Backfilling Using Runtime Predictions Rather Than User Estimates

Dan Tsafrir Yoav Etsion Dror G. Feitelson

School of Computer Science and Engineering

The Hebrew University, 91904 Jerusalem, Israel

{dants,etsman,feit}@cs.huji.ac.il

Abstract

The most commonly used scheduling algorithm for parallel supercomputers is FCFS with backfilling, as originally introduced in the EASY scheduler. Backfilling means that short jobs are allowed to run ahead of their time provided they do not delay previously queued jobs (or at least the first queued job). To make such determinations possible, users are required to provide estimates of how long jobs will run, and jobs that violate these estimates are killed. Empirical studies have repeatedly shown that user estimates are inaccurate, and that history based system-generated predictions may be significantly better. However, predictions have not been incorporated into production schedulers, partially due to common wisdom claiming inaccuracy actually improves performance, but mainly due to the difficulty of what to do when job runtimes exceed system-generated predictions: With backfilling such jobs are killed, but users will not tolerate jobs being killed just because system predictions were too short. We solve this problem by divorcing kill-time from the runtime prediction, and correcting predictions adaptively as needed if they are proved wrong. The end result is a surprisingly simple scheduler, which requires minimal deviations from current practices (e.g. using FCFS as the basis), and behaves exactly like EASY as far as users are concerned; nevertheless, it achieves significant improvements in performance, predictability, and accuracy. Moreover, the techniques presented in this paper can be used to enhance any backfilling algorithm.

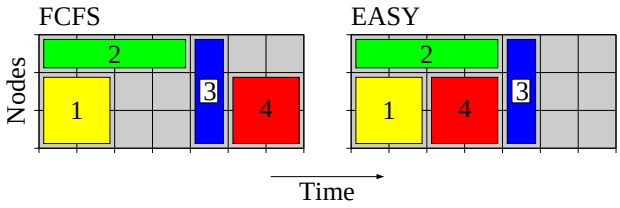

Figure 1: *EASY scheduling reduces fragmentation by backfilling. Note that it would be impossible to backfill job 4 had its length been more than 2, as the reservation for job 3 would have been violated.*

1 Introduction

The default algorithms used by current batch job schedulers for parallel supercomputers are all rather similar to each other [6]. In essence, they select jobs for execution in first-come-first-serve (FCFS) order, and run each job to completion. The problem is that this simplistic approach causes significant fragmentation, as jobs do not pack perfectly and processors are left idle. Most schedulers therefore use *backfilling*: if the next queued job cannot run because sufficient processors are not available, the scheduler nevertheless continues to scan the queue, and selects smaller jobs that may utilize the available resources. This improves utilization by $\sim$15 percentage points [15].

A potential problem with backfilling is that the first queued job may be starved as subsequent jobs continually jump over it. This is solved by making a reservation for this job, and allowing subsequent jobs to run only if they do not violate this reservation (Fig. 1). The approach of making a reservation for the first job only was introduced by EASY: the first backfilling scheduler [20]. Many backfilling variants have been suggested since, e.g. using more reservations, scanning waiting jobs for backfilling in non-FCFS order, etc. [10]. However, the default of most parallel schedulers (including Maui/Moab [13] and IBM's load-leveler [16]) has remained plain EASY [6].

Note that backfilling requires the running time of jobs to be known: first, we need to know when running jobs will terminate and free up their processors, to enable us to compute when to make the reservation. Second, we need to know that backfilled jobs are short enough to terminate before the reservation time. Therefore EASY required users to provide a runtime estimate for all submitted jobs [20], and the practice continues to this day. These estimates are used by the scheduler to make scheduling decisions, and jobs that exceed their estimates are killed so as not to

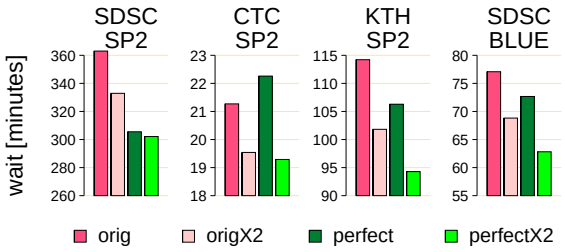

Figure 2: *The average wait-time of jobs improves when runtime estimates are doubled, both when using the original user estimates ("orig") and when using accurate estimates ("prefect").*

violate subsequent commitments.

Workload traces from sites that actually use EASY allow for empirical studies of how it works in practice. These studies show that user estimates are generally inaccurate [21]. A possible reason is that users find the motivation to overestimate — so that jobs will not be killed — much stronger than the motivation to provide accurate estimates and help the scheduler to perform better packing. Moreover, a recent study indicates that users are actually quite confident of their estimates, and most probably would not be able to provide much better estimates [18].

The dire results regarding user estimates have prompted research in two directions: evaluating the impact of such bad estimates, and looking for alternatives. Surprisingly, studies regarding the impact of inaccuracy have found that it actually leads to improved performance [8]. This has even led to the suggestion that estimates should be *doubled* [31, 21] or *randomized* [22], to make them even less accurate. The results of doubling (Fig. 2) indeed exhibit remarkable improvements.

The search for alternative estimates has focused on using historical data. Users of parallel machines tend to repeat the same type of work over and over again (Fig. 3). It is therefore conceivable that historical data can be used to predict the future. Suggested prediction schemes include using the top of a 95% confidence interval of job runtimes [12], a statistical model based on the assumption that runtimes come from a log-uniform distribution [5], and simply to use the mean plus 1.5 standard deviations [21].

Despite all this work, schedulers in actual use still prefer plain backfilling based on user estimates over system-generated predictions based on history. The reasons for this situation are

1. Some of the suggested prediction schemes are complex. For example, Smith et al. have

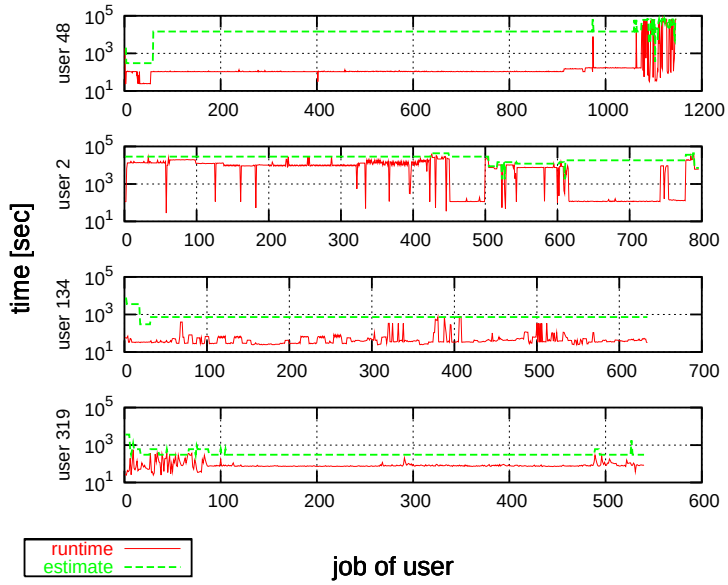

Figure 3: *Runtime and estimate of all the jobs submitted by four arbitrary users (from the SDSC-SP2 trace) show remarkable repetitiveness.*

suggested the use of sophisticated greedy and genetic algorithms for on-line classification of jobs into similarity classes [24, 26].

2. Accurate predictions are problematic for a backfilling scheduler, because of the need to kill jobs that exceed their predictions. In one case, predictions that are tight enough to be useful led to underestimation for about 20% of the jobs [21].

3. The perception that inaccurate estimates lead to better performance negates the motivation to incorporate mechanisms for better predictions.

