Deterministic Batch Scheduling Without Static Partitioning

Kostadis Roussos[1], Nawaf Bitar[2] and Robert English[3]

[1] kostadis@sgi.com
SGI, 2011 N. Shoreline Blvd, USA
[2] nawaf@netapp.com
[3] renglish@netapp.com
Network Appliance, Santa Clara CA, USA

Abstract. The Irix 6.5 scheduling system provides intrinsic support for batch processing, including support for guaranteed access to resources and policy-based static scheduling. Long range scheduling decisions are made by Miser, a user level daemon, which reserves the resources needed by a batch job to complete its tasks. Short-term decisions are made by the kernel in accordance with the reservations established by Miser. Unlike many batch systems, the processes in a batch job remain in the system from the time originally scheduled until the job completes. This gives the system considerable flexibility in choosing jobs to use idle resources. Unreserved or reserved but unused resources are available to either interactive jobs or batch jobs that haven't yet been scheduled. The system thus gains the advantages of static partitioning and static scheduling, without their inherent costs.

1 Introduction

Supercomputers have two distinct resource management problems. The first is the allocation of resources between batch and interactive applications while guaranteeing deterministic deadlines. In order to ensure deterministic run times, batch applications require that resources become available according to some fixed schedule and not be reclaimed until the application terminates. Interactive users expect that resources become available immediately, but can tolerate time-sharing of those resources. When both classes of users share a machine, the batch users experience poor performance since resources are being time-shared with interactive applications. To remedy this problem, supercomputer resources are typically partitioned statically between interactive and batch resource pools that cannot be shared, resulting in wasted idle resources.

The second resource management problem is how to improve throughput of systems that only run batch applications while still guaranteeing resource availability and thus deterministic run times. Batch systems that guarantee resource availability require that applications specify the resources they require, and will only schedule an application on a supercomputer if there are sufficient free resources. This approach results in idle resources, either because the users

overestimate the resources they require, or because there is no batch application that is small enough to run on the subset of resources available on the computer.

We present in this a paper a new approach, consisting of a user level resource manager, Miser, and kernel level support code that address both resource management problems, thus improving throughput and overall system performance. Miser is responsible for generating a schedule of start and end times for batch applications such that the resources are never over-committed. The kernel, on the other hand, is responsible for ensuring that resources guaranteed by Miser are made available according to the schedule specified by Miser, while simultaneously making idle resources available to applications that are not yet scheduled to run.

To support the CPU scheduling requirements of the Miser resource manager, namely good interactive behavior, non-interruptible CPU time to particular applications, and dynamic partitioning of CPUs between batch and interactive applications, we implemented a new scheduler. To support the physical memory requirements, namely guaranteed physical memory for particular classes of applications while simultaneously not wasting physical memory, we added a new accounting data structure to the virtual memory subsystem. The new accounting data structure allows us to reserve memory for applications without actually allocating the memory. As a result, memory is guaranteed to be available, but remains unused until the application actually requires it, and therefore, available to other applications.

It is the combination of scheduler and VM kernel level support that enables our system to deliver better throughput. Our approach enables system administrators to have deterministic run times for batch applications without the inherent waste in static partitioning, as well as improving throughput on dedicated batch systems.

Our paper is divided into five sections. The first section will compare Miser with existing batch systems and describe some theoretical work. The second section will then describe Miser, showing how Miser manages supercomputer resources so as to generate a schedule of applications that does not over-commit the system. The second section will also describe the interaction between Miser and the kernel. The third section will describe the kernel modifications to the scheduler and VM required to support resource reservation and resource sharing. The fourth section will present some empirical evidence that Miser does indeed work. The paper will conclude with remarks on future directions for Miser and remarks on the work as a whole.

2 Related Work

Current commercial batch schedulers only provide static mechanisms for managing the workload between batch and interactive jobs on systems using tools such as PSET [PSET], cpusets [CPUSET], and the HP sub-complex manager [SCM].

Ashok and Zahorjan [AsZa92] considered a more dynamic mechanism of allocating resources between batch and interactive users. They proposed the partitioning of memory between interactive and batch applications and proposed that CPUs be available on demand to interactive applications. Their approach results in wasted memory when there is insufficient interactive or batch load to utilize the reserved memory and, by allowing arbitrary preemption of CPUs to favor interactive users, they cannot guarantee deterministic run times.

Other commercial batch schedulers [LSF][LL] are able to manage resource utilization between batch jobs so as to prevent over-subscription. However, they do not manage resource utilization of applications that are scheduled outside of the batch system. Consequently, on a machine that is shared by interactive and batch users, a batch system can not guarantee resource availability to batch jobs and thus deterministic run times, because it can not control resource usage of the interactive jobs.

