Supporting Real-Time Jobs on the IBM Blue Gene/Q: Simulation-Based Study

Daihou Wang[1], Eun-Sung Jung[2], Rajkumar Kettimuthu[3], Ian Foster[3,4], David J. Foran[5], and Manish Parashar[1]

[1] Rutgers Discovery Informatics Institute, Rutgers University, Piscataway, NJ. USA
[2] Hongik University, South Korea
[3] MCS Division, Argonne National Laboratory, Lemont, IL. USA
[4] Department of Computer Science, University of Chicago, Chicago, IL. USA
[5] Rutgers Cancer Institute of New Jersey, New Brunswick, NJ. USA

Abstract. As the volume and velocity of data generated by scientific experiments increase, the analysis of those data inevitably requires HPC resources. Successful research in a growing number of scientific fields depends on the ability to analyze data rapidly. In many situations, scientists and engineers want quasi-instant feedback, so that results from one experiment can guide selection of the next or even improve the course of a single experiment. Such real-time requirements are hard to meet on current HPC systems, which are typically batch-scheduled under policies in which an arriving job is run immediately only if enough resources are available, and is otherwise queued. Real-time jobs, in order to meet their requirements, should sometimes have higher priority than batch jobs that were submitted earlier. Hence, accommodating more real-time jobs will negatively impact the performance of batch jobs, which may have to be preempted. The overhead involved in preempting and restarting batch jobs will, in turn, negatively impact system utilization. Here we evaluate various scheduling schemes to support real-time jobs along with the traditional batch jobs. We perform simulation studies using trace logs of Mira, the IBM BG/Q system at Argonne National Laboratory, to quantify the impact of real-time jobs on batch job performance for various percentages of real-time jobs in the workload. We present new insights gained from grouping the jobs into different categories and studying the performance of each category. Our results show that real-time jobs in all categories can achieve an average slowdown less than 1.5 (most categories achieve an average slowdown close to 1 with at most 20% increase in average slowdown for some categories of batch-jobs (average slowdown for batch jobs in other categories decrease) with 20% or fewer real-time jobs.

Keywords: Real-time job scheduling; preemptive scheduling; scheduler simulation; supercomputing

1 Introduction

Scientific instruments such as accelerators, telescopes, light sources, and colliders generate large amounts of data. Because of advances in technology, the rate and size of these data are rapidly increasing. Advanced instruments such as state-of-the-art detectors at light source facilities generate tens of terabytes of data per day, and future camera-storage bus technologies are expected to increase data rates by an order of magnitude or more. The ability to quickly perform computations on these data sets will improve the quality of science. A central theme in experimental and observational science workflow research is the need for quasi-instant feedback, so that the result of one experiment can guide selection of the next. Online analysis so far has typically been done by dedicated compute resources available locally at the experimental facilities. With the massive increase in data volumes, however, the computational power required to do the fast analysis of these complex data sets often exceeds the resources available locally. Hence, many instruments are operated in a blind fashion without quick analysis of the data to give insight into how the experiment is progressing.

Large-scale high-performance computing (HPC) platforms and supercomputing are required in order to do on-demand processing of experimental data. Such processing will help detect problems in the experimental setup and operational methods early on and will allow for adjusting experimental parameters on the fly [41]. Even slight improvements can have far-reaching benefits for many experiments. However, building a large HPC system or supercomputer dedicated for this purpose is not economical, because the computation in these facilities typically is relatively small compared with the lengthy process of setting up and operating the experiments.

We define real-time computing as the ability to perform on-demand execution. The real-time computation may represent either analysis or simulation. Recently NERSC set up a "real-time" queue on its new Cori supercomputer to address real-time analysis needs. It uses a small number of dedicated compute nodes to serve the jobs in the real-time queue, and it allows jobs in the real-time queue to take priority on other resources. It is also possible to preempt "killable" jobs on these other resources.

NERSC is an exception, however. The operating policy of most supercomputers and scientific HPC systems is not suitable for real-time computations. The systems instead adopt a batch-scheduling model where a job may stay in the queue for an indeterminate period of time. Thus, existing schedulers have to be extended to support real-time jobs in addition to the batch jobs. The main challenge of using supercomputers to do real-time computation is that these systems do not support preemptive scheduling. A better understanding of preemptive scheduling mechanisms is required in order to develop appropriate policies that support real-time jobs while maintaining the efficient use of resources.

In this paper, we present our work on evaluating various scheduling schemes to support mixes of real-time jobs and traditional batch jobs. We perform simulation studies using trace logs of Mira, the IBM BG/Q system at Argonne National Laboratory, to quantify the impact of real-time jobs on batch job performance and system utilization for various percentages of real-time jobs in the workload. Parallel job scheduling has been widely studied [14, 13, 15]. It includes strategies such as backfilling [25, 33, 23, 35], preemption [26, 19], moldability [29, 9, 36] malleability [5], techniques to use distributed resources [28, 37], mechanisms to handle fairness [30, 40], and methods to handle inaccuracies in user run-time estimates [39]. Sophisticated scheduling algorithms have been developed that can optimize resource allocation while also addressing other goals such as minimizing average slowdown [16] and turnaround time. We explore several scheduling strategies to make real-time jobs more likely to be scheduled in due time. Although the techniques that we employ are not new, our context and objective are new. Using Mira trace logs, we quantify the impact of real-time jobs on batch job performance for various percentages of real-time jobs in the workload. We present new insights gained from studying the performance of different categories of jobs grouped based on runtime and the number of nodes used. Our results show that real-time jobs in all categories can achieve an average slowdown less than 1.5 (most categories achieve an average slowdown close to 1) with at most 20% increase in average slowdown for some categories of batch jobs (average slowdown for batch jobs in other categories decreases) with 20% or fewer real-time jobs. With 30% real-time jobs, slowdown for real-time jobs in one of the categories goes above 2, but the impact on batch jobs is comparable to the case with 20% real-time jobs. With 40% or more real-time jobs, average slowdown of batch jobs in one of the categories increases by around 90%, and the average slowdown of real-time jobs also goes above 3.

The rest of the paper is organized as follows. Section 2 describes the background on parallel job scheduling, checkpointing, the Mira supercomputer, and the simulator used for our study. In Section 3 we discuss related work, and in Section 4 we give the problem statement. In Section 5 we present the scheduling techniques studied, and in Section 6 we describe the extensions we did to enable real-time scheduling in the Qsim simulator. Section 7 presents the experimental setup and the simulation results of various scheduling techniques. Section 8 provides the conclusions.

2 Background

We provide in this section some background on parallel job scheduling, Mira, Qsim, and checkpointing.

