Computer Systems, 3(4), Oct., 1990, pp. 551--579.

A Comparison of basic CPU Scheduling Algorithms for Multiprocessor Unix †

Stephen Curran
Michael Stumm

Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada M5S 1A4

ABSTRACT

In this paper, we present the results of a simulation study comparing three basic algorithms that schedule independent tasks in multiprocessor versions of Unix. Two of these algorithms, namely Central Queue and Initial Placement, are obvious extensions to the standard uniprocessor scheduling algorithm and are in use in a number of multiprocessor systems. A third algorithm, Take, is a variation on Initial Placement, where processors are allowed to raid the task queues of the other processors. Our simulation results show the difference between the performance of the three algorithms to be small when scheduling a typical Unix workload running on a small, bus-based, shared memory multiprocessor. They also show that the Take algorithm performs best for those multiprocessors on which tasks incur overhead each time they migrate. In particular, the Take algorithm appears to be more stable than the other two algorithms under extreme conditions.

1. Introduction

In this paper, we consider ways to organize and manage the ready tasks in a shared memory multiprocessor. The uniprocessor Unix kernel uses a single priority queue for this purpose: A task is added to the ready queue behind all tasks of higher or equal priority, and the CPU is allocated to the task with the highest priority which is at the head of the queue. For the shared memory multiprocessor case, we consider three basic scheduling algorithms in a simulation study, and compare their behavior and performance in scheduling a Unix workload of tasks. We focus on bus-based multiprocessors that incur extra overhead each time a task is migrated from one processor to another. Many existing multiprocessors belong to this class, including the Encore Multimax [MNO86], the Sequent Symmetry [LT88], and the DEC Firefly [TSS88]. Two of the scheduling algorithms we study are obvious extensions of the method used in the uniprocessor case:

- **Central Queue** (CQ): All processors of the multiprocessor share a single ready queue. (This requires that accesses to the queue be synchronized.)
- **Initial Placement** (IP): Each processor has its own separate ready queue and only executes tasks from its own queue. When a task is first created, it is permanently assigned to one of the processors.

The problem with Initial Placement is that a *load imbalance* may occur, where some processors are idle with nothing to execute, while other processors are busy and have tasks in their ready queues. Any load imbalance will result in poorer task response times. With Central Queue, the load is always perfectly balanced, since an idle processor will execute any available task. However, the probability is high that a task

† Appeared in *Computer Systems*, 3(4), Oct., 1990, pp. 551--579.

will execute on a different processor each time it is dispatched; that is, it will be migrated. For many multiprocessors, each task migration incurs overhead both for the task being migrated and the rest of the system, increasing average task response times if the Central Queue algorithm is used.

These two algorithms represent opposite ends of a spectrum, where Initial Placement has no migration overhead and unbalanced loads, and Central Queue has migration overhead but a perfectly balanced load. A third algorithm, which is a variation of Initial Placement, lies between these two extremes:

- **Take** (TK): Each processor has its own separate ready queue, and tasks are initially assigned to the queue of one of the processors. Each processor executes tasks from its own queue whenever possible, but raids the queues of the other processors when its own queue is empty.

Of these algorithms, Central Queue will perform best if there is no migration overhead, but we show that the Take algorithm performs better than both the Initial Placement and Central Queue algorithms under most operating conditions on those systems that have migration overhead. Although the difference between the three algorithms is relatively small, the behavior of the Take algorithm is much more stable than the other two algorithms under extreme conditions. Initial placement performs worse than the other two algorithms because of load imbalances. Central Queue performs poorly when the load is high, due to the overhead of the large number of migrations that occur. In addition, Central Queue gives poor response to low priority tasks under high loads. The Take algorithm performs well, because it resorts to task migration (with its associated overhead) only when load imbalances occur and does not migrate tasks when there is little benefit in doing so.

This paper is theoretical in the sense that the results are obtained from simulation studies instead of from measurements of real systems. Nevertheless, we believe that the results we present are of practical value. All three algorithms we study are practical in that they are simple and have been implemented; two of them are used in most of the existing multiprocessors today. Moreover, we use measurements obtained from a real Unix system to derive the simulator's input parameters. Finally, although we don't derive absolute performance predictions for specific systems and workloads, we study the relative performance of the three algorithms across a broad range of architectures and workloads and analyze how changes in the system or workload affect performance. This allows us to study the behavior of the algorithms under different circumstances.

The following section describes the different sources of migration overhead in order to assess the cost they inflict. In Section 3, we describe in detail the simulation model, the input parameters, the scheduling algorithms, and the performance metric used. Section 4 presents the results of simulating a baseline system, and Section 5 extends these results by considering other workloads and system parameters.

Related Work

A number of papers exist that describe how scheduling is performed in existing multiprocessor operating system implementations [Bla90, Kel89, LT88, TSS88, RW87]. All of these systems implement variants of the Central Queue algorithm.

Two additional studies are concerned with the overhead of the scheduling algorithms themselves. Wendorf [Wen87] concludes that scheduling decisions should only be made on a (small) subset of the processors in a multiprocessor. Ni and Wu [NW89] analytically studied several scheduling algorithms to show the effects of contention to the ready queue on the performance of the algorithms. In this paper, we assume that the overhead for the scheduling algorithms is negligible and that there is no contention for the ready queue(s). These are reasonable assumptions, given the simplicity of the algorithms and the small number of processors being considered here.

Much of the early multiprocessor scheduling research focused on scheduling parallel programs on systems dedicated to the particular application; see Gonzalez [Gon77] for a survey. It is still an open question how to schedule parallel programs in general multiprogramming environments. This is an area of active research [ZM90, LV90, Bla90, TG89, MEB88, Ous82]. In this paper, we do not consider parallel programs, but only consider the scheduling of *independent* tasks.

Considerable effort has also been spent studying network-wide scheduling issues [Stu88, Zho88, LM82]. For example, Leland and Ott [LO86] show that for the Unix workload they investigated, the use of migration can improve the performance of large tasks by up to 25%. In contrast, Eager et.al. [ELZ88] use an analytical model to show that there is no major performance gain in using migration if a good initial placement load balancing algorithm is already being used. These results are not directly applicable to shared memory multiprocessors, however. In a multiprocessor, state information can be kept in shared memory and is quickly accessible by all processors. Also, the overhead of task migration is on the order of milliseconds on a multiprocessor but on the order of seconds in a distributed system.

2. Task Migration Costs

Migration costs can come from three sources in a modern shared memory multiprocessor:

1. the loss of cache context when a task moves from one processor to another,
2. the overhead required to keep data consistent across multiple caches, and
3. the processing overhead that affects the performance of the system as a whole.

In this section, we consider each of these cost sources in more detail.

