Guarantee Strict Fairness and Utilize Prediction Better in Parallel Job Scheduling

Yulai Yuan, Yongwei Wu, *Member, IEEE*, Weimin Zheng, *Senior Member, IEEE*, and Keqin Li, *Senior Member, IEEE*

Abstract—As the most widely used parallel job scheduling strategy, EASY backfilling achieved great success, not only because it can balance fairness and performance, but also because it is universally applicable to most HPC systems. However, unfairness still exists in EASY. Our simulation shows that a blocked job can be delayed by later jobs for more than 90 hours on real workloads. Additionally, directly employing runtime prediction techniques in EASY would lead to a serious situation called reservation violation. In this paper, we aim at guaranteeing strict fairness (no job is delayed by any jobs of lower priority) while achieving attractive performance, and employing prediction without causing reservation violation in parallel job scheduling. We propose two novel strategies, namely, shadow load preemption (SLP) and venture backfilling (VB), which are integrated into EASY to construct preemptive venture EASY backfilling (PV-EASY). Experimental results on three real HPC workloads demonstrate that PV-EASY is more attractive than EASY in parallel job scheduling, from both academic and industry perspectives.

Index Terms—Checkpoints, modeling and prediction, parallel system, scheduling, virtualization

1 Introduction

Parallel job scheduling is critical in large-scale high performance computing (HPC) systems, since different scheduling policies can result in different user experience and resource utilization. Fairness and performance are two eternal topics in parallel job scheduling, and previous works focus either on fair scheduling [16], [19], [20] or performance [13], [14], [23]. However, fairness and performance should be considered in concert. The first come first served (FCFS) method mainly guarantees fairness, while the shortest job first (SJF) method [6] mainly targets performance. This explains why they are rarely used alone in practice. EASY backfilling [1] allows later jobs to backfill in idle processors that cannot satisfy the request of the first blocked job, provided that these backfilled jobs would not delay the expected start time (reservation) of the first blocked job in the queue. EASY provides "relaxed" fairness to jobs via reservation, as well as achieves good performance by shifting shorter jobs forward. Because of a better balance between fairness and performance, EASY is the most widely used job scheduling strategy in HPC systems, and adopted by lots of major production schedulers [8].

In order to obtain better fairness while achieving attractive performance, many approaches have been proposed based on EASY. However, most of them suffer from the same problem that they mix up fairness with performance, and try to measure the fairness of schedulers with performance metrics, such as slowdown queuing fairness (SQF) [19], fair-slowdown [9], and fair start time [20]. We claim that fairness is independent of performance metrics, and propose our definition of strict fairness (no job is delayed by any jobs of lower priority). Therefore, our first objective is to guarantee strict fairness as well as achieve attractive performance in parallel job scheduling.

Another interesting phenomenon is that job runtime prediction technologies are rarely employed by EASY in real HPC systems, though EASY is built on highly inaccurate user estimates of jobs' runtime [3], [12]. This is due to two reasons. 1) There is a misconception that inaccurate user estimates of jobs' runtime can improve performance [3], [28]. 2) Replacing user estimates directly with system-generated prediction in EASY would lead to reservation violation (a blocked job cannot start at its reservation time (Section 3.2)). These two reasons cause lots of prediction techniques [24], [26] to be put on the shelf, even if they have attractive prediction accuracy. To solve above dilemma, our second objective is to employ prediction techniques without causing reservation violation.

In this paper, we propose a novel preemptive venture EASY backfilling (PV-EASY) method which integrates shadow load preemption (SLP) and venture backfilling (VB) into EASY. Our work makes two significant contributions.

First, we propose shadow load preemption to guarantee strict fairness and employ prediction techniques without causing reservation violation, which makes PV-EASY more attractive than EASY with regard to fairness. The load of a system is classified into *sunny load* and *shadow load* (Section 4.1). SLP (Section 4.2) can preempt the processors occupied by shadow load to start blocked jobs, and thus guarantee strict fairness and avoid reservation violation. Additionally, as observed from our experiments, the mean bounded slowdown (MBS) and mean weighted bounded slowdown (MWBS) of blocked jobs in PV-EASY are mostly lower than those in EASY.

- *Y. Yuan, Y. Wu, and W. Zheng are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: {yuanyulai, wuyw, zwm-dcs}@tsinghua.edu.cn.*
- *K. Li is with the Department of Computer Science, State University of New York, New Paltz, NY 12561. E-mail: lik@newpaltz.edu.*

Manuscript received 28 Nov. 2012; revised 26 Feb. 2013; accepted 10 Mar. 2013; date of publication 26 Mar. 2013; date of current version 21 Feb. 2014. Recommended for acceptance S.-Q. Zheng.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2013.88

Second, we propose venture backfilling to improve performance, which makes PV-EASY more attractive than EASY in terms of performance. VB (Section 4.3) can counteract the negative effects of preemption in SLP and improve performance. After backfill decisions are made based on runtime prediction, high priority jobs may still be venturesomely backfilled on idle processors, without considering job runtime and reservation. Benefiting from VB, PV-EASY with simple kill/restart mode and Last Model predictor achieves the same performance as EASY. Moreover, PV-EASY achieves better performance than EASY if both of them adopt the same job runtime prediction techniques.

We evaluate PV-EASY by means of real workloads from production HPC systems. Our results show that, due to low resource waste and simple implementations, PV-EASY can easily facilitate all HPC systems, which makes PV-EASY attractive in real production environments. Even with the simple kill/restart preemption mode and Last Model predictor, the wasted resources in PV-EASY are limited at a low level (2.48∼5.66 percent), and therefore would not affect system throughput even when the system load is up to 80 percent. Meanwhile, PV-EASY does not need the support of any complex system features and can be easily implemented in all HPC systems and production schedulers.

Additionally, current checkpoints and virtualization techniques can improve the performance of PV-EASY significantly, and reduce the resource waste dramatically. By replacing kill/restart mode in PV-EASY with checkpoint/restart mode (CP-PV-EASY) and suspend/resume mode (VM-PV-EASY), the job slowdown and resource waste are further improved.

The rest of this paper is organized as follows. Section 2 provides the background and related work of parallel job scheduling. In Section 3 we discuss the motivation of this paper, and then we propose our novel PV-EASY in Section 4. The experimental design and results are presented in Section 5. In Sections 6 and 7, we discuss and conclude.

2 Background and Related Work

In the past, besides the most natural and simplest FCFS strategy, many kinds of order-based parallel job scheduling strategies [4], [5] have been proposed, such as shortest job first, smallest job first [6], and smallest cumulative demand first [6], etc. However, all of them share similar inherent problems. 1) If the first waiting job is blocked, a "hole" of idle processors would appear as time goes by, and therefore results in low system utilization. 2) Most of the above order-based scheduling strategies, except FCFS, can cause unfairness, even starvation. A successful approach to solve these two problems is backfilling [1].

Backfilling has two versions, i.e., conservative and aggressive backfilling. Conservative backfilling only backfills jobs that would not delay any previous jobs in the queue, while aggressive backfilling takes a more aggressive approach that selects backfilled jobs provided that they would not delay the expected start time of the first job in the queue. Aggressive backfilling like EASY is reported to have better performance than conservative backfilling [2].

To enhance the performance of EASY, many variants have been proposed, and most of them can be classified into two categories according to their changes: 1) variants of reservation calculation; 2) variants of backfill selection.