2.1 Parallel Job Scheduling

Scheduling of parallel jobs can be viewed in terms of a 2D chart with time along one axis and the number of processors along the other axis. Each job can be thought of as a rectangle whose width is the user-estimated run time and height is the number of processors requested. The simplest way to schedule jobs is to use the first-come, first-served, (FCFS) policy. If the number of free processors available is less than the number of processors requested by the job at the head of queue, an FCFS scheduler would leave the free processors idle even if there are waiting queued jobs requiring less than the available free processors. Backfilling addresses this issue. It identifies holes in the 2D schedule and smaller jobs that fit those holes. With backfilling, users are required to provide an estimate of the length of the jobs submitted for execution. A scheduler can use this information to determine whether a job is sufficiently small to run without delaying any previously reserved jobs.

2.2 Mira Supercomputer

Mira is a Blue Gene/Q system operated by the Argonne Leadership Computing Facility (ALCF) at Argonne National Laboratory [3]. It was ranked 9th in the 2016 Top500 list, with peak performance at 10,066 TFlop/s. Mira is a 48-rack system, with 786,432 cores. It has a hierarchical structure connected via a 5D torus network. Nodes are grouped into midplanes, each of which contains 512 nodes; and each rack has two midplanes. Partitions on Mira are composed of such midplanes. Thus, jobs on Mira are scheduled to run on partitions that have integer multiples of 512 nodes. The smallest production job on Mira occupies 512 nodes, and the largest job occupies 49,152 nodes. The Cobalt [1] batch scheduler used on Mira is an open-source, component-based resource management tool developed at Argonne. It has been used as the job scheduler on Intrepid (the supercomputer at ALCF before Mira) and is being used in other Blue Gene systems such as Frost at the National Center for Atmospheric Research [2].

2.3 Qsim Simulator

We used the Qsim discrete event simulator [4] because it was designed for the Cobalt scheduler. Job scheduling behavior is triggered by job-submit(Q)/job-start(S)/job-end(E) events. The latest version of Qsim supports three versions of backfilling-based job scheduling policies: first-fit (FF) backfilling, best-fit (BF) backfilling, and shortest-job-first (SJF) backfilling [25]. By design, Qsim supports the simulation only of batch job scheduling. In this study, we extended the Qsim simulator to support real-time job scheduling using a high-priority queue and preemption.

2.4 Checkpointing Applications

Checkpoint and restart mechanisms were first introduced into modern supercomputing systems to provide fault tolerance [12]. Checkpointing is the process of saving a running applications state to nonvolatile storage. The saved state can be used to restart the application from when the last checkpoint was taken. Over the years, these mechanisms have evolved along with the new generations of supercomputing architecture and network developments. Among many variations, major checkpointing approaches can be categorized as either application level or system level [12].

In the application-level approach, checkpointing is done by individual applications (Cornell Checkpoint(pre) Compiler (C3) [31] is an example of this approach and such works are surveyed in [43]). It requires changes to the application code but it can significantly reduce the amount of data that need to be saved for restarting.

In the system-level approach, checkpointing is done outside of applications. Checkpointing can be implemented either in the operating systems (MOSIX [7] and BLCR [11] are examples of this approach) or in the runtime library or system. In this approach, checkpointing is done by copying the applications memory into persistent storage without specific knowledge of application. It does not require any changes to the application.

3 Related Work

Parallel job scheduling has been widely studied [17, 23, 25, 38, 6], and a number of surveys [14, 13, 15] and evaluations [18, 20, 22, 8] have been published. However, not been much work has been done in the context of supporting on-demand jobs on supercomputers that operate in batch-processing mode.

Although preemptive scheduling is universally used at the operating-system level to multiplex processes on single-processor systems and shared-memory multiprocessors, it is rarely used in parallel job scheduling. Studies of preemptive scheduling schemes have focused on their overheads and their effectiveness in reducing average job turnaround time [24, 10, 34, 8, 19, 21].

Others have studied preemptive scheduling for *malleable parallel jobs* [10, 27, 32, 44], in which the number of processors used to execute a job is permitted to vary dynamically over time. In practice, parallel jobs submitted to supercomputer centers are generally rigid; that is, the number of processors used to execute a job is fixed. The work most similar to ours is SPRUCE (Special Priority and Urgent Computing Environment) [42], which investigated mechanisms for supporting *urgent* jobs such as hurricane analysis on HPC resources. The authors define urgent computing jobs as having time critical needs, such that late results are useless. SPRUCE considered only a basic preemptive scheduling scheme with no checkpointing and assumed that urgent jobs are infrequent. Our work differs in terms of both its job model and the scheduling schemes considered. Our job model assumes that jobs with real-time constraints arrive more frequently and that jobs are not total failure even if the job timing requirements are missed. We evaluate more sophisticated preemptive scheduling schemes.

4 Problem Statement

Our goal is to study the impact of accommodating real-time jobs in (batch) supercomputer systems. We consider two kinds of jobs: *batch jobs* and *real-time jobs*. Real-time jobs expect to execute immediately, whereas batch jobs expect best-effort service. We assume that all jobs are rigid: jobs are submitted to run on a specified fixed number of processors. We assume that a certain percentage

($R\%$) of the system workload will be *real-time jobs* and that the rest are *batch jobs*. We study different values of R. We evaluate different scheduling schemes that prioritize real-time jobs over batch jobs in order to meet the expectations of real-time jobs to the extent possible. In addition to performance, we study the impact of various scheduling schemes on system utilization.

5 Scheduling Techniques

We evaluate five scheduling schemes that accommodate real-time jobs in addition to the traditional batch jobs. Detailed description of the schemes is given below.

5.1 High-Priority Queue-Based Scheduling

Real-time jobs are enqueued in a *high-priority queue* (hpQ), whereas batch jobs are enqueued in a *normal queue*. The scheduler gives priority to the jobs in the high-priority queue and blocks all the jobs in the normal queue until all the jobs in the high-priority queue are scheduled.

5.2 Preemptive Real-Time Scheduling

In the preemptive scheduling schemes, if there are not enough resources to schedule a real-time job, the scheduler selects a partition for real-time job that maximizes system utilization, preempts any batch job running on this partition, or its child partitions, and schedule the real-time job. It then resubmits those batch jobs to the normal queue for later restart/resume. The overhead introduced by preemption impacts the jobs that are preempted as well as the system utilization. Checkpointing can help reduce the overhead of preemption, but checkpointing does not come for free. Checkpointing's impact on job runtime and system utilization needs to be accounted for as well. For the preemptive scheduling schemes, $t^j_{ckpt}, t^j_{pre}, ch^j_{ckpt}, ch^{sys}_{ckpt},\ and\ ch^{sys}_{pre}$ capture these overheads. Here t^j_{ckpt} and t^j_{pre} are the additional time incurred for $job\ j$ due to checkpointing overhead and preemption overhead, respectively; ch^{sys}_{ckpt} and ch^{sys}_{pre} are the core-hours lost by the system due to checkpointing overhead and preemption overhead, respectively; and ch^j_{ckpt} is core-hours lost by $job\ j$ due to checkpointing overhead.

