scheduling strategies, each based on a strategy previously proposed in the literature. While these represent only a subset of the eight possible strategies defined by our three dimensions, examination of their performance is sufficient to determine the appropriate decision in each dimension. For example, one of the policies uses time sharing while the other two use space sharing. The performance of the strategies under a workload that exercises only this aspect of the disciplines makes clear the better choice.

Section 2 begins our discussion with a description of the environment in which the processor allocator must operate. Section 3 fully defines the abstract policies we have examined. Section 4 describes their implementation in the testbed system. In Section 5 we present the workloads used in our experiments and in Section 6 the results of scheduling them under the policies. Section 7 summarizes our conclusions.

## 2. THE SOFTWARE ENVIRONMENT: USER-LEVEL THREADS

In the environment we are addressing, communication between executing threads is relatively cheap because the underlying hardware provides shared memory at a uniform distance from all processors. Thus, the limiting factor in achieving good performance typically is the inability to find sufficient parallelism.

User-level thread packages address this limitation by reducing the costs of creating, synchronizing (blocking and unblocking), and scheduling parallel threads of execution by an order of magnitude over kernel-level implementations [5, 6]. Since parallelism is cheap, the application programmer is free to exploit much finer-grained computations in an attempt to keep all processors busy, leading to improved performance.

The efficiency of user-level threads is obtained by performing operations at the user, rather than kernel, level. This avoids the overhead of entering and exiting the kernel, which can be substantial relative to the amount of work required, for instance, to block a thread. Also, because the operations are executed within the context of a particular job, much of the "bullet-proofing" required in kernel code can be avoided in user-level implementations.

While user-level thread packages differ somewhat in the details of their implementation, the basic concepts are the same in all cases. Upon starting execution, an application forks a kernel-schedulable thread of execution per physical processor. These threads, which we call *virtual processors*, merely provide the generic environment in which to run the application.

Each virtual processor, when allocated a physical processor, alternates between running one of two kinds of user-level thread: *scheduler threads* and *application threads*. A scheduler thread, which executes code provided by the thread package, examines a data structure that holds ready application threads, and if one is found, removes it from the structure and initiates its execution on its virtual processor. The application thread, so named because it executes code provided by the application, performs the actual work necessary to implement that application. Application threads typically run on virtual processors until they either block or terminate. At that time, a scheduler thread regains control and looks for another ready application thread to run.

It is important to note that blocking can occur at two distinct levels when using a threads package. A user-level thread can block by executing some synchronization primitive defined by the threads package. For instance, an application thread may block when executing a barrier synchronization with a set of other application threads. This kind of blocking is handled quickly and efficiently, and the virtual processor is then available to run some other thread.

On the other hand, if a user-level thread performs (blocking) I/O, or experiences a page fault, the event is caught below the threads package, at the kernel level. The kernel sees this as an event pertaining to the virtual processor, and so blocks it. This means that not only is the thread that was running on that virtual processor blocked (as it must be), but also that the virtual processor itself is unavailable to run any other user-level threads.

A similar situation arises when the processor allocator reallocates a processor from one job to another, preempting the virtual processor running there. In this case, while the virtual processor must be blocked in the kernel, the user-level thread need not be. If it is an application thread, it should be reentered in the data structure containing ready threads, since it may be completed by any active virtual processor. One motivation for coordinated processor preemption (as defined in Section 1) is to avoid blocking the user-level thread in these cases.

While some applications can run efficiently without user-level threads, these typically exhibit a very large amount of (data) parallelism, which the user partitions into large-grained pieces that can tolerate the overheads of kernel-level operations. Such applications are often sensitive to the number of physical processors they are allocated and can behave badly if their allocation is changed during execution [26], either because of scheduling or because of I/O or page faults. This leads to "work heap" based implementations, in which a number of identical worker threads share a pool of task descriptions, each cycling between removing a descriptor from the pool and performing the task it describes. Such implementations mimic the basic functionality and structure of user-level thread packages, and so present the same set of considerations for the system processor allocator.

User-level thread packages thus provide a single, coherent, programming paradigm in which to implement parallel applications (a benefit even to massively and statically parallel applications). Further, (the functions of) such packages are required for the efficient implementation of some applications. We therefore assert that any scheduling discipline proposed for the shared-memory multiprocessors we are considering must interact in a reasonable way with the characteristics of user-level thread implementations. In the remainder of this paper, we assume that applications use such a threads package.

## 3. PROCESSOR ALLOCATION POLICIES

In this section, we fully define the abstract versions of the three processor allocation policies that we consider. The approximations to these policies actually implemented in our prototype are discussed in Section 4.

The three policies are called *Round-Robin Job*, *Equipartition*, and *Dynamic*. Table I summarizes the decisions each makes along the dimensions of time versus space sharing, static versus dynamic allocation, and uncoordinated versus coordinated preemption.

As suggested by the table, comparing Round-Robin Job to Equipartition yields an evaluation of both time sharing versus space sharing and uncoordinated versus coordinated preemption; subsets of our results (Section 6) isolate the relative influence of each policy characteristic. Similarly, the comparison between Equipartition and Dynamic allows us to focus on the impact of static versus dynamic reallocation, since the two policies differ only along that dimension.

In what follows, we use $P$ to denote the number of processors in the system and $N_j$ to denote the *maximum parallelism* of job $j$. We define maximum parallelism as the maximum number of processors that a job would simultaneously keep busy at any point during its execution if it were allocated an unlimited number of processors.

### 3.1 Round-Robin Job (RRJob)

RRJob is a static policy that time-shares processors among the jobs in the system. Allocation is quantum driven. During job $j$'s quantum, it is assigned $N_j$ processors. The remaining $P - N_j$ unassigned processors are given to the job whose quantum follows next. The length of a quantum depends on

Table I. Basic Characteristics of Scheduling Policies

| | Time vs. Space Sharing | Static vs. Dynamic Allocation | Uncoord. vs. Coord. Reallocations |
|---|---|---|---|
| **Round-Robin Job** | Time | Static | Uncoordinated |
| **Equipartition** | Space | (Quasi-)Static | Coordinated |
| **Dynamic** | Space | Dynamic | Coordinated |

the number of processors allocated to the currently active job, which is $N_j$. In particular, job $j$'s quantum is of duration $Q/N_j$, where $Q$ is a constant measured in processor seconds. This ensures that all jobs receive the same total amount of service ($Q$ processor seconds) during their quanta.

To give slight preference to any newly arriving job, which might be of very short duration, its quantum is inserted in the ordering immediately following the quantum in progress when the job arrives. Thus, an arrival experiences the minimum delay possible without prematurely terminating the quantum of some other job.

RRJob performs uncoordinated processor preemption, taking all of a job's processors at the end of its quantum without any explicit notification to the job.

The basic operation of this policy was originally defined by Leutenegger and Vernon [14], who examined its performance relative to a number of alternative disciplines in an abstract setting. We have modified their policy to provide a per-job, rather than a per-processor, quantum, preempting all processors of that job at once rather than as each processor completes its individual quantum. We expect that this change reduces the detrimental impact of uncoordinated preemptions relative to the original definition of the policy.

## 3.2 Equipartition

Equipartition is a space-sharing policy that, to the extent possible, maintains an equal allocation of processors to all jobs. To do this requires reallocations only on job arrival and completion. Since reallocations are relatively infrequent, and because the reallocation decisions are independent of the instantaneous processor requirements of the jobs, we consider Equipartition to be a quasistatic discipline.

The Equipartition policy is based on the "process control" policy proposed by Tucker and Gupta [22]. Each time a reallocation must take place, the number of processors to allocate each job is computed by the following procedure. The initial allocation number of all jobs is set to zero. Then the allocation number of each job is incremented by one in turn, and any job whose allocation number has reached its maximum parallelism drops out. This process continues until either there are no remaining jobs or until all $P$

processors have been allocated. The set of allocation numbers computed in this way gives the number of processors that should be allocated to the jobs.

When a new job arrives, a reallocation is required that typically causes preemption of processors from some current jobs for reassignment to the new arrival. As proposed by Tucker and Gupta [22], these preemptions occur immediately: the kernel simply suspends the virtual processors (and thus the user-level threads) running on the targeted processors, reassigning the processors to the new job. The kernel than sets a flag for each job that lost a processor (via memory shared with that job) indicating the new number of processors assigned to that job. Because recently suspended virtual processors typically embody the state of user-level threads that must be completed, they are scheduled round-robin on their job's processors (by the kernel). It is the application's responsibility eventually to reduce the number of runnable virtual processors in this case. As a routine matter, when each virtual processor reaches a "safe point" (typically when the user-level thread it has been running has just blocked or terminated, so that the virtual processor is executing the scheduler thread rather than any user-level thread), it checks this flag and suspends itself if the number of runnable virtual processors for that job exceeds its current processor allocation. The round-robin scheduling therefore persists for an indeterminate period, since it depends on the frequency with which user-level threads block or complete. Thus, while this preemption scheme has the advantage of allowing immediate preemption of processors without requiring application cooperation, it has the disadvantage of introducing potential performance problems for applications that do not react well to the period of round-robin scheduling (e.g., because one of the preempted threads holds an important lock).

Since we intend the comparison of Equipartition with Dynamic to isolate the effect of static versus dynamic allocation, we have used a more closely coordinated preemption method in our version of Equipartition. (In this way, Equipartition and Dynamic differ only along the dimension of static versus dynamic allocation, and not in the manner of preemption.) Under our preemption scheme, the processor allocator "asks" a running job to hand over processors when it is next convenient, i.e., as virtual processors reach safe points in their computations. Because there is no interval of round-robin scheduling under this scheme, it does not suffer from the problems associated with round-robin scheduling of synchronizing threads. Thus, the average response time results we obtain with this preemption scheme should be at least as good as those resulting from the immediate preemption scheme of Tucker and Gupta [22]. At the same time, our preemption scheme is impractical for use in real systems because of its inferior performance in two other dimensions, the average delay to acquire a processor (see Section 6.4) and resilience to user countermeasures to the scheduler. Thus, while Equipartition serves our purpose in isolating the effect of static versus dynamic allocation, it would not be a suitable alternative (for reasons other than average response time) to the Tucker and Gupta scheme for implementation in a real system.