This paper is about refuting or dealing with these three claims. Regarding complexity, we show that even extremely trivial algorithms result in significant improvement, both in the accuracy of the prediction itself and the resulting performance when using the improved prediction. For example, when done correctly, setting the prediction of a job to be the average runtime of the two preceding jobs by the same user constitutes a prefectly good predictor.

The issue of integrating good predictions with backfilling turned out to be quite simple as well. In the original EASY scheduler, and in all schedulers since then, user runtime estimates have two roles: to tell the system how long jobs will run, and to serve as part of the user contract — if the

job runs longer than the estimate, the system may kill it. When good predictions are introduced, they should only replace the first role, and be used for better scheduling decisions. The second role, being part of the user contract, is left to the user estimates.

The third argument, that "inaccuracy helps", is actually false in three respects. First, Fig. 2 indeed shows that doubling user estimates outperforms using the original and even perfect estimates. But doubling may also be applied to perfect estimates, leading to consistently better performance. We therefore have reason to believe that doubling of good predictions will be similar (outperform doubling of estimates).

Second, the reason that doubling helps is that it allows short waiting jobs to move forward within an FCFS setting, implicitly approximating an SJF-like schedule [29]. (Indeed, most studies dealing with predictions indicate that increased accuracy improves performance when shorter jobs are favored [12, 26, 31, 17, 22, 2].) Consequently, attributing the improvement to inaccurate estimates is wrong, as SJF backfill-order may be enforced explicitly with far greater success. By still preserving FCFS reservation-order, we can maintain EASY's appeal and enjoy both worlds: a fair scheduler that nevertheless backfills effectively.

The third fallacy in the "inaccuracy helps" claim is the underlaying implied assumption that good predictions are unimportant. In fact, they are important in various contexts. One example is advance reservations for grid allocation and co-allocation, shown to considerably benefit from better accuracy [25, 19]. Another is scheduling moldable jobs that may run on any number of nodes [5, 26, 4]. The scheduler's goal is to minimize response time, by considering whether waiting for a while for more nodes to become available is preferable over running immediately. Thus a reliable prediction of how long it will take for additional nodes to become available is crucial.

The rest of the paper is structured as follows. As our approach is based on experimentation, the next section presents our methodology. Section 3 explains how prediction-based backfilling is done, and demonstrates the improvements in performance. Section 4 shows the effect on predictability. Section 5 demonstrates the generality of the techniques we suggest, and Section 6 presents an investigation of optimal parameter settings for the algorithm.

Abbreviation	Site	CPUs	Jobs	Start	End	Util
CTC-SP2	Cornell Theory Center	512	77,222	Jun 96	May 97	56%
KTH-SP2	Swedish Royal Instit. Tech.	100	28,490	Sep 96	Aug 97	69%
SDSC-SP2	San-Diego Supercomp. Ctr.	128	59,725	Apr 98	Apr 00	84%
SDSC-BLUE	San-Diego Supercomp. Ctr.	1,152	243,314	Apr 00	Jun 03	76%

Table 1: *Traces used to drive simulations. The first three traces are from machines using the EASY scheduler. The fourth (SDSC Blue Horizon) uses the LoadLeveler infrastructure and the Catalina scheduler (that also performs backfilling and supports reservations).*

2 Methodology

The experiments are based on an event-based simulation of EASY scheduling, where events are job arrival / termination. Upon arrival, the scheduler is informed of the processor-number the job needs, and its estimated runtime. It can then start the job's simulated execution or place it in a queue. Upon a job termination, the scheduler is notified and can schedule other queued jobs on free processors. Job runtimes are part of the simulation input, but are not given to the scheduler.

Table 1 lists the four traces we used to drive the simulations. As suggested in the Parallel Workloads Archive, we've chosen to use their "cleaned" versions [1, 30]. Since traces span the past decade, were generated at different sites, by machines with different sizes, and reflect different load conditions, we have reason to believe consistent results obtained in this paper are truly representative. Traces are simulated using the exact data provided, with possible modifications as noted (e.g. to check the impact of replacing user estimates with system generated predictions).

Scheduler performance is measured using wait times and bounded slowdown. Slowdown is response time (wait plus running time) normalized by running time. Bounded slowdown eliminates the emphasis on very short jobs due to having the running time in the denominator; a commonly used threshold of 10 seconds was set [11] yielding the formula

$$bounded\_slowdown = \max\left(1\ ,\ \frac{T_w + T_r}{\max\left(10, T_r\right)}\right)$$

where T_r and T_w are the job's run and wait times, respectively. To reduce warmup effects, the first 1% of terminated jobs were not included in the metric averages [14].

The measure of accuracy is the ratio of the real runtime to the prediction. If the prediction is larger than the runtime, this reflects the fraction of predicted time that was actually used. But predictions can also be too short. Consequently, to avoid under- and over-prediction canceling themselves out (when averaged), we define

$$accuracy = \begin{cases} 1 & \text{if } P = T_r \\ T_r/P & \text{if } P > T_r \\ P/T_r & \text{if } P < T_r \end{cases}$$

where P is the prediction; the closer the accuracy is to 1 (never bigger), the more accurate the prediction. This can be averaged across jobs, and also along the lifetime of a single job, if the system updates its prediction. In that case a weighted average is used, where weights reflect the relative time that each prediction was in effect.

3 Incorporating Predictions into Backfilling Schedulers

The simplest way to incorporate system-generated predictions into a backfilling scheduler is to use them in place of user-provided estimates[1]. The problem of this approach is that aside from serving as a runtime *approximation*, estimates also serve as the runtime *upper-bound* (kill-time). But predictions might happen to be shorter than actual runtimes, and users will not tolerate their jobs being killed just because the system speculated they were shorter than the user estimate. And so, predictions can't "just" replace estimates.

Previous studies have dealt with this difficulty either by eliminating the need for backfilling (e.g. by using pure SJF [12, 26]), by assuming preemption is available (stopping jobs that exceed their prediction and reinserting them into the wait queue [12]), or by considering only artificial estimates generated as multiples of actual runtimes (effectively assuming underestimation — and

[1]Note the terminology: we will consistently use "estimate" for the runtime approximation provided by the user upon job submittal, and "prediction" for the approximation as used by the scheduler. In EASY, predictions and estimates are equal, that is, the predictions are set to be the user estimates. The alternative is to use historical data to generate better predictions, as we do.

underprediction — never occurs) [31, 17, 22, 3, 2, 27, 28]. Mu'alem and Feitelson [21] noted this problem, and investigated whether underprediction does in fact occur when using a conservative predictor (average of previous jobs with the same user / size / executable, plus $1\frac{1}{2}$ times their standard deviation). They found that around 20% of the jobs suffered from underprediction and would have been killed prematurely by a backfilling scheduler. They concluded that "it seems using system-generated predictions for backfilling is not a feasible approach".

3.1 Separating the Dual Roles of Estimates

The key idea of our solution is recognizing that the underestimation problem emanates from the dual role an estimate plays: both as a prediction and as a kill-time. We argue that these should be separated. It is only legitimate to kill a job *once its user estimate is reached*, but not any sooner; therefore the main function of user estimates is to serve as kill-times. All the other considerations of a backfilling scheduler should be based upon *the best available predictions* of how long jobs will really run; this can be the user estimate, but it can also be generated by the system, and moreover, it can change over time. Consequently, other than replacing estimates with predictions, the only modification we introduce to the scheduler itself is that a running job is *not* killed when its prediction is reached; rather, it is allowed to continue, and is only killed when (if) it reaches its estimate. This entirely eliminates the problem of premature killings.