Commercial batch schedulers allow resources to go idle even if there is work on the system either because of holes in the schedule or because jobs are not using all the resources they requested. A hole is a point in a schedule of batch jobs where there are still free resources available, but no job sufficiently small enough to use them. A conventional batch system can not start a job that is too big to fit in the hole, because it can not restrict a job to a subset of its resources. Commercial batch schedulers can not exploit resources that are committed but unused, because it has no mechanism for reclaiming them later on if the application should require them.

3 Miser

Miser is a resource manager that runs as a user-level daemon to support batch scheduling. Miser, given a set of resources to manage, a policy to manage the resources, and a job with specified space and time requirements will generate a start and end time for that job. Given a set of these jobs, Miser will generate a schedule such that the resources are never oversubscribed. Miser will then pass the start and end times to the kernel, which will manage the actual allocation of physical resources to the applications according to the schedule defined by Miser.

3.1 Resource Accounting

Miser's resource management was designed both to guarantee that resources allocated to Miser are never oversubscribed, and to give system administrators fine grain control over those resources. The basic abstraction for resource accounting is the resource pool. The resource pool keeps track of the availability of resources over time. To guarantee that resources are never oversubscribed, Miser deducts resources from the resource pool as it schedules applications; if an application requests more resources than are available, it is rejected. To enable fine grain control over the resource pools, Miser uses a two-tiered hierarchy of pools. At

the top level is a single resource pool, the system pool, that represents the total resources available to the system. Below the system pool are one or more user pools whose aggregate resources are equal to or less than the resources available to the system pool. A two-tiered hierarchy of pools, whose size can vary over time, allows administrators to control access to resources either by restricting access to user pools, or by controlling how many resources each particular user pools has. For example, a 32 CPU system that needs to be shared equally between three departments for batch use and the university for interactive use, would have Miser configured to manage 24 CPUs leaving 8 CPUs for general purpose use. The Miser resources would then in turn be divided into three user pools of 8 CPUs each that would have their access restricted to users from their respective departments (see figure 1).

3.2 Job Scheduling

Job scheduling by Miser is similar to job scheduling by other batch schedulers; applications are submitted to Miser and Miser determines when they can run. Unlike conventional batch schedulers, applications that are scheduled to run are not waiting to be started on the host system, but are in a suspended state inside of IRIX waiting for the kernel to actually allocate them real physical resources so that they can run. As a result, the kernel can start the application before its scheduled start time by giving it any currently idle resources. Furthermore, because it is the kernel that is allocating actual physical resources, and not a user level process, the kernel is also able to reclaim any idle resources if it needs them to run another batch process or interactive workload.

To schedule a job, users submit jobs to Miser using the miser_submit command. Miser uses a user-defined policy that has an associated user resource pool to schedule jobs. The policy and the resource pool are collectively referred to as a *queue* (see figure 1). The miser_submit command requires the user to specify the queue to which the job will be submitted, a resource request, and the job invocation. The resource request is a tuple of time and space: the time is the total CPU wall-clock time and the space is the logical number of CPUs and physical memory required. Upon submission, Miser schedules the job using the queue's associated scheduling policy and resources and returns a guaranteed start and completion time for the job, if there are sufficient resources, otherwise the job is rejected.

Once Miser successfully schedules a job, and miser_submit starts it, the job waits in the kernel for resources to be made available so it can run. Prior to the job's start time, it is in *batch* state. A job in *batch* state may run *opportunistically*. That is, it may run on the otherwise idle resources of other pools. Idle batch queue resources are first made available to interactive users and then to batch queues; idle interactive resources are made available to the batch queues. When a job's start time arrives, the kernel transitions the job to the batch-critical state and provides it with the resources it requested from Miser allowing it to run.

Figure 2 shows an example of how Miser schedules a job. The user, in this example, is submitting to the math queue from figure 1 a program called *Tuple-*