Fairness and performance are two most important metrics of parallel job scheduling. Many existing works (e.g., [13], [14], [23]) tried to promote the performance of schedulers on certain metrics, including user-aware metrics (e.g., turnaround time, slowdown) and system-aware metrics (e.g., utility and energy saving). Meanwhile, users in queueing systems are sensitive to fairness [17], [18], and lots of scheduling studies focused on fairness.

EASY backfilling is built on jobs' runtime estimates given by users. However, user estimates of jobs' runtime are reported to be highly inaccurate [3], [12]. Even worse, most users are unable or reluctant to provide better job runtime estimates when they submit their jobs to HPC systems [22]. System-generated runtime prediction techniques [24], [26] are reported to have better accuracy than user estimates, but they are rarely integrated into EASY in real production systems, due to two misconceptions. 1) Inaccurate runtime estimates of jobs can improve performance [3]. 2) Users would not tolerate jobs being killed just because predictions were too short. In the rest of this paper, we use "estimate" to denote the requested time of a job forecasted by a user, and use "prediction" to indicate the result of system-generated runtime prediction.

Due to page limitation, more related work about backfill variants and fairness metrics are reviewed in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2013.88.

3 Design Objectives

The wide use of EASY in production schedulers and HPC systems has proved its practicability. Unfortunately, EASY and its variants still suffer from some serious problems that will be analyzed in this section. These analyses motivate us to propose a more powerful parallel job scheduler to guarantee strict fairness and employ prediction.

3.1 Guarantee Strict Fairness

Existing works of parallel job scheduling pay more attention to performance than to fairness. However, HPC users might be more sensitive to fairness than to performance [15]. We believe that fairness and performance are both important to attract users in parallel job scheduling.

Many existing works mix up fairness and performance by measuring the fairness through performance metrics [19], [20]. In fact, fairness is independent of performance and unsuitable to be measured by performance metrics. Fairness is more like a baseline that needs to be guaranteed and cannot be broken, like laws, rather than being judged by metrics which pay more attention to the average situation, like performance. For fairness, what users concern about is that they never want to be treated unfairly, or, they need services with guaranteed fairness.

What is the guaranteed fairness in parallel job scheduling? FCFS is an absolutely fair scheduling strategy, which holds the view that users are sensitive to service sequences,

TABLE 1
Start Time Delay of Blocked Jobs in EASY

Workload	$\#Job_{Delay}$ / $\#Job_{Blocked}$	Mean Delay Time of Delay Jobs	Maximum Delay Time
CTC	736/3833	80.05 min	14.81 hour
SDSC-BLUE	1770/10409	169.08 min	90.70 hour
SDSC-DS	827/4147	133.23 min	17.25 hour

and jumping in a queue is unacceptable. Actually, in a parallel system with N resources, most jobs cannot occupy all the system resources. Suppose the priority factor of jobs in a parallel system S is submission time. Job A is blocked because of insufficient idle resources in S. If the jobs submitted later than job A can run on these idle resources without delaying the start time of A, we believe that the user of A would not raise any complaint. Thus, we define the notion of "***strict fairness***" in parallel job scheduling as follows, namely, *if no job is delayed by any jobs with lower priority, this scheduling sequence can be viewed as strictly fair*.

EASY backfilling maintains a view of "relaxed" fairness that a job can be backfilled if it would not delay the "expected" start time (reservation) of the first blocked job. However, because of the "heel and toe" dynamic [29], the start time of blocked job is often gradually pushed away by later jobs unfairly according to the definition of strice fairness. We have observed significant unfairness in EASY from the experiment on three real workloads (Section 5.1). As summarized in Table 1, around 20 percent of the blocked jobs in each workload suffered from unfairness (delayed by later jobs, denoted as delay jobs). On the average, they were delayed 80, 169, and 133 minutes in three workloads respectively by later jobs. Moreover, the maximum delay caused by later jobs can be 90 hours or more (SDSC-BLUE).

In order to overcome the misconception of existing works about fairness and help the blocked jobs suffering from unfairness in EASY, we aim at designing a scheduler that can guarantee strict fairness. Besides, because fairness and performance are both attractive to users, we do not want to sacrifice performance in exchange for strict fairness. Thus, our design objective 1 is that *our scheduler should guarantee strict fairness to the jobs with attractive performance.*

3.2 Employ Prediction

Existing prediction techniques are rarely applied in real production schedulers, because of the misconceptions stated in Section 2. Tsafrir et al. [12] well clarified how these misconceptions failed and proposed an EASY variant that adopts prediction by separating the role of kill-time from prediction, but we found that directly replacing user estimates with prediction in EASY as it is proposed in [12] can lead to serious problems as analyzed below and demonstrated in Fig. 1 (left side is the scheduling decision and right side is the consequence). These problems are derived from inherent and unavoidable properties of prediction, i.e., *underestimate* (shorter than jobs' actual runtime) and *overestimate* (longer than jobs' actual runtime and shorter than user estimates). Notice that user estimates have been proven to be suitable to play the role of kill-time [12] and we hold the same view. Thus, a prediction that exceeds a user estimate is meaningless, and *overestimate* in this paper is between runtime and user estimates.

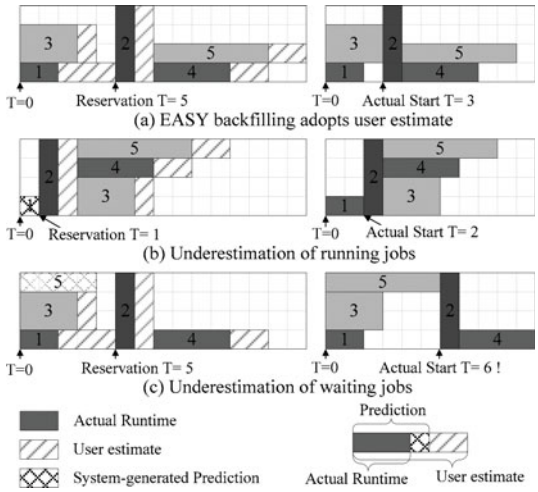


Fig. 1. Replacing user estimates with prediction in EASY.

Overestimate of running jobs' runtime might lead to resource waste. Because the prediction is shorter than the user estimate, the reservation time is shortened and fewer jobs can be backfilled, thus more holes are left and more computational resource is wasted. On the contrary, overestimate of waiting jobs can result in better performance, for more jobs can be backfilled because of shorter predictions compared with user estimates. These consequences of overestimate are easy to be deduced, and due to space limitation, they are not shown in Fig. 1.

Underestimate of running jobs' runtime would shorten the reservation of the first blocked job in the queue and therefore reduce the possibility of other waiting jobs being backfilled. Moreover, this reservation would be violated because the running jobs cannot actually finish before reservation and release sufficient processors for the blocked job (user estimates act kill-time). Fig. 1b demonstrates one possible case of this issue. However, this situation can be viewed as "benign", because the reservation of the blocked job is actually delayed by its predecessors, so the user of the blocked job would unlikely feel uncomfortable.

Underestimate of waiting jobs' runtime would lead to backfilling of a job whose execution time is actually longer than the reservation of the first blocked job, and thus pushing away the actual start time of the first blocked job to exceed its reservation. This undesired situation, which is called "reservation violation" in this paper, damages the original intention of reservation in backfilling. As shown in Fig. 1c, job 5 is underestimated and not thought to delay the start of blocked job 2, so job 5 is backfilled at $T = 0$. But actually job 5 cannot finish before $T = 5$ and therefore job 2's reservation guaranteed by EASY at $T = 5$ is violated.