The system-generated prediction algorithm we use is very simple. The prediction of a new job J is set to be the average runtime of the two most recent jobs that were submitted by the same user prior to J, and that have already finished (if only one previously finished job exists we use its runtime as the prediction; if no such job exists we fall back on the associated user estimate; other ways to select the history jobs are considered in Section 6.1). Requiring previous jobs to be finished is of course necessary since only then are their runtimes known. A prediction is assigned to a job only if it is smaller than its estimate (maximal kill-time). Implementing this predictor is truly trivial and requires less than a dozen lines of code: saving the runtime of the two most recent

trace	wait [minutes]			bounded slowdown			accuracy [%]		
	EASY	EASY-PRED		EASY	EASY-PRED		EASY	EASY-PRED	
SDSC-SP2	363	757	(+109%)	99	233	(+136%)	32	55	(+70%)
CTC-SP2	21	29	(+38%)	5	7	(+53%)	39	56	(+44%)
KTH-SP2	114	968	(+748%)	90	746	(+729%)	47	49	(+4%)
SDSC-BLUE	77	178	(+130%)	22	56	(+156%)	30	57	(+90%)

Table 2: *Average wait time, bounded slowdown, and accuracy for EASY using user estimates or system generated predictions (EASY-PRED). Numbers in parentheses are change relative to EASY; these are always positive, which is good for accuracy, but bad for the other performance metrics.*

jobs in a per-user data structure, updating it when more recently submitted jobs terminate, and averaging the two runtimes when a new job arrives.

Table 2 shows performance results of a system using original EASY vs. a system in which estimates are replaced with our automatically generated predictions. These results indicate a colossal failure. Both performance metrics (average wait and slowdown) consistently show that using predictions results in severe performance degradation (more than a factor of 8 for KTH). This happens despite the improved accuracy of the predictions.

3.2 The Role of Underprediction

The reason for the dismal performance results is not our simplistic prediction algorithm, as even these simple predictions are usually far superior to the estimates supplied by users in terms of accuracy (right of Table 2). Rather, it is an unfortunate interaction between underprediction (cases in which a system-generated prediction is smaller than the job's actual runtime) and the workings of the backfilling scheduler.

The problem is that a backfill scheduler blindly uses the supplied predictions as the basis for all its decisions. In order to make a reservation, the scheduler traverses currently running jobs in order of predicted termination, and accumulates the processors allocated to these jobs. Once this sum is equal to or bigger than the size of the first queued job, a reservation is made for the predicted termination time of the last job in the traversal, based on the assumption that at this time all the required processors will be free. This includes the implicit assumption that jobs that exceed their

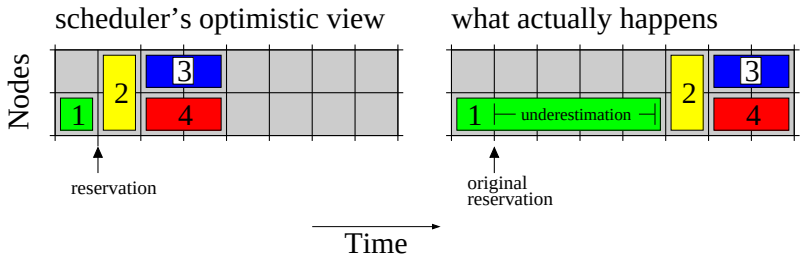

Figure 4: *Underprediction of runtimes causes the backfill scheduler to make reservations that are too early. This misconception then prevents subsequent jobs from being backfilled, due to fear that they will interfere with the reservation for the first queued job.*

predicted runtime are killed, and that killing occurs instantly.

The truth of the matter is that jobs may actually continue to run beyond their predicted termination time. This is obviously the case when predictions are used and some of the predictions come out short. But it also happens in the original traces, because some jobs are not killed for some reason, and even if they are, this may take several minutes.

When a job continues to run beyond its predicted termination, there is a discrepancy between the number of processors the backfill scheduler expects to be available, and the actual capacity, which is smaller. The scheduler checks the capacity, and finds that the first queued job cannot run. It therefore decides to make a reservation. But the reservation is based on the predicted end times, and includes processors that should be available but in reality are not. This leads to a reservation which is unrealistically early. In the extreme case, the scheduler might even be tricked into making a reservation for the current time! (This happens if predictions for enough running jobs have already expired.)

As the reservation time sets an upper-bound on the duration of backfilled jobs (at least those that don't use the extra processors [20, 21]), the practical meaning of such a situation is a massive reduction in backfilling activity. When the reservation is too early, a smaller group of jobs is eligible for backfilling: those jobs that fit into the "hole" in the schedule, which appears to be much smaller than it actually is; backfilling is completely disabled if the reservation is for the current time. The scheduling then largely reverts to plain FCFS, as shown in Fig. 4, leading to the poor performance shown in Table 2. This situation is rectified only when underpredicted jobs

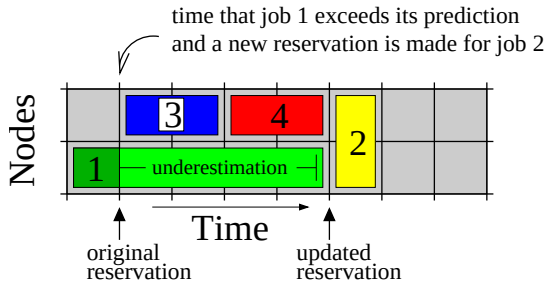

Figure 5: *Prediction correction enables the backfill scheduler to escape from the early reservations that prevent backfilling. Compare with Fig. 4.*

eventually terminate.

3.3 Prediction Correction

One way to tackle the underestimation problem is to try to minimize it by producing more conservative (bigger) predictions. The drawbacks of this approach are reduced accuracy and performance (recall that jobs with tight estimates have increased chance to be backfilled). In addition, this will not completely eliminate the danger of underestimation.

An alternative is to modify the scheduling algorithm, and increase expired predictions proven to be too short. In other words, if a job's prediction indicated it would run for 10 minutes, and this time has already passed but the job is still alive, simply accept the fact it will run longer. Once the prediction is updated, this effects reservations for queued jobs and re-enables backfilling (Fig. 5).

Prediction-correction is required when the prediction was too short. The common case is that this prediction was smaller than the user estimate. In this case we acknowledge the fact that the user was smarter than us and set the new prediction to be the estimate as was given by the user.

On rare occasions the prediction is equal to or bigger than the estimate. Jobs that exceed their estimates actually occur in our data traces. In most cases the overshoot is very short, not more than a couple of minutes, and probably reflects the time needed to kill the job. But in some cases it is much longer, for unknown reasons. In any case, we need to somehow extend the prediction used by the scheduler, to keep it up to date with what is actually happening. Note that this is independent of the fact that the job should be killed (and maybe *is* being killed).

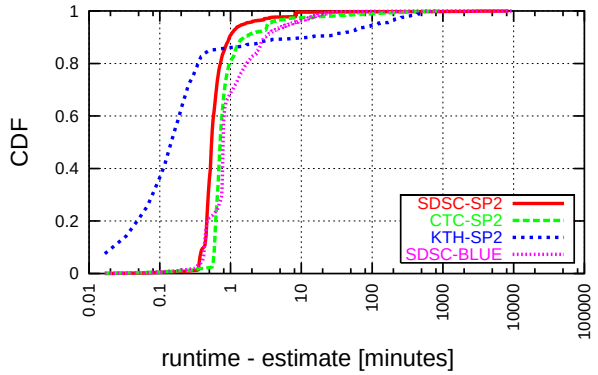

Figure 6: *Cumulative distribution function of the difference between runtime and estimate of underestimated jobs (estimate smaller than runtime). Most estimate violations are less than one minute.*

As most of these jobs only exceed their estimate by a short time, independent of their total length (Fig. 6), we enlarge post-estimate predictions in a gradual but fixed manner. The first adjustment adds only one minute to the old prediction. This will cover the majority of the jobs. if this is not enough, we add 5 minutes, then 15, then 30, followed by 1, 2, 5, and 10 hours (and so on).

