Employing Checkpoint to Improve Job Scheduling in Large-Scale Systems

Shuangcheng Niu[1], Jidong Zhai[1], Xiaosong Ma[2],
Mingliang Liu[1], Yan Zhai[1], Wenguang Chen[1], and Weimin Zheng[1]

[1] Tsinghua University, Beijing, China,
[2] North Carolina State University and Oak Ridge National Laboratory, USA
{niu.shuangcheng,zhaijidong,liuml07,zhaiyan920}@gmail.com,
ma@cs.ncsu.edu, {cwg,zwm}@tsinghua.edu.cn

Abstract. The FCFS-based backfill algorithm is widely used in scheduling high-performance computer systems. The algorithm relies on runtime estimate of jobs which is provided by users. However, statistics show the accuracy of user-provided estimate is poor. Users are very likely to provide a much longer runtime estimate than its real execution time.
In this paper, we propose an aggressive backfilling approach with checkpoint based preemption to address the inaccuracy in user-provided runtime estimate. The approach is evaluated with real workload traces. The results show that compared with the FCFS-based backfill algorithm, our scheme improves the job scheduling performance in waiting time, slowdown and mean queue length by up to 40%. Meanwhile, only 4% of the jobs need to perform checkpoints.

Keywords: job scheduling, backfill algorithm, runtime estimate, checkpoint/restart

1 Introduction

Supercomputers and clusters today are usually managed by batch job scheduling systems, which partition the available set of compute nodes according to resource requests submitted through job scripts, and allocate such node partitions to parallel jobs. A parallel job will occupy the allocated node partition till *1)* the job completes or crashes, or *2)* the job runs out of the maximum wall execution time specified in its job script. Jobs that cannot immediately get their requested resources will have to wait in a job queue. The goal for parallel job schedulers is to reduce the average queue waiting time, and maximize the system throughput/utilization, while maintaining fairness to all jobs [1].

Parallel job scheduling technology has matured over the past decades. Though many strategies and algorithms have been proposed, such as dynamic partitioning [2] and gang scheduling [3], they are not widely used due to practical limitations. Instead, FCFS (First Come First Served) based algorithms with backfilling are currently adopted by most major batch schedulers running on

supercomputers and clusters alike. It was first developed for the IBM SP1 parallel supercomputer installed at Argonne National Laboratory, as part of EASY (the Extensible Argonne Scheduling sYstem) [4]. Several variants of FCFS-based backfilling (referred to as *backfilling* in the rest of this paper) serve as the default setting in today's prominent job schedulers, such as LSF [5], Moab [6], Maui [7], PBS/Torque [8], and LoadLeveler [9].

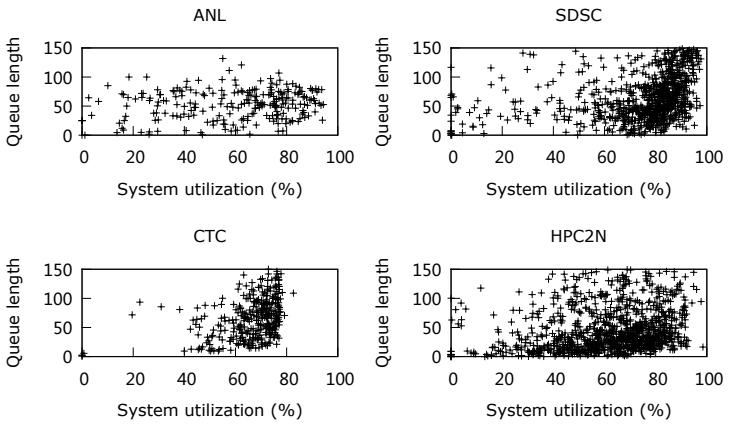


Fig. 1. Queue length vs. system utilization level based on job traces on four parallel systems

The backfilling algorithm is able to make use of idle "holes" left caused by advance reservation for jobs that arrived earlier but unable to run due to insufficient resources, by filling in small jobs that are guaranteed to finish before the reservation starts (more details of the algorithm will be given later). However, systems using such mechanism still suffer from resource under-utilization. Figure 1 illustrates such problem based on the workload traces from four production systems (available online at the Parallel Workloads Archive [10]): Intrepid at Argonne National Laboratory (ANL), Blue Horizon at the San Diego Supercomputing Center (SDSC), IBM SP2 at the Cornell Theory Center (CTC) and Linux cluster at High Performance Computing Center North, Sweden (HPC2N). The parallel job schedulers used at these systems Cobalt, Catalina, EASY, and Maui, respectively, most of which use FCFS+backfilling algorithms. [3] For each system, Figure 1 plots the average system utilization level (x axis) and the average job queue length (y axis), calculated for each day from the traces. The results show that on all four systems, it is common for a significant amount of system resource (20% to 80% of nodes) to be idle while there are (many) jobs waiting in the queue. In Peta- and the upcoming Exa-FLOP era, such system

[3] Cobalt uses a WFP-based backfill algorithm.

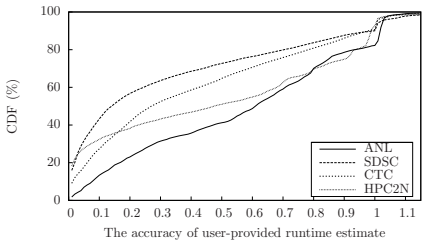


Fig. 2. Distribution of the ratio between the actual and the user-estimated job execution time. Note that supercomputers scheduling policies often allow a "grace period", such as 30 minutes, before terminating jobs that exceed their specified maximum wall time. This, plus extra storage and cleanup operations involved in job termination, explains the existence of ratios larger than 1.

under-utilization translates into more severe waste of both compute resources and energy.

One key factor leading to this defect is that the backfilling effectiveness depends on the job execution wall time estimate submitted by the user, which has been found in multiple studies to be highly inaccurate [11, 12]. By comparing the user-estimated and the actual job execution times from the aforementioned traces, we calculate the probability density function of the wall time estimate accuracy, as t_{run}/t_{req}, where t_{run} is the actual runtime and t_{req} is the user-provided estimate.