To prove the existence of reservation violation, we implemented a prediction-based EASY backfilling by replacing user estimates with a predictor when calculating reservations and choosing backfilled jobs as [12] did, and performed a simulation on three real workloads. The predictor in this experiment is the Last Model, which predicts the *lifetime accuracy* (runtime/user estimates) of a job to be the same as the last job of the same user, and if no such predecessor exists, user estimates will be used instead. The "benign" situation illustrated in Fig. 1b was filtered out.

TABLE 2
RV Caused by Last Model in EASY

Workload	CTC	SDSC-Blue	SDSC-DS
#JobDelay	92	407	152
Mean_Parallelism$_{RV}$/ System Processors	147.78/ 430	533.48/ 1152	680.33/ 1664
Mean DTR (min)	100.19	154.76	110.69
Mean SI	4.78	36.13	40.78
Maximum DTR (hour)	15.13	32.90	16.62
Maximum SI	137.60	1320.63	1581.00

RV =Reservation Violation; DTR =Delay Time from Reservation; SI =Slowdown Increment caused by Reservation Violation

Simulation results are shown in Table 2. In each workload, tens or hundreds (92∼407) of jobs are suffered from serious reservation violation. The start time of these jobs are delayed more than 100 minutes from their reservations on average. In the worst cases, some jobs even experienced a reservation violation of more than 30 hours. Besides, the bounded slowdown (defined in Section 5.1) of these victims is dramatically increased. The mean *slowdown increment* (SI) is 4.78, 36.13, and 40.78 respectively. The maximum SI was even up to 1,581 in SDSC-DS. If a job's reservation is violated, its owner would have the illusion that this job is starving. Furthermore, most of these victims are large parallel jobs, which are the target jobs of HPC systems.

All above analyses indicate that prediction cannot be directly integrated into EASY as some previous works (e.g., [12], [28]) did, mainly due to the existence of reservation violation caused by unavoidable prediction errors. So, how can current studies of runtime prediction be able to facilitate parallel job scheduling, especially for the most widely used EASY in real production? We answer this question from a different perspective, i.e., to design a scheduler that can achieve our design objective 2: *a scheduler should employ prediction without causing reservation violation.*

4 Preemptive Venture Easy Backfilling

In this section, we first introduce our view about the classification of running load in EASY. Then, we propose a *shadow load preemptive* (SLP) backfilling scheduling scheme, which can guarantee strict fairness and employ prediction based on EASY. Afterwards, we propose a *venture backfilling* strategy, which is used to improve the performance of SLP. We integrate SLP and VB into the traditional EASY to form a new *preemptive venture EASY backfilling*. When a new job is submitted or a running job is finished, PV-EASY will first try to schedule jobs according to their priorities, until the idle resource is not sufficient for the highest priority job in the queue. Then PV-EASY will start SLP, and afterwards start VB.

4.1 Classification of Running Load in EASY

If a running job's priority is higher than all jobs in the waiting queue, no one can question its right of holding resources, and it seems running under sunshine. So the running load consisting of this kind of jobs is called "*sunny load*". On the contrary, if a running job's priority is lower than any waiting jobs, one might consider that this running job got its resources through improper means, and therefore it is more likely running in the shadow (not willing to be noticed by others). We call the running load that consists of this kind of running jobs "*shadow load*".

In EASY, a job can be either "regularly" started (regular job) if it holds the highest priority, or started by backfilling. Every regular job has the highest priority in the queue and deserves its running, so it definitely belongs to the *sunny load* during its whole lifetime. Backfilled jobs start only when the idle resources cannot satisfy the highest priority job in the queue, so they belong to the *shadow load* at the beginning. However, a backfilled job could transit from *shadow load* to *sunny load* during its lifetime.

4.2 Shadow Load Preemption

Shadow load is the root cause of unfairness in EASY. Recall from the definition of strict fairness, unfairness always happens in the blocked jobs of EASY. Because of inaccurate estimates of the blocked jobs' reservation and the backfilled jobs' runtime, the start time of a blocked job could be delayed by the backfilled jobs. It is clear that the jobs that cause unfairness all belong to *shadow load*.

Moreover, *shadow load* could lead to reservation violation and thus hinders the employment of prediction technologies in EASY. Underestimate of waiting jobs' runtime would mislead EASY to backfill long runtime jobs and therefore cause reservation violation (Section 3.2). These backfilled jobs with long runtime also belong to *shadow load*.

Jobs in the *shadow load* must be unnoticeable. They should not delay the running of other jobs, especially the blocked jobs with higher priorities. Unfortunately, *shadow load* is treated the same as *sunny load* in EASY, which exposes the existence of jobs in the *shadow load* and affects blocked jobs, thus leading to unfairness and reservation violation. One approach to solve this inherent issue of EASY is to restrict the activities of the jobs in the *shadow load* within the scope of real "invisible shadow", and prevent them from delaying the start of higher priority jobs, by implementing *shadow load preemption* as follows:

When idle resources of a system are not sufficient for the highest priority job in the waiting queue, the resources occupied by the jobs in the *shadow load* can be preempted, if this preemption can enable the highest priority job to start right away. The preemption occurs according to job priorities, from low to high, and the preempted jobs are put back to the waiting queue.

Because of preemption in SLP, the backfilled jobs of the *shadow load* would no longer delay the start of the blocked jobs and thus strict fairness is guaranteed. If the jobs in the *shadow load* are underestimated by the predictor, SLP ignores their existence and preempts their resources. Therefore, the risk of reservation violation is eliminated.

Three modes exist in SLP, kill/restart, checkpoint/restart, and suspend/resume. Kill/restart mode is the default setting in SLP, but for the HPCs and jobs which support checkpoints or virtualization techniques, a preempted job can either restart from its latest checkpoint, or suspended when preempted and resume from its saved status.

TABLE 3
An Overview of the Workloads

Workload	Duration	#Processors	#Jobs	Load (%)	Mean Parallelism	Mean Runtime
CTC	Jun 1996 ~ May 1997	430	77222	66.18	10.9853	11277 s
SDSC-BLUE	Apr 2000 ~ Jan 2003	1152	223407	76.21	41.5910	4381 s
SDSC-DS	Mar 2004 ~ Apr 2005	1664	85003	63.02	60.9240	7569 s

4.3 Venture Backfilling

Kill/restart mode in SLP is simple, universally supported, but also resource costly. If a backfilled job in the *shadow load* is preempted, all the work it has already done will be totally lost and it has to restart from its origin next time.

Checkpoint/restart and suspend/resume modes supported by checkpoint and virtualization technologies are two better ways to reduce the cost of preemption. Checkpoints can enable a preempted job restart from the closest saved status, and a job running in virtualization environment can suspend and resume at anytime theoretically without wasting any work it has already done. But both of these two modes rely on much more complex techniques which also bring slowdown to the running jobs and system.

In addition, due to inaccurate runtime prediction, though a waiting job can actually finish before future preemption happens, it might be misunderstood by SLP that it cannot survive from possible preemption. In this situation, these jobs cannot be backfilled and some holes would appear without full utilizing system resources.

In order to maximize the surviving opportunity of backfilled jobs in the *shadow load* and increase the utilization of the system, so as to reduce resource waste and improve performance, we propose a *venture backfilling* method:

1. Compute the reservation time of the first blocked job in the waiting queue based on *sunny load* and runtime prediction.
2. Determine the possible runtime of waiting jobs by employing system-generated prediction.
3. Select waiting jobs that can be satisfied by idle resources and have the largest likelihood of successful completion before preemption to be backfilled, from the nearest predicted completion time to the furthest within the reservation time.
4. If there still exist idle resources, select waiting jobs to be backfilled according to their priorities, from high to low, no matter whether the prediction indicates they could complete before possible preemption or not.