The results of using prediction correction are shown in Table 3. This compares the original EASY algorithm with a version that includes prediction correction (EASY-PCOR), and a version that combines prediction correction with system generated predictions (EASY+).

Prediction-correction is applicable even for vanilla EASY, to deal with underestimation cases as mentioned above. This by itself only has a marginal (positive) effect, because only a small fraction of the jobs run for more than their user estimates. The real value of prediction correction is revealed in EASY+, when system-generated predictions are added. This extremely trivial optimization turns a consistent failure of nearly an order of magnitude degradation in performance (KTH's wait time in Table 2) into a consistent improvement of up to 19% (BLUE's slowdown in Table 3). Although the improvement in performance is moderate, this is an important result that shouldn't be taken lightly. It teaches us that the only thing preventing backfilling schedulers from utilizing good prediction schemes is the absence of prediction correction. The fact that historical information can be successfully used to generate runtime predictions is known for more than a decade [9, 12]. Results in Table 3 prove for the first time that this may be put to productive use within backfilling

metric	trace	EASY	EASY-PCOR	EASY+
wait [min]	SDSC-SP2	363.0	362.8 (−0%)	340.7 (−6%)
	CTC-SP2	21.3	21.1 (−1%)	18.3 (−14%)
	KTH-SP2	114.2	114.4 (+0%)	108.3 (−5%)
	SDSC-BLUE	77.1	76.9 (−0%)	69.0 (−10%)
bounded slowdown	SDSC-SP2	99.0	94.1 (−5%)	87.5 (−12%)
	CTC-SP2	4.6	4.5 (−2%)	3.8 (−17%)
	KTH-SP2	89.9	89.9 (0%)	79.2 (−12%)
	SDSC-BLUE	21.7	21.3 (−2%)	17.5 (−19%)
accuracy [%]	SDSC-SP2	32.2	32.2 (0%)	55.0 (+71%)
	CTC-SP2	38.8	38.9 (0%)	58.4 (+51%)
	KTH-SP2	47.3	47.4 (0%)	57.9 (+22%)
	SDSC-BLUE	30.1	30.2 (0%)	55.4 (+84%)

Table 3: *Average wait, bounded slowdown, and accuracy for scheduler variants. EASY-PCOR adds prediction correction, and EASY+ adds system generated predictions. Numbers in parentheses are improvement relative to EASY.*

batch schedulers, without violating the contract with users. Moreover, the overhead is low, with predictions corrected 0.1 times on average per job.

Note that obtaining the reported improvement is almost free. All one has to do to obtain it is create predictions as the average runtime of two previously submitted jobs, and set an alarm event to correct those predictions that prove too short. Importantly, this does not change the way users view the scheduler, allowing the popularity of EASY to be retained. Finally, we remark that this scheme also has a positive effect on average accuracy, which stabilizes between 55–58% across all four traces when using EASY+.

3.4 Shortest Job Backfilled First (SJBF)

A well known scheduling principle is that favoring shorter jobs significantly improves overall performance. Supercomputer batch schedulers are one of the few types of systems which enjoy a-priori knowledge regarding runtimes of scheduled tasks, whether through estimates or predictions. Therefore SJF scheduling may actually be applied.

There is a wealth of studies related to predictions and accuracy (within the context of backfilling schedulers) demonstrating that the benefit of accuracy dramatically increases if shorter jobs are

favored [12, 26, 31, 17, 22, 2, 23]. For example, Chiang et al. [2] show that when replacing user estimates with actual runtimes, while ordering the wait queue by descending $\sqrt{\frac{T_w+T_r}{T_r}}+\frac{T_w}{100}$, average and maximal wait times are halved and slowdowns are an order of magnitude lower![2]

Unlike predictions, usually considered as unusable in the context of backfilling, configuring mainstream schedulers to favor (estimated) short jobs is certainly possible (and in PBS this is even the default). However, in most schedulers the often-used default is essentially the same as in EASY [6], which may perhaps be attributed to a reluctance to change FCFS-semantics perceived as being the most fair. Such reluctance has probably hurt previously suggested non-FCFS schedulers, that impose the new ordering as a "package deal", affecting both backfilling and reservation order (for example, with SJF, a reservation made for the first queued job helps the shortest job, rather than the one that has been delayed the most). In contrast, we suggest separating the two.

Our scheme introduces a controlled amount of "SJFness", but preserves EASY's FCFS nature. The idea is to keep reservation order FCFS (as in EASY) so that *no job will be backfilled if it delays the first job in the wait queue*. On the other hand, the backfilling optimization is implemented in SJF order, that is, Shortest Job Backfilled First — *SJBF*. This is acceptable because the first-fit essence of backfilling is a departure from FCFS anyway. We argue that in any case, explicit SJBF is more sensible than "tricking" EASY into SJFness by doubling estimates [31, 21], randomizing them [22], or other similar stunts.

Results of applying SJBF are shown in Table 4. In its simplest version this reordering is used with conventional EASY (i.e. using user estimates and no prediction correction). Even this leads to typical improvements of more than 10%, and up to 27% (BLUE's bounded slowdown), quite similar to that of EASY+.

Much more interesting is EASY++ which adds SJBF to EASY+ (namely combines prediction correction, system-generated predictions, and SJBF). This usually results in double to triple the performance improvement in comparison to EASY-SJBF and EASY+. Performance gains are especially pronounced for bounded slowdown where EASY++ may actually double performance

[2]Recall that T_w and T_r are wait- and run-times. Short jobs are favored since the numerator of the first term rapidly becomes bigger than its denominator. The second term is added in an effort to avoid starvation.

metric	trace	EASY	EASY-SJBF	EASY++	PERFECT++
wait [min]	SDSC-SP2	363	362 (−0%)	326 (−10%)	278 (−23%)
	CTC-SP2	21	19 (−10%)	14 (−33%)	19 (−10%)
	KTH-SP2	114	102 (−11%)	94 (−18%)	91 (−20%)
	SDSC-BLUE	77	68 (−12%)	53 (−31%)	57 (−26%)
bounded slowdown	SDSC-SP2	99	89 (−10%)	72 (−28%)	58 (−42%)
	CTC-SP2	5	4 (−13%)	3 (−37%)	3 (−39%)
	KTH-SP2	90	73 (−18%)	57 (−37%)	50 (−44%)
	SDSC-BLUE	22	16 (−27%)	11 (−50%)	10 (−54%)
accuracy [%]	SDSC-SP2	32	32 (0%)	61 (+89%)	100 (+211%)
	CTC-SP2	39	39 (0%)	63 (+63%)	100 (+158%)
	KTH-SP2	47	47 (0%)	63 (+33%)	100 (+111%)
	SDSC-BLUE	30	30 (0%)	60 (+98%)	100 (+232%)

Table 4: *Average wait, bounded slowdown, and accuracy of EASY compared with three improved variants: EASY-SJBF just adds SJF backfilling (based on original user estimates). EASY++ employs all our optimizations: system-generated predictions, prediction correction, and SJBF. PERFECT++ is the optimum, using SJBF with perfect predictions (actual runtimes). Improvement is shown relative to traditional EASY.*

in comparison to EASY (SDSC-BLUE). There is also a non-negligible 33% peak improvement in average wait (CTC). This is quite impressive for a scheduler with basic FCFS semantics. Even more impressive is the *consistency* the of results, which all point to the same conclusion, as opposed to other experimental evaluations in which results depended on the trace or even the metric being used [26, 7]. Accuracy is also improved from 30–47% when using estimates, through 55–58% in EASY+, to 60–63% in EASY++.