Figure 2 suggests that a large portion ($\sim$40%) of jobs actually terminated within 20% of the estimated time. The distribution of the actual/estimated execution time ratio spread quite uniformly over the rest of the spectrum, except a noticeable burst around 1. Overall, only 17% of jobs completed within the 0.9-1.1 range of the estimated time, across the four systems. As to be discussed in more detail in the next section, such dramatic job wall time estimate error leads to significant problems in backfilling effectiveness.

To improve scheduling performance, many previous studies have targeted improving the accuracy of job execution time estimate [13–15], with only limited success. In fact, there are several factors leading to inaccurate estimates, mostly overestimates. Firstly, users may not have enough knowledge on the expected execution time of their jobs (especially with short testing jobs and jobs with new input/parameters/algorithms), and choose to err on the safe side. Secondly, in many cases the large estimated-to-actual execution time ratio is caused by unexpected (early) crash of the job. Thirdly, users tend to use a round value (such as 30 min or 1 hour) as the estimated wall time, causing large "rounding error" for short jobs. All the above factors help making job wall time overestimation a perpetual phenomenon in batch systems. Actually, a study by Cynthia et al. revealed that such behavior is quite stubborn, with little estimate accuracy improvement even when the threat of over-time jobs being killed is removed [16].

In this paper, we propose a new scheduling algorithm that couples advance reservation, backfilling, and job preemption. Our approach is motivated by the observation that on today's ever-larger systems, checkpointing has become a standard fault tolerance practice, in answer to the shortening MTBF (mean time between failures). With portable, optimized tools such as BLCR [17], parallel file systems designed with checkpointing as a major workload or even specifically for the checkpointing purpose (such as PLFS [18]), emerging high-performance hardware such as aggregated SSD storage, and a large amount of recent/ongoing research on checkpointing [19–21], the efficiency and scalability of job checkpointing has been improving significantly. Such growing checkpoint capability on large-scale systems enables us to relax the backfill conditions, by allowing jobs to be aggressively backfilled, and suspended for later execution if resources are due for advance reservation.

This approach manages to make use of idle nodes without wasting resources by aborting opportunistically backfilled jobs. By limiting the per-job checkpointing occurrences and adjusting the backfill aggressiveness, our algorithm is able to achieve a balance between improving resource utilization, controlling the extra checkpointing and restarting overhead, and maintaining scheduling fairness.

We consider our major contributions as follows:

1. *A checkpoint-based scheduling algorithm.* To our knowledge, this is the first paper that discusses how to leverage checkpointing techniques to optimize the backfilling scheduling algorithm. Although some previous works proposed preemption based backfill, no works discussed how to use checkpoint technique to improving backfilling scheduling. We analyzed the limitations of an existing FCFS-based backfilling algorithm and improved it by enabling aggressive backfilling with estimated wall time adjustment and checkpoint-based job preemption.
2. *Comprehensive evaluation with trace-driven simulation.* We conducted experiments using job traces collected from four production systems, including Intrepid, currently ranked 15th on the Top500 list. Our results indicate that compared with the classical FCFS-based backfill algorithm, the checkpoint-based approach is capable of producing significant improvements (by up to 40%) to key scheduling performance metrics, such as job average wait time, slowdown, and the mean queue length.
   In our evaluation, we also estimated the overhead incurred by checkpoint/restart operations, based on system information from the Intrepid system. Different I/O bandwidths are considered, simulation results suggest that extra checkpoint/restart operations hardly cause any impact on the key scheduling performance metrics.

The remainder of this paper is organized as follows. Section 2 reviews the traditional FCFS-based backfilling algorithm. Section 3 presents our checkpoint-based scheduling approach and gives detailed algorithms. Section 4 reports our simulation results on job scheduling performance, while Section 5 analyzes the introduced checkpoint overhead. Section 6 discusses related work, and finally, Section 7 suggests potential future work and concludes the paper.

2 Classical Backfilling Algorithm Overview

In this section, we review the classical backfilling algorithm to set a stage for presenting our checkpoint-based approach.

Most modern parallel job schedulers give static allocations to jobs with spatial partitioning, i.e., a job is allocated the number of nodes it requested in its job script and uses this partition in a dedicated manner throughout its execution. The widely used FCFS-based backfill algorithm does the following:

- maintain jobs in the order of their arrival in the job queue and schedule them in order if possible,
- upon a job's completion or arrival, dispatch jobs from the queue front and reserve resources for the first job in the queue that cannot be run due to insufficient resource available,
- based on the user estimated wall times of the running jobs, calculate the backfill time window, and
- traverse the job queue and schedule jobs that can fit into the backfill window, whose execution will not interfere with the advance reservation. Such jobs should either complete before the reserved "queue head job" start time or occupy only nodes that the advance reservation does not need to use.

A simplified version of such strategy, used in the popular Maui scheduler [7], is shown as Algorithm 1, which is also used as the classical backfilling algorithm in our trace-driven simulation experiments. Other popular schedulers such as EASY and LoadLeveler also use similar frameworks.

As can be seen from the algorithm, user-provided estimated runtime is crucial to the overall scheduling performance. Overestimation will not only postpone the reservation time, but also affect the chance for a job to be backfilled. We will see later that increased estimate inaccuracy can significantly impact the classical algorithm's scheduling performance, particularly in the "job slowdown" category.

3 Checkpoint-based Backfilling

In this section, we explain our scheduling approach and the proposed algorithm in details. Our algorithm overcomes the limitation in the classical backfilling algorithm by weakening the impact of the user job run time estimation accuracy on the system performance. With our approach, both the wait time and the chance of backfill can be significantly improved.

3.1 Assumptions

First, we list several major assumptions made in this work:

1. *Rigid jobs:* Our design assumes that the jobs are rigid jobs, running on a fixed number of processors (nodes) during its end-to-end execution. This resource requirement is specified in the job script.

