PV-EASY: A Strict Fairness Guaranteed and Prediction Enabled Scheduler in Parallel Job Scheduling

Yulai Yuan[*], Guangwen Yang[†] , Yongwei Wu[†] ,Weimin Zheng[†]

Tsinghua National Laboratory for Information Science and Technology (TNList)

Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China

[*]yuan-yl05@mails.tsinghua.edu.cn

[†] {ygw, wuyw, zwm-dcs}@tsinghua.edu.cn

ABSTRACT

As the most widely used parallel job scheduling strategy in production schedulers, EASY has 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. For real workloads used in this work, our simulation shows that a blocked job can be delayed by later jobs for more than 90 hours. In addition, EASY cannot directly employ parallel job runtime prediction techniques, because this 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, *shadow load preemption* (SLP) and *venture backfilling* (VB), which are together integrated into EASY to construct a *preemptive venture EASY backfilling* (PV-EASY) strategy. Experimental results on three workloads of real HPC systems demonstrate that: First, PV-EASY guarantees strict fairness, in addition to avoiding reservation violation when employing job runtime prediction techniques in scheduling; Second, PV-EASY achieves the same performance as EASY, and outperforms prediction employed EASY; Third, the preemption in PV-EASY is not resource costly and simple enough to be implemented in all HPC systems where EASY works. These advantages make PV-EASY more attractive than EASY in parallel job scheduling, both from academic and industry perspectives.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management – *Scheduling.*

General Terms

Algorithms, Management, Performance

Keywords

Scheduling, preemptive, prediction, backfilling

1. INTRODUCTION

Parallel job scheduling is critical in large-scale high performance computing (HPC) systems, such as Clusters, Grids and Clouds, since different scheduling policies can result in different user experiences and resource utilization. Fairness and performance are two eternal topics in parallel job scheduling. Previous works either focus on providing fair scheduling [24][28][29] or improving performance [15][19][20][32]. However, fairness and performance should be considered in concert. First Come First Serve (FCFS) mainly guarantees fairness, while Short Job First (SJF) [6] mainly targets performance. This explains why they are rarely used alone in practice. EASY backfilling [1] leverages fairness and performance in a simple and efficient manner. It allows later jobs to backfill in idle processors that cannot satisfy the request of the first blocked job, provided these backfilled jobs would not delay the expected start time (reservation) of the first blocked job in the queue. EASY can guarantee "relaxed" fairness to jobs through reservation, as well as achieve good performance by allowing shorter jobs to be shifted forward. Because of a better balance between fairness and performance, EASY is the most widely used job scheduling strategy in high performance parallel computing systems. It has been adopted by a number of major production schedulers, including IBM's LoadLever, Cluster Resources' Moab and Maui, Platform's LSF, Altair's OpenPBS and PBS-Pro, and Sun's GridEngine[9].

In order to obtain better fairness while achieving attractive performance, many approaches have been proposed to improve EASY in the past. While they provide better fairness in scheduling, most of them suffer from the same problem: they mix up fairness and performance by trying to measure the fairness of schedulers with performance metrics, such as *Slowdown Queuing Fairness* (SQF) [28], *fair-slowdown* [10] and *fair start time* [29]. We claim that fairness has to be guaranteed independently from performance metrics. Unfortunately, our results show that this does not happen yet, and on production systems, EASY still suffers from unfairness. Based on the definition of *strict fairness* (no job is delayed by any jobs of lower priority), a blocked job can be delayed by later jobs for up to 90.7 hours in the simulation with real workloads. Moreover, to the best of our knowledge, no existing studies can guarantee strict fairness. 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 in real production schedulers of EASY, though EASY is built on highly inaccurate user estimates of jobs' runtime[3][30][16]. This is due to two reasons. First, there is a misconception that inaccurate user estimates of jobs' runtime can improve performance [3][37][38]. Second, replacing user estimates directly with system-generated prediction in EASY would lead to *reservation violation*, which means a blocked job in the queue cannot start at its reservation time. The delay from reservation time can be as large as 32.9 hours in the simulation with real workloads. These two reasons cause lots of prediction techniques [33][34][35] to be put on the shelf, even if they were reported to have better prediction accuracy than user estimates. In order to solve the dilemma of prediction techniques, 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) which integrates *shadow load preemption* (SLP) and *venture backfilling* (VB) into EASY. Our paper makes two main contributions:

(1) Propose *shadow load preemption* (SLP) 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 running load of a system is classified into *sunny load* and *shadow load* (Section 4.1), and SLP (Section 4.2) can preempt the processors occupied by *shadow load* to start blocked jobs according to the definition of strict fairness. Preemption in SLP can guarantee strict fairness in scheduling and avoid reservation violation. Additionally, as observed from the experiments performed on three workloads collected from real HPC systems, the Mean Bounded Slowdown (MBS) and Mean Weighted Bounded Slowdown (MWBS) of blocked jobs in PV-EASY are mostly lower than that in EASY.

(2)Propose *venture backfilling* (VB) to promote performance, which makes PV-EASY more attractive than EASY in terms of performance.

VB (Section 4.3) can counteract the negative effects of the kill/restart preemption mode in SLP and therefore promote performance. After backfill decisions made based on runtime prediction, high priority jobs may still be venturesomely backfilled if there still exist idle processors, without considering job runtime and reservation. Benefiting from VB, PV-EASY with the simplest Last Model predictor can achieve the same MBS and MWBS as EASY. Moreover, PV-EASY better employs prediction techniques than EASY on performance. If both PV-EASY and EASY adopt the same existing job runtime prediction techniques, PV-EASY achieves better MBS and MWBS than EASY.

We validate the effectiveness of our proposed solution 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 where EASY works, which makes PV-EASY attractive in real production environments. PV-EASY employs the simple kill/restart preemption mode and Last Model predictor. Unlike suspend/resume and checkpoint/restart preemption modes, or predictors which use profiling, machine learning, etc, PV-EASY does not need the support of any complex system features and thus can be easily implemented in HPC systems and production schedulers. Additionally, even by employing kill/restart preemption, the wasted resources in PV-EASY are controlled at a low level (2.48%~5.66%), and therefore do not affect system throughput when the system load is close to 80% (the common upper load in production HPC systems).

The rest of this paper is organized as follows. Section 2 provides the background and related work of parallel job scheduling, including EASY and its variants. In Section 3 we discuss the motivations 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 section 6 and 7 we discuss and conclude.

2. BACKGROUND AND RELATED WORK

In the past, besides the most natural and simplest First Come First Serve (FCFS) strategy, many kinds of order-based parallel job scheduling strategies[4][5] have been proposed, such as Shortest Job First (SJF), Smallest Job First[6], and Smallest Cumulative Demand First[6][7], etc. But all of them share similar inherent problems: 1) Due to different sizes of jobs, if the first waiting job is blocked, some processors would be idle and a "hole" would appear in the system 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].