## 3.3 Dynamic

Dynamic is a dynamic, space-sharing policy. Under Dynamic, processors are reallocated among jobs in response to changes in the jobs' parallelisms. Each job's scheduler threads continually advertise to the allocator the number of additional processors the job could use. (This number corresponds to the number of ready but not executing application threads of that job.) Additionally, when a scheduler thread finds that there are no ready application threads to be run, it notifies the allocator that its processor is available for reallocation to another job. Such processors are said to be *willing to yield*. Using this information, the allocator can implement the basic mechanism of the Dynamic policy: moving processors from jobs that currently have too many to jobs that have too few.

When a job requests additional processors, Dynamic attempts to satisfy it by using the least valuable processors (i.e., those whose reallocations cause the least detriment to their current holders) available:

A.1 First, any unallocated processors are assigned.

A.2 Next, "willing to yield" processors (those allocated to some job but not currently in use) are assigned.

A.3 Finally, an equipartition allocation is enforced by preempting processors from the job(s) with the largest current allocation.

As under Equipartition, processor preemption is performed in a coordinated manner. However, Dynamic forcibly preempts a processor when it needs one to give to another job, as under Tucker and Gupta's [22] process control policy. Both schemes stand in contrast to Equipartition's policy of requesting that a job give up a processor at the next convenient point in its computation. Forced preemption runs the previously described risk of stalling the progress of the preempted job (if the user-level thread that was running is critical to the progress of the job), and of seriously affecting overall system performance (e.g., by causing jobs to waste their allocations on useless spinning). Furthermore, with the policy as stated thus far, it is possible for deadlock to occur. However, these problems can be avoided by coordinating preemptions with the application.

While Dynamic performs preemptions in the same manner as the process control policy, we coordinate preemptions using a different scheme, one based on that of Anderson et al. [3]. When a job has a processor preempted from it, but still has at least one processor allocated, it is notified immediately of the preemption. In response to this notification, the job determines whether the thread that had been running was *critical* or *noncritical*. A critical thread might be one holding a lock, for instance; a noncritical thread is one that was computing, but whose failure to progress immediately would not impede the progress of other threads. If the preempted thread is critical, the application can cause rescheduling of its threads on its remaining processors to restart the suspended thread (at the expense of suspending some other, currently running thread). Thus, Dynamic coordinates preemptions by allowing the application to decide which of its virtual processors should run when a preemption occurs, avoiding the performance degradation that could arise if the virtual processors were to be scheduled by the kernel without regard to application needs.

3.3.1 *Encouraging Applications to Cooperate with the Dynamic Policy.* The Dynamic policy described thus far relies on the application both to flag processors that it is not using actively and to indicate accurately how many more processors it could use when its parallelism exceeds its allocation. In a real system, unscrupulous users might try to maximize their individual benefit at the expense of overall system performance by misrepresenting this information. For instance, a job might declare that it can use all the processors, even at times when it cannot, and might never flag unused processors. (Note that the kernel cannot reliably distinguish an unused processor from a busy one, since a scheduler thread spinning waiting for ready application threads will make the processor appear busy.)

In anticipation of this problem, we augment the basic Dynamic policy with an adaptive priority mechanism. Each job is assigned a priority level that depends on its processor usage to that time. Let $l$ be the priority level of the job to be allocated a processor, and $L$ the lowest priority level of any job in the system. Allocation rule A.3 above is modified so that processors may be preempted from the largest job at priority level $L$ if $L < l$, or, if $L = l$, from the largest job at that priority level if its allocation is two or more greater than the job under consideration. In no case are processors preempted from any higher-priority job. (Rules A.1 and A.2 are applied independently of priorities.) By setting priorities in a way that ensures a low priority for any job that is greedy beyond its true needs, the priority-based allocation decisions discourage unscrupulous behavior.

We set job priorities using a scheme that raises them as a reward for using few processors and lowers them as a result of using many. In this way, a job acquires "credit" during periods when it uses few processors. The job may "spend" these credits later—possibly even during a period of high processor demand—to obtain temporarily more than its fair share of processors.

We define job $j$'s accumulated credit at time $T$ as:

$$credit_j(T) = \frac{\int_{s_j}^{T} \left( EQ(t) - A_j(t) \right) dt}{T - s_j}$$

where $A_j(t)$ is the job's allocation at time $t$, $EQ(t)$ is the system equipartition (number of processors divided by the number of jobs in the system) at time $t$, and $s_j$ is the start time of the job. Thus, job $j$'s credit is equal to the mean number of processors above or below the equipartition allocation that it has used during its lifetime. Using the rate of credit, rather than total accumulated credit, eliminates certain undesirable behavior. For example, under the latter scheme, a job may gain enormous credit during a period of low contention, which it may spend to monopolize the machine during a period of high contention; this is impossible when a rate-based scheme is used.

It is important to note that while this mechanism resembles the multilevel feedback scheduling common in many time-sharing systems, its purpose is really quite different. Multilevel feedback is used to give preference to short jobs over long ones. This is done by dropping a job's priority as it accumulates processor time, on the (empirically verified [13]) assumption that a job that has already run a long time will continue to run a long time while a newly arriving job is likely to be short. The adaptive priority mechanism described here, on the other hand, is simply a way to encourage jobs to give up processors not currently needed and has nothing to do with discriminating among jobs based on expected job length. (Such discrimination could be inserted into the policy quite simply through redefinition of the priority function. However, to our knowledge no clear relationship between accumulated and expected remaining running time has been shown for parallel workload, nor is it clear how to make use of this information were it available. This topic is an area of active research [14, 17].) The Dynamic policy as defined here is a modification of a policy proposed by Zahorjan and McCann [25].

## 4. THE PROTOTYPE IMPLEMENTATION: MINOS

We implemented our prototype, Minos, in C++ on a Sequent Symmetry (a bus-based shared-memory multiprocessor) consisting of 20 Intel 80386 processors and running the DYNIX 3.0.16 operating system. While each policy requires its own implementation, many functions must be supported by all of them. To the extent possible, we shared code among the three implementations, both to reduce implementation effort and to minimize non-policy distinctions in the implementations that might affect performance. In the discussion that follows, we use the name Minos to refer to those aspects of the implementation that are common to all three allocators.

While a production processor allocator would most likely be implemented as a distributed kernel function, Minos runs at the user level as a single process on a dedicated processor. Ease of implementation and availability of debugging tools were our primary motivations for implementing in this way as opposed to modifying the DYNIX operating system. Since the processor allocators are implemented at the user level, they cannot perform scheduling operations directly. To achieve their goals, then, they create situations in which DYNIX will decide to schedule processes in the way the allocator wants. This is accomplished primarily through the use of two (modified) system calls. These calls allow a process to be bound to a particular processor, either by the process itself or by the processor allocator. Because DYNIX gives preemptive priority on a processor to any process that has been bound there, the allocators can arrange processor allocation by binding and unbinding processes to processors.

Each allocation policy requires some amount of interaction between the processor allocator and the applications. The application's side of these interfaces was implemented in Minos by modifying slightly the PRESTO user-level threads package [5], which is included as library routines by applications.

Communication between an application (via the PRESTO runtime) and a processor allocator takes place through a shared-memory segment. With one exception, the allocator only reads the information in the shared segment, and the application only writes it. (The exception occurs under Equipartition, as described below.)

## 4.1 RRJob Implementation

The shared-memory interface between the applications and the RRJob allocator is used to communicate the value of the maximum parallelism of the job.

The RRJob policy requires a parameter, $Q$, representing the number of processor seconds of service in a scheduling quantum. In our experiments $Q$ was set to slightly more than 4.68 seconds. This length was chosen because, by the definition of RRJob, it implies that the duration of a quantum could vary from 4.68 seconds (for a job using one processor) to 293 msec. (for a job using 16 processors); the ratio of quantum duration to processor reallocation time under Minos (see Section 4.4) can thus vary from 2128 : 1 to 133 : 1. This range was designed to be consonant with typical quantum lengths for multiprocessors (for example, DYNIX, with a 0.75 msec. reallocation time, uses a 100 msec. quantum (a ratio of 133 : 1)) and ensures that reallocation overhead always contributes less than 1% to the response time obtainable under RRJob.

While this choice of quantum seems justified, we observe that another choice may well have been equally justifiable. One of the shortcomings of the RRJob policy is that it is difficult to determine what the "proper" value of $Q$ should be. We therefore ran experiments for values of $Q$ ranging from 9.6 seconds and 600 msec. These different values of $Q$ did not produce significantly different performance, in accordance with the expectation that the influence of preemption overhead is slight. For brevity, Section 6 presents only the results for $Q = 4.68$ seconds.

## 4.2 Equipartition Implementation

The implementation of Equipartition follows in a straightforward way from the definition of that policy (Section 3.2). In addition to the maximum-parallelism information, the shared-memory interface with the allocator is used to control processor preemption. When it is necessary to preempt a processor from a job, Equipartition sets a bit to indicate that the next available processor should be relinquished by the job. Each scheduler thread checks this bit just before looking for a ready application thread to execute. When a scheduler thread finds the bit set, it writes its processor number into the shared-memory interface and suspends itself. When the Equipartition allocator sees the processor number, it reallocates the processor.

## 4.3 Dynamic Allocator Implementation

An application requests additional processors through the shared-memory interface. In our prototype, we ask for a number of processors equal to the number of ready (but not running) application threads. The PRESTO scheduler threads update the requested number of processors with each change to the number of ready application threads.

