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. Remarkably, predictions were never incorporated into production schedulers. One reason explaining this anomaly is studies claiming inaccuracy is actually good for performance. More important is the fact no study overcame 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. To make this work, predictions need to be corrected adaptively if proved wrong. The end result is a surprisingly simple scheduler we call EASY++, 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, and we argue it can (and in our opinion should) replace the default currently in use on production systems. In addition, our techniques can be used to enhance any backfilling algorithm previously suggested.

1. INTRODUCTION
The default algorithms used by current batch job schedulers for parallel supercomputers are all rather similar to each other. In essence, they select jobs for execution in first-come-first-serve (FCFS) order, and run each job to completion (see appendix for a detailed survey). The problem is that this simplistic approach tends to cause significant fragmentation, as jobs do not pack perfectly and processors are left idle. Most schedulers therefore use the backfilling optimization: when 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 about 15 percentage points [10].

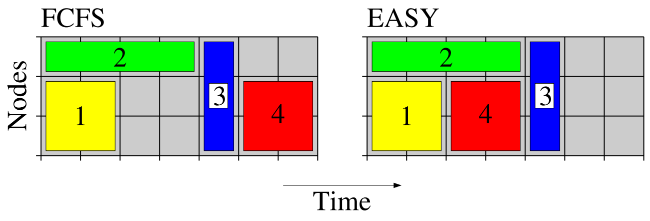

Figure 1: *EASY scheduling reduces fragmentation by backfilling. Note that it would be impossible to backfill job 4 had its length been more than 2, as the reservation for job 3 would have been violated.*

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 only allowing subsequent jobs to run provided they do not violate this reservation [13] (Fig. 1). Of course, many different options exist regarding the details of the implementation. For example, the Maui scheduler has dozens of configurable parameters that control the number of reservations, the prioritization of jobs, etc. [8]. This leads to an embarrassment of riches: system administrators are faced with myriad options, but little guidance about their use.

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, the EASY scheduler — which introduced backfilling — required users to provide a runtime estimate for all submitted jobs [13], 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 violate subsequent commitments.

The vast popularity of the EASY scheduler has enabled empirical studies of how it works in practice, based on accounting logs from multiple installations [15]. These studies showed that user estimates are generally inaccurate [14], and are reproduced in Fig. 2. The X axis in these histograms is the percentage of the estimate that was actually used; thus a job that was estimated to run for one hour but only ran for six minutes falls into the 10% bin. As can be seen, in each system about 10–20% of the jobs exceed their estimate and are killed. Also, a large number are very short (less than

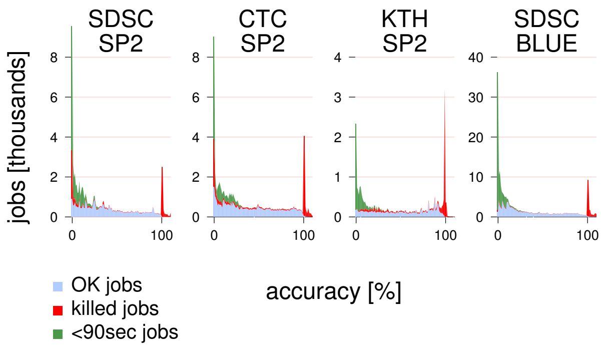

Figure 2: *Histograms of job runtime estimate accuracy from four sites (1% bins). Sites are further described in Table 1.*

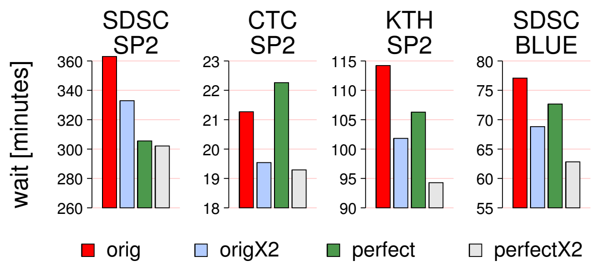

Figure 3: *The average wait-time of jobs improves when runtime estimates are doubled, both when using the original user estimates and when using accurate estimates.*

90 seconds) and only use a very small part of their estimate; these are conjectured to have failed upon startup.

The remaining jobs, representing those that ran to completion, create a somewhat flat histogram. The conclusion is that user estimates are actually rather poor, probably since users find the motivation to overestimate, so that jobs will not be killed, much stronger than the motivation to provide accurate estimates and potentially enable the scheduler to perform better packing. However, 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 [12].

The dire results regarding user estimates has 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. This has even led to the suggestion that estimates should be *doubled*, to make them even less accurate [23, 14]. The results of doing so are shown in Fig. 3, and 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. 4). It is therefore conceivable that historical data can be used to predict the future. Suggested schemes include the top of a 95% confidence interval of job runtimes [7], a statistical model based on the assumption that runtimes come from a log-uniform distribution [3], and simply to use the mean plus 1.5 standard deviations [14].

Despite all this work, schedulers in actual use still prefer plain

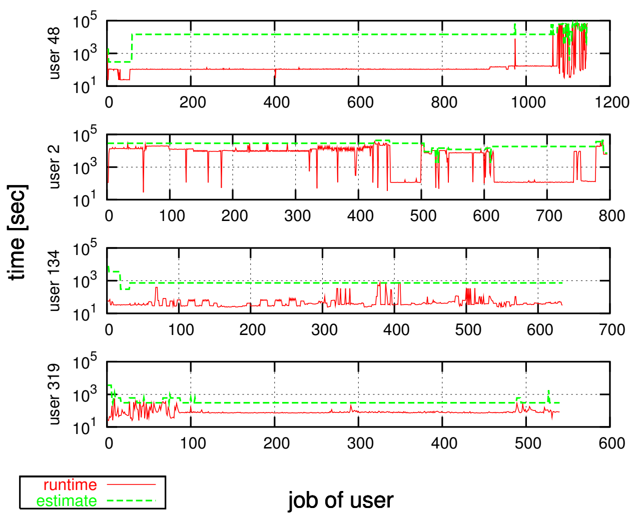

Figure 4: *Runtime and estimate of all the jobs submitted by four arbitrary users from the SDSC-SP2 trace shows remarkable repetitiveness.*

backfilling based on user estimates rather than 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 suggested the use of sophisticated greedy and genetic algorithms for on-line classification of jobs into similarity classes [17, 19].
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 [14].
3. The perception that inaccurate estimates leads to better performance negates the motivation to incorporate mechanisms for better predictions.

This paper is about refuting or dealing with all of the above claims.

Starting with the issue of complexity, it is true that some of the proposed predictors are very complex. But this need not be the case. We will 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 jobs by the same user that preceded it, constitutes a prefectly good predictor which may even double performance. In addition, not every resource management system needs to implement its predictors from scratch. We propose a standard predictor API, that if used will allow any scheduler to use new predictors as they become available. The potential benefits of a perfect predictor are such that we expect the quest for a better predictor to continue, rewarding any scheduler that utilizes this interface.

The issue of integrating good predictions with backfilling is actually quite simple. 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

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 is from the SDSC Blue Horizon, which uses the LoadLeveler infrastructure and the Catalina scheduler (that performs backfilling and supports reservations)*

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. From a user's point of view this is actually the same as what the EASY scheduler does. What really happens under the covers is not known to users.

