SCHEDULING ALGORITHMS AND THEIR PERFORMANCE ON SHARED MEMORY MULTIPROCESSORS

K.S. DiBella, K. Ramamritham,
P.K. Chrysanthis and S. Raghuram

COINS Technical Report 92-37
September 1992

Scheduling Algorithms and their Performance on Shared Memory Multiprocessors

Kathleen S. DiBella[1]
Krithi Ramamritham[1]
Panos Chrysanthis[2]
S. Raghuram[1]

Abstract

Many multithreaded multiprocessing systems have simply adopted the scheduling structure used in single processing systems: the single central queue of ready processes maintained for the system creates a source of contention. Hence researchers have been trying to eliminate or redefine redundant and inefficient data structures and deal with other aspects that contribute to excessive waits.

This paper examines an alternative scheduling structure which reduces the negative effects of the ready queue bottleneck. The proposed structure is composed of multiple ready queues which make it possible for several processes to access the queue(s) simultaneously. Each processor removes a ready thread (request) from a selected ready queue for execution. The *scheduling policy* determines which ready queue is selected. The application processes post ready to run threads in the ready queues according to a *queuing policy*. Different combinations of the scheduling and queuing policies are possible but the policies have to be compatible. This relative independence of the two types of policies has allowed us to implement and test many policy combinations. We also compare their performance with existing scheduling implementations. Results of tests based on our implementation on a Sequent Symmetry multiprocessor show that multi-access distributed ready queues offer potential performance benefits for multiprocessor systems and that the overheads of managing multiple queues is more than overcome by their improved performance.

[1]Computer Science Dept. University of Massachusetts, Amherst, MA 01003
[2]Computer Science Dept. University of Pittsburgh, Pittsburgh, PA 15260

1 Introduction

Current multiprocessor operating systems such as MACH [4] and Dynix [19] implement schedulers with FIFO ready queues which have the ability to distinguish between processes of differing priorities (*e.g.,* multi-level feedback queues). Access to the queue is controlled by a single mutual exclusion lock. This creates a central source of contention and a system bottleneck for processes. Every processor and every process must access the ready queue, therefore a slowdown at the queue will degrade application performance.

In order to reduce the bottleneck, the following changes to the scheduler can be made.

- **Distributed Queue and Scheduler.** In lieu of a single central ready queue, each processor is assigned its own ready queue and scheduler. The scheduler is responsible for all the scheduling activity for one particular processor and executes whenever the processor is idle.
- **Multi-Access Queues.** Current scheduling implementations allow only a single process to access a ready queue at any point in time. The queue is locked for the entire time necessary to place a ready process on the queue or remove a process from the queue. While the lock is being held, there may be other ready processes and idle processors wasting valuable cpu time waiting for that lock. A queue implemented with multiple locks would allow multiple processors to access the queue simultaneously, thus reducing the wasted cpu cycles. The cost for multiple access is the maintainence of the additional locks for the queue.

In this paper, we propose a scheduling structure in which both mechanisms have been implemented. More precisely, we modify the ready queue to be made up of multiple independently accessed queues, each controlled by a different set of locks. This allows multiple processes (or threads) concurrent access to the system queue structures. Typically, there are as many queues as there are processors in the system. This configuration attempts to minimize interprocess conflicts which occur with a single global ready queue and scheduler. Tests show that this scheduling structure, described in more detail in the next section, reduces the ready queue bottleneck for shared memory multiprocessor systems.

Considering a system which consists of multiple processes each consisting of a number of threads, two approaches are possible: The system level scheduler schedules the processes and only when a process runs do the threads within it get a chance to run. The other is for the system scheduler to handle all the thread scheduling. The latter allows multiple threads of one process to run concurrently and is the preferred approach due to its improved performance. Hence we take the latter approach in this paper.

Given such a multiple queue structure, two questions arise:

1. Into which queue does a client process's thread get enqueued?

2. From which queue does a processors' scheduler dequeue a ready thread?

Whereas most of the prior work (discussed in Section 6) has focussed on individual policies and their performance, this paper's contribution lies in the following:

- We have identified a set of *client policies* (also refered to in this paper as *enqueuing policies*) that client processes use to place their requests in queues. (This is in response to the first question raised above.)
- We have identified a set of *scheduler policies* (also refered to in this paper as *dequeuing policies*) that the schedulers use to remove ready requests from the queues. (This is in response to the second question raised above.)
- We have implemented both types of policies in a real multiprocessor system, namely Sequent's Symmetry.
- We have conducted extensive tests to investigate the performance characteristics of the individual policies as well as of the policy combinations.
- We have also compared the results of our policies with two baseline scheduling implementations: a single global queue and the FCFS distributed ready queues [1].

The next section describes the scheduler structure and defines the policies governing the enqueueing and dequeueing policies. The following sections discuss implementation details, performance results and related research papers.

2 System Structure

The system uses a single data structure, called the *ready queue* to support both communication between the scheduler and client processes, and to schedule client threads.

The Dynix operating system, that we used to conduct our experiments, does not distinguish threads from processes in terms of scheduling, and since schedulers and clients are not created dynamically, the tests were done using processes. For the purposes of the tests, the client processes post requests in lieu of ready threads. The scheduler services a request and returns a reply to the client. That is, the system is structured along the client/server model in which a process, called the *client*, makes a system call by placing a request in the ready queue. The scheduler then handles the request and sends back a reply.

2.1 The Queue

The *ready queue* data structure is a set of logically distributed queues[2]; one per available processor (see Figure 1). Each queue consists of an array of *mailboxes* and a bitmap. A mailbox is divided into an IN-box and an OUT-box. Client processes place a request into

[2]The queue referred to in this paper is not a strict FIFO queue.

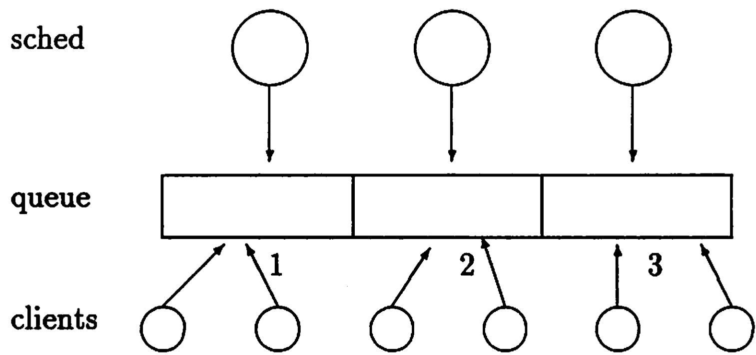

Figure 1: Static : Static queue and process structure.

an IN-box. Once a client has successfully posted a request in an IN-box, the OUT-box is implicitly reserved for the reply to that request. The bitmap provides a 1 : 1 mapping of bits to mailboxes in the queue. If a bit is set it implies that the corresponding mailbox in the queue contains a request, i.e., a ready thread.

A specific mailbox location is identified using two indices. The queue index, which is a number from $0..MaxProc-1$ where $MaxProc$ is the maximum number of processors in the system. This queue index is unique for every queue and is used as its id. The second index indicates exactly which mailbox in the queue is being addressed. The mailbox index is an integer from $0..MaxReqs-1$ where $MaxReqs$ is the maximum number of client requests allowed per queue. In our implementation, using an unsigned integer as the bitmap yields $MaxReqs = 32$.

In order to enhance concurrency, a two-level locking scheme in conjunction with optimistic synchronization is used to control access to a queue. At the queue level, a lock is associated with the bitmap. That is, there is one bitmap lock per processor queue. At the mailbox level each IN-box and OUT-box has its own lock. Mailbox locks are used so that processes accessing the queue always have a valid state description of the mailbox information before altering any of the information (*e.g.*, posting or removing a client thread (request)).

