Self-adapting Backfilling Scheduling for Parallel Systems*

Barry G. Lawson
Department of Mathematics and Computer Science
University of Richmond
Richmond, VA 23173, USA
blawson@richmond.edu

Evgenia Smirni, Daniela Puiu
Department of Computer Science
College of William and Mary
Williamsburg, VA 23187-8795, USA
{esmirni,dxpuiu}@cs.wm.edu

Abstract

We focus on non-FCFS job scheduling policies for parallel systems that allow jobs to backfill, i.e., to move ahead in the queue, given that they do not delay certain previously submitted jobs. Consistent with commercial schedulers that maintain multiple queues where jobs are assigned according to the user-estimated duration, we propose a self-adapting backfilling policy that maintains multiple job queues to separate short from long jobs. The proposed policy adjusts its configuration parameters by continuously monitoring the system and quickly reacting to sudden fluctuations in the workload arrival pattern and/or severe changes in resource demands. Detailed performance comparisons via simulation using actual Supercomputing traces from the Parallel Workload Archive indicate that the proposed policy consistently outperforms traditional backfilling.

Keywords: *batch schedulers, parallel systems, backfilling schedulers, performance analysis.*

1. Introduction

In recent years, scheduling parallel programs in multiprocessor architectures has consistently puzzled researchers and practitioners. Parallel systems consist of resources that have to be shared among a community of users. Resource allocation in such systems is a non-trivial problem. Examples of issues that exacerbate the resource allocation problem include the number of users that attempt to use the system simultaneously, the parallelism of the applications and their respective computational and storage needs, the wide variability of the average job execution time coupled with the variability of requested resources (e.g., processors, memory), the continuously changing job arrival rate, meeting the execution deadlines of applications, and co-scheduling distributed applications across multiple independent systems each of which may itself be parallel with its own scheduler.

Many scheduling policies have been developed with the goal of providing better ways to handle the incoming workload by treating interactive jobs differently than batch jobs [1]. Among the various batch schedulers that have been proposed, we distinguish a set of schedulers that allows the system administrator to customize the scheduling policy according to the site's needs. The Maui Scheduler is widely used by the high performance computing community [6] and provides a wide range of configuration parameters that allows for site customization. Similarly, the PBS scheduler [10] operates on networked, multi-platform UNIX environments, including heterogeneous clusters of workstations, Supercomputers, and massively parallel systems, and allows for the implementation of a wide variety of scheduling solutions. Generally, these schedulers maintain several queues (to which different job classes are assigned), permit assigning priorities to jobs, and allow for a wide variety of scheduling policies per queue. The immediate benefit of such flexibility in policy parameterization is the ability to change the policy to better meet the incoming workload demands. Policy customization to meet the needs of an ever changing workload is a difficult task.

We concentrate on 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) that are often found in the heart of many popular parallel workload 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 [5, 13]. Users are expected to provide nearly accurate estimates of the job execution times. Using these estimates, the scheduler rearranges the queue, allowing short jobs to move to the top of the queue provided they do not starve certain previously submitted jobs. Backfilling is ex-

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

Workload	Mean Exec. Time	Median Exec. Time	C.V. Exec. Time	Mean Number Processors	Median Number Processors	C.V. Number Processors
CTC	10,983	946	1.65	11	2	2.26
KTH	8,876	845	2.19	8	3	1.61
SP2	6,118	514	2.37	10	4	1.59
PAR	7,416	175	2.01	16	8	1.46

Table 1. Summary statistics of the four selected workloads. All times are reported in seconds.

tensively used by many schedulers, most notably the IBM LoadLeveler scheduler [4] and the Maui Scheduler [6]. Various versions of backfilling have been proposed [3, 5, 9]. [3] characterizes the effect of job length and parallelism on backfilling performance and [9] proposes sorting by job length to improve backfilling.

In this paper, we propose a batch scheduler that is based on the *aggressive* backfilling strategy extensively analyzed in [5]. In contrast to all of the above backfilling-related works, we maintain multiple queues and *separate* effectively long from short jobs. 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 [11, 12]. Similarly, observed high variance in job execution times in parallel workload traces advocates separating long from short jobs in parallel schedulers.

Our multiple-queue policy assigns incoming jobs to different queues using user estimates of the job execution times. 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. Furthermore, our policy modifies the subsystem boundaries on the fly according to the incoming workload intensities and execution demands. By continuously monitoring the scheduler's ability to handle the incoming workload, the policy adjusts its parameters to guarantee high system utilization and throughput while improving the average job slowdown.

We conduct a set of simulation experiments using trace data from the Parallel Workload Archive [7]. The traces offer a rich set of workloads taken from actual Supercomputing centers. Detailed workload characterization, focusing on how the job arrivals and resource demands *change across time*, guides us into the development of a robust policy that performs well under transient workload conditions.

This paper is organized as follows. Section 2 contains a characterization of the workloads used to drive our simulations. Section 3 presents the proposed policy. Detailed performance analysis of the proposed policy is given in Section 4. Concluding remarks are given in Section 5.

