Preemption Based Backfill

Quinn O Snell, Mark J Clement
Fulton Supercomputing Center
Brigham Young University
[snell,clement]@cs.byu.edu

David B Jackson
Center for HPC Cluster Resource Management,
Supercluster.org

Abstract

Recent advances in DNA analysis, global climate modeling and computational fluid dynamics have increased the demand for supercomputing resources. Through increasing the efficiency and throughput of existing supercomputing centers, additional computational power can be provided for these applications. Backfill has been shown to increase the efficiency of supercomputer schedulers for large, homogenous machines[3]. Utilizations can still be as low as 60% for machines with heterogeneous resources and strict administrative requirements. Preemption based backfill allows the scheduler to be more aggressive in filling up the schedule for a supercomputer[2]. Utilization can be increased and administrative requirements relaxed if it is possible to preempt a running job to allow a higher priority task to run.

1 Introduction

The last few years have witnessed an explosion in supercomputing applications. Biologists, who have traditionally performed most of their analysis by hand, are now able to analyze the complete human genome. Although this information promises to revolutionize the way research is performed in several disciplines, the corresponding computational requirements are increasing at a rate that surpasses funding increases for supercomputing resources.

The supercomputing infrastructure landscape is also changing rapidly in response to the demand for computational resources. Many researchers are creating clusters of workstations to run computationally intense workloads. These clusters often grow at an incremental rate with additional machines being added to the cluster as new funding for research is received. Additions to existing clusters often have different memory capabilities and may have licenses that allow software to run on only a subset of the processors in the cluster. These additions invariably cause the cluster to become more heterogeneous. This complicates the job of a scheduler since tasks that require a certain set of resources may only be schedulable on a small subset of the nodes in the system.

Another cause of heterogeneity arises due to administrative policies. As new infrastructure is purchased by a particular organization, the organization may impose scheduling policies intended to insure that their community has priority access to the resources. Other users are normally welcome to use the nodes if they are available, but should not delay the execution of users in the community. These administrative policies cause a normal scheduler to be overly conservative in scheduling low priority jobs and computational resources will often be wasted in order to insure the priority of community tasks.

1.1 Backfill

Backfill is performed when the highest priority job requires more resources than are currently available on the supercomputer. A lower priority task that is guaranteed to complete before the anticipated initiation time of a high priority job is allowed to run. Since these resources are not usable by the highest priority job, throughput and efficiency are increased on the machine. In some cases, there will be no lower priority task that will fit in the idle time slot. In this case, processing time will be wasted in order to guarantee that the high priority task will run as soon as possible.

Preemptive backfill allows the scheduler to start lower priority jobs even if they are not assured of completion before higher priority users require the resource. In the majority of cases, the low priority task will finish and have no impact on the high priority schedule. If the low priority job takes longer than the available idle slot, it will be preempted by suspending it or killing it. This research addresses the following questions with respect to preemptive backfill:

- *Which jobs are the best candidates for preemptive backfill?* Users normally provide an estimate of execution time when they submit a job. If this time limit is exceeded, the job is killed. As a result, users often submit estimates that are far greater than the real time used by a task. By looking at the accuracy of past estimates, the scheduler is able to select tasks for preemptive backfill that have the highest probability of completion in the allotted time.
- *Which preemption policies provide the best utilization and throughput on the machine?* Many supercomputer operating systems do not provide the ability to suspend a parallel task. This is due to the fact that all threads and spawned processes must be suspended along with the main task. The operating system must also implement security measures so that a running task does not have access to data and files used by suspended jobs. When a job must be preempted, all of the progress that the job has made toward completion will be lost. This is the main downside of preemption-based backfill. In order to minimize this negative effect, appropriate policies must be adopted. Possible policies for determining which jobs to preempt include:
    - Most Recently Started
    - Furthest From Wall Clock Limit
    - Furthest From Historical Information Scaled Wall Clock Limit
    - Minimum Work Completed

Supercomputer schedulers that do not perform backfill are typically limited to significantly lower utilization and throughput than classical backfill systems, but priority jobs are guaranteed to run as soon as resources are available. Classical backfill improves the efficiency of the machine by filling idle time with lower priority tasks, but without preemption, high priority jobs may be delayed. Preemption based backfill provides for high utilization, while guaranteeing that prioritization is preserved.

2 Background

Several different backfill strategies have been employed in the past to improve utilization in supercomputing systems. This section outlines important concepts and existing strategies for improving supercomputer utilization.

When a user submits a job, several attributes are included with the submission. These attributes include:

- Resources required– This includes memory, network and software licenses.
- Number of Processors – The job will not be scheduled until this number of processors is available. These processors may be scheduled in a contiguous block, or may be distributed across the machine. Some submission systems may allow the user to specify that processors should be allocated in

groups that share the same memory.

- Wallclock Time Limit – After the job has been running for this amount of time, the job will be killed. Users will often overestimate this parameter. The IBM SP2 at the Cornell Theory Center was examined in prior research. Of the jobs submitted, 38% used less than 4% of their wall clock limit. Less than 5% of the jobs used 40% of the wall clock limit [4].

2.1 Non-backfill

All of the jobs that are submitted will be accounted for in a scheduling queue. The scheduler prioritizes jobs in the queue according to several different policies. Jobs that are associated with certain users and projects may be given higher priority. As jobs wait in the queue, their priority will normally increase so that indefinite postponement is avoided. When a job completes, the scheduler will attempt to start the highest priority job in the queue. If resources are not available for this job, processors may remain idle waiting for another job to complete that will allow the highest priority job to run. Figure 1 shows this wasted processing power with Non-backfill scheduling.