Finally, we have also checked what would be the impact of having perfect predictions when SJBF is employed (in this scenario there is no meaning to prediction correction because predictions are always correct). It turns out PERFECT++ is sometimes marginally and sometimes significantly better than EASY++ with the difference being most pronounced in SDSC-SP2, which is the site with the highest load (Table 1). Analysis not included in this paper indeed revels that the role of accuracy becomes crucial as load conditions increase, generating a strong incentive for developing better prediction schemes than those presented in this paper.

Interestingly, EASY++ outperforms PERFECT++ in wait times obtained for SDSC-BLUE (minor difference) and CTC (major). The reason explaining the former is probably similar to the explanation of why PERFECT++ is only marginally better than EASY in some cases: since EASY++

algorithm	optimization		
	prediction correction	replace estimate with prediction	SJBF
EASY			
EASY-PCOR	√		
EASY-SJBF			√
EASY+	√	√	
EASY++	√	√	√
PERFECT++	N/A	(with runtime)	√

Table 5: *Summary of algorithms and optimizations they employ.*

is inherently inaccurate it enjoys an effect similar to doubling of estimates (creates "holes" in the schedule in which shorter jobs may fit). In this sense PERFECT++ is "more FCFS" than EASY++ and therefore pays the price. The major CTC difference is probably due to subtle backfilling issues and a fundamental difference between CTC and the other traces, as analyzed by Feitelson [7].

3.5 Optimizations Summary

To summarize, three optimizations were suggested: (1) prediction correction where predictions are updated when proven wrong, (2) simple system-generated predictions based on recent history of users, and (3) SJBF in which backfilling order is shortest job first. All optimizations maintain basic FCFS semantics. They are all orthogonal in the sense that they may be applied separately. However, using system generated predictions without prediction correction leads to substantially decreased performance. The combination of all three consistently yields the best improvement of up to doubling performance in comparison to the default configuration of EASY. The algorithms covered and the optimizations they employ are summarized in Table 5 for convenience. The rest of the paper will focus on EASY+ and EASY++.

4 Predictability

Previous sections have shown that, on average, replacing estimates with system-generated predictions is beneficial in terms of performance and accuracy. However, when abandoning estimates in

favor of predictions, we might lose *predictability*. The original EASY backfilling rule states that a job J can be backfilled if its estimated termination time does not violate the reservation time T of the first job in the wait queue. Since J is killed when reaching its estimate, it is guaranteed that the first job will indeed be started no later than T. However, this is no longer the case when replacing estimates with predictions, as T is computed based on predictions, but jobs are not killed when their predicted termination time is reached; rather, they are simply assigned a bigger prediction.

For example, if J is predicted to run for 10 minutes and T happens to be 10 minutes away, then J will be backfilled, even if it was estimated to run for (say) three hours. Now, if our prediction turned out to be too short and J uses up its entire allowed three hours, the first queued job might be delayed by nearly 3 hours beyond its reservation.

Predictability is important for two main reasons. One is the support of moldable jobs. Such jobs may run on any partition size, and the scheduler is trusted to make the optimal decision on their behalf [5, 26, 4]. The consideration is simple: whether waiting for more nodes to become available is preferable or not over running immediately on what's available now. Predictability is of course crucial in this scenario. For example, a situation in which we decide to wait for (say) 30 minutes because it is predicted a hundred additional nodes will be available by then, only to find that the prediction was wrong, is highly undesirable. The second reason predictability is important is that it is needed to support advance reservations. These are used to determine which of the sites composing a grid is able to run a job at the earliest time [19], or to coordinate co-allocation in a grid environment [25], i.e. to cause cooperating applications to run at the same time on distinct machines.

The question is therefore which alternative (using estimates or predictions) yields more credible reservation times. To answer this question, we have characterized the distribution of the absolute difference between a reservation allocated to a job and its actual start time. This is only computed for a subset of the jobs: those that actually wait in the queue, become first, and get a reservation; jobs that are backfilled or started immediately don't have reservations, and are therefore excluded. A scheduler aspires to minimize both the number of jobs that need reservations and the differences

metric	trace	EASY	EASY+	EASY++
rate (% jobs)	SDSC-SP2	17	14 (−17%)	15 (−16%)
	CTC-SP2	7	5 (−19%)	6 (−16%)
	KTH-SP2	15	14 (−7%)	14 (−8%)
	SDSC-BLUE	12	10 (−16%)	10 (−14%)
average diff. [min]	SDSC-SP2	171	91 (−47%)	91 (−47%)
	CTC-SP2	51	29 (−43%)	27 (−46%)
	KTH-SP2	38	35 (−8%)	35 (−8%)
	SDSC-BLUE	65	43 (−34%)	43 (−34%)
median diff. [min]	SDSC-SP2	64	18 (−71%)	19 (−71%)
	CTC-SP2	8	2 (−73%)	2 (−78%)
	KTH-SP2	6	3 (−51%)	3 (−49%)
	SDSC-BLUE	16	4 (−77%)	4 (−75%)
standard deviation of diff. [min]	SDSC-SP2	471	175 (−63%)	173 (−63%)
	CTC-SP2	92	73 (−21%)	69 (−25%)
	KTH-SP2	84	88 (+5%)	88 (+5%)
	SDSC-BLUE	236	206 (−13%)	206 (−13%)

Table 6: *Effect of scheduler on the absolute difference between reservation time and actual start time. Rate is the percentage of jobs that wait and get a reservation. Both the rate and statistics of the distribution of differences are reduced when predictions are used, indicating improved performance and superior predictability, respectively.*

between their reservations and start times.

The predictor we have used in this section is slightly different from the one used in Section 3: instead of using the last two jobs to make a prediction, it uses the last two *similar* jobs, meaning that they had the same estimate (the effect of this change is discussed in Section 6). The results are shown in Table 6. Rate refers to the percentage of jobs that wait and get reservations. Evidently, this is consistently reduced (by 7–19%) when predictions are used, indicating more jobs enjoy backfilling and therefore reduced wait times. The remainder of the table characterize the associated distribution of absolute differences between reservations and start times. Both EASY+ and EASY++ obtain big reduction in the average differences: on SDSC-SP2, from almost 3 hours (171 minutes) to about an hour and a half (91 minutes). Reductions in median differences are even more pronounced, as these are at least halved across all traces, with a top improvement of 78% obtained by EASY++ on CTC. The variance of differences is typically also reduced, sometimes significantly (with the exception of a 5% increase for KTH). The bottom line is therefore that using

metric	trace	EASY	$X2$	SJF	EASY++
wait [min]	SDSC-SP2	363	333 (−8%)	566 (+56%)	326 (−10%)
	CTC-SP2	21	20 (−8%)	13 (−38%)	14 (−33%)
	KTH-SP2	114	102 (−11%)	79 (−31%)	94 (−18%)
	SDSC-BLUE	77	69 (−11%)	47 (−39%)	53 (−31%)
bounded slowdown	SDSC-SP2	99	89 (−10%)	71 (−28%)	72 (−28%)
	CTC-SP2	5	4 (−10%)	3 (−39%)	3 (−37%)
	KTH-SP2	90	80 (−11%)	45 (−50%)	57 (−37%)
	SDSC-BLUE	22	18 (−18%)	9 (−61%)	11 (−50%)

Table 7: *Average wait and bounded slowdown achieved by EASY++ compared with two other schedulers proposed in the literature: doubling user estimates and using SJF scheduling.*

runtime predictions consistently and significantly improves predictability of jobs' starting time.