Algorithm 1 FCFS-based backfilling algorithm

Require: Job queue Q is not empty, Resource manager RM

/* Schedule runnable jobs */
for each $job \in Q$ **do**
  if $job.size > RM.free\_size$ **then**
    $break$
  end if
  $job.predictEndtime \leftarrow job.estimateRuntime + now()$
  $run(job)$
  $RM.free\_size- = job.size$
end for
if $Q.isEmpty()$ **then**
  $exit$
end if
/* Make Reservation */
$reserve\_job \leftarrow Q.top()$
$future\_available\_size \leftarrow RM.free\_size$
$running\_jobs\_list \leftarrow$ Sort running jobs in order of their predicted termination time
for each $job \in running\_jobs\_list$ **do**
  $future\_available\_size$ += job.size
  if $future\_available\_size \geq reserve\_job$.size **then**
    $reserve\_time \leftarrow job.predictEndtime$
    $break$
  end if
end for
$backfill\_window \leftarrow reserve\_time - now()$
$extra\_size \leftarrow future\_available\_size$ - $reserve\_job$.size
/* Backfilling */
for each $job \in Q$ **do**
  if $job.size > RM.free\_size$ **then**
    $continue$
  end if
  $predictRuntime \leftarrow$job.estimateRuntime
  if $predictRuntime \leq backfill\_window$ **or** $job.size \leq extra\_size$ **then**
    $job.predictEndtime \leftarrow predictRuntime + now()$
    $backfill(job)$
    $RM.free\_size- = job.size$
    if $predictRuntime > backfill\_window$ **then**
      $extra\_size- = job.size$
    end if
  end if
end for

2. *Jobs not bound to specific nodes:* Each job can execute on any set of nodes that meets the processor number requirement.
3. *Checkpoint/Restart abilities:* The applications or the supercomputing center has checkpoint/restart (C/R) abilities, which can be initiated any time by the scheduler.

Note that the first two assumptions apply to most of classical backfilling algorithms too, allowing each job to be scheduled once and assigned an arbitrary subset of all the available nodes. In the context of our checkpoint-based backfilling scheduling, however, they allow an interrupted and checkpointed job to resume its execution with the same number of nodes without any placement constraints.

3.2 Methodology Overview

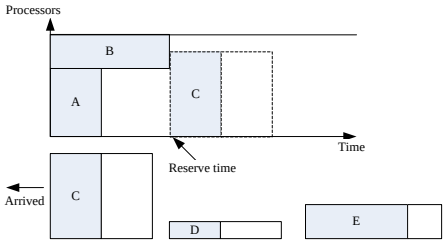
The main idea behind our checkpoint-based scheduling is to allow the queued jobs to be backfilled more aggressively, but reserve the right to suspend such jobs when they are considered to delay the execution of a higher-priority job. The flow of our algorithm works similarly as the classical one. The major difference lies in that at the time a high priority waiting job with advance reservation is expected to run, if any backfilled jobs are still running, we checkpoint rather than kill them. This allows us to cope with the highly overestimated run time and perform aggressive backfilling. In selecting backfill candidates, rather than directly using the user provided run time estimate, our scheduler predicts a job's actual execution time by intentionally *scale down* the estimated run time. According to Figure 2, most of the overestimated jobs can safely complete before the reserved deadline. For those jobs that such prediction actually underestimates their execution time, checkpointing avoids wasting the amount of work already performed, yet maintains the high priority of the job holding advance reservation.

Figure 3 illustrates the working of the checkpoint-based scheduling scheme with a set of sample jobs. Each job is portrayed as a rectangle, with the horizontal and vertical dimensions representing the 2D resource requirement: estimated wall time and processor/node number requested. The gray area within each waiting job's rectangle indicates its actual execution time. At a certain time point, jobs A and B are running, and jobs C, D and E are waiting in the queue, as shown in Figure 3(a). Upon A's completion, there are not sufficient resources for the queue head job C. The scheduler performs advance reservation for C, based on the estimated wall time of B. The moment that B terminates is the *reserve time* for C. The time from A's to B's termination forms the *backfill window*.

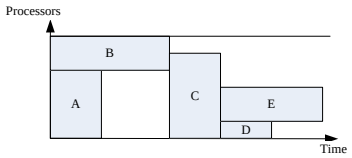
With the classical backfilling algorithm (Figure 3(b)), because D's and E's runtime estimates exceed the backfill window, D and E will not be backfilled, although D's actual runtime can fit in the window. D and E will run after C terminates, wasting resources within the backfill window.

With the checkpoint-based backfilling algorithm (Figure 3(c)), instead, both D and E will be backfilled. D will terminate before the reserved time. E, on the other hand, will be preempted at the reserved time and broken into two segments: E1 and E2. When C terminates, E will be restarted to resume its execution.

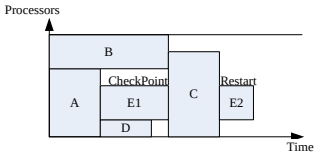
This example illustrates several advantages of the checkpoint-based scheduling approach. First, overestimated jobs benefit from eager backfilling, producing both shorter wait time and higher system utilization. Second, the checkpointed



(a) In the system, 2 jobs are running, 3 jobs are waiting.



(b) Scheduling with FCFS-based backfill algorithm



(c) Scheduling with checkpoint-based backfill algorithm

Fig. 3. With checkpoint-based backfill algorithm, job D which is overestimated benefits from the backfilling, job E also reduces the response time. Meanwhile, the top queue job is not affected.

Algorithm 2 Job.predictRuntime()

Require: job runtime estimate threshold t_{thr}, the split factor p

```
1: if job.estimateRuntime < t_thr then
2:     return job.estimateRuntime
3: else
4:     if job.isCheckPointJob() then
5:         return job.estimateRuntime − job.hasRunTime
6:     else
7:         return job.estimateRuntime ∗ p
8:     end if
9: end if
```

jobs also observe a reduced response time. Finally, the priority of the job holding advance reservation is preserved.

3.3 The Checkpoint-based Scheduling Algorithm

Next, we give more detailed description of our proposed checkpoint-based scheduling algorithm and discuss important parameters.

The main checkpoint-based scheduling framework is same with classical backfilling (Algorithm 1). It is executed repeatedly whenever a new job arrives or when a running job terminates. The major different is in Line 31 (highlighted in

Algorithm 3 Preempt algorithm

Require: the reserve time arrives

```
1: if RM.free_size < reserve_job.size then
2:    preempting_jobs_list ← RM.getPreemptingJobsList(reserve_job)
3:    for each job ∈ preempting_jobs_list do
4:       checkpoint (job)
5:       kill (job)
6:       RM.free_size+ = job.size
7:       Q.push(job)
8:    end for
9: end if
10: run (reserve_job)
11: RM.free_size− = reserve_job.size
```

the algorithm with underlined subroutine calls). When checking backfill eligibility and calculating the predicted end time, it according to the scheduler-predicted run time instead of the user-provided runtime estimate.