PRE-REST: PRE-REST corresponds to preemption and restart of batch jobs. No system- or application-level checkpointing occurs. Thus, the preempted jobs have to be restarted from the beginning. Equations 1 to 5 describe the overhead associated with this scheme.

$$t^j_{ckpt} = 0 \tag{1}$$

$$t^j_{pre} = \sum_{i=1}^{\#preemptions_j} t^j_{used_i} \tag{2}$$

$$ch^{sys}_{ckpt} = 0 \tag{3}$$

$$ch^{sys}_{pre} = \sum_{k\ in\ batch\ jobs} t^k_{pre} * nodes_k \tag{4}$$

$$ch^j_{ckpt} = 0 \tag{5}$$

Here, $\#preemptions_j$ is the number of times *job j* is preempted, $t^j_{used_i}$ is the time *job j* (preempted job) has run in its i^{th} execution, and $nodes_j$ is the number of nodes used by *job j*.

PRE-CKPT-SYS: This scheme corresponds to the system-level checkpoint support. All batch jobs are checkpointed *periodically* by the system (without any application assistance), and the checkpoint data (the process memory including the job context) are written to parallel file system (PFS) for job restart. Batch jobs running on partitions chosen for real-time jobs are killed immediately, and they are resubmitted to the normal queue. When the preempted batch job gets to run again, the system resumes it from the latest checkpoint. The system checkpoint interval ($ckpIntv_{sys}$) is universal for all running batch jobs. Equations 6 to 10 describe the overhead incurred by the preempted jobs (in terms of time) and the system (in terms of core hours).

$$t^j_{ckpt} = \sum_{i=1}^{\lfloor \frac{t^j_{runtime}}{ckpIntv_{sys}} \rfloor} \frac{ckpData^j_i}{bandwidth^{write}_{PFS}} \tag{6}$$

$$t^j_{pre} = \sum_{i=1}^{\#preemptions_j} \frac{ckpData^j_{latest}}{bandwidth^{read}_{PFS}} + ckpTgap^j_i \tag{7}$$

$$ch^{sys}_{ckpt} = \sum_{k\ in\ batch\ jobs} t^k_{ckpt} * nodes_k \tag{8}$$

$$ch^{sys}_{pre} = \sum_{k\ in\ batch\ jobs} t^k_{pre} * nodes_k \tag{9}$$

$$ch^j_{ckpt} = 0 \tag{10}$$

Here $ckpData^j_i$ is the amount of data to be checkpointed for job j for i_{th} checkpoint; $ckpData^j_{latest}$ is the amount of data checkpointed in the most recent checkpoint for *job j*; $bandwidth^{write}_{PFS}$ and $bandwidth^{read}_{PFS}$ represent the write and read bandwidth of the PFS, respectively; and $ckpTgap^j_i$ is the time elapsed between the time *job j* was checkpointed last and the time *job j* gets preempted for i^{th} preemption.

PRE-CKPT-APP: This scheme corresponds to the application-level checkpointing. Applications checkpoint themselves by storing their execution contexts and recover using that data when restarted without explicit assistance from the system. The checkpoint interval ($ckpIntv_{app}^{j}$) and the amount of data checkpointed ($ckpData^{j}$) change based on the application. Equations 11 to 15 describe the overhead incurred by the preempted jobs (in terms of time and core hours) and the system (in terms of core-hours).

$$t_{ckpt}^{j} = \sum_{i=1}^{\lfloor \frac{t_{runtime}^{j}}{ckpIntv_{app}^{j}} \rfloor} \frac{ckpData_{i}^{j}}{bandwidth_{PFS}^{write}} \tag{11}$$

$$t_{pre}^{j} = \sum_{i=1}^{\#preemptions_j} \frac{ckpData_{latest}^{j}}{bandwidth_{PFS}^{read}} + ckpTgap_{i}^{j} \tag{12}$$

$$ch_{ckpt}^{sys} = 0 \tag{13}$$

$$ch_{pre}^{sys} = \sum_{k\ in\ batch\ jobs} t_{pre}^{k} * nodes_k \tag{14}$$

$$ch_{ckpt}^{j} = t_{ckpt}^{j} * nodes_j \tag{15}$$

PRE-CKPT: In this scheme, jobs are checkpointed right before they get preempted. The premise here is that there is interaction between the scheduler and the checkpointing module. When the scheduler is about to preempt a job, it informs the appropriate checkpointing module and waits for a checkpoint completion notification before it actually preempts the job. The checkpoint and preemption overhead in this scheme is minimal since there is no need to checkpoint at periodic intervals and there will not be any redundant computation (since checkpoint and preemption happen in tandem). Equations 16 to 20 describe the overhead incurred by the preempted jobs (in terms of time) and the system (in terms of core-hours).

$$t_{ckpt}^{j} = \sum_{i=1}^{\#preemptions_j} \frac{ckpData_{i}^{j}}{bandwidth_{PFS}^{write}} \tag{16}$$

$$t_{pre}^{j} = \sum_{i=1}^{\#preemptions_j} \frac{ckpData_{i}^{j}}{bandwidth_{PFS}^{read}} \tag{17}$$

$$ch_{ckpt}^{sys} = \sum_{k\ in\ batch\ jobs} t_{ckpt}^{k} * nodes_k \tag{18}$$

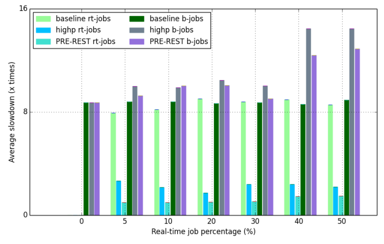
$$ch_{pre}^{sys} = \sum_{k\ in\ batch\ jobs} t_{pre}^{k} * nodes_k \tag{19}$$

$$ch_{ckpt}^{j} = 0 \tag{20}$$

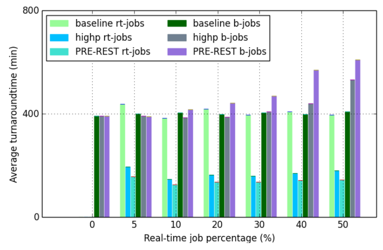
6 Qsim Extensions

We used Qsim [4], an event-driven parallel job scheduling simulator, for our study. We extended QSim to support preemption and accommodate real-time jobs. We implemented in Qsim all five scheduling schemes described in the preceding section. Our extensions provide two ways to mark certain jobs in the job log as real-time jobs: *user-specified* and *random*. In the user-specified approach, users can provide an index list, and the jobs with the index provided in the list are picked as real-time jobs. In the *random* approach, $R\%$ of the jobs are picked randomly as real-time jobs. Our extensions also allow the users to provide inputs as the following:

