Deterministic Batch Scheduling Without Static Partitioning

Nawaf Bitar
Robert English
Kostadis Roussos

Silicon Graphics Computer Systems
Mountain View, California
{nawaf, renglish, knr}@engr.sgi.com

ABSTRACT

This paper describes a dynamic batch process scheduler that provides deterministic batch scheduling and increased system throughput without static partitioning of the system. Statically partitioning allows resources to be targeted at specific classes of processes, but can lead to waste. Our system gives batch processing priority over resources needed to complete on schedule, but reduces waste by allowing idle resources to be used. The batch scheduler consists of Miser, a user process which schedules job resources, and kernel support for delivering resources to jobs on schedule. The combination allows flexibility while improving responsiveness and performance.

1. Introduction

Supercomputers must support two very different types of use. On the one hand, they must deliver fast turnaround for non-interactive applications, and on the other, they must deliver fast response time for interactive developers and users. These two classes of use place different demands on the system and demand different scheduling disciplines. Good interactive response requires process resources to be consistently resident in the system. Good batch turnaround requires many large jobs to be scheduled simultaneously, with resource requirements that may greatly exceed the capacity of the machine. Interactive tasks also differ from batch requests in their unpredictability. While a batch request will often carry with it specifications of time, memory, or CPUs required, interactive tasks usually lack descriptive information.

As an illustration of the difference between time-sharing and batch scheduling, consider a bank with one teller and 50 customers. First suppose that we have no knowledge of the transactions each customer will perform, and that we know that a significant number of customers have short transactions (for example, asking the time of day or seeking change for a dollar). If all customers are served in order, some customers will wait a very long time for a small amount of service. A time-sharing schedule that gave each customer in turn a small amount of service would be appropriate and fair. Suppose, on the other hand that we knew each customer would require one minute of service. If we adopted a time-sharing policy that gave each customer in turn one second of service, every customer would wait 50 minutes for service. Serving the customers in order would reduce the average response time by half, without penalizing the final customer relative to the time-sharing schedule.

Since the requirements and information available for the two types of jobs are distinct, it is common practice to partition system resources and satisfy requests separately. Ashok and Zahorjan[AsZa92], for example, show that balancing the requirement of batch performance and interactive response time requires partitioning. Static partitioning, however, wastes resources by denying idle resources to classes of jobs even when they are not required by the preferred class. The system we describe here attempts to balance the needs of different classes for reserved resources with the need to get full utilization from all system resources. Resources are partitioned, but the partitions are soft, and can change in response to changes in load.

2. Related Work

As Ashok and Zahorjan[AsZa92] point out the main problem with allocating resources between batch jobs and interactive jobs is that they have different resource requirements. A typical supercomputer batch job typically requires a large amount of uninterrupted CPU time and physical memory at some point in the future. A typical interactive job requires that the job get immediately serviced when the job is active so that the response time is low. The problem is that when a batch job is running the worst thing that could happen is that it relinquish the CPU whereas, for an interactive job the worst thing that could happen is that it have to wait for a long time before it ran on the CPU.

Ashok and Zahorjan[AsZa92] consider the competing issue of batch requirements and interactive jobs and concludes that to achieve optimal performance of the system, memory should be statically partitioned and CPUs be dynamically partitioned. In his paper he considers the performance of a single batch job that runs continuously and a series of interactive jobs and concludes that the optimal throughput of the batch job is achieved when the batch job is never blocked waiting for memory. Good interactive response time is achieved by reserving sufficient memory for typical interactive loads and allowing interactive applications to steal batch CPUs when they become active and acquire memory.

