Session-Based, Estimation-less, and Information-less Runtime Prediction Algorithms for Parallel and Grid Job Scheduling

David Talby Dan Tsafrir Zviki Goldberg Dror G. Feitelson
Hebrew University of Jerusalem, Israel
Email: {davidt,dants,zviki,feit}@cs.huji.ac.il

Abstract

The default setting of most production parallel job schedulers is FCFS with backfilling. Under this setting, users must supply job runtime estimates, which are known as being highly inaccurate and inferior to system generated predictions. Recent research revealed how to utilize system predictions for backfilling, and this potential performance gain motivates searching for better prediction techniques. We present three prediction techniques using decreasing levels of information as is suitable for the situation at hand. The first is based on "user sessions": continuous temporal periods of per-user work. This algorithm exploits the entire long-term historical data of the workload, along with user runtime estimates. The second is "estimation-less", that is, uses historical data only, relieving users from the annoying need to supply estimates. The third is completely "information-less" and is suitable for cases in which neither historical information nor estimates are available, as happens in some grid environments. We evaluate the algorithms by simulating real data from production systems. We find all of them to be successful in terms of both accuracy and performance.

1. Introduction

The scheduler is a key component in determining the overall performance of a parallel system. Improving it can have a dramatic visible effect on a system's usability. The most dominant parallel scheduling algorithm to date [3] is *EASY*, that is, FCFS (First-Come First-Served) with backfilling. Backfilling is an optimization that allows small jobs to execute ahead of their time provided they do not delay the first queued job [12]. This simple optimization is known to be extremely effective: it dramatically improves utilization [6] and yields performance which is comparable to that of more sophisticated algorithms that utilize preemption and migration [17].

Upon submittal, backfilling mandates users to estimate how long their jobs will run and bases its scheduling decisions on this information. It is well known that user estimates are highly inaccurate [8, 10, 16] and that it's possible for system-generated predictions that are based on users history to do a far better job [4, 9, 10, 14]. Nevertheless, system predictions were never incorporated into production backfilling schedulers due to two reasons: First, several studies have shown that making user estimates even less accurate, for example by doubling them, actually improves performance [10, 19]. And second, there was no clear way of handling the case of job runtimes that exceed system-generated predictions.

In a recent study [15] we have managed to overcome the technical difficulty presented by too-short predictions, which made it possible to incorporate predictions within backfilling schedulers. We have also shown that doubling of estimates improves performance because it implicitly nudges the system to favor shorter and smaller jobs at the expense of longer and bigger jobs, effectively trading off fairness for performance. We have further shown that both performance and fairness can benefit if (1) estimates are improved instead of doubled, (2) shorter jobs are explicitly backfilled first (SJBF), and (3) the base algorithm remains FCFS, that is, non-backfilled jobs are scheduled by arrival time, and backfilling is allowed only when it doesn't interfere.

The vast popularity of backfilling among production systems, the new ability to make backfilling utilize system predictions, and the significant resulting improvement in accuracy and performance, provide strong incentive for improving prediction techniques. Additional motivation is provided by the fact that improved accuracy is becoming a design goal by itself for schedulers, in particular in grid environments which utilize co-scheduling [13]. Finally, the ability to use system-predictions for backfilling makes it theoretically possible (and rather tempting) to eliminate user estimates altogether. This is highly desirable as it simplifies job submittal, and rids users from an annoying task which they perform poorly [8].

Considering the motivation as presented above, our effort in this paper is threefold. We wish to devise three prediction algorithms that utilize a decreasing amount of information as is suitable for the situation in question:

1. **Session based algorithm**. This algorithm uses all the information available to traditional backfill schedulers, namely accumulated historical data about users and the user runtime estimates of submitted jobs. It expands the accumulated art [2, 4, 9, 14], and its novelty is that it is based on *user sessions* — identifying consecutive temporal work periods of users and basing the predictions on the jobs populating these sessions.
2. **Estimation-less algorithm**. The next step we take is refining the above algorithm such that it ceases using user estimates. Once this is established and incorporated into a backfill scheduler, users are able to freely submit jobs, without estimating how long they will run. Comparing results obtained using this predictor relative to the previous one reveals the true value of information of user estimates.
3. **Information-less algorithm**. This predictor does not base its predictions on any past or present information. Such a predictor is often required in the context of grid, where a local scheduler must schedule a remote job. The history of the job's user will usually not be available, since the user's jobs ran on multiple machines, and the available job attributes may vary according to the grid's protocols.