2.2 Classical Backfill

With classical backfill, the scheduler may attempt to backfill jobs with lower priority into this wasted processor space. If the backfilled jobs can complete before Job 1 completes, then Job 2 will not be delayed. Classical backfill will only select jobs with a wallclock time estimate such that the backfilled job will complete before the end time of Job 1. There may be many jobs that will actually complete in this time, but none of them will be scheduled into this time if the user estimate is longer than the available space. This conservative backfill policy causes lower utilization. If preemption is allowed, the scheduler can run jobs that have a high priority of completing before Job 1 completes. If Job 1 completes early, the scheduler can preempt these backfilled jobs and run Job 2, increasing the overall efficiency of the machine.

Even when no jobs are backfilled that have wallclock estimates greater than available time, backfill can cause high priority jobs to be delayed longer than they would have been on a non-backfill system. Figure 2 shows a Classical backfill schedule with wallclock estimates and actual runtimes. When a High priority job completes early, jobs that have been backfilled may prevent the next high priority job from running. This increases the expansion factor for these jobs. Expansion factor is defined as the total time from submission to completion, divided by the actual runtime of a job. Administrators can disallow backfilled jobs on their resources in order to eliminate this expansion in the runtime for their high priority jobs.

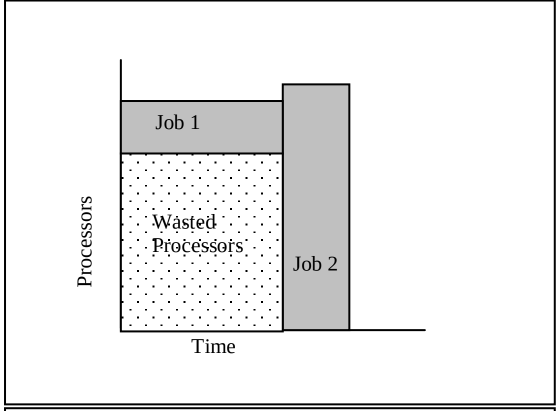

Figure 1 Non-backfill Scheduling

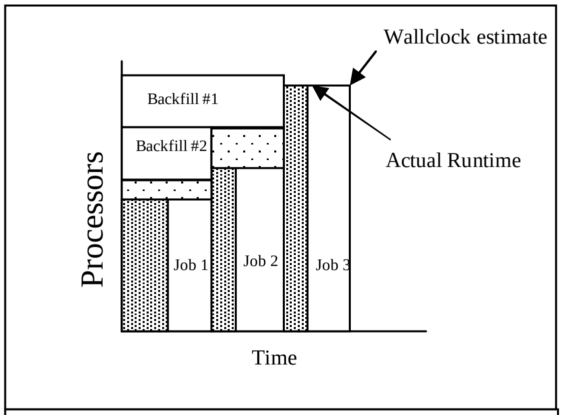

Figure 2 Classical Backfill

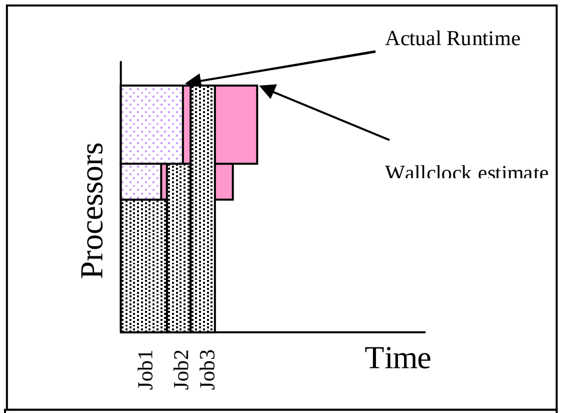

Figure 3 Preemptive Backfill

2.3 Preemptive Backfill

The wasted processors in Figure 2 are present because there were no jobs in the queue that matched the available resources. Preemptive backfill will run jobs in these slots that have a high probability of completion in the available time, even if their wallclock estimate exceeds the backfill space. Figure 3 shows the same set of tasks with preemptive backfill. The expansion factor for high priority jobs is minimized because backfilled jobs are preempted when resources become available for the next high priority job. Efficiency is higher because backfill slots can be filled, even if no job exists in the queue with small enough resource requirements.

2.4 Gang Scheduling

Gang Scheduling provides some of the same advantages as preemptive backfill. With Gang Scheduling, all of the processes associated with a task are suspended periodically so that more than one task can be time sliced on the same set of processors. A lower priority job can be backfilled into a slot using gang scheduling and if resources become available to let a higher priority job run, the backfilled job can be preempted by the higher priority job and may only run when the high priority job completes.

Several problems arise with gang scheduling.

- Migration - It may be better to run the backfilled job in some other processor space than to wait for the high priority task to complete. Some Gang Scheduling systems allow jobs to migrate to processors where they can complete more quickly, but since data files and memory must be migrated with the process, migration can lead to inefficiency in supercomputing systems [6].
- Memory Contention – Many supercomputing applications require large amounts of memory. When time slicing occurs, the caches and memory may have to be swapped out to disk, causing significant performance degradation [1].
- Security – As mentioned previously, each job must be isolated from the file space and environment of other tasks. This is difficult to implement when more than one job is being time-sliced on a machine. This occurs because in a meta-scheduling environment, each user may not have a separate account.
- Availability – Gang scheduling is not available on many operating systems. Even when it is available, preemptive backfill may provide higher performance for many applications.