2. Variability in Workloads

The difficulty of scheduling parallel resources is deeply interwoven with the inherent variability in parallel workloads. Because our goal is to propose a robust policy that works efficiently regardless of the workload type, we first closely examine real parallel workloads of production systems. We select four workload logs from the parallel workload archive [7]. Each log provides the arrival time of each job (i.e., the job submit time), 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 Oct. 1996 to Aug. 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.
- **SP2**: This trace contains entries for 67 667 jobs executed on a 128-node IBM SP2 at the San Diego Supercomputing Center from May 1998 to April 2000.

Table 1 provides summary statistics for the selected traces [1]. Observe the wide disparity of the mean job execution time across workloads. Also notice the difference (of as much as two orders of magnitude) between the mean and the median within a workload. The high coefficients of variation (C.V.) in job execution times coupled with the large differences between mean and median values suggest the existence of a "fat tail" in the distribution of execution times. Log-log complementary distribution plots confirm the absence of a

[1] A common characteristic in many of these traces is that the system administrator places an upper limit on the job execution time. If this limit is reached, the job is killed. Our statistics include the terminated jobs; therefore, some of our output statistics are higher than those reported elsewhere (e.g., see [2]).

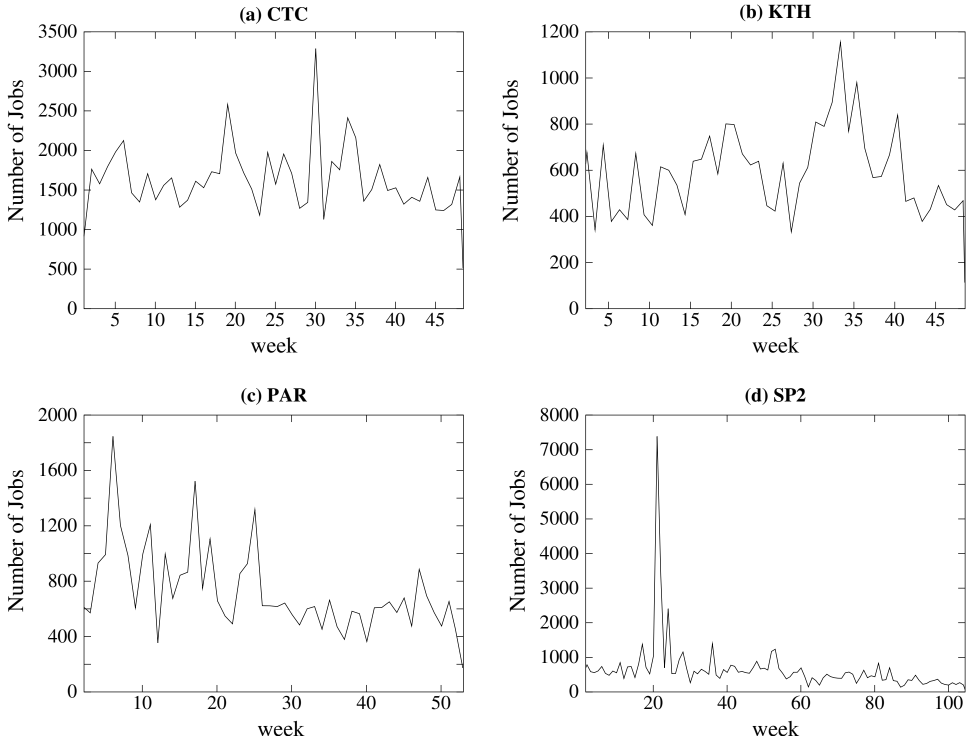

Figure 1. Total number of arriving jobs per week as a function of time (weeks).

heavy tail in the distributions [2], but run times nonetheless remain very skewed within each workload. This type of distribution advocates separating jobs according to their duration to different queues in order to minimize queuing time of short jobs that are delayed behind very long jobs.

Significant variability is also observed in the average "width" of each job, i.e., the number of per-job requested processors. To determine whether job duration and job width are independent attributes, we compute their statistical correlation for each workload. Results are mixed. In some cases, positive correlation is detected, while in other cases there is no correlation at all. Because job duration and job width strongly affect the backfilling ability and performance of a policy, we further elaborate on these two metrics later in this section.

The two parameters that affect performance and scheduling decisions in queuing systems are the arrival process and the service process. To visualize the time evolution of the arrival process, we plot for each trace the total number of arriving jobs per week as a function of time (see Figure 1). We observe bursts in the arrival process[2], but not of the same magnitude as the "flash crowds" experienced by web servers. Significant differences in the per-week arrival intensity exist within each workload, as well as across all workloads. For this reason we focus not only on *aggregate* statistics (i.e., the average performance measures obtained after simulating the system using the entire workload trace), but also on *transient* statistics within specific time windows.