The third argument, that "inaccuracy helps", is actually false in three respects. First, Fig. 3 indeed shows that doubling user estimates usually outperforms using perfect estimates. However, as can be seen, doubling may be applied to the perfect estimates themselves, leading to performance consistently better than all the other models, including the doubled user estimates. And so, doubling predictions produced by a quality predictor has the potential to be comparable to doubled perfect predictions.

The reason that doubling helps is that it allows short jobs from the back of the queue to move forward. Since the dawn of (scheduling) times, it has been known that favoring shorter jobs significantly improves overall performance. Most of the studies dealing with predictions and accuracy indicate that improved performance due to increased accuracy really kicks in when shorter jobs are favored. [7, 19, 23, 11, 16, 1]. Thus attributing the improved performance to the inaccurate estimates is wrong — in reality, the inaccurate estimates just make the schedule resemble a SJF schedule, and it is the "SJFness" that leads to better performance [24].

If that is the case, why not simply use SJF outright? To make the change as small as possible while eliminating starvation, we suggest to keep reservations order as it is in EASY, namely FCFS. The only change we introduce is to the order in which backfilling operations are performed such that shorter jobs are *backfilled* first (SJBF if you will). We hope that the conservative nature of our algorithm, along with the significantly better performance it yields, will encourage developers to implement it, and to set it as the default.

The third fallacy in the "inaccuracy helps" claim is that good predictions are unimportant. In fact, they are important in various contexts, most notably advanced reservation for grid co-allocation, shown to considerably benefit from better accuracy [18]. Another context is batch scheduling of *moldable* jobs for which the user doesn't care about the size of the partition as the jobs may run on any size [3, 19]. The goal of such a scheduler is to minimize response time of jobs and therefore it must consider whether waiting for a while in order for more nodes to become available is preferable over running immediately on what's available. Of course a reliable prediction of how long it will take for the additional nodes to become available is crucial.

2. METHODOLOGY

The experiments are based on an event-based simulation, where events are job arrival and termination. Upon arrival, the scheduler is informed of the number of processors the job needs, and its estimated runtime. It can then either 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 freed processors. The runtime of jobs is part of the input to the simulation, but is not given to the scheduler.

Table 1 lists the four traces we've used to drive the simulations. As suggested in the Parallel Workloads Archive, we've chosen to use their "cleaned" versions [15, 22]. 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).

Two types of metrics are used: to measure the accuracy of predictions, and to measure the performance of scheduling. The measure of accuracy is the ratio of the real runtime to the prediction. This reflects the amount of predicted time that was actually used. Section 4 elaborates on the manner in which average accuracy is measured.

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 [6] yielding the formula

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

where T_r and T_w are the job's run and wait time, respectively.

To reduce warmup effects, the first 1% of terminated jobs were not included in the metric averages [9].

3. INCORPORATING PREDICTIONS INTO BACKFILLING SCHEDULERS

Before we start, let us set terminology straight. The term ***estimate*** is used to describe the runtime approximation supplied *by the user* upon job submission. The major role of this value is that it serves as a runtime upper bound after which jobs are killed. This value never changes. The term ***prediction*** is used to describe a value that is used *by the scheduler* to approximate the expected runtime of a job. We argue this value may in principle change dynamically during the lifetime of a job, for example, when it is proven wrong due to the fact the job has outlived it. The prediction may be set to be equal to the estimate, but it does not have to be set thus.

3.1 The Problem

The simplest way to incorporate predictions into a backfilling scheduler is to use them in place of estimates. The problem is that estimates serve both as an approximation of how long a job will run, and as an upper-bound on how long the job will be allowed to run. On the other hand, predictions might very well be smaller than actual runtimes. There is no doubt users will not appreciate their jobs being killed just because the system speculated they were short. 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 [7, 19]), by assuming preemption is available (stopping jobs that exceed their prediction and reinserting them into the wait queue [7]), or by using artificial estimates generated as multiples of actual runtimes (thus assuming underestimation never occurs [23, 11, 16, 2, 1, 20, 21]). In other words, no previous study has dealt with this difficulty, which is probably the main reason why predictions were never incorporated into contemporary production batch schedulers.

Mu'alem and Feitelson [14] noted this problem, and investigated whether underestimations do 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 underestimation and would have been killed prematurely by a backfilling scheduler. They concluded that "it seems using system-generated estimates for backfilling is not a feasible approach".

3.2 First Part of the Solution: Split Estimates Dual Roles

The key idea of our solution is recognizing that the underestimation problem emanates from the dual role an estimate plays: both as kill-time and as prediction. We argue that these should be separated. It is only legitimate to kill a job once its estimate is reached, but not any sooner; therefore user estimates should only retain their role as kill-times. All the other considerations of a backfilling scheduler should be based upon predictions, which are much more accurate. There is no technical problem preventing us from running any backfill scheduler using predictions instead of estimates. The only change is that a running job is *not* killed when its prediction is reached; rather, it is allowed to continue, and is only killed when 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). 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. Implementing this predictor is truly trivial and requires less than a dozen lines of code: saving the runtime of the two most recent 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.

Fig. 5 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 of up to an order of magnitude (KTH's wait time). This happens despite the improved accuracy of the predictions.

3.3 The Role of Underestimation

The first suspect of being responsible for the dismal results was of course our ridiculously simplistic prediction algorithm. However, as noted, even these simple predictions are usually far superior to the estimates supplied by users, and may almost double accuracy.

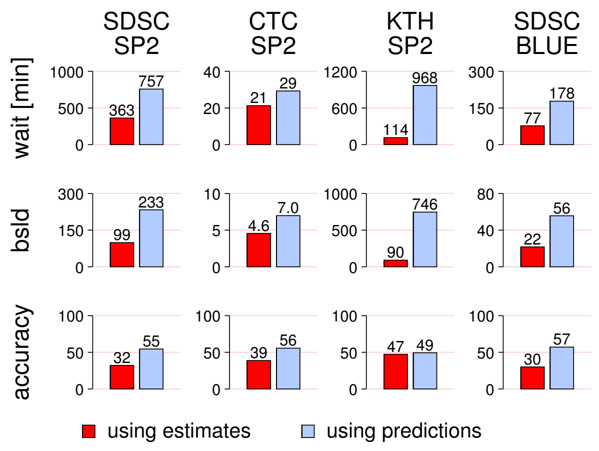

Figure 5: *Average wait time (minutes), bounded slowdown, and accuracy for EASY, using user estimates or system generated predictions.*

Discovering the underlaying reason for the performance loss required a thorough investigation of backfilling dynamics. Our in-depth analysis revealed that the true guilty party is *underestimation*, that is, cases in which a generated prediction is smaller than the job's actual runtime.

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 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!