3 Preemptive Backfill

This research was performed using the Maui Scheduler [5,7]. The Maui Scheduler is used in many supercomputing sites throughout the world and provides for a simulation mode as well as production mode. Extensive trace data is available from production runs of large jobs over a long period of time. This trace data is used to evaluate different options for preemptive scheduling.

3.1 Preemption

Several forms of preemption are possible. The form used is dependent on operating system functionality. For this research we assume the minimal functionality of being able to kill a job and all of its spawned processes. Several other forms may be available depending on the operating system used:

- *stop/restart* – All supercomputer operating systems provide functionality to stop a parallel job and to start it over again at a later time. All preliminary results are lost and the task must be able to remove any changes that have been made to input files so that the second invocation of the task will have exactly the same environment as the first. Once the job has been stopped, it will return to the queue and backfilled onto another set of processors or run as a priority job when it reaches the head of the queue. Any partial results will be lost and any time spent running a job that is stopped is wasted.
- *suspend/resume* – Many operating systems provide functionality that allows a task to be suspended and then resumed at a later time. All of the processes associated with this task are made non-runnable, but they retain process state and any file system changes remain in place so that the job can continue when it is resumed. The operating system must insure that jobs do not have access to each others files and memory will need to be transferred to a swap file before the new task can start. This may cause some additional delay in the start time of the high priority task when compared to *stop/restart*. When the job is resumed, it must run on the same set of processors that is was suspended on. This may cause a large expansion factor for jobs that are backfilled in this way.
- *checkpoint/restart* – Many applications perform checkpoint operations to save their intermediate state. Once a checkpoint is performed, the job can be terminated and restarted with the same state present when the checkpoint was performed. This preemption form has an advantage over *suspend/resume* since the task can be migrated to another set of processors for continued execution after preemption. It also has an advantage over *stop/restart* since intermediate computations performed before preemption are not wasted. The principle disadvantage of this option lies in the fact that in most operating systems, the application itself must perform checkpoints. Most operating systems do not support checkpointing because it is difficult for an operating system to determine which parts of the memory space should be saved in order to restart effectively.

Many parallel processing systems do not have support for suspend/resume or checkpoint/restart. The results presented in this paper are all based on stop/restart program semantics. If additional functionality is available, preemption becomes even more advantageous in scheduling resources.

3.2 Backfill Options

Several factors must be considered in determining the implementation of a preemptive backfill scheduler. The user and system administrator have different goals in mind, and these goals must all be addressed in order to provide optimal service. Three distinct goals must be considered:

- Priority scheduling Goal: The scheduler should run the most important job (based on value of job) first. In many cases one organization will own computing resources, but will be willing to have other jobs serviced if the resources are idle. When a high priority job is submitted, it should be given preferential treatment, or the owner of the resource will not be willing to allow lower priority jobs to run at all.

- Backfill scheduling Goal: The scheduler should run the greatest aggregate sum of jobs with the greatest likelihood of successful completion. (based on the value of the job, probability of successful completion). This goal translates into high utilization from the system administrator's perspective. Possible priorities for this goal include:
    - Backfill jobs using the most resources first
    - Backfill the jobs that are most likely to complete first
    - Backfill highest priority jobs first. High priority jobs may not be candidates for backfill since they may be preempted if an existing job finishes early, or the predicted completion time is inaccurate.
- Job Preemption Goal: The scheduler should stop the minimum set of jobs required to allow priority jobs to run immediately (based on the value of jobs, probability of successful completion and the value of the completed work).

Experiments performed in this research indicate that preemptive backfill provides lower delay for users and higher utilization for system administrators. By giving more weight to each of the backfill goals, the scheduling algorithm can favor the goal that is most important to the user community.

4 Experimental Results

Several Experiments were performed to determine the impact of preemptive backfill on system performance. These experiments were performed on the Maui Scheduler[5] with job trace files from the Center for High Performance Computing at the University of Utah. The trace files included 11,445 jobs ranging from 1 to 32 processors, lasting up to 68 hours. Figure 4 shows the job mix present in the trace. There are a large number of jobs requiring less than 8 processors, providing a large opportunity for backfill. Figure 5 shows the percentage of processor hours used in each job classification. Although there are a large number of jobs with small numbers of processors and a short duration, they account for a small percentage of the overall runtime on the machine. Jobs with duration of more than 17 hours and more than 8 processors dominate in this area. Any preemptive backfill scheme should not hurt the performance of these jobs.