We have embodied these prediction algorithms in a backfill scheduler. This combination provides a complete, highly practical scheduling solution that improves the bottom-line performance of current systems, in addition to improved accuracy. We provide empirical measures of performance and accuracy. In addition, this solution is usable in systems which serve a mix of the above three situations – for example, a local system also serving some remote requests, or featuring optional user estimate – although we do not provide empirical results here, due to the lack of a representative workload.

The rest of this paper is organized as follows. After providing necessary background and surveying related work (Section 2), we suggest a methodology for comparing predictors in terms of accuracy and performance (Section 3). Section 4 uses this methodology to provide reference on some previously suggested prediction algorithms. Sections 5, 6 and 7 deal with the three predictors noted above, respectively. Finally, Section 8 concludes.

2. Background and Motivation

2.1. Backfilling

The parallel computers considered in this paper are of the most widespread type today, which use *variable partitioning*. Upon submittal, a user specifies the number of processors the job requires. The job is then placed in a queue until enough resources are available; then, it is assigned the processors it needs for its exclusive use and runs to completion. As stated earlier, most parallel job schedulers today – including IBM's LoadLeveler, Maui [5], Moab and others – use EASY as their default settings [3]. The backfilling optimization employed by EASY allows jobs to leapfrog over the first queued job provided they do not delay it, i.e. run beyond the *shadow time*. This is illustrated in Figure 1.

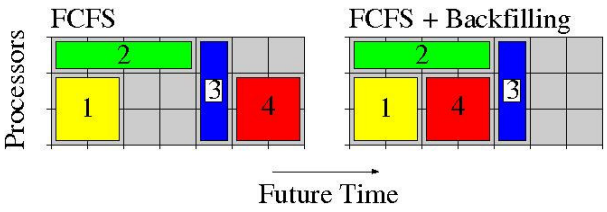

Figure 1: *EASY scheduling reduces fragmentation by using backfilling. The numbers indicate jobs' arrival order. It would be impossible to backfill job 4 had its length been more than 2 time units, or else 3 would have been delayed.*

2.2. Inaccuracy of User Estimates

The popularity of EASY has enabled empirical studies about the quality of user estimates, based on accounting logs from multiple installations [11]. These show estimates are generally inaccurate [10, 15, 16], as reproduced in Figure 2.

The seemingly promising peak at 100% unfortunately reflects jobs that reached their allocated time and were then killed by the system. The hump near zero reflects jobs that failed on startup. The rest of the jobs, that actually ran successfully, have a rather flat uniform-like histogram – meaning that for such jobs, any level of accuracy is almost equally likely.

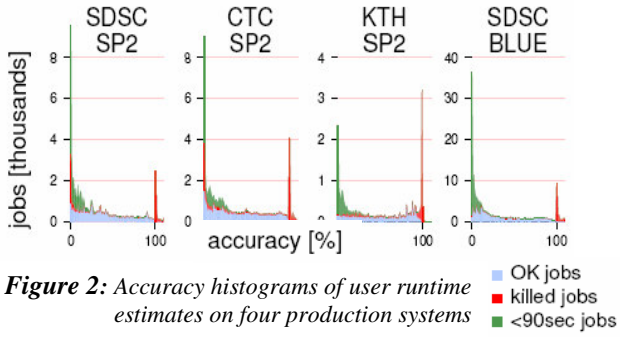

Figure 2: *Accuracy histograms of user runtime estimates on four production systems*

Thus user estimates are usually 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 enable the scheduler to perform better packing. However, a recent study [8] indicates users are quite confident of their estimates, and probably would not be able to do better.