As the reservation time sets an upper-bound on the duration of backfill jobs (at least those that don't use the extra processors [13, 14]), 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 that it actually is. If the reservation is for the current time, backfill-

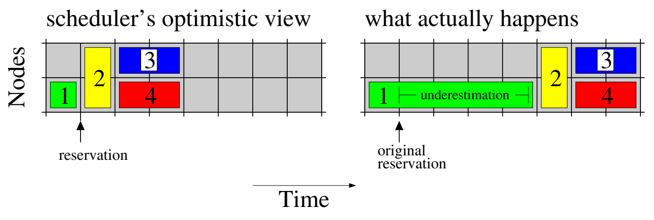

Figure 6: *Underprediction of runtimes causes the backfill scheduler to make reservations that are too early. This misconception then prevents subsequent jobs from being backfilled, due to fear that they will interfere with the reservation for the first queued job.*

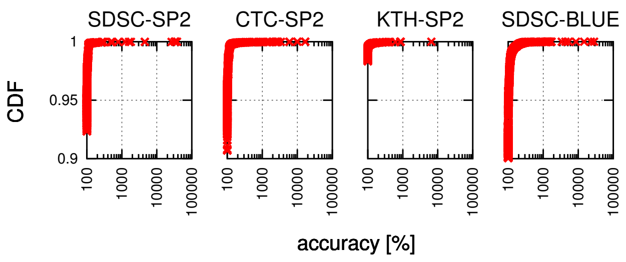

Figure 7: *Tail of the cumulative distribution function (CDF) of accuracy as computed with respect to user estimates. Only underestimated jobs are shown, for which runtime>estimate. This is a common phenomenon suffered by 2% (KTH) to 10% (SDSC-BLUE) of the jobs.*

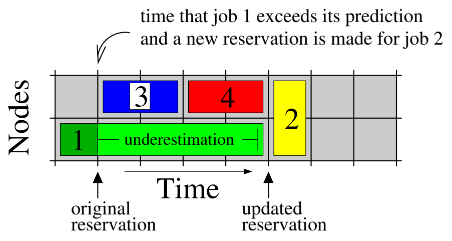
time that job 1 exceeds its prediction and a new reservation is made for job 2
Nodes
3
4
1
underestimation
2
Time
original reservation
updated reservation

Figure 8: *Prediction correction enables the backfill scheduler to escape from the early reservations that prevent backfilling. Compare with Fig. 6.*

ing is even stopped completely. The scheduling then largely reverts to plain FCFS, as shown in Fig. 6, leading to the poor performance shown in Fig. 5. This situation is rectified only when underestimated jobs eventually do terminate.

Interestingly, this also happens to some degree with current schedulers. In principle the problem should not exist in this context, as underestimation is avoided by using estimates as predictions, and killing jobs once their estimates (=predictions) are reached. However, in reality, this is not always the case. The punctilious reader may have noticed values bigger than 100% in the histogram of estimates' accuracy as presented in Fig. 2. Such data do in fact represent jobs with estimates *smaller* than runtimes. Further, not all data is shown due to limiting the X axis to 110%, and in fact much more extreme underestimations exist.

In SDSC-SP2 for example, the maximal user estimate allowed is 18 hours. However, 618 jobs had a longer runtime. The majority of these have runtimes which are only up to 10 minutes longer than 18 hours, and probably represent jobs that were killed and this process took a few minutes. But 17 jobs have runtimes which range from 22 to 142 hours. The longest job was estimated to run for only 25 minutes, which means it was underestimated by a factor of $\frac{142 hours}{25 minutes} \approx 3400$ (the record underestimation factor in this trace is even bigger: 3771). Fig. 7 shows the tails of accuracy distributions associated with original user estimates. Only underestimated jobs (for which accuracy is *bigger* than 100%) are shown. Evidently, 2–10% of the jobs suffer from underestimation. Most of these are very close to 100% (concentration of points on the Y axis), but a small fraction is associated with much bigger values (top X axis).

We are not certain why the phenomenon of extreme underestimation exists, even though it is strictly forbidden in the context of backfilling schedulers. We speculate that the large majority of underestimated jobs were killed by the system upon reaching their estimates, and that completing the kill event took a few seconds to several minutes. The remaining minority, with huge underestimation factors, probably reflect jobs run by privileged users (with ties to sysadmin personnel?) or situations in which the system didn't kill jobs because there were no queued jobs. Regardless of the reason, in reality the phenomenon occurs and must be handled by the scheduler.

3.4 Second Part of the Solution: Prediction Correction

One way to tackle the underestimation problem is to try to minimize it by producing more conservative (bigger) predictions than those previously suggested. The drawbacks of this approach are reduced accuracy and performance (recall that jobs with tight estimates have increased chance to be backfilled). Additionally there is always a chance we are not conservative enough, namely, we can never be sure underestimation is eliminated. For example, the prediction of the 101st job submitted by some user for which all 100 preceding jobs ran for exactly one minute would most certainly be in the proximity of one minute, but still the 101st job may actually be different and run for 100 hours.

A simpler approach is to avoid placing the burdon of solving the underestimation problem on the predictor. Instead, modify the algorithm to increase expired predictions proven to be too short. No doubt this is preferable to ignoring the problem and using stale data which we know for a fact is incorrect. 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, why not do the sensible thing and accept the fact it would run longer (say for 20 minutes)? Once the prediction is updated, this effects reservations for queued jobs and re-enables backfilling (Fig. 8).

The action of updating a job's prediction is named ***prediction-correction***. We do this as follows. Whenever a prediction has expired, that is, a job outlives its prediction, we distinguish between two cases. If the old prediction was smaller than the estimate, we acknowledge the fact the user was smarter than us and set the new prediction to be the estimate as was given by the user. On the other hand, if the prediction is equal to or bigger than the estimate, we have no alternative but to somehow arbitrarily enlarge it. Note that this is independent of the fact that the job should be killed. Our empirical data from the workload logs indicates that sometimes jobs continue to run far longer than they should have. The algorithm must track this somehow.

A simple way to enlarge the predictions is to add, say, 10% of the previous value. However, Fig. 9 suggests that the duration between

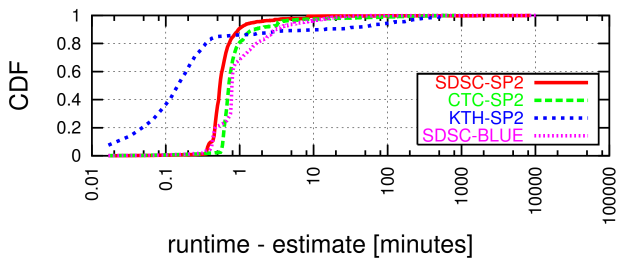

Figure 9: *Cumulative distribution function of the difference between runtime and estimate of underestimated jobs. Most estimate violations are less than one minute.*

metric	trace	EASY	EASY-PCOR		EASY+	
			metric value	improve %	metric value	improve %
wait	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
bsld	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

Table 2: *Average wait (minutes) and bounded slowdown obtained by ordinary EASY and two improved variants: EASY-PCOR adds prediction correction and achieves marginal improvement. EASY+ adds system generated predictions making improvement more pronounced.*

the expiration of an estimate until the associated job actually terminates isn't correlated with the estimate itself. This is true because the vast majority of jobs will terminate within several minutes of their estimates, regardless of their value. We have therefore decided to 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. The second adds 5, then 15, then 30 minutes, followed by 1, 2, 5, 10, 20, 50, and 100 hours.

Table 2 lists the performance results of running three algorithms with incremental sophistication: the original EASY, the same with prediction correction (EASY-PCOR), and combining prediction correction with system generated predictions (EASY+).

Checking EASY-PCOR is warranted because, as shown above, schedulers do not always kill jobs that exceed their estimates. But in most cases this has a marginal effect on performance (the maximum seen was a 5% reduction of SDSC-SP2's average bounded slowdown). This is not surprising because most estimate violations were less than a minute long (Fig. 9). However, the improvements experienced by jobs submitted shortly after gross estimate violations was much more significant (e.g. jobs in the neighborhood of the 17 very long SDSC-SP2 underestimated jobs mentioned earlier).

