Backfilling Using Runtime Predictions Rather Than User Estimates

Dan Tsafrir Yoav Etsion Dror G. Feitelson
The Hebrew University of Jerusalem, Israel

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 system-generated predictions based on history may be significantly better. However, predictions have not been incorporated into production schedulers, partially due to a misconception (that we resolve) claiming inaccuracy actually improves performance, and partly because underprediction is unacceptable: 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. Notably, this is based on an extremely simple predictor, that just averages the runtimes of the last two jobs by the same user; counterintuitively, our results indicate that using recent data is more important than mining the history for similar jobs. All these techniques can be used to enhance any backfilling algorithm, and are not limited to EASY.

1 Introduction

Backfilling. 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-

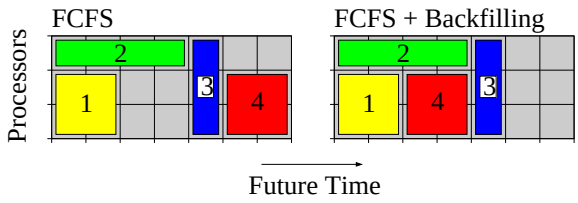

Figure 1: *EASY backfilling reduces fragmentation. It would have been impossible to backfill job 4 had its length been more than 2, as the reservation for job 3 would have been violated.*

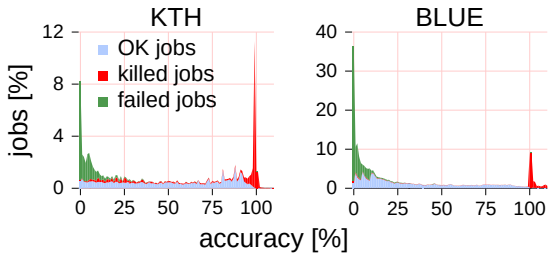

Figure 2: *The accuracy histogram is rather flat when ignoring jobs that reached their estimate and were killed by the system (100% peak) or that failed on startup (0% hump).*

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.

A potential problem with this is that the first queued job may be starved as subsequent jobs continually jump over it. The solution is making a *reservation* for this job, and allowing subsequent jobs to run only if they respect it (Fig. 1). This approach was originally introduced by EASY, the first backfilling scheduler [24]. Many backfilling variants have been suggested since, e.g. using more reservations, employing a non-FCFS wait queue order, etc. [10]. However, the default of most parallel schedulers (e.g. Maui/Moab [15] and IBM's load-leveler [18]) has remained plain EASY [6], and it has been estimated that 90-95% of the installations do not change this default configuration [14]. Indeed, while simple, backfilling dramatically improves utilization [17] and yields comparable performance to that of more sophisticated algorithms [36].

User Runtime Estimates. Backfilling requires the runtime of jobs to be known: both when computing the reservation (requires knowing when processors of currently running jobs will become available) and when determining if waiting jobs are eligible for backfilling (must terminate before the reservation). Therefore, EASY required users to provide a runtime estimate for all submitted jobs [24], and the practice continues to this day. Jobs that exceed their estimates are killed, so as not to violate subsequent commitments. The assumption is that users would be motivated to pro-

vide accurate estimates, because (1) jobs would have a better chance to backfill if their estimates are tight, but (2) would be killed if they are too short.

Nevertheless, empirical studies of traces from sites that actually use EASY show that user estimates are generally inaccurate [25]. This is exemplified in Fig. 2 showing a typical accuracy $(= 100 \cdot \frac{runtime}{estimate})$ histogram: when only considering jobs that have terminated successfully we get a rather uniform-like distribution, meaning any level of accuracy is almost equally likely to happen. 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 information [22].

Estimates also embody a characteristic that is particularly harmful for backfilling: they are inherently modal, as users tend to choose "round" estimates (e.g. one hour) resulting in 90% of the jobs using the same 20 values [34]. This modality limits the scheduler's ability to exploit existing holes in the schedule because all jobs look the same. Both inaccuracy and modality deteriorate performance (Fig. 3; compare "orig" to "perfect") and motivate searching for an alternative.

The Alternative. The search for better estimates has focused on using historical data. As users of parallel machines tend to repeatedly do the same work [9], it's conceivable historical data can be used to predict the future (Fig. 4). Suggested prediction schemes include using the top of a 95% confidence interval of job runtimes [12], a statistical model based on the (usually) log-uniform distribution of runtimes [5], using the mean plus 1.5 standard deviations [25], and several other techniques [29, 19, 20]. Despite all this work, backfill schedulers in actual use still employ user estimates rather than history-based system-generated predictions, due to three difficulties: (1) a technicality, (2) usability issues, and (3) misconceptions, to be described in detail next. *This paper is about refuting or dealing with these difficulties*.

Technicality. The core problem is that it's simply impossible to naively replace estimates with system predictions, as these might turn out too short leading to premature killing of jobs according to the backfilling rules. Suggested solutions have included simply ignoring it, using preemption,

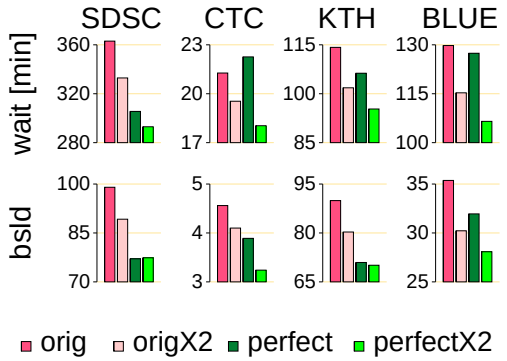

Figure 3: *Average wait-time and bounded slowdown of jobs improve when user estimates ("orig") are replaced by real runtimes ("perfect"). Doubling helps both original estimates and perfect ones.*

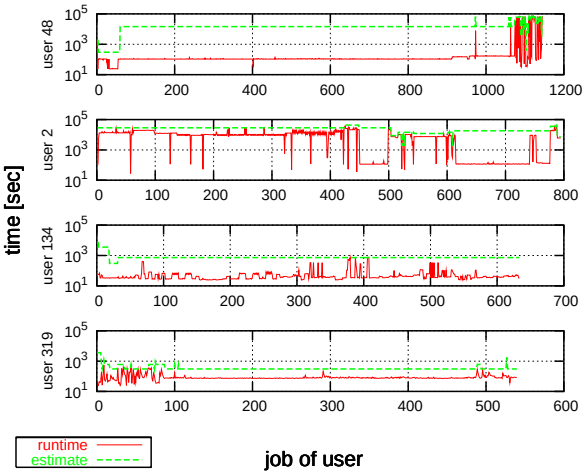

Figure 4: *Runtime and estimate of jobs by four arbitrary SDSC users show remarkable repetitiveness.*

employing test runs, or replacing backfilling by shortest job first (SJF) [12, 37, 2, 21].[1] None of these retain the appeal of plain EASY. Mu'alem and Feitelson checked the extent of the underprediction phenomenon, showed it to be significant (20% of the jobs), and concluded that "it seems using system-generated predictions for backfilling is not a feasible approach" [25]. However, as we will show, solving this problem is actually quite simple: user estimates must serve as kill times (part of the user contract), while system predictions can be used for everything else.

Usability. Previous prediction techniques have assumed that an important component is to identify the most similar jobs in the history, and base the predictions on them. To this end they employed genetic algorithms [29, 31], instance based learning [19], and rough set theory [20]. Unfortunately, these are all much more complex than the EASY scheduler itself, and more importantly, they require a training period which can be significant. For example, Smith et al. [29, 31] used an entire trace to guide the selection of job templates, before evaluating their algorithm (on the very same trace, using the selected templates). While in principle it's possible to use such algorithms on-line (repeatedly training on what's available), the success of this approach along with the overheads it entails remains to be evaluated.

In contrast, we show that even extremely trivial algorithms (e.g. using the average runtime of

[1] Smith et al. didn't specify how they utilized system predictions for backfilling [31].

two preceding jobs by the same user) result in significant improvement, both in the accuracy of the prediction itself and in the resulting performance. We chose such a simple predictor in order to focus on how predictions are integrated into backfilling schedulers, and not on the prediction algorithm itself. However, our evaluations indicate that this was a fortuitous choice, and that recency is actually more important than similarity when using historical data.

Misconceptions. Surprisingly, studies regarding the impact of inaccuracy have found that it actually improves performance [8]. This has even led to the suggestion that estimates should be *doubled* [37, 25] or *randomized* [27], to make them even less accurate. Doubling indeed exhibits remarkable improvements (Fig. 3), which seemingly negates the motivation to incorporate mechanisms for better predictions, deeming user estimates as "unimportant". We solve this mystery and show the "inaccuracy helps" myth is actually false in three respects.