We now consider the service process. Because Table 1 indicates wide variation in job service times, we classify jobs according to job duration. After experimenting with several classifications, we choose the following four-part classification. Across all workloads, this classification provides a representative proportion of jobs in each class (see Figure 2).

- **class 1**: Short jobs are those with execution time ≤ 100 seconds.
- **class 2**: Medium jobs are those with execution time > 100 seconds and ≤ 1000 seconds.
- **class 3**: Long jobs are those with execution time $>$ 1000 seconds and $\leq 10\,000$ seconds.
- **class 4**: Extra-long jobs are those with execution time $> 10\,000$ seconds.

Figure 2 presents the service time characteristics of the four workloads. The left column depicts the overall and per-class mean job execution time as a function of the trace time[3].

[2]Bursts also exist relative to smaller time units (e.g., days and hours), but such graphs are omitted for the sake of brevity.

[3]We compute statistics for batches of 1000 jobs, but plot each batch as a function of the arrival time of the first job in the batch.

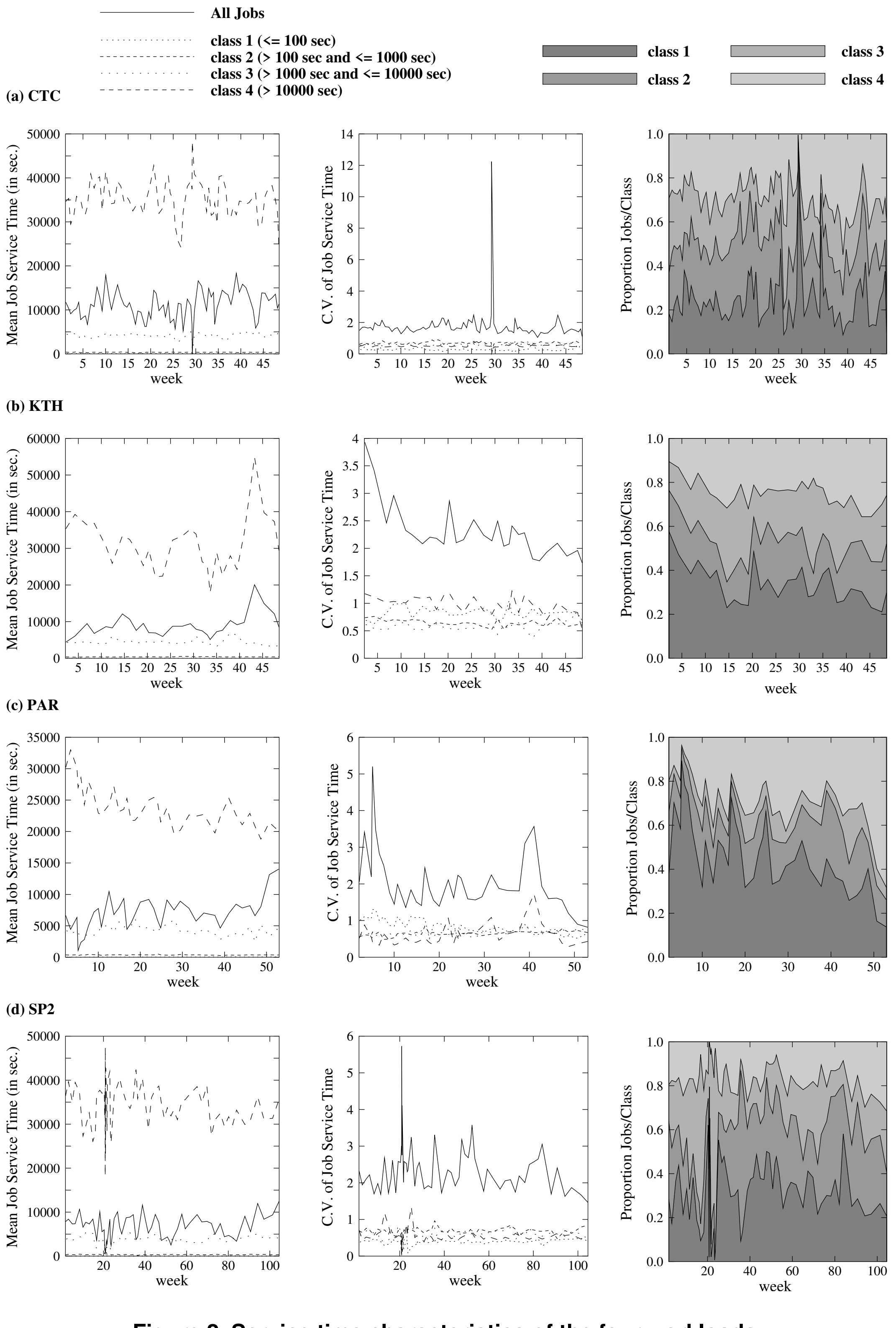

Figure 2. Service time characteristics of the four workloads.

