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

Barry G. Lawson and Evgenia Smirni

The College of William and Mary, Department of Computer Science,
P.O. Box 8795, 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., the intensity of incoming jobs and variability of their resource demands), and adapts its scheduling parameters according to workload fluctuations. The proposed policy is based on backfilling, which reduces resource fragmentation by executing jobs in an order different than their arrival order without delaying certain previously submitted jobs. We maintain multiple job queues that effectively separate jobs according to their projected execution time. Our policy supports different job priorities and 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 backfilling.
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 via the Internet, makes parallel resources easily available to researchers and practitioners. Because there is such a commodity of parallel resources that is often underutilized, 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 nontrivial. 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 [1] in order to maximize the utilization of an (often) expensive system. Because it reduces resource fragmentation

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

and increases system utilization, backfilling has been proposed as a more efficient alternative to simple FCFS schedulers [7, 10]. Users are expected to provide nearly accurate estimates of job execution times. Using these estimates, the scheduler rearranges the waiting queue, allowing short jobs to move ahead of long jobs provided certain previously submitted jobs are not delayed. Various versions of backfilling have been proposed [4, 7, 9]. Keleher et al. characterize the effect of job length and parallelism on backfilling performance [4]. Perkovic and Keleher propose sorting by job length to improve backfilling and introduce the idea of speculative execution, in which long jobs are given a short trial execution to detect whether or not the jobs crash [9].

Industrial schedulers that are widely accepted by the high performance community, including the Maui Scheduler [6] and PBS scheduler [8], offer a variety of configuration parameters. Such parameters include multiple queues to which different job classes are assigned, multiple job priorities, multiple scheduling policies per queue, and the capability to treat interactive jobs differently from batch jobs. The immediate benefit of such flexibility in policy parameterization is the ability 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 high priority (i.e., jobs that can execute when the system is not busy serving local jobs), and jobs that require reservations (i.e., require resources within a very restricted time frame to be successful).

In previous work, we proposed a multiple-queue aggressive backfilling scheduling policy that continuously monitors system performance and changes its own parameters according to changes in the workload [5]. In this paper, we propose modifications to the policy that address job priorities and job reservations. We conduct a set of simulation experiments using trace data from the Parallel Workload Archive [2]. Our simulations indicate that, even in the presence of inaccurate estimates, the proposed multiple-queue backfilling policy outperforms traditional backfilling when job priorities and reservations are considered.

This paper is organized as follows. Section 2 describes the proposed multiple-queue backfilling policy. Detailed performance analysis of the policy is given in Section 3. Concluding remarks are provided 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 [4, 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 [3] and the Maui Scheduler [6] are examples of popular schedulers that incorporate backfilling.

Aggressive backfilling permits a job to backfill provided the job does not delay the first job in the queue. Alternatively, *conservative* backfilling permits a job to backfill provided the job does not delay any previous job in the queue. Because the performance of aggressive backfilling has been shown superior to that of conservative backfilling [7], in this work we consider only aggressive backfilling.

Standard aggressive backfilling assumes a single queue of jobs to be executed. In previous work, we showed that the performance of aggressive backfilling improves by directing incoming jobs to separate queues according to job duration [5]. The goal of this multiple-queue policy is to reduce the likelihood of delaying a short job behind a long job. By separating jobs into different queues, a queued job competes *directly* only with jobs in the same queue for access to resources. Relative to using a single queue, short jobs therefore tend to gain access to resources more quickly, while long jobs tend to be delayed slightly. As a result, short jobs are assisted at the expense of long jobs using the multiple-queue policy, thereby improving the average job slowdown. Using detailed workload characterization of traces from the Parallel Workload Archive, four queues provided a nearly equal proportion of jobs per queue. Hence, we employed a four-part job classification (using *actual* job execution times) to effectively separate short from long jobs, thereby improving job slowdown.[1] However, if estimates are not accurate, the classification does not separate jobs of different lengths effectively, and the policy performance may degrade significantly.

Indeed, according to trace data, users tend to overestimate run times. Analysis of the workloads shows that the mean estimated run time is consistently twice the mean actual run time. In addition, many jobs appear to crash, i.e., have *very* long estimates but *very* short actual run times. For three traces from the Parallel Workload Archive, Table 1 shows the significant proportion of total jobs that have estimated run times greater than 1000 seconds but actual run times less than 180 seconds. This combination of overestimates and crashed jobs causes the four-part classification presented in [5] to fail.

Table 1. Proportion of (possibly) crashed jobs for three parallel workload traces

Trace	Jobs	Crashed	Proportion
CTC	79 302	12 903	0.16
KTH	28 487	3000	0.11
SP2	67 665	15 974	0.24

In this work, we use *estimated* job execution times from the traces and assume that users overestimate the actual run times of jobs. Correspondingly, we use a three-part job classification in response to these overestimates. In addi-

[1] We direct the interested reader to [5] for further details on the four-part classification.

tion, because actual job execution times cannot be known *a priori*, we employ speculative execution of jobs [9] to quickly remove from the system a large proportion of the jobs that appear to crash. The immediate benefit is that such jobs, which are actually short jobs, are not unwittingly grouped and scheduled with long jobs. These modifications permit our multiple-queue policy to improve job slowdown even in the presence of poor user estimates.