First, doubling original user estimates indeed helps, but even more so if applied to perfect estimates (Fig. 3; compare "origX2" to "perfectX2"). We show that doubling of good predictions is similar: the more accurate original predictions are, the more the doubling is effective.

Second, we show that the reason doubling helps is because it allows shorter jobs to move forward within an FCFS setting, implicitly approximating an SJF-like schedule. (Indeed, most studies dealing with predictions indicate that increased accuracy improves performance when shorter jobs are favored [12, 31, 37, 27, 2]). This is obtained by gradually pushing away the start time of the first queued job, in a kind of "heel and toe" dynamics that effectively trades off FCFS-fairness for performance. A main contribution of this paper is showing this tradeoff can be avoided by explicitly using a *shortest job backfilled first* (SJBF) backfilling order. By still preserving FCFS *reservation-order*, we 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 underlying implied assumption that predictions are only important for performance. In fact, they are also important for various functions. One example is advance reservations for grid allocation and co-allocation, shown to considerably benefit from better accuracy [19, 30, 23]. Another is scheduling *moldable* jobs that may run on

Abbreviation	Site	CPUs	Jobs	Start	End	Util	Avg. Runtime [min]
CTC	Cornell Theory Center	512	77,222	Jun 96	May 97	56%	123
KTH	Swedish Royal Instit. of Tech.	100	28,490	Sep 96	Aug 97	69%	188
SDSC	San-Diego Supercomputer Center	128	59,725	Apr 98	Apr 00	84%	148
BLUE	San-Diego Supercomputer Center	1,152	243,314	Apr 00	Jun 03	76%	73

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).*

any number of nodes [5, 31, 3]. The scheduler's goal is to minimize response time, considering whether waiting 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.

Roadmap. This rest of the paper is structured thus. After describing our methodology (Sec. 2), we explain how prediction-based backfilling is done and demonstrate the improvements (Sec. 3–4). We show the generality of our techniques (Sec. 5), face the above misconceptions (Sec. 6), investigate the optimal parameter settings for our algorithms (Sec. 7), and conclude (Sec. 8).

2 Methodology

The experiments are based on an event-based simulation of EASY scheduling, where events are arrivals and terminations. Upon arrival, the scheduler is informed of the number of processors 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 the 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 are using their "cleaned" versions [26, 11, 35]. Since the traces span the past decade, were generated at different sites, on 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 average wait time and bounded slowdown. The wait

period is between a job's submittal and its start time.[2] Slowdown is response time (wait- plus run-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 yielding the formula $\max\left(1\ ,\ \frac{T_w+T_r}{\max(10,T_r)}\right)$. To reduce warmup effects, the first 1% of terminated jobs were not included in the metric averages; to reduce cooldown effects, jobs that terminated after the last arrival were also not included in the metric averages [16].

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 the more accurate the prediction. This is 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. More formally, given a job J, its weighted accuracy is $\sum_{i=1}^{N} A_i \cdot \left(\frac{T_i - T_{i-1}}{T_N - T_0}\right)$ where T_0 and T_N are J's submission and termination time, respectively, and A_i is the accuracy of j's prediction that was in effect from time T_{i-1} until time T_i.

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.[3] 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

[2]For batch scheduling, the average wait time differs from the average response time by the average runtime, which is a *constant*: $\frac{1}{n}\sum(T_w + T_r) = \frac{1}{n}\sum T_w + \frac{1}{n}\sum T_r$, where T_w and T_r are jobs' wait and running times, respectively. Using average wait focuses on the scheduling activity, neutralizing the highly variable average runtime (Table. 1).

[3]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.

being killed just because the system speculated they were shorter than the user estimate. So it is not advisable to just replace estimates by predictions. Previous studies have dealt with this difficulty either by: eliminating the need for backfilling (e.g. using pure SJF [12, 31]), employing test runs [2, 21], assuming preemption is available (so jobs that exceed their prediction can be stopped and reinserted into the wait queue [12]), or considering only artificial estimates generated as multiples of actual runtimes (effectively assuming underprediction never occurs) [37, 27, 2, 32, 33].

3.1 Separating the Dual Roles of Estimates

The key idea of our solution is recognizing that the underprediction 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 legitimate to kill a job *once its user estimate is reached*, but not any sooner; therefore the main function of estimates is to serve as kill-times. All other scheduling considerations should be based upon *the best available predictions* of how long jobs will run; this can be the user estimate, but it can also be generated by the system, and moreover, it can change over time.

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 terminated. If no such jobs exist we fall back on the associated user estimate (other ways to select the history jobs are considered in Sec. 7). If a prediction turns out higher than the job's estimate it is discarded, and the estimate is used, because the job would be killed anyway when it reached its estimate. Implementing this predictor is obviously trivial. Nevertheless, as shown below, this simple predictor is enough to significantly improve the accuracy of the data used by the scheduler, which is sufficient for our needs in this paper. Investigation of the effect of better predictors is left for future work.

3.2 Prediction Correction

A reservation computed based on user estimates will never be smaller than the start time of the associated job, as estimates are runtime upper bounds. This is no longer true for predictions,

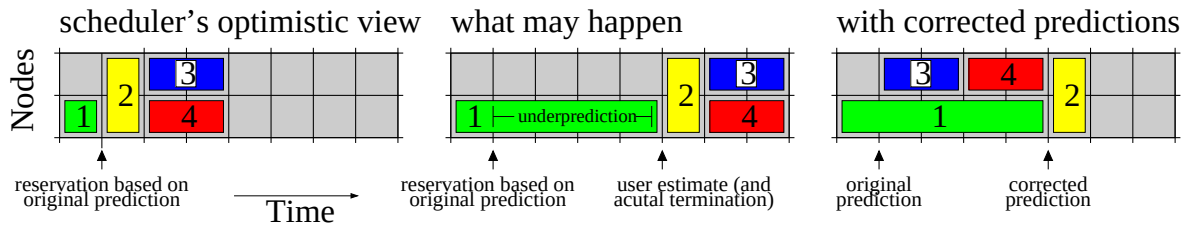

Figure 5: *Underpredicting the runtime of job 1 causes the scheduler to make an early reservation for job 2 (left). This misconception prevents jobs 3 and 4 from being backfilled (middle). Correcting the prediction once proved wrong, enables the scheduler to reschedule the reservation and re-enables backfilling (right).*

as they are occasionally too short. At the extreme, predictions might erroneously indicate that certain jobs should have terminated by now and thus their processors should be already available. Assuming there aren't enough processors for the first queued job J, this discrepancy might lead to a situation where J's reservation is made for the present time, because the scheduler erroneously thinks the required processors should already be available.

Note that the backfilling window is between the current time (lower bound) and the reservation (upper). When these are made equal, backfill activity effectively stops[4] and the scheduler largely reverts to plain FCFS, eliminating the potential benefits of backfilling (Fig. 5, left and middle).

The solution is to modify the scheduler to increase expired predictions proven to be too short. For example, if a job's prediction indicated it would run for 10 minutes, and this time has already passed but the job is still running, we must generate a new prediction. The simplest approach is to acknowledge that the user was smarter than us and set the new prediction to be the user's estimate. Once the prediction is updated, this affects reservations for queued jobs and re-enables backfilling (Fig. 5, right). While this may undesirably delay the reservations made for queued jobs, such delays are still bounded by the original runtime estimates of the running (underpredicted) jobs.

On rare occasions prediction correction is necessary even beyond the estimate, as in real systems jobs sometimes exceed their estimates (Fig. 6, left). In most cases the overshoot is very short (not more than a minute) and probably reflects the time needed to kill the job. But in some cases it is much longer, for unknown reasons. Regardless of the exact reason, the prediction should be

[4]The only remaining backfill activity is on the expense of the "extra" processors, which are the "leftover" after satisfying the reservation for the first queued job [24, 25].

trace	underestimated jobs	
	number	%
SDSC	4,138	7.7%
CTC	7,174	9.3%
KTH	478	1.7%
BLUE	22,216	9.9%

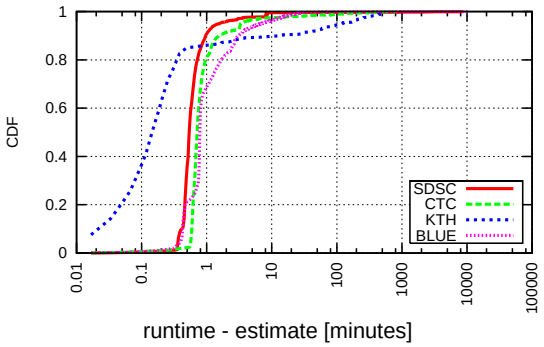

Figure 6: *Left: up to 10% of the jobs have runtimes bigger than user estimates. Right: CDF of differences between runtime and estimate, of underestimated jobs. Most estimate violations are less than one minute.*

extended to keep the scheduler up to date (independent of the fact the job should be killed, and maybe *is* being killed). As most of these jobs only exceed their estimate by a short time, we enlarge post-estimate predictions in a gradual manner: The first adjustment adds only one minute to the old prediction. This will cover the majority of the underestimated jobs (Fig. 6, right). If this is not enough, the i^{th} prediction correction adds $15 \times 2^{i-2}$ minutes (15min, 30min, 1h, 2h, etc.).