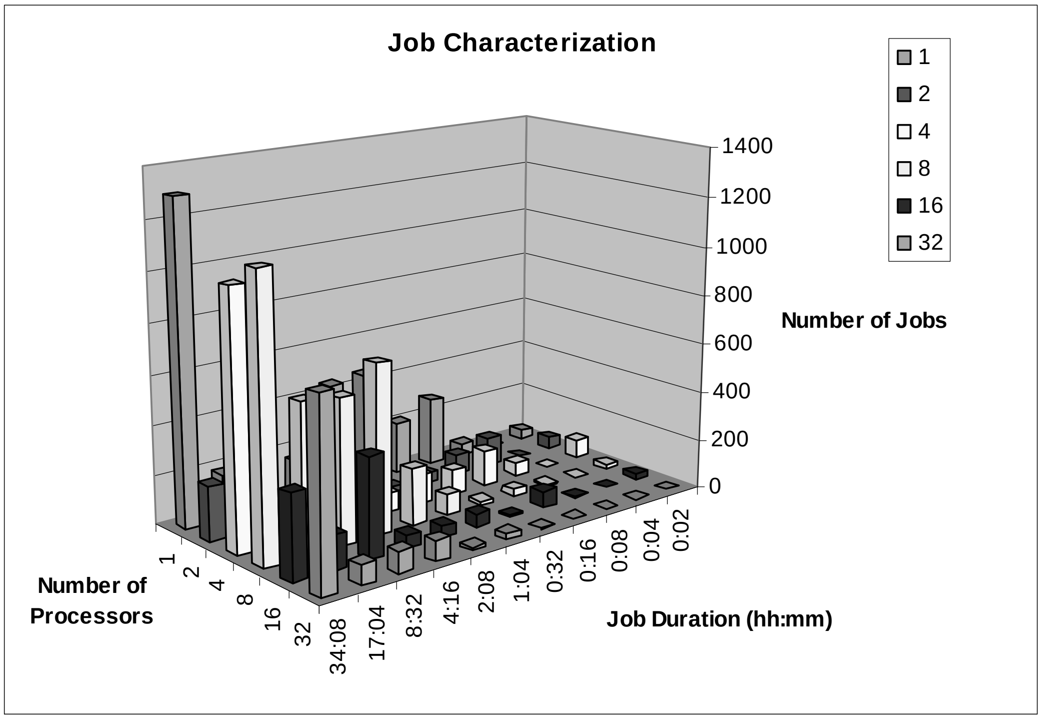

Figure 4: Job Characterization of trace data. Many of the jobs are in the 8 processor range and can be backfilled effectively.

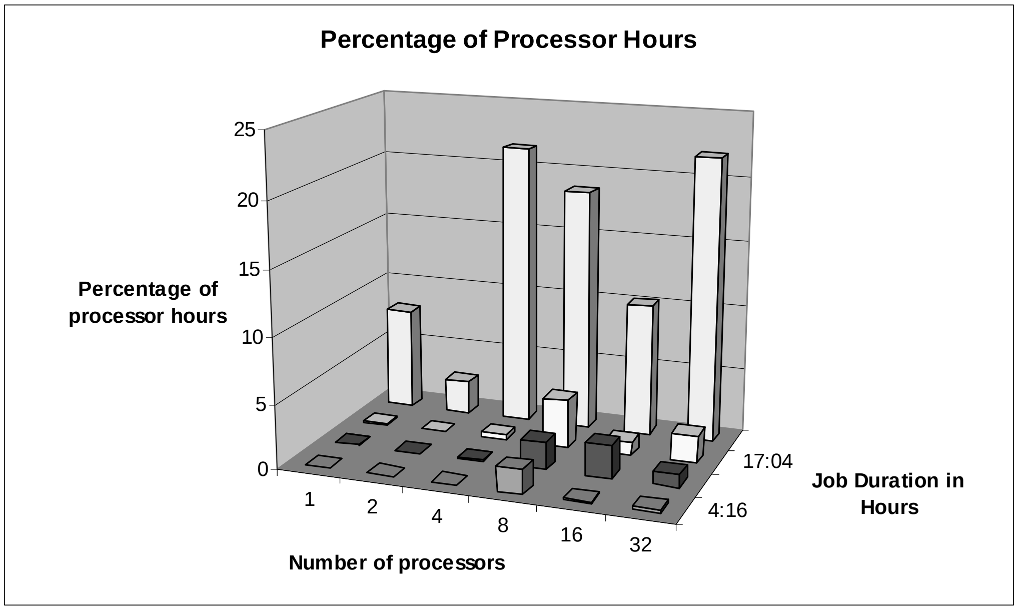

Figure 5: Distribution of job processor hours. Most of the time in the trace is spent with jobs that take more than 17 hours and have more than 8 processors.

4.1 Non-prioritized Schedule Analysis

Several experiments were run to determine the impact of preemption on the performance of jobs in the trace when no priority was used. The scheduling algorithm was changed so that jobs could be backfilled even if their wallclock estimate of completion time was after the start time of the next normal job. When a scheduling iteration occurs and backfilled jobs are preventing normal jobs from running, a set of the backfilled jobs will be preempted or terminated (under the current model where restart is not available.

Metrics used in evaluating the algorithms were Queue Time and Expansion factor. An effective preemptive backfill algorithm will decrease idle time for processors since jobs with a wallclock estimate precluding them from backfill under normal circumstances can be scheduled if they can be preempted when a higher priority job must be run. It is expected that queue time will be reduced with preemptive backfill. Expansion factor considers the percentage of additional time that a job has to wait because of queuing delay (Xfactor=[QueueTime+RunTime]/RunTime). A job with a 5 minute run time that is queued for 100 minutes will notice the delay much more than a job with a 1000 minute run time and the same delay. The expansion factor for the 5 minute job would be 21, where the expansion factor for the 1000 minute job would be 1.1. Optimal expansion factors are close to 1.

Figure 6 shows expansion factor results for several algorithms that were used to select which jobs to preempt. In one case a *random* set of jobs were selected to preempt. This will often result in a job being terminated when it is nearly complete. The random plot is presented to show the worst case behavior of a naïve algorithm. The *First Fit* plot shows the behavior of the normal backfill algorithm without preemption. Notice that the random algorithm behavior has significantly higher Expansion factor and queue time than the non-preemption firstfit algorithm.