2.1 Backfilling and Variants

Backfilling allows later jobs to fill in the hole created by the first blocked job in the waiting queue as long as they do not delay the expected start time (reservation) of blocked jobs in the queue. In this way, system utilization is improved by filling the holes, and starvation is avoided by assigning reservations to the blocked jobs. The implementation of backfilling relies on user estimates of jobs' runtime, based on which the scheduler can determine reservations and which job can be backfilled without violating the reservations of its predecessors. Generally, the possible runtime of jobs is estimated by users. If a job runs beyond its estimated time, it will be killed and never restart.

There exist two major versions of backfilling, 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 they would not delay the expected start time of the first job in the queue. Aggressive backfilling has been reported to have better performance than conservative backfilling [2]. Additionally, EASY backfilling[1], the original aggressive backfilling algorithm in practice, is the most widely used scheduling strategy in parallel job scheduling, and has been adopted by most major production schedulers, including IBM's LoadLever, Cluster Resources' Moab and Maui, Platform's LSF, Altair's OpenPBS and PBS-Pro, and Sun's GridEngine[9].

To enhance the performance of EASY from different perspectives, many variants of EASY have been proposed, most of which can be classified by their changes in the following two aspects:

- **Reservation Calculation**: Generally, a loosened reservation of a blocked job in EASY would result in better performance, such as small average slowdown, because a longer reservation allows more jobs to be backfilled. Ward et al. [12] explored the possibility of using a relaxed backfill strategy, where jobs with lower priorities could be backfilled as long as they would not delay the job that holds the highest priority too much. Dynamic backfilling [14] allows the scheduler to overrule a previous reservation under the condition that a slight delay will considerably get utilization improvement in return. Talby et al. [15] presented a slack based backfilling, which assigns each waiting job a slack to determine how long it has to wait before running. Srinivasan et al. [11] designed a selective backfilling where a reservation is selectively made when a job's expected slowdown exceeds a certain threshold. These variants could improve performance by enabling more small jobs to be backfilled than EASY, while sacrificing the blocked jobs.

• **Backfill Selection:** Some existing works modified the selection sequence of backfilled jobs in EASY. Chiang et al. [10] reported that prioritizing jobs by decreasing runtime estimates or increasing expected slowdown can improve the performance of scheduling. Shmueli et al. [13] proposed a look-ahead optimizing scheduler to generate the local optimal backfill selection by using dynamic programming. Some other existing works employ system-generated runtime prediction to replace the request time estimated by user. Tsafrir et al.[16] integrated system-generated prediction into EASY, only keeping user estimates as the kill time of jobs. Cuim et al. [18] proposed a resource usage aware backfilling, by using *LessConsume* resource selection policy to decide which job has to be executed and how jobs have to be backfilled.

Besides the above two major backfilling variants of EASY, Thebe et al. [17] examined the concept of giving every job a short trial run to allow immediate detection of job failures and benefit short jobs that can finish during the trial run. Based on market-inspired utility functions, utility-based scheduling [32][36] strategies have also been proposed recently and claimed performance improvements ranging from 4% to 20% compared with backfilling[36]. But utility-based scheduling is rarely deployed in practice [8], due to its potential harm of fairness.

2.2 Scheduling Metrics

Performance is an important factor that attracts users of HPC systems. So, many existing works (e.g.,[15][19][20][32]) paid a lot of attention to promoting 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).Shorter turnaround time and smaller slowdown of jobs would stimulate users to submit more jobs [21], and higher utility and more energy saving are appreciated by the owners of HPC systems.

However, it is also known that users in queue systems are sensitive to fairness [25][26], and they might consider fairness even more important than productivity in HPC systems [21]. Due to this reason, various fairness metrics have been proposed to evaluate the fairness of scheduling, such as *Resource Allocation Queuing Fairness Metric* (RAQFM) [22][23], *Slowdown Queuing Fairness* (SQF) [28] and *fair-slowdown* metrics [11]. In [27], each job is assigned a "fair start time" and fairness is measured by judging whether a job starts after its "fair start time". J. Ngubiri et al. [29] examined the characteristics of three fairness evaluation approaches in parallel job scheduling. All these works about fairness mainly focus on fairness measurement and promoting average fairness according to their fairness metrics, rather than designing a scheme to guarantee a certain level of fairness.

2.3 Prediction in EASY Backfilling

EASY backfilling is built on jobs' runtime estimates given by users, which 1) determine how long jobs can execute before they are killed, 2) compute the reservation of blocked jobs, and 3) decide which waiting job can be backfilled. In EASY, users are required to estimate "request time" for their jobs [1] based on the hypothesis that users would be motivated to provide as accurate runtime estimates as they can, due to a tradeoff that short (long) request time of a job would make it have more (less) chance to be backfilled and therefore achieve better (worse) performance, while also enlarging (reducing) its risk of being killed before it successfully finish.

Unfortunately, the above hypothesis often fails in practice. User estimates of jobs' runtime are reported to be highly inaccurate [3][16][30]. Even worse, most users are unable or reluctant to provide better job runtime estimates when they submit their jobs to HPC systems [31]. These facts stimulate researchers to propose more accurate system-generated runtime prediction techniques. Notice that, in the rest of this paper, we use "estimate" to denote the request time of job forecasted by users, and use "prediction" to indicate the result of system-generated runtime prediction. Several existing studies propose various runtime prediction methods by learning from historical data [33][34][35], and they are all reported to have better accuracy than user estimates.

However, prediction techniques are rarely integrated into EASY in real production systems, due to two misconceptions. First, inaccurate runtime estimates of jobs can improve performance [3]. Due to this misconception, some previous works even suggested using *randomized* [37] or *doubled* [38] runtime estimates in EASY. Second, users would not tolerate their jobs being killed just because predictions were too short. In [16], the authors well clarified how the first misconception failed in three aspects: 1) perfect estimates improve performance more than doubling original user estimates; 2) the performance gained from doubled estimate is derived from a "heel and toe" dynamic [39] which allows more short jobs to be backfilled, but also gradually pushes away the start time of the first blocked job in the queue; 3) a reliable job runtime prediction method could facilitate advance reservation in grid resource allocation and co-allocation. The second misconception is handled by still using user estimates as kill-time while employing prediction in scheduling [16]. With these misconceptions removed, prediction is integrated into EASY [16].

3. MOTIVATION

The wide use of EASY in production schedulers and HPC systems has already proved its practicability. But unfortunately, EASY and its variants still suffer from some serious problems that were partly discussed in the previous section and will be further 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 fairness. However, as mentioned in Section 2.2, HPC users might be more sensitive to fairness than performance [21]. We believe that fairness and performance are both important to attract users in parallel job scheduling.