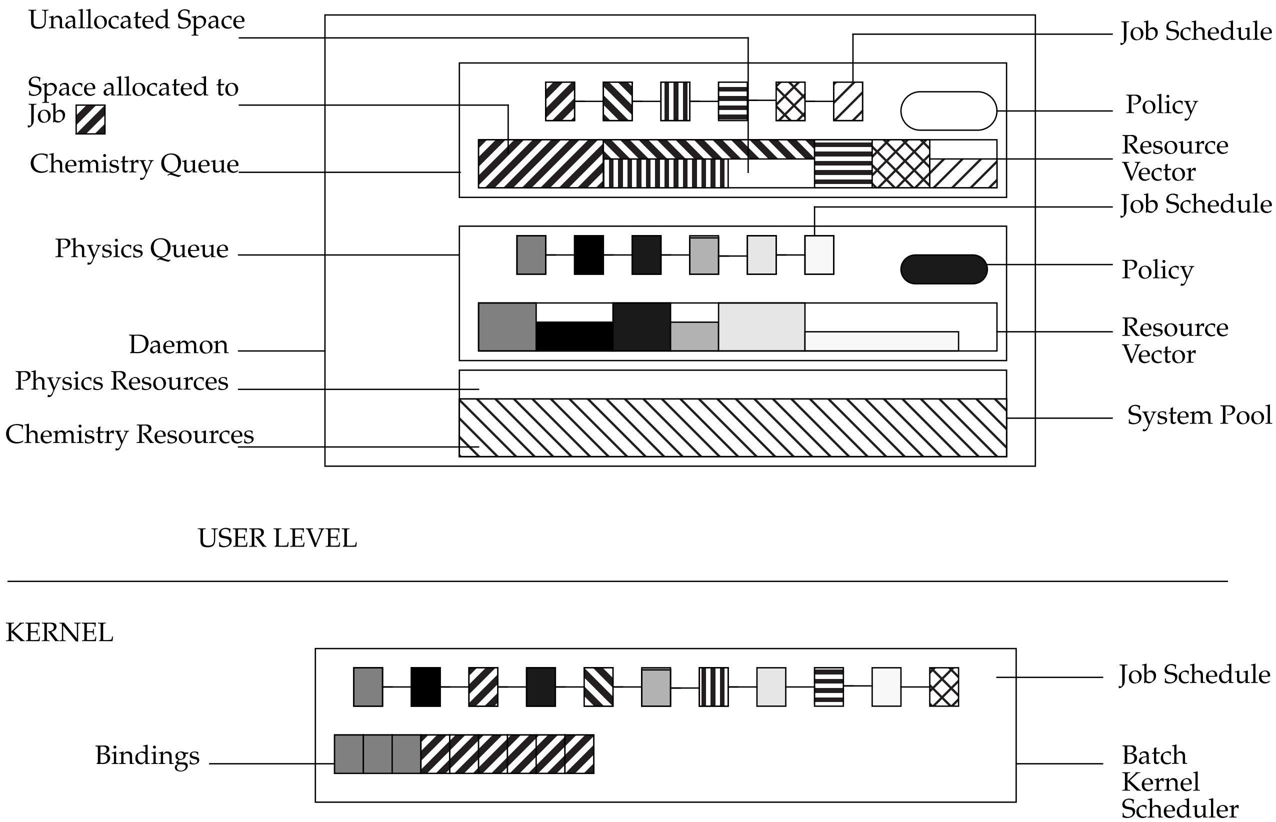
Ashok and Zahorjan do consider the issue of memory waste but in their model, memory that is unused by one class is unavailable to the other class. Furthermore, Ashok and Zahorjan dismiss the issue of cost of switching between processes as insignificant and ignores issues of cache affinity. Ashok and Zahorjan do not consider the issue of performance of a particular batch application that dynamically space shares its processors with a time-share process. Their main interest is the performance improvement of a batch application if the working set is always in memory versus if it can be swapped out and the consequent throughput improvement.

Although the theoretical results show that static partitioning of memory and dynamic allocation of CPUs is the optimal configuration for supercomputers, in practice, supercomputer system administrators statically partition their systems, so as to guarantee CPU time and memory availability, because current state of the art batch schedulers require static partitions to provide deterministic deadlines. Users want deterministic guarantees about CPU execution time so that they can schedule long or repetitive calculations. System administrators want deterministic guarantees so that they can bill users of the systems. System administrators statically partition their systems using tools such as PSET[PSET] on the IRIX Power Challenge and the Sub Complex Manager[SCM] on the HP Convex. State of the art batch schedulers such as Codine[KaNe94] and LoadLeveller[KaNe94][LL], allow the user to create queues of resources and schedule jobs against those resources but cannot guarantee execution time because of the lack of kernel support to prevent jobs that are not scheduled via the batch system from running on resources allocated to it. Both LL and Codine allow the user to specify the resource requirements, with Codine allowing the user to specify CPU run time as well. LL and Codine ensure that the system is not overcommitted.Static partitioning of the CPUs make it possible to guarantee resources using a user level batch scheduler because it excludes all but particular threads from running on CPUs, unfortunately, any idle resources in any static partition is unusable by other partitions.

3. The System

Miser aims to provide deterministic run times of batch applications while allowing idle resources to be used. A predictive scheduler uses information supplied by the user to reserve resources needed by batch jobs during their runs. A reactive scheduler in the kernel makes idle resources available to any process able to use them. For any resource, the preferred class of job retains absolute priority over its use.

Figure 1: Layout of Miser



Miser [Figure 1] consists of a user level scheduler and a kernel level scheduler. The user level scheduler is a user level process that acts as a server processing requests to schedule batch jobs [Figure 2], which are UNIX process groups from which processes cannot exit. The user level scheduler generates a schedule of start and end times for batch jobs consistent with the resources that are allocated to it and delivers that schedule

Figure 2: Batch Job Structure

Start Time
End Time
Deadline
Batch ID (Process Group ID)
Memory
CPUs
Urgency
Thread List

to the kernel. During its scheduled period, a batch job has priority over its allocated resources and is prevented from exceeding its resource requests. Allocated resources that it does not use are available for use by other tasks within the system. Prior to its scheduled period, a batch job may use idle system resources but does not compete for them. When its scheduled period ends, the batch job is either terminated or returned to the predictive scheduler for further action. The use of a kernel level batch scheduler allows us to guarantee CPU time to batch jobs without requiring CPU static partitioning.

The life-cycle of a typical batch job is detailed in Figure 3. A time-sharing process becomes a batch job by requesting to be scheduled as one through the miser_submit system call. The kernel delivers its resource requirements to the Miser daemon. The daemon's response converts the process to batch and places it on the batch queue, where it stays until a CPU becomes idle or until its start time. If the batch job has not yet terminated and the current time is equal to its start time, the job becomes *batch-critical,* is bound to a set of CPUs, and begins to run. At that point the he job has the highest non realtime priority and will preempt all non-realtime threads running on the CPU.Before it can begin to run the job must acquire all the physical memory that it requires to run. If insufficient number of physical pages exist, then the system pages out any non-batch-critical jobs until it gets the physical resources it requires. The time taken to perform this operation is part of the batch running at time. The job will continue to run at batch critical until the job runs out of time or terminates normally. In either case, the kernel releases any resources associated with the job.