The center column in Figure 2 depicts the overall and per-class C.V. of the average job execution time. Finally, the right column depicts the proportion of jobs per class. Observe that the mean job execution times and the overall C.V. (solid line) vary significantly across time. As expected, for all workloads the per-class C.V. is considerably smaller than the overall C.V. For all traces the proportion of jobs in each class varies dramatically with time.

3. Scheduling Policies

In actual parallel systems, successful scheduling policies use *backfilling*, a non-FCFS approach. Backfilling permits a limited number of queued jobs to jump ahead of jobs that cannot begin execution immediately. Backfilling is a core component of commercial schedulers including the IBM LoadLeveler [4] and the popular Maui Scheduler [6]. Here we propose a new policy, based on backfilling, that adapts its scheduling parameters according to changing workload conditions. Before introducing our new policy, we first describe the basic backfilling paradigm.

3.1. Single-Queue Backfilling

Backfilling is a commonly used scheduling policy that attempts to minimize fragmentation of system resources by executing jobs in an order different than their submission order [3, 5]. A job that is backfilled is allowed to jump ahead of jobs that arrived earlier (but are delayed because of insufficient idle processors) in an attempt to exploit otherwise currently idle processors. The order of job execution is handled differently by two types of backfilling. *Conservative* backfilling permits a job to be backfilled provided it does not delay *any* previous job in the queue. *Aggressive* backfilling ensures only that the *first* job in the queue is not delayed. We consider aggressive backfilling as the baseline policy because results have shown its performance superior to conservative backfilling [5].

Basic aggressive backfilling assumes a single queue of jobs to be executed. Jobs enter this queue when submitted by the user. Each job is characterized by its arrival time, by the number of processors required (i.e., the job width), and by an estimate of the expected execution time. Aggressive backfilling is a non-preemptive, space-sharing policy. Any job that attempts to execute for a time greater than its estimated execution time is terminated by the system.

The single-queue backfilling policy always attempts to backfill as many queued jobs as possible. In general, the process of backfilling *exactly one* of these many jobs occurs as follows. Define the *pivot job* to be the first job in the queue. If there are currently idle processors sufficient for the pivot job, the scheduler starts executing the pivot immediately, and a new pivot is defined appropriately. Otherwise, the scheduler sorts all currently executing jobs in order of their expected completion time. The scheduler can then determine the *pivot time*, i.e., the time when sufficient processors will be available for the pivot job. At the pivot time, any idle processors not required for the pivot job are denoted as *extra* processors. The scheduler then searches for the first queued job that

- requires no more than the currently idle processors *and* will finish by the pivot time, or
- requires no more than the minimum of the currently idle processors and the extra processors.

If such a job is found, the job is backfilled, i.e., the scheduler starts executing the job immediately; otherwise, the scheduler continues searching the list of queued jobs until either a job is backfilled or the search is exhausted.

This process of backfilling exactly one job is repeated until all queued jobs have been considered for backfilling. Hence, the single-queue backfilling policy attempts to backfill as many jobs as possible until no more jobs can be backfilled. This basic single-queue aggressive backfilling algorithm, employed whenever a job is submitted to the system or whenever a job completes execution, is outlined in Figure 3.

Single-queue aggressive backfilling ensures that once a job becomes the pivot, it cannot be delayed. A job may be delayed in the queue before becoming the pivot, but when the job reaches the front of the queue, it is assigned a scheduled starting time. If a currently executing job finishes early, the pivot may begin executing earlier than its assigned starting time, but it will *never* begin executing after the assigned starting time.

3.2. Multiple-Queue Backfilling

Because the performance of any scheduling policy is sensitive to the transient nature of the impending workload, we propose a multiple-queue backfilling policy that permits the scheduler to quickly change parameters in response to workload fluctuations. Our goal is to decrease the average job slowdown by reducing the number of short jobs delayed by longer jobs.

The multiple-queue backfilling policy splits the system into multiple *disjoint* partitions. The splitting is accomplished by classifying jobs according to the job duration as described in Section 2. We incorporate four separate queues, one per job class (i.e., per system partition), indexed by $q = 1, 2, 3, 4$. As jobs are submitted to the system, they are assigned to exactly one of these queues based on the user estimate of execution time. Let t_e be the estimate (in seconds) of the execution time of a submitted job. Here, we

for (all jobs in queue)
1. *pivot job* $\longleftarrow$ first job in queue
2. if possible, start pivot job immediately
3. else
  a. sort running jobs in order of completion time
  b. *pivot time* $\longleftarrow$ time when sufficient processors will be available for pivot job
  c. *extra procs* $\longleftarrow$ idle processors at pivot time not used by pivot job
  d. while (no job backfilled <u>and</u> more queued jobs to consider)
    I. consider next job in queue
    II. if job requires $\leq$ currently idle procs <u>and</u> will finish by pivot time, start job immediately
    III. else if job requires $\leq$ min$\{$currently idle procs, extra procs$\}$, start job immediately

Figure 3. Basic single-queue aggressive backfilling algorithm.

consider that the user provides *accurate* estimates of the expected execution time[4]. The job is assigned to the queue in partition q according to the following equation, consistent with the job classification presented in Section 2.