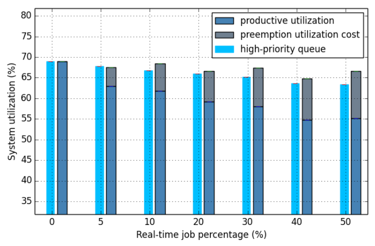
- $ckpData$ - amount of data to be checkpointed
- $bandwidth_{PFS}^{write}$ - write bandwidth of parallel file system
- $bandwidth_{PFS}^{read}$ - read bandwidth of parallel file system
- $ckpIntv_{sys}$ - checkpoint interval for system-level checkpointing
- $ckpIntv_{app}$ - checkpoint interval for application-level checkpointing (percentage of job walltime)



(a)



(b)



(c)

Fig. 1. Performance comparison under different real-time job (RTJ) percentage: (a) *job slowdown* of **PRE-REST**, **hpQ**, and **baseline**; (b) *job turnaround time* of **PRE-REST**, **hpQ**, and **baseline**; (c) *system utilization* of **PRE-REST** and **hpQ** under different RTJ percentage.

7 Experimental Evaluation

In this section we present details of the experimental setup and the workload traces used for our experiments. We then present the simulation results.

7.1 Workload Trace

For this study, we used four week-long trace logs collected from the Mira supercomputer at Argonne. The statistics of the logs are summarized in Table 1. The logs are denoted by Wk-a, Wk-b, Wk-c, and Wk-d to anonymize the specific week; *#Job* represents the number of completed jobs in the trace log; *AvgNumCore* represents the average number of cores required by completed jobs in the trace log; *AvgWallTime* represents the average wall time (in minutes) required by completed jobs in the trace log; *AvgRsc* represents the average amount of resources (in core hours) required by completed jobs in the trace log; and *AvgWallTimeAccu* represents the accuracy of average wall time (in percentage) relative to the average runtime of completed jobs.

Table 1. Statistics of Mira trace logs.

Log	#Job	AvgNumCore (cores)	AvgWallTime (min)	AvgRsc (core hours)	AvgWallTimeAccu (%)
Wk-a	403	2650.9	206.90	7931.1	86.7
Wk-b	1217	2659.8	132.83	5043.5	85.6
Wk-c	852	2437.4	165.39	6206.2	92.3
Wk-d	943	2195.3	235.68	7021.5	81.4

7.2 Experimental Setup

To fully evaluate the performance of all the scheduling schemes under different amount of real-time jobs, we randomly chose $R\%$ (real-time job percentage) of jobs in the experimental trace log and set them as real-time jobs (RTJ), with the rest $(100 - R)\%$ as batch jobs (BJ). In our experiments we used $R \in \{5, 10, 20, 30, 40, 50\}$. Experimental results were averaged over 20 random sample groups for each R value.

We analyzed the performance of the scheduling schemes in terms of the following performance metrics.

- *Job turnaround time*: time difference between job completion time and job submission time
- *Bounded job slowdown (slowdown)*:

$$Bounded\,slowdown = (Wait\,time + Max(Run\,time, 10))/ \\ Max(Run\,time, 10) \qquad (21)$$

The threshold of 10 minutes was used to limit the influence of very short jobs on the metric.

– *System utilization*: proportion of the total available processor cycles that are used.

$$System\,utilization = \frac{(\sum_j runtime_j \cdot nodes_j + ch^j_{ckpt}) + ch^{sys}_{pre} + ch^{sys}_{ckpt}}{Makespan \cdot nodes_{total}} \tag{22}$$

– *Productive utilization (productive_util)*: proportion of the total available processor cycles that are used for actual job execution, which excludes checkpoint and preemption overhead.

$$Productive\,utilization = \frac{\sum_j runtime_j \cdot nodes_j}{Makespan \cdot nodes_{total}} \tag{23}$$

We compare performance of scheduling schemes described in Section 5 with the baseline performance. Baseline peformance is obtained by running both *RTJ* and *BJ* as batch jobs on Qsim with the default scheduling algorithm, which is FCFS with first-fit backfilling. Though we gathered experimental results for four week-long traces (Wk-a, Wk-b, Wk-c, Wk-d described in Table 1), we present the results for only Wk-a due to space constraints. We note that the trends for other three logs are similar to that for the log presented here.

7.3 High-priority Queue and Preemption without Checkpointing

We first evaluated the performance of the high-priority queue and preemption with no checkpointing (PRE-REST) schemes. Figure 1 shows the *average slowdown* and *average turnaround time* of jobs and *system utilization* for different RTJ percentages. From Figures 1(a) and 1(b), we can observe that both the high-priority queue and PRE-REST schemes improve the performance of RTJ significantly without a huge impact to the batch jobs when %RTJ $\leq$ 30.

With the high-priority queue, RTJ achieve much lower *job slowdown* and *job turnaround time* compared with their baseline metrics. But the absolute values are still much higher than the desired values. For example, *job slowdown* ranges from 1.72 to 3.0, significantly higher than desired slowdown of 1. Even though RTJ have higher priority than BJ, they have to wait for the running batch jobs to finish if there are not enough free nodes available for RTJ to start immediately.

From Figure 1(a), we see that preemptive scheduling can achieve a *slowdown* close to 1 for RTJ for workloads with up to 30% RTJ. For workloads with higher percentage of real-time jobs (40% and 50%), however, as more system resources are occupied by RTJ, some RTJ have to wait for the required resources, resulting in a higher average *slowdown* ($\sim$1.5) for RTJ.

Comparing *hpQ* and *PRE-REST*, we see that *PRE-REST* is consistently better than *hpQ* for RTJ in terms of both slowdown and turnaround time. This result is expected because RTJ can preempt the running BJ in *PRE-REST* while they cannot do that in *hpQ*. Regarding BJ, we note that *PRE-REST* is almost always better than *hpQ* in terms of average slowdown, whereas *hpQ* is almost always better than *PRE-REST* in terms of average turnaround time. We conjecture that preemption of batch jobs to schedule RTJ in *PRE-REST* benefits the shorter BJ indirectly. In other words, preemption creates opportunities for shorter BJ to backfill. Of the batch jobs that are preempted, longer jobs will likely have a hard time backfilling and thus will suffer the most in *PRE-REST*. Since *hpQ* does not allow any batch job to be scheduled if an RTJ is waiting, the shorter jobs will not be able to backfill even if they could. The average job slowdown is influenced significantly by the short jobs. In contrast, the average job turnaround time tends to be influenced much more by the long jobs. Since *PRE-REST* causes relatively more negative impact to longer BJ and indirectly benefits shorter BJ, and since *high-priority queue* causes more negative impact to shorter BJ by denying the backfill opportunities that they would have otherwise had, *PRE-REST* is better in terms of average slowdown and *high-priority queue* is better in terms of average turnaround time for BJ. *PRE-REST* scheme having a lower productive utilization than *hpQ* (see Figure 1(c) and the text below) also supports our theory.