The calculating predicate runtime algorithm is the core algorithm, which is shown in Algorithm 2. It describes how the checkpoint-based scheduler performs job execution time prediction. As mentioned earlier, it scales down the user-estimated job wall time t_{req} by a factor of p, $0 < p <= 1$. However, such adjustment is only applied to jobs that are over a certain threshold t_{thr}. This is due to several considerations. First, short jobs are fairly easy to be backfilled even without such scale-down. Second, checkpointing and restart will turn out as more expensive to short jobs. Third, short jobs are often meant for testing and debugging, where a split execution might cause more degradation in user experience. Both p and t_{thr} are tunable parameters, controlling the aggressiveness of backfilling. In our experiments, we find that such simple schemes seem to work well and system administrators can select a proper parameter values based on empirical results collected from their actual workloads.

Finally, Algorithm 3 specifies the preempt scheme. It is executed whenever the reserved time arrives but there are no sufficient resources to allocate for the top priority job. In this algorithm, the backfilled jobs are checkpointed and terminated, until the top queue job gets its requested resources. The checkpointed jobs are put on the head of the queue. Also, in selecting checkpoint candidates, we start from jobs that are using more nodes, to reduce the number of preemption and checkpointing. Note that each backfilled jobs use fewer nodes than requested by the high-priority job holding advance reservation. Otherwise the latter could be scheduled at an earlier time point. Therefore, our checkpoint candidate identification is essentially performing a "best-fit" selection.

4 Evaluation Results

In this section we present our evaluation results obtained from trace-driven simulation experiments. We first analyze the performance of our proposed checkpoint-

based backfill approach using real job traces, and compare it with that of the classical FCFS-based backfilling algorithm. Second, we evaluate the impact of the accuracy of user-provided estimate runtime on the performance of scheduling algorithms.

Simulator We designed and implemented a trace-driven simulator to evaluate the performance of various of scheduling algorithms, which simulate events and states related to batch job scheduling, such as node allocation, job scheduling, job queue status, etc. The input of the simulator includes a job submission trace and a scheduling algorithm. The output of the simulator includes key performance metrics for scheduling systems, which are to be discussed in more details below.

Workload Traces In our evaluation, we analyze four real workload traces collected from production systems from the Parallel Workload Archive [10]. Variants of backfilling is used as the default scheduling strategy in these systems. Each trace entry contain job description information items such as user-provided run time estimate, job submission time, actual execution time, and job size (number of processors requested). Below we briefly describe these traces:

- *ANL*: This trace contains entries for 68,936 jobs that were executed on a 40,960-node IBM Blue Gene/P system at Argonne National Laboratory called Intrepid. It was collected during the first 8 months of 2009 from the 40-rack production Intrepid.
- *SDSC*: This trace contains entries for 250,440 jobs that were executed on the 144-node IBM SP2 called Blue Horizon at the San Diego Supercomputer Center from April 2000 to January 2003.
- *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.
- *HPC2N*: This trace contains entries for 527,371 jobs that were executed on a 120-node Linux cluster from the High-Performance Computing Center North (HPC2N) in Sweden from July 2002 to January 2006.

Metrics We evaluate our proposed approach with four commonly used scheduling performance metrics, as defined below.

- *Wait Time (wait)*: the average per-job wait time in the job queue.
- *Bounded Slowdown (slowdown)*: the ratio of a job's response time to its actual execution time. In this work, we use bounded slowdown to reduce the impact of very short jobs on the average value, calculated as shown in the formula below. The bound value of 10 seconds is used in all our experiments.

$$\frac{WaitTime + Max(RunTime, BoundTime)}{Max(RunTime, BoundTime)}$$

- *Queue Length (qLength)*: the number of average waiting jobs in the job queue at a given time.
- *Backfill Ratio (bRatio)*: the ratio of the number of backfilled jobs to the total number of jobs.

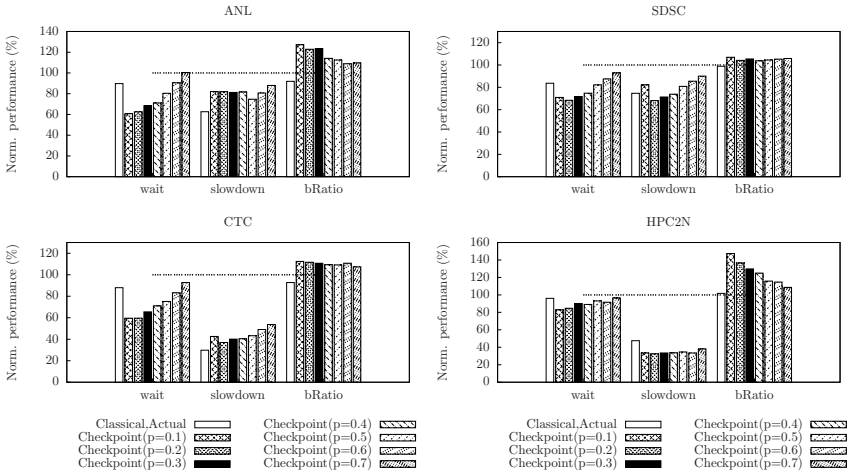


Fig. 4. Performance of checkpoint-based algorithm with different p values, normalized against the performance of the classical backfilling algorithm, which is marked by the dotted "100%" reference line. The "Classical, Actual" bar shows the performance with the classical algorithm, but supplied with the actual wall time (i.e., without any estimation error).

As will be shown later, with the checkpoint-based backfilling algorithm, most of the jobs still wait only once in the queue. Thus its wait time is the difference between its execution and submission. However, for jobs that do go through checkpointing, the wait time includes all segments of waiting the job spends in the queue.