An application indicates that it has more processors than it can currently use via a per-processor bit in shared memory. When a scheduler thread executing on processor *p* finds the application thread queue empty, it sets a bit notifying the allocator that processor *p* may be preempted. If not suspended by the Dynamic allocator, the scheduler thread continues to examine the set of ready application threads and resets the bit if an application thread is subsequently found. In this way, if no other job requires the processor, the application thread can begin execution without a processor reallocation penalty.

Detection of asynchronous blocking events, such as I/O or page faults, by the user-level allocator requires special handling. DYNIX does not notify the allocator when such events occur, so an explicit (and somewhat indirect) mechanism must be used. The allocator achieves this by binding an "idle" process to each processor behind the virtual processor assigned there. The idle process simply sits in the queue for that processor until the virtual processor stops executing for any reason, at which point the idle process becomes active. It notifies the Dynamic allocator (via a bit in shared memory) that it has run and then executes a *wait*. When the Dynamic allocator notices that the idle bit for a processor is set, it can conclude that the virtual processor that had been running on that processor has blocked for some reason and that the processor is available for reallocation to some (possibly the same) job.

Detecting that a previously blocked virtual processor has become unblocked (e.g., its I/O operation has completed), also requires an indirect mechanism. At the time the allocator finds the idle bit for a processor set, it unbinds the virtual processor that was running there and sends it a *stop* signal. When the virtual processor is runnable once again (e.g., it returns from I/O), it resumes execution (on a processor not controlled by the allocator) and catches the *stop* signal. The signal handler runs briefly, setting a bit in the job's shared-memory interface to the allocator, and then suspends itself. The set bit lets the Dynamic allocator know that the virtual processor is once again ready to be scheduled.

When a user-level thread suspended as a result of processor preemption or I/O is once again ready to run (this happens immediately in the case of preemption), the job's preemption handler is used to indicate to the Dynamic allocator the "criticality" of the user-level thread executing at the time of preemption. In our prototype implementation, we use a very simple concept of thread criticality:

(1) **Critical**. The user-level thread executing on the processor was holding a lock, and so the virtual processor must resume execution immediately. If no additional processor is available for allocation to the job, the allocator preempts a processor allocated to the same job for this purpose. If the newly suspended virtual processor is also running a critical thread, the procedure is

repeated: first an additional processor is allocated to the job if possible, and failing that, another processor within the same job is preempted.

(2) **Noncritical**. No locks were held. If the virtual processor was executing an application thread, that thread is placed back in that job's pool of ready threads. (This may cause the job to request additional processors.)

## 4.4 Processor Reallocation Overhead

Table II lists the measured cost of reallocating a processor under Minos for each policy implementation. Processor allocation overhead includes the cost of finding a processor to assign and the delay associated with activating one of the job's virtual processors to run on it. These components vary depending on whether or not the processor must be preempted from another job.

For allocation of a currently unallocated processor, each of the policies performs similarly. The main components of this overhead are the DYNIX context switch overhead (about 750 $\mu$sec.) and the kernel cost of binding a process to a processor (about 500 $\mu$sec.). Equipartition incurs a small additional delay due to complexity of the computation of each job's allocation.

For allocation of a preempted processor, the virtual processor currently executing there must be suspended before any other virtual processor can run. For RRJob, this includes the cost of a preemption signal (about 100 $\mu$sec.) and the cost of unbinding the preempted process (about 500 $\mu$sec.). For Equipartition, this includes the unbinding cost and the time for Equipartition to notice that the process has suspended. Not included in the table for Equipartition is a nondeterministic delay before the next thread completion for a job.

Processor reallocation for preempted processors is twice as expensive for the Dynamic allocator. In addition to the cost of suspending and unbinding the currently running process (500 $\mu$sec.), an average of 600 $\mu$sec. is spent first looking for an available (unallocated or idle) processor. Most of the allocation cost, however, is attributable to the cost of handling asynchronous blocking events such as I/O or page faults at the user-level level. The major components of this include the overhead to schedule the idle process (900 $\mu$sec.), the delay before the Dynamic allocator notices that the idle bit is set (300 $\mu$sec.), and the kernel call executed by the Dynamic allocator to retrieve the status of the preempted process (500 $\mu$sec.). As a result, processor reallocation under our Dynamic allocation policy implementation is very expensive relative to the other policies and relative to the cost of context switching within the kernel (750 $\mu$sec.).

This high reallocation cost does not significantly affect our results. As previously described, reallocation overhead adds no more than 1% to the response time under RRJob and is thus not a factor when comparing RRJob to either Equipartition or Dynamic. The results for Dynamic versus Equipartition are somewhat conservative, since the former reallocates far more frequently than the latter.

Table II. Summary of Allocation Overhead Costs

| Allocator | Unallocated Processor | Preempted Processor |
|---|---|---|
| RRJob | 1.3 msec. | 2.2 msec. |
| Equipartition | 1.5 msec. | 2.0+ msec. |
| Dynamic | 1.3 msec. | 4.4 msec. |

+ not including thread completion delay

## 5. APPLICATIONS

We describe here the applications used in our evaluation of the allocation policies. We chose a set of programs previously written for shared-memory multiprocessors in order to model realistic workloads. These programs represent a variety of applications with differing parallelism structures.

Figures 1 through 4 illustrate a number of characteristics of the applications. Included for each application is the thread dependence graph. Here the nodes of the graph represent application threads, and the edges represent the precedence relationships among them. Also shown is the percentage of elapsed time spent by the application at each level of physical parallelism when the application was run in isolation on 16 processors. Finally, the total execution time and average allocation of the application when run in isolation on 16 processors are listed.

The first application, MATRIX, is an implementation of a parallel matrix multiply algorithm. This application represents the broad class of highly parallel programs. The program uses a "blocked" algorithm designed to improve performance by exploiting cache locality [8]: the matrices to be multiplied are divided into fixed-size blocks, and partial results are computed for the corresponding blocks in the result matrix. After a brief initialization phase, the main loop for MATRIX achieves parallelism by spawning one application thread for each block in the result matrix. In our experiments, we multiply two 360-element-by-360-element matrices, using 144 blocks of 30 elements by 30 elements each. Thus, after the sequential initialization phase, MATRIX maintains a maximum parallelism of 16 processors until a very short end effect during termination.

The second application, PVERIFY, is a parallel implementation of a CAD program that determines whether or not two circuit implementations are functionally equivalent. Like MATRIX, PVERIFY is a highly parallel program. However, PVERIFY exhibits a significant sequential phase during which the data structures for the circuits are initialized.

The third application, MVA, is a parallel Mean Value Analysis [1, 20] solution package for product form queuing networks containing two classes of $N$ customers each. Its precedence structure is representative of many "wave front" computations. As shown by its precedence graph, its parallelism at first increases gradually from one to $N + 1$ and then decreases back to one. The application repeats this computation $I$ times, reflecting the iterative

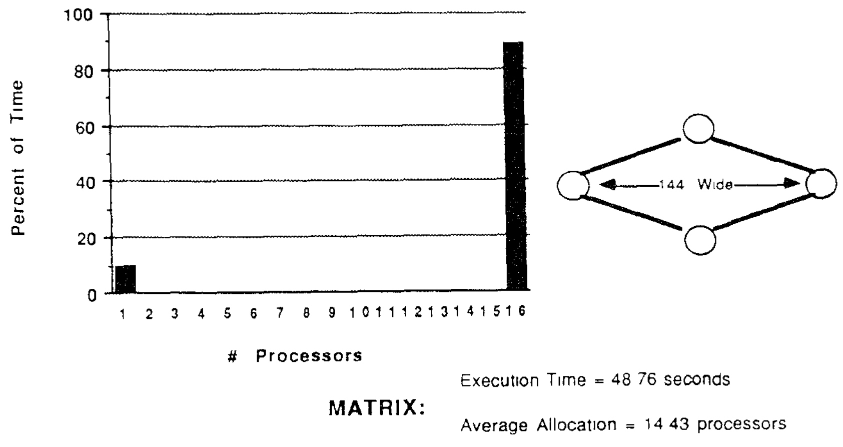


Fig 1. The MATRIX application.

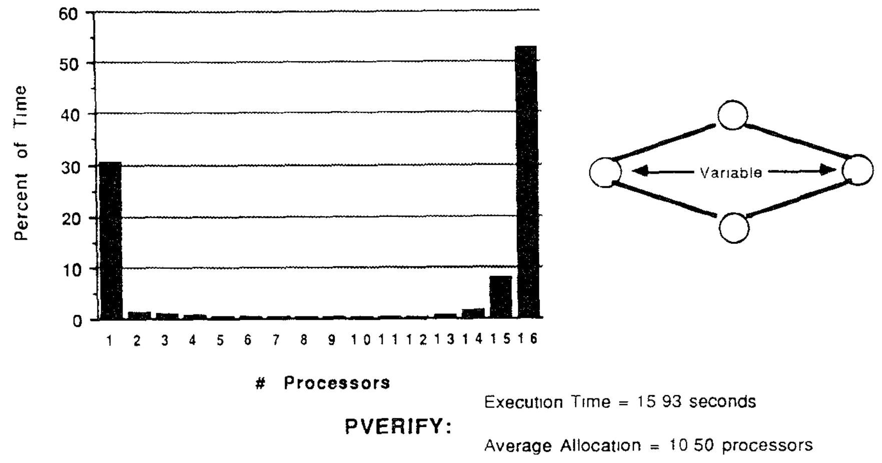


Fig. 2. The PVERIFY application.

analysis techniques common in the use of this kind of queuing model. In our experiments, we chose an input of $N = 10$ customers and $I = 30$ iterations.

