Multiple-queue Backfilling Scheduling with Priorities and Reservations for Parallel Systems*

Barry G. Lawson, Evgenia Smirni
Department of Computer Science
College of William and Mary
Williamsburg, VA 23187-8795, USA
{bglaws,esmirni}@cs.wm.edu

Abstract

We describe a new, non-FCFS policy to schedule parallel jobs on systems that may be part of a computational grid. Our algorithm continuously monitors the system (i.e., intensity of incoming jobs and variability of their resource demands) and continuously adapts its scheduling parameters to sudden workload fluctuations. The proposed policy is based on backfilling which permits job rearrangement in the waiting queue. By exploiting otherwise idle processors, this rearrangement reduces fragmentation of system resources, thereby providing higher system utilization. We propose to maintain multiple job queues that effectively separate jobs according to their projected execution time. Our policy supports different job priority classes as well as job reservations, making it appropriate for scheduling jobs on parallel systems that are part of a computational grid. Detailed performance comparisons via simulation using traces from the Parallel Workload Archive indicate that the proposed policy consistently outperforms traditional scheduling approaches.
Keywords: batch schedulers, computational grids, parallel systems, backfilling schedulers, performance analysis.

1 Introduction

The ubiquity of parallel systems, from clusters of workstations to large-scale supercomputers interconnected over the Internet, makes parallel resources highly available to researchers and practitioners. Because there is such a commodity of parallel resources that is often under-utilized, new research challenges emerge that focus on how to best harness the available parallelism of such computational grids. Resource allocation in parallel systems that are part of a grid is non-trivial. One of the major challenges includes co-scheduling distributed applications across multiple independent systems, each of which may itself be parallel with its own scheduler.

Traditional scheduling policies for stand-alone parallel systems focus on treating differently interactive versus batch jobs [3], with the goal of maximizing the utilization of an (often) expensive system. Commercial schedulers that are widely accepted by the high performance community, including the Maui scheduler [8] and PBS scheduler [11], offer a variety of configuration parameters that allow the system administrator to customize the scheduling policy according to the site's needs. The available configuration parameters include multiple queues to which different job classes are assigned, different job priorities, different scheduling policies per queue, and the ability to treat interactive jobs differently from batch jobs. The immediate benefit of such flexibility in policy parameterization is to allow the system administrator to customize the scheduling policy according to the site's needs. Yet, optimal policy customization to meet the needs of an ever changing workload is an elusive goal.

Scheduling jobs on a site that is part of a computational grid imposes additional challenges. The policy must cater to three classes of jobs: local jobs (parallel or sequential) that should be executed in a timely manner, jobs external to the site that do not have

*This work was partially supported by the National Science Foundation under grants EIA-9977030, EIA-9974992, CCR-0098278, and ACI-0090221.

high priority (i.e., jobs that can execute when the system is not busy serving local jobs), and external jobs that require reservations (i.e., jobs that require resources within a very restricted time frame to be successful).

In this paper, we propose a scheduling policy that self-adapts its configuration parameters according to the demands of the incoming workload and can cater to the three classes of jobs that we describe above. The proposed scheduling policy belongs to a class of space-sharing run-to-completion policies (i.e., no job preemption is allowed after a job is allocated its required processor resources) [2, 7, 13] that are often found in the heart of many popular parallel workload schedulers, including the Maui and PBS schedulers. This class of policies, commonly cited as *backfilling* policies, opt not to execute incoming jobs in their order of arrival, but rather rearrange their execution order to reduce system fragmentation and ensure better system utilization [7, 15]. Backfilling is extensively used by many schedulers, most notably the IBM LoadLeveler scheduler [4] and the Maui and PBS schedulers [1]. Various versions of backfilling have been proposed [5, 7, 10]; [5] characterizes the effect of job length and parallelism on backfilling performance and [10] proposes sorting by job length to improve backfilling.

In this paper, we propose modifications to a scheduling policy that we have previously proposed [6] in order to cater to jobs with different priorities and reservations. The proposed policy is based on the *aggressive* backfilling strategy extensively analyzed in [7]. In contrast to other backfilling-based schedulers, we maintain multiple queues and separate long from short jobs. Essentially, we split the system into multiple non-overlapping subsystems, one subsystem per queue. In this fashion, we manage to reduce the average job slowdown by reducing the likelihood that a short job is queued behind a long job.