Note that our trace-driven simulation is asynchronous, in the sense that we replay each job's submission according to the submission time specified in the trace. Therefore, even if an algorithm can improve system utilization, it will not be able to shorten the makespan for all jobs. In this context, it can be proved that the average wait time and the average queue length are equivalent. Therefore, in our results discussion, we only report the job wait time.

4.1 Performance of checkpoint-based backfill algorithm

Recall that in the checkpoint-based backfill algorithm, there are two key parameters:

- t_{thr}: The estimated wall time threshold, to mask jobs with wall time estimate smaller than this given threshold from being scaled down with p when calculating the predicted execution time. If t_{thr} is set too low, many short jobs will be checkpointed, causing increased overhead. In this paper, we fixed this parameter at 1800 seconds.
- p: The scale-down factor used to predict the actual job run time. The process of tuning p needs to consider the tradeoff between system utilization and

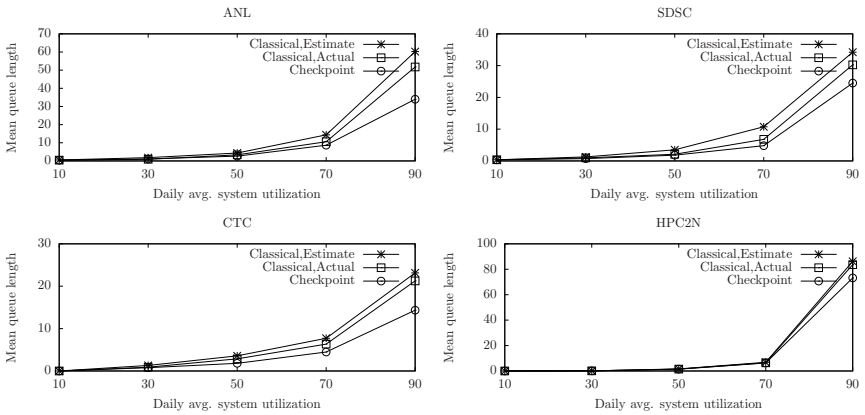


Fig. 5. Scheduling performance of different backfilling algorithms, averaged over subset of days where the system utilization level falls into certain intervals.

cost. If p is set too low, many of the backfilled jobs might not finish by the reserved time for higher-priority jobs, and have to be checkpointed. In contrast, a very high value of p can reduce the backfill ratio and work quite similar to the classical algorithm. So we suggest that system administrators utilize their systems' historical job statistics in selecting a proper p.

Figure 4 shows the simulation results and compares the checkpoint-based algorithm with two variants of the classical one (with the user supplied run time estimate and with the actual job run time as a "ideal" estimate).

From the result, our algorithm performs better than both the base and ideal cases of classical algorithm in wait time (equivalent to queue length) and the backfill ratio. This is expected because of our relaxed condition. At the point of $p = 0.1$ or $p = 0.2$, our algorithm performs better than other selection of p.

One thing to note is the slowdown factor. Except for trace HPC2N, the slowdown in our algorithm can be larger than the ideal case of classical algorithm (but still far better than that of the base classical algorithm). The dominant reason for this is that the accurate estimation of small jobs can significantly improve the slowdown factor. However, it's impossible for the users to know exactly how long their jobs can run. In fact, our algorithm still performs better than classical algorithm if we directly use the runtime estimate in the collected trace. In trace set CTC and HPC2N, the slowdown decreases more than 50%.

So from these experiments, we have verified that our algorithm is more advanced than the classical algorithm in improving scheduling performance. Across the traces we obtained, it appears that a p value of 0.1 or 0.2 delivers the best overall performance. It matches the observation from Figure 2: more than 40% jobs have at least a 5-time overestimation in specifying their expected wall time.

Figure 5 further compares the three algorithms by partitioning days recorded in the trace into multiple buckets according to the daily average system

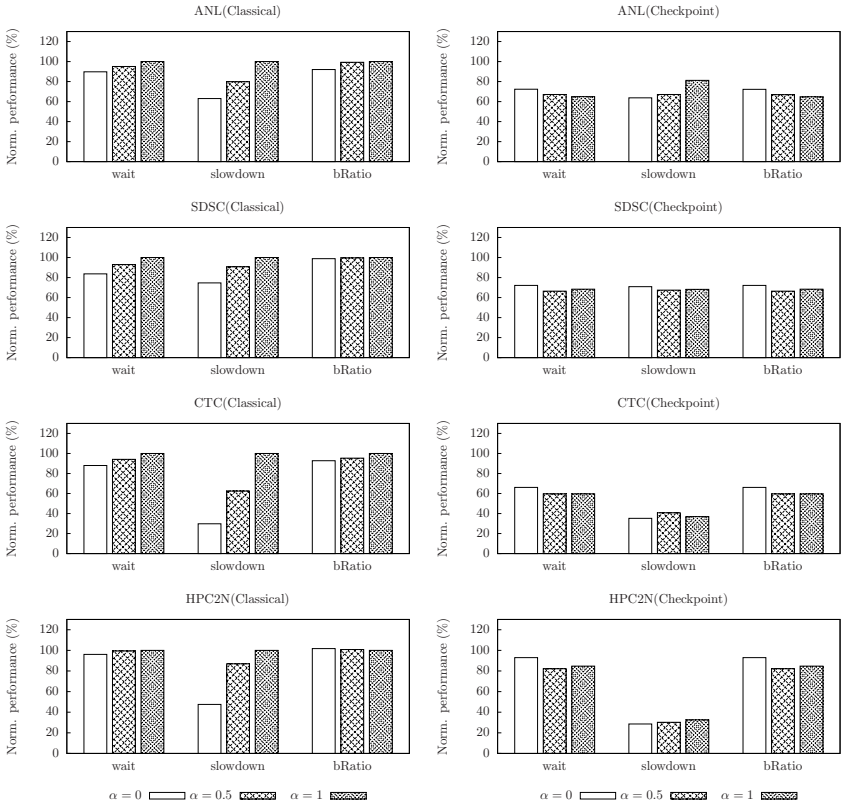


Fig. 6. Evaluation of the impact of estimate accuracy on scheduling algorithms. The left side uses classical algorithm, and right side uses the checkpoint-based algorithm with $p = 0.2$. The higher α is, the more inaccurate runtime estimate will be. When $\alpha = 0$, it is scheduled with actual runtime. When $\alpha = 1$, it is scheduled with user-provided runtime estimate.

utilization level, and depicting average queue length over jobs in each bucket. It demonstrates that our proposed checkpoint-based algorithm significantly reduces the average queue length on most of the four platforms, especially when the system is busy.