Current studies mainly focus on proposing various metrics to measure the fairness of schedulers[22][23][11][28], and try to deliver as best average fairness as possible[27][29]. In addition, many existing works mix up fairness and performance by measuring the fairness of schedulers through performance metrics [11][28][29]. In fact, fairness is independent from performance and unsuitable to be measured by performance metrics. For example, people might keep going to the same restaurant as long as its service is quick most of the time, even if occasionally the service time is unstable and lengthened. But, suppose once you were seriously unfairly served in that restaurant, for instance, no waiter responded to your order and you were served after 10 people who came later than you, would you go there again? That's the point which distinguishes fairness from performance: 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 in other words, they need services with guaranteed fairness.

A question is naturally raised, what is the guaranteed fairness in parallel job scheduling? Fairness judgment depends on priority factors. Submission time is regarded as a natural priority factor of jobs in most parallel computing systems. FCFS can be regarded as an absolutely fair scheduling strategy, which holds the view that users are sensitive to service sequences, and jumping the queue is unacceptable. It is clear that this kind of fairness definition is proper for systems where every job needs to fully occupy all resources and the resource availability has only two statuses, idle or full.

Does the view of fairness in FCFS still work in parallel job scheduling? Our answer is no. Because in a parallel system with N resources, most jobs cannot occupy all the system resources and the resource availability has N+1 statuses, from completely idle (0/N), 1/N, 2/N, …, to completely full(N/N). 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 the user of A would not raise any complains. Thus we define the notion of "**strict fairness**" in parallel job scheduling: **If no job is delayed by any jobs with lower priority, this scheduling sequence can be viewed as strict fair.**

EASY backfilling maintains a view which is similar to strict fairness, but it is only a "relaxed fairness" that a job can be backfilled if it would not delay the "expected" start time (reservation) of the first blocked job in the waiting queue. But actually these backfilled jobs often cause the blocked job to be unfairly treated according to the definition of strict fairness. Because of the "heel and toe" dynamic [39], the start time of the blocked job is often gradually pushed away by its later jobs, which therefore results in unfair experiences of the blocked jobs. We have observed significant unfairness in EASY from the experiment performed on three real production workloads (Section 5.1.3). Results are summarized in Table 1. It is clear that around 20% of the blocked jobs in each workload suffered from unfairness (delayed by later jobs, denoted as *delay jobs*). On average, they were delayed 80, 169, and 133 minutes in CTC, SDSC-BLUE and SDSC-DS respectively by later jobs. Moreover, the maximum delay caused by later jobs can be as long as 90 hours or more (in SDSC-BLUE).

Table 1. Start time delay of blocked jobs in EASY. (# denotes the number)

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

In order to overcome the misconception of existing works about fairness and help the blocked jobs suffering from unfairness in EASY, our objective is to design 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 first motivation of designing a new scheduler can be specified as follows:

Objective 1: Our scheduler should guarantee strict fairness to the jobs with attractive performance.

3.2 Employ Prediction

In parallel job scheduling, EASY backfilling also achieves great success based on user estimates of jobs' runtime, even if these estimates are highly inaccurate [3][16][30].

Performance modeling and runtime prediction technologies (e.g., [33][34][35]) have been reported to have better accuracy than user estimates. However, these advanced prediction techniques are rarely applied in real production schedulers, because of the misconceptions stated in Section 2.3. D. Tsafrir[16] well clarified how these misconceptions failed and proposed an EASY variant that adopts prediction by separating the role of kill-time from prediction. It seems that the obstacles of prediction techniques in parallel job scheduling are eliminated in this way. But unfortunately, we found that directly replacing user estimates with prediction in EASY as it is proposed in [16] can lead to lots of problems. These problems, as analyzed below and demonstrated in Figure 1, are derived from inherent and unavoidable properties of prediction, *underestimate* (shorter than jobs' actual runtime) and *overestimate* (longer than jobs' actual runtime and shorter than user estimates). Notice that user estimates has been proven to be suitable to play the role of kill-time [16] and we hold the same view, so prediction that exceeds user estimates is meaningless since a job will be killed when it reaches its estimated time. Thus *overestimate* in this analysis is defined as the interval between runtime and user estimates.

• Overestimate of running jobs' runtime might lead to resource waste. Because the prediction is shorter than the user estimates, the reservation time is shorten and fewer jobs can be backfilled, thus more holes are left and more computational resource is wasted, compared with EASY which adopts estimates of users. On the contrary, overestimate of waiting jobs can result in better performance than EASY, 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 Figure 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,

Table 2. **Reservation Violation caused by Last Model in EASY. (Reservation Violation is denoted as RV; Delay Time from Reservation is denoted as DTR; Slowdown Increment caused by Reservation Violation is denoted as SI)**

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

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). Figure 1(b) demonstrates one possible case of this issue. If job 1 is underestimated (Figure 1 (b)), job 3 is unable to be backfilled as in EASY (Figure 1(a)), and six squares of CPUTime resource (number of processors X Time) is wasted before job 2 starts (Figure 1(b)), while in EASY (Figure 1(a)) only four squares of CPUTime is wasted before job 2 starts. Furthermore, job 2 actually starts at T=2 and violates its reservation T=1. 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 the backfilling of a job whose execution time is actually longer than the reservation of the first blocked job, and thus push 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 demonstrated in Figure 1(c), job 5 is underestimated to be able to finish in four time units (it actually takes six time units) and would not delay the start time of the blocked job 2, so job 5 is backfilled at time T=0. But actually job 5 cannot finish before the reservation of job 2 at time T=5 and job 5 keeps running until time T=6. Therefore job 2's reservation guaranteed by EASY is violated.

To prove the existence of reservation violation caused by directly replacing user estimates with prediction in EASY, we implemented a prediction-based EASY backfilling as [16] by replacing user estimates with predictor when calculating reservations and choosing backfilled jobs, and performed a simulation on three real workloads (more detail in Section 5.1.3). The "benign" situation that illustrated in Figure 1(b) was filtered out, and we only focused on the most serious situation: reservation violation. The predictor in this experiment is 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.

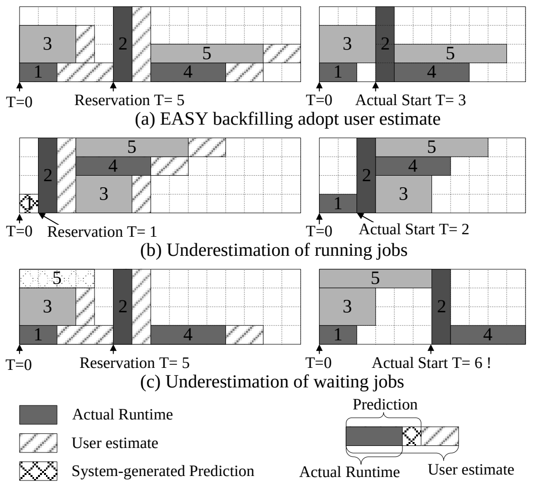

Figure 1. Example of directly replacing user estimates with prediction in EASY. Left side of each sub-figure is the scheduling decision and right side is the actual job running status corresponding to each scheduling decision.