The tremendous value of prediction correction revels itself in EASY+, when system-generated prediction are added. This extremely trivial optimization turns a consistent failure of up to an order of magnitude degradation in performance (KTH's wait time in Fig. 5) into a consistent improvement of up to 19% (BLUE's slowdown in Table 2). 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 [5, 7]. Results in Table 2 prove for the first time that this may be put to productive use in backfilling batch schedulers, without violating the contract with users.

Note that obtaining the reported improvement is almost free. All one has to do to obtain it is set an appropriate alarm event that increments predictions if necessary, and replace estimates with the average runtime of two previously submitted jobs. Importantly, this does not change the FCFS semantics of the scheduler which are arguably one of the reasons for the tremendous popularity of EASY. Finally, we remark that prediction correction also has a positive effect on average accuracy, which stabilizes between 55–58% across all four traces when using EASY+.

3.5 Third Part of the Solution: Shortest Job Backfilled First (SJBF)

A well known scheduling principal is that favoring shorter jobs significantly improves overall performance. Supercomputers batch scheduler 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 [7, 19, 23, 11, 16, 1]. For example, Zotkin and Keleher[23] have argued that the improvement due to doubling of estimates (exemplified in Fig. 3) is actually because shorter jobs may better utilize the resulting "holes" in the schedule. Another example is given by Chiang et al. [1] showing that when ordering the wait queue by descending $\sqrt{\frac{W+R}{R}} + \frac{W}{100}$ (where W is a job's current wait-time and R is its perfect estimate=runtime), average and maximal wait times are halved, and slowdowns are order of magnitude lower! [1]

Unlike predictors which didn't find their way into the mainstream, configuring 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 (i.e. based on FCFS) which may perhaps be attributed to a reluctance to change FCFS semantics perceived as being the most fair. This has probably hurt previous proposals, which considered changing the priority function used by a backfill scheduler as a "package deal", so once priority is changed, it is applied to *all* the jobs which populate the wait queue. The practical meaning of this is that backfilling is performed relative to the reservation of the job that happens to be at the head of the wait queue according to the non-FCFS priority function that was chosen[2]. In contrast, we suggest separating reservation order from backfill order.

To avoid too large a departure from current practice, we propose a scheme that will introduce a limited amount of "SJFness", but will not completely replace FCFS. The idea is to keep reservation order

[1]Short jobs are favored since for such jobs the numerator of the first component rapidly becomes bigger than its denominator. The second component is added in an effort to reduce maximal wait times (namely avoid starvation).

[2]Though [1] do allow jobs to hold on to already allocated reservations, so that they wouldn't be "stolen" when shorter jobs arrive. However, whenever a reservation is finally allocated, it is for the shortest job currently available, not the oldest.

metric	trace	EASY	EASY-SJBF		EASY++		PERFECT++	
			metric value	imp %	metric value	imp %	metric value	imp %
wait	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
bsld	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

Table 3: *Average wait (minutes) and bounded slowdown obtained by ordinary EASY, compared with three improved variants: EASY-SJBF is the traditional algorithm enhanced with SJBF. EASY++ employs all our optimizations: system generated predictions, prediction correction, and SJBF. Theoretical PERFECT++ is similar to EASY-SJBF with the difference that actual runtimes replace estimates. Improvement is shown relative to traditional EASY.*

as determined by EASY, namely FCFS, so that *no job will be backfilled if it delays the first job in the wait queue*. But the backfilling optimization will be implemented it in SJF order, that is, Shortest Job Backfilled First — *SJBF*. We argue that this is acceptable because backfilling is non-FCFS anyway! Indeed, EASY scans the wait queue in FCFS order, but if a job cannot be backfilled, EASY moves on in an effort to backfill its successors. The reason backfilling is so successful is that it has always been perceived as "getting something for nothing". This is still the case, even though the optimization is implemented differently. Thus we believe our proposal has a decent chance in overcoming any opposition and finding its way into the default configuration. In any case, we argue that it is more sensible than "tricking" EASY into SJFness by doubling estimates [23, 14], randomizing them [11], or other similar stunts.

The first thing we have checked regarding SJBF is its effect on traditional EASY, denoted EASY-SJBF (i.e. using user estimates and no prediction correction). Table 3 summarizes the results obtained by this scheduler and their improvement relative the to basic EASY. Results indicate that SJBF is usually responsible for an improvement of more than 10%, and up to 27% (BLUE's bounded slowdown). The performance of EASY-SJBF is actually quite similar to that of EASY+ (Table 2). EASY-SJBF will be further discussed in Section 5.

Much more interesting is EASY++ which adds SJBF to EASY+ (namely combines prediction correction, system generated predictions, and SJBF). EASY++ 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 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, when considering the fact that it is often the case with experimental batch schedulers that results turn out to be conflicting, depending on the trace or even the metric being used [19, 14]. Here, every trace-metric combination yields the same conclusion — using EASY++ is highly beneficial. Even accuracy is 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 a perfect predictor 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 a 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. It would be very interesting to see the results obtained by past predictors [7, 17] when incorporated in EASY++, and check whether these come closer to PERFECT++ or are comparable to our simplistic predictor.

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

Table 4: *Summary of algorithms and optimizations they employ.*

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++ 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 [4].

Finally, we note that while EASY++ / PERFECT++ may double performance, this result is not comparable to the order of magnitude improvement obtained by [1] for NCSA traces, when using close to perfect predictions and pure SJF. This is the price paid for maintaining FCFS semantics. Nevertheless, considering the fact that supercomputers are a scarce and expensive resource, benefits of EASY++ are substantial enough in their own right.

3.6 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 4 for convenience. The rest of the paper will focus on EASY+ and EASY++.

3.7 Standardized Predictor Interface

Implementing a prediction algorithm as described above and incorporating it into a scheduler is a fairly trivial task. However it is desirable any scheduler would be able to conveniently switch between and utilize other (more sophisticated [7, 17], newer, better) predictors as they become available, without placing the burdon of reimplementing and integrating them on the scheduler's developers. We therefore propose a simple object-oriented predictor API, that if incorporated within a scheduler would allow just that.

An abstract predictor has only four methods: jobArrival, jobStart, jobDeadlineMiss, and jobFinish, invoked when the associated events occur (a deadline is missed when a prediction is proven too short) and enabling the predictor to have a full view on what's going on. Parameters of these methods are the current time and a job handle encapsulating all its available data (e.g. its user ID, runtime estimate, size, requested memory etc.). The methods return a (possibly empty) vector of pairs such that each pair associates a job ID with its new prediction. A vector is returned because a predictor may decide the new information should influence more jobs than the one associated with the event (e.g. if this is the first job of the user to terminate, the predictor may use its now known runtime as a new prediction for all the other jobs by that user, currently associated with original estimates).

A possible implementation for this API is a predictor which reflects original EASY by associating a job with its original user estimate upon the invocation of jobArrival, while returning an empty vector from the rest of the methods. And so, introducing a new predictor or replacing an existing one is obtained by simply creating / replacing a concrete instance of the API by another (the scheduler doesn't know nor care which instance is used). Indeed, the simulator used in this paper represented all predictor variants (ideal PERFECT++, realistic EASY+ / EASY++, and others described in Section 6) as instances of the suggested API.