2.3. Using Predictors to Improve Performance

Until recently, motivation for devising runtime prediction algorithms and incorporating them into production schedulers was limited for two reasons: Findings which suggested that accuracy hinders performance, instead of improving it; and lack of other needs for accuracy. Both of these reasons have been recently refuted.

Several studies [10, 19] have found a surprising yet consistent result: EASY performs better if estimates are doubled, which means performance is actually improved if estimates are made even *less accurate*. This is shown in Figure 3 by bars associated with user estimates allegedly suggesting improved accuracy negates performance. However, in [15] we have shown this argument to be faulty, as is demonstrated by bars associated with actual runtimes in Figure 3. Indeed, doubling improves performance. But this argument can be applied with far greater success if the initial estimates are better, as it is evident that doubling of perfect estimates outperforms the doubling of user estimates. It therefore reasonable to expect, and was verified in [15], that doubling of high quality system predictions will have a similar effect.

Doubling improves performance by creating a dynamics in which the first queued job is delayed beyond its designated start-time due to shorter jobs that are implicitly prioritized. This effect can be partially emulated without this delay, if instead of doubling, shorter jobs are explicitly backfilled first: Shortest-Job-Backfilled-First scheduling (SJBF) [15].

In addition to being inaccurate, estimates embody one additional characteristic that is particularly harmful for backfilling performance: they are inherently modal [16]. Users tend to choose "round" values – for example, one hour – as estimates, resulting in 90% of the jobs repeatedly using the same 20 values. Such modality significantly limits the scheduler's ability to exploit existing holes in the schedule. Most notable is the maximal allowed value which is always very popular, despite the fact that jobs which use it are usually short. For example, in the CTC trace roughly a quarter of the jobs used this value. Naturally, such jobs will never be backfilled, hindering overall performance.

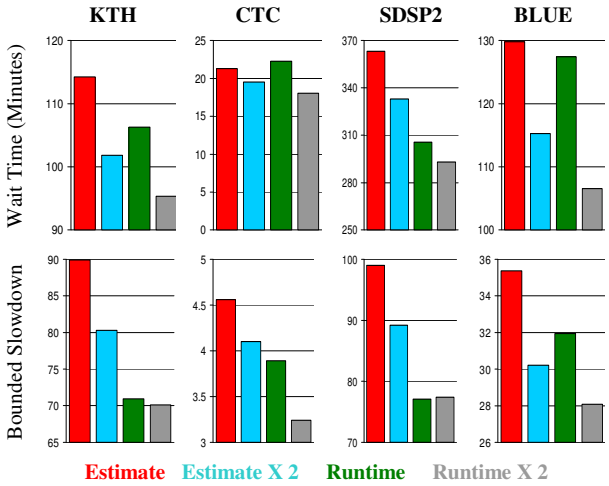

Figure 3: *Performance of EASY under different runtime predictions: user estimates, actual runtimes, and both doubled*

2.4. Using Predictors to Improve Accuracy

In addition for the motivation to improve performance, two other developments raise the requirement for accurate predictions of parallel jobs. The first is advanced reservation for grid co-allocation, shown to considerably benefit from better accuracy [13]. Knowing in advance when remote processors will be available is crucial for making reservations work.

The second is scheduling of moldable jobs that can utilize any number of processors [2, 14]. For such jobs the goal of the scheduler is to minimize response time, thus it must decide whether it's preferable to start running a job on the processors that are available now, or wait for more processors to accumulate. A good prediction of how long current jobs are expected to run is obviously crucial to making the correct decision.

2.5. Related Work

Several runtime prediction schemes have been published. Smith et al. suggested utilizing genetic algorithms to dynamically determine the set of jobs according to which predictions will be made [14]. Gibbons used fixed set criteria, but his predictor interface was complicated by varying confidence levels [4]. Kapadia et al. used instance based learning [7]. All these predictors are much more complex than the basic EASY scheduler; in contrast, in [15] we've shown that a simple prediction of averaging the runtime of the last two jobs by the same user is extremely successful.