Simulation results are shown in Table 2. In each workload, tens or hundreds (92~407) of jobs 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. Additionally, the bounded slowdown (defined in Section 5.1.2) of these victims dramatically increased, the average slowdown increment (SI) is 4.78 in CTC, 36.13 in SDSC-BLUE, and 40.78 in SDSC-DS. The maximum SI was even up to 1581 in SDSC-DS. If a job's reservation is violated, its owner would have the illusion that this job is starving. Furthermore, as shown in the column of Mean Parallelism, Table 2, 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., [16][38]) did, mainly due to the existence of reservation violation caused by unavoidable prediction errors. Even worse, most EASY variants also suffer from this problem, because they all inherit the same framework of reservation calculation and backfill selection as EASY. So how can current studies of performance modeling and runtime prediction, from serial job runtime prediction, to more complicated workflow runtime prediction [40] and parallel job runtime prediction [33][35][42], be able to facilitate parallel job scheduling strategies, especially for the most widely used EASY in real production? In this paper, we answer this question from a different perspective: design a scheduler that can limit the bad consequences caused by inaccurate predictions. Based on this motivation, the second objective of our ideal scheduler is:

Objective 2: Our scheduler should employ prediction without causing reservation violation.

4. PREEMPTIVE VENTURE EASY BACKFILLING

In this section, we first introduce our view about the classifications 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* (VB) 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 (PV-EASY) parallel job scheduling strategy. When a new job is submitted, or a running job is finished, PV-EASY will firstly try to schedule jobs according to their priorities, from high to low, 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 Classifications of Running Load in EASY

The load of a parallel computing system consists of a set of running jobs and thus it is called "running load" in this paper. In our view, if a running job's priority is higher than all the jobs in the waiting queue, it is likely running under the sunshine that no one can question its rights of holding the resources. So the running load consists 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 (called regular job) if it holds the highest priority in the queue, or otherwise 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 of their lifetime. However, a backfilled job could transit from *shadow load* to *sunny load* during its lifetime. Suppose that jobs 1,2,3 are sequentially submitted, job 1 is blocked and job 2 is backfilled at time t1, then a previous running job is finished and job 1 is started at time t2. In this situation, job 2 belongs to the *shadow load* from t1 to t2, and after t2, job 2 transits to the *sunny load*.

4.2 Shadow Load Preemption

The objectives of this work include guaranteeing strict fairness, as well as employing job runtime prediction technologies without causing reservation violation. In order to do so, a novel *shadow load preemption* (SLP) strategy is proposed based on the framework of EASY.

S*hadow load* is the root cause of unfairness in EASY. Recall from Section 3.1 that according to the definition of strict fairness, unfairness always happens in the blocked jobs within the waiting queue of EASY. Because of inaccurate estimates of the blocked jobs' reservation and the backfilled jobs' runtime, the start time of blocked jobs could be delayed by the backfilled jobs submitted later. It is clear that the jobs that cause unfairness all belong to the *shadow load*.

Also, the *shadow load* could lead to reservation violation and thus hinders the employment of prediction technologies in EASY. As analyzed in Section 3.2, underestimate of a waiting jobs' runtime would mislead EASY to backfill jobs with long runtime (exceeds the reservation of blocked jobs) and therefore cause reservation violation. It is clear that these backfilled jobs with long runtime belong to the *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. But 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*** **(SLP):**

> When the idle resources of a system are not sufficient for the highest priority job in the waiting queue, the system can preempt the resources occupied by the jobs in the *shadow load* 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 killed and returned to the waiting queue.

By integrating SLP into EASY, strict fairness can be guaranteed and prediction will not cause reservation violation anymore. In SLP, because of preemption, the *shadow load* is invisible to the blocked jobs, and thus the backfilled jobs of the *shadow load* would no longer delay the start of the blocked jobs. 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.

We select preempted jobs in the *shadow load* according to their priorities, from low to high, by considering 1) strict fairness(lower priority jobs should not delay higher priority jobs), and 2) the possible role switching of a job from the *shadow load* to the *sunny load*. By preempting from lower priority jobs to higher ones, high priority jobs can be left in the *shadow load* as long as possible, to maximize their opportunity of transition to the *sunny load*, and therefore minimize the number of jobs that suffer from kill and save computational resources.

4.3 Venture Backfilling

Not all existing HPC systems and applications can support suspend/ resume or checkpoint/restart of jobs. In order to make our study be able to work in existing systems and for all kinds of jobs, the simplest and universally supported kill/restart preemption mode is adopted in SLP.

The kill/restart mode in SLP is resource costly and may lead to performance degradation. If a backfilled job in the *shadow load* is preempted unluckily, all the work it has already done will be totally lost and it has to restart from its origin next time. 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.

We propose a *venture backfilling* (VB) 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. The process of VB is stated as follows:

1) Computes the reservation time of the first blocked job in the waiting queue. Only the *sunny load* is used to compute the reservation based on runtime prediction, while in EASY the whole load (including the *sunny load* and *shadow load*) is used to compute reservation based on user estimates.

2) Determines the possible runtime of waiting jobs by employing system-generated prediction. While EASY adopts user estimates.

3) Selects 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, selects waiting jobs to be backfilled according to their priorities, from high to low, no matter whether the prediction indicates they could successfully complete before possible preemption or not.

Step 3) aims to reduce the occurrence of preemption, and the purpose of Step 4) is to make full use of the computational resources.

Step 4) is a novel but adventurous approach which is different from existing EASY variants. It seems that the jobs backfilled in Step 4) have very little chance to survive from preemption. However this is not true, because prediction is always inaccurate and they may have opportunities, and recall from Section 4.2, preemption in SLP occurs from low priority to high priority, so the backfilled jobs with high priority in Step 4) would still have opportunities to successfully transit to the *sunny load* and complete.

5. EVALUATION

In this section, we first introduce the experimental design that is used to evaluate our PV-EASY, including the simulator, metrics and workloads. Then we present and analyze the experimental results.

5.1 Experimental Design

5.1.1 Simulator

We have constructed an event-based simulator to mimic different scheduling strategies in generic parallel computing clusters. It is driven by the workloads collected from real HPC systems (Section 5.1.3 for more details). Events in this simulator are job submit, start, finish, and kill. Upon submit and finish, the scheduler is informed to schedule the jobs in the waiting queue, and generate job start or kill events by scheduling decisions. Once a job is scheduled to start, a finish event related to this job is created based on the real runtime of this job from the workload, but this runtime is invisible to the scheduler. Our simulator considers the processor request of each job, as most existing works [1][3][8] do, because computational resources are usually the scarcest resources in HPC systems.