Figure 1(c) shows *overall utilization* and *productive utilization* of *PRE-REST* and *high-priority queue*. We note that overall utilization includes all the usage of the system including the redundant cycles used by the preempted jobs (if any) and the cycles spent on checkpointing and preemption (if applicable). In contrast, productive utilization includes only the cycles used for the productive execution of the jobs. For *high-priority queue*, the overall utilization is the same as that of productive utilization since it does not have any redundant computations or any other additional overhead. In *PRE-REST*, portions of preempted jobs get executed more than once since they have to start from the begining after each preemption. In Figure 1(c), the bars on the leftmost end (0% RTJ) correspond to the baseline utilization. We can see that the overall (productive) utilization for *high-priority queue* decreases with the increasing percentage of RTJ. We also see that *High-priority queue* blocks the batch jobs and prevents them from backfilling whenever one or more real-time jobs are waiting. Thus, batch jobs suffer more with increasing numbers of real-time jobs. Although the overall utilization of *PRE-REST* is higher than that of *high-priority queue*, its productive utilization is lower because of the cycles wasted by the restart of preempted jobs from scratch. Productive utilization for *high-priority queue* reduces by 5% (compared with the baseline) when there are 20% real-time jobs and by 10% when there are 50% real-time jobs. In contrast, for *PRE-REST*, productive utilization reduces by 15% when there are 20% real-time jobs and by 20% when there are 50% real-time jobs.

7.4 Performance of Checkpoint-Based Preemptive Scheduling

We compare the performance of preemptive scheduling schemes with the baseline and *hpQ* schemes in Figure 2. From Figure 2a, we can see that for RTJ, all preemptive scheduling schemes can maintain an average slowdown in the range of [1.0, 1.4], as opposed to slowdowns around 2.0 or above with *hpQ* and around

8.0 or above with the baseline. Even for BJ, preemptive scheduling schemes with checkpointing (PRE-CKPT, PRE-CKPT-SYS, and PRE-CKPT-APP) perform significantly better than hpQ and the baseline when the % RTJ ≤ 30 (see Figure 2b). We note that the average turnaround time results have similar trends as the average slowdown expect that the improvement for batch jobs for 30% RTJ is modest. Based on these results, there is no excuse not to support up to 30% RTJ in the workloads. The performance of RTJ is on the expected lines, but the performance improvement for batch jobs when %RTJ ≤ 30 is both surprising and counterintuitive. We suspect that certain categories of BJ are getting benefited at the expense of certain other categories of BJ. Also, not all RTJ are getting the same amount of benefit. To understand these results better, we divided the jobs into four categories: considering two partitions for the number of nodes used (narrow and wide) and two partitions for the runtime (short and long). The criteria used for classification is as follows:

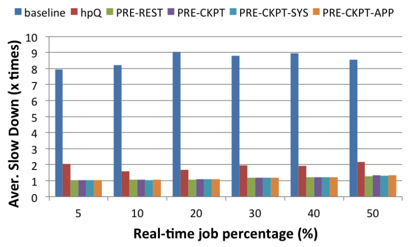
- Narrow: number of nodes used is in the range [512, 4096] inclusive (note that number of nodes allocated on Mira is a multiple of 512)
- Wide: number of nodes used is in the range [4608, 49152] inclusive.
- Short: jobs with runtime ≤ 120 minutes.
- Long: jobs with runtime > 120 minutes.

The performance of the baseline, *hpQ*, and preemptive scheduling schemes for narrow-short, narrow-long, wide-short, and wide-long categories of RTJ and BJ is shown in Figures 3, 4, 5, and 6, respectively. We can see from Figures 3b and 3d that narrow-short batch jobs slowdown and turnaround times with the preemption schemes are significantly better than the baseline and *hpQ* for cases where the %RTJ ≤ 30. For the same cases (%RTJ ≤ 30), however, the performance of the preemption schemes for narrow-long BJ is comparable to that of the baseline and *hpQ*, and for wide-short and wide-long BJ is (significantly) worse than baseline and *hpQ*. Since 63% of the total jobs (57% of RTJ and 64% of BJ) are narrow-short, the overall performance of all jobs shown in Figure 2 is influenced by the performance of narrow-short jobs much more than the performance of jobs in other categories.

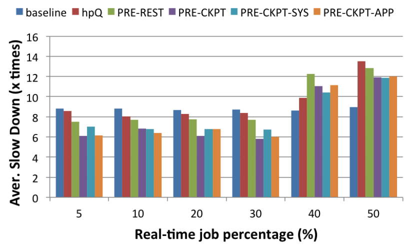
7.5 Impact of Checkpointing Implementations

In this section, we further evaluate the performance of preemptive scheduling in terms of checkpoint data size and checkpoint interval.

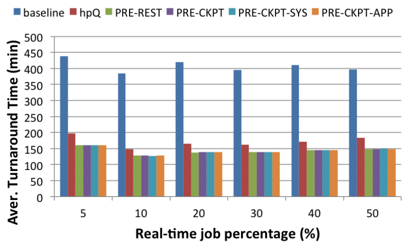
First, to evaluate the performance impact of checkpoint data size, we conducted experiments with different checkpoint data file size for PRE-CKPT. We define checkpoint data file size per node as *dsize*, with *dsize* $\in$ {1GB, 4GB, 16GB}, which represent checkpoint data with a compress rate of {92.75%, 75%, 0%} when the system memory size is assumed to be 16 GB. Based on the I/O performance benchmarks for Mira, we set the I/O bandwidth per node to 2 GB/s while we set the parallel file system (PFS) bandwidth cap for checkpoint/restart data write/read to 90% of the PFS bandwidth (240 GB/s). The results of the



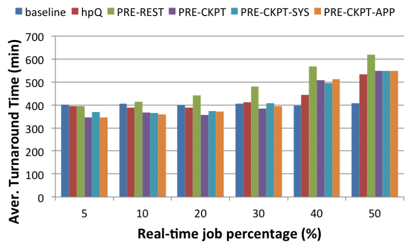
(a) Real-time jobs - slowdown



(b) Batch jobs - slowdown

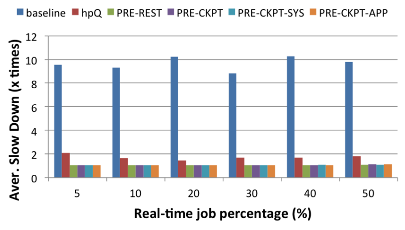


(c) Real-time jobs - turnaround time

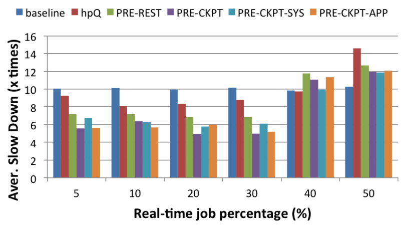


(d) Batch jobs - turnaround time

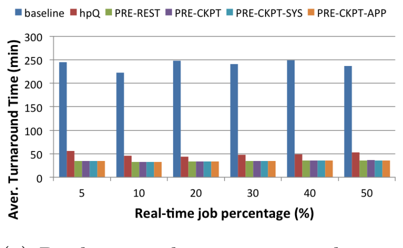
Fig. 2. Performance comparison of baseline, hpQ, PRE-REST, PRE-CKPT, PRE-CKPT-SYS, and PRE-CKPT-APP schemes.