5 Relationship With Other Algorithms

Our measurements so far have compared various scheduling schemes, culminating with EASY++, against vanilla EASY. However, other, more advanced backfilling schedulers have been proposed since the original EASY scheduler was introduced. In this respect, it is desirable to explore two aspects: comparing EASY++ against some other generic proposals, along with investigating the effect of directly applying our optimization techniques to the other schedulers themselves.

We have chosen to compare EASY++ against the two generic scheduling alternatives that were previously addressed in this paper: EASY with doubled user estimates (denoted $X2$), and SJF based on user estimates (as a representative of several different schemes that prioritize short jobs). The results are shown in Table 7. EASY++ outperforms $X2$ by a wide margin for all traces and both metrics. It is also rather close to SJF scheduling in all cases, and outperforms it in one case (SDSC-SP2's wait) where SJF fails for an unexplained reason. The advantage over SJF is, of course, the fact that EASY++ is fair, being based on FCFS scheduling with no danger of starvation. Also, the gap can potentially be reduced if better predictions are generated.

As mentioned earlier, EASY++ attempts to be similar to prevalent schedulers' default setting (usually EASY [6]) in order to increase its chances to replace them as the default configuration.

metric	trace	doubling			shortest job	
		$X2$	$X2$+	$X2$++	SJF	SJF+
wait	SDSC-SP2	333	357 (+7%)	332 (−0%)	566	430 (−24%)
[min]	CTC-SP2	20	16 (−18%)	15 (−25%)	13	12 (−11%)
	KTH-SP2	102	98 (−4%)	93 (−8%)	79	87 (+10%)
	SDSC-BLUE	69	60 (−13%)	52 (−25%)	47	43 (−9%)
bounded	SDSC-SP2	89	94 (+5%)	66 (−26%)	71	36 (−49%)
slowdown	CTC-SP2	4	4 (−13%)	3 (−28%)	3	2 (−12%)
	KTH-SP2	80	66 (−18%)	53 (−33%)	45	45 (−2%)
	SDSC-BLUE	18	16 (−13%)	12 (−35%)	9	8 (−9%)

Table 8: *Average performance and (parenthesized) improvement when optimizing vanilla $X2$ and SJF.*

But the techniques presented in this paper (as listed in Table 5) can be used to enhance *any* backfilling algorithm. Table 8 compares vanilla $X2$ and SJF to their corresponding optimized versions: In addition to doubling of estimates (recall that these serve as fallback predictions when there's not enough history), $X2$+ replaces estimates with (doubled) predictions, and employs prediction correction. $X2$++ adds SJBF to $X2$+. Finally, SJF+ is similar to EASY++, but allocates the reservation to the shortest (predicted) job, rather than to the one that has waited the most[3].

Table 8 shows that when switching from $X2$ to $X2$+, SDSC-SP2 exhibits inferior performance of 5-7%. Nevertheless, the common change in performance is actually an improvement of 13-18%. When further optimizing by adding SJBF ($X2$++), performance is consistently better, with a common improvement of 25-35%. Upgrading SJF to be SJF+ worsen performance in one occasion (+10% average wait in KTH-SP2), but results in a 2-49% improvement in all the other cases.

6 Tuning Parameters

Even the simplest backfilling algorithms have various tunable parameters, e.g. the number of reservations to make, the order of traversing the queue, etc. The EASY++ algorithm also has several selectable parameters, that may affect performance. We have identified ten parameters, some of which have only two optional values, but others have a wide spectrum of possibilities. To evaluate the effect of different settings, we simulated *all* possible combinations using our four different

[3]SJF+ and SJF++ are equivalent because both employ SJBF by definition.

workloads. This led to a total of about 220,000 simulations. This sections summarizes and presents the most important findings. In particular, only six of the parameters turned out to be significant, so the others are not mentioned.

6.1 The Prediction Window

Several parameters pertain to the prediction window — the set of previous jobs used to generate the prediction. One is the window size: how many jobs are used. In the previous sections we used 2, but maybe more would yield better results. Another is what to do when the window cannot be filled as not enough previous jobs have run. One option is to use a partial window, while another is to use the user estimate as a fallback. A third is the decision of exactly which jobs are included in the window. This can be immediate (only consecutive similar jobs, as used in Section 4), extended (similar jobs even if not consecutive), or all (consecutive jobs even if not similar, as used in Section 3). Similarity among jobs is defined as having the same estimate; it would make sense to also require the same executable, but this data is typically not available in the logs. In all cases, only jobs by the same user are considered.

Jobs included in window. Intuitively, we expect an immediate partial window to be best, as it uses the most recent data, and only similar jobs. Surprisingly, this turns out to be wrong, and in fact, the immediate window type is the worst, while all seems to be somewhat better than extended. This is demonstrated in Fig. 7. These graphs show a metric we call the *frequency factor*, which measures the success of each parameter setting (as will shortly be explained).

Recall that due to the full set of simulations we performed, there are very many simulations with each parameter value. But the number of runs with each parameter value is the same as the number of runs with other values of this parameter (e.g. with a boolean parameter, half the simulations are associated with “false” and the other half with “true”). Thus the frequency of each parameter value in the whole population of simulation results is the same. But if we sort the simulation results in order of a certain performance metric (e.g. average bounded slowdown), simulations with a given parameter value may be distributed in a non-uniform manner. In particular, we are interested in

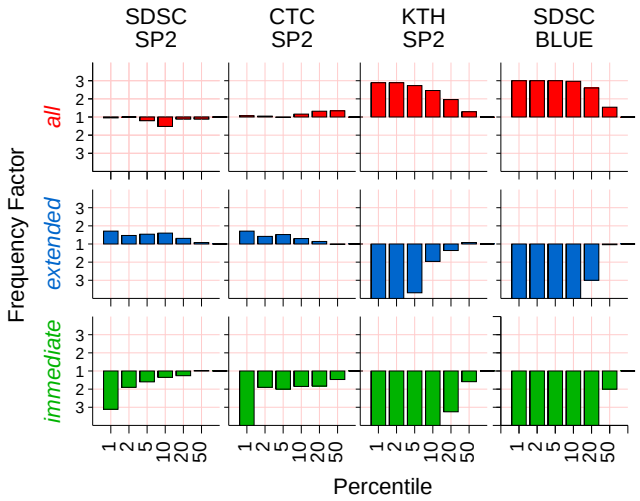

Figure 7: *Comparison of job selection schemes, with regards to the average bounded slowdown performance metric.*

situations where they are more common than they should be in the top percentile, top 2 percentiles, top 5 percentiles, etc.

The frequency-graphs show the factor by which the frequency of a parameter setting is higher or lower than expected, for different percentiles; a higher frequency is shown as a positive factor (above the axis), while a lower frequency is shown as negative (below the axis). For example, there are three job selection options (immediate/extended/all), so the frequency of each selection type in the sample space is $\frac{1}{3}$. However, we found that in the top 1 percentile of the configurations — as sorted by average bounded slowdown — only about $\frac{1}{9}$ of the samples have an immediate job selection. This is represented by a factor of 3 below the axis, meaning that this selection scheme is 3 times less common in that percentile than in the entire sample space, as depicted in the bottom left of Fig. 7. The slack is picked up by the extended scheme, which now appears in this percentile a fraction of about $\frac{1}{3} + \frac{2}{9}$ of the total, or about a factor of $1\frac{2}{3}$ more than it would if the distribution was uniform, as shown in the middle left of Fig. 7.