Cache Context Loss

Many of today's multiprocessors have a per-processor memory cache both to reduce bus traffic and to improve the average memory access time. A task, executing on a processor with a cache, can accumulate a considerable amount of context close to the processor in the form of cache entries. When a task restarts after a pause, some of its cached data may (depending on the type of cache) still be present at the cache of the processor it last ran on, and therefore, will execute faster if it is run on that processor. If it is migrated to a new processor, it will have little or no cache context and must build up its context through cache misses. This causes a decrease in the execution speed of the task itself and causes an increase in bus traffic which slows down the system as a whole.

How much cache context is lost when a task migrates depends on many factors, including the architecture of the cache subsystem, its structure and size, the degree of associativity, the characteristics of the transients (the building and decaying of the context), the number of other tasks executing on the processor, the length of time since the task last ran, and the time the task executed on the processor from which it is migrating. The amount of cache context lost is limited by the task's working set. It should also be noted that for some systems there may be no extra overhead for migrating a task. This is the case for all systems, for example, where the cache must be invalidated on each context switch.

Cache Consistency Overhead

After a task migrates from one processor to another, its data may be present in two caches at the same time. If this is the case then data that is local to a task will appear to be shared to the system, requiring a mechanism to ensure that the caches stay consistent. Cache consistency comes with a cost, whether it is performed in hardware or software. In systems without hardware cache consistency support, the operating system can invalidate the cache when a task is migrated. The cache of either the destination processor or the processor on which the task last executed can be invalidated, depending on how the caches operate. If a cache write-back protocol is used, the cache of the old processor must be flushed and invalidated to ensure that all of the tasks' data modifications are reflected in main memory before it is loaded into the new cache. If, on the other hand, a cache write-through protocol is used, invalidating either processor's cache is acceptable. A cache flush can take hundreds of cycles in real systems.

Even if caches are kept consistent by hardware, cache consistency overhead is still possible. On some systems, the consistency mechanism may cause additional bus traffic. This is the case, for example, on the DEC Firefly multiprocessor [TSS88]. The Firefly uses an update scheme, where each modification to data present in more than one cache is broadcast (across the bus) to all caches so that they can update their cache entries. (In contrast, when a data item is present in only one cache, a write-back scheme is used with no extra bus traffic.) On other systems, the consistency mechanism may slow down cache access on other processors. For example, in some multiprocessors, the processor may be prevented from accessing the cache while a snoopy invalidation or update is occurring.

System Migration Overhead

System Migration Overhead refers to the overhead of migration that affects the performance of the entire system. As described above, System Migration Overhead generally occurs in two forms. First, migration may increase contention for the system bus, because of increased bus traffic due to more frequent cache misses when a task starts executing on a new processor. This causes an increase in the average memory access time for all tasks, reducing the execution speeds of all tasks in execution. Secondly, additional bus traffic may be necessary to keep the caches consistent. For example, we have observed the bus on the Firefly multiprocessor to saturate at 100% utilization when running four small independent programs on the Firefly; the bus was being used entirely for cache updates that would not have occurred if the tasks had not migrated.

In addition to these two forms of System Migration Overhead, a more subtle, third form can occur on some architectures, where certain operations can be optimized if it is known that task migrations will not take place. For example, consider a system where all memory is globally accessible over the bus, but is partitioned across the processor boards such that local memory can be accessed faster than memory located on other processor boards. If migration is not permitted on such a system, then the operating system will always try to allocate faster, local memory to a task so that the vast majority of accesses will be handled locally. On the other hand, if migration is permitted on this type of system then the number of accesses to local memory decreases since it is no longer possible to keep a migrating task's memory local without copying it. The number of remote memory accesses therefore increases, as does the bus traffic and the average memory access time, with the attendant system-wide processor performance degradation.

3. The Simulator

Our event-driven simulator models a workload of Unix-like tasks executing on a small, bus-based, shared-memory multiprocessor. The system model consists of four components: the hardware, the workload, the scheduling algorithms and the task priority models. Figure 3.1 depicts the queueing network model implemented by the simulator. The figure shows both the hardware of the system (processors and the I/O subsystem) and the path tasks take as they pass through the system.

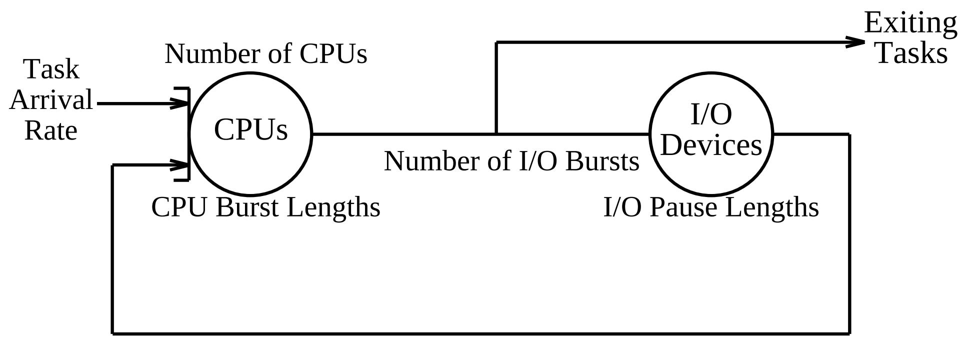

Figure 3.1: The queueing network model implemented in the simulator.
The labels on the figure show some of the key input parameters to the simulator.

This simulator is initially used to simulate a *Baseline System*, defined by two sets of input parameters. The first set describes the hardware model and includes, for example, the cost of task migration and the speed of the processors. The other set defines the workload that is executed on the system. The parameters used in the Baseline System tests define a particular system running a particular workload and the simulator makes numerous simplifying assumptions, so it is natural to question the parameters and the results of the simulation. For this reason, we analyzed the sensitivity of the results of our Baseline System tests by independently varying the input parameters to determine if and how each affects the performance of the scheduling algorithms. Most often, we found that changes in the input parameters do not change the results in a significant way. For example, when simulating slower processors, the behavior of the algorithms remains similar to that of the Baseline tests, although the relative differences become more pronounced. Similarly, changes in the number of processors, the size of the CPU quantum, or the length of the I/O pauses did not have much effect on our results. (See [Cur89] for details.) We therefore believe that the results of the Baseline System tests are valid across a large range of architectures and workloads, including workloads typically found in software development, text processing and computer aided design. Those cases where changes in input parameters produce interesting results are described in Section 5.

In the rest of this section, we describe in detail the assumptions made in the simulation model, the input parameters, the scheduling algorithms, and the performance metrics used in this study. The casual reader may want to skip to Section 4.

3.1. Assumptions

In our hardware model, all processors are homogeneous, and each can schedule and execute tasks. Tasks can execute on any processor and can move from one processor to another whenever they are dispatched (subject to a migration cost penalty). For simplicity, the I/O subsystem is assumed to be merely a delay; there is no queueing for I/O.†

The simulator workload is represented by a set of tasks executing an alternating series of CPU bursts and I/O pauses. New tasks arrive in the system at a given arrival rate. A task's characteristics, including its priority, the number of CPU bursts it will execute, and the length of each CPU burst and I/O pause, are generated based on a set of configurable workload model parameters. A task executing on a processor proceeds until its current CPU burst has completed or until one CPU quantum of 100 ms. completes. The distribution of CPU bursts and I/O pauses are taken from measurements of existing Unix systems (as described in Section 3.2).