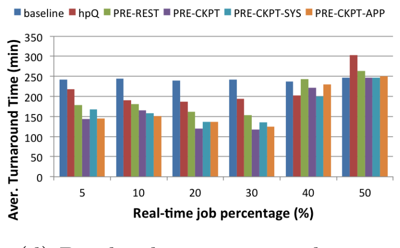
(a) Real-time jobs - slowdown



(b) Batch jobs - slowdown

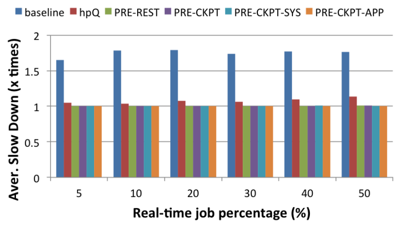


(c) Real-time jobs - turnaround time

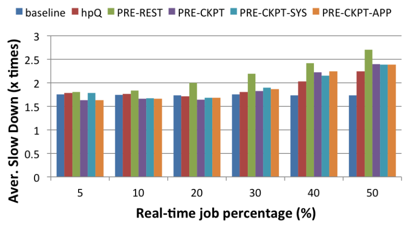


(d) Batch jobs - turnaround time

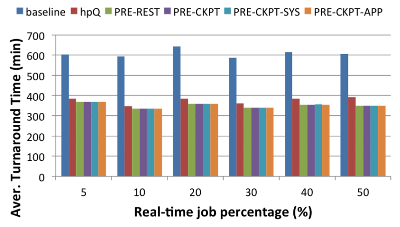
Fig. 3. Performance comparison of baseline, hpQ, PRE-REST, PRE-CKPT, PRE-CKPT-SYS, and PRE-CKPT-APP schemes for narrow-short jobs.



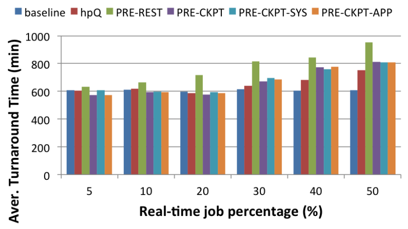
(a) Real-time jobs - slowdown



(b) Batch jobs - slowdown

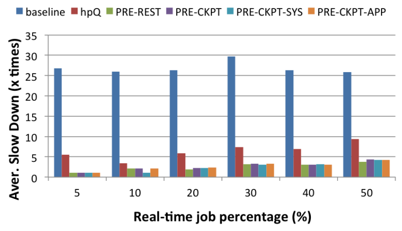


(c) Real-time jobs - turnaround time

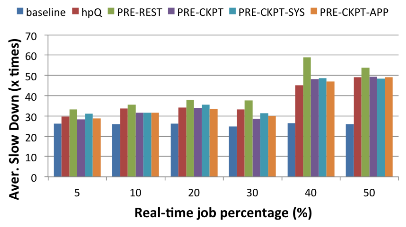


(d) Batch jobs - turnaround time

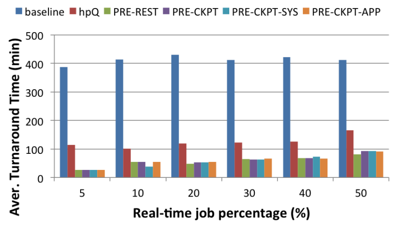
Fig. 4. Performance comparison of baseline, hpQ, PRE-REST, PRE-CKPT, PRE-CKPT-SYS, and PRE-CKPT-APP schemes for narrow-long jobs.



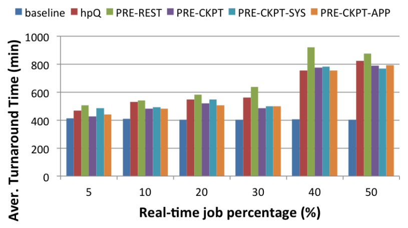
(a) Real-time jobs - slowdown



(b) Batch jobs - slowdown

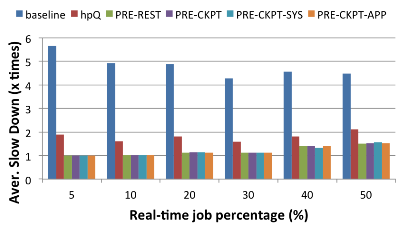


(c) Real-time jobs - turnaround time

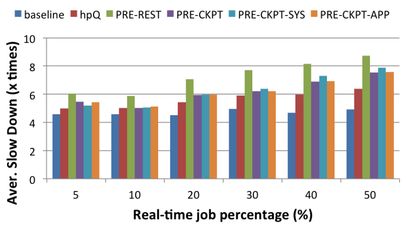


(d) Batch jobs - turnaround time

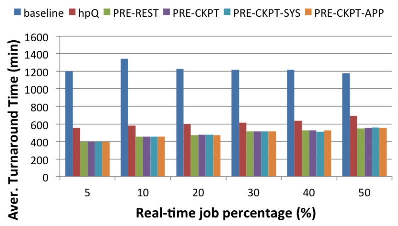
Fig. 5. Performance comparison of baseline, hpQ, PRE-REST, PRE-CKPT, PRE-CKPT-SYS and PRE-CKPT-APP schemes for wide-short jobs.



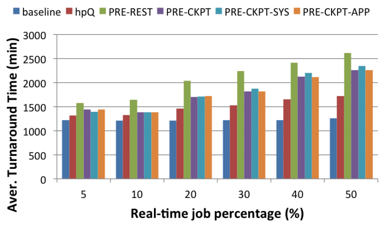
(a) Real-time jobs - slowdown



(b) Batch jobs - slowdown

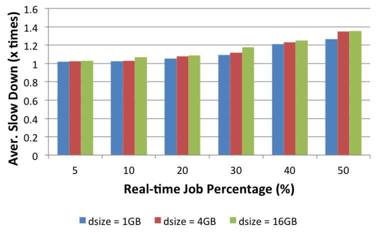


(c) Real-time jobs - turnaround time

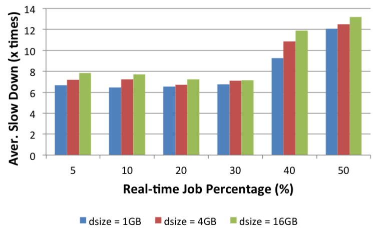


(d) Batch jobs - turnaround time

Fig. 6. Performance comparison of baseline, hpQ, PRE-REST, PRE-CKPT, PRE-CKPT-SYS, and PRE-CKPT-APP schemes for wide-long jobs.