The policy is inspired by related work in task assignment for distributed servers that strongly encourages separation of jobs according to their length, especially for workloads with execution times characterized by long-tailed distributions [12, 14]. By maintaining several queues and by continuously monitoring the performance of various job classes, we allow the scheduler to quickly change its configuration parameters according to fluctuations in job arrival intensities and service demands.

We conduct a set of simulation experiments using trace data from the Parallel Workload Archive [9]. Our simulations indicate that the proposed multiple-queue backfilling policy consistently outperforms a single-queue backfilling policy when job priorities and reservations are considered. Because of space restrictions, here we present results for different job priorities only.

This paper is organized as follows. Section 2 presents the proposed policy. Detailed performance analysis of the proposed policy is given in Section 3. Concluding remarks are given in Section 4.

2 Scheduling Policies

Successful contemporary schedulers utilize *backfilling*, a non-FCFS scheduling policy that reduces fragmentation of system resources by permitting jobs to execute in an order different than their arrival [5, 7]. A job that is backfilled is allowed to begin executing before previously submitted jobs that are delayed due to insufficient idle processors. Such non-FCFS execution order exploits otherwise idle processors, thereby increasing system utilization and throughput. The IBM LoadLeveler [4] and the Maui scheduler [8] are examples of popular commercial schedulers that incorporate backfilling.

For this work, we consider only *aggressive* backfilling, a non-preemptive policy in which a job is permitted to backfill provided it does not delay the anticipated starting time of first job in the queue[1]. In previous work [6], we have shown that the performance of aggressive backfilling improves by directing incoming jobs to separate queues according to the expected job duration. Each queue is assigned to a non-overlapping system partition.

Within the context of scheduling resources in a computational grid, we supplement the single- and multiple-queue backfilling policies by considering two static job priority levels. We consider those jobs submitted by local users to have high priority and those jobs submitted externally (i.e., from elsewhere in the computational grid) to have low priority. For the moment, we assume that external jobs do *not* require

[1] As an alternative, *conservative* backfilling permits a job to backfill provided it does not delay the anticipated starting time of any job in the queue. The performance of aggressive backfilling has been shown superior to that of conservative backfilling [7].

execution at a specific time, i.e., do not require reservations. Our goal is to serve these external jobs as quickly as possible without inflicting delays on local jobs. We now describe the necessary job prioritization in both the single-queue and multiple-queue backfilling policies.

2.1 Single-Queue Backfilling with Job Priorities

Standard aggressive backfilling incorporates a single queue of jobs to be executed. Each job in the queue is characterized by its arrival time, number of requested processors, and an estimate of the expected execution time. When a job is submitted to the system, the job is placed in the queue according to the job's priority. If the job has low priority, it is placed at the tail of the queue; if the job has high priority, it is placed after all previously queued high priority jobs but before the first low priority job in the queue. This ensures that all high priority jobs are considered for backfilling before any low priority job. Starting from the head of the queue, the scheduler iteratively considers each job in the queue, attempting to backfill (i.e., execute) any job that will not delay the first non-executing job in the queue. The scheduler terminates any job that attempts to execute for a time longer than its estimated execution time.

In general, the process of backfilling *exactly one* queued job (of possibly many queued jobs to be backfilled) proceeds as follows. Define the *pivot job* to be the first job in the queue. If sufficient idle processors are available, the pivot begins executing and a new pivot is defined. Otherwise, the *pivot start time* (i.e., the time when the pivot can begin executing) is determined. Any idle processors at the pivot start time not required by the pivot are denoted *extra* processors. The scheduler then backfills the first job (if any) encountered in the queue that (a) requires no more than the currently idle processors *and* will finish by the pivot start time, or (b) requires no more than the minimum of the currently idle and extra processors.

This process of backfilling exactly one job is repeated until all queued jobs have been considered. The single-queue aggressive backfilling policy, outlined in Figure 1, is used whenever a job is submitted or whenever an executing job completes. Note that if a pivot has low priority, an arriving job of high priority will become the new pivot.