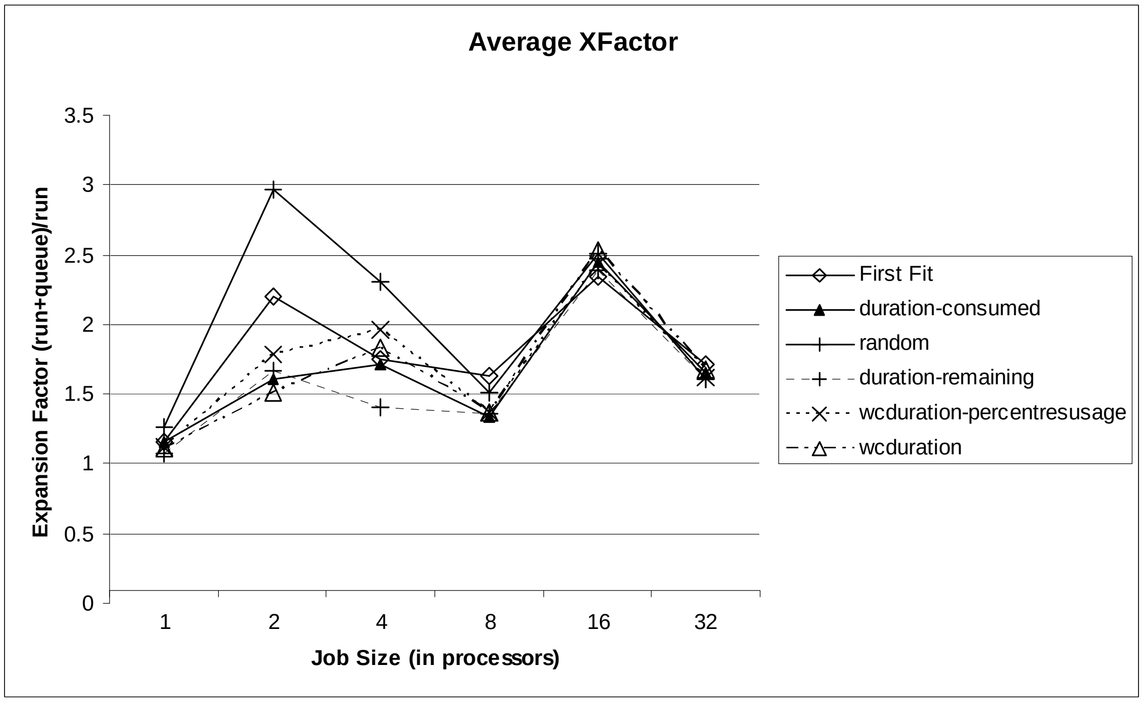

Figure 6: Expansion factor for several preemption strategies. Intelligent preemption backfill strategies outperform random strategies and in most cases also perform better than First Fit without preemption.

The *duration-consumed* algorithm selects a set of jobs to preempt that have been running for the shortest length of time. This strategy attempts to waste the minimum number of node-hours in the preemption. The *duration-remaining* plot results from preempting the jobs that have the most time left to run. By preempting these jobs, the scheduler will free up larger slots of time for each job preempted, and may be able to preempt a smaller number of jobs.

Each job has a Wall Clock prediction of execution time that is an estimate provided by the user. When this Wall Clock time expires, the job will be terminated. As a result, users tend to overestimate the time for their jobs, but the estimates can still provide information as to which jobs to preempt. The *wcduration* plot represents results from preempting the job with the longest estimate of duration and the *wcduration-percentresusage* plot shows the results of preempting the job using the largest percentage of resources on the machine.

The strategies resulting in the best expansion factor are *duration-remaining* and *duration-consumed*. These strategies consistently outperform *First Fit*. Similar results can be observed in queue time analysis.

Figure 7 shows the average queue times for each of the strategies. The average queue time for all of the preemption strategies is less than the *First Fit* non-preemption scheduling algorithm. It is difficult to select a clear winner, but *duration-remaining* and *duration-consumed both achieve good performance.*