When a client posts a request, the bit in the bitmap corresponding to the mailbox used is set. A scheduler then need only search the bitmap to check if a request is ready, instead of searching the queue directly. When the scheduler finds a bit set in the bitmap it is searching, it acquires the bitmap lock and, if the bit is still set, resets the bit thus reserving the mailbox with the ready request. The bitmap lock is relinquished. The request can then be serviced and a reply returned to the client. If once the bitmap lock is acquired the bit is no longer set, the lock is immediately relinquished and the scheduler continues to search for ready requests.

The bitmap locks could be a source of contention when there are many clients accessing a

single queue. This is not seen to be a problem in our tests as a process holds the lock only to set a single bit. The remainder of the request information is processed while a process has sole access to an IN-box or OUT-box lock, not a *queue* lock.

2.2 The Scheduler Structure and Policies

Every processor has its own scheduler which searches one or more ready queues for requests. The scheduler polls the queues to determine if there are new requests. When a request is found, the scheduler assigns the request to its processor and handles the request. It then posts a reply in the OUT-box for the client who originated the request, and then searches for new requests.

When a processor's queue is empty, it can either wait for new requests to arrive, or it may look in other processors' queues for requests. The method a scheduler uses when searching for requests is governed by the following scheduling policies (for a summary, see Table 1).

- **Static.** This policy assigns to a processor a single queue to service. Should the queue be empty, no attempt is made by the scheduler to look elsewhere for requests. There is a $1:1$ mapping of queues to processors, so every processor has a unique ready queue.
- **Random.** A scheduler randomly chooses which of the ready queues it will poll next. There is no order to the queue accesses.
- **Sequential.** A scheduler begins looking for requests in its *primary* queue; the primary queue has the same id as the scheduler process. When this queue becomes empty, the scheduler increments its queue index by $1(\mathrm{mod} MaxProc)$ and searches the next queue for requests (*i.e.*, this cycle continues until the queue index $= MaxProc - 1$. At this time, the scheduler resets its queue index to zero and continues searching for requests.).

  A scheduler will only look for requests elsewhere while its primary queue is empty. If a new request arrives to a scheduler's primary queue while a request from another queue is being serviced, the scheduler process will service its primary queue immediately after the current request is serviced.
- **Primary/Secondary.** For the Primary/Secondary policies, each processor is assigned a static primary queue as in the Sequential policy above. When this primary queue is empty, the scheduler searches for requests in other queues based on one of the following policies:

  1. Fixed. The processor is also assigned a secondary queue index. This is the only queue which the scheduler accesses when its primary queue is empty. This is referred to as the Primary/Fixed scheduler policy.
  2. Random. The scheduler randomly chooses one of the remaining queues to access when its primary queue is empty. This is referred to as the Primary/Random scheduler policy.

3. Sequential Search. This scheduler policy is referred to as Primary/Sequential. In this policy, a scheduler starts searching for requests in the queue with an index one greater than its primary queue. If this queue is empty, the queue index is incremented by one and the next queue is searched. The search for requests stops when a non-empty ready queue is encountered. This non-empty queue will be the *only* secondary queue serviced by the scheduler before returning to its primary queue. A scheduler using the Sequential policy described above, however, will continue to service *all* non-primary queues while its primary queue is empty.

An additional parameter for these policies governs the direction in which a scheduler process searches a chosen queue. If no direction is specified, all queues are searched from left to right. However, such a search policy may create unnecessary contention for mailboxes when one or more processes are accessing a queue. For example, when several scheduler processes are searching a queue's bitmap in the same direction, they may all find the same bit set. All the processes would then request the bitmap lock in order to reset the bit, but only one scheduler would be successful in scheduling the indicated request. The remaining scheduler processes must wait for their turn in the lock in order to determine the new queue state before continuing to search for ready requests (this is sometimes referred to as the convoy phenomenon).

Therefore, a *bidirectional* option was implemented where scheduler processes search their primary queues from left to right and search their secondary queues in the reverse direction; from right to left. Using bidirectional search, two scheduler processes are less likely to interfere with each other's acquisition of new requests. In the situation previously described, as long as there is more than one request in the queue, the primary scheduler and secondary scheduler will usually find different bits set and subsequently both will be successful in their attempts to schedule the requests.

Because access to each queue is not strictly FIFO in the above policies, lack of starvation is ensured by having each scheduler *remember* which mailbox it last serviced in its primary and secondary queues[3]. When a scheduler searches the bitmap for new requests, it increments its mailbox index by one and sequentially searches the queue. When the scheduler *remembers* where it left off searching a queue, this becomes a strictly increasing function (to a max of $MaxReqs - 1$) and all mailboxes in a queue will eventually be serviced. The manner in which the secondary queues are used merely supplements this primary queue service.

The following two policies were also implemented to serve as baselines to compare our policies with.

- **Global.** There is a single global FCFS queue which is maintained for the system. Each scheduler and client process must acquire the lock to the queue before it can post a request to or remove a request from the queue. This models a single queue typically used in shared memory multiprocessors.

[3] A scheduler does not remember an index for each queue in the system, therefore, some policies will provide better service than others.

Scheduler Policy	Policy Description
Static	1 : 1 static mapping of queues
Random	random 1 : 1 queue service
Sequential	sequential search of queues
Primary/Secondary	*bidirectional search policies*
- Random	primary static, random secondary
- Fixed	primary static, fixed secondary
- Seq. Search	primary static, sequential secondary
Global	single global queue - FCFS
Distributed FCFS	1 : 1 static mapping - FCFS guaranteed

Table 1: Summary of the Scheduler Policies.

- **Distributed FCFS.** The distributed FCFS policy is based upon the work done by Anderson et al [1]. There is a ready queue per processor in the system. The difference between this policy and the static policy is that this policy restricts access to one process posting or removing a request from a single queue at any time. Requests to a particular queue will be serviced in the order in which they were posted. The policy does allow a scheduler to search other processors' queues in the event that its own is empty. This secondary search is done in a sequential manner.

2.3 Client Processes and Policies

Client processes enqueue requests in the queues for the schedulers to service. The clients have the ability to post requests both synchronously and asychronously.

In the synchronous case, a client posts a request and only when a reply to the previous request is received can the client post another request to the queue.

We tested two asynchronous request arrivals.

1. Uniform Asynchronous. In this case, *all* client processes wait for the same average length of time between requests. It is NOT necessary for a client to wait for a reply to its previous request(s) before posting another.
2. Non-Uniform Asynchronous. In the non-uniform case, the client processes are divided into $MaxProc$ sets. Each set contains approximately the same number of client processes[4]. One set of clients generates requests at the maximum possible rate, while the other ($MaxProc-1$) sets wait a longer length of time between arrivals. The average wait between requests is the same for the ($MaxProc - 1$) sets of client processes.

[4]Some queues may contain *one* more client than another. For example, if there are 12 clients and 3 processors, there will be 4 clients/queue. If there are 13 clients and 3 schedulers, then there is one queue with 5 client processes and the other 2 queues will service 4 clients.

Which queue a client will use to post a request is governed by the following queueing policies. (for summary, see Table 2).

- **Static.** A client is allowed to assign requests to *only* one queue during its lifetime. There is not necessarily a 1 : 1 mapping of clients to queues, however, the policy attempts to distribute the clients equitably amongst the queues.

- **Random.** When a client is ready to post a request it randomly selects a queue.

- **Sequential.** Each client is assigned a starting queue based upon its id, to reduce startup contention for resources, and posts its first request in this queue. For each subsequent request posted by the client, the queue index is incremented by $1 \pmod{MaxProc}$ and the appropriate queue is used.

- **Primary/Secondary.** There are two primary/secondary policies governing the posting of requests. For each, the primary queue is assigned using the static policy. When this primary queue already contains a predetermined number of client requests, the client posts its next request in a secondary mailbox. The predetermined number of requests, referred to as the *cutoff capacity*, assumes a queue's length is an indication of the delay associated with that queue. When a primary queue is filled to the cutoff capacity, it is assumed that better service may be provided by posting the new request elsewhere. In these tests the cutoff capacity was set to be the number of client processes in the system.