Step 3 aims to reduce the occurrence of preemption, and Step 4 tries to make full use of the resources. Step 4 is a novel but adventurous approach that is different from existing EASY variants. It seems that jobs backfilled in Step 4 have very little chance to survive from preemption. But this is not true. Prediction is always inaccurate and preemption in SLP occurs from low priority to high priority, so backfilled jobs of high priority in Step 4 would still have opportunities to transit to *sunny load* and complete.

5 Experimental Evaluation

In this section, we first introduce the experiment design. Then, we analyze PV-EASY (default with kill/restart mode) on four important aspects, including the benefit of maintaining strict fairness, employing prediction, performance, and resource waste. In Section 5.3, the performance of CP-PV-EASY (with checkpoints/restart mode) and VM-PV-EASY (with suspend/resume mode) are analyzed in detail.

5.1 Experiment Design

We have constructed an event-based simulator to mimic different scheduling strategies in generic parallel computing clusters, and the workload traces used to drive our simulator and evaluate PV-EASY are collected from real HPC systems. We selected three workload traces (CTC, SDSC-BLUE, and SDSC-DS) from parallel workload archive (PWA) [30]. These workload traces are all named by the names of their HPC systems and affiliations. Table 3 gives an overview of them. They are collected during long production periods and contain large amount of job entries, with regular load between 60 and 80 percent. Jobs in these workloads use tens of processors on average, and their mean runtime always exceed 1 hour. More details about the simulator and workloads are available in Appendix B, available in the online supplemental material.

We only use EASY as a comparison target in our experiments, because the default setting of most parallel schedulers remains plain EASY [8], and furthermore, it is statistically reported that 90∼95 percent of the parallel scheduler installations do not change this default configuration [7].

The priority factor in our experiments is the submission time of a job. If not specified, the job runtime predictor used in each of the scheduling strategies is the Last Model.

In this study, two metrics, mean bounded slowdown and mean weighted bounded slowdown, are used to evaluate user-aware performance in parallel job scheduling. Slowdown is defined as turnaround time (waiting time + running time) normalized by running time. Bounded slowdown eliminates the influence of very short jobs on the metric [3], and it is defined as follows:

$$Bounded\_Slowdown = \frac{Waittime + Max(Runtime, 10)}{Max(Runtime, 10)}. \quad (1)$$

In this paper, we use a threshold of 10 seconds, which is often used in existing works.

MBS is an arithmetic mean value of all jobs' bounded slowdown. Every job is regarded as equal in MBS implicitly, without considering the number of processors that a job used. In fact, the purpose of building HPC systems is to enable the running of large parallel jobs rather than serial jobs. By considering the number of processors used

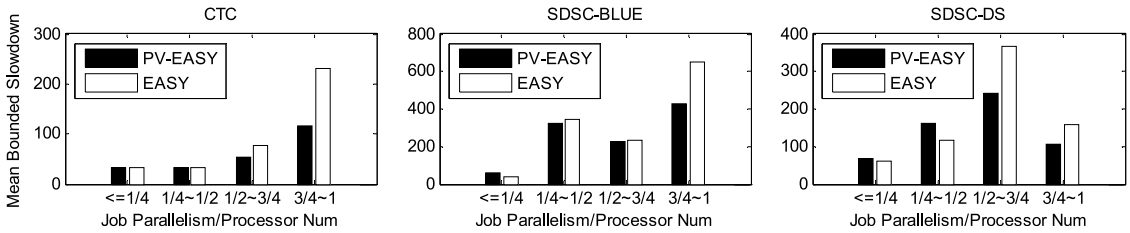


Fig. 2. Mean bounded slowdown comparison of blocked Jobs in PV-EASY and EASY.

by each job $(Parallelism_j)$ as weight, we propose MWBS as follows:

$$MWBS = \frac{\sum_{j=0}^{N-1} (Bounded\_Slowdown_j \times Parallelism_j)}{\sum_{j=0}^{N-1} Parallelism_j}. \quad (2)$$

We adopt system load to measure system-aware performance, which is computed as CPUTime consumed by all jobs divided by the total CPUTime available in the system.

5.2 Performance of PV-EASY

In this section, we demonstrate how PV-EASY achieves two attractive objectives: 1) guaranteeing strict fairness with good performance, and 2) employing prediction without causing reservation violation. In addition, the problem of resource waste in PV-EASY is also analyzed.

5.2.1 Benefits of Maintaining Strict Fairness

Benefiting from *shadow load preemption*, strict fairness is guaranteed in PV-EASY. Therefore, there is no need to measure PV-EASY with fairness metrics. Instead, advantages of maintaining strict fairness in PV-EASY can still be demonstrated from another perspective, i.e., how blocked jobs benefit from guaranteed strict fairness.

As shown in Fig. 2, we compare the performance of the blocked jobs between EASY and PV-EASY. The blocked jobs are grouped by their parallelism. The MBS of big blocked jobs (job parallelism larger than 1/4 of the system processor number) in PV-EASY are mostly smaller than that those in EASY. As shown in Fig. 3, in terms of the MWBS of the blocked jobs, big blocked jobs are also better treated in PV-EASY than in EASY.

Small blocked jobs (job parallelism smaller than 1/4 of system processor number), as shown in Figs. 2 and 3, do not receive better treatment in PV-EASY. This performance degradation of small blocked jobs in PV-EASY is a benign consequence of guaranteeing strict fairness, because fairness and performance are always a tradeoff, and we will further demonstrate that the overall performance of PV-EASY is also attractive in Section 5.2.3. More analysis of small blocked jobs is given in Appendix C, available in the online supplemental material.

5.2.2 Employing Prediction

Another objective of PV-EASY is to employ prediction without causing reservation violation. By applying preemption in PV-EASY, reservation violation can be theoretically and practically prevented. Moreover, PV-EASY provides better supports to prediction than EASY.

In this part of the experiments, to present a fair comparison, besides commonly used EASY backfilling which employs FCFS strategy to choose backfilled jobs (denoted as FCFS-EASY), a SJF-EASY which selects backfilled jobs by SJF is also introduced, because PV-EASY also selects backfilled jobs according to SJF.

Moreover, we proposed a "virtual" predictor to replace the Last Model in PV-EASY, and also replaced user estimates (Job Request Time) in FCFS-EASY and SJF-EASY. This "virtual" predictor generates prediction with the maximum error of $\pm x\%$ (implemented by setting the prediction to be $runtime \times (1 + random(-x\% \sim x\%))$, and the mean absolute prediction error of this "virtual" predictor is around $x/2$). For every scheduling strategy implemented on every trace with every x value, we repeated the simulation 10 times, and then report the mean result.

By employing the same prediction technique, PV-EASY achieves better performance than EASY. As shown in Fig. 4, PV-EASY outperforms FCFS-EASY and SJF-EASY when runtime prediction error is bounded within maximum 10 percent (mean absolute prediction error is around 5 percent). Considering that existing parallel job runtime prediction techniques (e.g., [24], [26], [31]) have been reported to achieve mean absolute prediction error of more than 20 percent (corresponding to the maximum runtime prediction error 40 percent in Fig. 4), we believe

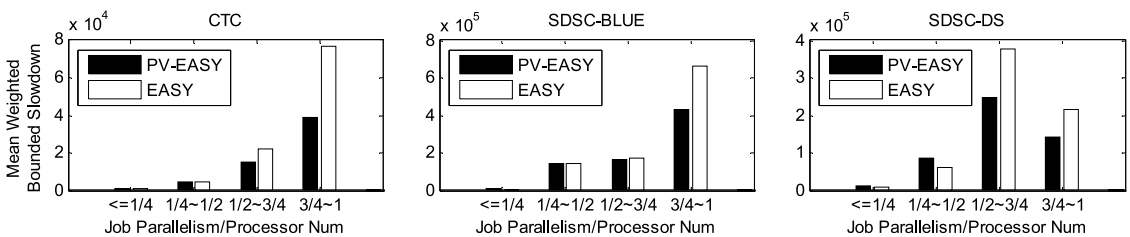


Fig. 3. Mean weighted bounded slowdown comparison of blocked Jobs in PV-EASY and EASY.