The final application, GRAVITY, is an implementation of the Barnes and Hut [4] clustering algorithm for simulating the gravitational interaction of a large number of stars over time. In our experiments, we model a system of 400 stars for 20 timesteps. This application contains five phases of execution that are repeated for each timestep. Phase 1 is sequential, while phases 2, 3, 4, and 5 exhibit parallelism of 8, 8, 16, and 10 respectively. (These represent the parallelisms that were determined to minimize the elapsed time to finish the phases in a uniprogramming environment.) Between each of the parallel phases is a barrier synchronization at which the parallelism

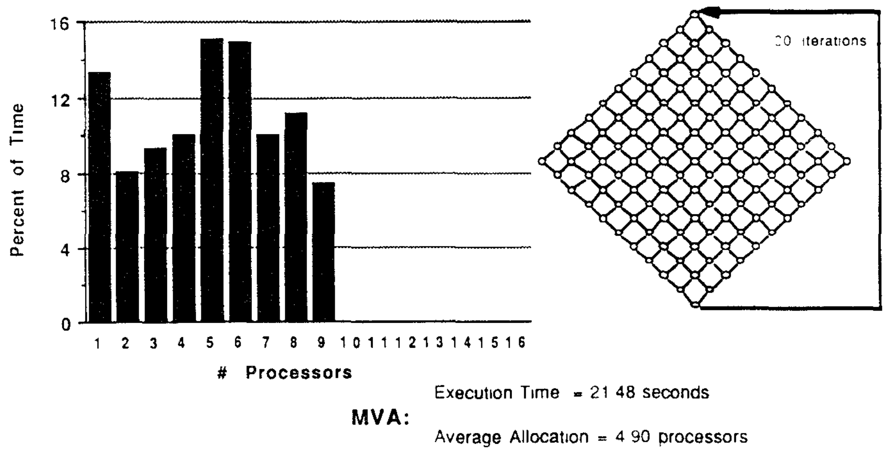


Fig. 3. The MVA application.

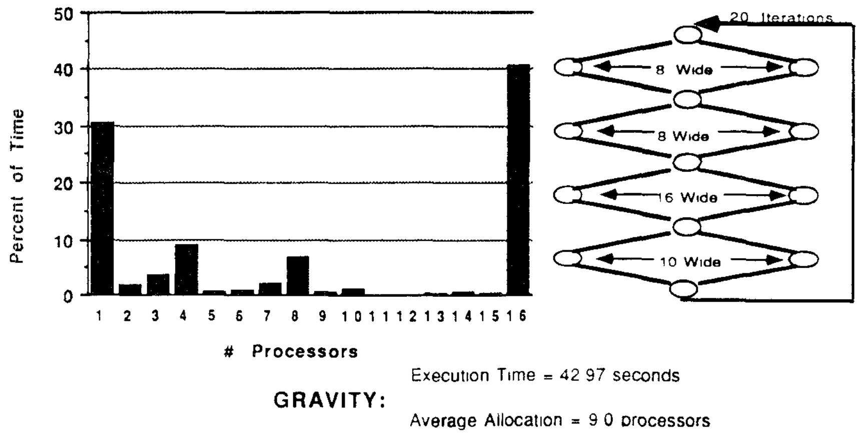


Fig. 4. The GRAVITY application.

decreases briefly to one. The thread execution times for GRAVITY differ during each phase of execution, and within some phases, thread times depend on synchronization delays for critical sections of code.

## 6. RESULTS

We compare the three allocation policies along a number of dimensions (average response time, fairness, and response time to short sequential requests), but stress average response time. Our results are based on experiments with workloads consisting of a mix of the applications described in Section 5.

### 6.1 Workload Mixes

Each workload mix used in our experiments is composed of some number (possibly zero) of jobs of each of the four application types discussed in the previous section.

We set the number of jobs of each type so that there is moderate competition for processors, which we define as total average processor demand (found by summing the average parallelisms of the jobs in the workload mix) between $P$ and $2P$. (In our experiments $P$ was 16). We restrict our attention to moderately loaded systems because we believe they present the most interesting challenges for schedulers. Under light loads, all reasonable scheduling policies behave the same. Under loads for which the number of jobs is less than the number of processors but the number of runnable tasks is much greater, our results indicate that the "best" policy is simply to allocate each job an equal number of processors. Additional scheduling questions do arise when the number of jobs exceeds the number of processors, but we believe that this situation will be rare, even for the moderately parallel processors we consider here. (In this case average realized speedups would be less than one, rendering the parallel machine unattractive except potentially for jobs requiring very large amounts of memory or I/O.) Further exploration of this very heavy load case remains an open problem.

Given the four job classes previously described, there are 15 possible choices of which classes may be included in a workload (the workload containing zero classes is degenerate). Of these 15 possible combinations of job classes, it is possible to choose job counts meeting the moderate load criterion for 13 of them. When there is more than one way to satisfy this criterion for a given workload mix, we chose the combination for which the sum of the jobs' average parallelisms was smallest. Table III summarizes the workload mixes used.

For each workload mix, we maintain a constant multiprogramming level: when a job finishes, another instance of the same application type is started immediately. We intend this to be a means of focusing our attention on those periods during normal system operation when jobs are in the system (and so scheduling is of interest) rather than an assertion that systems typically run with constant workloads. By maintaining a constant multiprogramming level we also avoid the end affects that would otherwise arise when running applications of different durations.

For similar reasons, we do not vary the jobs' maximum parallelisms: this additional parameter would have greatly increased the effort required to run our experiments, as well as the complexity of interpreting their results. Previous experience with programs of different sizes convinced us that the results for these workloads were representative.

### 6.2 Overall Performance

Our primary metric of performance is average job response time. (Speedup, a common metric of parallel software performance, is not particularly intuitive in our environment since a job's processor allocation changes throughout its

Table III. Number of Copies of Each Application in Each Workload Mix

| | Workload Mix | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| **MATRIX** | 2 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| **PVERIFY** | 0 | 2 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| **MVA** | 0 | 0 | 4 | 0 | 0 | 1 | 0 | 2 | 0 | 2 | 1 | 1 | 1 |
| **GRAVITY** | 0 | 0 | 0 | 2 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| **T.P.D.** | 28.9 | 21.0 | 19.6 | 18.0 | 25.6 | 20.0 | 24.1 | 20.3 | 19.5 | 18.8 | 29.8 | 28.3 | 24.4 |

**T.P.D.** = Total Processor Demand

lifetime.) While we report only point estimates for response times, we ran enough replications of each experiment that the 95% confidence interval is within 1% of the point estimate of the mean.

The results for all workload mixes under all three scheduling policies are shown in Appendix A. (As explained in Section 3.2, the response times observed under our Equipartition policy should be equal to or slightly better than those that would be observed using the preemption policy of Tucker and Gupta [22].) Based on these, we come to the following conclusions.

*Space Sharing is Preferable to Time Sharing.* This point can be illustrated by comparing the performance of RRJob to Equipartition for any workload that does not include any MVA jobs. Since for the three other types of jobs the maximum parallelism is equal to the number of processors in the system, under RRJob the entire machine is swapped from one job to the next at each quantum expiry. Thus, the uncoordinated nature of RRJob's preemptions has no influence for these workloads, and performance differences between it and Equipartition must be due to the difference between space and time sharing. (As explained earlier, the overhead of actually performing the preemptions is insignificant.)

As a concrete example of this comparison, we use the workload of two GRAVITY jobs. The results for this workload are shown in Figure 5. Results for other workloads suitable for this comparison (i.e., those not including MVA jobs) are found in the appendix and are qualitatively very similar (see Figures A.1, A.2, A.5, A.7, and A.9).

Space-sharing policies dominate time-sharing policies because they make more efficient use of processors. Time sharing makes large, short-lived allocations. Because most parallel applications have speedup functions that are convex (efficiency decreases with increasing number of processors), this is usually a bad choice.

There are two distinct factors that lead to convex speedups, and thus the preference of space over time sharing. First, time-sharing policies maximize both the likelihood that there will be periods of insufficient parallelism and

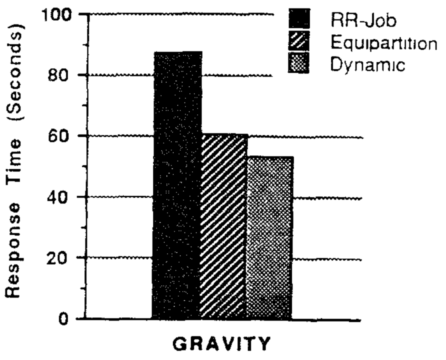


Fig. 5. Workload mix of 2 GRAVITY jobs.

the number of processors wasted during such periods. In our example, the RRJob allocator assigns all sixteen processors to each job in turn. During a GRAVITY job's sequential phase, 15 of the 16 processors remain idle. Equipartition, on the other hand, divides the processors equally between the jobs, so that during one job's sequential phase, only seven processors remain idle.

Second, even if the job has no periods of insufficient parallelism, there may be overheads that grow with the number of processors in use. These overheads result from increased spinning to enter critical sections [27], inefficient synchronization implementations [2, 9], and hardware effects such as increased bus contention and cache invalidation rates [11].

We quantify this second aspect by measuring the rate of useful work completed per processor as processor allocation increases for the four parallel phases of a GRAVITY job. We define the effective work rate $E_\alpha(n)$ for a phase $\alpha$ given an allocation of $n$ processors as:

$$E_\alpha(n) = \frac{T_\alpha(1)}{T_\alpha(n)^*n}$$

where $T_\alpha(n)$ is the elapsed execution time of phase $\alpha$ on $n$ processors. (Note that despite the apparent relation to efficiency, this is not efficiency in the standard sense since we condition our measurements on phases of execution during which the job has parallelism at least $n$.)