  The secondary policy is determined in one of the following two ways;

  1. Fixed. A secondary queue index is assigned to the client for the duration of the test.

  2. Random. The client process randomly chooses a secondary queue whenever the primary queue is overloaded.

  In both cases, the secondary queue is distinct from the primary queue.

- **Global.** There is a single global queue and it is accessed by a single process in a strictly FCFS manner.

- **Static FCFS.** This is one of the sister policies of the Distributed FCFS scheduler policy. Each client is assigned a queue index for the duration of the test and mutually exclusive access to the queue is enforced. As in the static policy, the policy distributes the clients amongst the queues as equitably as possible.

- **Random FCFS.** Another sister policy of the Distributed FCFS scheduler policy, permits a client process to randomly select a queue in which to post its next request. Mutually exclusive access to the queues is enforced.

Client Policy	Policy Description
Static	static queue assignment
Random	random use of queues
Sequential	sequential use of queues
Primary/Random	primary/ random secondary
Primary/Fixed	primary/ fixed secondary
Global	global queue - FCFS
Static FCFS	multiple queues - FCFS
Random FCFS	multiple queues - FCFS random

Table 2: Summary of the Client Policies.

3 Implementation Details

We tested a system with a set of scheduler processes and a disjoint set of client processes. This tested scheduling structure was implemented on top of the Dynix operating system. Therefore, it was necessary to minimize the interaction with Dynix. There are also a number of parameters which have an impact on the test performance results. The following subsections describe the choices made in the implementation and the effects these choices have on the performance results.

3.1 Signalling by Clients vs. Polling by Schedulers

One interaction between our code and Dynix arises in the manner in which the scheduler processes are made aware of new requests. There are two ways in which a scheduler process can be made aware of new ready requests in a queue. One method is by the use of Dynix signals. When a client posts a request, it then signals one or more scheduler processes that there is a new request ready for service. Which scheduler processes should be signalled by a client is dependent upon the scheduling policy in use during a particular test. For any of the static scheduler policies, because the scheduler processes service the queue with the same index as their process id only the index of the queue need be known to determine which scheduler to signal. However, once the scheduler policies assign secondary queues, it is not clear which schedulers need to be signalled. If both the primary and secondary schedulers for a queue are signalled, then the secondary scheduler may be unnecessarily interrupted, resulting in poorer overall system performance. If only the primary scheduler is signalled, the new request may remain undetected in the queue for a relatively long time, again resulting in poorer overall service. If a random policy is employed by the scheduler processes, it is impossible to determine which scheduler is servicing a queue at any time and issues similar to those of the primary/secondary service arise.

Another drawback of signalling is the length of time it takes for the signal to propogate to its destination. Using Dynix, the average delay due to signalling is approximately 0.6 - 0.7 ms, however, the service time for a NULL request is only 0.11 ms. Clearly, the time required to deliver a signal will dominate the per request delay measured by the NULL test. In addition

there are certain policies where explicit signals are not required. For all policies, a scheduler will continue to service requests as long as there are requests in the queue. A single global queue has a greater probability of continuously having ready requests than a distributed queue structure. Therefore, when using a global queue it is not necessary that a scheduler be signalled. With a distributed queue, a scheduler will more often need a signal to *restart* itself. Thus, between the global and distributed queue results a maximum discrepancy of 0.6-0.7 ms/req in delay may be solely attributable to a Dynix specific implementation.

There is also the risk of "losing" a signal with the Dynix operating system. That is, even if the client religiously signals the correct scheduler when a new request is enqueued, there is no guarantee that the signal will be received. To guarantee that all signals are received requires additional fault tolerance overhead; for example, duplication of signals after a timeout period and/or timer interrupts. The problem with using a timing interrupt as a backup mechanism is that the minimum granularity allowed by the Dynix operating system is 10 ms. This time is much greater than the typical service time for a request and is not useful for these tests.

Therefore, a different method was used: A scheduler process polls the queues. This is a much more robust implementation and was the method chosen for the tests.

Signals are, however, used when schedulers reply to the client requests. Returning replies to the clients involves the same amount of work for *all* scheduler and client policies. Therefore, the effects of using signals for replies is the same for all tests and the results remain comparable. In the synchronous tests, the time between requests will be the time to service the request plus the signal propogation time and the time to service the interrupt created by the signal. For the asynchronous tests, the reply signals do not affect the request generation rate directly.

3.2 Limiting the Effects of the Dynix Scheduler

A second source of interaction between the test code and the Dynix operating system is due to the Dynix scheduler and the context switch time involved in moving a process from one processor to another after a Dynix time quantum has expired. In order to minimize the test code's interaction with the Dynix scheduler, each scheduler process was given *affinity* for a particular processor. This insures that the scheduler process is not affected by the Dynix scheduler. The client processes were given affinity for the processors *not* used by scheduler processes. That is, the client process occupied a subset of processors disjoint from that used by the scheduler processes.

Assigning the client processes to a disjoint subset of processors also guarantees that the client and scheduler processes do not interfere with one another's progress during the tests. To guarantee that client processes do not interfere with one another's creation of requests, the total number of client and scheduler processes involved in a test was limited to the total number of available processors in the system. For our Sequent multiprocessor this number is 16.

Given these assignment constraints, a scheduler process now represents a single processor and a client process represents an application creating threads (i.e., requests) which must be scheduled on these processors. Tests can then be run to determine the relative performance of any combination of the policies described in the previous sections.

4 Performance Parameters

There are several factors which must be considered when running these tests.

4.1 Service Time

The amount of CPU time required by a request is a test parameter. The longer the average CPU requirements of the client requests, the longer the average queue length will be. In turn, this will affect the average delay per request. The delay refers to the total time a request is in the queue, or, more specifically, is the time from request generation until the end of servicing by a scheduler. A request's service time is determined by the client process generating the request. Client processes post an *operation* which specifies the requested CPU time as part of the request. The NULL operation simply requires the scheduler to remove a request from the IN-box, do the required bookkeeping[5] and return a reply in the OUT-box. Other operations increased the operation time in intervals of approximately 0.05 ms. The bookkeeping time for the NULL operation was approximately 0.11 ms. All requests for a particular test had the same average request time.

4.2 Interarrival Time

Synchronous tests are used to simulate a very *lightly loaded* system. At any point in time, there are at most the same number of requests in the queues as there are client processes. The time between request generation for a specific client process is based entirely on the delay and service time of the previous request. This includes the return signal's propogation time and the time necessary to service the interrupt generated by the signal.

Asynchronous request generation has the potential to impose higher loads. For a given set of test parameters, all policy combinations have the potential to generate an equal number of requests in the same amount of time. However, there are other factors which affect the generation rate of requests by the client processes.

The primary factor is the time spent waiting for a queue or mailbox lock. If this time is longer than a request's CPU time, a signal may be received by the client while it is posting a request. After posting a request, the client process will then service the interrupt before posting another request. This results in a slower request generation rate by client processes

[5]Bookkeeping refers to calculating the delay of a request in the queue, counting the number of requests serviced, etc.

for certain policies. The shorter the lock duration, the greater the number of requests which can be input into the queues by client processes between interrupts.

The *measured* arrival rate of requests to the system is referred to in the results as *throughput.* Some confusion may result from trying to interpret the results using a queueing theoretical definition of throughput. The difference between the two definitions, is that the throughput represented in the results is not *entirely* dependent upon the delay in the system. The delay plays a large factor, but the locking delay, which is generally much less than the end-to-end delay, affects the rate at which a client is able to generate requests. The performance of a policy pair must be represented not only by the average delay per request, but also by the effect it has on the system's ability to originate requests.

4.3 Load Distribution

Load balancing effects of different policies must also be considered. A viable scheduler/client policy pair must have provisions for distributing the work equally amongst the processors. During a test, each request has the same average service time, therefore, the total number of requests serviced by a scheduler (processor) can be tallied to determine the effective load balancing effects of a policy pair.