4. MEASURING ACCURACY

In the previous section we quoted various results regarding the accuracy achieved by different algorithms. But how does one measure accuracy when both underestimated and overestimated jobs are present? And how about jobs that have more then one prediction throughout their lifetimes? This section addresses these difficulties and introduces an appropriate metric.

In principle, inaccuracies in the context of a backfilling algorithm always take the form of *overestimation*, because the rule is that the jobs are killed when their runtime reaches their user's estimate (as witnessed by the 100% peak in Fig. 2). The common way to quantify the quality of estimates [7, 3, 19, 14] is therefore to compute the average accuracy defined as $\frac{1}{N}\sum\frac{runtime_J}{estimate_J}$ over all jobs J. This should lead to a metric between 0 and 1.

But as pointed out in Section 3.3, runtimes may be underestimated, and with system-generated predictions underprediction is even common. When mixing accuracies which reflect overestimation as well as underestimation in the same average, these tend to cancel themselves out, leading to an unrepresentative value. Consider for example two jobs with runtimes of 3 minutes and estimates of 2 and 6 minutes, respectively. The associated accuracies are therefore $\frac{3}{2}$ and $\frac{3}{6}$, respectively, leading to an average of 1 which indicates perfect accuracy. This is obviously not what we want. Fig. 10 shows the false positive impact of mixing over and underestimations, which accounts for up to a quarter of perceived accuracy. To our knowledge, all previous studies of the matter ignored this issue and therefore suffer from this misfeature (that is, reported unrealistic accuracy due to mixing under and overestimation).

The solution we propose for this problem is based on the assump-

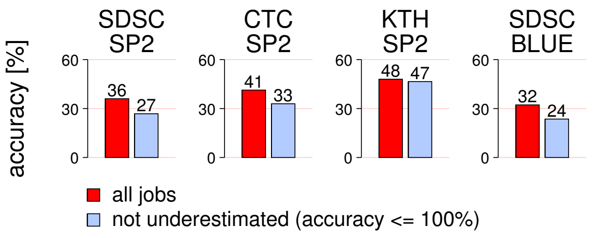

Figure 10: *Accuracy computed over all jobs (left bars) and limited to overestimated jobs (right bars). Including underestimated jobs usually introduces a significant (false) improvement to overall results.*

tion that an overestimation by some factor X has similar implications to an underestimation by the same factor. That is, predicting half the real value or twice the real value are equally bad at $X = 0.5$. We therefore propose to compute accuracy as follows

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

denoting runtime with R and estimate (or prediction) with E. This will insure accuracy is always expressed as a fraction in [0..1] and will allow meaningful summation and averaging. For example, a job with runtime=10 will have accuracy=0.5 regardless of whether its estimate is 5 or 20. Note that the first branch in the above formula is needed and may not be merged with one of the other branches, as both runtime and estimate may be zero (examples for both situations exist in the traces used). The formula suggested here is the one used throughout this paper (except when noted otherwise). When applied to the original estimates, it yields averages that fall somewhere in between the two values presented in Fig. 10.

Another problem is how to handle prediction correction, where jobs may have more than one prediction throughout their lifetimes. To overcome this difficulty we use a simple weighted average of a job's predictions, giving them a total weight of 1. For example, assume a job has been in the system for 3 minutes (from submission until termination including wait time), and had exactly two estimates associated with it throughout its lifetime: an estimate of 15 minutes was effective during the 1st minute, and an estimate of 30 minutes was in effect for the other 2. We define the average estimate of the job to be $\frac{15\times1+30\times2}{1+2} = 25$ minutes, which all things considered is quite reasonable as a representative value. In the general case, given a job J, it's average prediction is set to be $\frac{1}{(T_N - T_0)}\sum_{i=1}^{N} P_i \cdot (T_i - T_{i-1})$ such that T_0 and T_N are J's submission and termination time, respectively, and P_i is the prediction of J that was in effect from time T_{i-1} until time T_i (where $T_{i-1} \leq T_i$).

We remark that average accuracy as computed here might not be optimal in the context of backfilling, as underestimation has the potential of having worse consequences than overestimation. This might be the case because a long job that disguises as being short delays the remaining jobs in the wait queue by holding on to its resources for longer than expected, in addition to lying about its termination time which results in an earlier reservation, which in turn results in decreased backfilling options. On the other hand, overestimation harms only the job itself while it is in the wait queue, and results in a later reservation while running, thus allowing for more backfilling opportunities (at the expense of the job at the head of the FCFS wait queue). Therefore, a finer metric that differentiates between over and underestimation might be in order.

Another drawback of average accuracy is that it gives similar weight to a job with runtime of *one second* that was predicted to run for five seconds, and a job with runtime of *one hour* that was predicted to run for five hours (both have 20% accuracy). This is despite the fact the longer job has a higher potential of influencing the system, simply because it is present in the system for a much longer duration. Average accuracy is actually analogous in this respect to the average slowdown metric. Defining and using an additional metric that is analogous to average wait time (averaging the absolute *difference* between a job's runtime and its prediction, rather than their *quotient*) may be advisable. However, the above alternatives are beyond the scope of this paper and are left for future work.

5. 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, because 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 is 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.

5.1 Effect on Moldable Jobs

One of the reasons that predictability is important is for support of moldable jobs. These jobs may run on any partition size [3, 19], with the goal of earliest completion time. The scheduler is trusted to make the optimal decision on their behalf so that this time would be minimized. When a job reaches the head of the wait queue, the scheduler must consider whether waiting for more nodes to become available is preferable 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 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 such jobs and the difference between their reservation and start times. With perfect accuracy, this difference will always be zero and predictability would be complete, that is, a waiting job will have 100% certainty that additional promised nodes will be available *exactly* on time.

metric	trace	EASY	EASY-PCOR		EASY+		EASY++	
			metric value	imp %	metric value	imp %	metric value	imp %
rate (%)	SDSC-SP2	17	18	+1	14	-17	15	-16
	CTC-SP2	7	7	-1	5	-19	6	-16
	KTH-SP2	15	15	-1	14	-7	14	-8
	SDSC-BLUE	12	11	-0	10	-16	10	-14
avg.	SDSC-SP2	171	128	-25	91	-47	91	-47
	CTC-SP2	51	44	-13	29	-43	27	-46
	KTH-SP2	38	38	+1	35	-8	35	-8
	SDSC-BLUE	65	58	-11	43	-34	43	-34
med.	SDSC-SP2	64	59	-8	18	-71	19	-71
	CTC-SP2	8	7	-15	2	-73	2	-78
	KTH-SP2	6	6	+2	3	-51	3	-49
	SDSC-BLUE	16	17	+1	4	-77	4	-75
stddev	SDSC-SP2	471	179	-62	175	-63	173	-63
	CTC-SP2	92	85	-8	73	-21	69	-25
	KTH-SP2	84	84	+0	88	+5	88	+5
	SDSC-BLUE	236	196	-17	206	-13	206	-13

Table 5: *Effect of scheduler on the 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 superior predictability.*

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 5. Rate refers to the percentage of jobs that wait and get reservations. Evidently, this is consistently reduced (by 7–19%) when predictions are utilized, 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. The biggest reduction in average differences is obtained by EASY+ and EASY++ 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 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 predictions consistently results in significantly improved predictability.