Within the context of scheduling resources in a computational grid, we supplement our multiple-queue backfilling policy by considering static job priority levels and job reservations as follows.

- We consider jobs submitted by local users to have high priority and those jobs submitted from an external source (i.e., from elsewhere in the computational grid) to have low priority. Our goal is to serve these external jobs as quickly as possible without inflicting delays on local jobs.
- We also assume that the system serves jobs that require execution at a specific time. Our goal is to accommodate these reservations as quickly as possible regardless of the consequences on remaining jobs.

We now describe in detail the multiple-queue backfilling policy with the necessary job prioritization and reservation schemes.

2.1 Multiple-Queue Backfilling with Job Priorities and Speculation

Multiple-queue backfilling allows the scheduler to automatically change system parameters in response to workload fluctuations, thereby reducing the average job slowdown [5]. The system is divided into multiple disjoint partitions, with one queue per partition. As shown in Figure 1, each partition is initially assigned an equal number of processors. As time evolves, the partitions may exchange control of processors so that processors idle in one partition can be used for backfilling in another partition. 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.

In [5], based on workload characterization of *actual* run times, four queues provided the best separation of jobs to improve slowdown. Here, we empirically determined that a similar separation is achieved by directing jobs into three queues according to *estimated* run times. 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 in seconds, according to

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

If the arriving job cannot begin executing immediately, it is placed into the queue in partition p after all jobs of the same priority that arrived earlier. More specifically, if the job has high priority, it is placed into the queue after any high priority jobs that arrived before it. If the job has low priority, it is placed into

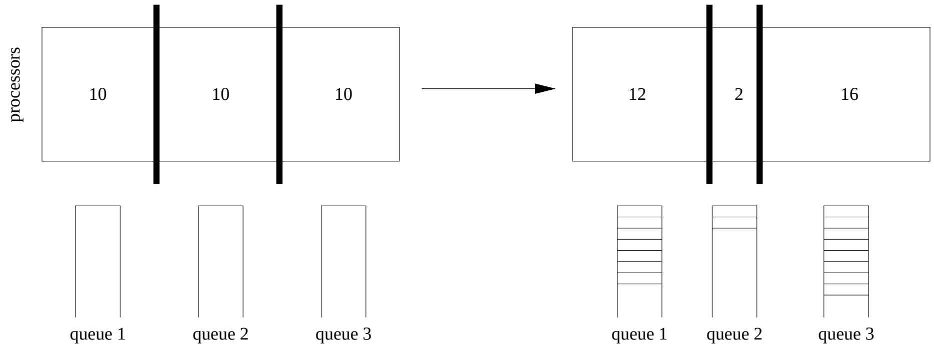

Fig. 1. In multiple-queue backfilling, initial partition boundaries adapt as workload conditions change. In this example, we consider 30 processors and three partitions (queues)

the queue after all high priority jobs and after any low priority jobs that arrived before it.

We use speculative execution to address the issue of a significant proportion of jobs that appear to crash.[2] If the estimated execution time of a submitted job is greater than 1000 seconds (i.e., belongs to partition two or three), the job is scheduled for speculative execution at the earliest possible time for a maximum of 180 seconds.[3] If the job does not terminate (successfully or unsuccessfully) within the allotted 180 seconds, the job is killed and is then placed into the queue in partition p according to the job's priority. Without speculative execution, jobs with long estimates that crash quickly and jobs with extremely poor estimates will be classified inappropriately, causing the performance of the multiple-queue policy to suffer.

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 $pivot_p$, the job 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. The policy considers high priority jobs first

[2] Within the context of real systems, as a general rule jobs cannot be killed and restarted. Speculative execution can be used, however, by permitting a user to flag a job as restartable (when appropriate) with the anticipation of improved slowdown.

[3] We experimented with speculative execution times from one to five minutes. Speculative execution for a maximum of three minutes removes most of the jobs that appear to crash, as depicted in Table 1.

```
if (non-speculative arriving job or speculative job killed to be queued)
   1. p ⟵ partition to which job is assigned
   2. insert into queue in partition p after all earlier-arriving, same-priority jobs
else
   schedule job immediately for speculative execution
for (high priority jobs in arrival order, then low priority jobs in arrival order)
   1. p ⟵ partition in which job resides
   2. pivot_p ⟵ first job in queue in partition p
   3. pivot start time_p ⟵ earliest time when sufficient procs (from this and
                              perhaps other partitions) will be available for pivot_p
                              without delaying any pivot of equal or higher priority
   4. idle_p ⟵ currently idle procs in partition p
   5. extra_p ⟵ idle procs 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 ≤ idle_p and will finish by pivot start time_p,
             start job immediately
        b. else if job requires ≤ min{idle_p, extra_p}, start job immediately
        c. else if job requires ≤ (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
```

Fig. 2. Multiple-queue aggressive backfilling algorithm with job priorities and speculation

(in their order of arrival, regardless of partition) followed by low priority jobs (in their order of arrival, regardless of partition). The multiple-queue aggressive backfilling policy with job priorities and speculation, outlined in Figure 2, is utilized whenever a job is submitted or whenever an executing job completes. 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 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.