We created two situations to model imbalance in work offered to different processors:

1. Provide a number of clients which is *not* a multiple of the number of schedulers. This causes more clients to be mapped to some schedulers than others.
2. Use the non-uniform asynchronous request generation rates. In addition to being unbalanced, the sets of clients are also offering requests to the system at different rates.

The load distribution of a test case is determined by measuring the percentage of the requests serviced by each scheduler process (*e.g.* processor).

5 Performance Results

This section defines the queueing policies tested, the test parameters used and then discusses the results. For a set of tests, a fixed number of client processes and another number of scheduler processes is chosen. Both synchronous and asynchronous requests are tested. The service time of the requests begins as NULL and is then incremented for each successive test. Not all policy pairs are tested for the entire set of tests. As a queueing policy is eliminated, the reasons for its elimination from future testing are discussed.

Several combinations of scheduler and client policies are possible as outlined in Table 3. The possible pairs can be determined by the table delineations. For example, Global : Global

Scheduler Policy	Client Policy
Static, Random, Sequential Primary/Secondary	Static, Random, Sequential Primary/Random, Primary/Fixed
Global	Global
Distributed FCFS	Static FCFS, Random FCFS

Table 3: Scheduling Algorithms

is the only possible global policy pair, whereas the Static scheduler policy can be combined with the Static, Random, Sequential or either of the Primary/Secondary client policies.

All scheduler primary/secondary policies use the bidirectional search option. Preliminary tests were run comparing results from the policies with and without the option. In all cases, using the bidirectional search option for the secondary queues resulted in delays and throughputs which were less than or equal to the results obtained without using the option.

5.1 First Round of Eliminations

The policy pairs from the first block of Table 3 were run initially with non-uniform asynchronous request generation, 3 scheduler processes, 10 client processes, and the NULL operation. Each test was run for the duration of 20,000 requests and was repeated 15 times. The average results of these runs are repored in the tables and graphs presented in this paper. Longer request durations were tested (*e.g.*, 100,000 requests) with very little difference in the performance results and standard deviations. Also, several tests were run with varying repetitions; 20, 30 and 50, also with little difference in the measured performance metrics. Therefore, 15 repetitions of 20,000 requests was determined to be an accurate representation of an algorithm's performance.

The results from these test runs were used to determine the poorer performers and eliminate them from future tests. The criteria for discontinuing a policy were load distribution, delay and throughput, defined in the previous section. The policies in the bottom block of Table 3 are discussed later in this section.

As the operation (service) time increases, the average length of the queue and the delay per request also increase. For large service times, the probability that a scheduler will access its secondary queue approaches zero. In addition, the amount of delay incurred waiting for a lock will become negligible when compared with the per request delay. Therefore, the potential benefits of the policies will be most evident from the tests using the NULL operation.

The results of the NULL tests are shown in Table 4, and the policies surviving the first cut are shown in Table 5. The *delay* in Table 4 is the average per request delay measured in ms.

Scheduler Policy	Client Policy	Delay (ms/req)	Thruput (req/ms)	Load Distrib (% off balanced)		
Static	Static	8.1710	5.6585	6.7	-3.3	-3.3
	Random	4.9520	5.9854	0.3	0	-0.3
	Sequential	4.2764	6.5503	0.1	-0.2	0.1
	Primary/Rand	3.2533	6.2391	3.1	-0.8	-2.3
	Primary/Fix	3.6144	6.1895	2.6	0.7	-3.3
Random	Static	6.7913	6.1674	-0.1	-0.6	0.7
	Random	5.1990	5.9835	0.2	-0.5	0.3
	Sequential	5.0565	6.4927	0.1	-0.3	0.2
	Primary/Rand	3.5762	6.2862	-0.1	-0.5	0.6
	Primary/Fix	4.2916	6.2908	0	-0.5	0.5
Primary/ Random	Static	6.9040	5.8623	2.1	-1.2	-0.9
	Random	3.8734	6.1394	0.1	-0.3	0.2
	Sequential	4.3092	6.5787	0.2	-0.3	0.1
	Primary/Rand	3.1489	6.2452	2	-1	-1
	Primary/Fix	3.2645	6.2989	2.2	-0.7	-1.5
Primary/ Fixed	Static	6.0331	6.0389	1.4	-1.5	0.2
	Random	3.4852	6.1544	0.1	-0.5	0.4
	Sequential	4.1814	6.5559	0.1	-0.4	0.3
	Primary/Rand	2.8564	6.2897	1.1	-1.1	0
	Primary/Fix	2.6960	6.2780	1.1	-0.6	-0.5
Sequential	Static	5.9908	6.0120	0.8	-0.5	-0.3
	Random	3.3805	6.1688	0.1	-0.5	0.4
	Sequential	4.2864	6.5777	0.1	-0.4	0.3
	Primary/Rand	2.8865	6.3131	1.1	-0.8	-0.3
	Primary/Fix	3.0036	6.3616	1.2	-0.5	-0.7
Primary/ Sequential	Static	6.0252	6.0319	0.8	-0.6	-0.2
	Random	3.5523	6.1241	0	-0.4	0.4
	Sequential	4.2806	6.5742	0.1	-0.4	0.3
	Primary/Rand	2.8435	6.2853	1.2	-1	-0.2
	Primary/Fix	2.8429	6.3162	1.1	-0.5	-0.6

Table 4: Results for Non-Uniform Asynchronous NULL requests (3 schedulers & 10 clients).

This is a system measurement, not a per processor measure. The throughput (*thruput* is the rate at which requests were generated by the test system when using the policy combination indicated. This is measured in requests/ms. The load distribution is measured during the tests and the percentage of the requests serviced by each scheduler process is indicated. The worst case is the Static : Static policy pair where one scheduler services 40% of the requests and the remaining two only service 30%of the requests each. We would like the load to be distributed equally across the processors (*e.g.* 33.3% for all processors). The ideal load distribution is indicated by 0% in the Load Distribution column of Table 4.

The Static client policy is eliminated due to its inability to equitably distribute the load amongst the processors when combined with most scheduling policies. When combined with the Sequential and Primary/Sequential scheduler policies, the load distribution is more

Scheduler Policy	Client Policies
Primary/Fixed, Sequential Primary/Sequential	Primary/Random Random, Sequential, Primary/Fixed

Table 5: Policies performing best with non-uniform asynchronous NULL requests.

equitable, but the delays are much longer than when using other client policies. All the scheduler policies except Random, assign a primary queue to the scheduler. Other queues are accessed only in the event that the primary queue becomes empty. In this test, only one queue is overloaded with requests. When the other queues become empty, this single queue with requests acts as a central global queue, and causes the same bottlenecks which our approach, in general, attempts to avoid. Therefore, it is necessary for the client processes to make an attempt to distribute the load to the queues.

The Static scheduler policy is also eliminated as it does not distribute the load equitably amongst the processors except when combined with the Random client policy. When combined with the Random client policy, the delays are longer than for other policies offering equivalent throughput. Relying solely on the client policy for load distribution, the Static scheduler performs poorly in the non-uniform loading environment.

Among the remaining policies, the Random and Primary/Random scheduler policies have a lot of variance in their performance. The time at which a queue is serviced varies, and therefore, these policies do not offer consistent performance, nor do they result in a competitive delay.

Increasing the service time for each test, several tests were run on the policy sets in Table 4 to determine which yield the best performance in a variety of loading situations. The throughput, delay and load balancing data from these tests were compared to determine the potential benefits of the algorithms with respect to each other and the base case global and distributed FCFS policies. These are discussed next.