Figure 6 graphs $E_\alpha(n)$ for the four parallel phases of GRAVITY for numbers of processors ($n$) from one to 16. As shown, the relative effective work rate for three of the four parallel phases decreases sharply with increasing allocation.

*Coordinated Preemption is Preferable to Uncoordinated Preemption.* We come to this conclusion because providing coordinated preemption is not a detriment to any job (there is insignificant overhead required to coordinate preemptions, relative to uncoordinated preemptions) but can be a great benefit to at least some jobs.

We illustrate the benefit of coordinated preemption by comparing the performance of RRJob to Equipartition and Dynamic for a workload mix

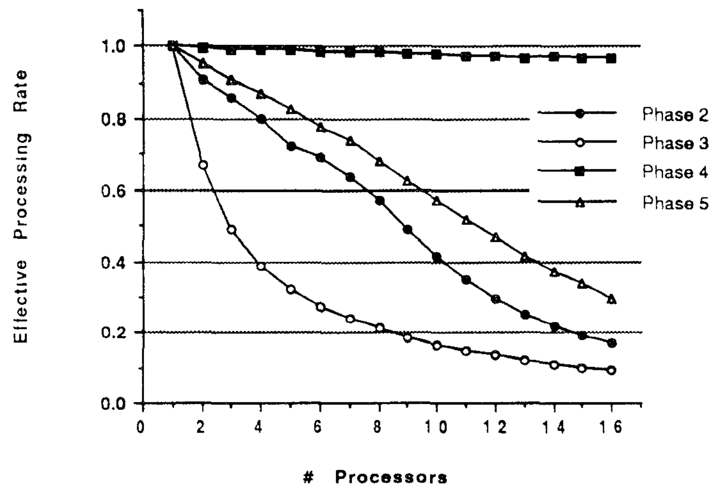


Fig. 6. Effective work rates of parallel phases of GRAVITY.

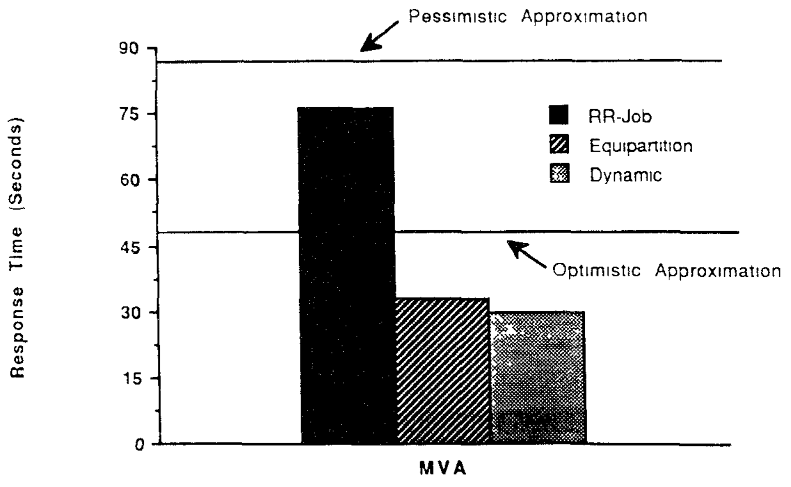


Fig. 7. Workload mix of 4 MVA jobs.

consisting of four MVA jobs, as shown in Figure 7. We note that while Equipartition and Dynamic give very similar performance, RRJob performs much worse. Most of this difference can be attributed to the manner in which preemptions are handled.

To help isolate and quantify the effect of uncoordinated preemption, we perform a simple analysis. We begin by noting that a pessimistic approximation for response time under RRJob is simply four times the response time of an MVA job when run alone on the system, since it is allocated the full machine one fourth of the time. This pessimistic approximation (calculated from the base execution time given in Figure 3) is indicated in Figure 7; we see that achieved performance is very nearly this bad.

On the other hand, there is reason to expect response time to be considerably better than this. During an "allocation cycle" of four quanta, each job is allocated 11 processors during its quantum, five processors during the quantum of the job ahead of it (since that job can use only 11), and no processors during the other two quanta. The extra allocation of five processors each four quanta should result in reduced response times.

To help quantify the maximum benefit possible from the extra allocation, we compute an optimistic approximation of response time under RRJob. Let $T(n)$ be the elapsed execution time of an MVA job when allocated $n$ processors. Then $Q/T(11) + Q/T(5)$ is the fraction of a job completed during an allocation cycle, where $Q$ is the length of an individual quantum, and the approximation is given by:

$$R_{optimistic} = 4Q^* \left( \frac{Q}{T(11)} + \frac{Q}{T(5)} \right)^{-1} = 4^* \frac{T(11)^* T(5)}{T(11) + T(5)}.$$

We measured $T(11)$ and $T(5)$ to be 21.48 and 27.58 seconds respectively, giving an optimistic approximated response time of 48.30 seconds for the MVA job.

The failure of RRJob to come close to this approximation indicates that the MVA job is unable to take advantage of the extra allocation of five processors each four quanta. The reason for this is that uncoordinated preemption suspends user-level threads along with virtual processors. When an MVA job receives a partial allocation, some of these user-level threads remain preempted. The precedence relations of an MVA job, as shown in Figure 3, prevent the job from making much progress while any user-level threads remain preempted.

*Dynamic Allocation is Preferable to Static Allocation.* We illustrate the benefits of dynamic allocation over static allocation by comparing the performance of the Equipartition and Dynamic policies, since these policies differ primarily in the frequency of processor reallocations among jobs in the system.

Figure 8 shows the results for a workload mix consisting of a MATRIX, an MVA, and a GRAVITY job. While Equipartition and Dynamic result in the same average allocation of processors to the jobs, Dynamic yields response times that are 6.1%, 10.4%, and 13.5% better than Equipartition for MATRIX, MVA, and GRAVITY respectively. Thus all jobs benefit from the improved processor efficiency attained by reallocating processors when they cannot be used.

We reiterate that our results were obtained using a user-level implementation in which reallocation overheads are large (see Section 4.4), and we therefore believe that they are conservative with respect to the performance possible using a kernelized implementation. The Dynamic policy results in far more processor reallocations than occur under (the more static) Equipartition, and more efficient reallocation could only increase the performance of the former. Thus, our results almost certainly understate the benefits of dynamic scheduling.

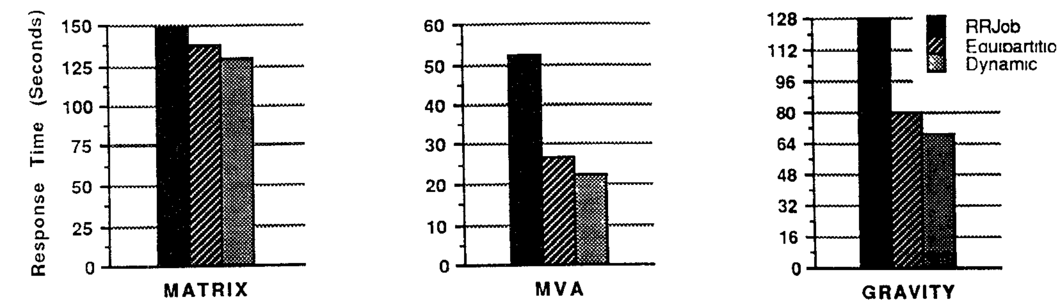


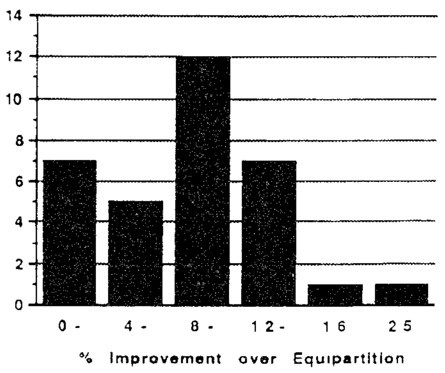
Fig. 8. Workload mix of 1 MATRIX, 1 MVA, and 1 GRAVITY job.

*The Dynamic Allocator, Characterized by Space Sharing, Coordinated Preemption, and Dynamic Reallocation, Outperforms the Other Allocators.* The benefit of the Dynamic allocation policy over the other policies is evident in the aggregate of results for all job mixes in Appendix A. As a summation of these experiments, Figure 9 shows a histogram of the percentage improvement of the Dynamic policy over Equipartition and RRJob for all workloads in Appendix A. The average percentage improvement over Equipartition and RRJob is 8.8% and 36.9% respectively. The Dynamic policy outperforms the Equipartition and RRJob policies for each job in each workload mix with only two exceptions, where it performs identically to Equipartition for at least one of two jobs in a mix.

The two instances where Dynamic does not strictly outperform Equipartition are shown (later) in Figures A.1 and A.9. For the former, the workload consists of two MATRIX jobs. Here, Dynamic is unable to capitalize on the small sequential component of each MATRIX job to improve the performance of the other because of our very controlled experimental environment: we constantly run two identical jobs on the same data. In this case, the reward structure of Dynamic tends to synchronize the progress of the two identical jobs so that they receive equal service, with the result that both jobs enter their sequential components at the same time, thus eliminating the benefit of Dynamic over Equipartition. (This same effect probably artificially reduces the observed benefit of Dynamic in other workloads consisting of only a single job type. It would rarely, if ever, arise in practice because it requires that multiple copies of the same job be run concurrently and repeatedly on essentially identical data.)

In the other workload, consisting of one PVERIFY and one GRAVITY job, the PVERIFY job does noticeably better under Dynamic than under Equipartition, while the GRAVITY job does equivalently under both. Dynamic fails to outperform Equipartition for the GRAVITY workload by a small margin attributable to the larger number of processor reallocations that occurred under Dynamic, coupled with the relatively high cost of reallocation.