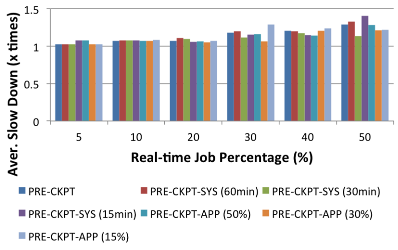


(a) Real-time jobs - slowdown

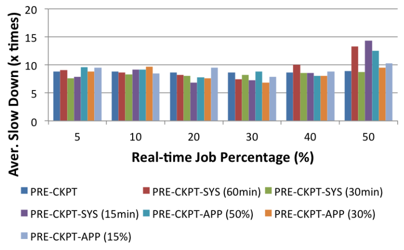


(b) Batch jobs - slowdown

Fig. 7. Performance of PRE-CKPT-SYS for different checkpoint data sizes.



(a) Real-time jobs - slowdown



(b) Batch jobs - slowdown

Fig. 8. Performance comparison of PRE-CKPT, PRE-CKPT-SYS, and PRE-CKPT-APP for different checkpoint intervals.

BJ slowdown are illustrated in Figure 7(b). From the results, we can see the general trend: the average slowdown for BJ increases as the checkpoint data size increases. For example, the average slowdown for BJ increases from 6.2 to 7.8 when the checkpoint data size varies from 1 GB to 16 GB for the workload with 10% RTJ. The results of the RTJ slowdown are illustrated in Figure 7(a). There is no clear trend, which is expected as the checkpoint data size should not affect RTJ.

Next, we study the impact of different checkpoint intervals for PRE-CKPT-SYS and PRE-CKP-APP. We study the performance of PRE-CKP-SYS, for checkpoint intervals $int \in \{15 \text{ min}, 30 \text{ min}, 60 \text{ min}\}$. These interval values are selected based on the average walltime of 207 minutes. We study the performance of PRE-CKP-APP by setting checkpoint intervals to different percentages of walltime (*pcent*). We use $pcent \in \{15\%, 30\%, 50\%\}$. Figure 8 shows RTJ and BJ slowdowns for PRE-CKPT and for different checkpoint intervals for PRE-CKP-SYS and PRE-CKP-APP. These results are for checkpoint data file size per node $dsize = 4$ GB. (I/O bandwidth per node and the PFS bandwidth cap are set to the same values mentioned before). No clear trend is seen from these results. Checkpoint interval should not affect the RTJ performance as only BJ are checkpointed. Longer checkpoint interval will result in lower checkpoint overhead for BJ but a potentially higher restart overhead for preempted BJ. The amount of restart overhead is highly dependent on the schedule. From the results in Figure 8(b), a checkpoint interval of 30 min for PRE-CKP-SYS and 30% walltime for PRE-CKP-APP perform better for most cases.

7.6 Summary of the Results

Even though the non-preemptive hpQ scheme can drastically reduce the slowdown of RTJ (4x or more) compared to the baseline scheme that treats all jobs equally, the absolute values of average slowdown of RTJ is still around 2, which may not be acceptable for RTJ. Preemption is required to bring the average slowdown of RTJ close to 1. Surprisingly, both non-preemptive and preemptive schemes that favor RTJ benefits BJ also, when $\%\text{RTJ} \leq 30$. Further analyses reveal that, in addition to RTJ *narrow-short* BJ also benefit significantly from the schemes that favor RTJ. With preemptive schemes, preemption of *wide* and *long* BJ can help *narrow-short* BJ (in addition to RTJ) through new backfilling opportunities. With hpQ, prioritizing RTJ over BJ (and making *wide* BJ wait) possibly creates additional backfilling opportunities for *narrow-short* BJ. When $\%\text{RTJ} \leq 20$, average slowdowns for *narrow-short*, *narrow-long*, and *wide-long* RTJ remain very close to 1 for all preemptive schemes; and the average slowdown for *wide-short* RTJ is ≤ 1.5 at least for some of the preemptive schemes. Checkpointing definitely helps reduce the negative impact on BJ. BJ slowdown increases with increasing checkpoint data size but there no clear trend with respect to checkpoint interval (checkpoint interval of 30 minutes or when interval is a percentage of walltime, 30% works best).

8 Conclusions

We have presented a simulation-based study of trade-offs that arise when supporting real-time jobs on a batch supercomputer. We studied both preemptive and non-preemptive scheduling schemes to support real-time jobs using production job logs by varying the percentage of real-time jobs in the workload. We compared both slowdown and turnaround time of real-time and batch jobs observed with these schemes against the ones observed with a baseline, which is the scheduling policy used in production for the system we studied. We also analyzed the performance of different categories of jobs and provided detailed insights. We showed that preemptive scheduling schemes can help real-time jobs in all categories achieve an average slowdown less than 1.5 with at most 20% increase in average slowdown for some categories of batch-jobs when the workload has 20% or fewer real-time jobs.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. We thank the Argonne Leadership Computing Facility at Argonne National Laboratory for providing the Mira trace log used in this study.

References

1. Cobalt project. http://trac.mcs.anl.gov/projects/cobalt.
2. Frost, NCAR/CU BG/L System. https://wiki.ucar.edu/display/BlueGene/Frost.
3. Mira. https://www.alcf.anl.gov/mira.
4. Qsim. https://trac.mcs.anl.gov/projects/cobalt.
5. G. Allen, D. Angulo, I. Foster, G. Lanfermann, C. Liu, T. Radke, E. Seidel, and J. Shalf. The Cactus Worm: Experiments with dynamic resource selection and allocation in a grid environment. *IJHPCA*, 15(4), 2001.
6. S. Anastasiadis and K. Sevcik. Parallel application scheduling on networks of workstations. *Journal of Parallel & Distributed Computing*, 43(2):109–124, 1997.
7. A. Barak, S. Guday, and R. Wheeler. *The MOSIX Distributed Operating System - Load Balancing for UNIX*. 1993.
8. S.-H. Chiang and M. K. Vernon. Production job scheduling for parallel shared memory systems. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, pages 47–, Washington, DC, USA, 2001.
9. W. Cirne and F. Berman. Adaptive selection of partition size for supercomputer requests. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 187–208. Springer-Verlag, London, UK, 2000.
10. X. Deng, N. Gu, T. Brecht, and K. Lu. Preemptive scheduling of parallel jobs on multiprocessors. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '96, pages 159–167, Philadelphia, PA, USA, 1996.
11. J. Duell. The design and implementation of Berkeley Labs Linux checkpoint/restart. Technical report, 2003. http://www.nersc.gov/research/FTG/checkpoint/reports.html.