5.2 Load Balancing

Some of the scheduler and client policies can be eliminated due to their load distribution characteristics. Partial results for these policies are represented in Table 6. Using the Random client policy, the Primary/Fixed and Sequential scheduler policy results are compared. The load distribution capabilities of both scheduler policies are similar, however, the standard deviation is greater for the Primary/Fixed policy indicating it is less reliable than the Sequential scheduler policy. When combined with the Primary/Random client policy the Primary/Fixed scheduler policy performed less effectively than the Sequential scheduler policy in terms of load balancing, yet delay and throughput results were similar. The Primary/Fixed scheduler has no distinctive characteristics that warrant it being included in

Client Policy	Scheduler Policy	Svc (ms)	Thruput (req/ms)	Delay (ms/req)	Load Distrib (% off balance)		
Random	Primary/Fix	0.319	4.587	16.433	0	-0.3	0.3
	Sequential	0.319	4.588	16.639	-0.1	-0.2	0.3
	Primary/Fix	0.420	4.032	21.313	0.2	-0.3	0.1
	Sequential	0.420	4.039	21.601	0.2	-0.3	0.1
Primary/ Random	Primary/Fix	0.319	4.999	12.769	0.3	-0.9	0.6
	Sequential	0.319	4.998	12.842	-0.1	-0.2	0.3
	Primary/Fix	0.420	4.570	16.434	0.8	-1.5	0.7
	Sequential	0.420	4.575	16.437	0.2	-0.2	0

Table 6: Load Distribution for Scheduler policies vs. Service time (3 schedulers & 10 clients).

future tests and therefore, the policy was eliminated from further consideration.

The Fixed FCFS client policy also did not provide load balancing to the system for the same reasons the static client policy did not. Therefore, only the Random FCFS client policy was considered further.

5.3 Throughput and Delay

In presenting the remaining results, since the Sequential and Primary/Sequential scheduler policies perform nearly equally, only the Sequential policy results are shown to make the graphs clearer. Their similar performance is attributed to the fact that, in our tests, there are *always* requests being generated, therefore, the time spent servicing a secondary queue while a scheduler's primary queue is empty is relatively short. If a primary queue were empty for a longer duration, it is anticipated that the Sequential policy would outperform the Primary/Sequential policy. In such a situation, a scheduler using the Sequential policy will continue to search for work in *all* the other queues in the system, whereas, a scheduler using the Primary/Sequential policy will search for the first queue with requests it finds and *only* service this queue, thus resulting in a possible load imbalance, and longer delays if the secondary queue selected is not the overloaded one.

The next performance metric is the throughput, *i.e*, the request generation rate, sustained by the system when using different policy pairs. The throughput of the remaining policies and the Distributed FCFS policy are competitive (see Figure 2 - a : b denotes scheduler policy *a* combined with client policy *b*). The global policy performs *much* worse than the distributed algorithms and is eliminated from further consideration as a useful scheduling structure. This is in line with the results in [1]; the distribution of the ready queue offered improved system performance, measured in terms of delay and throughput, over a single global ready queue.

The Primary/Fixed client policy is also eliminated based upon its throughput. The throughput is approximately 8% less than the throughput of the Random FCFS policy and the

Primary/Random or Random client policies.

Based on the delay results (See figure 3), the Sequential client policy can be eliminated. For short service times the delay incurred is approximately twice the delay measured for other policies in the worst case. When the average delay per request for each of the remaining algorithms is examined, there are policy pairs for which the multiple access queues described in this paper offer better performance than the single access distributed FCFS queues. The scheduler policies remaining offer this improved performance; the Sequential and the Primary/Sequential algorithms.

Using the Sequential scheduler policy, the client policies which perform the best are the Random and the Primary/Random policies. Of the two client policies, the Primary/Random performs slightly better than the Random client policy. The Primary/Random policy will *only* redistribute work when its primary queue has reached a certain predetermined capacity; for these tests, the capacity was set as the number of client processes in a test. This feature avoids randomly determining a new queue for *every* request a client makes. Generating a random number takes a certain amount of time. If the time necessary to generate a random number were significantly reduced, there should be no difference between the policies' performances. The randomness of the two policies allows the request load to be equally distributed amongst the processor queues, while reducing the probability that several clients are accessing the same queue at the same time.

Let us compare the top performance pair, Sequential : Primary/Random policies with the Distributed FCFS : Random FCFS base case policies. Though the throughput is roughly equivalent, there is a distinct difference in the per request delay for each algorithm. These results are illustrated in Figure 3. For NULL request (*i.e.* 0.11 ms service time) the Sequential : Primary/Random policy pair outperforms the Distributed FCFS queue structure by 50%. As the service time and hence, the queue length increases, the difference in per request delay diminishes to 8% but Sequential : Primary/Random remains the better performer.

Synchronous tests using 3 schedulers and 10 client processes were also run to determine whether the benefits of the multi-access queues is evident at very light loads. For very lightly loaded systems, the Primary/Random client and the Sequential scheduler policies in these tests offer no improvement over a Distributed FCFS scheduler tested with a Random FCFS client policy (See figure 4). This point is more clearly illustrated by the tests run using 2 schedulers and 11 clients (See figure 5) where the Sequential : Primary/Random policy has a slightly longer delay than the Distributed FCFS policy pair. However, as the the ratio of clients to schedulers increases, from 10 : 3 to 11 : 2, the performance of the Sequential : Primary/Random policy becomes similar to that of the Distributed FCFS queue.

Additional asynchronous tests were run using 4 scheduler processes and 9 client processes(See figures 6 and 7). The results from these tests more clearly illustrate the performance of the policies under a variety of loading conditions. When the system is lightly loaded (*e.g.* the queue lengths are relatively small), the Distributed FCFS policy pair performs better than

the distributed multi-access queues. However, as the average queue lengths increase, the multi-access policies perform better than the Distributed single access queue scheduling structure. Therefore, we project that the improvements already noted by distributing the scheduler and the ready queues [1] can again be improved by allowing multiple concurrent access to each queue in conjunction with other simple scheduling policies.

These results suggest that the Sequential : Primary/Random policy pair offers the best performance for asynchronous loads. For synchronous loads, this policy pair may offer improved performance over the Distributed FCFS : Random FCFS policy pair as the client to scheduler ratios is increased.

6 Related Work

Closely related to the work done in this paper is the work by Anderson et al and described in their paper [1]. They ran a series of experiments to determine how the reorganization of the system data structures would affect application performance in terms of latency and throughput. The tested configurations included:

- **A central (global) locked ready queue.** The global base case policies described in this paper model this particular implementation.
- **Per processor free lists with a single locked ready queue.** Each processor maintains a local list of deallocated control blocks which it uses when scheduling a new thread to run. This local "free list" requires no lock as each control block can be accessed by exactly one processor. This structure reduces the amount of time a processor must hold the ready queue lock if the processor has a free control block. In the event a processor has an empty free list, there is a global pool of control blocks which is guarded by a lock and is accessable to all processors. This protocol does not eliminate the central ready queue, it only intends to reduce the length of time a processor holds the lock when scheduling a thread.
- **Per processor free lists with a central queue of idle processors.** This structure is implemented to allow parallel creation of threads when a processor is idle and there is no outstanding work to be done. In such an instance, the idle processor creates a control block and places itself in the idle queue. The next task (thread) to enter the system checks the idle queue for available processors and schedules itself on the first available processor. In other words, the threads look for processors instead of processors searching for threads.
- **Per processor ready queues.** Each processor maintains a local ready queue of runnable processes (threads). Each ready queue is protected by a single mutex lock. Runnable processes are assigned to the queue in a randomly. In the event a processor's local ready queue is empty, it will search other queues for work. A processor searches the queues sequentially to find additional ready threads.

The application processes post their new threads in the queues, randomly selecting a queue to use for their next ready thread. The Distributed FCFS/Random Distributed FCFS base case policies of this paper were used to model this situation.

The experiments indicate that the *per processor ready queues*, yield the greatest system throughput and a reasonable latency due to the reduction in contention for scheduling locks. The only contention for locks occurs when a processor has no runnable threads in its local queue. In this case, the processor may access other ready queues. This results of this work make it clear that a single global ready queue is less efficient than multiple, per processor ready queues for scheduling.