2.2 Multiple-Queue Backfilling with Job Priorities

Multiple-queue backfilling allows the scheduler to change system parameters automatically in response to transient workload fluctuations, thereby reducing the average slowdown experienced by jobs [6]. The system is divided into multiple disjoint partitions by classifying jobs according to their duration[2] (*not* their requested number of processors). Initially, processors are divided evenly among the partitions. As time evolves, the partitions may exchange processors so that processors idle in one partition can be used for backfilling in another partition. If a job in a certain partition cannot begin executing due to insufficient idle processors in that partition, the partition will assume control of idle processors from one or more other partitions until the job can begin executing. Therefore, partition boundaries become dynamic, allowing the system to adapt itself to changing workload conditions. Furthermore, the policy does not starve a job that requires the entire machine for execution.

Each partition contains its own separate queue of jobs. When a job is submitted, it is classified and assigned to the queue in partition p according to the following equation, where t_e is the estimated job execution time[3].

$$p = \begin{cases} 1, & 0 < t_e < 100 \\ 2, & 100 \le t_e < 1000 \\ 3, & 1000 \le t_e < 10\,000 \\ 4, & 10\,000 \le t_e \end{cases}$$

The job is placed into the queue in partition p according to the job priority. If the job has low priority, it is placed at the tail of the queue. If the job has high priority, it is placed in the queue after all previously queued high priority jobs in partition p but before the first low priority job in partition p.

We also incorporate a global variable delay. Beginning with an initial delay of 2500 seconds, the scheduler continuously monitors the average job slowdown of each job class and adjusts the size of the delay accordingly. Any extra-long job that has low prior-

[2] For this work, we assume accurate estimates, i.e., we classify according to their actual duration. In future work, we will explore policy parameterization under inaccurate estimates.

[3] Because the actual workload traces exhibit wide variabilities in job duration, we empirically determined the job classification parameters to provide a representative proportion of jobs in each class across all traces. We direct the interested reader to [6] for details regarding this classification.

for (all high priority jobs in order of arrival, then all low priority jobs in order of arrival)
1. *pivot job* ⟵ first job in queue
2. if possible, start pivot job immediately
3. else
  a. *pivot start time* ⟵ time when pivot can begin executing
  b. *extra procs* ⟵ idle processors at pivot time not used by pivot job
  c. while (no job backfilled and more queued jobs to consider)
    I. consider next job in queue
    II. if job requires $\leq$ currently idle procs and will finish by pivot start time, start job immediately
    III. else if job requires $\leq$ min{currently idle procs, extra procs}, start job immediately

Figure 1: Single-queue aggressive backfilling algorithm with job priorities.

ity is subject to this mandatory delay. Because the mandatory delay enforced on extra-long low priority jobs is small relative to the execution time of those jobs, the execution time absorbs the impact of that delay on the computed slowdown. The goal is to increase the delay on extra-long low priority jobs when shorter jobs are suffering and to decrease the delay when shorter jobs are thriving, but without ever penalizing high priority jobs.

In general, the process of backfilling *exactly one* queued job (of possibly many queued jobs to be backfilled) proceeds as follows. Let p be the partition to which the job belongs. Define $pivot_p$ to be the first job in the queue in partition p, and define *pivot start* $time_p$ to be the time when $pivot_p$ can begin executing. If the job under consideration is $pivot_p$, it begins executing only if the current time is equal to *pivot start* $time_p$, in which case a new $pivot_p$ is defined. If the job is not $pivot_p$, it begins executing only if there are sufficient idle processors in partition p without delaying $pivot_p$, or if partition p can obtain sufficient idle processors from one or more other partitions without delaying *any* pivot.

This process of backfilling exactly one job is repeated, one job at a time, until all queued jobs have been considered. All high priority jobs are considered first (in their order of arrival across all queues) followed by low priority jobs (in their order of arrival across all queues). The multiple-queue aggressive backfilling policy, outlined in Figure 2, is utilized whenever a job is submitted or whenever an executing job completes. Similar to the single-queue policy, if a high priority job arrives at partition p and finds $pivot_p$ to have low priority, the high priority job immediately replaces the low priority job as $pivot_p$. Note that a high priority pivot takes precedence over any other low priority pivot(s). In other words, the scheduling of a start time for a high priority pivot is permitted to delay other low priority pivots (but not other high priority pivots). The scheduling of a start time for a low priority pivot cannot delay *any* other pivots.