The results of adding prediction correction are shown in Table 2. This compares the original EASY with a version that uses user estimates as predictions and adds prediction correction (EASY$_{PCOR}$), and a version that combines prediction correction with system-generated predictions (EASY$^+$). Note that while EASY$_{PCOR}$ employs user estimates as predictions, correction is still needed to handle the underestimated jobs discussed earlier. Prediction-correction by itself has only a marginal effect, because only a small fraction of the jobs are grossly underestimated. The real value of prediction correction is revealed in EASY$^+$, where system-generated predictions are added: results show a significant and consistent improvement of up to 28% (KTH's slowdown in Table 2). This is an important result that shouldn't be taken lightly. The fact that historical information can be successfully used to generate runtime predictions is known for more than a decade [9]. Our results in Table 2 demonstrate for the first time that this may be put to productive use within backfilling schedulers, without violating the contract with users. Moreover, the overhead is low, with predictions corrected only 0.56–0.63 times on average per job.

Note that obtaining the reported improvement is almost free. All one has to do is create pre-

trace	wait [minutes]					b. slowdown					accuracy [%]					avg. corr. [$\pm\sigma$]	
	$EASY$	$EASY$ $PCOR$		$EASY^+$ all		$EASY$	$EASY$ $PCOR$		$EASY^+$		$EASY$	$EASY$ $PCOR$		$EASY^+$		$EASY$ $PCOR$	$EASY^+$
SDSC	363	360	-1%	326	-10%	99	93	-6%	86	-13%	32	32	+0%	60	+87%	$0.09_{\pm 0.33}$	$0.56_{\pm 0.64}$
CTC	21	21	+0%	16	-26%	4.6	4.5	-2%	3.3	-27%	39	39	+0%	62	+61%	$0.12_{\pm 0.41}$	$0.63_{\pm 0.69}$
KTH	114	115	+1%	96	-16%	90	90	+1%	65	-28%	47	47	+0%	60	+28%	$0.02_{\pm 0.24}$	$0.53_{\pm 0.57}$
BLUE	130	128	-1%	102	-21%	35	36	+1%	26	-25%	31	31	+0%	61	+100%	$0.13_{\pm 0.48}$	$0.60_{\pm 0.69}$
avg.			-0%		-18%			-2%		-23%			+0%		+69%	0.09	0.58

Table 2: *Average performance, accuracy, and overhead for scheduler variants. $EASY_{PCOR}$ adds prediction correction, and $EASY^+$ also adds system generated predictions. Shaded columns give changes relative to EASY in percents; negative values are good for wait time and slowdown, while positive ones are good for accuracy. Right most metric shows the per-job average prediction-correction number ($\pm$ std. deviation).*

dictions as the average runtime of the user's two most recent 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, note that this scheme significantly improves the average accuracy, which can be up to doubled (BLUE) and is stabilized at 60–62% across all four traces when using EASY$^+$.

3.3 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. Moreover, several studies have demonstrated that the benefit of accuracy dramatically increases if shorter jobs are favored [12, 31, 37, 27, 2, 28]. 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.[5]

Contemporary schedulers such as Maui can be configured to favor (estimated) short jobs, but their default configuration is essentially the same as in EASY [6] (SJF is the default only in PBS). This 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

[5]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.

trace	wait [minutes]							b. slowdown							accuracy [%]						
	$EASY$	$EASY_{SJBF}$		$EASY^{++}$		$PERFECT^{++}$		$EASY$	$EASY_{SJBF}$		$EASY^{++}$		$PERFECT^{++}$		$EASY$	$EASY_{SJBF}$		$EASY^{++}$		$PERFECT^{++}$	
SDSC	363	361	-0%	327	-10%	278	-23%	99	87	-12%	70	-29%	58	-42%	32	32	+0%	60	+87%	100	+211%
CTC	21	19	-10%	14	-33%	19	-10%	4.6	3.9	-14%	2.9	-37%	2.8	-39%	39	39	+0%	62	+61%	100	+158%
KTH	114	102	-11%	95	-17%	91	-20%	90	73	-19%	57	-36%	50	-44%	47	47	+0%	61	+28%	100	+111%
BLUE	130	102	-21%	87	-33%	87	-33%	35	21	-42%	19	-47%	13	-64%	31	31	+0%	62	+102%	100	+225%
avg.			-10%		-23%		-22%			-22%		-37%		-47%			+0%		+70%		+176%

Table 3: *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. Shaded columns show improvement relative to traditional EASY.*

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 oldest job in the wait queue*. In contrast, backfilling is done 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 [37, 25] or randomizing [27] estimates (see Sec. 6).

Results of applying SJBF are shown in Table 3. 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 10–20%, and up to 42% (BLUE's bounded slowdown).

Much more interesting is $EASY^{++}$ which adds SJBF to $EASY^{+}$ (namely combines system-generated predictions, prediction correction, 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 (nearly halved in BLUE). There is also a 33% peak improvement in average wait (CTC and BLUE). This is quite impressive for a scheduler with basic FCFS semantics that differs from EASY by only a few dozens lines of code. Even more impressive is the *consistency* of the 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

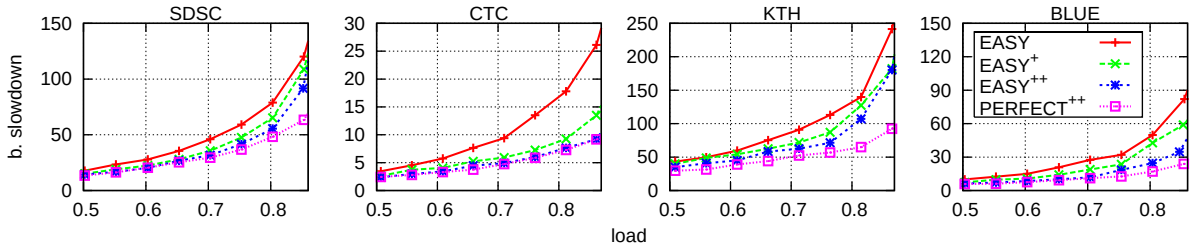

Figure 7: *EASY^{++}'s relative performance typically improves with load.*

used [31, 7]. The accuracy of EASY^{++} is similar to that of EASY^{+} at 60–62%.

Finally, we have also checked the impact of having perfect predictions when SJBF is employed (here there is no meaning to prediction correction as predictions are always correct). It turns out PERFECT^{++} is marginally to significantly better than EASY^{++} with the difference being most pronounced in SDSC, the site with the highest load (Table 1; further discussed below). Interestingly, EASY^{++} outperforms PERFECT^{++} in CTC's average wait. This is due to subtle backfill issues and a fundamental difference between CTC and the other logs, as analyzed by Feitelson [7].

3.4 Varying the Load

All results in this paper evaluate our suggested optimizations using four workloads "as is". Here, through trace manipulation, we complement our measurements by investigating the effect of load. Load is artificially varied by multiplying all arrival times by a constant (e.g. if BLUE's original load is 76%, we can raise it to 80% by multiplying all arrival times by $\frac{76}{80}$). Results show that PERFECT^{++} is better than EASY^{++}, which is better than EASY^{+}, which is better than EASY (Fig. 7). Higher loads usually intensify the trends pointed out earlier, but the precise effect of the optimizations is very workload dependent. EASY^{++} benefits are relatively small in SDSC, especially under high loads; for KTH the biggest improvement occurs for intermediate loads of around 70–80%; for CTC, the improvement over EASY grows with load, and becomes extremely significant at 90%. Examining PERFECT^{++}, we see that in all cases the role of accuracy becomes crucial as load conditions increase, generating a strong incentive for developing better prediction schemes.