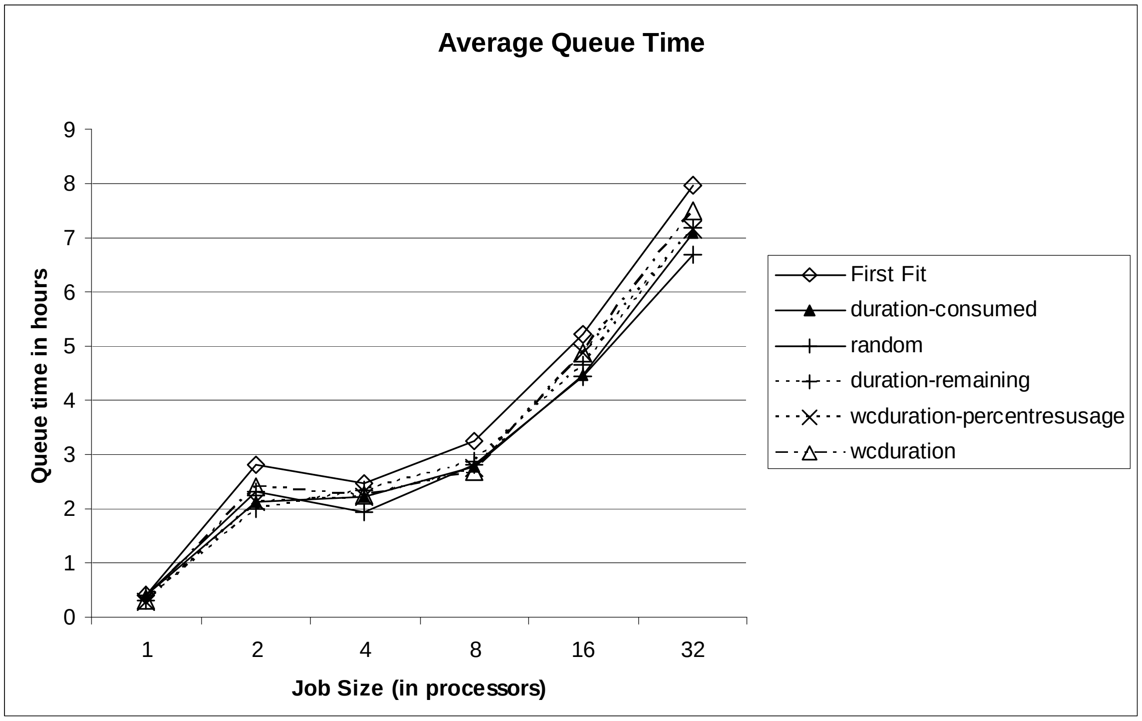

Figure 7: Average queue time for each of the preemption strategies. All of the strategies achieve lower queue times than First Fit due to a reduction in wasted node-hours since slots can be filed with jobs that wouldn't normally be backfilled due to their estimated run time.

4.2 Prioritized Schedule Analysis

Although non-prioritized jobs seem to benefit from preemption, one of the goals of preemption is to insure that high priority jobs receive preferential treatment. If the owner of a supercomputer can be assured that his jobs will always preempt jobs of other users, he will be more willing to allow low priority jobs to utilize idle node-hours when he doesn't have an active jobs. In our experiments high priority jobs can always preempt low priority jobs. High priority jobs can preempt backfilled medium priority jobs.

Figure 8 shows expansion factor when 80% of the jobs were marked as medium priority and 20% high priority. The high priority jobs achieve an expansion factor near unity, indicating that they have a queue time near zero. The expansion time is greater than one for the 32 processor case since high priority jobs only preempt medium priority jobs when they are backfilled. Medium priority jobs that are scheduled normally will not be preempted. Figure 9 shows the queue times for this case. Medium priority jobs achieve queue times that are very similar to First Fit.

Similar results were obtained for a combination of high and low priority jobs. Figure 10 shows the expansion factor with a mix of high priority and low priority jobs. Since preemption is possible, the high priority jobs are only queued behind other high priority jobs. When a low priority job is running and a high priority job arrives, the low priority job is preempted immediately. Figure 11 shows the queue time for this configuration. In this case low priority jobs with 32 processors received a higher average queue time than First Fit, but un most cases the queue time was similar to First Fit even for low priority jobs.

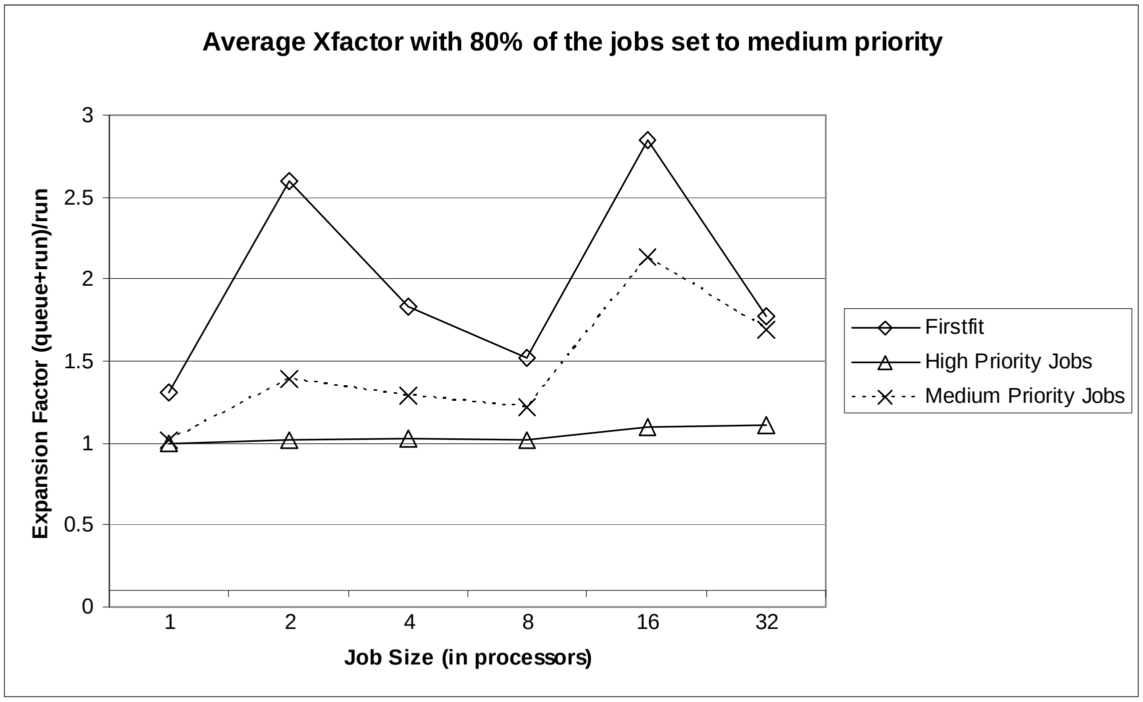

Figure 8: Expansion factor for a mix of high and medium priority jobs. High priority jobs are able to achieve near-optimal performance even though medium priority jobs are present.