In this paper we also propose that predictors will obey a standardized interface (Listing 1). This will allow schedulers to easily incorporate any previously implemented predictors, regardless of their complexity.

3. Predictors Comparison Framework

This section defines our methodological framework for comparing prediction algorithms: The scheduling algorithm which hosts predictors, the metrics used to measure performance and accuracy, and the empirical data set.

3.1. Standardized Predictor Interface

A scheduler is an event-based program, where events are job arrival or termination. Upon arrival, the scheduler is informed of the number of processors the job needs and its user estimated runtime. It can then simulate the job's execution or place it in a queue. Upon termination, the scheduler is notified and can schedule other queued jobs on free processors.

A runtime prediction algorithm is also an event-based program, used internally by a scheduler to handle four events as specified in Listing 1.

```
struct Prediction { Job job, int prediction }
list<Prediction> onJobArrival ( Job job )
list<Prediction> onJobStart ( Job job )
list<Prediction> onJobTermination ( Job job )
list<Prediction> onJobDeadlineMissed ( Job job )
```

Listing 1: *An abstract predictor standardized interface*

In each of these events, more information is available to the predictor, so it may decide to adjust the predictions of zero or more jobs (whether waiting or running). It therefore returns a list of updated predictions for these jobs, which may be empty. The only constraints are that a prediction must be given for a job on its arrival, must be given when a job has missed its deadline (meaning that it's still running, but its runtime has just exceeded its current prediction), and must be bigger than the job's current runtime, if it already began to run.

The simplest predictor would return a job's estimate upon its arrival and an empty list upon its start and termination (there will be no deadline misses, because a job is killed once its user estimate is reached). Note that this is actually plain EASY scheduling. A scheduler that uses the above interface would be able to quickly replace and utilize any new predictor that obeys this interface, as are all predictors appearing in this paper.

3.2. Measuring Performance

The effect of a given predictor on a given system is highly dependent on the scheduler being used. Selecting the scheduler used for empirical experiments in this paper was based on the following criteria:

1. The scheduler's performance should improve when the predictor's accuracy is improved.
2. The scheduler should maintain accepted fairness norms, and not be subject to starvation.
3. The scheduler should be practical – easy to implement and integrate in current systems.

Evidence shows [15] that a number of schedulers abide these criteria. These schedulers include those which employ the "doubling optimization", since as discussed earlier choosing whether to double the prediction or not is a property of the scheduler, not the predictor. Out of the possible reasonable choices, we have chosen to use Shortest-Job-Backfilled-First (SJBF) as the base scheduler. It is identical to EASY, except for three places:

1. Use predictions instead of user estimates to compute the shadow time.
2. Use predictions instead of estimates to test if a job terminates before the shadow time.
3. Backfill jobs in order of ascending predictions (shortest job first), instead of ascending arrival time (first come first serve).

SJBF was chosen for three reasons. First, it explicitly prioritizes shorter jobs, which makes the benefit of improved accuracy most pronounced [1,4,7,14,15,19]. Second, it does so without scarifying fairness. And third, it is very easy to implement in practice, particularly in EASY or Maui based installations.

Performance is measured in terms of average wait time of jobs and bounded slowdown [10], as is customary in this domain.

3.3. Measuring Accuracy

Defining and measuring the accuracy of predictions is non-trivial for several reasons. First, since a prediction may be lower than a job's actual runtime, it may have to be changed during its execution – so one job can have several different predictions during its lifetime. Second, since most prediction algorithms rely on history, they must rely on the list of terminated and running jobs – which means that their accuracy also depends on the scheduler being used, in addition to the workload. Third, different metrics may be sensitive to different aspects of accuracy, as is the case in performance metrics. For example, the slowdown and wait time metrics don't always agree, as the former is dominated by short jobs and the latter by long ones. This is why more than one metric is often required.