5.2 Effect on Advanced Reservations

Another reason predictability is important is that it is needed to support advance reservations. These are used to coordinate co-allocation in a grid environment [18], i.e. to cause cooperating applications to run at the same time on distinct machines at remote sites. In this scenario, one might argue that a small absolute deviation from the reservation time (either before or after) is not as important as having a strong guarantee that the job will not be delayed (after).

Table 6 quantifies the extent of this phenomenon, which involves 0.1–1.5% of the jobs, when using ordinary EASY. The reason the phenomenon exists even with EASY is that, as reported earlier, jobs sometimes outlive their estimates. And indeed, with EASY-PCOR the phenomenon is somewhat reduced. When predictions come into play, the delays becomes more frequent and involve 1.7–3.9% of the jobs. On the other hand, the average and standard deviation of delays beyond reservations are dramatically reduced, e.g. in SDSC-SP2 the average drops from about 8.5 hours (513 minutes) to less than an hour and a half (87 minutes). The improvement in the standard deviation is similar. Unfortunately, this does not coincide with median measurements which increase by up to a factor

metric	trace	EASY	EASY-PCOR		EASY+		EASY++	
			metric value	imp %	metric value	imp %	metric value	imp %
rate (%)	SDSC-SP2	1.5	1.2	-19	3.9	+154	3.9	+153
	CTC-SP2	0.7	0.5	-37	1.3	+82	1.3	+83
	KTH-SP2	0.1	0.1	-10	1.8	+1515	1.7	+1489
	SDSC-BLUE	1.1	1.0	-14	2.3	+103	2.3	+107
avg.	SDSC-SP2	513	14	-97	89	-83	87	-83
	CTC-SP2	72	4	-95	36	-50	33	-54
	KTH-SP2	58	62	+7	50	-14	44	-24
	SDSC-BLUE	85	3	-96	28	-67	26	-69
med.	SDSC-SP2	1	1	-10	8	+800	8	+873
	CTC-SP2	2	1	-38	3	+68	3	+45
	KTH-SP2	1	2	+167	10	+1494	9	+1346
	SDSC-BLUE	1	1	+0	3	+236	3	+218
stddev	SDSC-SP2	1442	115	-92	216	-85	208	-86
	CTC-SP2	119	11	-91	100	-16	94	-21
	KTH-SP2	108	128	+19	96	-11	82	-24
	SDSC-BLUE	425	17	-96	91	-79	103	-76

Table 6: *Effect of scheduler on the distribution of 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.*

of fourteen (KTH). Note however that, in absolute terms, median values (less than ten minutes across all traces) seem tolerable considering the context (grid).

If it is indeed necessary to reduce the rate and median of delays, two approaches may be taken. The simplest is to employ double booking: leave the internals of the algorithms as is (do everything based on predictions), while reporting to interested outside parties about reservations which would have been made based on user estimates. In other words, waiting jobs will have two reservations: one which is based on predictions and is used internally by the scheduler, and another which is based on user estimates and serves as a more robust upper bound. Simulation results indicate that this strategy indeed works, and reduces the fraction of jobs that are started after their (second) reservation to values similar to those obtain by original EASY. Since the core algorithm is essentially unchanged, performance metrics stay the same.

Another option is to backfill a job only if both its predicted *and* estimated termination times fall before the effective reservation. This will ensure backfilled jobs will not delay jobs with reservations, at the price of dramatically reducing the backfilling rate. Results indicate that this strategy also works and the fraction of jobs that were started after their reservation is almost identical to that of EASY. Performance is also good. Let us consider the modified EASY++. This algorithm is almost identical to EASY-SJBF, as the sole difference between the two is the order in which backfilling is done: in EASY-SJBF waiting jobs are ordered according to their estimates, and in EASY++ they are ordered according to their predictions. Therefore, intuitively, the performance of the two should be similar. In reality EASY++ is slightly better, producing results that are 1–10% better than those obtained by EASY-SJBF (as reported in Table 3). On the other hand, EASY++ maintains its superiority in terms of accuracy which remains around 60%.

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

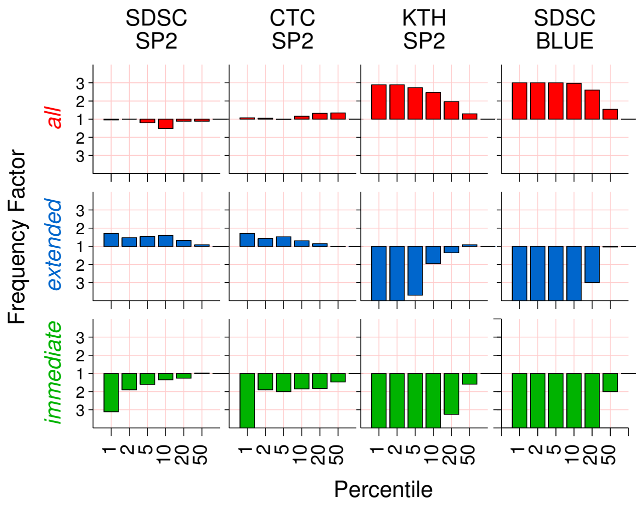

Figure 11: *Comparison of window types, with regards to the average bounded slowdown performance metric.*

using our four different workloads. This led to a total of about 220,000 simulations. This sections summarizes and presents only the main findings (the rest will be reported in a subsequent paper).

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 *window type*, which can be immediate (only consecutive similar jobs, as used in Section 5), 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. In all cases, only jobs by the same user are considered.

Window type. 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. 11. These graphs show a metric we call the *frequency factor*, which measures the success of each parameter. 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. 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 situations where they are more common than they should be in the top percentile, top 2 percentiles, top 5 percentiles, etc. The 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 window

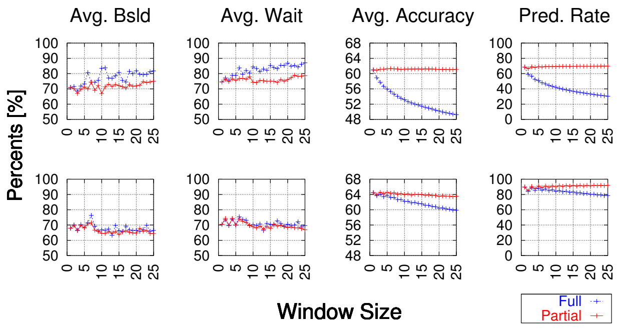

Figure 12: *The effect of window size and fullness on* immediate *windows (top) and* all *windows (bottom).*

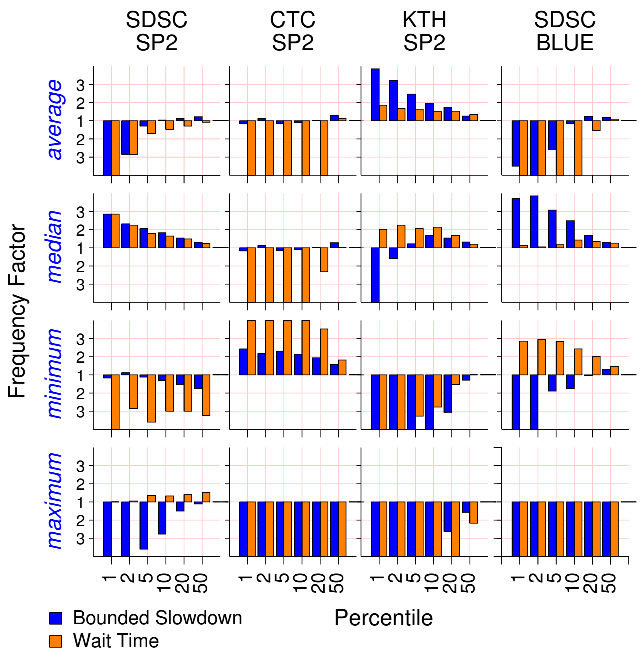

Figure 13: *Effect of different prediction methods on performance.*

types, so the frequency of each window 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 a window type immediate. This is represented by a factor of 3 below the axis, meaning that this window type is 3 times less common in that percentile than in the entire sample space, as depicted in the bottom left of Fig. 11.