4.2 The impact of estimate accuracy on scheduling algorithms

To understand the impact of overestimation on algorithms' behavior, we evaluate how the two algorithms perform under different degrees of overestimation. As in the Figure 6, the result indicates checkpoint-based algorithm is more stable regarding the degree of inaccuracy changing, especially slowdown metric.

The degree of overestimation is defined as a job's actual run time dividing the runtime estimate. To do quantitative analysis, we introduce a parameter α,

where $t_{sch} = t_{run} + \alpha \times (t_{req} - t_{run})$. Here t_{req} is the job's user requested run time in real world trace. We use the t_{sch} as the job's user estimated runtime to submit to the simulator. In this manner, we can obtain workload traces with different overestimation degrees by tuning the α. When $\alpha = 0$, $t_{sch} = t_{run}$ and when $\alpha = 1$, $t_{sch} = t_{req}$. The larger α is, the more inaccurate estimate time will be.

In Figure 6, we simulate the cases when α is 0, 0.5 and 1. The left side uses classical algorithm, and right side uses the checkpoint-based algorithm. It's quite clear left results vary in much larger extent when the runtime estimate tends to be more inaccurate. For example, the slowdown factor, which grows even more than twice when α changes from 0 to 1 in trace sets CTC and HPC2N.

These results indicate that classical algorithm might work well when the user estimation is accurate enough, and it's sensitive to the inaccurate factor. However, in Figure 2 we have shown it is usually unrealistic to have accurate estimate in real world system: a large portion of users tend to highly overestimate the actual run time by more than 5 times. Moreover, even in optimal case ($\alpha = 0$), classical algorithm can archive only comparable performance as checkpoint-based algorithm, which is also consistent with our results in previous section.

5 Analyzing Checkpoint/Restart overhead

In the previous section, we don't consider the overhead of the checkpoint/restart operations. Actually the software state and temporary data are saved to the storage in checkpoint process, and are restored in restart process. In this section, we evaluate the overhead introduced by checkpoint/restart via simulation.

5.1 Checkpoint overhead analysis

We select the trace of Argonne Intrepid to evaluate. Intrepid is a 557 TF, 40-rack Blue Gene/P system deployed at Argonne National Laboratory. This system was ranked No. 15 in the list released in June 2011 [22]. It has 40,960 compute nodes and 640 I/O nodes, which is configured with a single I/O node managing 64 compute nodes. These I/O nodes connect to the file servers with 10 Gb Ethernet network. The peak bandwidth between the I/O node and the network is limited to 6.8 Gb/s. All the 640 I/O nodes can theoretically deliver up to 4.25 Tbps [23].

The entire Intrepid system consists of 128 dual-core file servers. The file servers connect to DataDirect 9900 SAN storage arrays through the Infiniband DDR ports, each with a theoretical unidirectional bandwidth of 16 Gbps. All the 128 file servers can theoretically deliver up to 2 Tbps [23].

Generally, the bandwidth bottleneck for the maximum data throughput lies more towards the file servers than the I/O nodes when the system is running in its full capacity. However, only partial running jobs are involved in checkpoint in the checkpoint based backfilling scheduler. The bottleneck depends on the I/O nodes bandwidth, which is limited to 0.85 GB/s per I/O node [23].

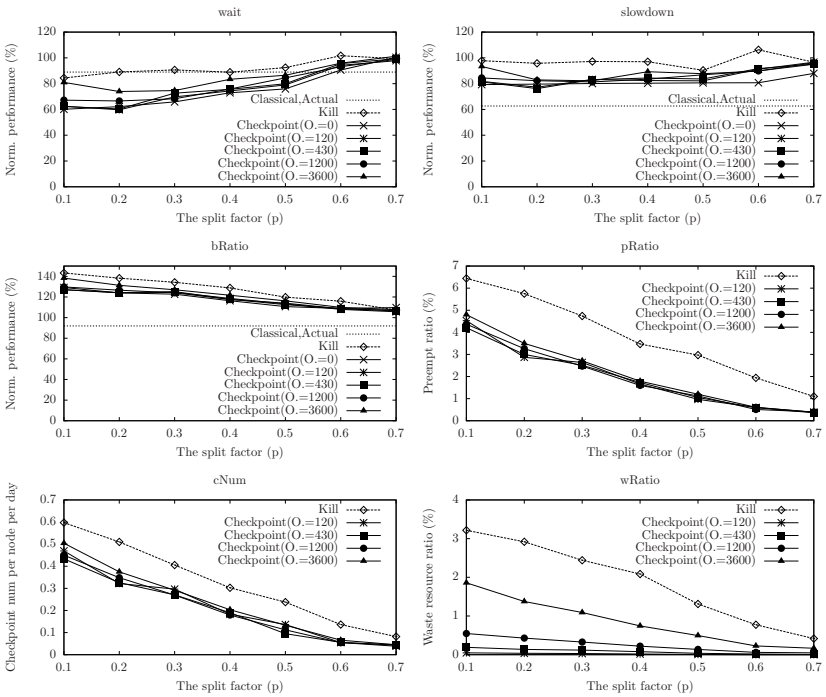


Fig. 7. Evaluation the impact of checkpoint overhead using ANL trace. Different checkpoint overhead are simulated. The simulate results that using classical backfill algorithm with runtime estimate are the baseline.

At the reserve time, it's not allowed to run the reserved job until all preempted job have done checkpoint. The delay of the reserved job depends on the maximal checkpoint duration of all preempted jobs. The restart process also consumes the resources, which extends the actual runtime. There is no memory requirement information in the workload trace of ANL. So we prepare for the worst, and simply assume the required memory is the total memory of the compute nodes. We assume that 70% of the bandwidth is available. Therefore, the delay time and extended runtime can be calculated as following.

$$T_{delay} = T_{chkpnt} = 2G \times 64/(0.85GB/s \times 70\%) = 215s.$$

$$T_{extend} = T_{chkpnt} + T_{restart} \leq 2 \times T_{chkpnt} = 430s.$$

Algorithm 2 should be modified to reflect the runtime extending. In the algorithm, the predicted runtime of checkpointed jobs need to add the T_{extend}.