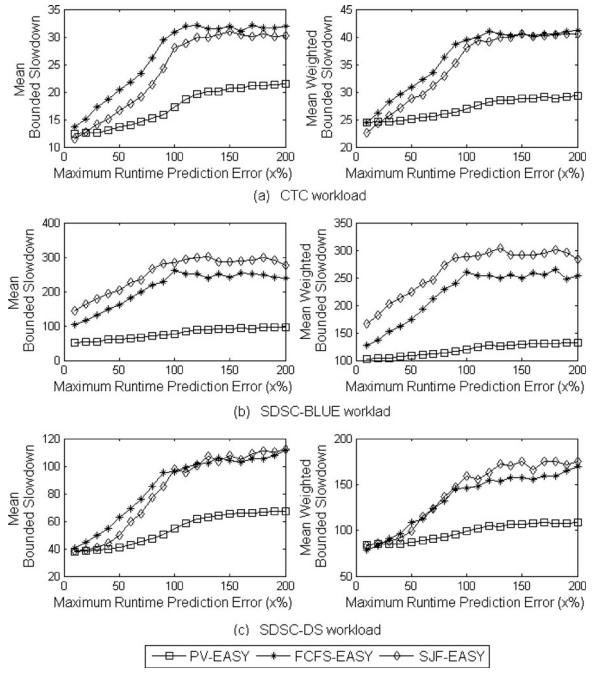


Fig. 4. Performance comparison among "virtual" predictor integrated PV-EASY, FCFS-EASY and SJF-EASY.

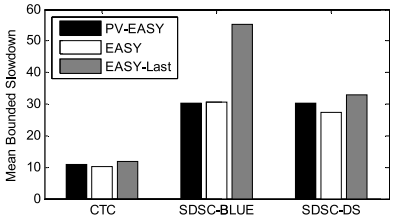


Fig. 5. Mean bounded slowdown comparison.

that PV-EASY can much better support prediction techniques than EASY in the long term.

5.2.3 Performance Comparison with EASY

In SLP, kill/restart preemption mode would cause computational resource waste and result in performance degradation. On the other hand, computational resources might still be left idle in SLP because of inaccurate prediction. So we proposed *venture backfilling* to solve these performance problems. Figs. 5 and 6 show the comparison between PV-EASY and EASY. It is clear that PV-EASY achieves smaller (in SDSC-BLUE) or similar (in CTC and SDSC-DS) MBS and MWBS as EASY. In order to eliminate the doubt that such performance of PV-EASY benefits from prediction method that is more accurate than user estimates, we also compare PV-EASY with EASY-Last (user estimates are replaced by Last Model in EASY). As shown in Fig. 6, EASY-Last is the worst, and we are therefore convinced that *venture backfilling* successfully promote the performance as we expected. Thus, we conclude that PV-EASY can better balance fairness and performance than EASY.

5.2.4 Resource Waste

Kill/restart preemption in PV-EASY is simple but resource costly. We analyze the total load and wasted

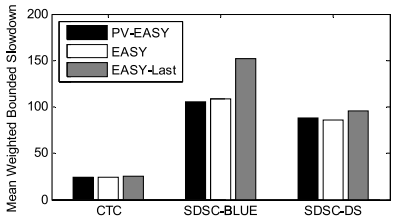


Fig. 6. Mean weighted bounded slowdown comparison.

Table 4
Load of PV-EASY in Three Workloads

Workload	Total Load (%)	Wasted Load (%)
CTC	68.6166	2.48
SDSC-BLUE	81.7564	5.66
SDSC-DS	66.4917	3.67

TABLE 5
Preempted Jobs in PV-EASY

Workload	$\#Job_{Preempted}$ / # Total Job	Rate (%)	MKT	Mean RTW (%)
CTC	10172/77222	13.17	1.72	38.95
SDSC-BLUE	17364/223407	7.77	1.46	48.27
SDSC-DS	10294/85003	12.11	1.62	45.33

load of the three workloads in PV-EASY and listed the results in Table 4. Notice that the definition of load in Table 4 is a little bit different from that in Table 3, because the system capability here is defined as (system processor number) X (the time when all jobs finish—the submission time of the first job).

It is clear that the wasted load of PV-EASY is relatively small (2.48 to 5.66 percent) and it does not worsen the system throughput. Each workload (the load varies from 63.02 to 76.21 percent, Table 3) finishes within the same time in PV-EASY and EASY. This result can be explained as follows.

First, as shown in Table 5, preemption rate rangings from 7.77 percent (SDSC-BLUE) to 13.17 percent (CTC). This small proportion of preempted jobs indicates that PV-EASY does not disturb too many running jobs. Second, the impacts of kill/restart are not serious on these preempted jobs. In order to quantify these impacts, we employ two metrics, Mean Killed Times (MKT) and Runtime Waste (RTW). MTK counts the mean occurrences of killing among preempted jobs, and RTW is defined in formula

$$RTW = (Time_{sum} - Runtime)/Runtime, \qquad (3)$$

where $Time_{sum}$ is the accumulative runtime of a job (actual runtime + runtime before preemption happens). The results of MKT and RTW of PV-EASY are shown in Table 5. On the average, preempted jobs in three workloads were killed less than twice (1.72, 1.46, and 1.62 respectively). Besides, these preempted jobs spent only 40~50 percent additional time of their actual running time. Based on the results of MKT and mean RTW, we conclude that kill/restart mode does not significantly impact these preempted jobs.

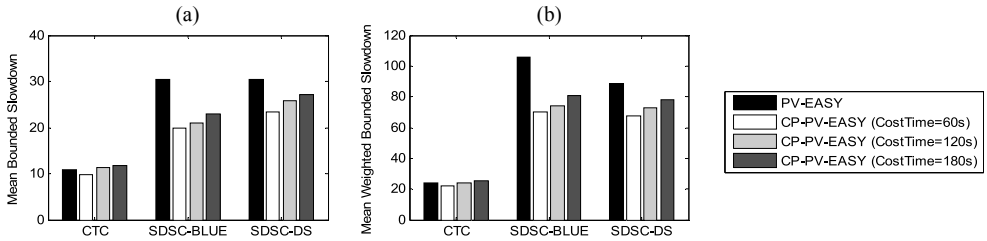


Fig. 7. Mean bounded slowdown and mean weighted bounded slowdown comparison between PV-EASY and CP-PV-EASY.

5.3 Performance Optimization of PV-EASY

In PV-EASY, SLP with kill/restart mode would waste the work that a job has already done. Checkpoint and virtualization can be used to decrease this kind of resource waste, hence promoting the performance of PV-EASY in HPCs.

In this section, we employ checkpoint/restart and suspend/resume modes to replace kill/restart mode in PV-EASY, and try to figure out whether current checkpoints and virtualization technologies can well facilitate PV-EASY and promote the performance of parallel job scheduling. PV-EASY with checkpoints (checkpoint/restart mode) is denoted as CP-PV-EASY, and PV-EASY with virtualization (suspend/resume mode) is denoted as VM-PV-EASY.