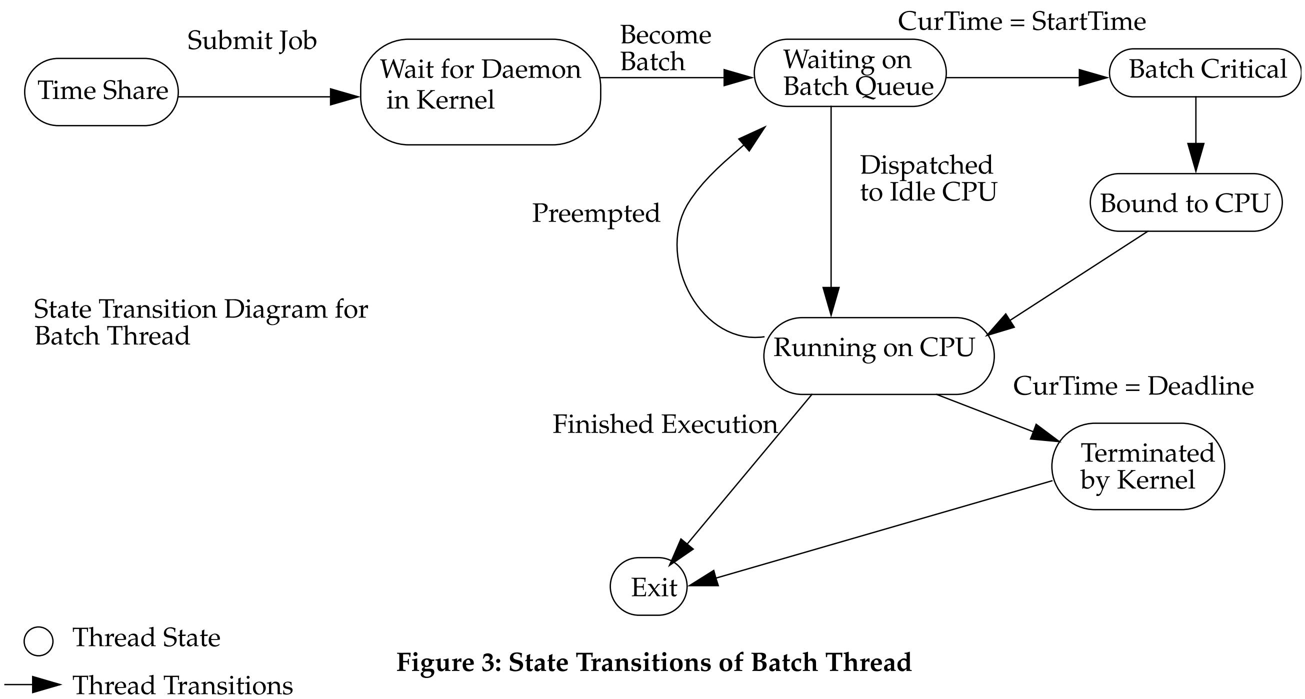


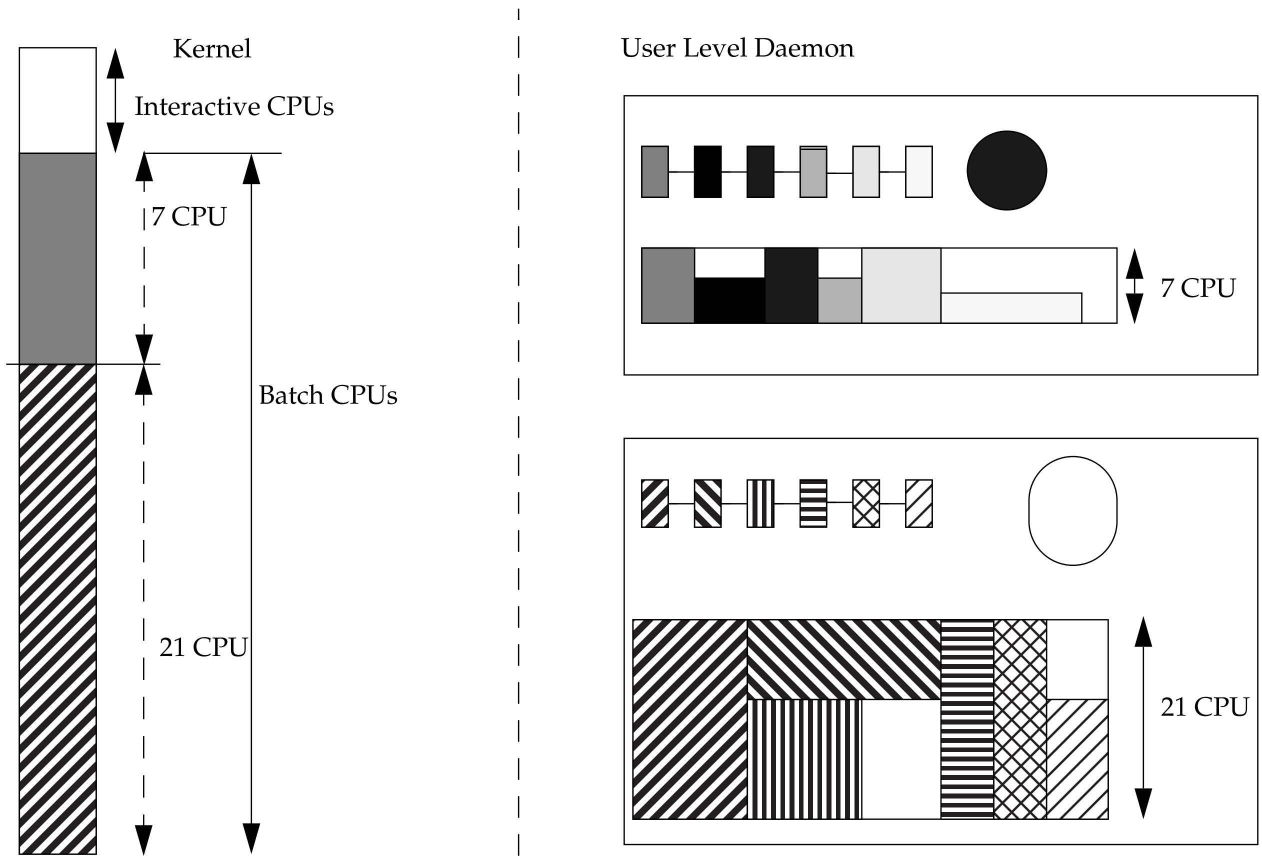
Figure 3: State Transitions of Batch Thread