For the initial simulations, a simplified priority model is used, where all tasks are of the same priority. Later, in Section 5, we consider a more realistic task priority model and determine how the scheduling algorithms are affected by the use of tasks running at multiple priority levels.

Migration costs are controlled by two parameters, the Task Migration Costs (TMC) and the System Migration Costs (SMC). The TMC parameter represents the migration cost to the task itself, and is therefore a penalty applied to the task being migrated. The penalty is a fixed amount of time added to the first CPU burst following a migration. The System Migration Cost parameter controls the magnitude of the system-wide processor performance penalty that results from the use and support of migration. The SMC represents a reduction in the speed of all of the processors in the system, and is implemented by increasing the length of all CPU bursts by a fixed percentage when a migration-based algorithm is used. This distribution of migration costs models the bulk of the migration costs from the sources described in Section 2.1.*

3.2. Input Parameters

A number of input parameters are used to define the *Baseline System* and come from several sources. First, the hardware characteristics and migration costs are based on a multiprocessor we our currently building and is representative of many existing systems. In this system, we assume there are four 20 MIPS processors. Initially, we assume a TMC of 1 ms, which is added to the length of a task's next CPU burst each time it is migrated, and we assume a SMC of 5%, which is added to all CPU bursts when a migration-based scheduling algorithm is used. Later, we vary these parameters to study their effects.

The workload parameters are based on measurements we performed on a (single processor, 3 MIPS) CVAX-based Unix system to obtain CPU burst and I/O pause distributions, using a clock with a microsecond resolution. As an example of the data obtained from these measurements, Figure 2.1 depicts the CPU burst length distribution for all processes that executed on the system for a period of 90 minutes, during which one user was active developing software (mainly editing) using the X window system, while

† The queueing delays are modeled in the length of the I/O pauses.

* Some of the System Migration Overhead costs (such as contention for the system bus) are dependent on the rate at which tasks migrate. In our model, the SMC parameter is a fixed value that is applied if the scheduling algorithm supports migration, regardless of whether migration is actually occurring or not.

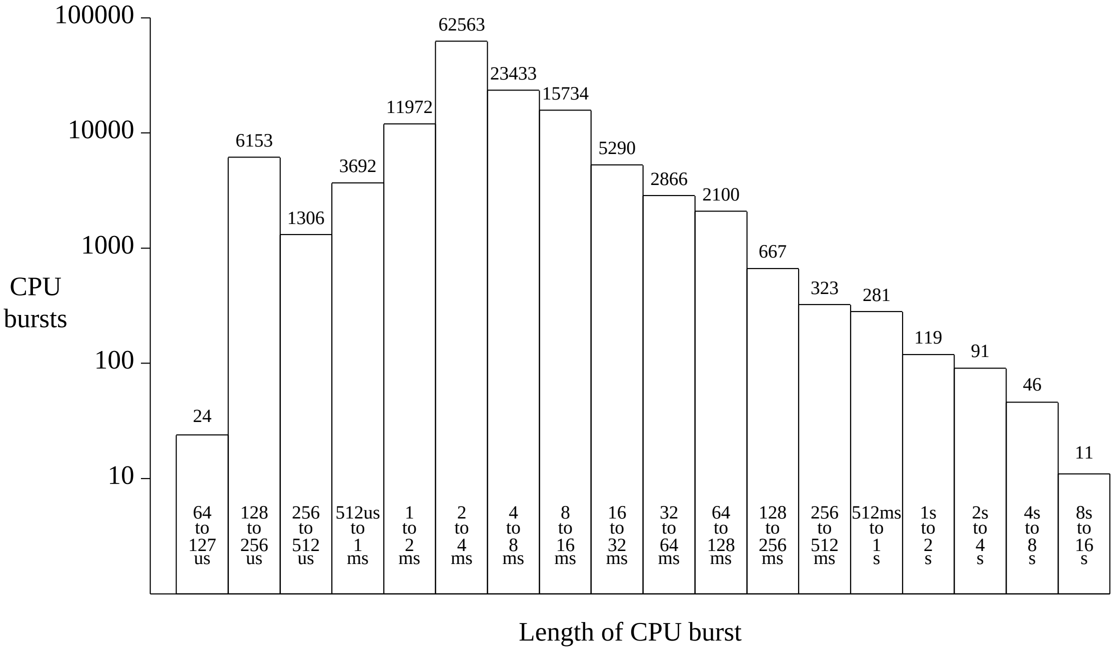

Figure 2.1: Distribution of CPU burst lengths

a BSD 4.2 Unix kernel was remade in the background.† (Note that both axes in the figure have logarithmic scales.) One can observe that most of the CPU bursts are relatively short, i.e. between 2 and 4 milliseconds. A few CPU bursts are longer (in the 8-16 second range); they occur at the beginning of the kernel make and when vmunix is linked.

During other periods, we ran more computationally intensive applications, including several instances of formatting this paper (with grap, pic, eqn, tbl and troff), a PGA routing application, and Spice, a circuit simulator (the last two belonging to CAD packages). The distribution of CPU bursts for these applications was found to be very similar. For example, troff had a slightly higher proportion of longer bursts, and Spice had an additional single large (60 min.) CPU burst. The PGA router did not have a higher proportion of long CPU bursts (as we initially expected it would) because of high paging activing due to the high memory demands of that application.

We also ran our simulator with workload numbers obtained in earlier studies by Mullender [Mul85] (for CPU burst lengths) and Zhou [Zho88] (for I/O pause lengths), with results that supported the ones we obtained in our baseline tests.*

3.3. Simulator Performance Metrics

The selection of a meaningful performance metric for comparing the simulation runs is a difficult issue. Two metrics commonly used in similar studies are *processor utilization* and *response time* [ELZ88, LO86]. We decided not to base performance quality on processor utilization, because utilization may not accurately reflect the performance of migration-based scheduling algorithms; processor utilization may increase due to the extra processing time inherent in migration and not because of an increase in the amount of useful work being achieved. The task response ratio appears to be a more appropriate metric for our purposes. We therefore present our result in terms of the Global Response Ratio (GRR) [LO86]. GRR is calculated as the ratio between the tasks' actual response times and the time they would need to execute

† More precisely, a CPU burst of a task is measured as the total time it is executing on a processor (in either user or kernel mode) from the time it is added to the ready queue after an I/O pause, until it executes *sleep* in the kernel to begin another I/O pause.