4 Predictability

Previous sections have shown that, on average, replacing user estimates with system-generated predictions is beneficial in terms of both performance and accuracy. However, when abandoning estimates in favor of predictions, we might lose *predictability*: The original backfilling rules state that a job J_b can be backfilled if its estimated termination time does not violate the reservation time R_1 of the first queued job J_1. Since J_b is killed when reaching its estimate, it is guaranteed that J_1 will indeed be started no later than R_1. However, this is no longer the case when replacing estimates with predictions, as R_1 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_b is predicted to run for 10 minutes and R_1 happens to be 10 minutes away, then J_b 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_b uses up its entire allowed three hours, J_1 might be delayed by nearly 3 hours beyond its reservation.

Predictability is important for two main reasons. One is the support of *moldable* jobs [5, 31, 3], that may run on any partition size (according to [4], $\backsim$98% of the jobs are moldable). Such jobs trust the scheduler to decide whether waiting for more nodes to become available is preferable over running immediately on what's available now. Predictability is crucial for such jobs. 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 [23], or to coordinate co-allocation in a grid environment [19, 30], i.e. to cause cooperating applications to run at the same time on distinct machines. Note that in this case underprediction is as bad as overprediction, e.g. for a grid broker that must select where to dispatch a job. Knowing that resources would become available earlier could shift the ballance.

The question is therefore which alternative (using estimates or predictions) yields more credible reservation times. To answer it, we have characterized the distribution of the absolute difference

trace	rate [% of jobs]					avg. diff. [minutes]					median diff. [minutes]					stddev diff. [minutes]				
	$EASY$	$EASY^+$		$EASY^{++}$		$EASY$	$EASY^+$		$EASY^{++}$		$EASY$	$EASY^+$		$EASY^{++}$		$EASY$	$EASY^+$		$EASY^{++}$	
SDSC	17	14	-18%	15	-15%	171	93	-46%	91	-47%	64	20	-69%	19	-70%	471	174	-63%	168	-64%
CTC	6.8	5.4	-19%	5.7	-16%	51	29	-43%	27	-46%	8.3	2.2	-73%	1.9	-78%	92	74	-20%	69	-25%
KTH	15	14	-8%	14	-8%	38	35	-7%	35	-7%	6.3	3.2	-49%	3.2	-49%	84	90	+7%	88	+5%
BLUE	9.6	7.5	-22%	7.8	-18%	68	45	-33%	45	-34%	16	3.3	-79%	3.4	-79%	212	191	-10%	184	-13%
avg.			-17%		-14%			-32%		-34%			-68%		-69%			-22%		-24%

Table 4: *Effect of predictions on the absolute difference between reservations and actual start times. Rate is the percentage of jobs that wait and get a reservation. Both rate and statistics of the distribution of differences are reduced with predictions, indicating improved performance and superior predictability, respectively.*

between a job's reservation and its actual start time. This is only computed for jobs that actually wait, 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 between their reservations and start times. Note that with prediction correction a job may have multiple reservations during its life; we use the first for the predictability measurements.

The predictor we use (in this section only) is slightly different from the one used in Sec. 3: instead of using the last two jobs to make a prediction, we only use them if their estimate is identical to that of the newly submitted job; otherwise, we fall back on the user estimate. The reason is that this is the optimal predictor in this case; a full discussion of the tradeoffs along with results for the predictor used so far are given in Sec. 6. Results are shown in Table 4. Evidently, the rate of jobs that need a reservation is consistently reduced by 8–22% when predictions are used, indicating more jobs enjoy backfilling and reduced wait times. The rest of the table characterizes the associated distribution of absolute differences between reservations and start times. Both EASY$^+$ and EASY^{++} obtain big reduction in the average differences: e.g. on SDSC, from almost 3 hours (171 minutes) to about an hour and a half (91 minutes). Reductions in median differences are even more pronounced: they are at least halved across all traces, with a 79% top improvement obtained by EASY^{++} on BLUE. The variance of differences is typically also reduced, sometimes significantly, with an exception of a 5–7% increase for KTH. The bottom line is therefore that using runtime predictions consistently and significantly improves predictability of jobs' starting time.

Improving the quality of reservations on average is desirable e.g. for grid co-allocation where

trace	rate [% of jobs]					avg. delay [minutes]				median delay [minutes]				stddev delay [minutes]			
	$EASY$	$EASY^+$		$EASY^{++}$		$EASY$	$EASY^+$		$EASY^{++}$	$EASY$	$EASY^+$		$EASY^{++}$	$EASY$	$EASY^+$		$EASY^{++}$
SDSC	1.5	3.8	+149%	3.8	+150%	513	92	-82%	86	-83%	0.9	8.6	+896%	8.3	+859%	1442	223	-85%	206	-86%
CTC	0.7	1.3	+81%	1.3	+83%	72	37	-49%	34	-53%	1.9	3.0	+62%	2.6	+43%	119	102	-14%	94	-21%
KTH	0.1	1.8	+1518%	1.8	+1493%	58	52	-11%	44	-23%	0.7	11	+1541%	9.9	+1428%	108	107	-1%	87	-20%
BLUE	0.9	1.8	+97%	1.8	+102%	48	35	-28%	31	-35%	0.8	2.2	+174%	2.1	+165%	318	154	-52%	136	-57%
avg.			+461%		+457%			-42%		-48%			+668%		+624%			-38%		-46%

Table 5: *Effect of predictions on the delays beyond a job's reservation. With predictions, the rate and median delay are increased, but the average and standard deviation of delays are reduced.*

it is important for a job to start exactly on time. However, it is conceivable some systems would care more about jobs being delayed beyond their reservation, than started earlier. Table 5 shows the rate of delayed jobs and the distribution of actual delays. Even with plain EASY 0.1–1.5% of the jobs are delayed, because (as reported earlier) jobs sometimes outlive their user estimates. Unfortunately, when predictions come into play, the delays become much more frequent and involve 1.3–3.8% of the jobs. On the other hand, both the average delay and its standard deviation are dramatically reduced, e.g. SDSC's average drops from about 8.5 hours (513 minutes) to less than 1.5 (86 minutes) and its standard deviation drops at a similar rate. Medians values, however, increase by up to an order of magnitude (KTH/SDSC), though in absolute terms they are all less than ten minutes. This indicates that EASY's delay-distribution is highly skewed and that our techniques curb the tail, at the expense of making short delays more frequent.

Nevertheless, there are two solutions for systems that do not tolerate delays. One is to employ double booking: leave the internals of the algorithms based on predictions, while reporting to interested outside parties about reservations which would have been made based on user estimates (never violated if jobs are killed on time). This solution enjoys EASY^{++}'s performance but suffers from EASY's (in)accuracy. The other solution is to backfill jobs in prediction order, but only if their user-estimated termination falls before the reservation. This ensures backfilled jobs do not interfere with reservations, at the price of reducing the backfilling rate. Indeed, this algorithm enjoys all the benefits of the "+" variants in terms of internal accuracy, while being similar or better than EASY with respect to unwarranted delays. As for performance, it is 1–10% better than that of EASY$_{SJBF}$ (Table 3).

trace	wait [minutes]						b. slowdown						accuracy [%]					
	$EASY$	$X2$		SJF		$EASY^{++}$	$EASY$	$X2$		SJF		$EASY^{++}$	$EASY$	$X2$		SJF		$EASY^{++}$
SDSC	363	333	-8%	535	+47%	327	-10%	99	89	-10%	69	-30%	70	-29%	32	16	-49%	32	-0%	60	+87%
CTC	21	20	-8%	13	-38%	14	-33%	4.6	4.1	-10%	2.8	-40%	2.9	-37%	39	20	-49%	39	+0%	62	+61%
KTH	114	102	-11%	79	-31%	95	-17%	90	80	-11%	45	-50%	57	-36%	47	24	-50%	47	-0%	61	+28%
BLUE	130	115	-11%	81	-38%	87	-33%	35	30	-15%	25	-29%	19	-47%	31	16	-47%	31	+0%	62	+102%
avg.			-10%		-15%		-23%			-12%		-37%		-37%			-49%		+0%		+70%

Table 6: *Average wait and bounded slowdown achieved by EASY^{++} compared with two other schedulers proposed in the literature: doubling user estimates and using SJF scheduling.*