3.1 User Level Scheduler

The user level scheduler [Figure 1]is a daemon that generates a non-conflicting schedule of jobs with known time and space requirements. The daemon, given a set of jobs, searches through time/space to find an allocation that fulfills the job requests subject to policy constraints. The daemon has an administrative interface that allows for modification of most parameters without requiring a restart.It runs as a trusted process with a kernel-mediated interface and accepts requests for process scheduling, process state changes, and batch system configuration control, and returns values and status information for those requests.

The daemon manages the time/space pool using a hierarchy of resource vectors indexed by time. All vectors are of equal size. Each entry in a vector specifies a set of resources allocated to the vector: a number of CPUs and an allocation of physical memory. The root vector defines the set of resources allocated to the daemon; all other vectors allocate their resources from the root, and may, in turn, allocate resources to sub-vectors of their own. The root vector defines the maximum amount of resources that all batch jobs currently scheduled can use at any one time. Associated with each vector is a generating function (to allocate new entries as time advances), a set of jobs with resources allocated from the vector, a policy module that controls the scheduling of jobs, and miscellaneous policy-related data. The collection of a resource vector and its associated code and data structures is called a Queue.

Users submit jobs to the Queue using a user level command or system call (*miser_submit*) which specifies the Queue to which the job should be attached and an array of resource requests to be made against that Queue. The kernel allocates a batch id to the job and passes the request, batch id and authentication information to Miser. The policy module schedules the job against the Queue's resources. The resulting allocation is returned to the user and the kernel via *MISER_RESPOND*. The fact that the resources came from a particular queue is of no consequence to the kernel, but the kernel records the job's Queue in case a Queue requires reconstruction.