6.2.1 *Cache Behavior.* As processors become faster, an application's cache behavior has an increasing effect on its performance. For example, as



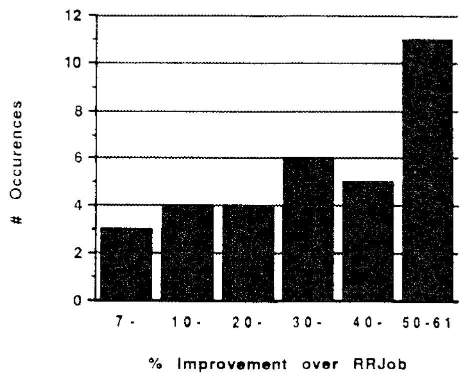


Fig. 9. Comparison of dynamic to equipartition of RRJob

discussed in Section 5, our MATRIX job uses a blocked algorithm designed to improve cache locality, and thus performance. Gupta et al. [11] observe that the process control policy (which makes the same allocation decisions as our Equipartition policy) increases the spatial locality of applications because the application's virtual processors execute on only a small number of processors. Equipartition's infrequent reallocations may also increase this cache locality. Our Dynamic policy, on the other hand, makes no effort to preserve an application's cache locality. For this reason, and because processors are reallocated frequently (as compared to Equipartition), one might expect the Dynamic policy to perform poorly with respect to application cache behavior.

We note, however, that cache locality is not a deciding factor for the bulk of our workloads. The Dynamic policy yields much higher processor utilization than does Equipartition, and the resulting performance improvement far outweighs any possible cache effect, as evidenced by the results above (and in Appendix A). With the two exceptions just noted (where performance is equivalent under the two policies), the Dynamic policy outperforms Equipartition for all of the workloads studied, cache effects notwithstanding.

A more comprehensive examination of the influence of cache effects on scheduling, including an estimation of their impact on future machines with faster processors and larger caches, is found in [24]. That work indicates that obtaining high processor utilization, as under the Dynamic policy, continues to outweigh the benefits of improved cache behavior available by more static scheduling, as under Equipartition, even for machines considerably more powerful than the Sequent used in our experiments.

6.2.2 *Eager Relinquishing of Processors under Dynamic.* One goal of the Dynamic policy is to keep as many processors as possible usefully busy by reallocating processors from jobs that cannot currently use them to those that can. Thus, when a scheduler thread finds that there are no application threads ready to run, it immediately makes its processor available for reallocation. While the willingness to give up the processor is indicated immediately, reallocation cannot employ the processor usefully in less than a reallocation overhead time (about 4.4 msec. under Minos for the Dynamic policy). Thus, if the job currently holding the processor were to produce additional useful work in less time than this, it would have been better simply to hold onto the processor by spinning.

Because there is no reliable way to determine how long it will take for additional work to be produced by the application, the choice between spinning and relinquishing can be a difficult one. Exactly these same considerations arise in implementing locks for mutual exclusion, and in this context a number of researchers have proposed a "spin-then-block" policy, the basic idea being to spin for a short time and then, if no work has arrived, to give up the processor [12, 15]. (In the context of spin locks, "arriving work" is simply the lock becoming free.)

Based on our knowledge of the GRAVITY application, we hypothesized that immediate release of processors by it resulted in a very large number of processor reallocations and was the reason GRAVITY performed no better under Dynamic than under Equipartition in the workload mix consisting of one PVERIFY and one GRAVITY job. (See Figure A.9.) We therefore investigated the benefit of briefly waiting for new work to be generated within an application before making a processor available for reallocation.

Figure 10 shows the number of preemptions and the response times relative to using no delay for various spin delay values for the PVERIFY/GRAVITY workload mix. The results show that while the number of preemptions drops significantly, the effect on response time is extremely modest. (The response time results appear jagged because of the very small differences involved.)

We tried this same experiment on a large number of workloads, always with the same result: while preemption counts were often reduced, response times were never greatly improved. We also noticed that any reasonable spin delay greater than zero gave very similar results, i.e., that there is little danger in using too large a delay. This characteristic, coupled with the steep decline in preemption count that occurs in going from a zero to a positive delay (see Figure 10), indicates that a small delay could be applied uniformly by all workloads without ill effect. (Applying the delay uniformly simplifies the role of the runtime library that implements user-level threads.)

## 6.3 Fairness

We now turn our attention to another aspect of scheduling: fairness. Our goal in this section is to examine the question of fairness quantitatively. Unfortunately, there is no standard measure by which to evaluate fairness. We test what may be considered a minimal condition to be satisfied by a fair policy: that identical jobs experience identical response times when submitted together. Note that this goal is implied by any policy that attempts to give equal service rates to all jobs.

To evaluate fairness, we examine a workload consisting of two MATRIX jobs and one MVA job. The maximum parallelism of the MATRIX jobs is 16, while that of the MVA job is 11. Figure 11 shows the response times of the

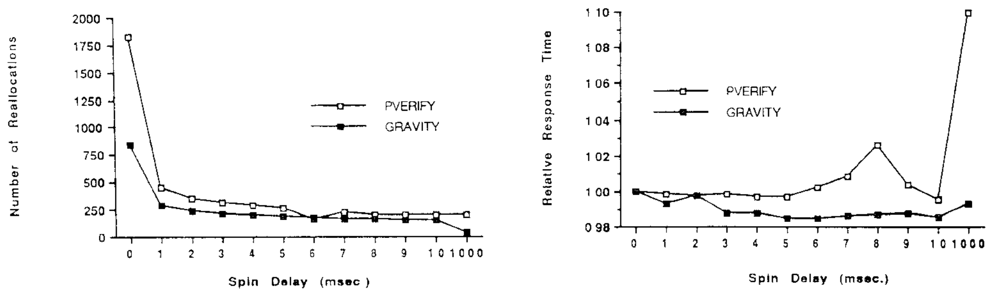


Fig. 10. The effect of delay on preemptions and response time.

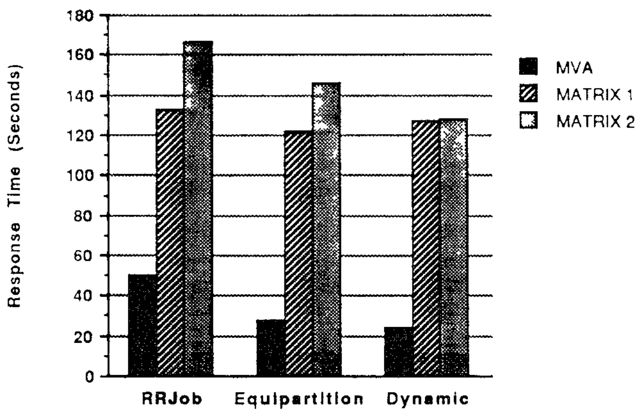


Fig. 11. Examining the fairness of the policies.

three jobs, and we now discuss the behavior of the three policies in this context.

The RRJob policy of allocating unused processors to the next job in line can result in significant unfairness. When the MVA job is active, it holds only 11 of the 16 processors; the remaining 5 processors are assigned to the next job in the queue for the duration of MVA's quantum. Therefore, the MATRIX job that follows the MVA job receives 16 processors during its own quantum, and 5 during MVA's. However, while a MATRIX job is active, it holds all 16 processors, leaving none available for reassignment. The second MATRIX job—which follows the first in the job queue—therefore receives exactly 16 processors, for exactly the time specified by the policy. As demonstrated by Figure 11, this inequity results in much shorter response times for one of the MATRIX jobs.

The Equipartition policy occasionally demonstrates unfair behavior, as this workload illustrates. With 3 jobs competing for 16 processors, Equipartition assigns each job 5 processors, leaving 1 "extra" processor. This processor is then assigned to one of the jobs—in this case one of the MATRIX jobs. As Figure 11 shows, this results in improved response times for the MATRIX job that has this advantage. While this unfairness can be overcome by circulating the extra processors among all jobs [10], it introduces an additional level of complexity and overhead to that scheme to handle a condition that is dealt with naturally by the Dynamic allocator.

The Dynamic policy can, for short periods, exhibit unfair behavior in the same fashion as Equipartition. However, the Dynamic policy of moving available processors between jobs ensures that any unfair decision is short lived. For example, suppose, as in the case of Equipartition, that an extra processor were assigned to one of the MATRIX jobs. Under Equipartition, this means that one MATRIX job holds the extra processor until some job terminates, and a reallocation can be done. On the other hand, the Dynamic policy exploits the fact that the MVA job periodically releases processors that it cannot immediately use, something that occurs relatively frequently. The policy dictates that each released processor is assigned to the MATRIX job that currently holds the fewest processors, as the Dynamic allocator attempts to preserve an equitable allocation. It should be noted that this behavior does not depend on jobs to willingly relinquish processors (as MVA does). The priority mechanism of the Dynamic policy, while designed to encourage jobs to give up unneeded processors, also serves to enforce fairness. A job holding more than its equipartition allocation of processors will eventually drop in priority below the others, and one (or more) of its processors will be moved to the other (now higher-priority) jobs. From the response times of the two MATRIX jobs shown in Figure 11, it is evident that these components of the Dynamic policy—moving willingly relinquished processors when appropriate, and using the priority mechanism to enforce equipartition via forcible but coordinated preemption—are quite successful at preserving fairness.

This example and analysis lead us to the following conclusions. Both Equipartition and Dynamic have much better fairness behavior than RRJob, which suffers due to its policy of assigning unused processors to the next job in the queue without coordinating preemption. Equipartition is reasonably fair, but if the number of jobs does not evenly divide the number of processors, Equipartition can exhibit some unfair behavior. Since the Dynamic policy couples equipartitioning of processors with dynamic movement of available (or preempted) processors, it is the most fair of the three policies.

## 6.4 Response Time to Short Sequential Jobs

To examine quantitatively each policy's behavior with respect to short sequential jobs, we chose the following procedure. We maintained a constant workload of 1 MATRIX, 1 MVA, and 1 GRAVITY job. This workload was chosen because it was one of the more "realistic" workloads available to us: it contains both long- and short-running jobs, which exhibit parallelism that varies in both amount and frequency of change.