We therefore define and measure two accuracy metrics: *absolute inaccuracy*, defined as the absolute difference between the prediction and the actual

runtime (analogous to wait-time and similarly desired to be as small as possible), and *relative accuracy*, defined as the ratio of runtime to prediction (analogous to slowdown). When analyzing a full log, we consider the averages of these metrics, which means relative accuracy is dominated by short jobs, while absolute inaccuracy is dominated by large prediction errors. To avoid under- and over-predictions canceling each other out when relative accuracy is averaged, we always set the smaller value to be the numerator in the ratio, and so, relative accuracy is always within 0% to 100%. Equation 1 formally defines the two metrics.

$$AI = |R - P| \qquad RA = \begin{cases} 1 & if \quad R = P \\ R/P & if \quad R < P \\ P/R & if \quad R > P \end{cases}$$

Absolute Inaccuracy (AI) *Relative Accuracy (RA)*

Equation 1. *Accuracy metrics. R stands for the job's actual runtime and P for its associated prediction.*

These definitions only work for jobs that have a single prediction throughout their lifetime. In the frequent case of under-predictions (and upon any other prediction update initiated by the predictor), jobs' predictions change. We therefore use its average weighted accuracy. Weights are determined according to the durations in which each prediction was in effect. Formally, if T_0 and T_N are a job's submission and termination time, and A_i is its accuracy (absolute or relative) from time T_{i-1} to T_i (where $T_i \leq T_{i+1}$), then its average weighted accuracy is:

$$A = \frac{1}{(T_N - T_0)} \cdot \sum_{i=1}^{N} A_i \cdot (T_i - T_{i-1})$$

Equation 2. *Accuracy in the face of multiple predictions.*

3.4. The Dataset

The dataset used in this study consists of four logs from the Parallel Workloads Archive [11]. Together, the logs constitute over 400,000 jobs and six years of real user activity, under different load conditions in different sites (Table 1). All logs have been studied before, and undergone extensive cleaning from errors, flurries and other minor issues. They are widely regarded as representative of parallel computer workloads, including their user estimates [16].

Abbrev.	Location	Nodes	Jobs	Weeks	Utilization
SDSP2	San Diego SC	128	59,725	105.2	84%
CTC	Cornell Theory Center	128	77,222	48.5	56%
KTH	Swedish Inst. of Technology	100	28,490	48.6	69%
BLUE	San Diego SC Blue Horizon	1,152	243,314	140.1	76%

Table 1: *Parallel logs used in this study.*

4. Reference Predictors

Now that our methodology is in place, we begin by studying four existing prediction algorithms, whose metrics will be used as reference from now on. The first algorithm is the perfect predictor, which guesses the actual runtime. This is a theoretical, optimal predictor, since the actual runtime is not known in advance. The second predictor is the constant predictor: predict the same, constant runtime for all jobs. This predictor is a reference to what can be achieved with no information, in contrast to Perfect which has full information. The third predictor is the estimate predictor: use the user estimate as the prediction. Formally:

PERFECT Predictor: OnJobArrival return <job, job.runtime>
CONSTANT Predictor: OnJobArrival return <job, *Constant*>
ESTIMATE Predictor: OnJobArrival return <job, job.estimate>

Listing 2. *Reference Predictors*

Note that SJBF with the Estimate Predictor is different from the original EASY in two aspects. First, it backfills jobs by order of ascending estimates, instead of ascending arrival time (FCFS). Second, it updates its prediction on deadlines misses – events in which the job's runtime exceeds its prediction. This can happen if a job's runtime exceeds its estimate (which happens in practice in rare cases), and is handled, as suggested in [15], by gradually increasing the prediction by predefined increasing values. This is also the strategy used in the Constant Predictor, if the runtime exceeds the constant. All other predictors in this paper use the following strategy: If a deadline is missed and the current prediction is smaller than the user estimate, then raise it to be equal to the user estimate; else, raise it by the predefined gradual increments.