5.2 Evaluation results with overhead

Our algorithm with checkpoint overhead is evaluated in Figure 7. It's compared against the same algorithm without adding overhead. To reflect the impact of

the checkpoint overhead, we simulated with different T_{extend} values such as 120, 430, 1200 and 3600 seconds.

In addition to the original metrics, we also import three metrics to depict the overhead:

- *Preempt Ratio (pRatio)*: The ratio of the preempted jobs to all jobs.
- *Checkpoint number per node in a day (cNum)*: The average checkpoint number for a compute node in a day.
- *Waste Resource Ratio (wRatio)*: The ratio of the waste computing resources by preempt to all computing resources.

From the result, even adding the overhead, our algorithm still outperforms the baseline a lot when p is 0.2. The waiting time and queue length are improved by up to 40%. The slowdown is improved about 20%. In fact, Figure 7 indicates that the chances we have to do checkpoint are rare. When $p = 0.2$, the average number of checkpoint on each node is about 0.4 times per day and only 4% of the jobs need to do checkpoint. The waste resource is not more than 1.5%.

One thing to note is the intercepted point in slowdown curve in Figure 7. In fact, the runtime extending enlarges the backfill window and lets more shorter jobs to be backfilled. Thus, the backfill ratio increases, and the slowdown becomes better in some cases.

In summary, the overhead in checkpoint does not hurt the algorithm a lot, since the number of checkpoint we need to do is not quite large in our settings. Thus we stand on solid ground to conclude that the overhead is tolerable compared with the benefits gained.

6 Related work

6.1 Job Scheduling Algorithms

Job scheduling systems take an important part in improving the efficiency of high performance computing (HPC) centers. Although a lot of scheduling algorithms have been proposed by industry and academia [2, 3], FCFS-based backfilling algorithm is regarded as the most efficient algorithm for modern HPC centers. Several variants of FCFS-based backfilling algorithms serve as the default setting in a lot of famous job scheduling systems, such as LSF [5], Moab [6], Maui [7], PBS/Torque [8] and LoadLeveler [9].

User estimated runtime is a key factor affecting performance of backfilling algorithms. The first study to identify the inaccuracy of user runtime estimates was Feitelson and Mu'alem Weil [24]. There are a lot of studies trying to improve the accuracy of user estimated runtime. Chiang et al. suggested a test run before running to acquire more accurate estimated runtime [13]. Zhai et al. developed a performance prediction tool to assist an accurate estimated time [14]. Tang et al. got usable information from historical information [15]. In order to improve the estimate accuracy, Cynthia Bailey Lee et al. gave a detailed survey. However, performance prediction is a very difficult problem for HPC users. Most of user

estimated runtime provided to scheduling systems is not accurate enough [16]. This results in very poor efficiency for scheduling systems.

Lots works suggested relaxing the backfilling conditions. These methods allowed jobs to be backfilled even that the estimated runtime are longer than the backfilling window. If the backfilled jobs can't finish in the specified period, uncompleted backfilled jobs are allowed to continue [15, 25, 26]. This strategy will postpone top-queue jobs.

Several works suggested preemption based backfill. Snell et al. studied aggressive backfill with kill-based preemption and discussed strategies that were used to select candidate preempted jobs [27]. Maui support PREEMPT backfill policy. It allows the scheduler to start backfill jobs even if required walltime is not available. If the job runs too long and interferes with another job which was guaranteed a particular timeslot, the backfill job is preempted and the priority job is allowed to run [28]. Perkovic et al. proposed "speculative backfilling" after regular backfilling [29]. If a speculatively run jobs runs longer than its speculated time, it will be killed. Most of these works focused on kill-based preemption. However, checkpoint-based preemption has different features than kill-based preemption. The latter hopes that preempted jobs have a short running time to reduce waste resources. It is more suitable for improving the short-run failure jobs. Our work uses checkpoint-based preemption, and it hopes that preempted jobs have a long running time to improve effect-cost ratio. It is more suitable for solving inaccurate user-provided runtime estimate.

Morris Jette et al. developed a preemption-based gang scheduler at Lawrence Livermore National Laboratory (LLNL) for the Cray T3D. The Gang Scheduler combines a checkpoint-based preemptive processor scheduler with the ability to relocate jobs within the pool of available processors. That approach can sustain machine utilization and provide interactive workload with a lower response time [30, 31]. Different, our work is focused on traditional backfilling algorithm.

6.2 Checkpoint/Restart Techniques

As the size of HPC systems increases, reliability is becoming more and more important for HPC users. Checkpoint/Restart technique has become a standard configuration for real systems.

A lot of system-level and application-level checkpoint techniques have been proposed. Berkeley Lab's Checkpoint/Restart (BLCR) is a system-level checkpoint technique, which is the Linux kernel-based coordinated checkpoint technique [17]. BLCR is integrated with LAM/MPI and OpenMPI to provide checkpoint and restart for parallel applications. IBM proposed a special application-level checkpoint library [32] for BlueGene/P applications. This library provides support for user initiated checkpoint. Users can insert checkpoint invocation manually in the application arbitrarily. Xue et al. propose a user-level file system which can guarantee the consistency between the application and its files during checkpoint and restart [33].

A lot of work tried to reduce the overhead of checkpoint. Liu et al. used exponential distributions to model the system failure, and proposed a reliability-

aware checkpoint/restart method [34]. Shastry et al. found that the optimal checkpoint interval was approximately directly proportional to the checkpoint cost while inversely proportional to shape parameter [21].

7 Conclusion

In this paper, we propose a checkpoint-based backfill algorithm, and evaluate it using real workload traces. Our analysis indicates that the checkpoint-based backfill algorithm can effectively improve the job scheduling in waiting time, slowdown and mean queue length by up to 40%. The checkpoint/restart overhead is also analyzed based on the real trace from Argonne Intrepid system. The results show that only 4% of the jobs need to be checkpointed due to preemption. This demonstrates that our checkpoint-based algorithm is able to improve overall system utilization considerably without spending significant amount of system resources on checkpoint/restart operations.