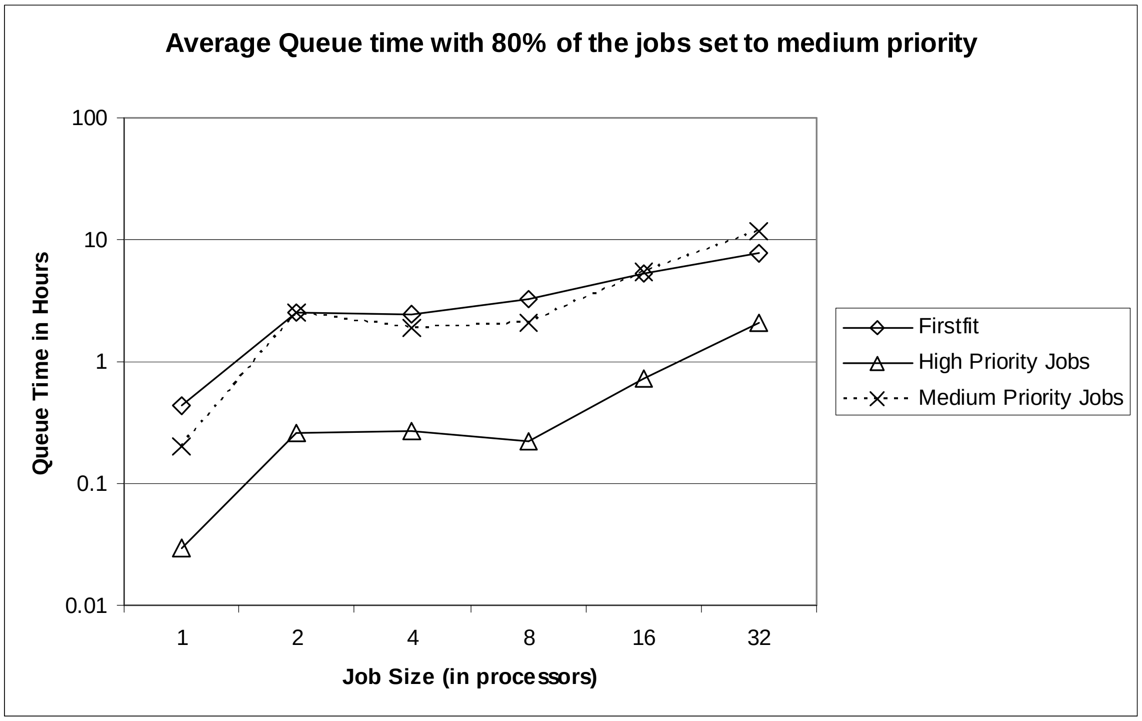

Figure 9: Queue time for a mix of high and medium priority jobs. High priority jobs are able to achieve low queue times and medium priority jobs have queue times similar to First Fit.

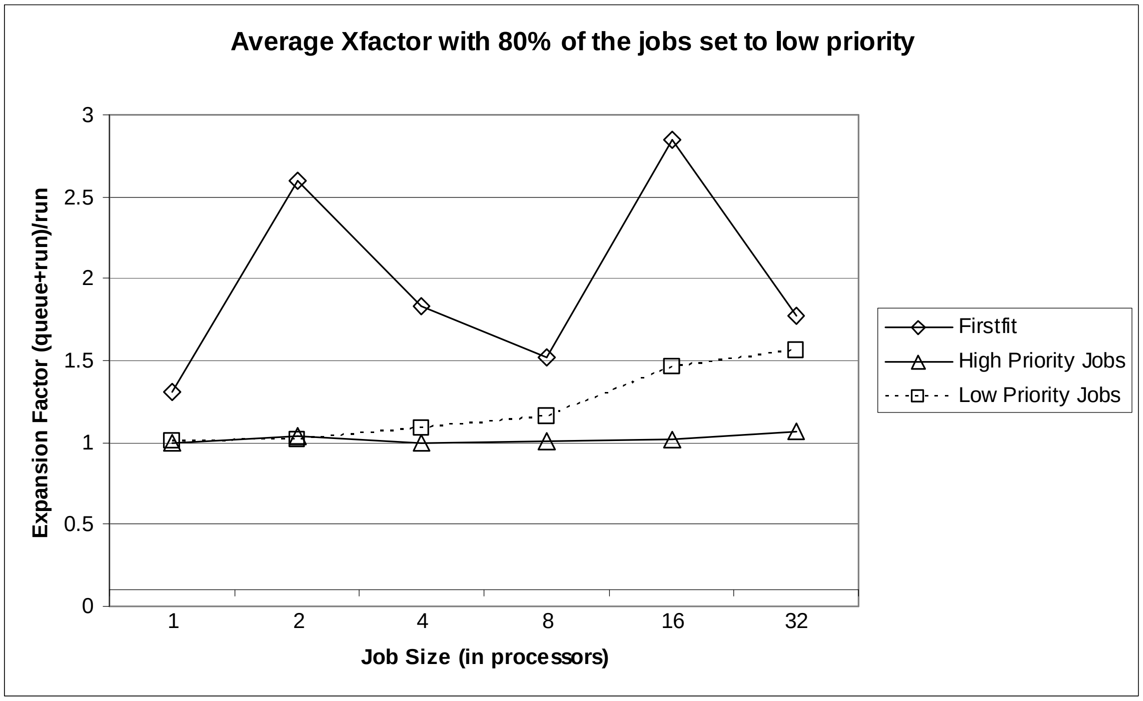

Figure 10: Expansion factor for a mix of high and low priority jobs. High priority jobs are able to achieve near-optimal performance even though low priority jobs are present.