In their paper, Anderson et al suggested using hyrids of their proposed data structures. The work presented in our paper, extends their ideas to allow multiple access to the per processor ready queues. We also extend the policies governing the posting and servicing of threads to/from the queues. In addition to the Random/Sequential policy pair described above, the policies outlined in Tables 1 and 2 offer a variety of scheduling implementations to be tested and compared with the original work.

Another related work is by Squillante and Nelson [14]. They created a queueing theoretic model of task migration for a shared memory multiprocessor. They examine the benefits and/or detractions of allowing processors to access ready queues other than their own. Their work indicates that there are upper and lower bounds for system loads beyond which migrating processes from one cpu to another has no noticeable benefits. They assume per processor ready queues as the data structure to be used by the system.

The performance of the locking mechanism, critical to these policies may benefit from the implementation discussed by Mellor-Crummey and Scott[11]. They suggest that altering the lock implementation may be the best way to reduce the traffic on the bus due to contention. They suggest an algorithm for mutual exclusion which virtually eliminates bus contention. This algorithm guarantees FCFS lock service, and shows improved application performance. This lock implementation would show improvement in a system where there is great contention for a single lock (*i.e.* the Distributed queue structure used by Anderson et al [1]). Our scheduler structure, however, is designed to minimize contention for ready queue locks. Therefore, though the locking algorithm proposed by Mellor-Crummey and Scott can be applied to our proposed scheduling structure, we do not expect this to improve the performance of the algorithms significantly.

7 Conclusions

A problem with many multithreaded multiprocessor operating systems is that they have been derived directly from uniprocessor operating systems with little modification. This paper examines the system resource contention caused by having a single central scheduling structure. To this end, the queue was distributed amongst the processors in the system and each processor given a scheduling thread. In addition, the ready queue was modified to have

multiple locks, allowing multiple processes (or threads) concurrent access to the structure. Using this configuration reduces interprocess conflicts which occur with a global ready queue and scheduler.

Various policies were used to govern the application and scheduler processes access to the queues. The policy combinations were tested, and the results compared to base case scheduling implementations: the single locked global queue and the FCFS distributed ready queues.

Our test results support Anderson et al's findings[1] that a distributed ready queue performs better in terms of delay and throughput than a single global ready queue. The results also support our expectation that concurrent access to a queue by several processes would improve the performance of the scheduler despite the additional locks required. At very low loads, our scheduler implementation offers little, if any, improvement over that examined by Anderson et al [1]. As the loads and hence the queue lengths increase, the Sequential : Primary/Random policies provide improved performance over the Distributed FCFS : Random FCFS policies. Our scheduling structure, however, is not a strict FIFO queue implementation. It is not necessarily true that the next request enqueued in a queue will be the next request removed from that queue. However, posting and removing requests to and from the queues is done in such a manner that starvation is avoided and has caused no increase in end to end service time for a request. Therefore, using our scheduling structure several *threads* may be scheduled from a queue concurrently. With the strictly FIFO distributed queues proposed in [1], access to a queue is confined to a single process at a time. If there is only one queue with ready threads and there are several idle processors, the distributed FIFO scheduling mechanism reduces to a single global queue.

It would be of interest to implement different types of schedulers using the multiple access ready queues described in this paper. For example, none of the algorithms tested for this paper emulate the MACH scheduler which is a two-level scheduling structure. There is a single global queue from which all the processors can schedule ready threads. In addition, each processor has a local queue of threads it can run. A processor will only access the global queue in the event that there are no runnable threads in its local queue (assuming all threads are of equal priority). Implementing our scheduling structure would require modifying the MACH scheduling structures to allow multiple processors concurrent access to all the queues, both global and local.

Other remaining work is to implement these algorithms and data structures so that they actually do the scheduling for a shared memory multiprocessor. It will be useful to test the approaches with respect to benchmark multiprocessor applications which attempt to utilize the parallelism of the shared memory multiprocessor. We expect the results will be similar if the schedulers are user-level processes scheduling application threads to the processors. If the schedulers are implemented as kernel threads, the kernel overhead may change the results [2].

In order to measure the load distribution characteristics of the scheduling policies, the service time required by each request was nearly equal. System loads are rarely uniform in

their service requirements. Implementing a working scheduler and measuring the user's response time may provide new performance results. We expect, however, that the relative performance of one policy pair to another will remain the same.

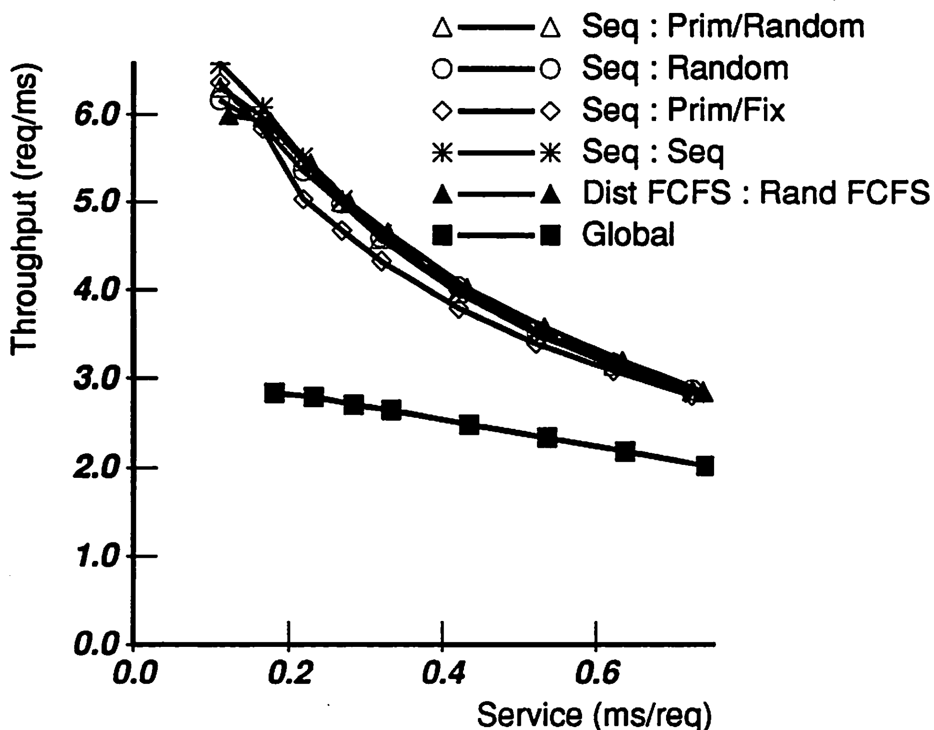

Figure 2: Non-Uniform asynchronous loading: Service Time vs. Throughput (3 schedu lers and 10 clients).