5.1.2 Metrics

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

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

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 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 by each job as weight, we propose the metric MWBS be defined as follows:

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

Where the $Parallelism_j$ is the number of processors occupied by job j, and N is the total job counts.

We do not employ turnaround time as an independent metric to evaluate schedulers in this paper, because they are already implicitly included in MBS and MWBS.

In order to measure system-aware performance, we adopt system load as the metric. Load is computed as the CPUTime consumed by all jobs divided by the total CPUTime available in the system (System Processor capability X log time), which demonstrates the utilization rate of an HPC system under a certain scheduling strategy.

Unlike existing works which prefer to measure the fairness, we do not employ any fairness metrics in this paper. We aim at providing strict fairness to all the jobs, and with the help of preemption, our method does achieve this objective and can guarantee that no job is delayed by any jobs with lower priorities than it.

5.1.3 Workloads

The workload traces used to evaluate our PV-EASY are collected from real HPC systems. They are composed of job entries that record submission and execution information of jobs. Typically the following data fields of each job in the workload are used to drive our simulator and scheduling strategies. In our experiments, the values of these data fields are faithful to the original workload.

- **Job ID:** is determined by the sequence of submission
- **Job Submission Time:** the time that a job is submitted
- **Job Parallelism:** the number of processors occupied by a job
- **Job Request Time:** the possible job runtime estimated by user
- **Job Runtime:** actual runtime of a job

From the collection of Parallel Workload Archive (PWA) [41], we selected three workload traces (CTC, SDSC-BLUE and SDSC-DS) to evaluate our PV-EASY and other scheduling strategies. These workload traces are all named by the names of their HPC systems and their affiliations. CTC is a 512-processor IBM SP2 machine located at the Cornell Theory Center, but only 430 processors are dedicated to running batch jobs. Therefore in our experiments, the computational capacity of CTC is set to 430 processors. SDSC-BLUE and SDSC-DS are all located at the San Diego Supercomputer Center (SDSC). SDSC-BLUE is a 144-node (8 processors per node) IBM SP machine. SDSC-DS is a 184-node IBM eServer pSeries 655/690 machine, and totally covers 1664 processors (DS is short for DataStar).

An overview of these three workloads is given in Table 3. They are all collected during long production periods (at least one year)

Table 3. An overview of the workloads. Load in this section is denoted as the percentage of total CPUTime of running jobs in system capability (#Processors X (Submission Time of the last job - Submission Time of the first job)).

Workload	Duration	#Processors	#Job	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

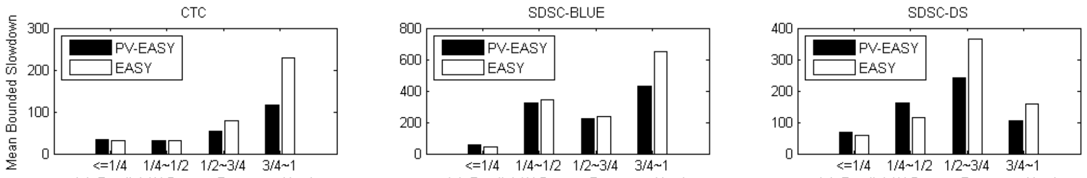

Figure 2. Mean Bounded Slowdown (MBS) comparison of Blocked Jobs in PV-EASY and EASY.

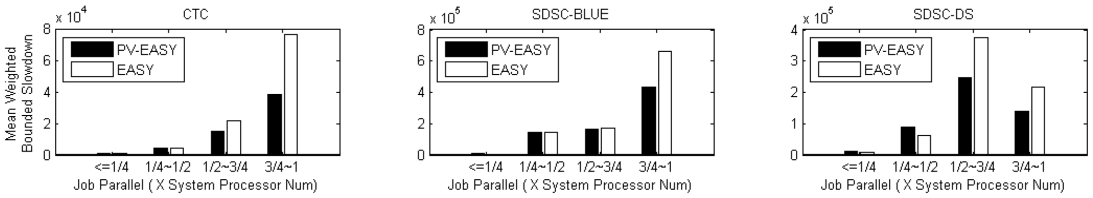

Figure 3. Mean Weighted Bounded Slowdown (MWBS) comparison of Blocked Jobs in PV-EASY and EASY.

and contain a large amount of job entries, with a regular load between 60% and 80%. The jobs in these workloads use tens of processors on average, and their mean runtime always exceeds 1 hour.

Other traces of PWA were not selected, because 1) some systems of the workloads are too small, for example, SDSC-SP2 has only 128 processors; 2) some workloads contain too many jobs that failed to record job parallelism or runtime, such as SHARCNET and LLNL-Altas; 3) some workloads only contain serial jobs, like LPC EGEE; and 4) some workloads are used for system testing rather than production, like LLNL-uBGL.

5.2 Experiments and Results

Motivated by guaranteeing strict fairness with well performance and employing prediction without causing reservation violation in parallel job scheduling, we propose PV-EASY in this paper. In this sub-section, we demonstrate and analyze how PV-EASY achieves above two attractive objectives based on experimental results. Besides, PV-EASY employs the simple but resource costly kill/restart preemption mode, so the problem of resource waste is also analyzed.

We only use EASY as a comparison target in our experiments. Because the default setting of most parallel schedulers remains plain EASY [9], and furthermore, it is statistically reported that 90%~95% of the parallel scheduler installations do not change this default configuration [43].

In our experiments, the priority factor in all parallel scheduling strategies is the submission time of jobs, as generally used in most production environments. If not specified, the job runtime predictor used in each of the scheduling strategies is Last Model, which predicts the *lifetime accuracy* (runtime / Request Time) of a job to be the same as the last job of the same user, and then generates runtime prediction with this lifetime accuracy and Request Time. If no such user exists, Request Time (user estimates) will be used as prediction result instead.

5.2.1 Benefits of Maintaining Strict Fairness

Benefiting from *shadow load preemption* (SLP), strict fairness is guaranteed in our PV-EASY. Therefore, there is no need to measure PV-EASY with fairness metrics (Section 5.1.2). Instead, advantages of maintaining strict fairness in PV-EASY can still be demonstrated from another perspective: how blocked jobs benefit from guaranteed strict fairness.