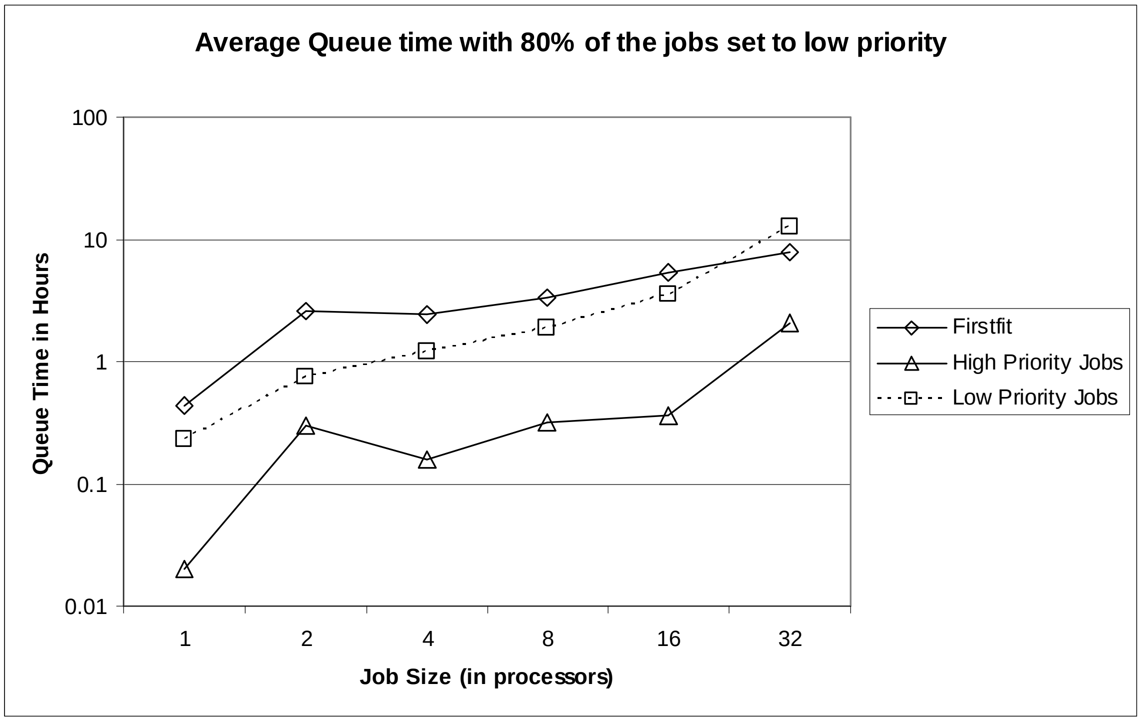

Figure 11: Queue time for a mix of high and low priority jobs. High priority jobs are able to achieve low queue times and low priority jobs have queue times similar to First Fit.

5 Conclusions

This paper examines the impact of preemption on the performance of the Maui supercomputer scheduler. If jobs can be preempted once they are started, more efficient use can be made of processor time that is to small for any eligible backfill jobs. Jobs that are larger than a backfill window can be started with the hope that they will complete before the window expires. Priority markings on lower priority jobs can allow high priority jobs to achieve a much higher quality of service than low priority jobs that run in idle time on a supercomputer. These priority markings allow users to share resources while maintaining preferential treatment for jobs submitted by privileged users.

The *duration-remaining* and *duration-consumed* preemption strategies result in the best average queue time and expansion factor for non-prioritized jobs. The queue time and expansion factor resulting from any of the strategies results in improved performance over First Fit without preemption.

This research shows that preemptive backfill algorithms can improve the performance of supercomputer schedulers. These results should make resource owners more willing to share their computing resources and will increase the utilization of supercomputing centers.

6 Acknowledgements

We appreciate the comments and suggestions we received from Brian Haymore and Julio Facelli from the Center for High Performance Computing at the University of Utah. They also provided us with job trace files from an environment where preemption could make a big difference. We also appreciate comments from Scott M Jackson from the Molecular Science Computing Facility at Pacific Northwest National Laboratory. Many of the features currently implemented in the Maui Scheduler were implemented in response to suggestions from PNNL.

7 References

[1] Anat Batat and Dror G. Feitelson, "Gang Scheduling with Memory Considerations", Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS'00)

[2] G. Berry, “Preemption in concurrent systems. In Proc. FSTTCS'93, Lecture Notes in Computer Science, volume 761, pages 72--93. Springer-Verlag, 1993.

[3] Dror G. Feitelson and L. Rudolph. “Parallel job scheduling: Issues and approaches”, IPPS'95 Workshop: Job Scheduling Strategies for Parallel Processing, pages 1-18. Springer{Verlag, Lecture Notes in Computer Science LNCS 949, 1995.

[4] Dror G. Feitelson and Morris Jette, “Improved Utilization and Responsiveness with Gang Scheduling”, Proceedings of the IPPS '97 Workshop on Job Scheduling Strategies for Parallel Processing, 1997.

[5] David Jackson, Quinn Snell, Mark Clement. “Core Algorithms of the Maui Scheduler”, Job Scheduling Strategies for Parallel Processing, Editors Dror G. Feitelson and Larry Rudolph, Springer Verlag, LNCS Vol. 2221, pages 87-102, June 2001

[5] David Jackson, "The Maui Scheduler", http://supercluster.org/projects/maui, May 6, 2002.

[6] Uwe Schwiegelshohn, "Preemptive weighted completion time scheduling of parallel jobs", 4th European Symp. Algorithms, pp. 39-51, Springer-Verlag, Sep 1996. Lecture Notes in Computer Science Vol. 1136.

[7] Q. Snell, M. Clement, D. Jackson and C. Gregory, "The Performance Impact of Advance Reservation Metascheduling", Lecture Notes in Computer Science: Job Scheduling Strategies for Parallel Processing, Springer-Verlag, volume 1911.