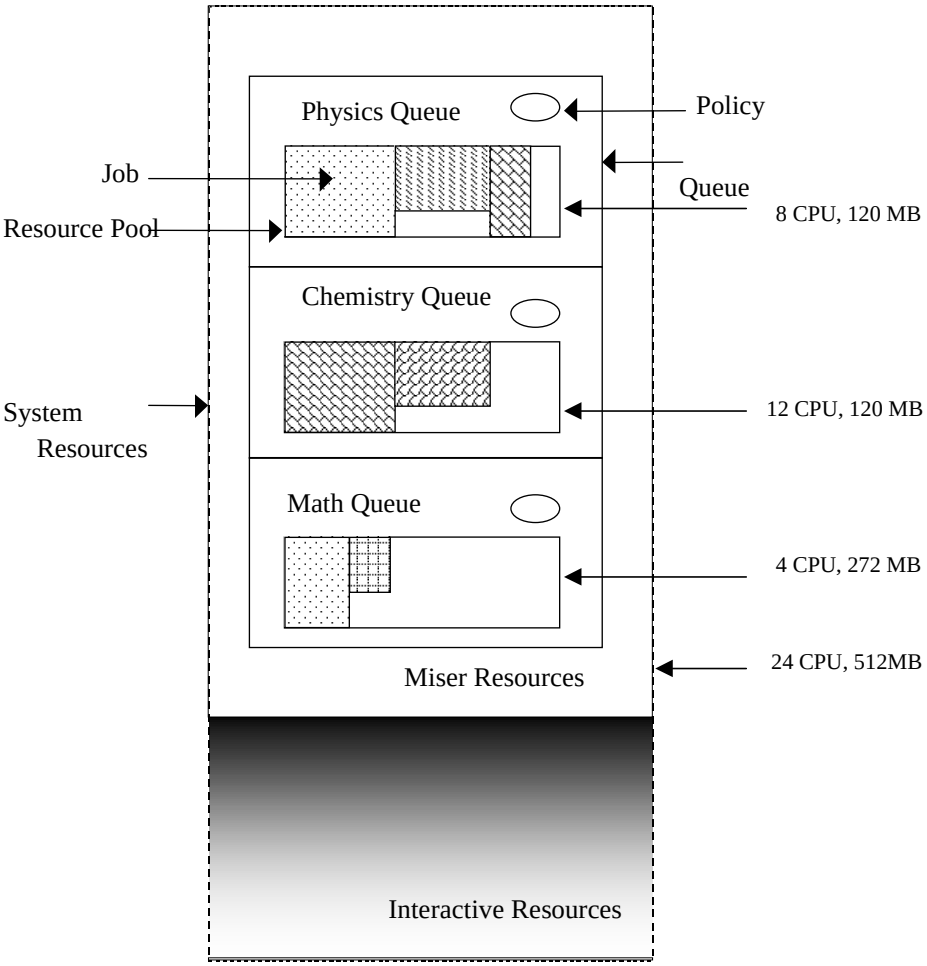

Fig. 1. An example Miser conifguration on a 32 CPU system. Miser is configured with 24 CPUs in its system pool and there are three queues with user pools of eight CPUs each. The reamining eight CPUs are being used by the general system.

Count that requires 4 CPUs, 10 hours of wall clock time and 100 MB of memory. The user specifies the resource requirements, and the program to be scheduled by Miser using miser_submit. Miser upon receipt of the job scheduling request, first tries to find the math queue, verifies that the user has sufficient privileges to schedule the job, and then asks the associated policy of the queue to schedule the request. The policy uses the resource pool associated with the queue to find a hole large enough to fit the job. Once the policy has found the hole, the resources are committed and the start and end times of the job are passed both to the kernel and the miser_submit command that then starts *TupleCount*. *TupleCount*

after being successfully scheduled by Miser now waits in the kernel for resources to be made available to

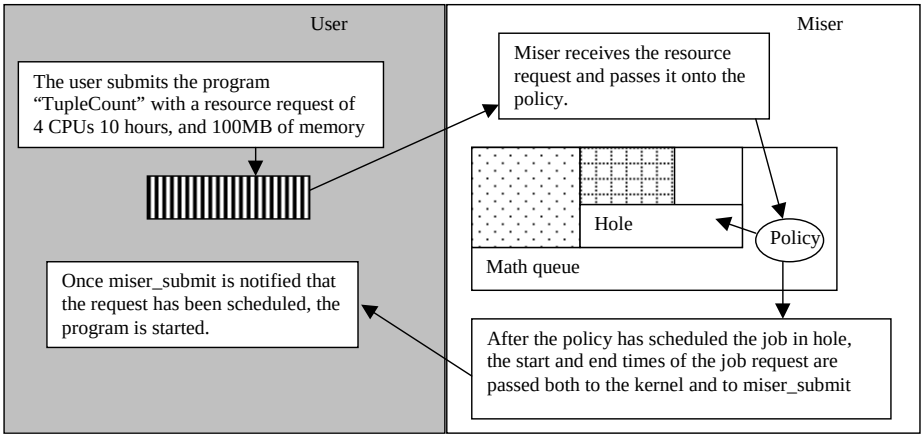

Fig. 2. Scheduling of a job by Miser

While *TupleCount* is waiting in the kernel for its start time to arrive, the kernel may let it run *oppurtunistically* on idle resources. These idle resources can either be part of the system resources that are unused by the interactive portion of the machine, or batch resources that are unused by other queues. So for example, if the physics queue is idle and there is no other interactive load while *TupleCount* is waiting in the kernel, the kernel will let it allocate sufficient resources to begin running. If *TupleCount*, however, tries to use more resources than are idle, then the kernel will suspend it. Similarly, any resources that *TupleCount* uses can be reclaimed, while the job is in the *batch* state to allow a physics or interactive job to run. When *TupleCount's* start time does arrive, however, the kernel transitions the job to the *batch-critical* state, wakes it up if necessary and allows it to allocate resources up to the total requested.