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

Note that the assignment of a job to a queue is based solely on the user estimate of job execution time and *not* on the number of requested processors. Initially, the processors are distributed evenly among the four partitions. As time evolves, processors may move from one partition to another (i.e., the partitions may contract or expand) so that currently idle processors in one partition can be used for immediate backfilling in another partition. Hence, the partition boundaries become dynamic, allowing the system to adapt itself to changing workload conditions. We stress that the policy does not starve a job that requires the entire machine for execution. When such a job is ready to begin execution (according to the job arrival order), the scheduler allocates all processors to the partition where the job is assigned. After the job completes, the processors will be redistributed among the four partitions according to the ongoing processor demands of each partition.

The multiple-queue backfilling policy considers all queued jobs (one at a time, in the order of arrival across all queues). Similar to the single-queue backfilling policy, define the following:

- idle$_q$: the number of currently idle processors in partition q ;
- pivot$_q$: the first job in the queue in partition q ;
- pivot-time$_q$: the scheduled starting time for pivot$_q$ (i.e., the earliest time when sufficient processors will be available for pivot$_q$) ;
- extra$_q$: the number of idle processors in partition q at pivot-time$_q$ not required for pivot$_q$.

The sufficient processors available at pivot-time$_q$ consist of idle$_q$ and, if necessary, some combination of idle and/or extra processors from other partitions such that no other pivot that arrived earlier than pivot$_q$ is delayed. The assignment of a scheduled starting time to a pivot job will never delay any current pivot in another partition (i.e., any other pivot that arrived earlier), suggesting that the algorithm is deadlock free.

The policy always attempts to backfill as many queued jobs as possible. In general, *exactly one* of these many jobs is backfilled as follows. Let q be the queue where the job resides. If the job is pivot$_q$, the scheduler starts executing the job immediately only if the current time is equal to pivot-time$_q$, in which case a new pivot$_q$ is defined appropriately. If the job is *not* pivot$_q$, the scheduler starts executing the job immediately only if there are sufficient idle processors in partition q without delaying pivot$_q$, or if the partition can take idle processors sufficient to meet the job's requirements from one or more other partitions without delaying any pivot.

This process of backfilling one job is repeated, one job at a time in the order of arrival across all queues, until all queued jobs have been considered for backfilling. Hence, the multiple-queue backfilling policy attempts to backfill as many jobs as possible until no more jobs can be backfilled. This multiple-queue aggressive backfilling algorithm, employed whenever a job is submitted to the system or whenever a job completes execution, is outlined in Figure 4.

In both the single-queue and multiple-queue aggressive backfilling policies, the goal is to backfill jobs in order to exploit idle processors and reduce system fragmentation.

[4]For details regarding sensitivity of the policy to inaccurate estimates we refer the interested reader to [8].

for (all jobs in order of arrival)
1. $q \longleftarrow$ queue in which job resides
2. pivot$_q$ $\longleftarrow$ first job in queue q
3. pivot-time$_q$ $\longleftarrow$ earliest time when sufficient procs (from this and perhaps other partitions) will be available for pivot$_q$
4. extra$_q$ $\longleftarrow$ idle processors in partition q at pivot-time$_q$ not used by pivot$_q$
5. if job is pivot$_q$
  a. if current time equals pivot-time$_q$
    I. if necessary, reassign procs from other partitions to partition q
    II. start job immediately
6. else
  a. if job requires $\leq$ idle$_q$ and will finish by pivot-time$_q$, start job immediately
  b. else if job requires $\leq$ min$\{$idle$_q$, extra$_q\}$, start job immediately
  c. else if job requires $\leq$ (idle$_q$ 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 q
    II. start job immediately

Figure 4. Multiple-queue aggressive backfilling algorithm.

Both policies ensure that once a job reaches the front of the queue, it cannot be delayed further.

By classifying jobs according to job length, the multiple-queue policy reduces the likelihood that a short job will be overly delayed in the queue behind a very long job. Additionally, because processors are permitted to cross partition boundaries, the multiple-queue policy can quickly adapt to a continuously changing workload. Unlike commercial schedulers that typically are difficult to parameterize, multiple-queue backfilling requires only an *a priori* definition of job classes, and then the policy automatically adjusts the processor-to-class allocations. In the following section, we elaborate on the above issues and their effect on performance.

4. Performance Analysis

We evaluate and compare via simulation the performance of the two backfilling policies presented in the previous section. Our simulation experiments are driven using the four workload traces from the Parallel Workload Archive described in Section 2. From each trace record, we extract three values: the job arrival time, the job execution time, and the number of requested processors. Consequently, our experiments fully capture the fluctuations in the average job arrival rate and service demands.