12. I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. In *The Journal of Supercomputing*, pages 885–900, 2005, 65(8).
13. D. G. Feitelson. Job scheduling in multiprogrammed parallel systems. Research Report RC 19790 (87657), IBM T. J. Watson Research Center, Oct 1994.
14. D. G. Feitelson and L. Rudolph. Parallel job scheduling: Issues and approaches. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS '95, pages 1–18. Springer-Verlag, London, UK, 1995.
15. D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling – a status report. In *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–16. Springer-Verlag, Berlin, Heidelberg, 2005.
16. D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In *Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 1–34. Springer-Verlag, London, UK, 1997.
17. J. P. Jones and B. Nitzberg. Scheduling for parallel supercomputing: A historical perspective of achievable utilization. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–16, 1999.
18. W. A. W. Jr., C. L. Mahood, and J. E. West. Scheduling jobs on parallel systems using a relaxed backfill strategy. In *Workshop on Job Scheduling Strategies for Parallel Processing*, 2002.
19. R. Kettimuthu, V. Subramani, S. Srinivasan, T. Gopalsamy, D. K. Panda, and P. Sadayappan. Selective preemption strategies for parallel job scheduling. *IJHPCN*, 3(2/3):122–152, 2005.
20. B. G. Lawson and E. Smirni. Multiple-queue backfilling scheduling with priorities and reservations for parallel systems. In *Workshop on Job Scheduling Strategies for Parallel Processing*, 2002.
21. V. J. Leung, G. Sabin, and P. Sadayappan. Parallel job scheduling policies to improve fairness: A case study. In W.-C. Lee and X. Yuan, editors, *ICPP Workshops*, pages 346–353. IEEE Computer Society, 2010.
22. L. T. Leutenneger and M. K. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. In *ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, pages 226–236, May 1990.
23. D. A. Lifka. The ANL/IBM SP scheduling system. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS '95, pages 295–303. Springer-Verlag, London, UK, 1995.
24. R. Motwani, S. Phillips, and E. Torng. Non-clairvoyant scheduling. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 422–431, Philadelphia, PA, USA, 1993.
25. A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Trans. Parallel Distrib. Syst.*, 12(6):529–543, June 2001.
26. S. Niu, J. Zhai, X. Ma, M. Liu, Y. Zhai, W. Chen, and W. Zheng. Employing checkpoint to improve job scheduling in large-scale systems. In *Job Scheduling Strategies for Parallel Processing*, volume 7698 of *Lecture Notes in Computer Science*, pages 36–55. Springer, Berlin Heidelberg, 2013.
27. E. W. Parsons and K. C. Sevcik. Implementing multiprocessor scheduling disciplines. In *Job Scheduling Strategies for Parallel Processing*, volume 1291, pages 166–192. Lecture Notes in Computer Science, Springer Verlag, 1997.
28. K. Ranganathan and I. Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *Proceedings of the 11th IEEE Interna-*

tional Symposium on High Performance Distributed Computing, HPDC '02, IEEE Computer Society, pages 352–, Washington, DC, USA, 2002.

29. G. Sabin, M. Lang, and P. Sadayappan. Moldable parallel job scheduling using job efficiency: An iterative approach. In *Proceedings of the 12th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 94–114. Springer-Verlag, Berlin, Heidelberg, 2007.
30. G. Sabin and P. Sadayappan. Unfairness metrics for space-sharing parallel job schedulers. In *Proceedings of the 11th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 238–256. Springer-Verlag, Berlin, Heidelberg, 2005.
31. M. Schulz, G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill. Implementation and Evaluation of a Scalable Application-Level Checkpoint-Recovery Scheme for MPI Programs. In *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, pages 38–38, Nov. 2004.
32. K. C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Performance Evaluation*, 19(2-3):107–140, 1994.
33. E. Shmueli and D. G. Feitelson. Backfilling with lookahead to optimize the packing of parallel jobs. *J. Parallel Distrib. Comput.*, 65(9):1090–1107, Sept. 2005.
34. Q. Snell, M. Clement, and D. Jackson. Preemption based backfill. In *Job Scheduling Strategies for Parallel Processing*, volume 2537 of *Lecture Notes in Computer Science*, pages 24–37. Springer, Berlin Heidelberg, 2002.
35. S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. Selective reservation strategies for backfill job scheduling. In *8th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 55–71. Springer-Verlag, London, UK, 2002.
36. S. Srinivasan, V. Subramani, R. Kettimuthu, P. Holenarsipur, and P. Sadayappan. Effective selection of partition sizes for moldable scheduling of parallel jobs. In *Proceedings of the 9th International Conference on High Performance Computing*, HiPC '02, pages 174–183. Springer-Verlag, London, UK, 2002.
37. V. Subramani, R. Kettimuthu, S. Srinivasan, and P. Sadayappan. Distributed job scheduling on computational grids using multiple simultaneous requests. In *Proceedings of the 11th International Symposium on High Performance Distributed Computing*, IEEE Computer Society, pages 359–, Washington, DC, USA, 2002.
38. D. Talby and D. G. Feitelson. Supporting priorities and improving utilization of the ibm sp scheduler using slack-based backfilling. In *Proceedings of the 13th International Parallel Processing Symposium*, pages 513–517, 1999.
39. W. Tang, N. Desai, D. Buettner, and Z. Lan. Job scheduling with adjusted runtime estimates on production supercomputers. *J. Parallel Distrib. Comput.*, 73(7):926–938, 2013.
40. W. Tang, D. Ren, Z. Lan, and N. Desai. Toward balanced and sustainable job scheduling for production supercomputers. *Parallel Comput.*, 39(12):753–768, Dec. 2013.
41. M. Thomas, K. Dam, M. Marshall, A. Kuprat, J. Carson, C. Lansing, Z. Guillen, E. Miller, I. Lanekoff, and J. Laskin. Towards Adaptive, Streaming Analysis of X-ray Tomography Data. *Synchrotron Radiation News*, 28(2):10–14, Mar. 2015.
42. N. Trebon. *Enabling Urgent Computing Within the Existing Distributed Computing Infrastructure*. PhD thesis, University of Chicago, 2011. AAI3472964.
43. J. P. Walters and V. Chaudhary. Application-Level Checkpointing Techniques for Parallel Programs. In *Distributed Computing and Internet Technology*, pages 221–234. Springer, Berlin, Heidelberg, Dec. 2006.
44. J. Zahorjan and C. McCann. Processor scheduling in shared memory multiprocessors. In *ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, pages 214–225, May 1990.