3.3 Kernel - Miser Interaction

The Miser-kernel interaction is limited to Miser providing sufficient information to the kernel such that applications are started and terminated by the kernel according to the schedule determined by Miser with the resources reserved by Miser. As each job is submitted to Miser, its parameters are passed to the kernel. After the job parameters have been passed to the kernel, the daemon does not intervene until the job has terminated. The parameters passed to the kernel do not include the queue information since the queue is a user-level abstraction. If the job terminates early, the kernel notifies Miser, so it can take any action it wants at that time.

4 Kernel Support

While it is the responsibility of the Miser daemon to generate a schedule of jobs that does not over-commit the resources of the machine, it is the responsibility of the kernel to manage system resources such that applications receive the resources requested and thus meet deadlines. Miser currently manages CPU and memory resources. The kernel support consists of a new batch scheduling policy for the IRIX scheduler and modifications to the virtual memory system. The kernel scheduler and batch scheduling policy are responsible for ensuring that jobs transition from *batch* to *batch-critical* state according to the user-supplied schedule, that *batch-critical* applications get the CPUs they requested, and that idle CPUs are made available to either *batch* or interactive applications. The virtual memory subsystem is responsible for providing physical memory to *batch-critical* applications and for ensuring that idle physical memory is available to jobs in the *batch* state and also to nteractive applications.

4.1 The Scheduler

To support the CPU scheduling requirements of the Miser resource manager, namely good interactive behavior, non-interruptible CPU time to particular applications, and dynamic partitioning of CPUs between batch and interactive applications, we implemented a new scheduler. It consists of a fully preemptive, priority based scheduler that has a number of simple abstractions: kernel threads, CPU run queues, a unified priority range and a batch and interactive scheduling policy. The operation of the scheduler is conceptually simple: whenever a CPU needs to make a scheduling decision it looks for work on its local run queue, the run queues of other CPUs, and any queues maintained by the scheduling policies. A scheduling decision is required whenever a higher priority thread becomes runnable, the current thread ran to the end of its time slice, or yielded the CPU. At a scheduling decision the processor examines, the local run queue for any work, and for purposes of load balancing, the local queue of another randomly selected CPU for a thread with a higher priority that can be stolen from the remote local run queue. If there is no work on the either the processors local run queue or on any other local run queue, then the processor, will look at the run queues maintained by the various scheduling policies. The interactive and batch scheduling policies can affect the scheduling decisions of the scheduler by modifying the placement on run queues and the priority of threads. The scheduler architecture enables both guaranteed CPU time and dynamic partitioning of CPU resources. CPU time is guaranteed by boosting the priority of any thread of a process so that it cannot be preempted during its execution. Dynamic partitioning is possible because the scheduler is fully preemptive. A CPU can run any thread when it is idle, because if a higher priority thread suddenly becomes runnable it will immediately preempt a lower priority thread.

The batch scheduling policy has two goals. The first is to ensure that Miser jobs receive the requested CPU time. The second goal is to start jobs early on

idle CPUs. To ensure that an application receives its requested CPU time, the scheduler must accomplish two tasks. The first is guarantee that an application runs for the total requested time without interruption. This is accomplished by raising the priority of the Miser job when it becomes *batch-critical* to a value higher than that of any interactive job. To guarantee that no two jobs in the *batch-critical* state time-slice with one another, the CPUs are partitioned between distinct jobs, and the threads of a particular job only run on the job's partition. The second task is to ensure that an application does not use any more CPU resources than requested, either by running on more CPUs than requested, or by over-running its time because that will prevent other applications from receiving their allotted time. This is accomplished by restricting the set of CPUs that an application can run on while it is *batch-critical* and by terminating the application should it over-run its time. The second goal of the batch scheduler is accomplished by maintaining a private queue of jobs in the *batch* state and using it to generate work for idle CPUs. These threads run on CPUs until they are pre-empted by time-share threads or *batch-critical* threads.