We plan to extend our work in several directions. We will develop an aging algorithm to adaptively tune the parameter p so as to achieve approximating optimal performance. Moreover, we will study other split policies, such as splitting the running process into three segments. We will also apply the checkpoint-ability to other job scheduling algorithms.

Acknowledgments. We want to express our thanks to anonymous reviewers who gave valuable suggestion that has helped to improve the quality of the manuscript. This work gets support partly from Chinese "863" project 2008AA01A204, 2010AA012403, NSFC project 61103021, NSF grants 0546301 (CAREER), 0915861, 0937908, and 0958311, in addition to a joint faculty appointment between Oak Ridge National Laboratory and NC State University, as well as a senior visiting scholarship at Tsinghua University.

References

1. Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam. An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration. *Parallel and Distributed Systems, IEEE Transactions on*, 14(3):236–247, 2003.
2. C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for iviukiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, 1993.
3. S. Majumdar, D.L. Eager, and R.B. Bunt. *Scheduling in multiprogrammed parallel systems*, volume 16. ACM, 1988.
4. D. Lifka. The ANL/IBM SP scheduling system. In *Job Scheduling Strategies for Parallel Processing*, pages 295–303. Springer, 1995.
5. Platform Computing Inc. Platform LSF. `http://www.platform.com/products/LSFfamily/`, 2012.
6. Adaptive Computing Enterprises Inc. MOAB workload manager. `http://www.supercluster.org/moab/`, 2012.

7. D. Jackson, Q. Snell, and M. Clement. Core algorithms of the Maui scheduler. In *Job Scheduling Strategies for Parallel Processing*, pages 87–102. Springer, 2001.
8. Adaptive Computing Enterprises Inc. PBS/Torque user manual. `http://www.clusterresources.com/torquedocs21/usersmanual.shtml`, 2012.
9. Joseph Skovira, Waiman Chan, Honbo Zhou, and David Lifka. The EASY – LoadLeveler API project. In *Job Scheduling Strategies for Parallel Processing*, pages 41–47. Springer, 1996.
10. Parallel Workloads Archive. `http://www.cs.huji.ac.il/labs/parallel/workload/`, 2012.
11. S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. Characterization of backfilling strategies for parallel job scheduling. In *Parallel Processing Workshops, 2002. Proceedings. International Conference on*, pages 514–519. IEEE, 2002.
12. W. Cirne and F. Berman. A comprehensive model of the supercomputer workload. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 140–148. IEEE, 2001.
13. S.H. Chiang, A. Arpaci-Dusseau, and M. Vernon. The impact of more accurate requested runtimes on production job scheduling performance. In *Job Scheduling Strategies for Parallel Processing*, pages 103–127. Springer, 2002.
14. J. Zhai, W. Chen, and W. Zheng. PHANTOM: predicting performance of parallel applications on large-scale parallel machines using a single node. In *ACM SIGPLAN Notices*, volume 45, pages 305–314. ACM, 2010.
15. W. Tang, N. Desai, D. Buettner, and Z. Lan. Analyzing and adjusting user runtime estimates to improve job scheduling on the Blue Gene/P. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–11. IEEE, 2010.
16. C. Bailey Lee, Y. Schwartzman, J. Hardy, and A. Snavely. Are user runtime estimates inherently inaccurate? In *Job Scheduling Strategies for Parallel Processing*, pages 253–263. Springer, 2005.
17. Berkeley Lab Checkpoint/Restart (BLCR). `https://ftg.lbl.gov/projects/CheckpointRestart/`, 2012.
18. J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. Plfs: A checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 21. ACM, 2009.
19. Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, and SL Scott. An optimal checkpoint/restart model for a large scale high performance computing system. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–9. IEEE, 2008.
20. G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of MPI programs. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 84–94. ACM, 2003.
21. M.S. PM and K. Venkatesh. Analysis of Dependencies of Checkpoint Cost and Checkpoint Interval of Fault Tolerant MPI Applications. *Analysis*, 2(08):2690–2697, 2010.
22. TOP500 Supercomputing web site. `http://www.top500.org`, 2012.
23. H. Naik, R. Gupta, and P. Beckman. Analyzing checkpointing trends for applications on the IBM Blue Gene/P system. In *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*, pages 81–88. IEEE, 2009.

24. D.G. Feitelson and A.M. Weil. Utilization and predictability in scheduling the ibm sp2 with backfilling. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998*, pages 542–546. IEEE, 1998.
25. W. Ward, C. Mahood, and J. West. Scheduling jobs on parallel systems using a relaxed backfill strategy. In *Job Scheduling Strategies for Parallel Processing*, pages 88–102. Springer, 2002.
26. D. Tsafrir, Y. Etsion, and D.G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *Parallel and Distributed Systems, IEEE Transactions on*, 18(6):789–803, 2007.
27. Q. Snell, M. Clement, and D. Jackson. Preemption based backfill. In *Job Scheduling Strategies for Parallel Processing*, pages 24–37. Springer, 2002.
28. Adaptive Computing Enterprises Inc. Preemption Policies. `http://www.adaptivecomputing.com/resources/docs/maui/8.4preemption.php`, 2012.
29. D. Perkovic and P.J. Keleher. Randomization, speculation, and adaptation in batch schedulers. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 7. IEEE Computer Society, 2000.
30. M.A. Jette. Performance characteristics of gang scheduling in multiprogrammed environments. In *Supercomputing, ACM/IEEE 1997 Conference*, pages 54–54. IEEE, 1997.
31. M. Jette, D. Storch, and E. Yim. Gang scheduler-timesharing the cray t3d. *Cray User Group*, pages 247–252, 1996.
32. C. Sosa and B. Knudson. IBM System Blue Gene/P Solution: Blue Gene/P Application Development. `http://www.redbooks.ibm.com/abstracts/sg247287.html`, 2007.
33. R. Xue, W. Chen, and W. Zheng. CprFS: a user-level file system to support consistent file states for checkpoint and restart. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 114–123. ACM, 2008.
34. Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, and SL Scott. An optimal checkpoint/restart model for a large scale high performance computing system. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–9. IEEE, 2008.