We concentrate both on *aggregate* performance measures, i.e., measures collected at the end of the simulation that reflect the average achieved performance across the entire life of the system, and on *transient* measures, i.e., the average performance measures perceived by the end-user during each time interval corresponding to 1000 job requests[5].

The performance measure of interest that we strive to optimize is the average job slowdown s defined by

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

where d and ν are respectively the queuing delay time and actual service time of a job[6].

To compare the single-queue and multiple-queue backfilling results, we define the *slowdown ratio* $\mathcal{R}$ by the equation

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

where s_1 and s_m are the single-queue and multiple-queue slowdowns respectively[7]. $\mathcal{R} > 0$ indicates the performance gain obtained using multiple queues relative to a single queue. $\mathcal{R} < 0$ indicates the performance loss that results from using multiple queues relative to a single queue.

[5] Consistent with Section 2, we use a batch size of 1000 for reasons of statistical significance.

[6] Bounded slowdown [5] is another popular performance measure. For the sake of brevity, we omit performance results for bounded slowdown that we obtained. Note that the performance of each of the two policies is qualitatively the same using *either* of the two measures.

[7] Because of the $\min\{s_1, s_m\}$ term in the denominator, $\mathcal{R}$ is a *fair*, properly scaled measure of the performance that equally quantifies gain or loss experienced using multiple queues relative to a single queue. If we instead use s_m (or for the same matter s_1) in the denominator, we bias the measure toward gains (or losses).

4.1. Multiple-Queue Versus Single-Queue Backfilling

Figure 5 depicts the aggregate slowdown ratio $\mathcal{R}$ of multiple-queue backfilling relative to single-queue backfilling for each of the four traces. Figures 5(b)-(e) depict the aggregate *per-class* slowdown ratios (i.e., for short, medium, long, and extra-long jobs). The figure clearly indicates that the multiple-queue algorithm offers dramatic performance gains for all but the extra-long job class. For overall average slowdown (see Figure 5(a)), the multiple-queue policy is superior to the single-queue policy.

Figures 5(b)–(e) confirm that, by splitting the system into multiple partitions, we manage to reduce the number of short jobs overly delayed behind extra-long jobs. Across all workloads, jobs belonging to all but the extra-long job class achieve significant performance gains. Additionally, extra-long jobs experience a decline in average slowdown, but the magnitude of decline is generally much less than the magnitude of improvement seen in the other job classes.

Transient measures illustrate how well each policy responds to sudden arrival bursts. Furthermore, transient measures reflect the end-user perception of system performance, i.e., how well the policy performs during the relatively small window of time that the user interacts with the system. Figure 6 displays transient snapshots of the slowdown ratio versus time for each of the four traces. For all traces, marked improvement (i.e., $\mathcal{R} > 0$) in slowdown is achieved using the multiple-queue backfilling policy. Although the single-queue policy gives better slowdown (i.e., $\mathcal{R} < 0$) for a relatively few batches, multiple-queue backfilling excels with more frequent and larger improvements.

4.2. Multiple-Queue Backfilling with Delays

Because the decline in average slowdown for extra-long jobs (Figure 5(e)) is typically much less than the improvement for all job classes combined (Figure 5(a)), a natural extension to multiple-queue backfilling is to further impede extra-long jobs. Therefore, we hinder any extra-long job by assigning to it a delay when submitted to the system. Let D be the global delay (in seconds) and let t_s be the time of submission of an extra-long job; the job can begin execution no earlier than $t_s + D$. The goal is to further assist shorter jobs in an attempt to improve the overall average slowdown. To address policy flexibility, we adjust the delay parameter on the fly according to the current perceived performance. By continuously monitoring the average job slowdown of each job class, the policy simply increments or decrements the delay parameter accordingly. Our goal is to increase the delay on extra-long jobs only when short jobs are suffering, and to reduce the delay when short jobs are overly favored.

More specifically, for batches of 100 completed jobs, we monitor the average slowdown of short jobs in each batch. Let D_k be the global variable delay imposed on extra-long jobs for the kth batch ($k = 1, 2, \ldots$), where D_1 is the *initial* delay. Let s_k represent the average slowdown of short jobs in the kth batch, and let δ_k represent the proportional difference in s_k and s_{k-1} according to the equation

$$\delta_k = \frac{s_k - s_{k-1}}{\max\{s_k, s_{k-1}\}} \qquad \text{for } k > 1$$

with $\delta_1 = s_1$. To avoid too frequent modifications, the delay for batch $k+1$ is modified only if the proportional difference δ_k is more than 0.25 (i.e., if the difference in average slowdown for short jobs from the previous batch to the current batch changes by more than 25%). If so, we change the global delay by an amount equal to the proportional difference multiplied by the original delay[8]; otherwise, the global delay remains unchanged for the next batch. To summarize, the adjusted delay used for batch $k+1$ is computed via the following algorithmic steps.