Figure 3 illustrates how dynamic partitioning takes place between batch and interactive threads. Initially, all the threads of the batch job are suspended and the only active threads are interactive threads (figure 3.a). When the first batch thread becomes active because the job has become *batch-critical*, it is placed on the run queue of CPU 4 (figure 3.b). At the next scheduling decision, the CPU selects the batch thread to run since it has a higher priority and the interactive thread ends up on another run queue as a result of load balancing (figure 3.c). Later, the second batch thread becomes active, and the interactive thread on CPU 1 is pre-empted so that the batch thread can run (figure 3.d). As a result of these preemption the CPUs are now partitioned evenly between batch and interactive jobs (figure 3.e). At some point in the future, the batch job exits, and CPUs 1 and 4 go idle (figure 3.f). At that point the CPUs go looking for work, and begin to run the interactive threads that are queued on CPU 2 and 3 (figure 3.g).

4.2 Virtual Memory Subsystem

To support the notion of guaranteed physical memory to a particular class of applications related by scheduling discipline without wasting unused physical memory, a new accounting mechanism was added to the virtual memory subsystem. The VM did not require major changes because it was already capable of guaranteeing physical memory to particular applications. What it was not capable of doing is account for memory across a set of applications related only by scheduling discipline. The new accounting data structure used is called a memory pool. There are currently only two pools, a Miser pool and a general pool. The Miser pool is used to keep track of the total amount of memory available for use by applications in the *batch-critical* state. The general pool is used to keep track of the total amount of physical memory and swap, also called logical swap, available to the rest of the system. By modifying the size of the general pool it is possible to reserve enough memory for the Miser pool. Since no actual

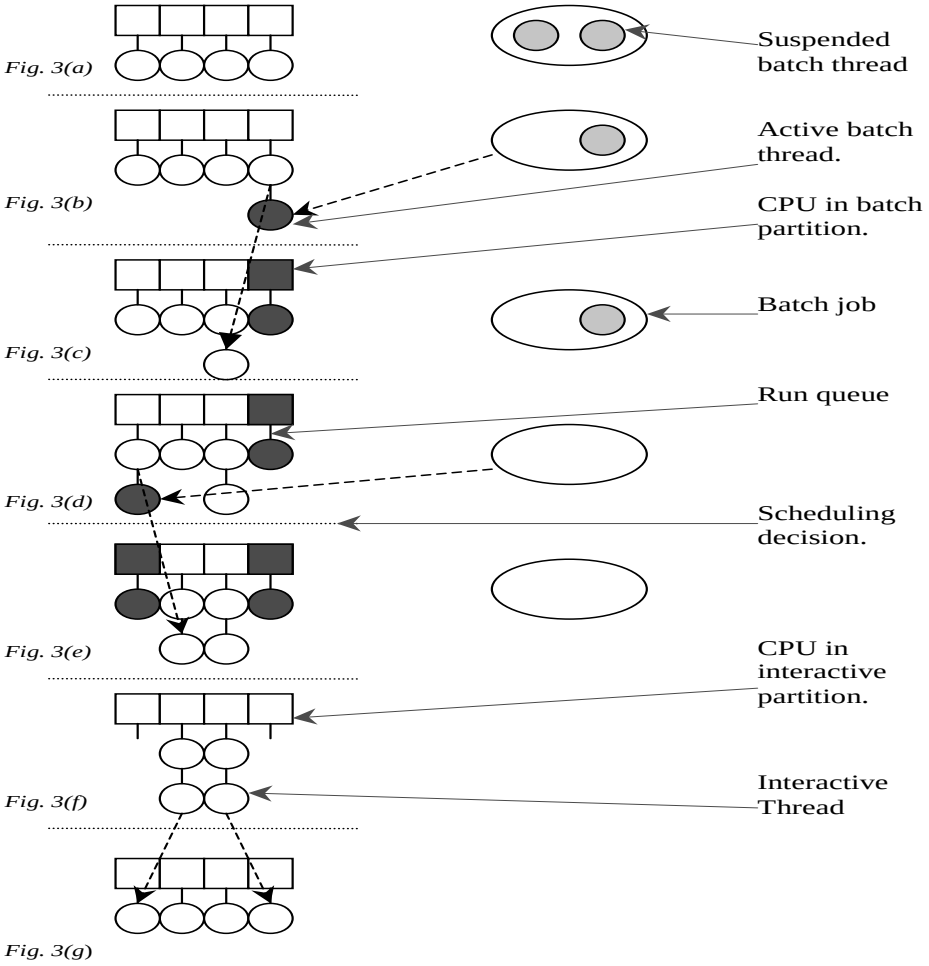

Fig. 3. *Example of Dynamic Partitioning of CPUs.* As the number of batch threads changes so does the number of the CPUs in the batch partition. The changes take effect after every scheduling decision

physical memory is reserved, any job using the general pool can use physical memory equal in size to the logical swap.