5.3.1 PV-EASY with Checkpoints

The overhead of checkpoint techniques cannot be ignored in a real environment. Thus we propose a simple checkpoint/restart model in CP-PV-EASY and experiment to simulate this overhead, which has two assumptions: 1) Checkpoint functions every $Interval_{checkpoint}$ for every job; 2) Every pair of checkpoint and restart operations costs constant times, i.e., $TimeCost_{checkpoint}$ and $TimeCost_{restart}$.

Many previous works evaluated the overhead of checkpoints, including traditional full checkpoints and incremental checkpoint schemes. Wang et al. [10] evaluated the overhead of several checkpoint strategies on different benchmarks, such as the MPI version of NPB suite [11] that includes BT, CG, LU, FT and SP, as well as mpiBLAST [21], and their methods can reduce the checkpoint overhead of above benchmarks by no more than 50 seconds, and keep the restart overhead within 10 seconds. According to these results, we set the typical value of each parameter in our checkpoint/restart model as: $Interval_{checkpoint} = 1$ hour, $TimeCost_{checkpoint} = 50$ seconds, and $TimeCost_{restart} = 10$ seconds.

To simplify the model, we propose $TimeCost = (TimeCost_{checkpoint} + TimeCost_{restart})$ as a basic combined parameter, and set the cost of every checkpoint or checkpoint&restart operation to be *TimeCost*. It is obvious that this simplification simulates the worst case. In order to evaluate PV-EASY with different level of overhead, experiments are performed under different *TimeCost* (60, 120, and 180 s).

In this experiment, we focus on two important metrics, job slowdown and resource cost.

Slowdown. As shown in Fig. 7, CP-PV-EASY performs significantly better than PV-EASY when the *CostTime* is limited in 60 seconds. CP-PV-EASY can reduce 9.00, 34.81 and 22.97 percent of the MBS (Fig. 7a) compared with PV-EASY on three workloads. Even when the *CostTime* is up to 180 s, CP-PV-EASY can also achieve 24.21 and 10.90 percent lower MBS than PV-EASY on SDSC-BLUE and SDSC-DS. The similar results happen on MWBS (Fig. 7b).

Some exceptions appear on CTC in Fig. 7. When *CostTime* rises to 120 seconds, MBS of CP-PV-EASY is 5.11 percent greater than that of PV-EASY. When *CostTime* is as large as 180 seconds, MBS and MWBS of CP-PV-EASY are all slightly greater than PV-EASY.

Above exceptions happened on CTC due to its characteristics. As shown in Table 3, CTC has the longest mean runtime among three workloads, 11,277 seconds, while SDSC-BLUE is 4,381 seconds and SDSC-DS is 7,569 seconds. In our experiments, $Interval_{checkpoint}$ is set to be 1 hour, which means each job in CTC would save three checkpoints on average, while SDSC-BLUE saves only 1, and SDSC-DS saves only 2. In this case, the jobs of CTC would experience more checkpoints operations than other two HPCs, and therefore MBS and MWBS of CTC increase more significantly.

Resource Cost. As shown in Table 6, when *CostTime* is 60 and 120 seconds, preempted job count of CP-PV-EASY is smaller than that of PV-EASY on three workloads. Only when *CostTime* is 180 seconds, preempted job count of CP-PV-EASY is greater than that of PV-EASY on CTC and SDSC-DS. Almost the same situation happens on metric "wasted load" as shown in Table 8. CP-PV-EASY performs better than PV-EASY when *CostTime* is 60 and 120 seconds, while wasted load of CP-PV-EASY increases significantly when *CostTime* rise to 180 seconds. For the mean preempted times (MPT) shown in Table 7, CP-PV-EASY outperforms PV-EASY for different levels of *CostTime*.

The above resource cost experiment results indicate that in modern HPC systems, which always run large scaled parallelism jobs, like SDSC-BLUE and SDSC-DS, checkpoints can optimize the performance of PV-EASY if we can control the *CostTime* of checkpoint/restart within 120 seconds and checkpoint functions every

TABLE 6
Preempted Job Count Comparison

		CTC	SDSC-BLUE	SDSC-DS
PV-EASY		10172	17364	10433
CP-PV-EASY	CostTime=60s	9621	16320	9813
	CostTime=120s	9998	16640	10150
	CostTime=180s	10499	16715	10502

TABLE 7
Mean Preempted Times Comparison

		CTC	SDSC-BLUE	SDSC-DS
PV-EASY		1.72	1.46	1.62
CP-PV-EASY	CostTime=60s	1.680	1.396	1.551
	CostTime=120s	1.705	1.413	1.555
	CostTime=180s	1.696	1.414	1.560

TABLE 8
Wasted Load Comparison

		CTC	SDSC-BLUE	SDSC-DS
PV-EASY		2.48%	5.66%	3.37%
CP-PV-EASY	CostTime=60s	1.70%	2.13%	1.94%
	CostTime=120s	2.75%	3.28%	2.91%
	CostTime=180s	3.80%	4.43%	3.88%

hour. When *CostTime* is as big as 180 seconds, the cost becomes so big that the benefit of checkpoint/restart would be counteracted.

5.3.2 PV-EASY with Virtualization

Some recent works, like [25], propose the techniques that limit scalable virtualization with <5 percent overhead for key HPC applications in a high-end message-passing parallel supercomputer, e.g., a Cray XT4 supercomputer [27] at scale in excess of 4,096 nodes. Based on these researches, we will figure out whether PV-EASY could benefit from current virtualization technology on HPC systems.

We propose a virtualization model that contains three key parameters: 1) $Slowdown_{virtualization}$, slowdown ratio of a job running on a virtualization platform; 2) $TimeCost_{suspend}$, time cost of the suspend operation; 3) $TimeCost_{resume}$, time cost for a job to resume from suspended status. The actual values of these parameters vary differently among HPC jobs, but the workloads and platforms used in the paper did not have these informations. Thus, this model has fixed $Slowdown_{virtualization}$, and $TimeCost = (TimeCost_{suspend} + TimeCost_{resume})$ for every job and every suspend&resume operation, without considering jobs' parallelism, runtime, and type.

Based on previous work in virtualization [25], we set the typical values of these parameters as $Slowdown_{virtualization} = 5$ percent and $TimeCost = 60$, 120, and 180 seconds, and we compare the performance of PV-EASY with virtualization suspend/resume mode with PV-EASY (VM-PV-EASY) in two aspects: slowdown of jobs and the resource cost.

Slowdown. As shown in Fig. 8, VM-PV-EASY performs so well on MBS and MWBS, even *CostTime* is up to 180 seconds. In SDSC-BLUE and SDSC-DS, the MBS of VM-PV-EASY outperforms PV-EASY significantly, while in CTC the MBS of VM-PV-EASY is slightly greater than that of PV-EASY. The reason of this phenomenon in CTC is caused by the character of its workloads. The workload of CTC has much longer mean runtime than SDSC-BLUE and SDSC-DS. In our virtualization model, longer original runtime plus a fixed $Slowdown_{virtualization}$ will result in a longer additional runtime. Obviously, longer additional runtime will increase the probability of being preempted in SLP, therefore causes more suspend/resume operation and increases the overhead in VM-PV-EASY.