∗ Since these studies are based on older, VAX-based systems, we also adapted the workload parameters to match more modern, faster technology by scaling processor speed appropriately.

if there were no overhead:

$$GRR \;=\; \frac{\sum \; Elapsed\ Time}{\sum \; Task\ Time}$$

The *Task Time* is the sum of the length of all I/O pauses and CPU bursts of a task (before migration costs and queueing time are added). The *Elapsed Time* is the Task Time plus overhead, consisting of migration costs and time spent queueing for access to the CPU. The summations are over N tasks, the number of tasks in the simulation run.

GRR effectively weighs the performance of each task by the size of the task and hence is less sensitive to the performance of individual small tasks. This is appropriate, because of the large number of small tasks relative to large tasks in most workloads, and because the performance of larger tasks is more noticeable to the user than that of smaller tasks; i.e., the difference between having the response time go from 0.5 seconds to 1 second and having the response time go from 1 hour to 2 hours. Moreover, to ensure that improvements in the performance of larger tasks do not hurt the smaller tasks, a second response ratio measure (also from [LO86]), the Average Per-Task Response Ratio (APR†), where all tasks are weighted equally, was also calculated for all simulations.

3.4. Scheduling Algorithm Details

We consider three CPU scheduling algorithms: the Initial Placement, the Central Queue, and the Take algorithms. We describe the three algorithms and some of the details of our implementations. Later, we consider several variations of our implementations.

Initial Placement

Under the Initial Placement (IP) algorithm, tasks arriving in the system are placed on the least-loaded processor in the system (according to some implementation dependent metric), where they remain until they complete. Each processor maintains its own run queue in priority order, with tasks of the same priority scheduled in a round-robin fashion. In our implementation, the least-loaded processor is the one with the fewest tasks that are ready to run or blocked on I/O.

Central Queue

Under Central Queue (CQ), all tasks are scheduled in strict priority order using a single, system-wide task queue. Tasks of equal priority are scheduled in round-robin order. A task migration occurs each time an idle processor retrieves a task from the queue that last executed on a different processor.

In our implementation of the CQ algorithm, an optimization is performed to reduce the number of migrations in a lightly loaded system. If the processor that last executed a task when the task becomes ready is idle, then the processor is selected to continue executing the task. (This optimization is implemented in the Topaz operating system [TSS88].)

Take

Under the Take (TK) algorithm, tasks arriving in the system are initially assigned to the least-loaded processor (according to some implementation dependent metric). As with IP, each processor maintains its own run queue in priority order, with tasks of the same priority scheduled in round-robin fashion. If a processor becomes idle (its run queue is empty), it checks the run queues of all other processors and migrates a task from the heaviest-loaded processor (according to some implementation dependent metric) to its own run queue and executes it.

The least-loaded and heaviest-loaded processor in our TK implementation are the processors with the fewest and most ready tasks, respectively. In a practical implementation, a counter per processor keeps a

---

† APR is calculated as follows:

$$\frac{1}{N}\sum\frac{Elapsedtime}{TaskTime}$$

count of the number of tasks in the queue, eliminating the need to lock the queues while their lengths are compared.

4. The Baseline Tests

In this section we analyze the performance and behavior of the Baseline System. The first issue we consider is how load affects the performance of the algorithms. Figure 4.1 (left) depicts the results of the Baseline System tests for all three scheduling algorithms, as the Offered Load is varied from 0.3 to 0.95. The Offered Load parameter controls the task load on the system. It can be thought of as the expected utilization of the processors, and is set by appropriately setting the arrival rate of tasks into the system.*

Three significant results arise from these tests. First, the difference between the three algorithms is small under most load conditions, always less than 10%. Second, for the most part, the migration based algorithms outperform the non-migration based IP algorithm. Third, the performance of the CQ algorithm is poor at very high loads. We discuss the latter two results in the following sections.

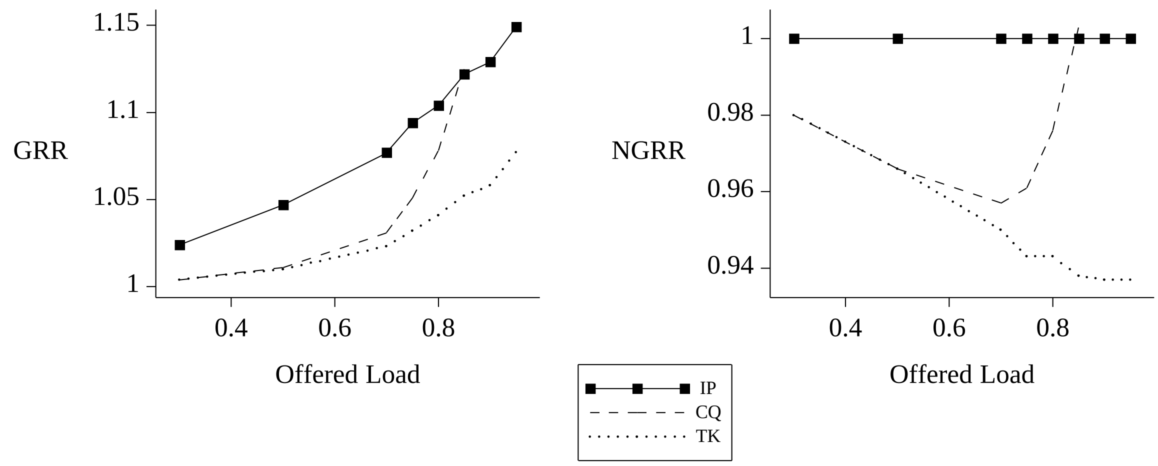

Figure 4.1: GRR and normalized GRR of Baseline system versus Offered Load

4.1. Comparing IP and the Migration-Based Algorithms

To clarify the magnitude of the performance difference between the algorithms, the second graph of Figure 4.1 shows the GRR results for the same tests *normalized* by the GRR results for the IP algorithm (NGRR). From the graphs, we see that there is little benefit in using migration at low loads, because of the limited performance improvement. As the load increases, the difference between IP and the migration-based algorithms first becomes larger, but then levels off or even decreases at high loads. Based on the GRR, the largest migration benefit – about 6% – is achieved under the TK algorithm.

The differences in performance are primarily due to load imbalances that occur under IP. To verify this, we monitored *Idle Waste*, which is defined as the aggregate time processors are idle while there are ready tasks waiting in other queues. Idle Waste is relatively small (less than 15%) at both low and high loads. At low loads the load imbalance is small, because there are few tasks in the system, and at high loads it is low, because there are usually enough tasks to keep all of the processors busy. At medium loads, however, more than 25% of the processing power of the system is wasted as a result of load imbalances. It is for this reason that the migration-based schemes perform better than IP at that load level. (A similar

* More formally, the Offered Load is defined as

$$\frac{Mean-CPU-Burst-Length \times Bursts-per-Task \times Mean-Task-Arrival-Rate}{Number-of-Processors}$$

result was found in a distributed systems study by Livny and Melman [LM82].)