3 Performance Analysis

In this section, we evaluate via simulation the performance of the single-queue and multiple-queue backfilling policies with job priorities presented in the previous section. Our simulation experiments are driven using four workload traces from the Parallel Workload Archive [9]. Each trace provides the submission time of each job, the number of processors requested, the estimated duration of the job, the actual duration of the job, the start time of the job, and possible additional resource requests (e.g., memory per node). The selected traces are summarized below.

- **CTC**: This trace contains entries for 79 302 jobs that were executed on a 512-node IBM SP2 at the Cornell Theory Center from July 1996 through May 1997.
- **KTH** : This trace contains entries for 28 490 jobs executed on a 100-node IBM SP2 at the Swedish Royal Institute of Technology from October 1996 through August 1997.

for (all high priority jobs in order of arrival, then all low priority jobs in order of arrival)
1. $p \longleftarrow$ partition in which job resides
2. $pivot_p \longleftarrow$ first job in queue in partition p
3. $pivot\ start\ time_p \longleftarrow$ earliest time when sufficient procs (from this and perhaps other partitions) will be available for $pivot_p$ without delaying any other pivot of equal or higher priority
4. $idle_p \longleftarrow$ currently idle processors in partition p
5. $extra_p \longleftarrow$ idle processors in partition p at $pivot\ start\ time_p$ not used by $pivot_p$
6. if job is $pivot_p$
  a. if current time equals $pivot\ start\ time_p$
    I. if necessary, reassign procs from other partitions to partition q
    II. start job immediately
7. else
  a. if job requires $\leq idle_p$ and will finish by $pivot\ start\ time_p$, start job immediately
  b. else if job requires $\leq \min\{idle_p,\ extra_p\}$, start job immediately
  c. else if job requires $\leq$ ($idle_p$ plus some combination of idle/extra procs from other partitions) such that no pivot is delayed
    I. reassign necessary procs from other partitions to partition p
    II. start job immediately

Figure 2: Multiple-queue aggressive backfilling algorithm with job priorities.

- **SDSC–SP2**: This trace contains entries for 67 667 jobs that were executed on a 128-node IBM SP2 at the San Diego Supercomputing Center from May 1998 through April 2000.
- **SDSC–PAR**: This trace contains entries for 38 723 jobs that were executed on a 416-node Intel Paragon at the San Diego Supercomputer Center during 1996.

We consider both *aggregate* performance measures, i.e., average statistics computed for all jobs for the entire experiment, and *transient* performance measures, i.e., snapshot statistics for batches of 1000 jobs that are plotted across the experiment time and illustrate how well the policy reacts to sudden workload changes. For all results to follow, we use the single-queue and multiple-queue backfilling policies with job priorities presented in Sections 2.1 and 2.2.

The performance measure of interest is the average job slowdown s defined as

$$s = 1 + \frac{d}{\nu}$$

where d and ν are respectively the average queueing delay time and actual service time of a job. To compare the performance results of single-queue and multiple-queue backfilling, we also define the slowdown ratio $\mathcal{R}$ as

$$\mathcal{R} = \frac{s_1 - s_m}{\min\{s_1, s_m\}}$$

where s_1 and s_m are the single-queue and multiple-queue slowdowns respectively. $\mathcal{R} > 0$ corresponds to the performance gain obtained using multiple queues relative to a single queue; $\mathcal{R} < 0$ corresponds to the performance loss suffered using multiple queues relative to a single queue.

We first consider a system in which 25% of the submissions are from an external source in the computational grid, i.e., 75% of the jobs are high priority. We select at random 75% of the jobs from the trace to be high priority jobs, so that the remaining 25% are low priority jobs. Figure 3 depicts the aggregate slowdown ratio $\mathcal{R}$ of multiple-queue backfilling relative to single-queue backfilling with 75% high priority jobs for each of the four traces. Figure 3(a) shows $\mathcal{R}$ for all job classes, while Figures 3(b)–(e) each show $\mathcal{R}$ for an individual job class. For each trace, we also provide $\mathcal{R}$ as computed for all jobs, for high priority jobs, and for low priority jobs. As shown in this figure, multiple-queue backfilling proves superior to single-queue backfilling across all job classes (see Figure 3(a)) for all jobs, for high priority jobs, and for

low priority jobs. We also note that, with the exception of the extra-long class (Figure 3(e)), multiple-queue backfilling performs better within each of the individual job classes (Figures 3(b)–(d)). Because the goal of multiple-queue backfilling is to improve average slowdown by assisting shorter jobs at the expense of very long jobs, a decline in performance of extra long jobs is expected.