Into this workload, we periodically (every 30 seconds) injected a fourth job. This fourth job was sequential, executing a single spin loop of approximately 100 msec. duration and then terminating. This was meant to simulate the behavior of an interactive workload component. For example, during a long-running editor session, a user might periodically type some input, waking up

Table IV. Response Time to Short, Sequential Requests

| | RRJob | Equipartition | Dynamic |
|---|---|---|---|
| **Response Time** | 262.27 msec. | 2391.50 msec. | 113.24 msec. |
| **Queueing Time** | 158.75 msec. | 2288.69 msec. | 5.45 msec. |

the waiting editor program, which might take some short action before waiting for more input.

Table IV shows, for each of the policies, the response time and queuing time experienced by the sequential job, where queuing time is defined as the time between job arrival and processor assignment.

The queuing time under the RRJob policy is relatively short, because our implementation inserts newly arriving jobs at the head of the job queue. The new arrival therefore queues only until the current quantum expires, at which point it is assigned a processor. Queuing time under RRJob is therefore very predictable given knowledge of (a) the maximum parallelism of each job and (b) the processor reallocation time.

As seen in Table IV, queuing time under Equipartition is the highest of that of the three policies. This is due to the "polite preemption" mechanism used by our version of Equipartition to obtain processors. The newly arriving job queues until a currently executing thread completes, at which point it obtains that thread's processor. Since the processor allocator has no control over a job's threads, this delay can be arbitrarily long.

The Dynamic policy results in the shortest queuing time, only slightly greater than 1 processor reallocation time. (The preemption scheme advocated by Tucker and Gupta [22] would have preemption times similar to those shown for Dynamic, although it may suffer from other performance problems (see Section 3.2).) This is because the Dynamic allocator need not wait for external events such as quantum expiry or thread completion: it simply assigns a processor to the new arrival, preempting one if necessary. In general, Dynamic's priority mechanism makes prediction of queuing time difficult. However, this example illustrates that this time may be expected to be short.

## 7. CONCLUSIONS

We have implemented three processor allocation policies for shared-memory multiprocessors. These policies are based on policies proposed and modeled in the recent literature: Round-Robin Job (RRJob) [14], Equipartition [22], and Dynamic [25]. We have characterized the jobs according to the properties of time versus space sharing of processors, uncoordinated versus coordinated processor preemption, and static versus dynamic allocation decisions.

Based on our comparisons of the performance of these three policies, we come to the following conclusions:

*Space sharing is preferable to time sharing.* Time sharing gives many processors to individual jobs for short periods of time; for all the reasons that speedups are typically sublinear, this is an inefficient way to provide concurrency.

*Preempting processors in a coordinated way is critical.* Our implementation shows that there is insignificant overhead to coordinating preemptions, relative to uncoordinated preemptions. Our experiments show that in at least some cases there may be enormous differences in response times between coordinated and uncoordinated preemption.

*Dynamic scheduling is preferable to static.* Under dynamic scheduling, whenever a job could use an additional processor and some other job has a processor it cannot currently use, the processor is reallocated. Our results indicate that the increased useful-processor utilization achieved by dynamic scheduling more than compensates for the increased system overhead resulting from frequent processor reallocations.

We noted that it may be possible to improve even this superior performance by considering applications' cache behavior when making scheduling decisions; Vaswani and Zahorjan [24] focuses on modifications to the Dynamic policy to incorporate this notion. We explored a simple "spin-then-block" policy to reduce needless preemptions under the Dynamic policy that can result from short-term changes in a job's parallelism. We found that because preemption overhead is an insignificant component of response time, implementing a general modification to the policy produced only an extremely minor performance improvement.

We then examined two lower-level policy issues: fairness and response to short sequential requests. We found that Equipartition can be unfair because of inability to achieve equal allocation when the number of jobs does not divide the number of processors, while RRJob is susceptible to anomalies related to allocation of processors not needed by the owner of a scheduling quantum. Since the Dynamic policy couples equipartitioning and dynamic movement of processors, it is the most fair of the three. We showed that the Dynamic policy provides the best service to short sequential requests due to its use of forcible but coordinated preemption. RRJob and Equipartition are less responsive since they must wait for external events—quantum expiry and thread completion, respectively.

Our Dynamic policy, then, incorporates the preferable choice in each of the high-level policy issues: it combines space sharing, coordinated preemption, and dynamic movement of processors between jobs. This results in better performance than that of previously proposed policies. In addition, the Dynamic policy incorporates a priority scheme that, while it encourages jobs to be well behaved, does not require this in order to provide its superior performance. This, as demonstrated by our experiments with fairness, robustness, etc., makes our policy a realistic one suitable for implementation in production systems. We therefore conclude that this combination of performance and implementation considerations makes a compelling case for our Dynamic policy.

## APPENDIX A: RESULTS FOR ALL WORKLOAD MIXES

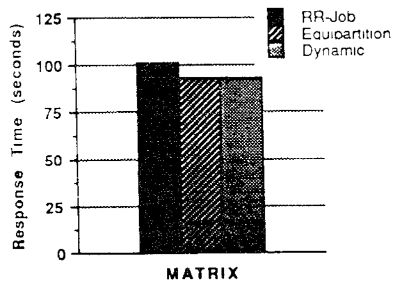


Fig. A.1. 2 MATRIX.

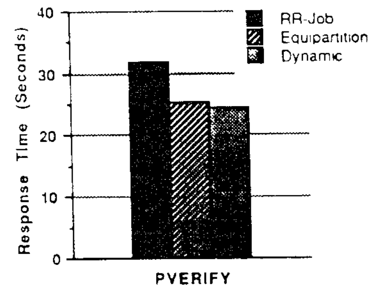


Fig. A.2. 2 PVERIFY.

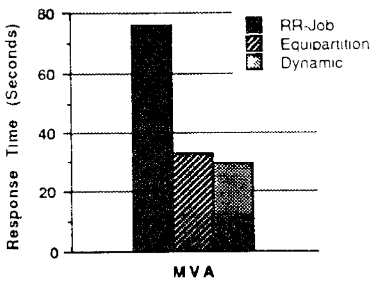


Fig. A.3. 4 MVA.

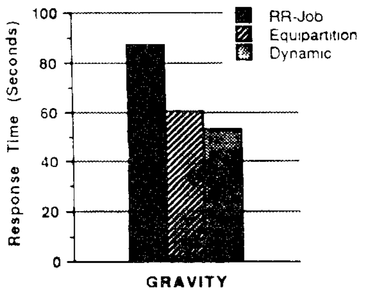


Fig. A.4. 2 GRAVITY.

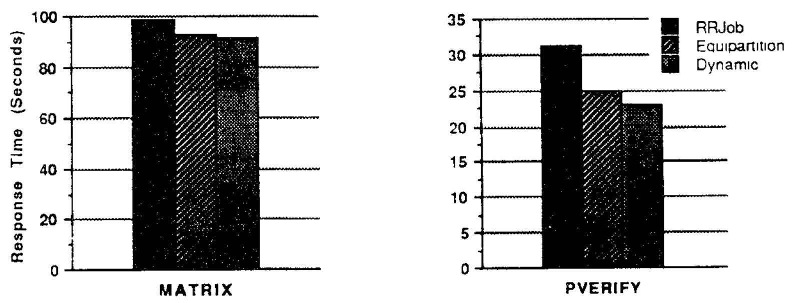


Fig. A.5. 1 MATRIX, 1 PVERIFY.

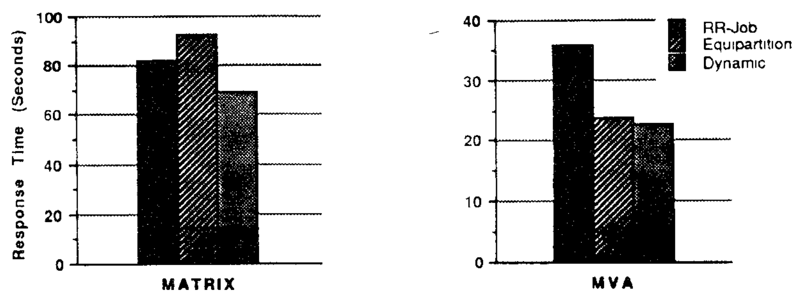


Fig. A.6. 1 MATRIX, 1 MVA.

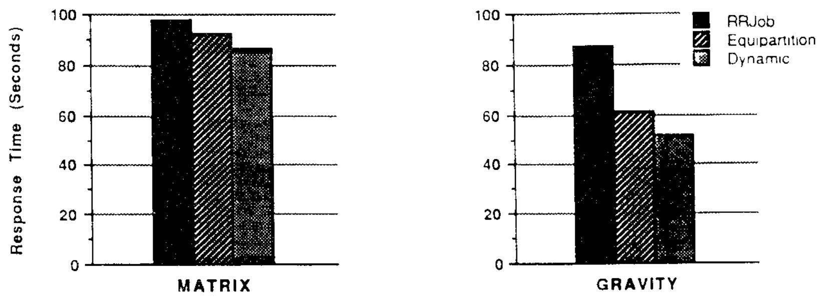


Fig. A.7. 1 MATRIX, 1 GRAVITY.

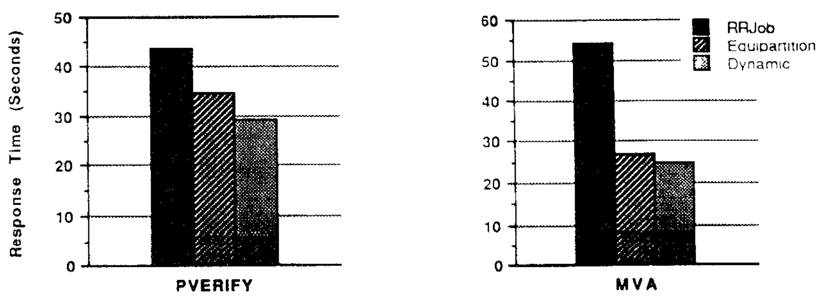


Fig. A.8. 1 PVERIFY, 2 MVA.