Returning to this figure, we see that the immediate window type appears less than expected in all high percentiles for all four logs. The extended type appears less than expected in only two logs, and slightly more than expected in the other two. But the all type appears more than expected in two, and has a uniform distribution in the other two, so it is the best overall choice.

To understand why this happens, consider the graphs in Fig. 12. These show that when using full windows, and any window type (immediate and all are shown), increasing the window size causes a degradation of prediction rate and prediction accuracy. This happens because in many cases data is not available, and the user estimate has to be used. However, all achieves better results, because it is less finicky and uses all available data, thus filling its window faster.

Fig. 11 only shows results for one metric: bounded slowdown. Results for wait time are similar. But there is also another metric to consider, namely predictability. It turns out that this involves a tradeoff. All windows are better for performance, while immediate windows are better for predictability. However, in both cases, the degradation involved in using the other window type is not more than about 0–15%.

Prediction window size. Figure 12 also shows that the actual performance metrics (bounded slowdown and wait time) are largely oblivious to window size. In actuality, 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. However, we have found that small window sizes generally yield good performance.

Prediction type. Another issue is how to calculate the prediction. Four options are to use the average, median, minimum, or maximum of the jobs in the window. The results are shown in Fig. 13. 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 (immediate or full windows), and 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. Our results indicate that relative estimates are much better. However, immediate windows with relative estimates are still not as good as all windows.

Global and local optimum. 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 predictions with a suboptimal parameter values is still significantly better than the original EASY algorithm in terms of both performance metrics. The bottom line is therefore that a default setting can be found that would provide performance benefits, but that even more benefits will be possible if the configuration is tuned to the local workload.

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 fixed 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 outperforming improvements that have so far been achieved by doubling 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 (this will be demonstrated in future publications). 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 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. This aspiration is reinforced by the fact 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 not much better; however, this result is limited to the relatively low loads in the used traces, and it is expected that better predictions will become meaningful for higher loads. This motivates the search for better predictors. In future work we therefore expect to check the performance of more sophisticated predictors that have been suggested in the literature. 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.

8. REFERENCES

[1] 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 vol. 2537.

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

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

[4] D. G. Feitelson, *Experimental Analysis of the Root Causes of Performance Evaluation Results: A Backfilling Case Study*. Technical Report 2002–4, School of Computer Science and Engineering, Hebrew University, Mar 2002.

[5] 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 vol. 949.

[6] 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*, pp. 1–34, Springer Verlag, 1997. LNCS vol. 1291.

[7] 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 vol. 1291.

[8] 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 vol. 2221.

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

[10] 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 vol. 1659.

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

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

[13] D. Lifka, "*The ANL/IBM SP scheduling system*". In *Job Scheduling Strategies for Parallel Processing*, pp. 295–303, Springer-Verlag, 1995. LNCS vol. 949.

[14] 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.

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

[16] D. Perkovic and P. Keleher, "*Randomization, speculation, and adaptation in batch schedulers*". In *Intl. Conf. Supercomputing*, Sep 2000.

[17] 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 vol. 1459.

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

[19] 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 vol. 1659.

[20] 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 vol. 2537.

[21] 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.

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

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

[24] *The dynamics of backfilling*, in preparation.

Survey of Commercial Batch Schedulers

As high performance computing clusters are getting cheaper, they are becoming more accessible. The various clusters are running a host of workload management software suites, that are getting more complex and offer administrators numerous features, scheduling policies, job prioritization schemes, etc.
There is no viable data about the extent to which cluster administrator tune the management software's configuration from its default values. To the best of our knowledge though, most of these high performance computing sites administrators do not stray far from the default configuration of their workload management software — if they even stray from it at all. It seems that only in rare cases do the administrators use the entire spectrum of tunable parameters.
With that in mind, we set off to survey the features concerning batch scheduling of some of the common workload management software suites, focusing on their default settings. To our surprise, we found that the prevalent default scheduler setting is FCFS. In those management suites that allow backfilling, the governing scheme used is EASY [9].

Moab/Maui

The *Moab Workload Manager* [1] is based on the Maui batch scheduler [7], with all its flexibility and added features – backfilling, service factors, resource constraints and weights, fair-share options, direct user/group/account prioritization, target wait times, etc.

However, based on the Maui Scheduler Administrator's Guide [3] its default behavior out of the box is a simple FCFS batch scheduler, with a backfilling policy that maintains a time reservation for the first job in the queue – EASY backfilling. We have verified that fact in the source code [2] of the job priority function (*MJobGetStartPriority(...)* in *MPriority.c*):

In the Maui scheduler, the priority of each job is a weighted sum of several factors, where the weights are set by the administrator. Each factor itself is a weighted sum of sub-factors, whose weights, again, are governed by the administrator. After looking at the source code we found that even though all the factor's weights are set to 1 (in an array called *CWeight*), all the weights of the sub-factors are set to 0, except for that of the job's queue time which is set to 1 (all the sub-factors weights are saved in the *SWeight* array). The result is that the job's queue time is the only factor that is not zero, and even though the factor weights are set to 1, the queue time is the priority function — resulting in a FCFS scheduler.

LoadLeveler

IBM's *LoadLeveler* [8] supports several schedulers, such as FCFS, FCFS with backfilling, gang scheduling, and can also interface with external schedulers. The system also supports checkpointing and restarting of running jobs, and specific *IBM SP* hardware.

Within its own set of schedulers many of the features are tunable: first and foremost, an administrator can rewrite the priority function *SYSPRIO* and use current system data. Examples for such data are a user's class, how many jobs the user/group has in the system, etc. Other parameters can also be used to establish a fair-share priority function. The administrator can also set specific privileged user/group/class accounts. This, coupled with support for job preemption, allows for high priority jobs to preempt low priority ones. At the user level, each user can change the running order (or explicitly specify one) of his own jobs.

Loadleveler supports backfilling, and can even can be tuned to use either the best-fit or first-fit metrics to choose jobs for backfilling.

The default scheduling of LoadLeveler is FCFS: the default priority function is FCFS, as the *SYSPRIO* function is simply the job's queue time. Backfilling is *not* set by default, but when enabled, its policy is first-fit, with time reservation set only for the first job in the queue When using backfilling, users are obligated to specify a runtime estimate for their jobs. When a job exceeds its time estimate, it is killed (sent a SIGKILL signal). This is similar to the EASY backfilling policy.

Load Sharing Facility

Platform's *Load Sharing Facility* (*LSF*) [11, 10] is a comprehensive solution for high performance workload management.