Based on the definition of strict fairness (Section 3.1), the blocked jobs often suffer from unfairness due to the “heel and toe” dynamic [39]. Because strict fairness is guaranteed in PV-EASY, the performance of these blocked jobs should be theoretically promoted. In this part of the experiments, we simulate the scheduling of PV-EASY and EASY on 3 workload traces, and compare the performance of the blocked jobs between these two scheduling strategies. In order to analyze the impacts of guaranteeing strict fairness on different sizes of jobs, the blocked jobs are grouped according to their parallelism. As shown in Figure 2, the MBS of big blocked jobs (job parallelism larger than 1/4 of the system processor numbers) in PV-EASY are mostly smaller than that those in EASY. As shown in Figure 3, in terms of the MWBS of the blocked jobs, big blocked jobs are also better treated in PV-EASY than in EASY. Notice that due to their large parallelism, big jobs are more likely to suffer from blocking and serious delay than small jobs in EASY. The experimental results clearly indicate that the performance of big (large parallelism) blocked jobs are promoted without the delay of later jobs in PV-EASY.

Small blocked jobs (job parallelism smaller than 1/4 of system processor numbers), as shown in the three workloads of Figure 2 and Figure 3, do not receive better treatment in PV-EASY com-

Table 4. Backfilled and Blocked Jobs in PV-EASY and EASY

Workload / Job Counts (#)		CTC	SDSC-BLUE	SDSC-DS
# Total Job		77222	223407	85003
# $\text{Job}_{\text{Backfilled}}$	EASY	38726	166014	51370
	PV-EASY	29061	148937	38683
# $\text{Job}_{\text{Blocked}}$	EASY	3833	10409	4147
	PV-EASY	6403	19262	8736

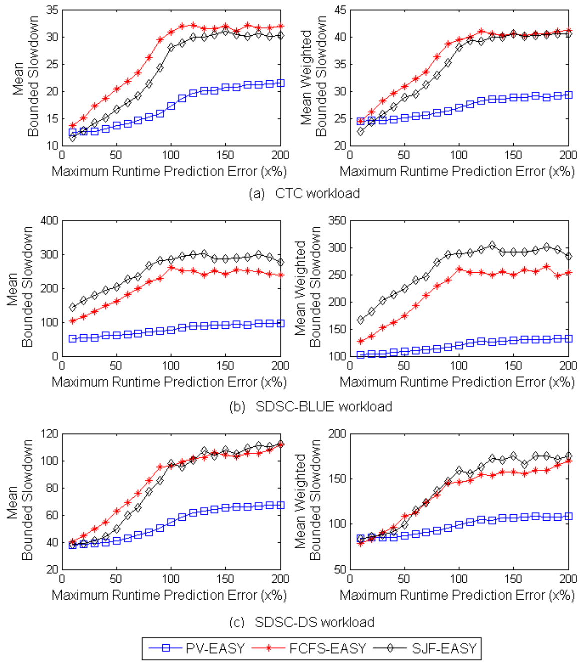

Figure 4. Performance comparison among "virtual" predictor integrated PV-EASY, FCFS-EASY and SJF-EASY, with different maximum runtime prediction errors.

pared with EASY. The reason is that, in order to guarantee strict fairness, big blocked jobs can preemptively start in PV-EASY. So it is harder for small jobs to successfully finish when they are backfilled, and therefore leads to more small jobs being killed. These killed jobs are returned to the waiting queue, re-backfilled, or blocked and finally regularly started after previous jobs release the processors. This process lengthens their turnaround time and bounded slowdown. Table 4 gives a statistic of the number of backfilled jobs and blocked jobs in PV-EASY and EASY. The common characteristic of these three workloads is that the number of backfilled jobs in PV-EASY is smaller than that in EASY, while the number of blocked jobs in PV-EASY is larger than that in EASY. These data indicate that compared with EASY, fewer jobs are finished during their backfill period and more jobs are regularly started in PV-EASY.

This performance degradation of small blocked jobs in PV-EASY can be viewed as 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.

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 (In our experiments, no reservation violation occurs in PV-EASY). Moreover, PV-EASY provides better supports to prediction than EASY. By employing the same prediction technique, PV-EASY achieves better performance than EASY.

In this part of the experiments, we integrate prediction into different scheduling strategies and compare their performances with different job runtime prediction accuracies. To present a fair comparison, besides commonly used EASY backfilling which employs FCFS strategy to choose backfilled jobs (denoted as FCFS-EASY in the rest of this sub-section), we also introduce a SJF-EASY which selects backfilled jobs according to the order of Short Job First (SJF) into the comparison, because PV-EASY selects backfilled jobs according to their runtime prediction (Section 4.3, step 3)), using Short Prediction Job First. Notice that we do not adopt any existing prediction technologies (e.g. Last Model) in this experiment. Instead, a "virtual" predictor is used to generate 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). This "virtual" predictor replaced the Last Model in PV-EASY, and also replaced user estimates (job Request Time) in FCFS-EASY and SJF-EASY in this part of experiments. For every scheduling strategy implemented on every trace with every x value, we repeated the simulation ten times, and then report the mean results.

As shown in Figure 4, PV-EASY achieves much better performance than FCFS-EASY and SJF-EASY in the three workloads. PV-EASY outperforms FCFS-EASY and SJF-EASY when runtime prediction error is bounded within maximum 10% (mean absolute prediction error is around 5%) on MBS and MWBS. Considering that existing parallel job runtime prediction techniques (e.g., [33][35][42]) have been reported to achieve mean absolute prediction error of more than 20% (corresponding to the maximum runtime prediction error 40% in Figure 4), we believe that PV-EASY can much better support prediction techniques in parallel job scheduling than EASY in the long term, until the time that prediction techniques could successfully limit the mean absolute runtime prediction error within less than 5% in real production applications and environments.

5.2.3 Performance Comparison with EASY

Fairness and performance are both attractive to users, but they are always a tradeoff in parallel job scheduling and overemphasizing any factors is unacceptable in reality. A successful scheduling strategy must well balance these two factors. Thus, recall from Section 3.1, our objective is not only guaranteeing strict fairness, but also providing attractive performance.

In SLP, kill/restart preemption mode would cause computational resource waste and result in performance degradation. On the other hand, because of inaccurate prediction, computational resources might still be left idle in SLP. So we proposed *venture backfilling* in PV-EASY to solve these performance problems. Figure 5 and Figure 6 show the comparison between PV-EASY and EASY on two performance metrics, MBS and MWBS, re-

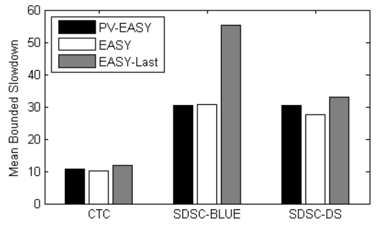

Figure 5. Mean Bounded Slowdown Comparison among PV-EASY, EASY and EASY-Last

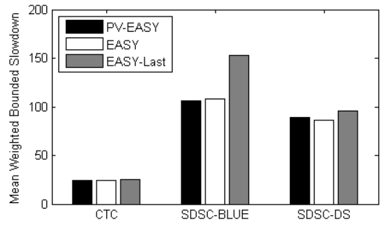

Figure 6. Mean Weighted Bounded Slowdown Comparison among PV-EASY, EASY and EASY-Last

spectively. 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 our 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 Figure 6, EASY-Last performs worst in three workloads, and we are therefore convinced that *venture backfilling* does successfully promote the performance as we expected.