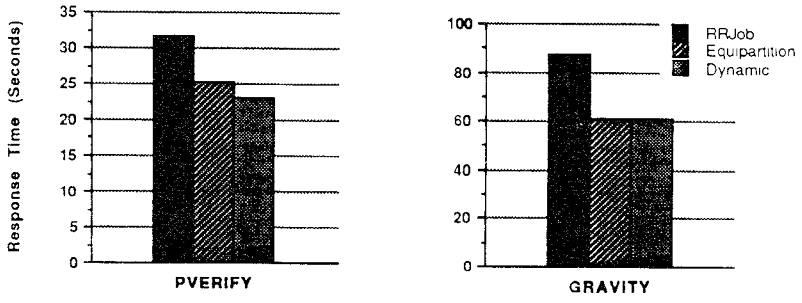


Fig. A.9. 1 PVERIFY, 1 GRAVITY.

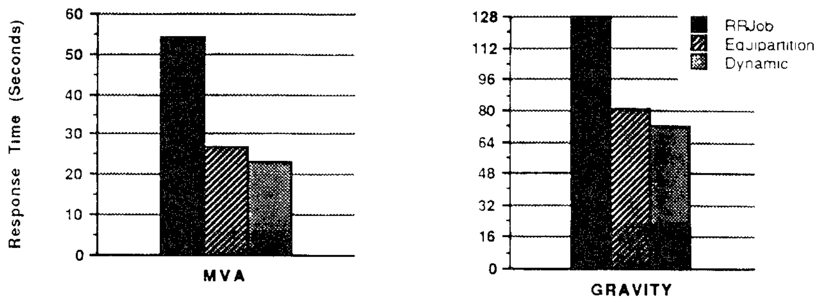


Fig. A 10. 2 MVA, 1 GRAVITY.

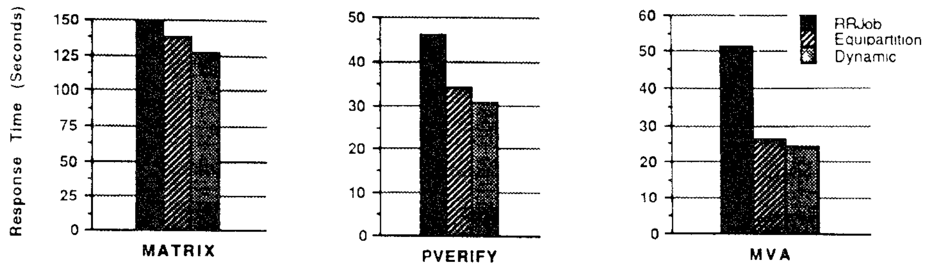


Fig. A.11 1 MATRIX, 1 PVERIFY, 1 MVA

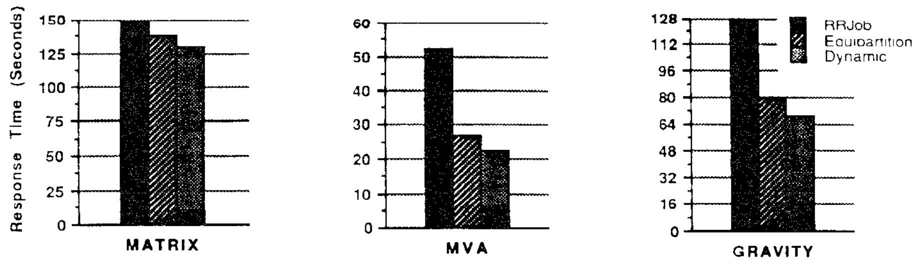


Fig. A.12. 1 MATRIX, 1 MVA, 1 GRAVITY.

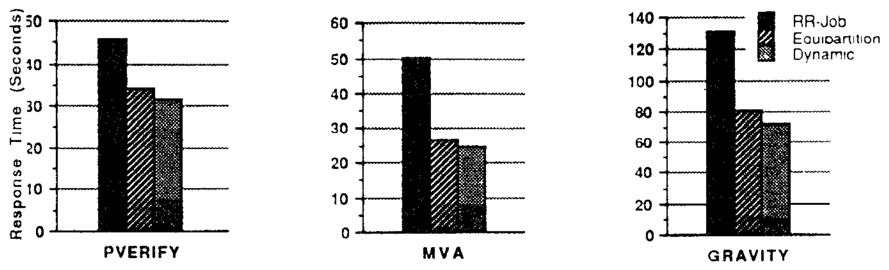


Fig. A.13. 1 PVERIFY, 1 MVA, 1 GRAVITY.

ACKNOWLEDGMENTS

Derek Eager, Ed. Lazowska, and the anonymous referees provided valuable comments on the content of this work and the presentation of this paper.

## REFERENCES

1. Almquist, K., Anderson, R. J., and Lazowska, E. D. The measured performance of parallel dynamic programming implementations. In *Proceedings of the 1989 International Conference on Parallel Processing* (Aug. 1989).
2. Anderson, T. E. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parall. Distrib. Syst. 1*, 1 (Jan. 1990), 6–16.
3. Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst. 10*, 1 (Feb. 1992).
4. Barnes, J., and Hut, P. A hierarchical $O(NlogN)$ force-calculation algorithm. *Nature 24* (1986), 446–449.
5. Bershad, B. N., Lazowska, E. D., and Levy, H. M. PRESTO: A system for object-oriented parallel programming. *Softw. Pract. Exper. 18*, 8 (Aug. 1988), 713–732.
6. Birrell, A. D. An introduction to programming with threads. DEC System Research Center, 1989.
7. Edler, J., Lipkis, J., and Schonberg, E. Process management for highly parallel UNIX systems. In *Proceedings of the USENIX Workshop on UNIX and Supercomputers* (Sept. 1988). USENIX.
8. Fox, G. C., Johnson, M. A., Lyzenga, G. A., Otto, S. W., Salomon, J. K., and Walker, D. W. *Solving Problems on Concurrent Processors.* Vol. 1. Prentice-Hall, Englewood Cliffs, N.J., 1988.
9. Graunke, G., and Thakkar, S. Synchronization algorithms for shared-memory multiprocessors. *IEEE Comput. 23*, 6 (June 1990), 60–70.
10. Gupta, A., Tucker, A., and Stevens, L. Making effective use of shared-memory multiprocessors: The process control approach. Tech. Rep. CSL-TR-91-475A, 1991.
11. Gupta, A., Tucker, A., and Urushibara, S. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of the 1991 ACM SIGMETRICS Conference* (May 1991). ACM, New York.
12. Karlin, A., Li, K., Manasse, M. S., and Owicki, S. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (Oct. 1991). ACM, New York.
13. Leland, W., and Ott, T. Load-balancing heuristics and process behavior. In *Proceedings of Performance '86 and 1986 ACM SIGMETRICS Conference* (May 1986). ACM, New York.
14. Leutenegger, S. T., and Vernon, M. K. The performance of multiprogrammed multiprocessor scheduling policies. In *Proceedings of the 1990 ACM SIGMETRICS Conference* (May 1990). ACM, New York.

15. Lo, S.-P., and Gligor, V. D. A comparative analysis of multiprocessor scheduling algorithms. In *Proceedings of the 7th International Conference on Distributed Computing Systems* (Sept. 1987).
16. Lovett, T., and Thakkar, S. The symmetry multiprocessor system. In *Proceedings of the 1988 International Conference on Parallel Processing* (Aug. 1988).
17. Majumdar, S., Eager, D. L., and Bunt, R. L. Scheduling in multiprogrammed parallel systems. In *Proceedings of the 1988 ACM SIGMETRICS Conference* (May 1988). ACM, New York.
18. Ousterhout, J. K. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems* (Oct. 1982).
19. Polychronopoulos, C. D. Multiprocessing versus multiprogramming. In *Proceedings of the 1989 International Conference on Parallel Processing* (Aug. 1989).
20. Reiser, M., and Lavenberg, S. S. Mean value analysis of closed multichain queueing networks. *J. ACM 27*, 2 (Apr. 1980), 313–322.
21. Sevcik, K C. Characterizations of parallelism in applications and their use in scheduling. In *Proceedings of the 1989 ACM SIGMETRICS and Performance '89 International Conference* (May 1989). ACM, New York.
22. Tucker, A., and Gupta, A. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors In *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (Dec. 1989). ACM, New York.
23. Thacker, C. P., Stewart, L. C., and Satterthwaite, E. H., Jr. Firefly: A multiprocessor workstation. *IEEE Trans Comput. 37*, 8 (Aug. 1988), 909–920.
24. Vaswani, R., and Zahorjan, J. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (Oct 1991). ACM, New York.
25. Zahorjan, J., and McCann, C. Processor scheduling in shared memory multiprocessors. In *Proceedings of the 1990 ACM SIGMETRICS Conference* (May 1990). ACM, New York.
26. Zahorjan, J., Lazowska, E. D., and Eager, D. L. The effect of scheduling discipline on spin overhead in shared memory parallel systems. *IEEE Trans. Parall. Distrib. Syst. 2*, 2 (Apr. 1991), 180–198.
27. Zahorjan, J., Lazowska, E. D., and Eager, D. L. Spinning versus blocking in parallel systems with uncertainty. In *Proceedings of the International Symposium on Performance of Distributed and Parallel Systems* (Kyoto, Japan, Dec. 1988).

Received March 1991; revised August and December 1992; accepted January 1993