2.2 Backfilling with Reservations

A user may schedule a reservation for future execution of a job if, for example, a dedicated environment is desired. Accordingly, when a request for a reservation is submitted, the scheduler determines the earliest time greater than or equal to the requested reservation time when the job can be serviced, and immediately schedules the job for execution at that time. For simplicity, we assume that once a job receives a reservation, the reservation will not be canceled nor can the time of the reservation be changed. Furthermore, we assume that all non-reservation jobs have the same priority. Therefore, the process of backfilling with reservations remains as described in Section 2.1, with the exception that all reservations must be honored.

3 Performance Analysis

In this section, we evaluate via simulation the performance of our multiple-queue backfilling policy relative to standard single-queue backfilling. Our simulation experiments are driven using the CTC, KTH, PAR (1996), and SP2 workload traces from the Parallel Workload Archive [2]. From the traces, for each job we extract the arrival time of the job (i.e., the submission time), the number of processors requested, the estimated duration of the job (if available), and the actual duration of the job. Because we do not use the job completion times from the traces, the scheduling strategies used on these systems are not relevant to our study. 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 28 487 jobs executed on a 100-node IBM SP2 at the Swedish Royal Institute of Technology from October 1996 through August 1997.
- **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. Because this trace contains no user estimates, we use the actual run times as accurate estimates.

- **SP2**: This trace contains entries for 67 665 jobs that were executed on a 128-node IBM SP2 at the San Diego Supercomputer Center from May 1998 through April 2000.

For all results to follow, we compare the performance of multiple-queue backfilling (using the three-part classification described in Section 2.1) to single-queue backfilling, both employing speculative execution. We consider *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 changes in the workload. The performance measure of interest here is the job slowdown s defined by

$$s = 1 + \frac{d}{\nu} \tag{2}$$

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

$$\mathcal{R} = \frac{s_1 - s_m}{\min\{s_1, s_m\}} \tag{3}$$

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

3.1 Multiple-Queue Backfilling Performance

We first consider the performance of multiple-queue backfilling with no job priorities or reservations. Figure 3 depicts the aggregate slowdown ratio $\mathcal{R}$ of multiple-queue backfilling relative to single-queue backfilling (computed using the average slowdown obtained using each policy) for each of the four traces. Figure 3(a) depicts $\mathcal{R}$ for all job classes combined, while Figures 3(b)–(d) each depict $\mathcal{R}$ for an individual job class. As shown, multiple-queue backfilling provides better job slowdown (i.e., $\mathcal{R} > 0$) for all classes combined (Figure 3(a)). With the exception of the long job class in the two SDSC workloads (Figure 3(d)), multiple-queue backfilling also provides better average job slowdown within each of the individual job classes.

Because a system can experience significant changes in workload across time, we also consider the transient performance of multiple-queue backfilling. Figure 4 depicts transient snapshots of the slowdown ratio versus time for each of the four traces. Again, marked improvement in job 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 a majority of the time corresponding to performance gains with multiple-queue backfilling.

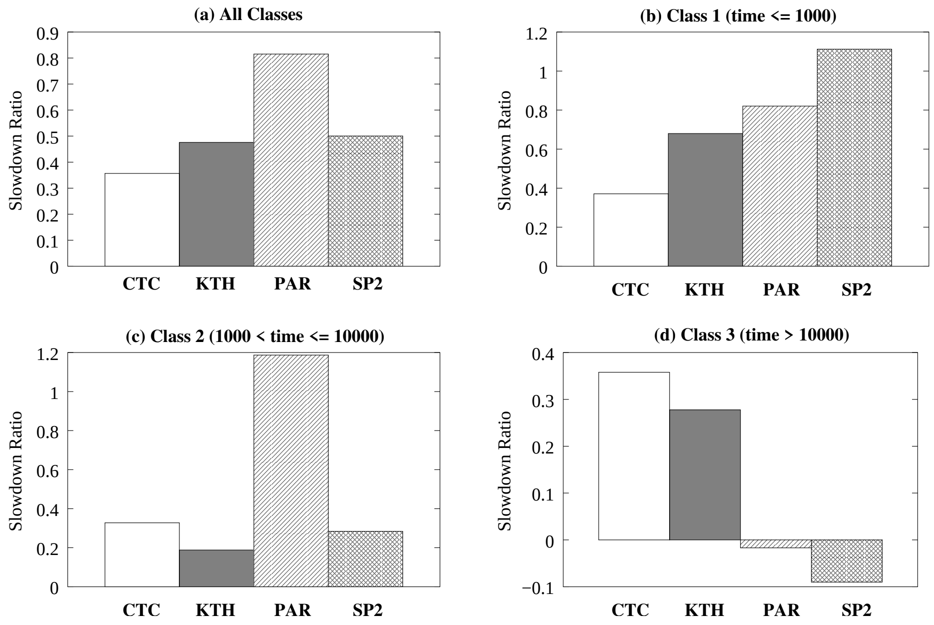

Fig. 3. Overall and per-class aggregate slowdown ratio $\mathcal{R}$ for each of the four traces

3.2 Performance Under Heavy Load

Most policies perform well under low system load because little, if any, queuing is present. To further evaluate multiple-queue backfilling, we now consider its performance under heavy system load when scheduling is more difficult. We impose a heavier system load than that of the trace by linearly scaling (reducing) subsequent interarrival times in the trace. Effectively, we linearly increase the arrival rate of jobs in the system. Note that with this modification, we preserve the statistical characteristics of the arrival pattern in the original trace, except that the same jobs now arrive "faster".