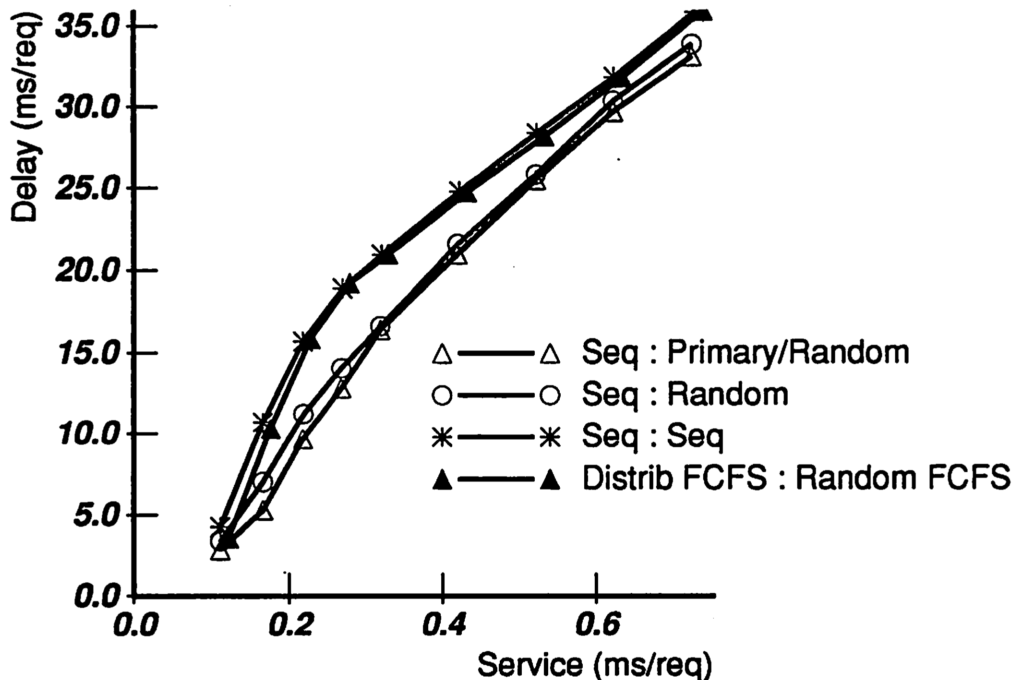

Figure 3: Non-Uniform asynchronous loading: Service Time vs. Delay (3 schedulers and 10 clients).

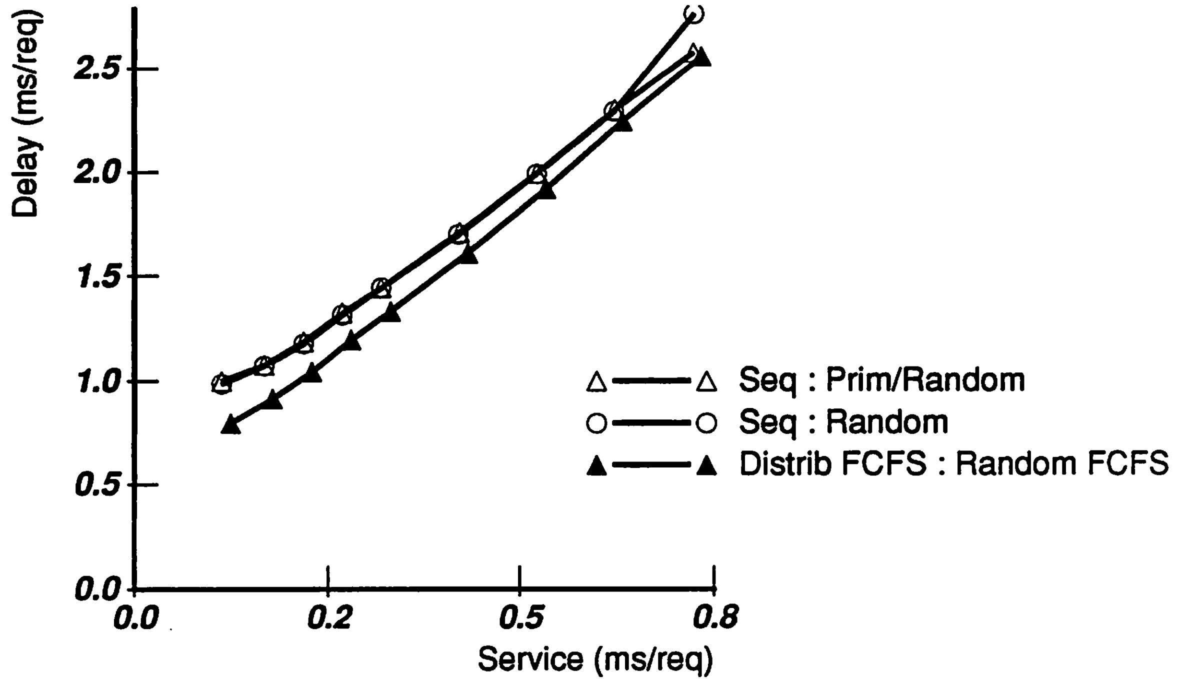

Figure 4: Synchronous loading: Service Time vs. Delay (3 schedulers and 10 client s).

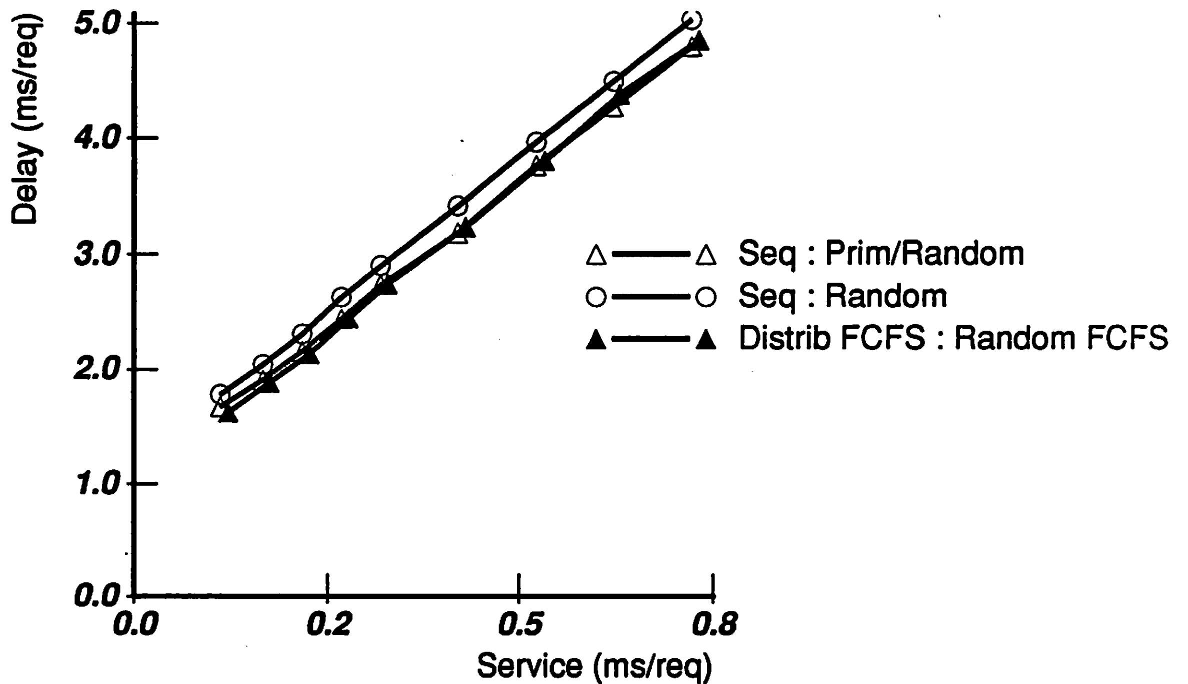

Figure 5: Synchronous loading: Service Time vs. Delay (2 schedulers and 11 clien ts).