The fourth predictor we will compare ourselves to is the Recent User History Predictor, presented in [15] and named EASY++ there. To the best of our knowledge, it is the best performing scheduler/predictor combination to date abiding our criteria from section 3.2. It is based on SJBF as well, and predicts runtimes based on a simple rule: A job's prediction is the median of the runtimes of the three last jobs of the same user.

OnJobArrival:
if *there exist at least three terminated jobs of job's user* then
  return [job, median of last three terminated jobs of *job*'s user]
else
  return [job, job.estimate]

Listing 3. *Recent User History (RUH) Predictor*

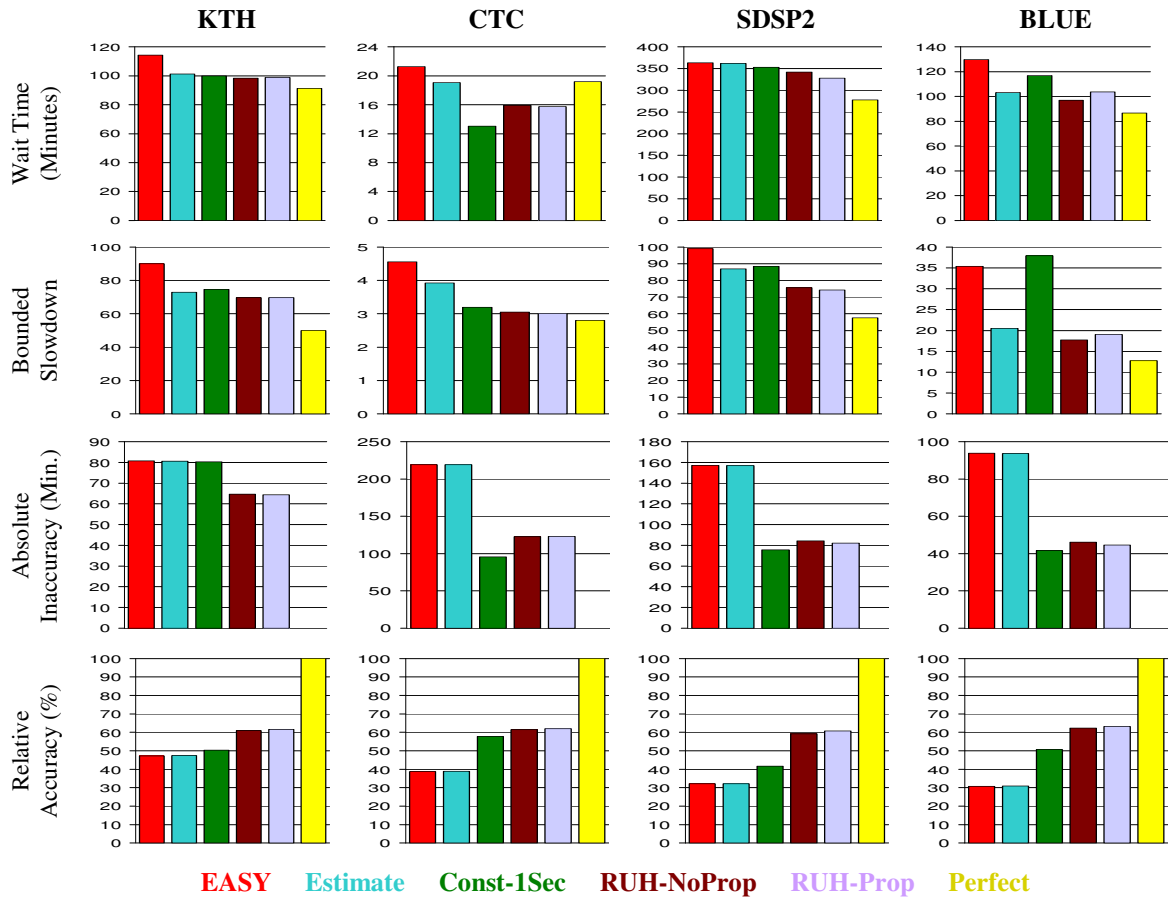

Figure 4: *Performance and Accuracy Comparison of Reference Predictors*

The RUH Predictor can be configured by several parameters, underlined in the above pseudo-code: how many past jobs to consider, which metric to use on them, should running or only terminated jobs be considered, and how to handle the first jobs of a new user. We have simulated it under its best published parameter configuration, in two variants – with and without propagation. Propagation is an optional optimization, which forwards new information on a job – specifically, its actual runtime when it terminates or its updated prediction on a deadline miss event – to the rest of the waiting and running jobs of the same user. The predictions of these jobs are then recomputed based on the updated information.