(a) Kernel State with batch critical jobs executing

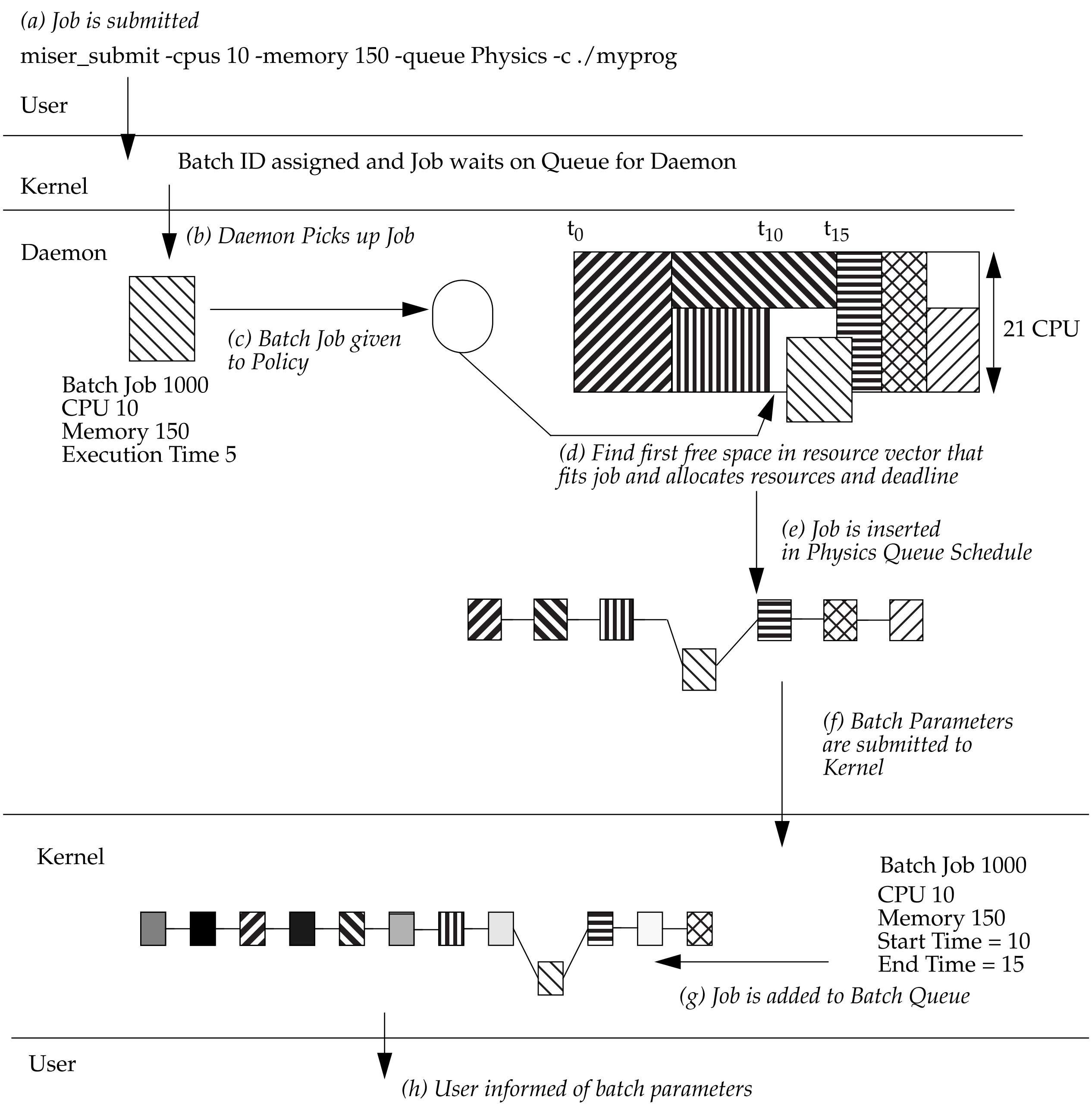
(b) Status of Miser

Figure 4: Configuration of Miser with Physics and Chemistry Department

As an example, consider the case of a supercomputer with 32 processors that has two sets of user the physics and chemistry departments. Both departments agree that they need time on the system to develop their applications, and they needed dedicated batch time to run them. Furthermore, the departments agreed that the physics department should get three quarters of the system when it needs to run particular applications. The departments also agreed that four CPUs should be reserved for time-shared jobs. The system is configured therefore, to have 28 CPUs reserved for batch applications, and four CPUs reserved for time-shared applications [Figure 4(a)]. The daemon is then configured to have two Queues, one with 21 CPUs for physics applications, and one with 7 CPUs for Chemistry applications [Figure 4(b)]. A Physics job that is *batch-critical* uses all the CPUs allocated to the Queue will run on 21 CPUs whereas a Chemistry Job will run on at most 7 CPUs as show in [Figure 4(b)]. Now consider the case of a job being scheduled on the Physics

Queues [Figure 5]. Assume a new job is submitted to the Physics Queue [Figure 5(a)]. After the daemon is informed of the job [Figure 5(b) and its parameters by the kernel it will look for space in the Physics Queue using the available policy [Figure 5(c)]. Finding space involves searching the time/space resource vector [Figure 5(d)] to find a period of time in which the job can execute. After the Job has had space allocated to it, it is then inserted in the Job schedule associated to the Queue [Figure 5(e)], inserted in the kernel batch queue [Figure 5(f)] and the deadline eventually returned to the user [Figure 5(g)]

Figure 5: Scheduling of a Batch Job using Miser



3.2 The Kernel Level Scheduler

The kernel level scheduler is responsible for managing the batch id, dispatching batch jobs that have been scheduled to run, terminating jobs that have exceeded their deadlines, and for scheduling jobs on idle CPUs. The batch queue of the kernel scheduler is an ordered list of batch jobs. The jobs are ordered by their urgency—the time remaining until the job's start time. Before its start time, the job runs opportunistically. After it start time, the job is termed *critical* and has priority over its assigned resources. Threads from critical jobs are bound to particular CPUs to improve affinity, prevent conflicts, and reduce scheduling overhead. Threads keep their bindings for their entire run.

The structure of the general purpose IRIX scheduler makes it possible for the batch scheduler to guarantee deadlines without static partitioning and thereby reduce waste due to idle resources. The scheduler consists of a CPU allocator and a series of schedulers. The CPU allocator gives out CPU time to each scheduler according to their relative priorities. At every time tick it determines if work from a higher priority scheduler is available and will preempt the currently running thread in favor of it. Each scheduler in turn when given CPU time picks the next thread to run. In the particular case of the batch scheduler, and the time-share scheduler, the CPU allocator will preempt a batch job in favor of a time-share job and will in turn preempt a time-share job in favor of batch critical job. When a CPU is idle the CPU allocator will first take work from the time-shared scheduler and then from the batch-share scheduler rather than spin in the idle loop. The two-tiered structure of the scheduler make it possible to provide deterministic deadlines and improve throughput because the kernel batch scheduler and CPU allocator satisfy a number of important properties. The kernel scheduler achieves deterministic deadlines because a batch job that is *batch-critical* executes uninterrupted on the CPU it is bound to. Batch schedulers that lack kernel support are unable to preempt running threads and must resort to static partitioning to guarantee CPU time. Throughput is improved because we are allowing other active threads to execute on the CPUs reserved to the batch scheduler when they are idle. The excess idle CPUs are then used by other applications achieving throughput improvement.