Batch applications when running opportunistically use the global pool and then transition to the Miser pool. To prevent batch jobs from failing a system call or memory allocation because of insufficient memory in the global pool, the job is suspended until it becomes *batch-critical* and the operation is restarted.

In figure 4 we show an example of how both the memory accounting and memory usage varies on a batch system where there is a batch and interactive job running. The boxes represent each of the resources, the Miser and global memory pools as well as the physical and swap memory. The shaded regions of

the boxes represent how much of a particular resource a particular job has at any point in time. The shading color indicates the job type.

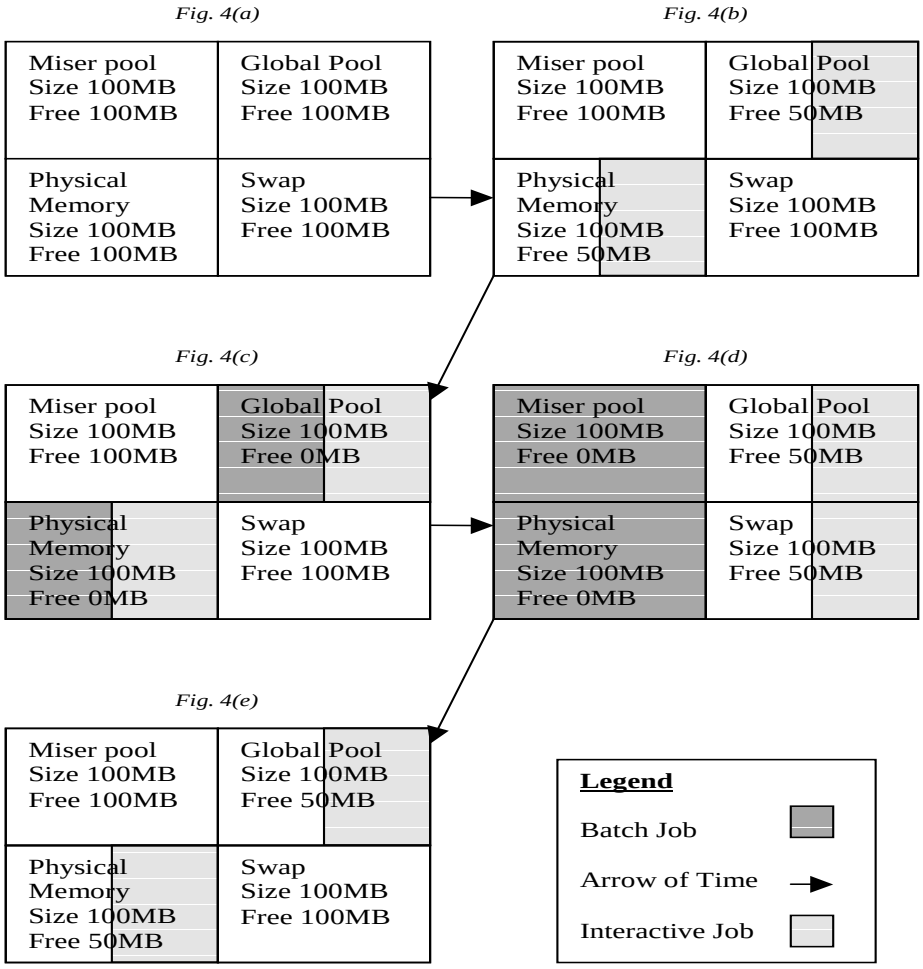

Fig. 4.

The machine in figure 4 is initially configured with 100MB of memory in the global pool, and 100 MB in the Miser pool. The system also has 100 MB of swap space and 100MB of physical memory (figure 4.a). The interactive job is the first job to start and requires 50MB of memory, so it is allocated 50MB from the global pool (figure 4.b). Since there is no other workload on the system, the interactive program is able to acquire 50MB physical memory. Later a batch job is started that has requested 100MB of memory (figure 4.c). There are idle resources on the system, so the batch application begins to run opportunistically, using up the

remaining 50 MB of the global pool. Since there is no other work on the system, the batch job is also able to use the remaining 50MB of physical memory, and now all physical memory is being used by applications. The batch application had requested 100MB, but there is no more memory in the global pool, so it is suspended. At its start time, the batch job is transitioned to the *batch-critical* state, and the job now claims 100MB from the Miser pool, and releases the 50MB from the global pool (figure 4.d). Now the global pool has 50MB free, and the Miser pool 0. The batch application, since it has preference over physical memory, forces the interactive application to be swapped out because there is no other free physical memory. The usage of the physical and swap memory now becomes 100MB for the batch job and 50MB for the interactive job respectively. When the batch job finally terminates, the Miser pool grows to 100MB, and the free physical memory is used once again by interactive program (figure 4.e). At this point there is 50MB of physical memory free and 50MB free in the global pool.