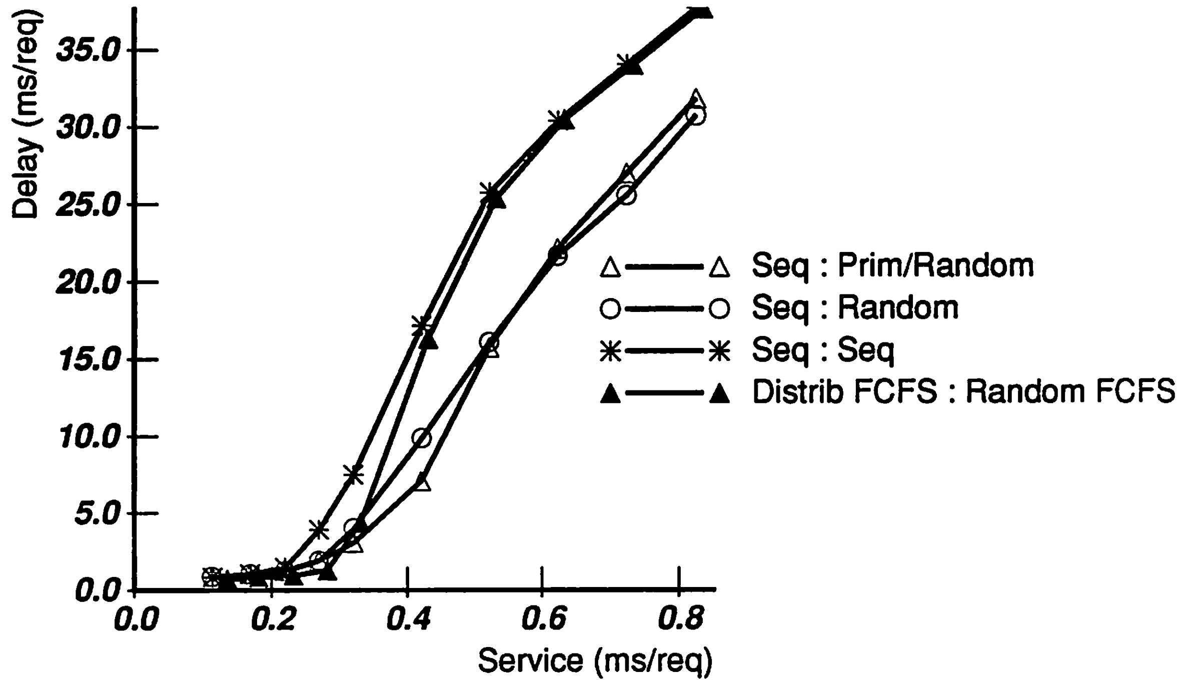

Figure 6: Asynchronous loading: Service Time vs. Delay (4 schedulers and 9 client s) and detail.

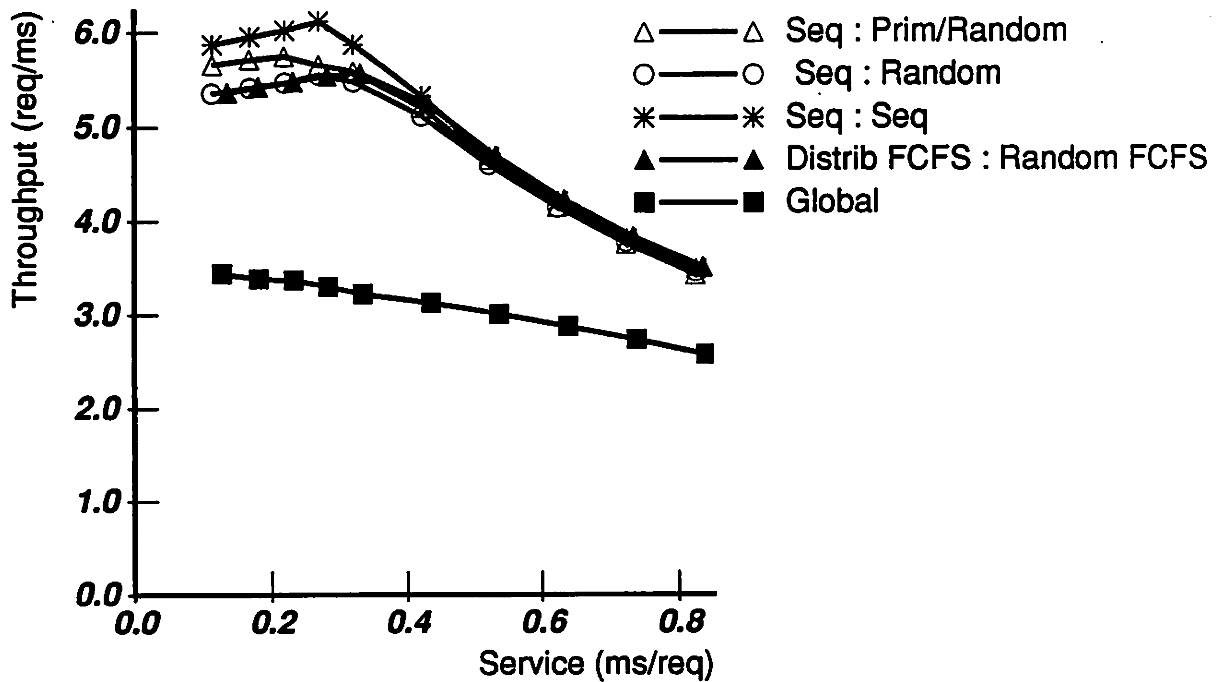

Figure 7: Asynchronous loading: Service Time vs. Throughput (4 schedulers and 9 c lients).

References

[1] Anderson, TE, Edward Lazowska and Henry Levy, The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors , *IEEE Transactions on Computers*, December 1989.

[2] Anderson, TE, Henry Levy, Brian Bershad and Edward LAZOWSKA, The Interaction of Architecture and Operating System Design, ASPLOS 1991.

[3] Bershad, Brian, Edward Lazowska and Henry Levy, PRESTO: A System for Object-Oriented Parallel Programming , *Software: Practice and Experience*, February 1988.

[4] Black, David L., Scheduling Support for Concurrency and Parallelism in the Mach OS , *IEEE Computer*, May 1990.

[5] Cheriton, David R., The V Distributed System , *Communications of the ACM*, March 1988.

[6] DiBella, Kathleen S., Krithi Ramamritham, Panos Chrysanthis, S. Raghuram, Scheduling Algorithms and their Performance on Shared Memory Multiprocessors, University of Massachusetts Technical Report, August 1992.

[7] Duncan, Ralph, A Survey of Parallel Computer Architectures , *IEEE Computer*, February 1990.

[8] Eager, Derek L, John Zohorjan, and Edward Lazowska, Speedup versus Efficiency in Parallel Systems , *IEEE Transactions on Computers*, March 1989.

[9] Gupta, Anoop, Andrew Tucker, and Shigeru Urushibara, The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications , *Proc. of the ACM SIGMETRICS Intl. Conference on Measurement and Modeling of Computer Systems*, 1991.

[10] Leutenegger, Scott, and Mary K Vernon, The Performance of Multiprogrammed Multiprocessor Scheduling Policies , *Proc. of the 1990 ACM SIGMETRICS Conference*, May 1990.

[11] Mellor-Crummey, John M. and Michael L. Scott, Synchronization Without Contention , ASPLOS 1991.

[12] Mogul, Jeffrey C. and Anita Borg, The Effect of Context Switches on Cache Performance ASPLOS 1991 .

[13] Ni, Lionel M. and Ching-Farn E. Wu, Design Tradeoffs for Process Scheduling in Shared Memory Multiprocessor Systems , *IEEE Transactions on Software Engineering*, March 1989 .

[14] Squillante, Mark S and Randolph D Nelson, Analysis of Task Migration in Shared-Memory Multiprocessor Scheduling ,*Proc. ACM SIGMETRICS 1991*, May 1991.

[15] Tucker, Andrew and Anoop Gupta, Process Control and Scheduling Issues for Multiprogrammed Shared Memory Multiprocessors , *Communications of the ACM*, March 1989.

[16] Zahorjan, John, Edward Lazowska, and Derek Eager, The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems , *IEEE Transactions on Parallel and Distributed Systems*, April 1991.

[17] Zahorjan, John and Cathy McCann, Processor Scheduling in Shared Memory Multiprocessors , *Proc. 1990 ACM SIGMETRICS*, May 1990.

[18] Zhou Songnian and Timothy Brecht, Processor Pool-Based Scheduling for Large-Scale NUMA Multiprocessors , *Proc 1991 ACM SIGMETRICS Conf. on Measurement and Modeling of Compter Systems*, May 1991.

[19] Guide to Parallel Programming on Sequent Computer Systems , Sequent Computer Systems, Inc., Prentice-Hall 1989.