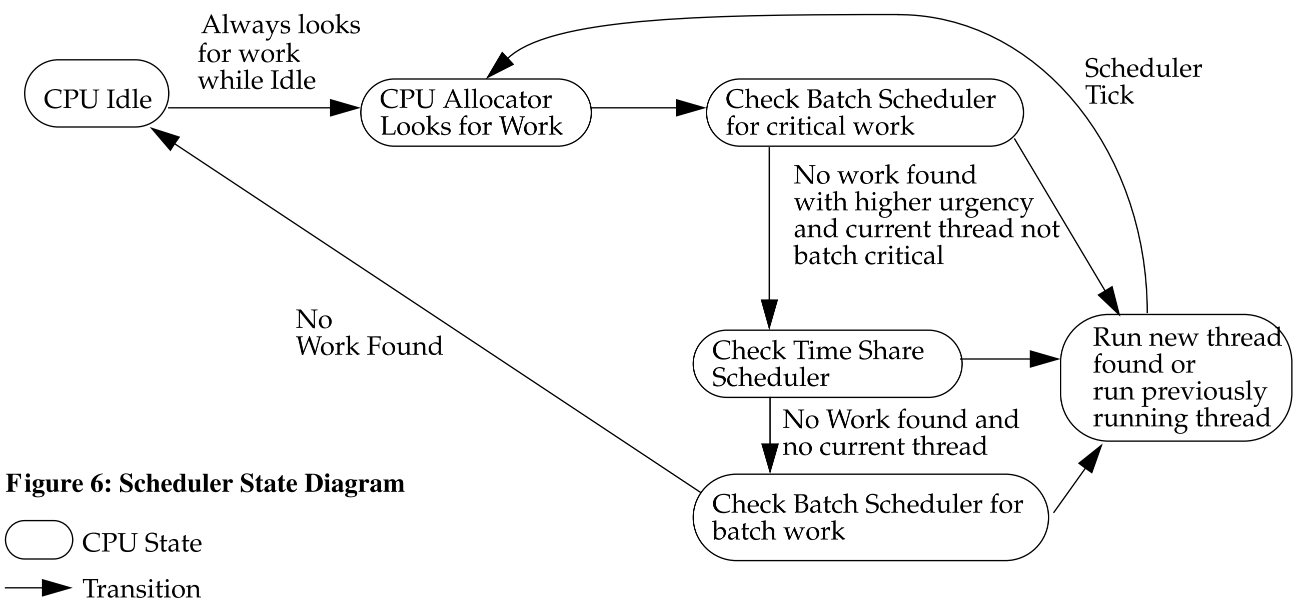


Figure 6: Scheduler State Diagram

As an example consider a particular CPU that has been allocated to the batch scheduler [Figure 6]. Assume that there is a time-share and batch load on the system. Consider the case where the CPU is idle. The CPU allocator will continuously scan for work. If there is any batch work available it will begin to execute that work. If at some later scheduler tick time-share work becomes available it will preempt the batch thread and begin running the time share thread. Assume that at some point the batch scheduler binds a batch critical thread on the CPU. The time-share thread will then be preempt and returned to the time-share scheduler. At every clock tick thereafter, the CPU will check to see if there is higher priority work to execute, and if not will continue to execute the current thread. At some point the thread will run out of time on the CPU. At that point, the batch scheduler will terminate the thread. When that happens the CPU will become idle and begin looking for work.

4. Performance

4.1 Overview

In this section we provide experimental evidence that a particular batch job terminates by its deadline regardless of the time sharing load, and that throughput of batch jobs improves by using Miser. The first experiment shows that the performance of the batch job on an idle system is equivalent to the performance of the same when run time-shared. The second experiment demonstrates that the performance of a batch critical job remains constant regardless of the load on the system which implies that deadlines are met. Finally a set of experiments demonstrates how batch and time-share throughput improves by using Miser.

4.2 Methodology

The experiments were conducted on an SGI Challenge with 16 R4000 CPUs, 16 R4400 CPUs and 512megabytes of main memory. The experiments use a modified version of EP from the NAS parallel benchmarks called MEP. Modified EP (MEP) runs the EP benchmark and measures the time take to complete an entire run of the EP parallel loop. An n-way MEP is defined to be a set of EPs that have n threads. The metric used to measure throughput is either the total number of EPs completed over some time interval (Total EPs) or the rate at which EPs are completed (EPs/sec).

4.3 Experiments

The first experiment demonstrated that the throughput of EPs/sec on an idle system is equivalent to that under Miser on an idle system. EP was run on an idler system and under Miser. In both cases, EP throughput was 0.47/sec.

The second experiment demonstrated that the number of EPs per second does not degrade when there is a CPU intensive time-shared load. MEP was run an increasing number of spinners, to simulate CPU intensive load, that were bound to a particular CPU. Throughput remained constant at 0.47 EPs/second as the number of spinners varied from 0 to 48.

A third experiment compared the number of iterations completed using Miser against static partitioning. In this experiment, a Physics Queue was defined with 5 CPUs and a Chemistry Queue with 11. The Physics Queue ran four five-way MEPs, while the Chemistry Queue was idle for the entire duration of the experiment. The throughput, the total number of EPs completed, was measured for the system running Miser, and compared against an assumed linear throughput graph for a static partition of 5 CPUs. The Miser-managed system has higher throughput because it uses idle CPUs [Figure 9].