5 Empirical Evidence

The goals of these experiments are to demonstrate that there is no performance cost and that the overall throughput of the system improves as a result of using Miser. The scheduling policy used by the Miser queue was first fit, the default

Our experiments were conducted on a 16 processor Origin 2000 using cg, bt, and ep from the NAS benchmarks[1]. We ran the benchmarks repeatedly and reported either individual performance numbers or the number of times a benchmark was able to complete over a period of time. The load was generated using a CPU cycle burner and the load number represents the number of cycle burners used.

5.1 Experiments

The first experiment shows the performance of cg, bt, and ep with different amounts of load (figure 5). For each benchmark, we ran the benchmark with varying degrees of load both using and not using Miser. When the load was 0, the performance of the application under Miser and not under Miser was identical, indicating that there is no performance penalty for using Miser. As the load increases, however, the applications not scheduled by Miser see degradation in performance. Applications scheduled by Miser, however, do not. This demonstrates that Miser is able to guarantee deterministic run times for applications even with large interactive load.

The next two experiments show that Miser can share idle resources between the batch and interactive portions of the machine. To measure this, we measured the throughput of the machine by simultaneously starting 5 copies of each

[1] The performance of the benchmarks cannot be considered official SGI benchmark values.

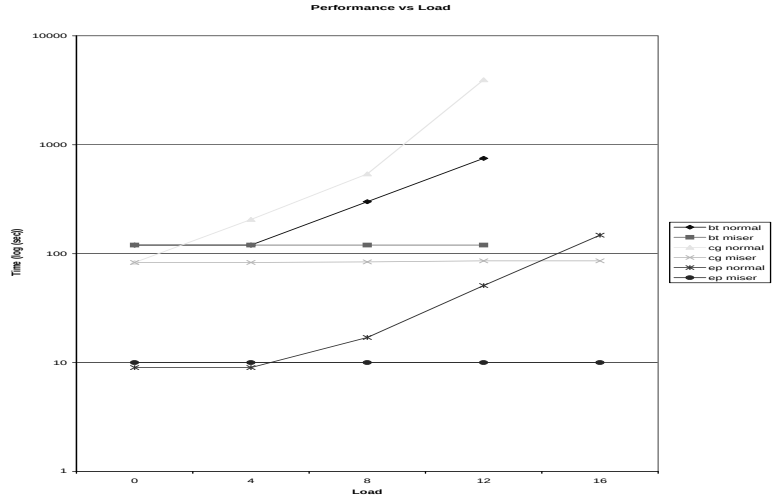

Fig. 5.

benchmark and recording both the total wall clock times for all five benchmarks to run, and the average latency. The benchmarks in all cases were run using eight threads. Miser was configured to manage half of the total system resources.

The second experiment (see figure 6) measured the latency of applications on a 16 CPU machine with Miser configured, but not using it and compares it to the performance of the benchmarks on an 8 CPU machine. For each benchmark, note that the latency improved dramatically, when compared to the performance of the benchmark on the smaller machine. The results show that resources reserved by Miser can be used by interactive applications. A 16 CPU machine configured to use Miser will result in better average latencies for interactive applications than two single 8 CPU machines reserved for batch and interactive users if the batch portions of the machine are idle.

The third experiment (see figure 7) examines the difference in overall throughput in a batch-scheduling environment. The total time taken for the jobs to complete their runs was measured when scheduled by a simulated batch scheduler, by Miser but with the jobs not taking advantage of the idle resources, and finally with Miser and with jobs taking advantage of the idle resources. To simulate a batch environment, applications were started sequentially until the total resources of the machine were consumed. On the 16 CPU system two 8-way threaded copies of each application ran at the same time. First note that the performance of batch applications using Miser when using idle resources is better than the performance of the applications when Miser does not take advantage of

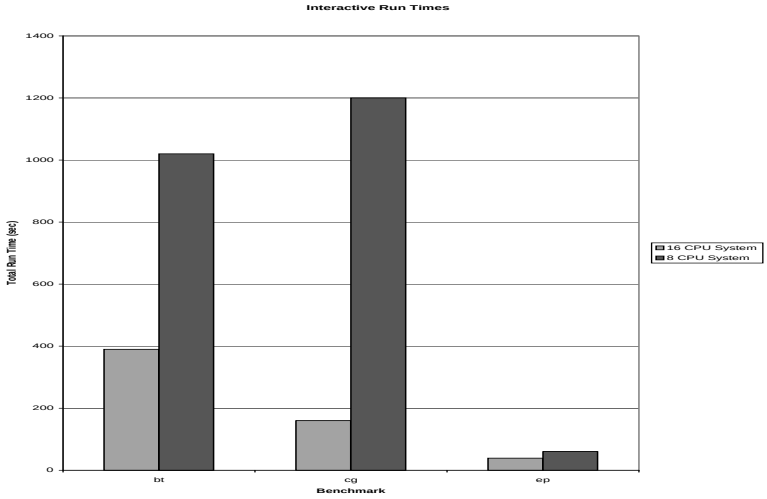

Fig. 6.