It supports numerous scheduling algorithms, including FCFS, fairshare, preemptive, backfilling and Service Level Agreements level (SLA). LSF can also interface with external schedulers such as Maui. Other features include system supports for automatic and manual checkpoints, migrations, automatic job dependencies and job reruns.

The fair-share scheduler lets the administrator assign shares to users and groups, and set a priority function that divides the resources according to the assigned shares. The shares can also be assigned in a hierarchical manner, so a group can be assigned shares, and divided into subgroups, each getting a percentage of the shares. The final priority function of the fair-share scheduler takes into account the standing shares for the user (either directly of via his group), and the number of running and queued jobs he has.

The Service Level Agreements (SLA) scheduler is a high level scheduler that allows the administrator to state a goal for the system — job deadlines, throughput etc. — without having to tune the lower levels of the scheduler for achieving that goal.

An interesting scheme, called *priority escalation* is also introduced in this software suite. In this scheme, the administrator can set an escalating parameter on a job's priority, so it's priority will increase every time interval – giving much higher priorities to waiting jobs, even when using a fair share scheduler.

Jobs are submitted to queues with different priorities, which the administrator defines. He can also define different scheduling schemes for each queue. Both preemption and backfilling are considered to be queue properties: a queue can be declared to be preemptive, in which case its jobs can preempt running jobs from any lower level queue that is set to be preemptible. Backfilling can be turned on for a queue in the queue configuration file. This flag is not set by default, but the default behavior of processor reservation is similar to EASY – reserve the processor for the first job in the queue when backfilling. Each backfilling queue is assigned a job time limit, which is used if the user did not specify a time limit upon submission. Backfilling queues have their limitations – backfilled jobs cannot be preemptible, as they would consume resource reserved for another job.

If the administrator does not define any queues, a default queue is used, and its scheduling is set to FCFS. The administrator guide [10] is careful to warn that this policy might not be best, and that the site administrator must take that into consideration and define special queues. As mentioned previously, backfilling is not enabled by default, but when enabled, it's default behavior is similar to EASY.

Portable Batch System

The *Portable Batch System* (PBS) comes in two flavors: *OpenPBS* [16] is intended for small clusters, and *PBS-Pro* [15] is the full fledged, industrial strength version (both are descendants of the system described in [6]). For simplicity, we will focus on PBS-Pro.

The suite includes a very versatile scheduler support. Schedulers included with the suite are FCFS, SJF, user/group priorities and fair-share. Also, site specific schedulers can be implemented natively in the C and TCL programming languages, or in a special language called BaSL. Other features include checkpoint support, re-pack and rerun support for failed or stopped jobs, and failed nodes recovery.

The fair-share scheduler uses a similar hierarchical approach, similar to *LSF*. An administrator can distribute shares among groups, whose shares can, in turn, be divided to subgroups. This creates a tree structure in which each node is given shares, which are distributed by administrator assigned ratios to its child nodes, all the way down to the tree leaves. The leaves themselves can be either groups or specific users.

As with other software suites, the administrator can define work queues with various features. Queues can have certain resource limits that are enforced on the jobs they hold. A job can even be queued according to its specified resource requirements — the administrator can define a queue for short jobs, and the queuing mechanism can automatically direct a job with small CPU requirements to the short jobs queue. Of course the administrator can define a priority for each queue, thus setting the dispatch order between queues, or can be selected for dispatch in a round robin fashion. Queues can also be set inactive for certain times, which allow using desktops as part of the cluster at night or holidays.

The *PBS-Pro* system support preemption between different priority jobs. An administrator can define a preemption order between queues, by which jobs from higher priority queues can preempt jobs from lower priority queues if not enough resources are available. Inter-queue preemption is enabled by default, but there is only one default queue.

Being the exception that makes the rule, the default scheduler in both PBS systems is SJF. To prevent starvation (which is the main problem of SJF scheduling), the system can declare a job as starving after some time it has been queued (with the default time set to 24 hours). A starving job has a special status — no job will begin to run until it does. The result begin is that declaring a job as starving causes the system to enter a draining mode, in which it lets running jobs finish until enough resources are available to run the starving job. The starvation prevention mechanism can be enabled only for specific queues.

Backfilling is supported, but only in context of scheduling jobs around a starving job waiting to run, and only if users specify a wall time CPU limit. Like the starvation prevention mechanism, backfilling can also be enabled for specific queues.

As mentioned before, the default scheduler is SJF, and both the starvation prevention mechanism and backfilling enabled for all queues.

Sun Grid Engine

The *Sun Grid Engine* (SGE) [13, 14, 12] is much simpler then its contenders.

SGE has two scheduling policies: FCFS, and an optional administrator set function of Equal-Share scheduler. The latter is a simple fair-share scheduler that tries to distribute resources equally among all users and groups. For example, to overcome a case where a user submits many jobs over a short period of time, its latter jobs will be queued until other users had a chance to run their jobs.

An administrator can also define new job queues, with specific dispatch order among the queues themselves.

Currently, the system does not support backfilling, although this features is planned to be incorporated in future versions [5].

The default behavior is still FCFS, since the default priority function is again the job's queue time.

OSCAR

During this survey we have also enquire about *OSCAR* [4], which is sometimes regarded as a workload management software suite. However, this is more of a cluster installation software, which helps manage the nodes belonging to the cluster, assign them IP addresses, network mounted root file systems, and other resources. The workload management itself is done using one of the aforementioned software suites, mainly *Maui* or *OpenPBS*.

1. REFERENCES

[1] "*MOAB Workload Manager*". http://www.supercluster.org/moab/.

[2] "*MOAB Workload Manager (Maui Scheduler) Source Code*". http://www.supercluster.org/moab/. Version 3.2.6.

[3] Cluster Resources, Inc, *Maui Scheduler Administrator's Guide*. Version 3.2.

[4] B. des Ligneris, S. Scott, T. Naughton, and N. Gorsuch, "*Open Source Cluster Application Resources (OSCAR): Design, Implementation and Interest for the [Computer] Scientific Community*". In *First OSCAR Symposium*, May 2003.

[5] A. Haas, "*Reservation / Preemption / Backfilling in Grid Engine 6.0*". In *2nd Grid Engine Workshop*, Sun Microsystems GmbH, Sep 2003.

[6] R. L. Henderson, "*Job Scheduling Under the Portable Batch System*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 279–294, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[7] D. Jackson, Q. Snell, , and M. Clement, "*Core Algorithms of the Maui Scheduler*". In *Job Scheduling Strategies for Parallel Processing*, Springer-Verlag, 2001.

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

[9] 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.

[10] Platform Computing Inc., *Administering Platform LSF*. Jan 2004. www.platform.com/services/support/docs_home.asp.

[11] Platform Computing Inc., "*Platform LSF*". http://www.platform.com/products/LSFfamily/.

[12] Sun Microsystems, Inc., *N1 Grid Engine 6 Administration Guide*. 2004.

[13] Sun Microsystems, Inc., "*Sun Grid Engine*". http://gridengine.sunsource.net/, 2004.

[14] Sun Microsystems, Inc., *Sun ONE Grid Engine Enterprise Edition Administration and User's Guide*. 2002. version 5.3.

[15] Altair Grid Technologies, *PBS Pro Administrator Guide 5.4*. 2004. Editor: James Patton Jones.

[16] Veridian Systems, *Portable Batch System, Administrator Guide*. 2000. OpenPBS Release 2.3.