1. compute δ_k
2. if $|\delta_k| > 0.25$, then $D_{k+1} = \max\{D_k + \delta_k D_1, 0\}$
3. else $D_{k+1} = D_k$

Figure 7 again depicts the aggregate slowdown ratio $\mathcal{R}$ for each of the four traces. For each trace, we show the gain/loss obtained using multiple-queue backfilling with no delay and with variable delay using $D_1 = 2500$. In all cases, multiple-queue backfilling with variable delay clearly surpasses single-queue backfilling (i.e., $\mathcal{R} \gg 0$).

5. Conclusions

We presented a self-adapting, multiple-queue backfilling policy for parallel systems that directs incoming jobs to different queues according to the user estimated job execution time. By separating short from long jobs, the multiple-queue policy reduces the likelihood that a short job is overly delayed in the queue behind a very long job, and therefore significantly improves the expected job slowdown. Each queue is assigned a non-overlapping partition of system resources on which jobs from the queue can execute. The proposed policy changes the partition boundaries to adapt to evolution of the workload across time.

Multiple-queue backfilling uses minimal parameterization. The policy only requires an *a priori* definition of job classes that regulates the assignment of jobs to queues. This definition of job classes can be easily changed as the system administrator deems appropriate. Furthermore, because of the dynamic nature of the partition boundaries, these external parameters should seldom require modification. Detailed performance comparisons via simulation using actual

[8] Clearly, D_{k+1} must be non-negative.

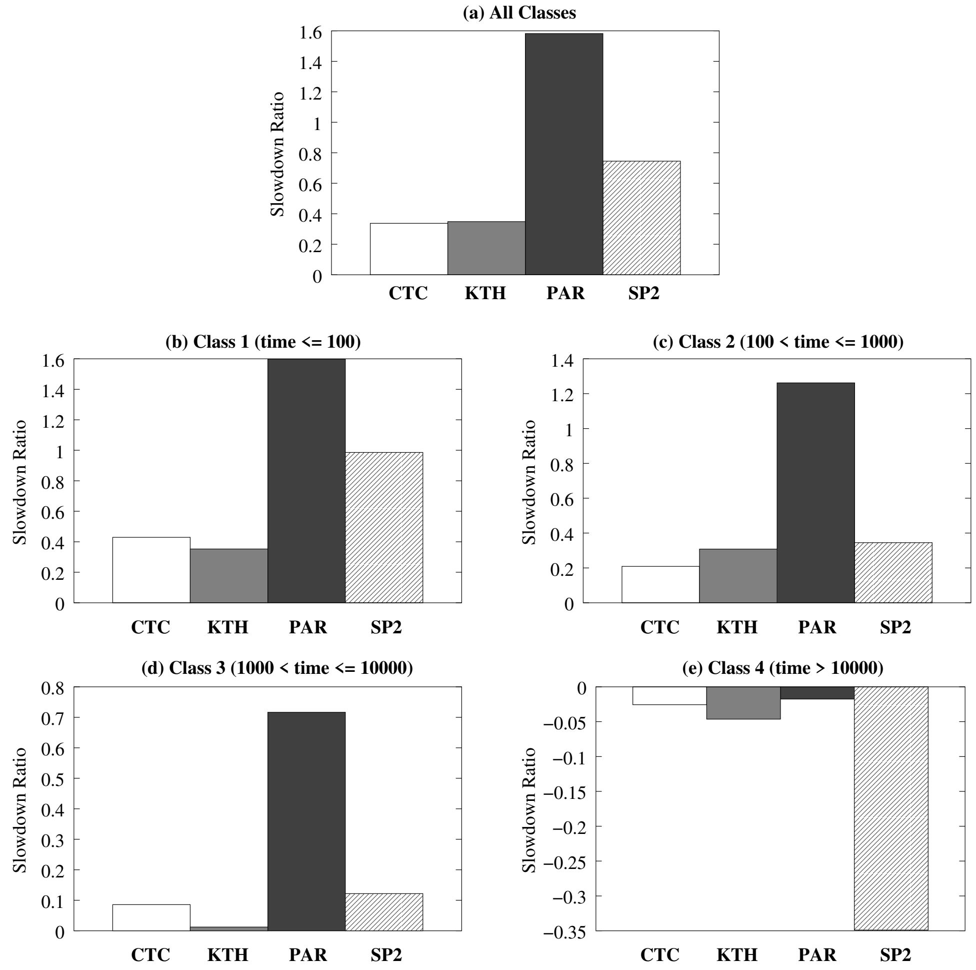

Figure 5. Overall and per-class aggregate slowdown ratio $\mathcal{R}$ for each of the four traces.

Supercomputing traces from the Parallel Workload Archive indicate that the proposed policy consistently outperforms traditional single-queue backfilling. Because of its robustness, simplicity, flexibility, and applicability to ever changing workloads, multiple-queue backfilling is an attractive policy for scheduling parallel resources.

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

[2] Feitelson D.G., "Metrics for Parallel Job Scheduling and Their Convergence", in *Proceedings of the 7th Workshop on Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (Eds.), pp. 188-206, Springer-Verlag, 2001. Lecture Notes in Computer Science Vol. 2221.