Because a system that is part of a computational grid can experience dramatic changes in workload across time, we also consider the transient performance of multiple-queue versus single-queue backfilling. Figure 4 depicts transient snapshots of the slowdown ratio versus time for each of the four traces with 75% high priority jobs. Each figure shows snapshots of both high priority jobs and low priority jobs. Again, marked improvement in slowdown is achieved ($\mathcal{R} > 0$) using multiple-queue backfilling. Although single-queue backfilling provides better slowdown ($\mathcal{R} < 0$) for a few batches, $\mathcal{R}$ is positive for a majority of the time corresponding to performance gains with multiple-queue backfilling.

We also considered a system in which 5% of the submissions are external, i.e., 95% of the jobs have high priority. Due to space restrictions, we do not present any graphs here but instead briefly summarize these results. For 95% high priority jobs, performance gains are achieved using multiple-queue backfilling across all job classes and for the first three individual job classes, with even larger performance gains than for 75% high priority jobs. Again, as expected we see a decline in performance for extra-long jobs necessary to boost the overall and other-class performance.

4 Conclusions

We presented multiple-queue backfilling as a viable approach for scheduling resources in parallel systems that are part of a computational grid. Each job is assigned to a queue according to its expected execution time. Each queue is assigned a non-overlapping partition of system resources on which jobs from the queue can execute. Partition boundaries change dynamically, adjusting to fluctuations in arrival intensities and workload mix.

The proposed policy consistently outperforms single-queue backfilling. The performance gains are a direct result of the fact that the multiple-queue policy significantly reduces the likelihood that a short job is overly delayed in the queue behind a very long job. Performance gains are even more prominent when jobs with different priorities are considered.

Acknowledgments

We thank Tom Crockett for useful discussions that contributed to this work. We also thank Dror Feitelson for making available the workload traces through the Parallel Workload Archive.

References

[1] Bode B., Halstead D.H., Kendall R., Lei Z., "The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters", in *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, Georgia, October 2000.

[2] Chiang S.-H., Mansharamani R.K., Vernon M.K., "Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies," *Proc. of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1994, pp. 33-44.

[3] Feitelson D.G., "A survey of scheduling in multiprogrammed parallel systems", Technical Report RC 19790, IBM Research Division, October 1994.

[4] IBM LoadLeveler, `http://www.ibm.com/products/`.

[5] Keleher P., Zotkin D., Perkovic D, "Attacking the Bottlenecks in Backfilling Schedulers", in *Cluster Computing: The Journal of Networks, Software Tools and Applications*, 3(4): 2000.

[6] Lawson B. G., Smirni E., Puiu D., "Self-adapting Backfilling Schedulers for Parallel Systems", Technical Report (*submitted for publication*), College of William and Mary, Williamsburg, VA, January 2002.

[7] Mualem A. W., Feitelson D. G., "Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling", IEEE Trans. Parallel and Distributed Syst. 12(6), pp. 529-543, June 2001.