Examining the full set of data shown in the figure, we see that immediate job selection appears less than expected in all high percentiles for all four logs. The extended selection appears less than expected in only two logs, and slightly more than expected in the other two. But the all selection appears more than expected in two, and has a uniform distribution in the other two, so it

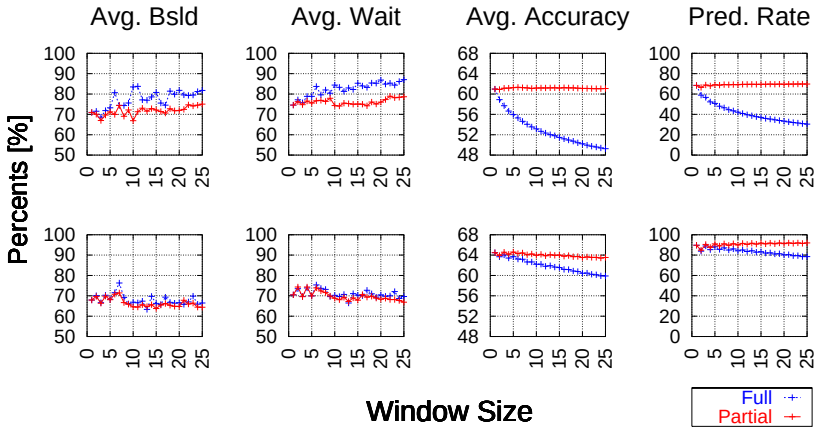

Figure 8: *The effect of window size and using full/partial on* immediate *windows (top) and* all *windows (bottom). Wait time and bounded slowdown are shown as percent of value for vanilla EASY. Data from simulations of the CTC log.*

is the best overall choice.

Fig. 7 only shows results for one metric: bounded slowdown. Results for wait time are similar. But there is also another metric to consider, namely accuracy. It turns out that this involves a trade-off: all windows are better for performance, while immediate windows are better for accuracy. However, in both cases, the degradation involved in using the other job selection method is not to high, in the range of about 0–15%.

A related issue is what to do with new data that becomes available. For example, if we make a prediction for a newly submitted job, and later a previous job terminates, should we update the prediction based on this new information? Our results indicate that this is indeed beneficial (not shown), but the performance gain is not very big. This is in keeping with the result that selecting all jobs is best, as it utilizes the most recent data.

Window size. Intuitively one might expect larger windows to lead to better performance, as more data is used. But as Fig. 8 shows, the average performance metrics (bounded slowdown and wait time) are largely oblivious to window size. In reality, the results are more complex and show that the effects of window size on performance — which do exists in some configurations — are very trace dependent. Nevertheless, we have found that small window sizes generally yield good performance.

In contrast to its marginal effect on performance, the window size does have a strong effect on the prediction rate and accuracy, at least when full windows are used (i.e. when we require jobs to fill the window size in order to make a prediction). This happens because in many cases sufficient data is not available, and the user estimate has to be used as a fallback. Using partial windows avoids this problem, apparently at no cost in accuracy or performance. "Partial" means that a prediction is made based on whatever data is available, and in particular, also if the window is not full.

These results regarding prediction accuracy also explain the advantage of all windows quoted above. This approach is less finicky and uses all available data. As a result it fills its window faster, and avoids having to rely on inaccurate user estimates. This in turn facilitates the generation of better predictions, and through them, the achievement of better performance results.

6.2 The Prediction Algorithm

Another issue is how to calculate the prediction given the data for the jobs in the window. Four options are to use the average, median, minimum, or maximum of the runtimes of these jobs. The results are shown in Fig. 9. Using the maximum is obviously bad, but the other three options each have some good cases and some bad ones. Nevertheless, it is possible to discern that using the median has a certain advantage — it fails to make the top 20 percentiles only for the wait time metric using the CTC trace.

Prediction fallback. In some configurations the scheduler cannot generate a prediction at all, because the required data does not exist. Examples are when a full window is required and there are insufficient jobs, or when an immediate job selection is desired, but the previous jobs were dissimilar. In such cases, the predictor has to fallback on the user estimate.

One option is to use the estimate as is. The other is to use a "relative" estimate, that is to scale it according to the accuracy the user had displayed previously [24]. Our results indicate that relative estimates are much better (not shown). But even with this improvement, immediate

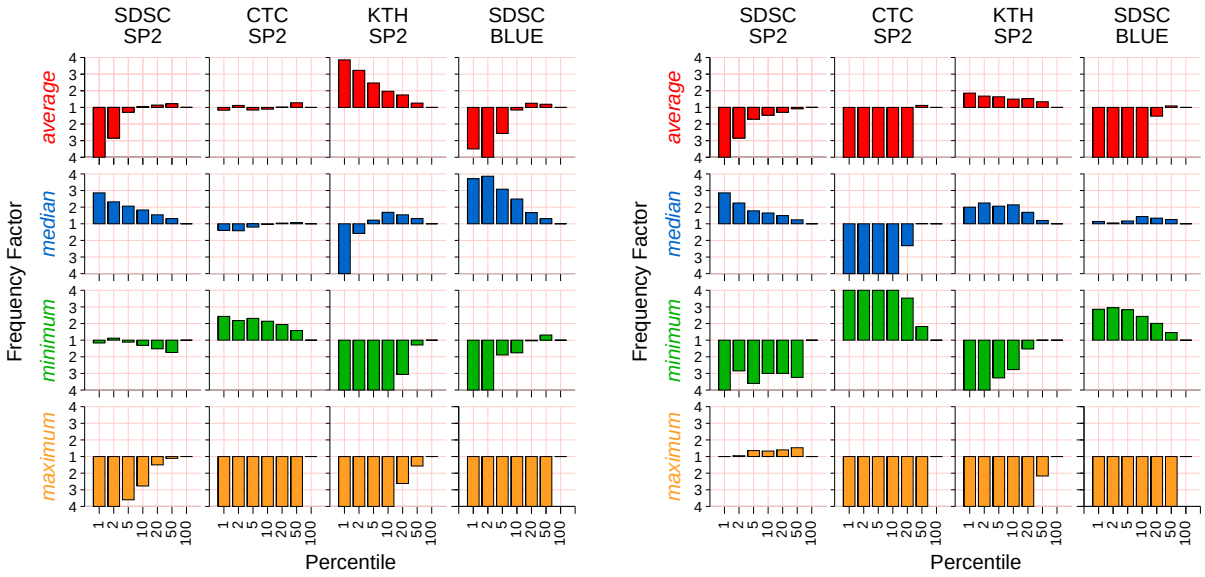

Figure 9: *Effect of different prediction methods on performance. Left: bounded slowdown. Right: wait time.*

windows with relative estimates are still not as good as all windows. Note that partial all windows practically never need the fallback, as they always have previous data except for the very first jobs.

6.3 Global and Local Optima

As seen in the above graphs, it is often the case that one parameter value is good for one trace and metric, while another is better for a different configuration. This leads to the question whether a global optimum exists that would be good for all situations.

The answer seems to be that there is no single global optimum: for each trace and metric, a different configuration would be optimal. Moreover, the gap between the best overall configuration and the best configuration for a specific situation can reach 18%. However, even using predictions with suboptimal parameter values is still significantly better than the original EASY algorithm in terms of both performance metrics (wait and slowdown), accuracy, and predictability. The bottom line is therefore that a default setting can be found that would provide substantial performance benefits, but that even more benefits will be possible if the configuration is tuned to the local workload.

parameter	description	default
window size	how many history jobs to consider	2
job selection	which jobs to include in the window	all
fullness	whether to require a full window to make a prediction	partial
algorithm	how to generate predictions	median
propagation	whether to use new data to update predictions	yes
fallback	how to use user estimates when needed	relative

Table 9: *Suggested default parameter settings for the scheduling with predictions algorithm.*

Summarizing the results presented in the previous subsection, our proposal for a default setting is given in Table 9.