These results indicate that in addition to guaranteeing strict fairness, PV-EASY with the simplest prediction technique (Last Model can be integrated into any system without any additional modification on that system) can achieve as attractive performance as EASY in real systems and workloads. Thus we can 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 (supported by all systems) but resource costly (the work that a job has already done can be totally lost when this job is preempted). Resource waste is an important issue that has already drawn the attention of industry. We analyze the total load and wasted load of the three workloads in PV-EASY, and the results are listed in Table 5. Notice that the definition of load in Table 5 is a little bit different from that in Table 3, because the system capability here is defined as System processor Number X (the time that all jobs finish – Submission Time of the first job).

It is clear that even by employing kill/restart preemption, the wasted load of PV-EASY is relatively small for three workloads (2.48% to 5.66%) and it does not worsen the system throughput. Each workload (the load varies from 63.02% to 76.21%, Table 3) finishes within the same time in PV-EASY and EASY. This result can be explained as follows.

Table 5. Load of PV-EASY in 3 workloads.

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

Table 6. 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

First, as the statistical results of preempted jobs in PV-EASY shown in Table 6, maximum 13.17% jobs (in CTC) suffered from preemption (nearly 1 out of 8 jobs), and in SDSC-BLUE, this rate is as low as 7.77%. 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 Run Time Waste (RTW). MTK counts the mean occurrences of killing among preempted jobs, and RTW is defined in formula (3):

$$RTW = \frac{Time_{sum} - Runtime}{Runtime} \quad (3)$$

Where $Time_{sum}$ is the accumulative runtime of a job, including its actual Runtime and the runtime it used before preemption happens. The results of MKT and RTW of PV-EASY are shown in Table 6. On average, preempted jobs in the three workloads were killed less than twice (1.72, 1.46 and 1.62 in CTC, SDSC-BLUE and SDSC-DS, respectively). Besides, these preempted jobs spent only 40%~50% additional time than their actual runtime. Based on the results of MKT and Mean RTW, we conclude that kill/restart mode does not significantly impact these preempted jobs.

6. DISCUSSION

EASY is widely applied and has achieved great success in HPC systems and production schedulers, not only because it can balance fairness and performance, but also due to its simple implementation. Based on EASY, lots of variants have been proposed. However, few of these variants really work in productions. Why? Because a production scheduler must be universally applicable to most HPC systems and EASY is the one that successfully achieves this.

In order to truly facilitate HPC systems in production environments, PV-EASY also employs the simplest and universally applicable mechanisms supported by all HPC systems. In *shadow load preemption* (SLP), the kill/restart preemption mode is employed and in *venture backfilling* (VB), the simplest Last Model is used to perform prediction. Both kill/restart mode and Last Model are supported by all HPC systems and can be easily implemented and replace EASY in production schedulers.

Currently, though suspend/resume and checkpoint/restart modes could better reduce resource waste than kill/restart mode, we do not adopt them in SLP, due to the following two consideration. First, both of these two modes need support from systems and applications, and it is also not realistic to ask existing application providers or users to modify their applications or jobs to fit these modes. Therefore, employing either of these two modes would limit the scope of the applicability of PV-EASY. Second, these two modes would still cause resource waste. Suspend and checkpoint operations also need time to save runtime environments, so part of a job's work would still be lost in checkpoint/restart mode.

However, benefiting from virtualization and other techniques, live job migration and fast runtime environments save/load will be widely employed in HPC systems while operation cost will decrease as technology advances. In such cases, replacing the kill/restart mode with suspend/resume or checkpoint/restart mode in PV-EASY will become natural and definitely lead to better performance.

Another noticeable phenomenon in our experiment is that even with poor accuracy, EASY with user estimates achieves smaller MBS and MWBS than both EASY-Last (Figure 5 and Figure 6) and EASY with a "virtual" predictor (Figure 4) whose maximum prediction error is limited within 10%. 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 [39] would occur, making EASY approximate SJF, and therefore achieves good performance via serious sacrifice of fairness. Thus in our view, runtime prediction is helpful for parallel job scheduling, 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* (PV-EASY) strategy, which integrates novel *shadow load preemption* (SLP) and *venture backfilling* (VB) approaches. We designed an event-based parallel job scheduling simulator and conducted experiments on three workloads collected from real HPC systems. Results show that our PV-EASY is very attractive from both academic and industry perspectives in the following aspects:

• PV-EASY can guarantee strict fairness because of SLP, and also achieves attractive performance compared with EASY due to VB. These facts indicate that PV-EASY can leverage fairness and performance much better than EASY in parallel job scheduling.

• PV-EASY can benefit more from prediction techniques than EASY. PV-EASY can avoid reservation violation that arises from employing prediction in EASY. Moreover, PV-EASY can achieve much better performance than EASY with existing parallel job runtime prediction techniques, and will continue its superiority against EASY for the foreseeable future (as long as the maximum prediction error is not smaller than 10%, which is far beyond the capability of current prediction techniques).

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

8. ACKNOWLEDGMENTS

The authors would like to thank Dr. Zhiling Lan and Dr. Michela Taufer for their shepherds of this paper. The authors would also like to thank Dan Dwyer and Steve Hotovy for the CTC workload logs, Travis Earheart and Nancy Wilkins Diehr for the SDSC-Blue workload logs, and Victor Hazlewood for SDSC-DS workload logs, especially Dror Feitelson for his great work of collecting and publishing high performance computing workloads in Parallel Workload Archive on line. This Work is supported by Natural Science Foundation of China (60803121, 60773145, 60911130371, 90812001, 60963005), National High-Tech R&D (863) Program of China (2009AA01A130, 2006AA01A101, 2006AA01A108, 2006AA01A111, 2006AA01A117) and MOE-Intel Foundation.

9. REFERENCES

[1] Lifka, D.A., The ANL/IBM SP scheduling system. In 1st Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), 1995.

[2] Feitelson, D.G., Experimental analysis of the root causes of performance evaluation results: a backfilling case study. IEEE Transactions on Parallel and Distributed Systems, 2005: p. 175-182.

[3] Mu'Alem, A.W. and Feitelson, D.G., Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP 2 with backfilling. IEEE Transactions on Parallel and Distributed Systems, 2001. 12(6): p. 529-543.

[4] Karger, D., Stein, C., and Wein, J., Scheduling algorithms. CRC Handbook of Computer Science, 1997.