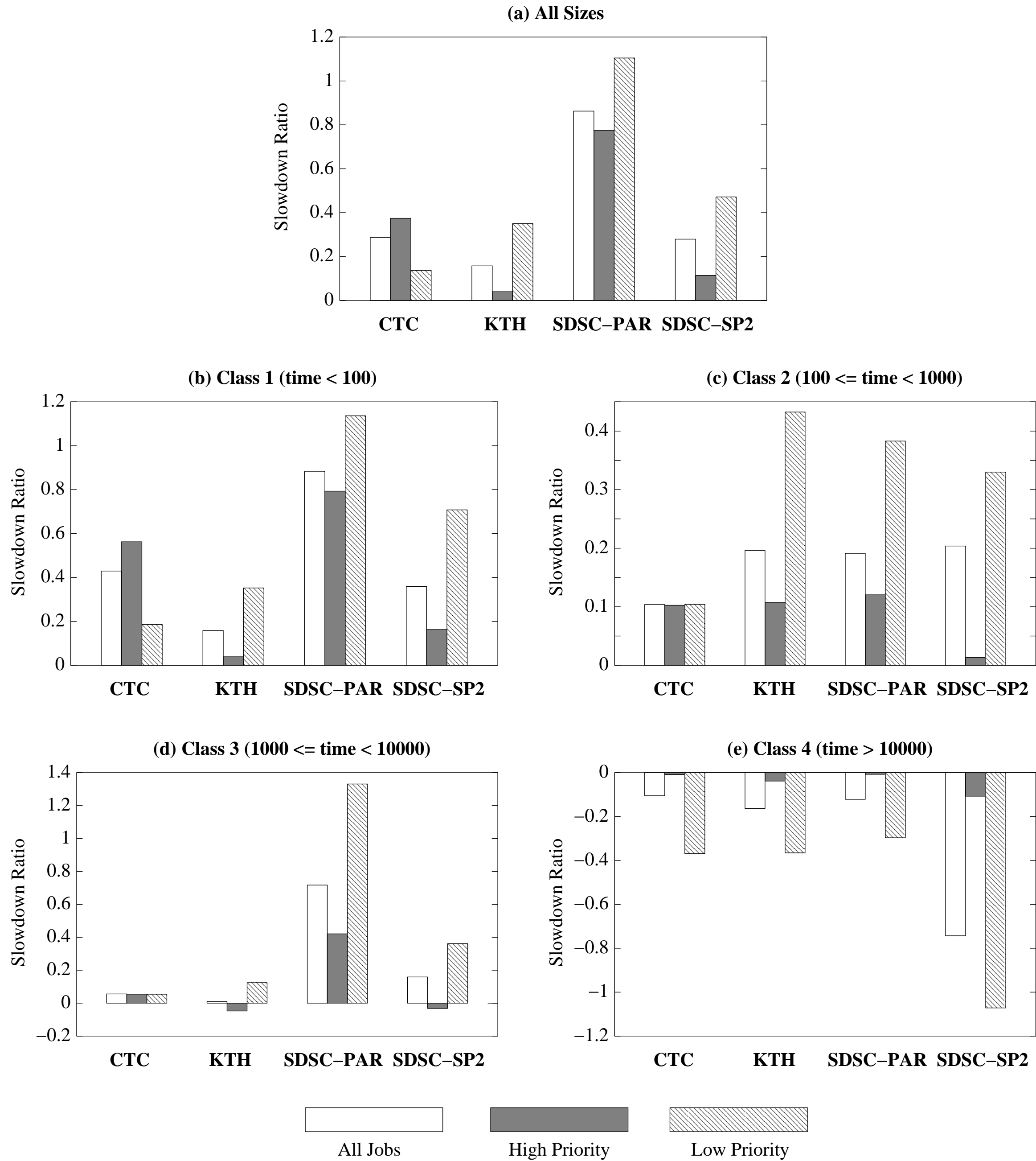

Figure 3: Overall and per-class aggregate slowdown ratio $\mathcal{R}$ for each of the four traces with 75% high priority jobs. For each trace, $\mathcal{R}$ is computed for all jobs, for only high priority jobs, and for only low priority jobs.

[8] Maui Scheduler Open Cluster Software, `http://mauischeduler.sourceforge.net/`.

[9] Parallel Workload Archive, `http://www.cs.huji.ac.il/labs/parallel/workload/`.

[10] Perkovic D., Keleher P., "Randomization, Speculation, and Adaptation in Batch Schedulers", in *Proceedings of Supercomputing 2000 (SC2000)*, November 2000.