5 Relationship With Other Algorithms

Our measurements so far have compared various scheduling schemes, culminating with EASY^{++}, against vanilla EASY. However, other variants of 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 mentioned 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 6. 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'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. But the techniques presented in this paper can be used to enhance *any* backfilling algorithm. Table 7 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

trace	doubling															shortest job									
	wait [minutes]								b. slowdown								wait [minutes]					b. slowdown				
	$X2$	$X2^+$		$X2_{perf}$		$X2^{++}$		$X2^{++}_{perf}$		$X2$	$X2^+$		$X2_{perf}$		$X2^{++}$		$X2^{++}_{perf}$		SJF	SJF^+		SJF_{perf}		SJF	SJF^+		SJF_{perf}
SDSC	333	357	+7%	293	-12%	333	-0%	270	-19%	89	94	+6%	77	-13%	67	-25%	58	-34%	535	308	-42%	270	-50%	69	34	-51%	19	-73%
CTC	20	16	-18%	18	-8%	15	-25%	16	-16%	4.1	3.6	-13%	3.2	-21%	3.0	-28%	2.5	-38%	13	12	-11%	12	-11%	2.8	2.4	-12%	1.8	-35%
KTH	102	98	-4%	95	-6%	93	-8%	84	-18%	80	66	-18%	70	-13%	53	-33%	50	-38%	79	87	+10%	67	-16%	45	44	-2%	24	-46%
BLUE	115	105	-9%	107	-8%	86	-26%	80	-31%	30	33	+8%	28	-7%	21	-32%	12	-59%	81	90	+11%	50	-39%	25	37	+49%	5.4	-78%
avg.			-6%		-8%		-15%		-21%			-4%		-14%		-30%		-42%			-8%		-29%			-4%		-58%

Table 7: *Average performance and (shaded) improvement when optimizing vanilla $X2$ and SJF.*

(predicted) job, rather than to the one that has waited the most.[6] The theoretical optima of $X2^+$, $X2^{++}$, and SJF$^+$, are $X2_{perf}$, $X2^{++}_{perf}$, and SJF$_{perf}$, respectively (use perfect estimates instead of system-generated predictions).

Table 7 shows that switching from $X2$ to $X2^+$ can better performance (up to -18% in CTC's wait and KTH's slowdown) or worsen it (up to +8% in BLUE's slowdown), though improvements are more frequent and on average, $X2^+$ is 4-6% better than $X2$. When further optimizing by adding SJBF ($X2^{++}$), performance is consistently better, with a common improvement of 25-33%. The result of upgrading SJF to SJF$^+$ is once again inconsistent among traces/metrics, but here too improvements are more frequent (4-8% on average). In all cases, using prefect predictions ($X2_{perf}$, $X2^{++}_{perf}$, and SJF$_{perf}$) leads to consistent improvements in performance, indicating prior inconsistency steamed from our simplistic predictor and motivating the search for a better one.

6 Does Better Accuracy Imply Better Performance/Predictability?

This study is based on the notion that superior accuracy should result in improved performance (better packing) and predictability (better individual runtime predictions). However, we have also witnessed several occasions in which these metrics conflict. The **first** and most obvious is shown in Fig. 3 where deliberately making estimates less accurate (doubling) consistently improves performance. A **second** example is related to the predictor switch done in Sec. 4. Throughout this paper we've used an all prediction window, where the last two terminated jobs by the same user were used for prediction, regardless of their attributes. In contrast, in Sec. 4 we've used an immediate

[6] SJF$^+$ and SJF^{++} are equivalent because both employ SJBF by definition.

trace	performance						accuracy			prediction			py_{abs}						py_{delay}					
	wait [min]			b. slowdown						rate [job %]			rate [job %]			minutes			rate [job %]			minutes		
	imm	all		imm	all		imm	all		imm	all		imm	all		imm	all		imm	all		imm	all	
SDSC	363	327	-10%	81	70	-13%	56	60	+8%	59	89	+52%	15	12	-21%	91	98	+8%	3.8	5.0	+30%	86	150	+74%
CTC	17	14	-16%	3.6	2.9	-20%	59	62	+6%	63	90	+44%	5.7	5.1	-11%	27	27	-1%	1.3	1.7	+26%	34	53	+56%
KTH	98	95	-3%	67	57	-15%	58	61	+5%	39	84	+115%	14	11	-22%	35	63	+78%	1.8	3.6	+106%	44	149	+236%
BLUE	100	87	-13%	19	19	-2%	59	62	+5%	70	90	+29%	7.8	5.2	-33%	45	65	+46%	1.8	2.0	+10%	31	136	+333%
avg.			-10%			-12%			+6%			+60%			-22%			+33%			+43%			+175%

Table 8: *Comparing the immediate and all versions of EASY^{++}: py_{abs} relates to metrics from Table 4 (absolute difference between start time and reservation); py_{delay} relates to metrics from Table 5 (delay beyond a reservation). The all version is $\backsim$10% better in terms of average performance and 6% more accurate. Nevertheless, despite its improved accuracy, it seems to loose in predictability: its py_{abs} rate is 11-33% lower (good), but the actual difference might be 78% higher (KTH); worse, both rate and duration of delayed jobs are significantly increased (KTH's rate is doubled, BLUE's delay is more than quadrupled).*

window, in which we generate a prediction only if these two jobs have user runtime estimates that are equal to that of the newly submitted job (i.e. they are "similar"). The fact of the matter is that all (which is more accurate) is better for performance, whereas immediate appears as better for predictability (Table 8). Further, a **third** example is that the performance of immediate-EASY^{+} and $X2$ is very similar (Table 9). These schedulers are identical in every respect, except EASY^{+} uses runtime predictions whereas $X2$ uses something that is even less accurate than user estimates (user estimates that are doubled). The fact the two yield similar performance raises the question of whether it is worthwhile to even bother with runtime prediction.

This section addresses these three examples and explains what really happens to make accuracy and performance seem contradictory. We begin by explaining why doubling estimates helps performance. Assume all runtime estimates are perfectly accurate (the same explanation holds for user estimates). Fig. 8 illustrates the dynamics of $X2$ backfilling. Based on the information available to the scheduler at T_0 (time 0), it appears the earliest time for J_3 (job 3) to start is T_{12}, even though the *real* earliest start time is actually T_6. Thus, the scheduler makes a reservation on J_3's behalf for T_{12} and can only backfill jobs that honor this reservation. At T_4, J_2 terminates. As J_1 is still running, nothing has changed with respect to J_3's reservation, and so the scheduler scans the wait queue in search of appropriate candidates for backfilling. J_4 fits the gap between T_4 and the reservation (T_{12}) and so it is backfilled, effectively pushing back the real earliest time at which J_3 could have started from T_6 to T_8.

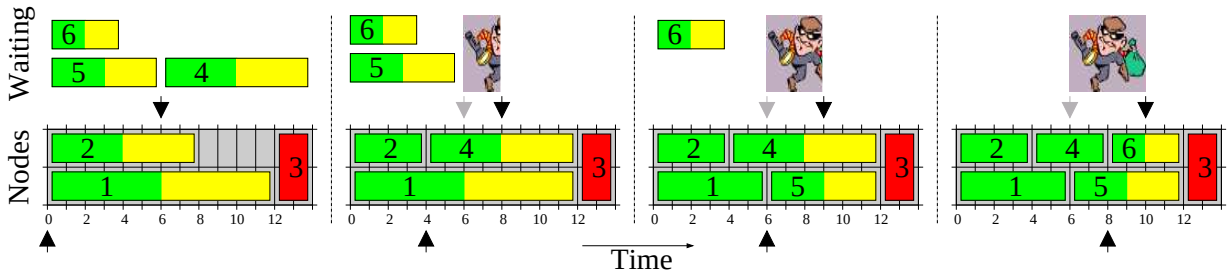

Figure 8: *The dynamics of $X2$ backfilling. Job numbers indicate arrival order. The left portion of jobs (green/dark) indicates their real runtimes. Due to the doubling, the scheduler views jobs as twice as long (right portion; yellow/bright). The bottom arrows show the progress of time, whereas top black arrows show the earliest time at which job 3 would have been started, had real runtimes been known (at that particular point in time). The thief's width shows the amount of "stolen" time, at the expense of job 3.*

trace	wait [minutes]			b. slowdown		
	imm	*X2*		*imm*	*X2*	
SDSC	343	333	-3%	92	89	-3%
CTC	18	20	+7%	3.7	4.1	+10%
KTH	108	102	-6%	79	80	+2%
BLUE	121	115	-4%	31	30	-3%
avg.			-2%			+2%

Table 9: *$X2$ and immediate $EASY^+$ yield similar performance despite the fact they are identical except the latter improves predictions whereas the former worsens them.*

trace	stalled rate [%]			stall duration [minutes]			avg. thieves #		
	EASY	*X2*		*EASY*	*X2*		*EASY*	*X2*	
SDSC	7.2	11	+49%	91	137	+50%	1.9	2.1	+9%
CTC	9.1	11	+9%	35	50	+42%	2.5	2.8	+10%
KTH	7.0	11	+61%	51	107	+111%	1.6	2.3	+45%
BLUE	9.2	12	+29%	54	118	+119%	2.3	3.1	+33%
avg.			+39%			+80%			+24%

Table 10: *"Heel and toe" effect is amplified due to $X2$. Rate is the percent of jobs that had their earliest start time pushed back due to the effect, out of waiting jobs that got a reservation. Duration is the average period between a job's earliest start-time (computed according to perfect estimates) and its actual start-time. "Thieves" indicate the per-job average number of times the earliest start-time is pushed back (3 times for J_3 in Fig. 8).*

This "heel and toe" scenario, of repeatedly pushing away the earliest starting point of the first queued job, may continue until T_{12} is reached. During this time, the window between the current time and the reservation time is continuously shortened, such that waiting jobs that fit this open gap get shorter and shorter. It is this limited form of "SJFness" which is the source of the $X2$ performance improvement: Note that waiting jobs that are assigned a reservation are usually both long and wide (under EASY, we measured the average runtime-estimate and size to be 7 hours and 17% of the machine's processors, respectively). As shorter jobs are prioritized at the expense of these jobs, what $X2$ is really doing is *trading off FCFS-fairness for performance*. Indeed, when doubling real user estimates the "heel and toe" effect is greatly amplified (Table 10).