Figure 5 again depicts the aggregate slowdown ratio $\mathcal{R}$ of multiple-queue backfilling relative to single-queue backfilling for each of the four workloads. However, here for each workload we display $\mathcal{R}$ for the original arrival rate and for arrival rates multiplied by factors of 1.25 and 1.50. In all figures, the multiple-queue and single-queue backfilling policies experience the same rate of arriving jobs. Consistent with the original trace results presented in Figure 3 (i.e., a multiplicative factor of 1.00), multiple-queue backfilling provides better average job slowdown than single-queue backfilling for all job classes combined and for each individual job class (with the exception of the two SDSC workloads in the long job class). When we increase the arrival rate, multiple-queue backfilling continues to provide better average job slowdown for all job classes combined (Figure 5(a)) and for the small and medium job classes (Figure 5(b) and (c)). As discussed earlier, a queued job competes directly only with other jobs in the

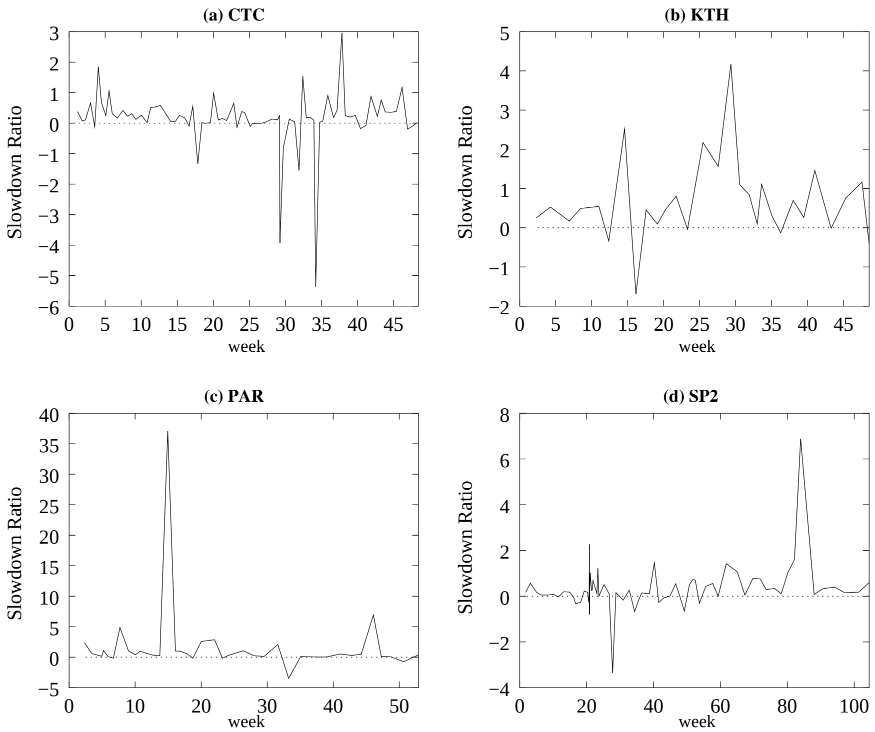

Fig. 4. Slowdown ratio $\mathcal{R}$ per 1000 job submissions as a function of time

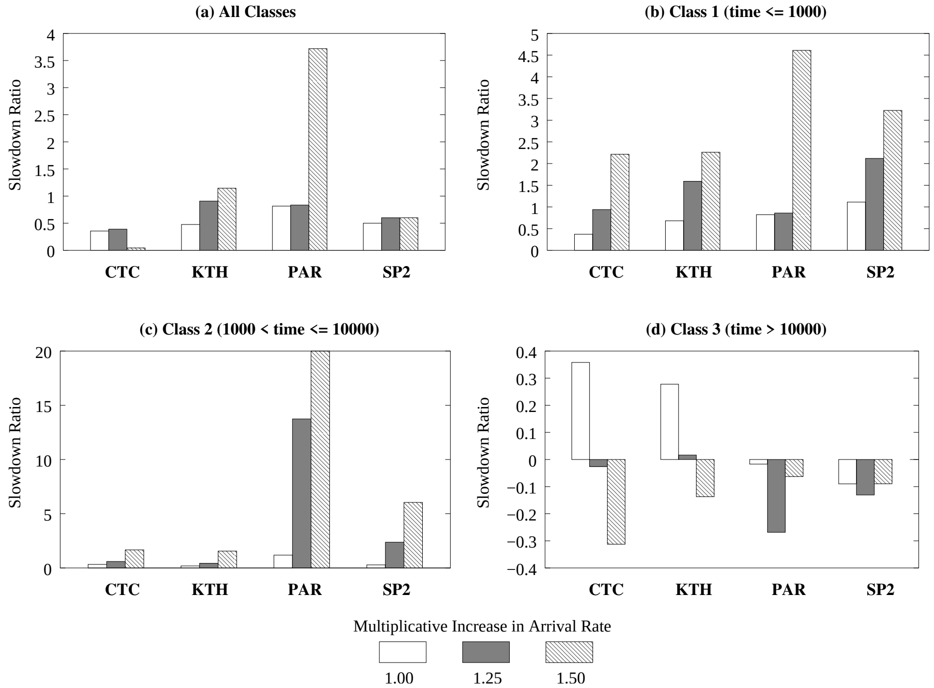

Fig. 5. Overall and per-class aggregate slowdown ratio $\mathcal{R}$ for each of the four traces with increasing system load. All ratios are computed relative to single-queue backfilling under the same load

same queue so that short jobs tend to be scheduled more quickly and long jobs tend to be delayed slightly. Therefore, multiple-queue backfilling assists shorter jobs at the expense of long jobs, and a decline in the performance of the long job class is unavoidable (Figure 5(d)). However, the magnitude of this decline is generally much smaller than the magnitude of improvement achieved for the other job classes.