[11] Portable Batch System,

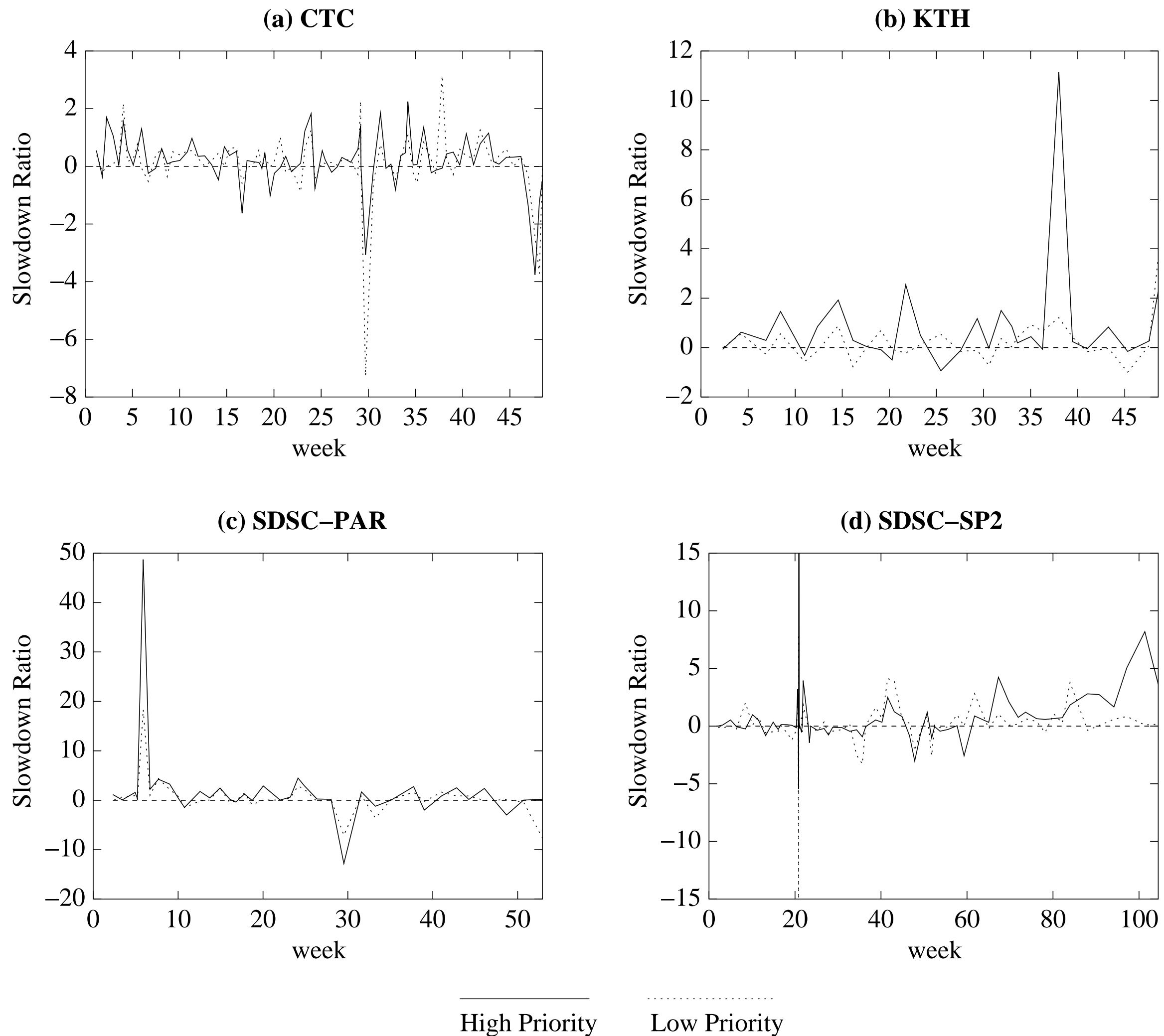

Figure 4: Slowdown ratio $\mathcal{R}$ per 1000 job submissions as a function of time for high priority and low priority jobs for each of the four traces with 75% high priority jobs.

http://www.openpbs.org/.

[12] A. Riska, W. Sun, E. Smirni, G. Ciardo, "AdaptLoad: effective balancing in clustered web servers under transient load conditions", to appear at the *22nd International Conference on. Distributed Computing Systems*, (ICDCS 2002), Vienna, Austria, July 2002.

[13] Rosti E., Smirni E., Dowdy L.W., Serazzi G., Carlson B.M., "Robust partitioning policies for multiprocessor systems", *Performance Evaluation* 19 (1994), pp.141-165, Special Issue on Parallel Systems.

[14] Schroeder B., Harchol-Balter M., "Evaluation of Task Assignment Policies for Supercomputing Servers: The Case for Load Unbalancing and Fairness", in *Proceedings of the 9th IEEE Symposium on High Performance Distributed Computing (HPDC '00)*, Pittsburgh, Pennsylvania, August 2000.

[15] Talby D., Feitelson D.G., "Supporting priorities and improving utilization of the IBM SP2 scheduler using slack-based backfilling", in *Proceedings of the 13th International Parallel Processing Symposium*, pp. 513-517, Apr 1999.