TABLE 9
Preempted Job Counts Comparison

Schedule Strategy		CTC	SDSC-BLUE	SDSC-DS
PV-EASY		10172	17364	10433
VM-PV-EASY	CostTime = 60s	10489	16692	10334
	CostTime = 120s	10486	16642	10330
	CostTime = 180s	10442	16798	10326

TABLE 10
Mean Preempted Times Comparison

Schedule Strategy		CTC	SDSC-BLUE	SDSC-DS
PV-EASY		1.72	1.46	1.62
VM-PV-EASY	CostTime = 60s	1.640	1.345	1.510
	CostTime = 120s	1.649	1.357	1.519
	CostTime = 180s	1.6494	1.359	1.516

TABLE 11
Wasted Load Comparison

Schedule Strategy		CTC	SDSC-BLUE	SDSC-DS
PV-EASY		2.48%	5.66%	3.67%
VM-PV-EASY	CostTime = 60s	3.33%	3.84%	3.17%
	CostTime=120s	3.32%	3.88%	3.21%
	CostTime=180s	3.39%	3.92%	3.25%

Resource Cost. As shown in Table 9, preempted job count of VM-PV-EASY is slightly less than PV-EASY in SDSC-BLUE and SDSC-DS, while a little greater than PV-EASY in CTC. Furthermore, in Table 10, the Mean Preemption Times (MPT) of VM-PV-EASY outperforms PV-EASY on three different *CostTime* level. Table 11 illustrates the wasted load of VM-PV-EASY on different workload, and it is clear that VM-PV-EASY wastes less resource than PV-EASY in SDSC-BLUE and SDSC-DS, while VM-PV-EASY causes more resource waste in CTC.

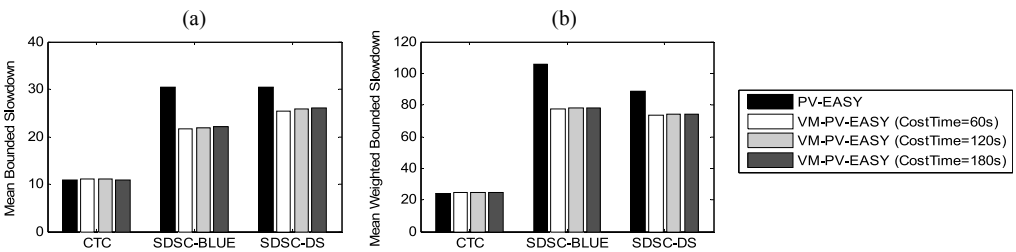


Fig. 8. Mean bounded slowdown and mean weighted bounded slowdown comparison between PV-EASY and VM-PV-EASY.

The above resource cost results indicate that compared with the kill/restart mode, the suspend/resume mode leads to less resource waste. The exception happening in CTC workload is related to its characteristics. CTC has the smallest mean RTW (shown in Table 5), which means that PV-EASY with the kill/restart mode in CTC workloads wasted the smallest resource. In this kind of workload, the overhead of virtualization and suspend/resume operation in PV-EASY is greater than performance optimization gain that suspend/resume mode could achieve.

6 Discussion

In order to truly facilitate HPC systems, PV-EASY employs the simplest and universally applicable mechanisms. Both the kill/restart mode and the Last Model are easily implemented and supported by all HPCs, which makes it easy for PV-EASY to replace EASY in all production schedulers.

In our experiments, CP-PV-EASY (checkpoint/restart mode) and VM-PV-EASY (suspend/resume mode) work so well in modern HPCs which are running highly parallelism jobs like SDSC-BLUE and SDSC-DS. We believe that with the development of virtualization and checkpoints techniques, their operation costs would decrease as technology advances, and they will facilitate modern highly parallel HPC systems not only in the consideration of fault tolerance, but also in scheduling strategies like PV-EASY.

Another noticeable phenomenon is that even with poor accuracy, EASY with user estimates achieves smaller MBS and MWBS than both EASY-Last (Figs. 5 and 6) and EASY with a "virtual" predictor (Fig. 4) whose maxi-mum prediction error is limited within 10 percent. This phenomenon once led to a pessimistic view that accurate prediction is not useful in parallel job scheduling. Actually, when user estimates are inaccurate, a "heel and toe" dynamic [29] would occur, making EASY approximate to SJF, and therefore achieveing good performance via serious sacrifice of fairness. Thus, in our view, runtime prediction is helpful, since it can enable schedulers to make better scheduling decisions to balance fairness and performance.

7 Conclusion

EASY backfilling is one of the most widely applied parallel job scheduling strategies in production schedulers. However, jobs scheduled by EASY may suffer from serious unfairness, and EASY cannot directly support prediction because this would cause reservation violation. In this paper, we proposed a new *preemptive venture EASY backfilling* strategy, which integrates novel *shadow load preemption* and *venture backfilling* approaches. Experiments on three workloads collected from real HPC systems show that our PV-EASY is very attractive from both academic and industry perspectives in the following aspects.

PV-EASY leverages fairness and performance much better than EASY in parallel job scheduling. PV-EASY guarantees strict fairness because of SLP, and achieves attractive performance compared with EASY due to VB.

PV-EASY benefits more from prediction techniques than EASY. PV-EASY avoids reservation violation that arises from employing prediction in EASY. Moreover, with the same runtime predition techniques, PV-EASY achieves much better performance than EASY.

PV-EASY is easy to implement and not resourcecosting. It is applicable to all kinds of HPC systems and production schedulers where EASY works, without introducing any additional system or application modifications.

PV-EASY with the checkpoints/restart and the suspend/resume modes provide better schedule performance and less resource waste. Modern HPC systems with high parallelism and fully supported by checkpoints or virtualization will benefit more from PV-EASY.

Acknowledgments

The comments of three anonymous reviewers are acknowledged. The authors would like to thank Dror Feitelson for his great work of collecting and publishing HPC workloads in Parallel Workload Archive. This work was supported by National Basic Research (973) Program of China (2011CB302505), Natural Science Foundation of China (60963005, 61170210), National High-Tech R&D (863) Program of China (2012AA012600, 2011AA01A203), Chinese Special Project of Science and Technology (2012ZX01039001). The work of K. Li was partially performed while he was visiting Tsinghua University during the summer of 2012 and winter of 2013 as an Intellectual Ventures endowed visiting chair professor.

References

[1] D.A. Lifka, "The ANL/IBM SP Scheduling System," *Proc. First Workshop Job Scheduling Strategies for Parallel Processing*, pp. 295-303, 1995.

[2] D.G. Feitelson, "Experimental Analysis of the Root Causes of Performance Evaluation Results: A Backfilling Case Study," *IEEE Trans. Parallel and Distributed Systems*, vol. 16, no. 2, pp. 175-182, Feb. 2005.

[3] A.W. Mu'Alem and D.G. Feitelson, "Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM Sp 2 with Backfilling," *IEEE Trans. Parallel and Distributed Systems*, vol. 12, no. 6, pp. 529-543, June 2001.

[4] D. Karger, C. Stein, and J. Wein, "Scheduling Algorithms," *CRC Handbook of Computer Science*. CRC Press, 1997.

[5] J. Sgall, "On-line Scheduling," *On-Line Algorithms*, A. Fiat and G. Woeginger, eds., pp. 196-231, , 1998.

[6] S. Majumdar, D.L. Eager, and R.B. Bunt, "Scheduling in Multiprogrammed Parallel Systems," *ACM SIGMETRICS Performance Evaluation Rev.*, vol. 16, no. 1, pp. 104-113, 1988.

[7] D. Jackson, "Maui/Moab Default Configuration," with CTO of Cluster Resources, 2006.

[8] Y. Etsion and D. Tsafrir, "A Short Survey of Commercial Cluster Batch Schedulers," Technical Report 2005-13, The Hebrew Univ. of Jerusalem, May 2005.