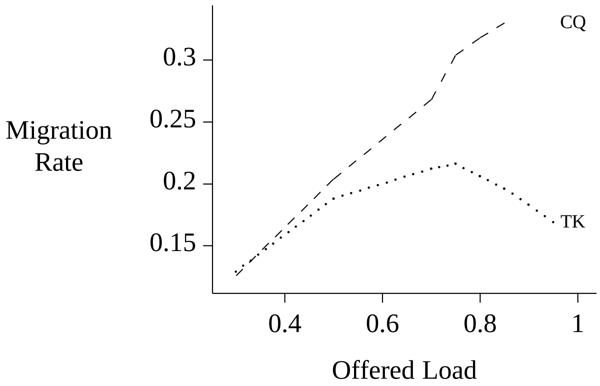

Figure 4.2: The number of migrations per task dispatch versus Offered Load

4.2. Central Queue at High Loads

The Baseline System tests also expose the poor performance exhibited by the CQ algorithm at high loads. In Figure 4.1, the CQ data points at Offered Loads of 0.9 and 0.95 are missing, because the system saturated at those loads (i.e. the task load exceeded the system's processing capabilities). The reason for this behavior can be found by monitoring the number of migrations performed under CQ as the load increases. Figure 4.2 shows the migration rate (i.e. the proportion of task dispatches that result in migrations) as the Offered Load increases. The migration rate increases for both TK and CQ while the Offered Load is increased to 0.7. As the Offered Load moves beyond 0.7, however, the migration rate under CQ continues to increase, but it levels off and begins to decrease for TK. The migration rate under TK decreases at very high loads, because tasks are only migrated when a processor is idle and processors are idle less often as the load increases. At these loads, TK will tend to behave more like Initial Placement: tasks are placed on a processor on entry into the system and typically remain there because processors are rarely idle. This is a positive attribute of TK, since migrations that occur when all processors are busy are of little use.

When CQ scheduling is used, processors always take the task at the head of the global run queue. For a migration to be avoided, the task to be selected must have last executed on the processor that selects it. The probability that a processor looking for work is the same processor that last executed the head-of-queue task is relatively small, resulting in many migrations. The migration rate is reduced at low loads, because tasks are returned (without migrating) to the same processor they last ran on, if that processor is idle when the task becomes ready. At higher loads, however, the processors are busy much of the time, and tasks are therefore assigned to processors more or less at random. Since migration increases the length of the task's next CPU burst (by 1 ms. in the Baseline System), the load is increased by the aggregate migration overhead.

To verify that migration overhead causes the poor performance of the CQ algorithm at high loads, the tests were repeated with the Task Migration Cost parameter reduced to zero. This models an environment where tasks can move from processor to processor with no overhead. The simulation results confirm that the performance of the CQ algorithm is much improved over the original results at high loads (see Figure 5.1), although the migration rate remains essentially the same.

In an attempt to correct the inferior performance of CQ at high loads, several variations on the algorithm were implemented. Instead of always selecting the task at the head of the run queue, some number of tasks on the queue are scanned in the hope of finding a candidate for execution that does not require migration. The number of tasks searched is controlled by a *Search Length* parameter, the value of which ranges from one (equivalent to the original CQ algorithm) to infinity, where the entire queue is scanned (essentially equivalent to the TK algorithm). The new versions of CQ demonstrate better high load characteristics than the original CQ algorithm, and the improvement increases as the Search Length parameter increases. However, since the modified CQ algorithm tends toward the TK algorithm as the Search Length increases, CQ’s performance never exceeds that of TK at high loads. Moreover, the overhead of the modified CQ algorithm itself will become significant with a large search length, as the load and, hence, queue size increase. Since access to the queue must be synchronized, the modified CQ requires that the processor hold the queue while a search of the queued tasks is performed. (In the original CG, a task could be retrieved from the queue in a very short, fixed time.)

5. The Deviations from the Baseline System

In this section, we consider how changes to the input parameters of the Baseline System affect the results. For a more detailed analysis and discussion, see [Cur89].

5.1. Varying Task and System Migration Costs

We simulated the system with the Offered Load set at 0.7, while varying the Task Migration Cost from 0 to 4 ms. The System Migration Cost parameter was held at 5%, as in the Baseline System tests. The results of the tests, shown in Figure 5.1, show again that the CQ algorithm is severely affected by the magnitude of the migration penalty. While the TK algorithm is only minimally affected by the change in the TMC parameter, the system saturates under CQ, when the migration cost rises above 2 ms. Observe that TK is better than IP even when migration costs are quite high.

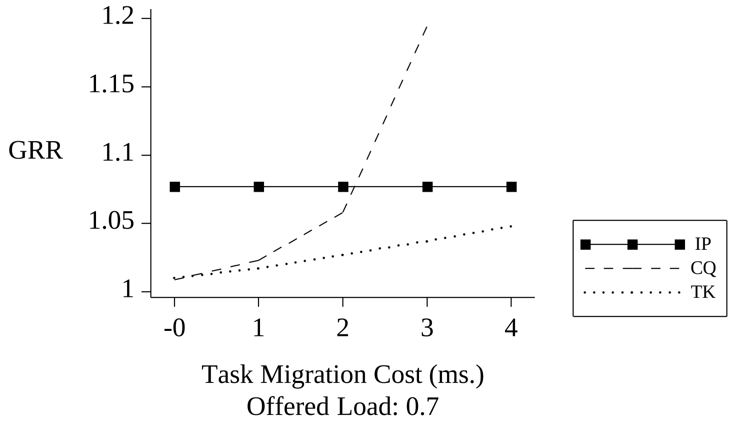

Figure 5.1 The Affect of Task Migration Costs

We now consider the effect of varying the magnitude of the System Migration Cost. Figure 5.2 shows the GRR results of the TK and CQ algorithms compared to the IP GRR, as the system migration costs are increased from 0 to 20% for varying levels of Offered Load. As the system migration overhead increases, the benefit of using a migration-based strategy over Initial Placement decreases. From the graphs, it is also obvious that the impact of the migration penalty on the performance of the algorithm increases with the load. The migration costs affect both the time tasks spend at the CPU and the time they queue for the CPU waiting for other tasks to execute. The extra time at the CPU is insignificant (in our case, about 3% of the average task time). However, the extra time tasks wait in run queues can be significant and, since queueing time increases with load, the penalty of the reduced system performance also increases with load.