idle resources. This shows that Miser can indeed take advantage of idle resources to improve total throughput. The second thing to observe is that performance of applications using Miser is better than that of the simulated batch scheduler when the applications scheduled by Miser can use the idle resources.

Jobs scheduled by Miser perform better than the simulated batch scheduler because of the way Miser schedules applications. The individual threads are always run together, the threads are never preempted, and the threads of distinct applications do not interfere with each other. This results in two benefits. The first is that applications make better use of the memory system. The second is that application threads are more efficiently co-scheduled. Note, however, that the improvement depends on how sensitive the applications are to co-scheduling. The amount of sensitivity depends on how frequently the application performs busy-wait synchronization. Ep that does no busy wait synchronization showed no improvement and cg that does the most showed the most with bt being in the middle

6 Future Work

Miser does not fully exploit the NUMA properties of the underlying architecture very well. Although Miser is able to reserve total memory well, in order to achieve optimal performance on NUMA systems, Miser will need to reserve memory on specific nodes, and also reserve particular topologies. In this case Miser would

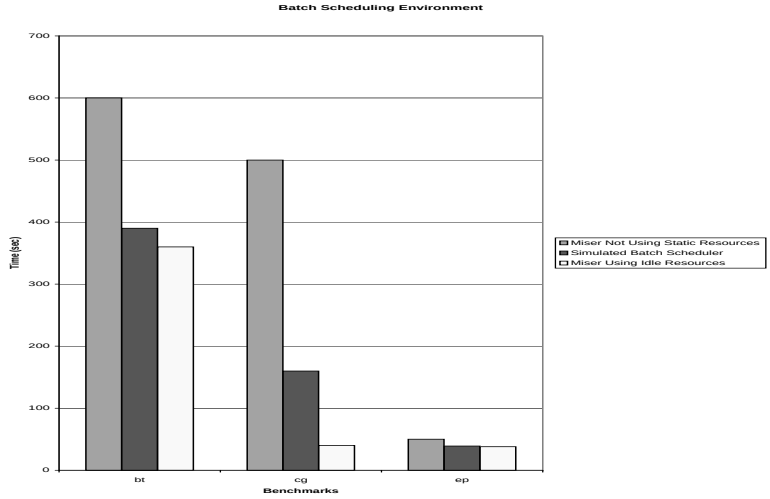

Fig. 7.

have a fundamental advantage over normal kernel mechanisms because Miser jobs have known run times.

Miser currently requires that applications not use more than the available physical memory. If the application requires more, Miser can not schedule the application. A better solution would be to allow batch jobs to self-page. In this model, the batch application still has a reservation of memory, but it also has a reservation of swap. It can thus swap out portions of its physical memory to disk, so as to have a working size that is potentially much larger than the physical memory size.

Miser was originally envisioned as a general-purpose resource management facility for scheduling applications that required particular physical resources to run. We hope to extend Miser to other classes of applications such as real-time, and to manage more resources such as disk. The problem with real-time on IRIX, is that the configuration of a real-time application requires a multi-step process that is error prone. Using Miser, it would be possible to configure a particular queue that had the desired real-time attributes, and to start a real-time application by simply submitting the application to Miser. Miser would then take the necessary kernel level actions to guarantee the requirements of the real-time application rather than leave it to the application writer. This approach not only is less error prone, it allows a real-time system to be shared by different users, in a way that prevents them from interfering with each other.

Finally, although there is support for user-defined policies, the mechanisms have not yet been fully defined. We hope to define and export interfaces that would allow users to provide scheduling policies.

6.1 Conclusions

Theoretical results have shown that 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 providing kernel support. Using the underlying kernel scheduler we are able to guarantee CPU time and memory 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 in order to achieve guaranteed performance, and thus achieve better throughput as we have demonstrated. Miser is a new mechanism for scheduling batch jobs with deterministic deadlines without the inherent waste of resources that result from static partitioning.

References

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

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

[CPUSET] man Pages (1M): User Commands, IRIX 6.5, Silicon Graphics 1998

[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

[LSF] LSF Batch Scheduler Users Guide, Platform Computing documentation. Available on line at http://www.platform.com