A fourth experiment compared interactive and batch throughput on a system with varying interactive load against the throughput that could be achieved with static partitioning. An commercially available interactive benchmark was used to simulate the interactive load because it has a tunable load and was designed to simulate a large interactive system. [1] In this benchmark, load is offered in units of users, and throughput is measured in completions/sec. Beyond a some load point, the benchmark becomes I/O bound, making further load increases are meaningless for our purposes. In the test configuration, I/O saturation occurred at 150 users.

1. The particular benchmark is not important, and cannot be mentioned because this is not an official run. The goal was not to achieve a high score on that benchmark, but to observe the effects of running the benchmark on overall system performance.

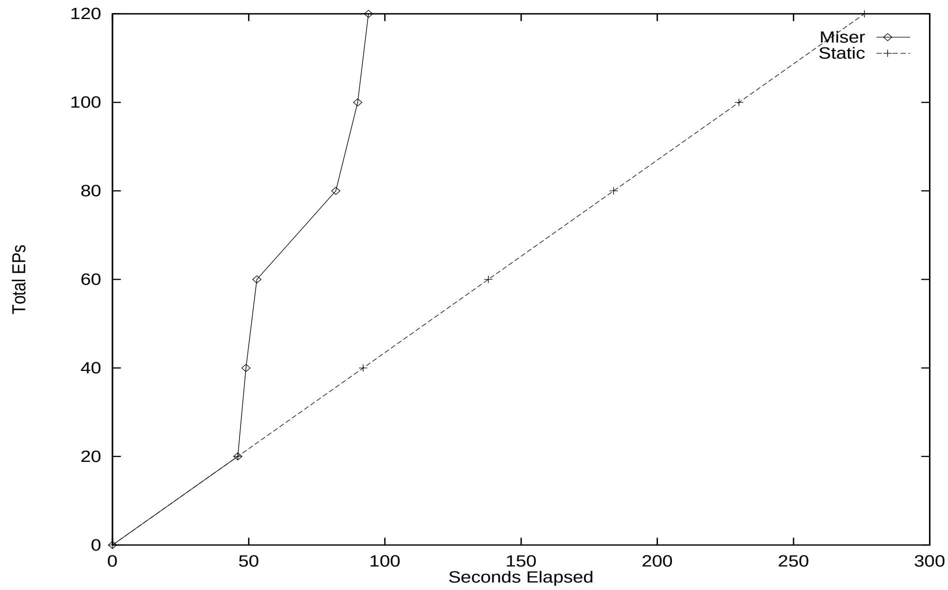


Figure 9: EP Throughput on Idle Batch System

Figure 10 shows the performance of the timesharing benchmark with a batch load. The batch load consisted of 5 way meps that started within one second of the previous one terminating. Interactive throughput is plotted over a range of interactive load levels. The same load levels were also run against a system with 5 CPUs disabled to simulate static partitioning. The Miser-managed system has better interactive performance than the statically-partitioning system because interactive jobs can use batch CPUs.

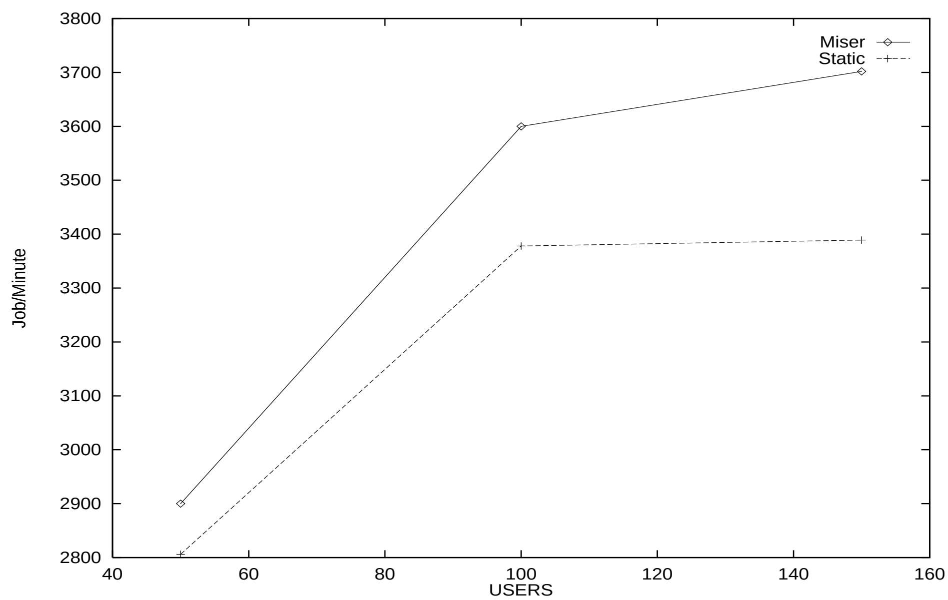


Figure 10: Interactive throughput with Batch Load

Finally, Figure 11 shows the Miser-mediated EP throughput as a function of the increasing interactive