Based on this analysis, comparing between $X2$ and immediate EASY$^+$ (Table 9) is actually comparing between different types of unrelated and *orthogonal* optimizations: favoring shorter

jobs vs. improving predictions. Thus, we contend that doubling should be viewed as a property of a scheduler, *not* the prediction algorithm. Indeed, both Fig. 3 and Table 7 indicate that doubling of improved predictions (whether perfect or based on history) yields better performance than when doubling the lower quality user runtime estimates. So predictors should strive to make the best predictions they can and leave the choice of whether to double or not in the hands of the scheduler.

The remaining open issue is that all, which is more accurate, seems less predictabile than immediate in Table 8. Nevertheless, all is actually more predictable. First, consider py_{abs}. While the absolute difference under immediate is reduced, the rate of jobs that suffer such a difference is significantly higher. To see which of the two metrics have more impact (rate or difference), we computed the average difference with respect to *all* the jobs in the log (product of Table 8's "rate" and "minutes" columns, divided by 100). This reveals that all is actually more predictable than immediate in 3 out of the 4 logs.

As for py_{delay}, this metric is actually very problematic and should not be used alone. For example, $X2$ obviously reduces the accuracy of estimates, but has much lower py_{delay} than using the estimates as is, because it computes reservations based on unrealistically too-long predictions; tripling the estimates would make the effect even more pronounced. Likewise, immediate produces less predictions than all and therefore falls back on user estimates more often (Table 8's prediction rate). This explains why immediate is less accurate. Additionally, as estimates are bigger than predictions (by definition), immediate's reservations are further away in the future. In other words, py_{delay} is an unreliable predictability metric as it only accounts for "one side to the coin": jobs that run *later* than their reservation.

7 Tuning Parameters

The EASY^{++} algorithm has several selectable parameters that may affect performance. We have identified seven parameters, mainly concerned with the definition of the *history window*: which previous jobs to use, and how to generate the prediction. Some of these have only two optional values, while others have a wide spectrum of possibilities.

To evaluate the effect of different settings, we simulated *all* 8,640 possible parameter combinations[7], henceforth called *configurations*, using our four different workloads. This led to a total of nearly 35,000 simulations,[8] each yielding two performance metrics. Thus, each configuration is evaluated by eight *testcases* ($\{$SDSC,CTC,KTH,BLUE$\}\times\{$wait,slowdown$\}$).

The results of the simulations indicate that the "performance surface" is extremely noisy. There are many different and seemingly unrelated configurations that achieve high performance, but there is no single configuration that is best for all eight testcases. In order to provide effective guidance in choosing the parameters we therefore performed a joint analysis of all the data. Our goal is to find the best configuration, where "best" means robust good performance under all eight testcases. We anticipate that such a configuration will also perform well under other conditions, e.g. with new workloads, as will be explained below.

The analysis is done as follows. We start by ranking all 8,640 configurations in two steps. First, we evaluate the degradation in performance of each configuration c under each testcase t relative to the best configuration b for this test case. (Let P_b and P_c be the performance of b and c under t, respectively, then c's degradation under t is $100\frac{P_c}{P_b} - 100$.) Thus, each configuration is now characterized by eight numbers reflecting its relative performance degradations under the eight testcases. In the second step we average these eight values and the configurations are ranked accordingly (1 is the best and has the lowest average; 8,640 is the worst). Even with this ranking, the top configurations are rather diverse (Table 11; parameters will be discussed shortly).

It is important to note that our methodology is finding a *compromise* that reflects all eight testcases. For example, the top ranking configuration is not top-ranked for any of the testcases individually. Instead, it suffers a degradations ranging from 4.1% to 21.8% relative to the best configurations for each testcase. But its average degradation is only 9.4%, which is lower than the average of any other configuration.

Recall we are searching for *robust* configurations. This robustness should manifest itself by

[7] Product of the number of different values each parameter in Fig. 9 may have: $8,640 = 3 \times 2 \times 3 \times 30 \times 2 \times 2 \times 4$.

[8] Some combinations were actually equivalent and were therefore only done once; an example is making predictions using the average, median, maximum, or minimum of history jobs when there is only one history job.

rank	average degradation	configuration						
		window size	window type	fullness	metric	fallback	propagation	prediction correction
1	9.41%	11	all	partial	avg	est	yes	estimate
2	10.60%	3	ext	full	avg	rel	yes	estimate
3	10.84%	16	all	full	med	rel	yes	estimate
4	11.16%	21	all	partial	med	rel	yes	estimate
5	11.25%	10	all	full	med	est	yes	estimate
6	11.31%	4	ext	partial	min	rel	yes	estimate
7/8	11.47%	9	all	partial	med	rel/est	no	estimate
9	11.56%	2	ext	full	min	est	yes	estimate
10/11	11.61%	22	all	partial	med	rel/est	no	estimate
8640	239.88%	26	all	full	min	est	no	gradual

Table 11: *Top and bottom ranked configurations.*

being immune to trivial changes and small modification. Our top ranking configuration does not qualify as such: it uses 11 jobs for its prediction window, but when this value is replaced with 12, the associated configuration is ranked 1,295 and suffers *double* the average performance degradation. It would be ludicrous to assume 11 is a magic number and to recommend using it based on this analysis. We therefore search for a *contiguous subspace* within the configuration space, such that all its population yields good results.

The distributions of the different parameter values are shown in Fig. 9, and we now discuss each one in turn, starting with those that are easiest to characterize (left to right). The first parameter is how to perform **prediction correction** (when the predicted termination has arrived but the job continues to run). One option is to simply revert to the original user estimate. Other options are to grow the prediction gradually (by predefined increments as in Section 3), or in an exponential manner (by adding e.g. 20% each time). The results (top left of Fig. 9) clearly indicate that it is best to jump directly to the full user estimate, and not to first try lower predictions, as this option dominates 90% of top-ranked configuration. This is probably so because using the full user estimate opens the largest window for backfilling. Using a gradual increase is especially bad, and dominates the bottom half of the ranked configurations.

When we cannot generate a prediction due to lack of historical information, we use the user estimate as a **prediction fallback**. The estimate can be used as is, or it can be relatively scaled according to the accuracy the user had displayed previously [29]. The results (bottom left) show

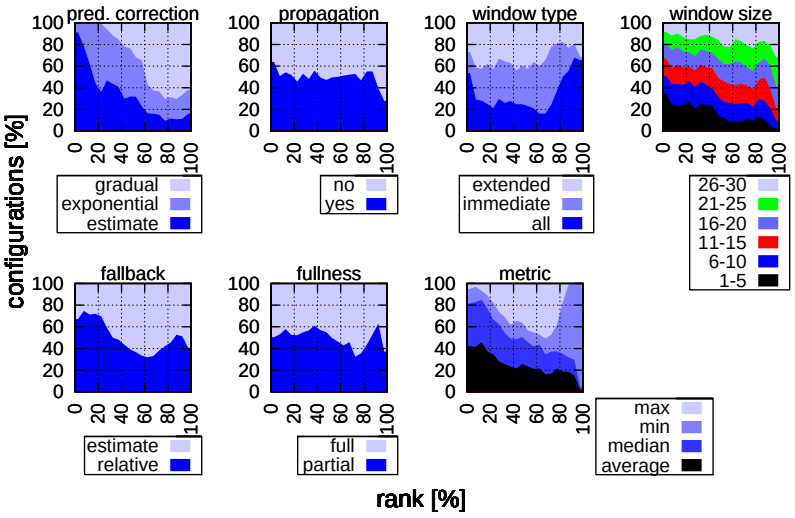

Figure 9: *Distributions of ranked configurations.*

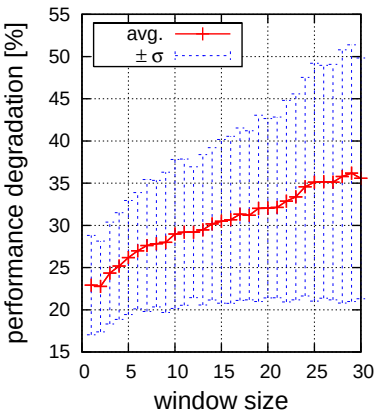

Figure 10: *Correlation of window size with degradation in performance.*

that relative provides a slight advantage, as it appears more often in high ranking configurations.