7 Conclusions and Future Work

Backfilling has been studied extensively in the last few years. One of the most surprising results was the inability to improve upon the inaccurate user estimates of runtime. On one hand, better runtime predictions could be generated, but not without substantial risk of underestimation. On the other hand, several papers reported on improved performance when the estimates were artificially made even less accurate.

We have shown that accurate predictions can indeed be incorporated into a backfilling scheduler, and that doing this correctly leads to substantial benefits. The solution is composed of three parts:

1. The accurate predictions are only used to make scheduling decisions, while the original user estimates retain their role in determining when jobs overrun their time and should be killed. This eliminates the unacceptable killing of jobs due to underestimation.
2. When a job exceeds its prediction, the prediction needs to be corrected to reflect this new reality. This enables the scheduler to continue and optimize under a truthful view of the state of the machine.
3. While the order of making reservations remains FCFS, the order of backfilling is SJF. This leads to an additional performance improvement, and is a direct and explicable way of out-

performing improvements that have so far been achieved by doubling or randomizing user estimates (without explaining why this helps).

We applied these improvements to the EASY scheduler, but they can be applied equally well to any other backfilling scheduler. The reason we have chosen to focus on EASY is its proven continuous popularity over the past decade, which may be attributed to the fact it maintains conservative FCFS semantics, while achieving better utilization and performance. Since our improved scheduler (called EASY++) essentially preserves these qualities, but consistently outperforms its predecessor in terms of accuracy, predictability, and performance (up to doubled), we believe it has an honest chance to replace EASY as the default configuration of production systems. Moreover, EASY++ is fairly easy to implement.

The main results reported in this study employed a very rudimentary predictor, using the previous two jobs submitted by the user. We have quantified the accuracy of these predictions (including the case when they change during a job's execution) and show that they are much better than the original user estimates. We also checked the performance that would be obtained with perfect predictions, and found that it is much better yet. This motivates the search for better predictors.

In future work we intend to check the results obtained by previously proposed predictors [12, 24] when incorporated in EASY++, and see whether they come closer to PERFECT++ or are comparable to our simplistic predictor. We also intend to develop completely new predictors. A promising new direction we are currently checking is predictions based on user-sessions, that is, dynamically identifying time limited work session of users (a consecutive period of time in which a user continuously submit jobs) and assigning a prediction per session.

Acknowledgments

This research was supported in part by the Israel Science Foundation (grant no. 167/03). Many thanks are due to the people and organizations who deposited their workload logs in the Parallel Workloads Archive, and made this research possible.

References

[1] "*Parallel workloads archive*". URL http://www.cs.huji.ac.il/labs/parallel/workload/.

[2] S-H. Chiang, A. Arpaci-Dusseau, and M. K. Vernon, "*The impact of more accurate requested runtimes on production job scheduling performance*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 103–127, Springer Verlag, 2002. Lect. Notes Comput. Sci. vol. 2537.

[3] S-H. Chiang and M. K. Vernon, "*Production job scheduling for parallel shared memory systems*". In 15th *Intl. Parallel & Distributed Processing Symp.*, Apr 2001.

[4] W. Cirne and F. Berman, "*Using moldability to improve the performance of supercomputer jobs*". *J. Parallel & Distributed Comput.* **62(10)**, pp. 1571–1601, Oct 2002.

[5] A. B. Downey, "*Predicting queue times on space-sharing parallel computers*". In 11th *Intl. Parallel Processing Symp.*, pp. 209–218, Apr 1997.

[6] Y. Etsion and D. Tsafrir, *A Short Survey of Commercial Cluster Batch Schedulers*. Technical Report 2005-??, Hebrew University, May 2005.

[7] D. G. Feitelson, "*Experimental analysis of the root causes of performance evaluation results: a backfilling case study*". *IEEE Trans. Parallel & Distributed Syst.* **16(2)**, pp. 175–182, Feb 2005.

[8] D. G. Feitelson and A. Mu'alem Weil, "*Utilization and predictability in scheduling the IBM SP2 with backfilling*". In 12th *Intl. Parallel Processing Symp.*, pp. 542–546, Apr 1998.

[9] D. G. Feitelson and B. Nitzberg, "*Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 337–360, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[10] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, "*Parallel job scheduling — a status report*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), Springer Verlag, 2004.

[11] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong, "*Theory and practice in parallel job scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 1–34, Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[12] R. Gibbons, "*A historical application profiler for use by parallel schedulers*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 58–77, Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[13] D. Jackson, Q. Snell, and M. Clement, "*Core algorithms of the Maui scheduler*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 87–102, Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.

[14] R. Jain, *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991.

[15] J. P. Jones and B. Nitzberg, "*Scheduling for parallel supercomputing: a historical perspective of achievable utilization*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 1–16, Springer-Verlag, 1999. Lect. Notes Comput. Sci. vol. 1659.

[16] S. Kannan, M. Roberts, P. Mayes, D. Brelsford, and J. F. Skovira, *Workload Management with LoadLeveler*. IBM, first ed., Nov 2001. ibm.com/redbooks.

[17] P. J. Keleher, D. Zotkin, and D. Perkovic, "*Attacking the bottlenecks of backfilling schedulers*". *Cluster Comput.* **3(4)**, pp. 255–263, 2000.

[18] C. B. Lee, Y. schwartzman, J. Hardy, and A. Snavely, "*Are user runtime estimates inherently inaccurate?*". In *Job Scheduling Strategies for Parallel Processing*, Springer-Verlag, 2004.

[19] H. Li, D. Groep, and J. T. L. Wolters, "*Predicting job start times on clusters*". In *International Symposium on Cluster Computing and the Grid (CCGrid)*, 2004.

[20] D. Lifka, "*The ANL/IBM SP scheduling system*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 295–303, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[21] A. W. Mu'alem and D. G. Feitelson, "*Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling*". *IEEE Trans. Parallel & Distributed Syst.* **12(6)**, pp. 529–543, Jun 2001.

[22] D. Perkovic and P. J. Keleher, "*Randomization, speculation, and adaptation in batch schedulers*". In *Supercomputing*, p. 7, Sep 2000.

[23] E. Shmueli and D. G. Feitelson, "*Backfilling with lookahead to optimize the performance of parallel job scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 228–251, Springer-Verlag, 2003. Lect. Notes Comput. Sci. vol. 2862.

[24] W. Smith, I. Foster, and V. Taylor, "*Predicting application run times using historical information*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 122–142, Springer Verlag, 1998. Lect. Notes Comput. Sci. vol. 1459.

[25] W. Smith, I. Foster, and V. Taylor, "*Scheduling with advanced reservations*". In 14th *Intl. Parallel & Distributed Processing Symp.*, pp. 127–132, May 2000.

[26] W. Smith, V. Taylor, and I. Foster, "*Using run-time predictions to estimate queue wait times and improve scheduler performance*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 202–219, Springer Verlag, 1999. Lect. Notes Comput. Sci. vol. 1659.

[27] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan, "*Selective reservation strategies for backfill job scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn (eds.), pp. 55–71, Springer-Verlag, 2002. Lect. Notes Comput. Sci. vol. 2537.

[28] S. Srinivasan, R. Kettimuthu, V. Subrarnani, and P. Sadayappan, "*Characterization of backfilling strategies for parallel job scheduling*". In *Intl. Conf. Parallel Processing*, pp. 514–522, Aug 2002.

[29] D. Tsafrir, "*The dynamics of backfilling*". (in preparation).

[30] D. Tsafrir and D. G. Feitelson, *Workload Flurries*. Technical Report 2003-85, Hebrew University, Nov 2003.

[31] D. Zotkin and P. J. Keleher, "*Job-length estimation and performance in backfilling schedulers*". In 8th *Intl. Symp. High Performance Distributed Comput.*, Aug 1999.