Also notice in Figure 5(d) that, relative to single-queue backfilling, multiple-queue backfilling performs worse for a multiplicative increase of 1.25 than for 1.50 for the PAR and SP2 traces. Because job scheduling is very dependent on the arriving workload, the backfilling that occurs for an increase of 1.25 may be very different than for an increase of 1.50. Therefore, a monotone pattern in improvement or decline cannot be expected.

3.3 Performance Under Job Priorities

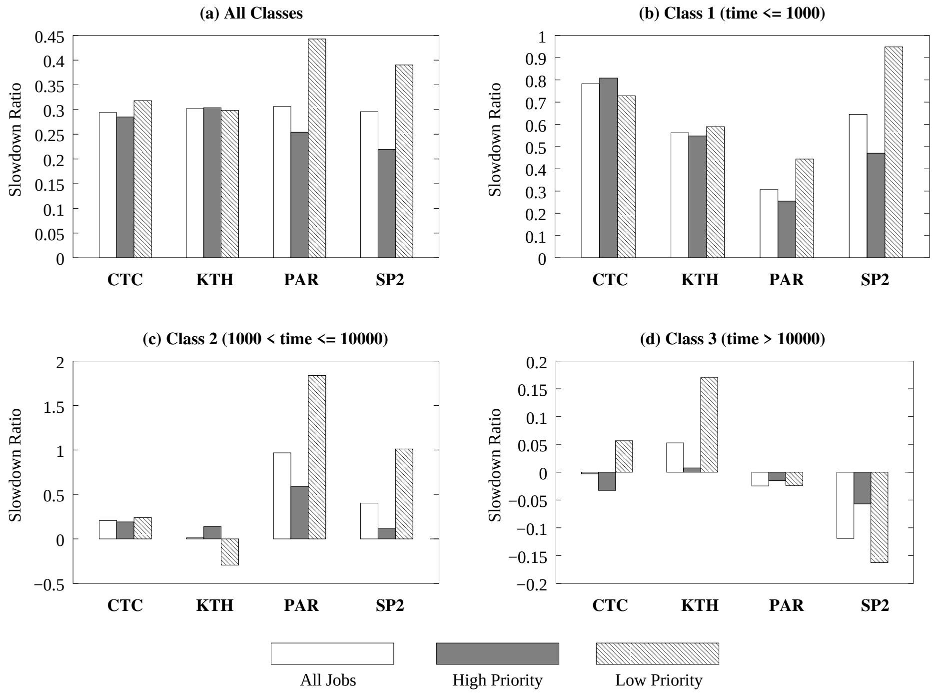

Fig. 6. Overall and per-class aggregate slowdown ratio $\mathcal{R}$ for each of the four traces with 75% high priority jobs

We now consider a system in which 75% of the jobs are high priority, i.e., 25% of the submissions are from an external source in the computational grid. We select at random 75% of the jobs from the trace to be high priority jobs, so