5.2. The Effects Varying the Workload

The workload of the Baseline System was derived by extrapolating results of uniprocessor studies to the expected environment of a modern multiprocessor. Inevitably, there will be differences between the workload of uniprocessors and that of the multiprocessors we are studying (and in fact, between one

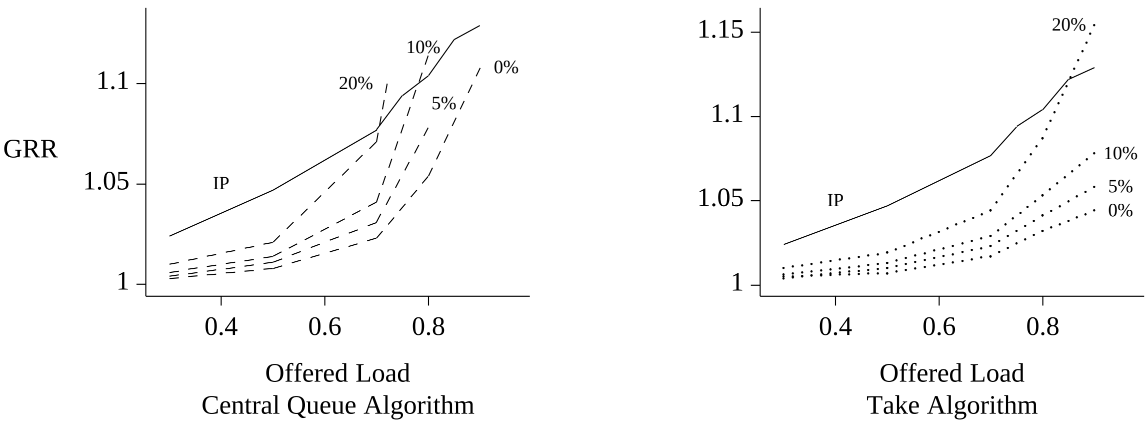

Figure 5.2: The Effects of System Migration Cost on CG and TK

uniprocessor and the next). For example, intuitively, we expect more processor-intensive tasks to run on multiprocessor systems. To simulate these types of tasks we added a *processor-intensive* workload to the original *Baseline System* workload. Most of the parameters of the new processor-intensive workload are identical to those of the Baseline System workload with the exception of the average CPU burst length which is roughly three orders of magnitude larger for the Processor-Intensive workload.

In simulating the combined Baseline and processor-intensive job classes, a fixed Offered Load of 0.7 was applied to the system, and the proportion of the load on the CPU coming from the processor-intensive tasks was varied from 0 to 75%. This represents a range of how a system might be used: from running only interactive tasks (the original Baseline Workload), to a system where most of the load comes from processor-intensive jobs. Figure 5.3 shows the GRR results of the simulations, as a function of the fraction of the Offered Load that comes processor-intensive tasks. As can be seen, processor intensive tasks primarily affect the performance of IP and only to a much lesser extent the other algorithms.

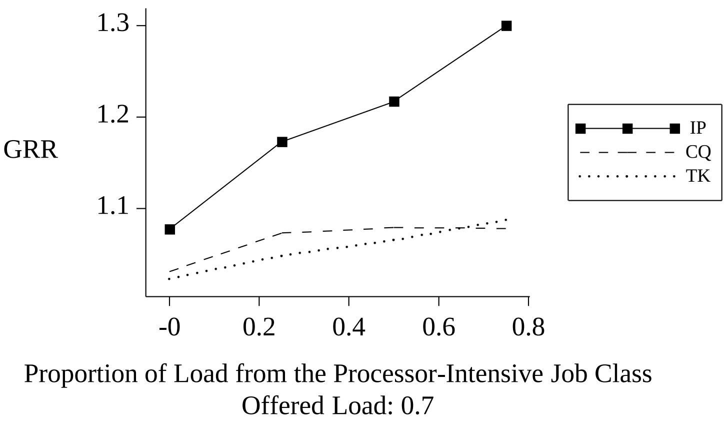

Figure 5.3: The Effects of large tasks

The poor performance of the IP algorithm with processor intensive tasks is due to the way our implementation of the algorithm bases its placement decisions on task counts (as opposed to the load tasks place on the system). While this policy is acceptable if all of the tasks are of similar size, a problem develops when the task size variance is large. A single processor-intensive task in our simulation utilizes virtually all of the resources of a processor, while the average Baseline System task utilizes only a small fraction. Obviously, a processor executing a single processor-intensive task will usually have a significantly higher load than the other processors. When the task placements are based on task counts, the over-loaded processor will also have to execute its share of the smaller tasks, further accentuating the load imbalance.

To verify this, we modified the implementation of IP to count the processor intensive tasks as being 25 times as large as the tasks from the Baseline workload (assuming it is possible to identify such tasks on arrival into the system) when deciding where to place a task. This has the effect that processor-intensive tasks are given virtually exclusive use of a processor for their execution. Once they are assigned to a processor, new tasks are assigned to other processors (unless all of the other processors have task counts of 25 or more). The only Baseline System tasks that compete with processor-intensive tasks are the ones present on the processor when the processor-intensive tasks began executing, but they are small so the conflict time is typically short. The results simulations (not shown) indicate that the modifications improve the performance of the IP algorithm to the range expected from previous studies, where IP performs slightly worse than the migration-based algorithms: a difference of from 5 to 10%.

The modified IP algorithm is, however, not without problems. The additional functionality introduces a considerable degree of complexity to an otherwise simple algorithm. The cost of implementing a job class detection scheme is not insignificant, since the mechanism requires additional processing on every context switch.

The Central Queue algorithm is not subject to the same imbalance as IP, since by using a single queue, the load is always balanced among the processors, regardless of how the load is distributed among the tasks. Under TK, if a processor-intensive task is executing on one processor, the tasks queued behind it will be migrated to other processors as they become idle. (Since the Offered Load is far from 100% in these simulations, there should be plenty of processing power available to execute all of the tasks.)

5.3. The Effects of Supporting Task Priorities on Scheduling

In the simulations presented so far, all of the tasks had the same priority and task queueing was on a first come, first serve (FCFS) basis. In the majority of real systems, however, tasks are assigned priorities and, when multiple tasks are queued for execution, the task with the highest priority is selected. In this section, we how introduction of task priorities affects performance.

For this purpose, we introduced a simple task priority model to the simulation model. The scheme uses four priority levels, roughly analogous to System high and low priority and User high and low priority tasks. Each task is assigned a priority on arrival in the system, with a given proportion of tasks assigned to each priority level. No effort is made to differentiate between the characteristics of tasks running at different priorities.

The scheduling algorithms are changed to support priorities as follows. In the IP algorithm, the processor selected to execute a new task now is the processor with the least number of tasks (ready or blocked) of equal or higher priority than the new task. This is a greedy algorithm in that the task is placed on the processor where it has the fewest number of tasks to compete with, regardless of the number of lower priority tasks it will interfere with.

The TK algorithm was modified such that when a new task arrives in the system, it is placed on the processor with the fewest waiting tasks of equal or higher priority. Again, the task is placed on the processor where it should get the fastest service, regardless of how many lower priority tasks exist. Also, when a processor becomes idle, it searches for the processor that has the most waiting tasks of the current highest priority in its run queue and migrates one of those tasks.

The CQ algorithm need not be changed, since it already supports true priority scheduling on a system-wide basis. (By definition, it is the only algorithm to provide such support. The other algorithms only implement priority scheduling within separate queues and not on a system-wide basis.)