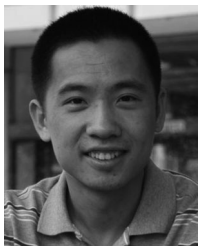
[9] S.H. Chiang, A. Arpaci-Dusseau, and M.K. Vernon, "The Impact of More Accurate Requested Runtimes on Production Job Scheduling Performance," *Proc. Eighth Workshop Job Scheduling Strategies for Parallel Processing*, pp. 103-127, 2002.

[10] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Hybrid Full/Incremental Check-Point/Restart for MPI Jobs in HPC Environments," *Proc. Int'l Conf. Parallel and Distributed Systems*, 2011.

[11] F.C. Wong, R.P. Martin, R.H. Arpaci-Dusseau, D.T. Wu, and D.E. Culler, "Architectural Requirements and Scalability of the NAS Parallel Benchmarks," *Proc. ACM/IEEE Conf. Supercomputing*, p. 41, 1999.

[12] D. Tsafrir, Y. Etsion, and D.G. Feitelson, "Backfilling Using System-Generated Predictions Rather than User Runtime Estimates," *IEEE Trans. Parallel and Distributed Systems*, vol. 18, no. 6, pp. 789-803, June 2007.

[13] R. Kurian, P. Balaji, and P. Sadayappan, "Opportune Job Shredding: An Effective Approach for Scheduling Parameter Sweep Applications," *Proc. Los Alamos Computer Science Inst. Symp.*, 2003.
[14] G. Sabin, R. Kettimuthu, A. Rajan, and P. Sadayappan, "Scheduling Of Parallel Jobs in a Heterogeneous Multi-Site Environment," *Proc. Ninth Workshop Job Scheduling Strategies for Parallel Processing*, pp. 87-104, 2003.
[15] E. Shmueli and D.G. Feitelson, "On Simulation and Design Of Parallel-Systems Schedulers: Are We Doing the Right Thing?" *IEEE Trans. Parallel and Distributed Systems*, vol. 20, no. 7, pp. 983-996, July 2009.
[16] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair Scheduling for Distributed Computing Clusters," *Proc. ACM SIGOPS 22nd Symp. Operating Systems Principles*, pp. 261-276, 2009.
[17] L. Mann, "Queue Culture: The Waiting Line as a Social System," *The Am. J. Sociology*, vol. 75, no. 3, pp. 340-354, 1969.
[18] R.C. Larson, "Perspectives on Queues: Social Justice and the Psychology Of Queueing," *Operations Research*, vol. 35, no. 6, pp. 895-905, 1987.
[19] B. Avi-Itzhak, E. Brosh, and H. Levy, "SQF: A Slowdown Queueing Fairness Measure," *Performance Evaluation*, vol. 64, no. 9, pp. 1121-1136, 2007.
[20] J. Ngubiri and M. van Vliet, "Characteristics of Fairness Metrics and Their Effect on Perceived Scheduler Effectiveness," *Int'l J. Computers & Applications*, vol. 32, no. 2, p. 188, 2010.
[21] S. Agarwal, R. Garg, M.S. Gupta, and J.E. Moreira, "Adaptive Incremental Checkpointing for Massively Parallel Systems," *Proc. 18th Ann. Int'l Conf. Supercomputing*, pp. 277-286, 2004.
[22] C.B. Lee, Y. Schwartzman, J. Hardy, and A. Snavely, "Are User Runtime Estimates Inherently Inaccurate?" *Proc. 10th Workshop Job Scheduling Strategies for Parallel Processing*, pp. 253-263, 2005.
[23] W. Tang, Z. Lan, N. Desai, and D. Buettner, "Fault-Aware, Utility-Based Job Scheduling on Blue Gene/P Systems," *Proc. IEEE Int'l Conf. Cluster Computing*, pp. 1-10, 2009.
[24] R. Susukita, H. Ando, M. Aoyagi, H. Honda, Y. Inadomi, K. Inoue, S. Ishizuki, Y. Kimura, H. Komatsu, M. Kurokawa, K.J. Murakami, H. Shibamura, S. Yamamura, and Y. Yu, "Performance Prediction of Large-Scale Parallel System and Application Using Macro-Level Simulation," *Proc. ACM/IEEE Conf. Supercomputing*, p. 20, 2008.
[25] J.R. Lange, K. Pedretti, P. Dinda, P.G. Bridges, C. Bae, P. Soltero, and A. Merritt, "Minimal-Overhead Virtualization of a Large Scale Supercomputer," *ACM SIGPLAN Notices*, vol. 46, no. 7, pp. 169-180, 2011.
[26] S. Krishnaswamy, S.W. Loke, and A. Zaslavsky, "Estimating Computation Times Of Data-Intensive Applications," *IEEE Distributed Systems Online*, vol. 5, no. 4, Apr. 2004.
[27] S.R. Alam, J.A. Kuehn, R.F. Barrett, J.M. Larkin, M.R. Fahey, R. Sankaran, and P.H. Worley, "Cray XT4: An Early Evaluation for Petascale Scientific Simulation," *Proc. ACM/ IEEE Conf. Supercomputing*, pp. 1-12, 2007.
[28] D. Zotkin and P.J. Keleher, "Job-Length Estimation and Performance in Backfilling Schedulers," *Proc. Eighth IEEE Int'l Symp. High Performance Distributed Computing*, pp. 236-243, 1999.
[29] D. Tsafrir and D.G. Feitelson, "The Dynamics Of Backfilling: Solving the Mystery of Why Increased Inaccuracy May Help," *Proc. IEEE Int'l Symp. Workload Characterization*, pp. 131-141, 2006.
[30] D.G. Feitelson, *Parallel Workloads Archive*, http://www.cs. huji. ac.il/labs/parallel/workload/, 2005.
[31] E. Yero and M. Henriques, "Contention-Sensitive Static Performance Prediction for Parallel Distributed Applications," *Performance Evaluation*, vol. 63, no. 4, pp. 265-277, 2006.



Yulai Yuan received the PhD degree in computer science from Tsinghua University in 2010. He is currently a postdoctor in the Department of Computer Science and Technology at Tsinghua University, China. His research interests include parallel and distributed systems, performance modeling, prediction and optimation.

Yongwei Wu received the PhD degree in applied mathematics from the Chinese Academy of Sciences in 2002. He is currently a full professor of computer science and technology at Tsinghua University of China. His research interests include parallel and distributed processing, and cloud storage. He has published more than 80 research publications and has received two Best Paper Awards. He is currently on the editorial board of the *International Journal of Networked and Distributed Computing* and *Communication of China Computer Federation*. He is a member of the IEEE.

Weimin Zheng received the BS and MS degrees from the Department of Automatics, Tsinghua University, in 1970 and 1982, respectively. He is a full professor of computer science and technology, Tsinghua University, China. He is currently the director of the Chinese Computer Society. His research interests include computer architecture, operating systems, storage networks, and distributed computing. He is a senior member of the IEEE.

Keqin Li is a SUNY distinguished professor of computer science in the State University of New York. He is also an Intellectual Ventures endowed visiting chair professor at the National Laboratory for Information Science and Technology, Tsinghua University, Beijing, China. His research interests are mainly in design and analysis of algorithms, parallel and distributed computing, and computer networking. He has published more than 280 research articles. He is currently or has served on the editorial board of *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *Journal of Parallel and Distributed Computing*, *International Journal of Parallel, Emergent and Distributed Systems*, *International Journal of High Performance Computing and Networking*, and *Optimization Letters*. He is a senior member of the IEEE.