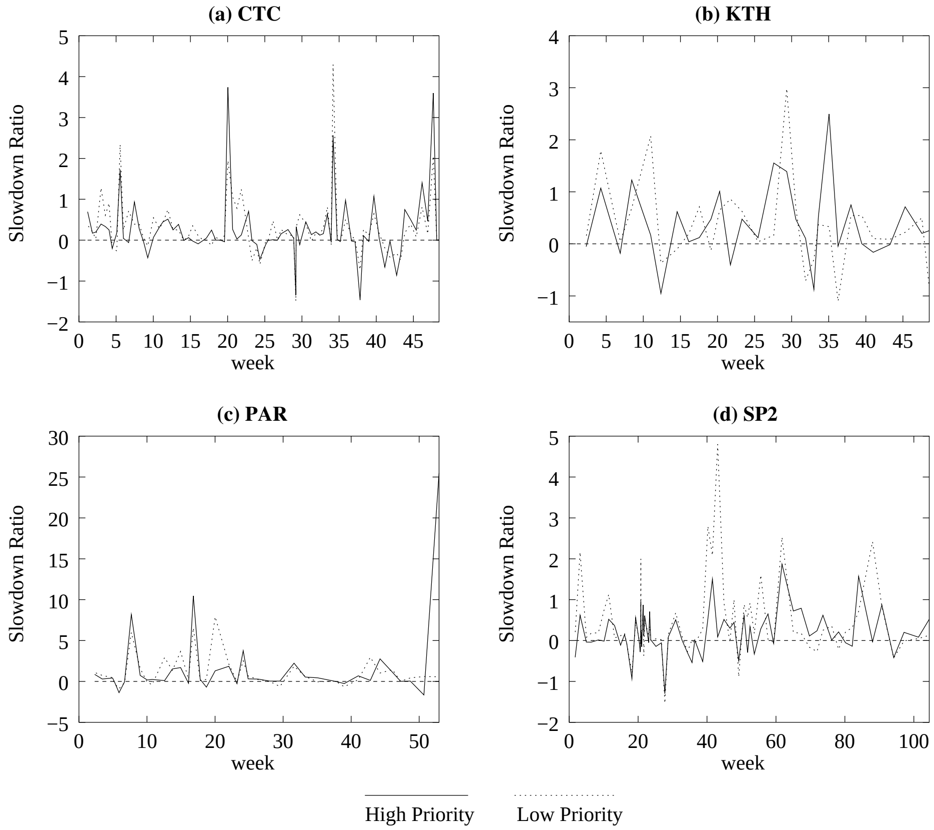

Fig. 7. Slowdown ratio $\mathcal{R}$ per 1000 job submissions as a function of time for high priority and low priority jobs for each of the four traces with 75% high priority jobs

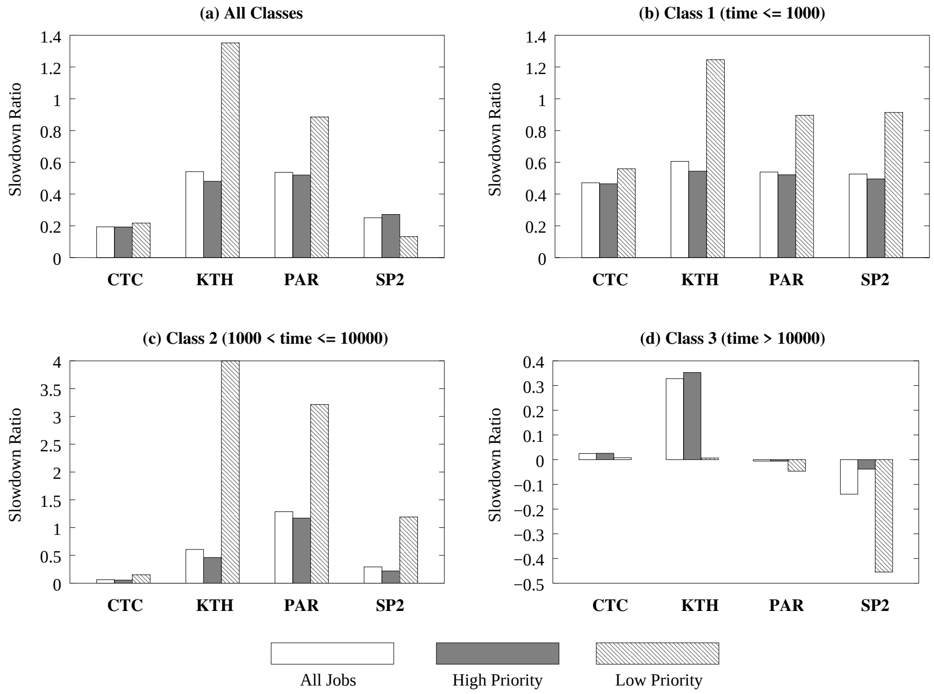

Fig. 8. Overall and per-class aggregate slowdown ratio $\mathcal{R}$ for each of the four traces with 95% high priority jobs

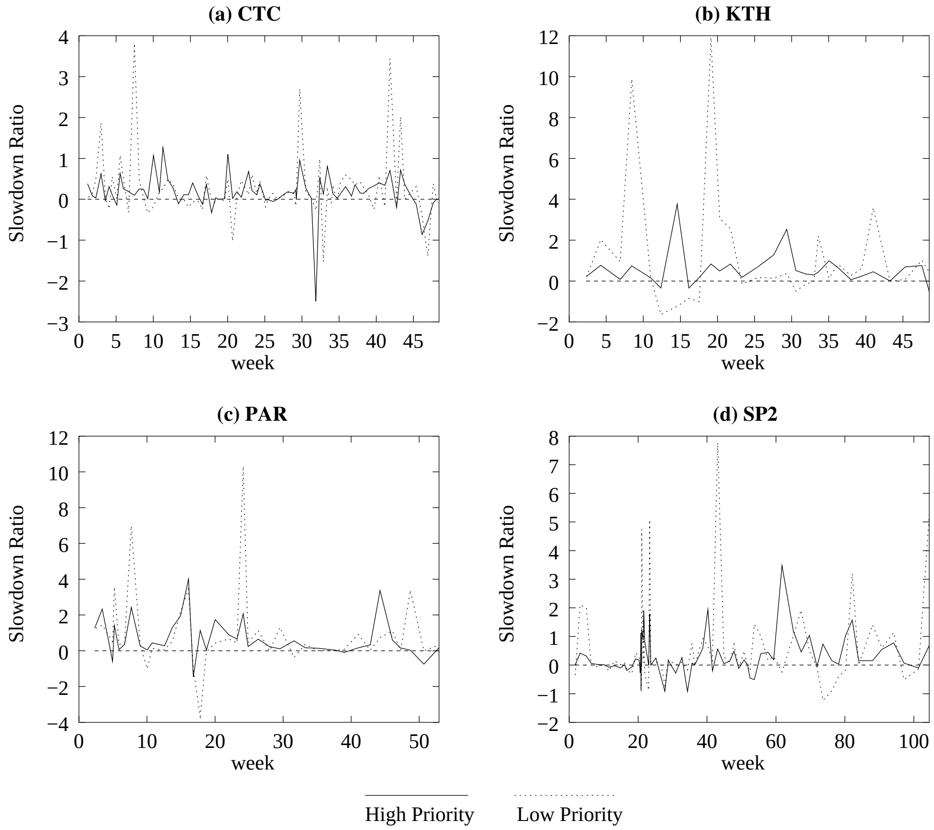

Fig. 9. Slowdown ratio $\mathcal{R}$ per 1000 job submissions as a function of time for high priority and low priority jobs for each of the four traces with 95% high priority jobs