To test the performance of the three algorithms in handling a workload of prioritized tasks, a series of tests were run using the Baseline System parameters with 10% of the tasks assigned to Priority Level 1 (the highest priority), 20% to Level 2, 50% to Level 3 and the remaining 20% assigned to the lowest priority, Level 4. This means that half the tasks are ‘‘normal’’ user tasks, a few are low priority user background tasks (Level 4), while the remainder are higher priority system tasks.

Of primary interest in these tests is how close the separate queue algorithms (IP and TK) perform relative to the true priority scheduling of the CQ algorithm. To see what direct effects priorities may have on IP and TK, they were first compared to CQ under the assumption that migration was free (i.e. both the cost per migration and the processor performance degradation were removed). Perhaps surprisingly, TK performed comparably to the CQ algorithm within each priority class. The GRR for TK was at most 1% worse than that of CQ. As expected from our earlier tests, IP performed significantly worse than the migration based algorithms. The relative performance of all of the scheduling algorithms is therefore unaffected by the use of task priorities.

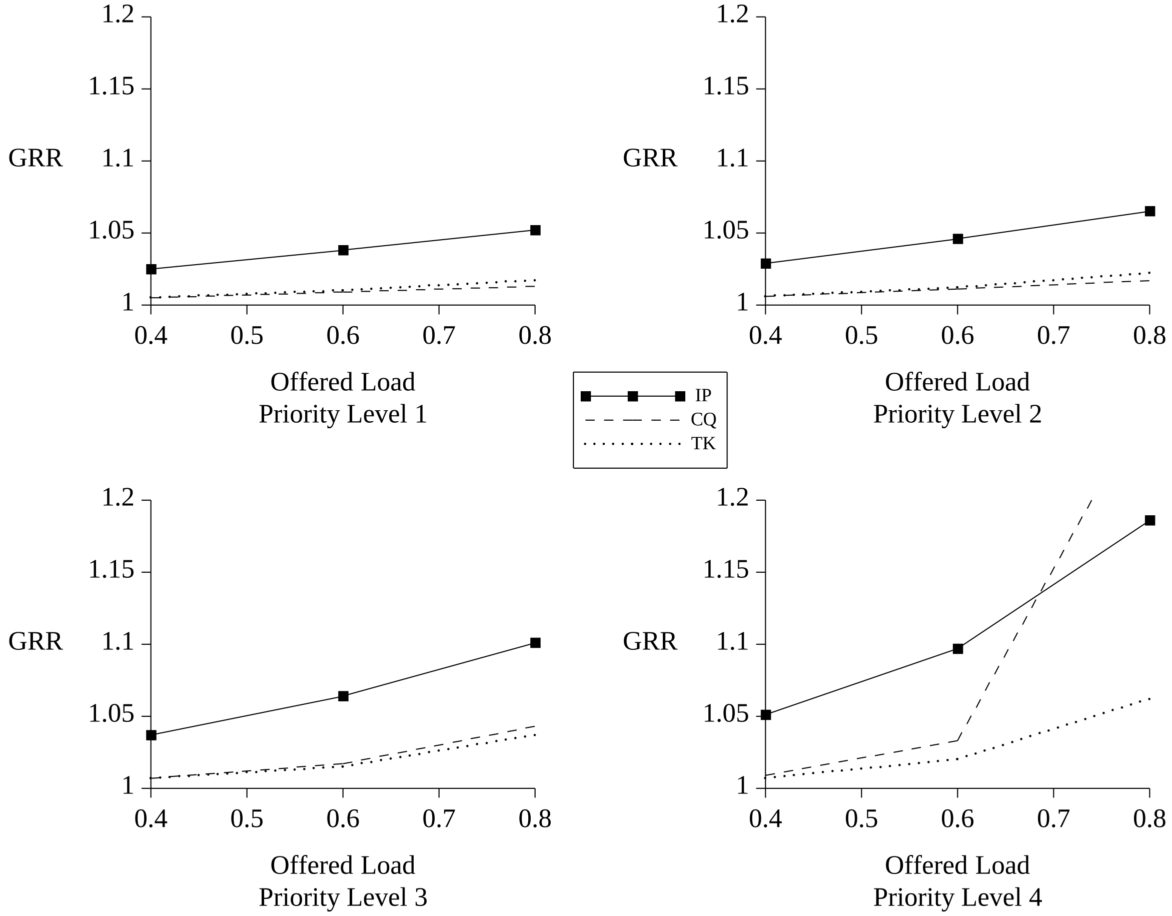

Figure 5.4: The performance of the scheduling algorithms at different priority levels

In the second set of priority tests, the Baseline migration costs (1 ms. TMC and 5% SMC) are included and the same tests are repeated. The results are depicted in Figure 5.4. For the high priority tasks, the results are similar to the case where migration is free. However, the GRR of the low priority tasks under CQ is far higher than that of TK and, in fact, worse than that of IP under very high loads.

Why do the low priority tasks under CQ incur so much overhead? These tasks are at the end of the ready queue, and their execution is delayed not only by the execution time of all tasks ahead of them, but also by the aggregate migration overhead caused by all tasks ahead of them. At high loads, the higher migration rate with its attendant overhead further increases the load on the system, causing an increase in the average number of tasks waiting in the queue, as discussed in Section 4.1. The fact that the poor performance of the low priority tasks under CQ does not occur when migrations are free verifies that migration costs combined with the increased migration rate are the chief performance difficulties.

To reduce the migration rate under CQ, the algorithm was modified to allow a processor looking for work to scan all of the top priority tasks in the ready queue to find one that could be executed without a migration. (If no such task can be found, then the task at the head of the queue is selected for execution). With this modification, a task of the current highest priority is still always selected (i.e. true priority scheduling is retained), but tasks are no longer scheduled in strictly FCFS fashion. The simulation results, (not shown) indicate that the modifications have the desired effect − the response ratio of the low priority tasks improve, to a level only slightly worse than those under TK. With this modification, CQ again performs up to 1% better than TK for the high priorities, and the migration rate is reduced from 30% to 24%.

Of course, in systems with many priority levels (such as Unix), such a change is not realistic while retaining true priority scheduling, since it is unlikely there will be multiple tasks to choose from at any one priority level. However, by deviating slightly from true priority scheduling and grouping priority levels together and employing the same strategy on a group level, similar results can be achieved.

6. Concluding Remarks

A number of factors were found that affect the performance of the scheduling algorithms. First, the *processor load* impacts the performance of the scheduling algorithms studied. Under low loads (less than 50% utilization), all of the algorithms perform comparably. At higher loads, the migration-based algorithms perform better than the non-migrating Initial Placement algorithm, because of a more equally balanced load.