The next two parameters turned out not to have such decisive results, at least not when considered in isolation (second column from the left). **Propagation** refers to the action taken when new data 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? The second is window **fullness**. The window is the set of history jobs that is used to generate predictions. The two options are to allow a partial window, meaning that a prediction is made based on whatever data is available, or to require a full window and use the user estimate as a fallback if enough jobs are not available. For both these parameters, the possible values are approximately evenly spread across the ranked configuration. The slight advantage of propagation seems not enough to justify its computational complexity. On the other hand, partial is significantly better when larger prediction windows are employed (not shown).

The last three parameters have intricate interactions that will eventually lead to the configuration subspace we seek. The first is the **prediction metric**. Given a set of history jobs, how should a prediction be generated? Four simple options are to use the average, median, minimum, or maximum of the runtimes of these jobs. Evidently, minimum tends to lead to a low-ranking configuration, and the maximum to a middle rank. The average and the median share 80% of the

top-ranked configurations, leaving the question of which one should be used.

A harder question occurs with the **window type**. The three types are all, meaning that all recent jobs are eligible, immediate, meaning that recent jobs are used only if they are similar to the new job (same estimate), or extended, meaning similar jobs are used even if they are not the most recent (using the entire user history). The problem is that the all distribution has a U shape: it accounts for more than half the top-ranked configurations, but also for two-thirds of the lower-ranked ones.

Finally, a third difficult question is how to set the **window size** (the number of history jobs to consider). We simulated all sizes in the range 1–30; the graph (top right in Fig. 9) shows them in bins of 5. While smaller windows are more common in high-ranking configurations, there is no range-size that can be said to *dominate* high-ranking configurations.

To solve these problems we need to employ additional considerations, and to carefully study the interactions among the problematic parameters. We start with the window type parameter. There are actually big advantages to using the all window type. First, its evident top ranking peak. Second, it is easier and more efficient to implement, because we just need to keep a record of the runtimes of the last terminated (and most recently submitted) jobs by the user, and don't need to check for job similarity. The problem is that many configurations that employ an all window type are low ranking. The question is therefore whether we can avoid them (and how). Luckily, it seems that this can be done by a judicious choice of the other parameter values.

Specifically, there are 1298 configurations in the bottom-ranked 30% that employ an all window type. Of these, only 194 use the estimate directly as a prediction correction. As using estimate was shown above to be obviously beneficial, this helps eliminate 85% of the problematic configuration. Of the remaining configurations, 186 use the minimum prediction metric *and* employ a relatively large prediction window (≥ 7, with average of 18.8). It turns out the huge tail of minimum (Fig. 9) is mostly associated with large window sizes, and that increasing the window size consistently worsen the average degradation across *all* configurations. In fact, Fig. 10 shows the connection between size and degradation is almost linear (both average and variance), with the

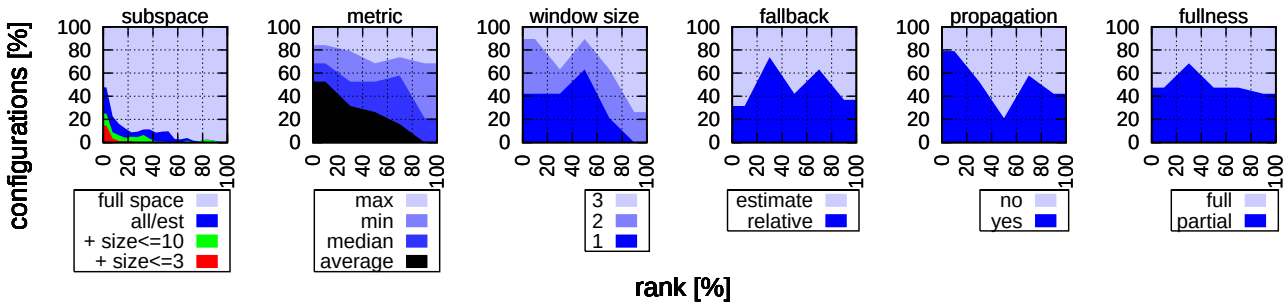

Figure 11: *Left: chosen subspace (type=*all*, correction=*estimate*) with decreasing size of window. Others: parameters distribution within the smallest shown subspace (size $\leq$ 3).*

parameter	*description*	*suggestion*
window size	how many history jobs to consider	1-2
job selection	which jobs to include in the window	all
metric	how to generate predictions	average
correction	how to increase too-short prediction	user estimate

perf. degradation	*size=1*	*size=2*	*full space*
min	11.7%	12.0%	9.4%
max	15.3%	14.7%	239.9%
average	14.2%	13.4%	30.4%
std. deviation	1.2%	0.7%	10.1%

Table 12: *Suggested settings for EASY^{++} (left), and performance degradation statistics of size=1 / size=2 configurations within this chosen subspace, compared to the statistics of all 8,640 configurations (right).*

exception that 2 is slightly better than 1.

The bottom line is that using an all window-type is actually safe in combination with estimate prediction correction and a small window size (< 7), eliminating more than 99% of all's tail configurations and clearly making it the best choice. Indeed, this subspace seems to meet our robustness demands, as is shown in Fig. 11 (left). Accordingly, we choose to limit the window size of our chosen subspace to be ≤ 3. The rest of the sub-figures explore the remaining parameters within this subspace. Clearly, average is the preferable metric. Additionally, 1-2 sized windows are preferable over 3. However, it is hard to decide between the two because size=2 dominates the top (50%) but the worst-case of size=1 is better than that of size=2, and so we seem to have a tie. As there are also no clear winners within the other parameters, we conclude by summarizing our recommendations in Table 12, which match the prediction algorithms used in this paper.

Note this is in disagreement with previous work: Gibbons used all the history available [12], and Smith experimented with a limited history to reduce the size of the search space, implying a preference for the full history [29]. However, he did not show results. We also intuitively felt that when using historical information, it would be necessary to focus on *similar* jobs, i.e. those with the same partition size, executable, estimate, etc. This motivated the definition of the extened

window type. However, the results clearly show that *recency* is more important than similarity (Fig. 10) — it is better to use the last job by the same user than to search for the most similar job. This implies that the overheads for storing and searching through data about different classes of history jobs can be avoided altogether[9]. We note in passing that Gibbons also used a different prediction metric: the 95th percentile of history jobs, which is close to the maximum metric, and was shown above to be inferior to average.

8 Conclusions

The most popular scheduling policy for parallel systems is FCFS with backfilling, as introduced by the EASY scheduler [10, 6]. With backfilling, users must supply an estimate of how long their jobs will run, to enable the scheduler to make reservations and ensure that they are respected. But user estimates are highly inaccurate and significantly reduce system performance [34]. The alternative is system-generated predictions based on users' history [9, 12, 29, 19, 25, 20], which are considerably more accurate. Nevertheless, predictions were never incorporated into production systems. This paper is about identifying the problems causing this situation, and providing applicable and easy to use solutions to all of them. Specifically, we identify three major difficulties and thus the contribution of this paper is threefold.

The first difficulty is of technical nature. Under backfilling, user estimates are part of the user contract: jobs that exceed their estimates are killed by the system, so as not to violate subsequent commitments. This makes system-generated predictions unsuitable, as some predictions inevitably turn out too short, and users will not tolerate their jobs being killed prematurely just because of erroneous system speculations. Researchers that noted this problem failed to solve it within the native backfilling framework [12, 25, 2, 21], but our solution is rather simple: (1) use user estimates exclusively as kill-times, (2) base all other scheduling decisions on system-generated predictions, and (3) dynamically increase predictions outlived by their jobs, and push back affected reservations, in order to provide the scheduler with a truthful view of the state of the machine.

[9] Arlitt et al. reached a similar conclusion in the context of the World Wide Web, claiming "only the topmost stack element is seeing significant reuse" when predicting a destination of a work session based on the user's history [1].

Applying this to EASY usually results in a $\backsim$25% reduction in average wait time and slowdown. We call this improved algorithm EASY$^+$.