[5] Sgall, J. On-line scheduling – a survey. In A. Fiat and G. Woeginger, editors, On-Line Algorithms: The State of the Art, Lecture Notes in Computer Science, pages 196–231. Springer-Verlag, 1998.

[6] Majumdar, S., Eager, D.L., and Bunt, R.B., Scheduling in multiprogrammed parallel systems. ACM SIGMETRICS Performance Evaluation Review, 1988. 16(1): p. 104-113.

[7] Sevcik, K.C., Application scheduling and processor allocation in multiprogrammed parallel processing systems. Journal of Performance Evaluation, 1994. 19: p. 107-140

[8] AuYoung, A., Vahdat A., and Snoeren, A.C., Evaluating the Impact of Inaccurate Information in Utility-Based Scheduling. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC), 2009.

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

[10] Chiang, S.H., Arpaci-Dusseau, A., and Vernon, M.K., The impact of more accurate requested runtimes on production job scheduling performance. In 8th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), 2002.

[11] Srinivasan, S., Kettimuthu,R., Subramani,V., and Sadayappan, P., Selective reservation strategies for backfill job scheduling. In 8th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), 2002.

[12] Ward, W.A., Mahood, C.L. and West, J.E., Scheduling jobs on parallel systems using a relaxed backfill strategy. In 8th

Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), 2002.

[13] Shmueli, E. and Feitelson, D.G., Backfilling with lookahead to optimize the packing of parallel jobs. Journal of Parallel and Distributed Computing, 2005. 65(9): p. 1090-1107.

[14] Jones, J.P. and Nitzberg, B., Scheduling for parallel supercomputing: a historical perspective of achievable utilization. In 5th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), 1999.

[15] Talby, D. and Feitelson, D.G., Supporting Priorities and Improving Utilization of the IBM SP Scheduler Using Slack-Based Backfilling. In Proceedings of the 13th International Symposium on Parallel Processing (IPPS),1999.

[16] Tsafrir, D., Etsion, Y. and Feitelson, D.G., Backfilling using system-generated predictions rather than user runtime estimates. IEEE Transactions on Parallel and Distributed Systems, 2007. 18(6): p. 789.

[17] Thebe, O., Bunde, D.P. and Leung. V.J., Scheduling Restartable Jobs with Short Test Runs. In 14th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), 2009.

[18] Guim, F., Rodero, I. and Corbalan, J., The resource usage aware backfilling. In 14th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), 2009.

[19] Kurian, R., Balaji, P. and Sadayappan, P., Opportune job shredding: An effective approach for scheduling parameter sweep applications. In Los Alamos Computer Science Institute Symposium, New Mexico, 2003.

[20] Sabin, G., et al., Scheduling of parallel jobs in a heterogeneous multi-site environment. In 9th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), 2003.

[21] Shmueli, E. and Feitelson, D.G., On simulation and design of parallel-systems schedulers: are we doing the right thing?. IEEE Transactions on Parallel and Distributed Systems, 2009. 20(7): p. 983-996

[22] Raz, D., Levy, H. and Avi-Itzhak, B., A resource-allocation queueing fairness measure. ACM SIGMETRICS Performance Evaluation Review, 2004. 32(1): p. 130-141.

[23] Avi-Itzhak, B., Levy, H. and Raz, D., Quantifying fairness in queueing systems: Principles and applications, in the Engineering and Informational Sciences, v.22 n.4, p.495-517, October 2008.

[24] Isard, M., et al., Quincy: Fair Scheduling for Distributed Computing Clusters. In ACM SIGOPS 22nd symposium on Operating systems principles (SOSP), 2009.

[25] Mann, L., Queue culture: The waiting line as a social system. The American Journal of Sociology, 1969. 75(3): p. 340-354.

[26] Larson, R.C., Perspectives on queues: social justice and the psychology of queueing. Operations Research, 1987. 35(6): p. 895-905.

[27] Sabin, G. and Kochhar, G., Job Fairness in Non-Preemptive Job Scheduling. In Proceedings of the 2004 International Conference on Parallel Processing (ICPP), 2004

[28] Avi-Itzhak, B., Brosh, E. and Levy, H., SQF: A slowdown queueing fairness measure. Performance Evaluation, 2007. 64(9-12): p. 1121-1136

[29] Ngubiri, J. and van Vliet, M., Characteristics of fairness metrics and their effect on perceived scheduler effectiveness. 2007, Technical Report, Radboud University Nijmegen.

[30] Lee, C.B. and Snavely, A., On the user-scheduler dialogue: Studies of user-provided runtime estimates and utility functions. International Journal of High Performance Computing Applications, 2006. 20(4): p. 495.

[31] Lee, C.B., et al., Are user runtime estimates inherently inaccurate?. In 10th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), 2005.

[32] Tang, W., Lan, Z., Desai, N. and Buettner, D., Fault-Aware, Utility-Based Job Scheduling on Blue Gene/P Systems. In 2009 IEEE International Conference on Cluster Computing (Cluster),2009.

[33] Susukita, R., et al. Performance prediction of large-scale parallell system and application using macro-level simulation, in Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC),2008.

[34] Kapadia, N.H., Fortes, J. and Brodley, C.E., Predictive application-performance modeling in a computational grid environment . In 8th IEEE Int'l Symp. on High Performance Distributed Computing (HPDC), p. 6, Aug 1999.

[35] Krishnaswamy, S., Loke, S.W. and Zaslavsky, A., Estimating computation times of data-intensive applications. IEEE Distributed Systems Online, 2004. 5(4).

[36] Lee, C.B. and Snavely, A.E., Precise and realistic utility functions for user-centric performance analysis of schedulers, In 16th International Symposium on High Performance Distributed Computing (HPDC),2007

[37] Perkovic, D. and Keleher, P.J., Randomization, speculation, and adaptation in batch schedulers. in Proceedings of the 2000 ACM/IEEE conference on Supercomputing (SC). 2000.

[38] Zotkin, D. and Keleher, P.J., Job-Length Estimation and Performance in Backfilling Schedulers. in Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC). 1999

[39] Tsafrir, D., Feitelson, D.G.: The dynamics of backfilling: solving the mystery of why increased inaccuracy may help. In: IEEE International Symposium on Workload Characterization, pp. 131–141 (2006)

[40] Nadeem, F. and Fahringer, T., Predicting the execution time of grid workflow applications through local learning. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC),2009

[41] Parallel Workloads Archive, http://www.cs.huji.ac.il/labs/parallel/workload/.

[42] Yero, E. and Henriques, M., Contention-sensitive static performance prediction for parallel distributed applications. Performance Evaluation, 2006. 63(4-5): p. 265-277.

[43] Jackson, D., Maui/Moab default configuration. with CTO of Cluster Resources, 2006