Second, the *cost of migration* affects scheduling algorithm performance when the load on the processors is high. When the task migration cost is high, the performance of the Central Queue algorithm deteriorates because of its tendency to frequently migrate tasks at high loads. The Take algorithm migrates tasks less frequently at high loads and, as a result, is less sensitive to changes in task migration costs. System migration costs also affects performance. At low or medium loads, the effect of system migration costs is negligible, because the tasks' CPU times are dwarfed by their I/O time. However, at high loads, queuing becomes a significant factor in the time it takes a task to complete, making the system migration costs much more significant.

Finally, changes in the *workload* affect performance. The Initial Placement algorithm is sensitive to the introduction of processor-intensive tasks to the job mix, while the migration-based algorithms, Central Queue and Take, are not. With the introduction of prioritized tasks, the Central Queue algorithm gives low-priority tasks a poor response, while priorities have little impact on the IP and Take algorithms.

Overall, we found the difference in performance between algorithms to be relatively small, usually less than 10%. Although the *Initial Placement* algorithm performs consistently worse than the migration-based algorithms, it performs adequately at loads below 0.5. It deteriorates faster than the migration-based algorithms as the load increases (up to a load of 0.8). Initial Placement also performs poorly when there is a large variance in the size of the tasks. For example, two very large tasks that happen to have been assigned to the same processor will compete with each other for the processor's cycles, even though other processors may have become idle in the mean time. This problem can be avoided by using load measures that are more closely tied with the actual utilization of the processors by each task. However, such schemes are slower (require more processing overhead) and more complicated to implement.

The primary benefit of using *Central Queue* scheduling is its adherence to pure priority scheduling, a feature unique to the algorithm. Although in most situations CQ's performance is comparable to that of the TK algorithm, the algorithm performs much worse in some circumstances. In particular, CQ performs poorly under high loads and when the cost per migration is high. This is a direct result of the algorithm's high migration rate when the load is high. It is not surprising that the algorithm provides the best service for high priority tasks, since the Central Queue algorithm is the only algorithm that employs system-wide priority scheduling. However, its handling of low priority tasks can be poor under high loads, when most of the migration overhead is passed on to the low priority tasks.

The performance of the *Take* algorithm is as good or better than the other algorithms under all conditions studied. Perhaps the key attribute of the algorithm is that at high loads, its migration rate decreases.

Somewhat surprisingly, Take schedules high priority tasks almost as well as CQ (within 1 or 2% in the simulations), even under conditions favorable to CQ (i.e. when migrations are free). Therefore, although the Take algorithm does not provide true system-wide priority scheduling, it appears to provide performance very close to that of CQ for the scheduling of prioritized tasks, in practice.

In conclusion, of the algorithms we studied, we found the Take algorithm to be the most suitable for scheduling a Unix workload on small-scale, shared-memory multiprocessors. It is easy to implement and it performs best under most operating conditions. Although the difference between the performance of the three algorithms is relatively small, the behavior of the Take algorithm appears much more stable than the other two algorithms under extreme conditions.

References

[Bla90] D.L. Black, "Scheduling support for concurrency and parallelism in the Mach Operating System", *IEEE Computer*, 23(5), 1990, pp. 35–43.

[Cur89] S.W. Curran, *A Simulation Study of Shared-Memory Multiprocessor CPU Scheduling Algorithms*, Masters Thesis, University of Toronto, 1989.

[ELZ88] D.L. Eager, E.D. Lazowska, and J. Zahorjan, "The limited performance benefits of migrating active processes for load sharing", *Proc. 1988 ACM Sigmetrics Conf. on Measurement and Analysis of Computer Systems*, 1988, pp. 63–72.

[Gon77] M.J. Gonzalez, "Deterministic processor scheduling", *Computing Surveys*, 14(3) 1977, pp. 173–204.

[Kel89] M.H. Kelley, "Multiprocessor aspects of the DG/UX kernel", *Proc. 1990 Winter Usenix Conf.*, 1990, pp. 85–99.

[LO86] W.E. Leland and T.J. Ott, "Load-balancing heuristics and process behavior", *Proc. Performance ’86*, 1986, pp. 54–69.

[LV90] S.T. Leutenegger and M.K. Vernon, "The performance of multiprogrammed multiprocessor scheduling algorithms", *Proc. ACM Sigmetrics 1990 Conf. on Measurement and Modeling of Computer Systems*, 1990, pp. 226–236.

[LM82] M. Livny and M. Melman, "Load balancing and homogeneous broadcast distributed systems", *Proc. ACM Computer Network Performance Symposium*, 1982, pp. 47–55.

[LT88] T. Lovett and S.Thakkar, "The Symmetry Multiprocessor System", *Proc. 1988 Intl. Conf. on Parallel Processing*, 1988.

[MEB88] S. Majumdar, D. Eager, and R. Bunt, "Scheduling in multiprogrammed parallel systems", *Proc. ACM Sigmetrics 1988 Conf. on Measurement and Modeling of Computer Systems*, 1988, pp. 104–113.

[MNO86] R. Moore, I. Nassi, J. O’Neil and D.P. Siewiorek, *The Encore Multimax: A multiprocessor computing environment*., Technical Report ETR 86-004, Encore Computer Corporation, 1986.

[Mul85] S.J. Mullender, *Principles of Distributed Operating System Design*, Habilitation Thesis, Vrije Universiteit te Amsterdam, 1985.

[NW89] L.M. Ni and C.F.E. Wu, "Design tradeoffs for process scheduling in shared memory multiprocessor systems", *IEEE Trans. on Software Eng.*, 15(3), 1989, pp. 327–334.

[Ous82] J. Ousterhout, "Scheduling techniques for concurrent systems", Proc. Distributed Computing Systems Conf., 1982, pp. 22–30.

[RW87] C.H. Russel and P.J. Waterman, "Variations on Unix for parallel-processing computers", *Comm. of the ACM*, Dec. 1987, pp. 1048–1055.

[Stu88] M. Stumm, "The design and implementation of a decentralized scheduling facility", *2nd IEEE Conf. on Workstations*, 1988, pp. 12–22.

[TSS88] C.P. Thacker, L.C. Stewart, and E.H. Satterthwaite, Jr., "Firefly: A multiprocessor workstation", *IEEE Trans. on Computers*, 37(8), 1988, pp. 909–920.

[TG89] A. Tucker and A. Gupta, "Process control and scheduling issues for multiprogrammed shared-memory multiprocessors", *Proc. 12th ACM Symp. on Operating System Principles*, 1989.

[Wen87] J.W. Wendorf, *Operating system/application concurrency in tightly-coupled multiple-processor systems*, PhD Thesis, Carnegie-Mellon University, Aug. 1987.

[ZM90] J. Zahorjan and C. McCann, "Processor scheduling in shared memory multiprocessors", *Proc. ACM Sigmetrics 1990 Conf. on Measurement and Modeling of Computer Systems*, 1990, pp. 214–225.

[Zho88] S. Zhou, "A trace-driven simulation study of dynamic load balancing", *IEEE Trans. on Software Eng.*, 11(9), 1988, pp. 1327–1341.