The second major difficulty is related to a common misconception suggesting inaccuracy actually improves performance, implying that good estimates are actually "unimportant". This relies on a number of studies showing significant improvements when deliberately making user estimates even less accurate (by doubling or randomizing them [37, 27, 25]). In this respect, our contribution has two parts: (1) explaining this surprising phenomenon, and (2) exploiting it. Doubling helps because it induces "heel and toe" backfilling dynamics that approximates an SJF-like schedule, by repeatedly preventing the first queued job from being started. Thus doubling trades off fairness for performance and should be viewed as a property of the scheduler, not the predictor (indeed, we've shown that the more accurate predictions are, the better the results that doubling obtains). We exploit this new understanding to avoid the aforementioned tradeoff by explicitly using a shortest job *backfilled* first (SJBF) backfilling order. This leads directly to a performance improvement that was previously incorrectly attributed to stunts like doubling or randomizing user estimates. By still preserving FCFS as the basis, we manage to enjoy both worlds: a fair scheduler that nevertheless backfills effectively. Applying this to EASY$^+$ can nearly double the performance (up to 47% reduction in average slowdown). We call this enhanced algorithm EASY^{++}.

The third and final difficulty is related to the usability of previously suggested prediction algorithms. These all suffer from at least one (and sometimes all) of the following drawbacks: (1) they require significant memory and complex data structures to save the history of users, (2) they employ a complicated prediction algorithm (to the point of being off-line), and (3) they pay the price in terms of computational overheads for maintaining the history and searching it [12, 29, 19, 20]. Here too our contribution is twofold: (1) showing that a very simple predictor can do an excellent job, and (2) explaining why. Indeed, the improvements of EASY$^+$ / EASY^{++} reported above were obtained by employing a very simple predictor that is both easy to implement and suffers almost no overheads: the average runtime of the two most recently submitted (and already terminated) jobs by the same user. We have argued that our predictor's success stems from the fact it focuses

on *recent* jobs, in contrast to previous predictors that focused on *similar* ones (in terms of various job attributes). This claim is supported by our finding that performance degradation is more or less linearly proportional to the amount of past jobs upon which the prediction is based, suggesting a prediction window of only one or two jobs is optimal (Fig. 10).

Finally, note that while we focus on improving EASY, we have also shown our techniques can be applied equally well to any other backfilling scheduler (indeed, our work has already inspired researchers working on the eNanos grid to incorporate runtime predictions using techniques described in this paper [13]). The reason we choose to focus on EASY is its popularity in production systems, which may be attributed to the combination of conservative FCFS semantics with improved utilization and performance. Since EASY^{++} essentially preserves these qualities, but consistently outperforms its predecessor in terms of accuracy, predictability, and performance, we believe it has an honest chance to replace EASY as the default configuration of production systems.

Acknowledgments Supported in part by the Israel Science Foundation (grant no. 167/03).

References

[1] M. F. Arlitt and C. L. Williamson, "*A synthetic workload model for Internet Mosaic traffic*". In *Summer Computer Simulation Conf.*, pp. 852–857, Jul 1995.

[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*, pp. 103–127, Springer Verlag, 2002. LNCS 2537.

[3] 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.

[4] W. Cirne and F. Berman, "*When the herd is smart: aggregate behavior in the selection of job request*". *IEEE Trans. on Parallel & Distributed Syst. (TPDS)* **14(2)**, pp. 181–192, Feb 2003.

[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 2005-13, 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*, pp. 337–360, Springer-Verlag, 1995. LNCS 949.

[10] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, "*Parallel job scheduling — a status report*". In *Job Scheduling Strategies for Parallel Processing*, pp. 1–16, Springer-Verlag, 2004. LNCS 3277.

[11] D. G. Feitelson and D. Tsafrir, "*Workload sanitation for performance evaluation*". In *IEEE Intl. Symp. Performance Analysis of Systems and Software*, Mar 2006.

[12] R. Gibbons, "*A historical application profiler for use by parallel schedulers*". In *Job Scheduling Strategies for Parallel Processing*, pp. 58–77, Springer Verlag, 1997. LNCS 1291.

[13] F. Guim, J. Corbalán, and J. Labarta, "*Evaluation of using selective prediction techniques on the EASY backfilling scheduling policy*". 2006. Technical University of Catalonia (UPC). In preparation.
[14] D. Jackson, personal communication, Jan 2006.
[15] D. Jackson, Q. Snell, and M. Clement, "*Core algorithms of the Maui scheduler*". In *Job Scheduling Strategies for Parallel Processing*, pp. 87–102, Springer Verlag, 2001. LNCS 2221.
[16] R. Jain, *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991.
[17] J. P. Jones and B. Nitzberg, "*Scheduling for parallel supercomputing: a historical perspective of achievable utilization*". In *Job Scheduling Strategies for Parallel Processing*, pp. 1–16, Springer-Verlag, 1999. LNCS 1659.
[18] S. Kannan, M. Roberts, P. Mayes, D. Brelsford, and J. F. Skovira, *Workload Management with LoadLeveler*. IBM, first ed., Nov 2001. ibm.com/redbooks.
[19] N. H. Kapadia, J. A. B. Fortes, and C. E. Brodley, "*Predictive application-performance modeling in a computational grid environment*". In 8th *IEEE Intl. Symp. High Performance Distributed Computing*, p. 6, Aug 1999.
[20] S. Krishnaswamy, S. W. Loke, and A. Zaslavsky, "*Estimating computation times of data-intensive applications*". *IEEE Distributed Syst. Online* **5(4)**, Apr 2004.
[21] B. G. Lawson and E. Smirni, "*Multiple-queue backfilling scheduling with priorities and reservations for parallel systems*". In *Job Scheduling Strategies for Parallel Processing*, pp. 72–87, Springer Verlag, Jul 2002. LNCS 2537.
[22] C. B. Lee, Y. Schwartzman, J. Hardy, and A. Snavely, "*Are user runtime estimates inherently inaccurate?*". In *Job Scheduling Strategies for Parallel Processing*, pp. 253–263, Springer Verlag, Jun 2004. LNCS 3277.
[23] H. Li, D. Groep, and J. T. L. Wolters, "*Predicting job start times on clusters*". In *Intl. Symp. Cluster Computing & Grid*, 2004.
[24] D. Lifka, "*The ANL/IBM SP scheduling system*". In *Job Scheduling Strategies for Parallel Processing*, pp. 295–303, Springer-Verlag, 1995. LNCS 949.
[25] A. W. Mu'alem and D. G. Feitelson, "*Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling*". *IEEE Trans. Parallel & Distributed Syst.* **12(6)**, pp. 529–543, Jun 2001.
[26] *Parallel workloads archive*. URL http://www.cs.huji.ac.il/labs/parallel/workload/.
[27] D. Perkovic and P. J. Keleher, "*Randomization, speculation, and adaptation in batch schedulers*". In *Supercomputing*, p. 7, Sep 2000.
[28] E. Shmueli and D. G. Feitelson, "*Backfilling with lookahead to optimize the performance of parallel job scheduling*". In *Job Scheduling Strategies for Parallel Processing*, pp. 228–251, Springer-Verlag, 2003. LNCS 2862.
[29] W. Smith, I. Foster, and V. Taylor, "*Predicting application run times using historical information*". In *Job Scheduling Strategies for Parallel Processing*, pp. 122–142, Springer Verlag, 1998. LNCS 1459.
[30] W. Smith, I. Foster, and V. Taylor, "*Scheduling with advanced reservations*". In 14th *Intl. Parallel & Distributed Processing Symp.*, pp. 127–132, May 2000.
[31] 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*, pp. 202–219, Springer Verlag, 1999. LNCS 1659.
[32] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan, "*Selective reservation strategies for backfill job scheduling*". In *Job Scheduling Strategies for Parallel Processing*, pp. 55–71, Springer-Verlag, 2002. LNCS 2537.
[33] 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.
[34] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "*Modeling user runtime estimates*". In *Job Scheduling Strategies for Parallel Processing*, pp. 1–35, Springer Verlag, Jun 2005. LNCS 3834.
[35] D. Tsafrir and D. G. Feitelson, "*Instability in parallel job scheduling simulation: the role of workload flurries*". In 20th *Intl. Parallel & Distributed Processing Symp.*, Apr 2006.
[36] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam, "*An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration*". *IEEE Trans. on Parallel & Distributed Syst. (TPDS)* **14(3)**, pp. 236–247, Mar 2003.
[37] D. Zotkin and P. J. Keleher, "*Job-length estimation and performance in backfilling schedulers*". In 8th *Intl. Symp. High Performance Distributed Comput.*, Aug 1999.