that the remaining 25% have low priority. Figure 6 depicts the corresponding aggregate slowdown ratio $\mathcal{R}$ of multiple-queue backfilling relative to single-queue backfilling for each of the four traces. Figure 6(a) shows $\mathcal{R}$ for all job classes combined, while Figures 6(b)–(d) each show $\mathcal{R}$ for an individual job class. For each trace, we also provide $\mathcal{R}$ as computed for high priority jobs, for low priority jobs, and for both priorities combined. As shown in this figure, multiple-queue backfilling provides better average job slowdown than single-queue backfilling for all job classes combined (Figure 6(a)). Also note that, with the exception of the long job class (Figure 6(d)), multiple-queue backfilling tends to perform better within each of the individual job classes. Again, because multiple-queue backfilling assists shorter jobs at the expense of long jobs, a relatively small decline in the performance of the long job class is unavoidable.

We also consider the transient performance of multiple-queue backfilling under job priorities. Figure 7 depicts transient snapshots of the slowdown ratio versus time for each of the four traces with 75% high priority jobs. Each figure shows slowdown ratio snapshots for 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 a majority of the time corresponding to performance gains with multiple-queue backfilling.

We also consider a system in which only 5% of the submissions are external, i.e., 95% of the jobs have high priority. Figures 8 and 9 are analogous to Figures 6 and 7 but with 95% high priority jobs. Again, we see that improved job slowdown is achieved using multiple-queue backfilling for all job classes combined and, with the exception of the long jobs in the two SDSC workloads, for the individual job classes. Also note the larger vertical axis scales in Figure 8 corresponding to even larger performance gains than with 75% high priority jobs.

Notice that in Figures 6 and 8, multiple-queue backfilling often achieves a greater performance improvement for low priority jobs than for high priority jobs. By its very nature, a high priority job will receive preferential treatment in either single-queue or multiple-queue backfilling. However, by appropriately separating jobs, multiple-queue backfilling is able to assist low priority jobs as well, thereby achieving performance improvements over single-queue backfilling for high *and* low priority jobs.

3.4 Performance Under Reservations

We now consider a system incorporating reservation requests. For each of the four traces, Figure 10 depicts the average job slowdown for all classes combined with proportions of 0.01, 0.05, and 0.25 of the total jobs requesting reservations. As shown, multiple-queue backfilling provides better average job slowdown for the 0.01 and 0.05 proportions, and provides comparable slowdown for the 0.25 proportion. In addition, Table 2 shows the number of missed reservations for single- and multiple-queue backfilling for each of the four traces with proportions of 0.01, 0.05, and 0.25 of the total jobs requesting reservations. Note that single-queue and multiple-queue backfilling miss roughly the same number of reservations. For a proportion of 0.25 of the total jobs requesting reservations, Figure 11 depicts for each trace the tail of the distribution of delays experienced by jobs requesting reservations. As shown, multiple-queue and single-queue backfilling achieve roughly the same distribution for reservation delays. Although we cannot claim significant improvement relative to the number of missed reservations and the distribution of reservation delays, multiple-queue backfilling performs as well as single-queue backfilling.

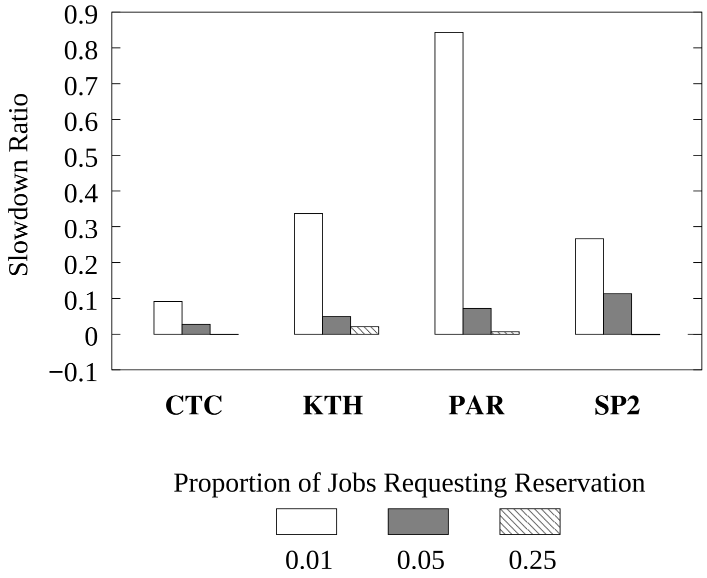

Fig. 10. Overall aggregate slowdown ratio $\mathcal{R}$ for each of the four traces with proportions of 0.01, 0.05, and 0.25 of the total jobs requesting reservations

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. For the partitioning criteria, we assume users overestimate the job run time. We also incorporate speculative execution to combat the detrimental effect of crashed jobs on the policy.