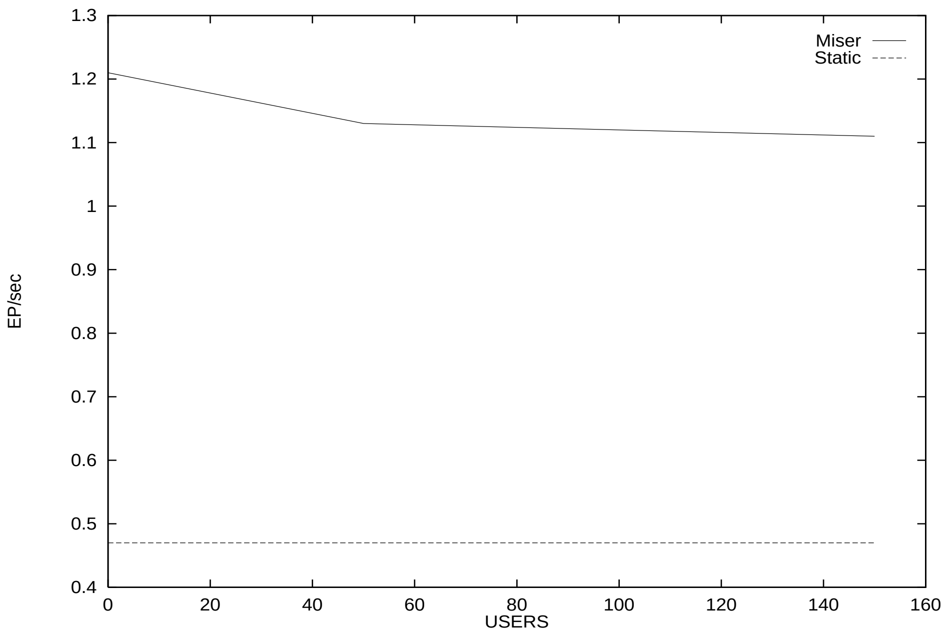


Figure 11: Throughput with Increasing Time-Share Load

execute on CPUs not being used by time-share jobs, throughput with Miser case is in all cases substantially higher than with static partitioning. As expected, batch throughput decreases somewhat as interactive load increases and available CPU power decreases.

5. Future Work

Miser does not schedule memory topologically or allow applications to have data sets larger than physical memory. Large memory jobs will require VM support to allow them to page against themselves rather than competing for memory with the rest of the system. Miser must also be enhanced to account for memory location when allocating memory to batch jobs on NUMA architectures such as the SGI Origin series. It may not be enough to ensure enough memory in the system; Miser may have to ensure the presence of memory on specific nodes or in specific shapes. Here, Miser would have a fundamental advantage over normal kernel mechanisms because Miser jobs have expected run times.

Miser is currently implemented against a single kernel with no fault containment. It needs to be extended to work both in traditional distributed environments (arrays) and across distributed kernels (Cellular IRIX). Both cases will require enhancing Miser's failure recovery mechanisms.

6. Conclusions

Theoretical results have shown that the dynamic partitioning of CPUs and the static partitioning of memory provide the best batch throughput and interactive response time. These theoretical results must be balanced against the real requirement for deterministic batch scheduling that has forced system administrators to statically partition memory and CPU time. Our system departs from the norm of user-level schedulers by also having kernel support. Using the underlying kernel scheduler we are able to guarantee CPU time to batch jobs and are thus able to guarantee deadlines for particular applications. Furthermore, because we have no static scheduling we are able to schedule jobs on CPUs that other batch schedulers must leave idle to achieve guaranteed performance and achieve better throughput, which we have demonstrated. Miser is a new mechanism for scheduling batch jobs with deterministic deadlines without the implied waste of resources due to static partitioning.

7. References

[AsZa92] I. Ashok, J. Zahorjan, "Scheduling a Mixed Interactive and Batch Workload on a Parallel, Shared Memory Supercomputer", Supercomputing 92.

[KaNe94] J. Kaplan, M. Nelson, "A Comparison of Queueing Clustering and Distributed Computing Systems", NASA TM 109025, June 1994.

[Pre94] S. Presnell, "The Batch Reference System", Documentation for Batch, a batch scheduler.

[ZhWa92] S. Zhou, J. Wang, X. Zheng, P. Delisle, "UTOPIA: A Load Sharing Facility for Large, Heterogenous Distributed Computer Systems", University of Toronto Tech Report CSRI-257, 1992.

[ZhWa93] J. Wang, S. Zhou, K. Ahmed, W. Long, "LSBATCH: A Distributed Load Sharing Batch System", University of Toronto, Tech Report CSRI-286, 1993

[BaHe92] A. Bayucan, R. Henderson, T. Proett, D. Tweten, "Portable Batch System: Reference Guide", NASA, May 1996

[BB95] J. Barton, N. Bitar, "A Scalable multi-discipline, multi-processor scheduling framework for IRIX", In Dror G. Feitelson and Larry Rudolph, *Job Scheduling for Parallel Processing*, volume 949, pages 45-49. Springer, 1995

[PSET] man Pages(1M): User Commands, IRIX 6.2, Silicon Graphics 1995

[SCM] SubComplex Manager, static partitioning facility by HP. On line documentation: http://www.convex.com/prod_serv/exemplar/exemplar_scm.html

[LL] LoadLeveller, batch scheduler by IBM. On line documentation: http://www.austin.ibm.com/software/sp_products/loadlev.html

[QB] QBATCH, batch scheduler. On line documentation http://www.cmpharm.ucsf.edu/~srp/batch/qbatch