[3] 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.

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

[5] 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.

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

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

[8] B.G. Lawson, E. Smirni, "Multiple-queue Backfilling Scheduling with Priorities and Reservations for Parallel Systems", *submitted for publication*, May 2002.

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

[10] Portable Batch System, `http://www.openpbs.org/`.

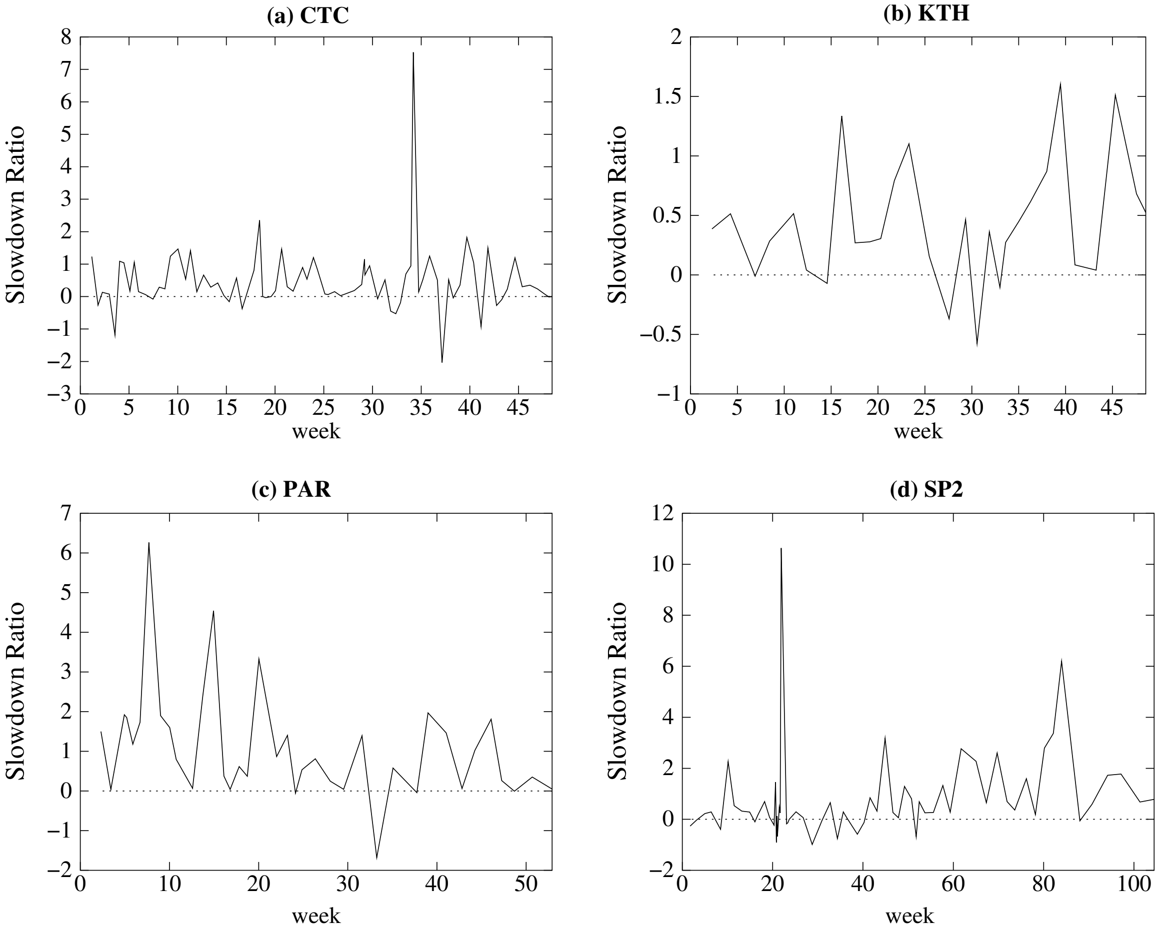

Figure 6. Slowdown ratio $\mathcal{R}$ per 1000-job submissions as a function of time for each of the four traces.

[11] 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.

[12] 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.

[13] 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.

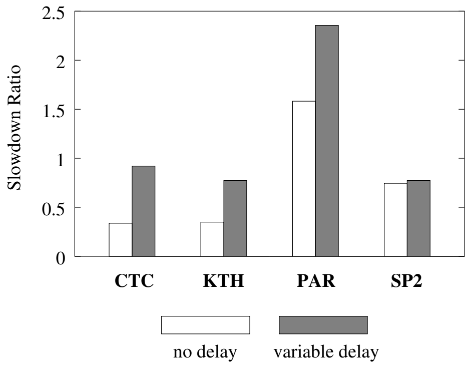

Figure 7. Aggregate slowdown ratio $\mathcal{R}$ using multiple-queue backfilling with no delay and variable delay. All slowdown ratios are computed relative to single-queue backfilling.