The proposed policy consistently outperforms single-queue backfilling, even under heavy load. The performance gains are a direct result of the fact that the multiple-queue policy significantly reduces the likelihood that a short job is

Table 2. Number of missed reservations for single- and multiple-queue backfilling with proportions of 0.01, 0.05, and 0.25 of the total jobs requesting reservations

Workload	Proportion of Reservations	Number of Reservations	Single-Queue Missed	Multiple-Queue Missed
CTC	0.01	761	19	13
	0.05	3908	90	90
	0.25	19 897	1396	1441
KTH	0.01	273	9	12
	0.05	1421	45	38
	0.25	7178	567	546
PAR	0.01	374	5	1
	0.05	1873	7	9
	0.25	9543	138	119
SP2	0.01	652	54	46
	0.05	3349	294	250
	0.25	16 968	4051	3983

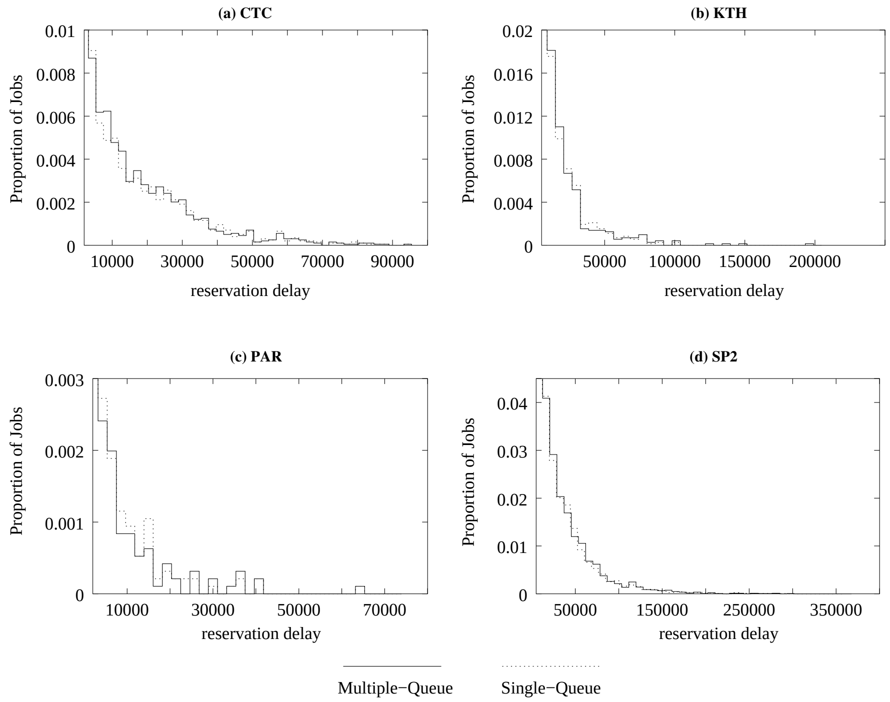

Fig. 11. Distribution tails of the delays experienced by jobs requesting reservations with a proportion of 0.25 of the total jobs requesting reservations

overly delayed in the queue behind a very long job. Multiple-queue backfilling also yields prominent performance gains when jobs with different priorities are considered, and performs as well as single-queue backfilling when reservations are considered.

Acknowledgments

We thank Tom Crockett and Daniela Puiu for useful discussions that contributed to this work. We also thank Dror Feitelson for the availability of workload traces through the Parallel Workload Archive. Thanks also to the anonymous referees for their helpful comments and suggestions.

References

1. Feitelson, D.G.: A Survey of Scheduling in Multiprogrammed Parallel Systems. Technical Report RC 19790, IBM Research Division, October 1994.
2. Parallel Workload Archive: `http://www.cs.huji.ac.il/labs/parallel/workload/`.
3. IBM LoadLeveler: `http://www.ibm.com/`.
4. Keleher, P., Zotkin, D., and Perkovic, D.: Attacking the Bottlenecks in Backfilling Schedulers. Cluster Computing: The Journal of Networks, Software Tools and Applications. **3**(4) (2000) 245–254.
5. Lawson, B.G., Smirni, E., and Puiu D.: Self-Adapting Backfilling Scheduling for Parallel Systems. Proceedings of the International Conference on Parallel Processing
6. Maui Scheduler Open Cluster Software: `http://mauischeduler.sourceforge.net/`.
7. Müalem, A. and Feitelson, D.G.: Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. IEEE Transactions on Parallel and Distributed Systems. **12**(6) (2001) 529–543.
8. Portable Batch System: `http://www.openpbs.org/`.
9. Perkovic, D. and Keleher, P.: Randomization, Speculation, and Adaptation in Batch Schedulers. Proceedings of Supercomputing 2000 (SC2000). November 2000.
10. Talby, D. and Feitelson, D.G.: Supporting Priorities and Improving Utilization of the IBM SP2 Scheduler Using Slack-Based Backfilling. Proceedings of the 13th International Parallel Processing Symposium. (1999) 513–517.