Table 2 and Figure 4 compare the reference predictors. As expected, EASY is substantially inferior to SJBF, even under the Estimate predictor. RUH behaves almost the same with and without propagation, and improves both performance and accuracy further, up to about two thirds of the potential gain, as assessed by the results of the Perfect predictor. The result for the Constant predictor will be discussed later.

Predictor:	**Improvement over EASY, Average over all logs:**			
	Wait Time	Bounded Slowdown	Absolute Inaccuracy	Relative Accuracy
Perfect	22%	47%	100%	176%
Estimate	11%	22%	0%	0%
RUH without Propagation	18%	32%	40%	69%
RUH with Propagation	17%	32%	41%	71%

Table 2. *Performance and accuracy gain of SJBF with reference predictors over EASY*

5. Session-Based Prediction

5.1. Rationale

It is well known that human users typically work in sessions – periods of intense, repetitive work. A recent study [18] formalized this notion, and identified five stable clusters of sessions in parallel workloads. It was also found that in four of the session clusters, consisting of over 95% of the observed sessions, the variance between jobs in the session is very small – a median of

CTC, 20 Sep 1996, User #289					BLUE, 21 Apr 2001, User #315				
Arrival	Size	Est.	Exe#	RT	Arrival	Size	Est	Exe#	RT
19:49:20	16	300	3012	91	04:29:56	8	30600	N/A	31
19:52:32	12	300	3012	67	04:30:14	8	30600	N/A	28
19:57:45	12	300	3012	348	04:33:16	8	30600	N/A	31
19:58:33	16	300	3012	342	04:36:26	8	30600	N/A	28
20:05:25	16	300	3012	105	04:37:09	8	30600	N/A	43
20:08:17	16	300	3012	87	04:40:33	8	30600	N/A	33
20:10:07	200	300	3032	332	04:43:29	8	30600	N/A	29
20:15:54	100	300	3033	314	04:46:34	8	30600	N/A	29
20:22:15	200	300	3033	389	04:49:40	8	30600	N/A	28
20:31:34	200	1800	3033	168	04:54:37	8	30600	N/A	28
20:31:45	200	1800	3033	352	04:58:04	8	30600	N/A	30
20:33:25	200	1800	3033	348	05:06:57	8	30600	N/A	28

Table 3. *Two Sample Sessions*

less than two unique runtimes and levels of parallelisms within a session. This explains why the Recent User History Predictor is successful, but also suggests that basing a predictor directly on sessions may be even better.

Table 3 shows two stereotypical sessions. On the right is a session from the BLUE log, in which the user repeatedly ran the same job; note how runtimes are easy to predict based on history, but on the other hand have no correlation to the user estimate. On the left is a session from the CTC log, which shows how sometimes the user hints about changes in runtime by changing the number of processors, the estimate, or the executable. Based on these observations, we designed the session-based predictor to use both proximity in time and similarity in job attributes to decide on which jobs a new prediction should be based.

5.2. Algorithm

The Session-Based History (SBH) Predictor works as follows. It maintains each user's past jobs partitioned by sessions, where two jobs are defined to be in the same session if the *think time* between them (the time between the termination of the first one and the arrival of the next) is smaller than twenty minutes. This threshold is taken from [18], where sensitivity analysis showed that it is stable to changes within the same order of magnitude. We have run simulations using different thresholds, including ones far different from 20 minutes, and concluded that other values sometimes show comparable performance, but cannot be used to obtain consistent superior performance or accuracy. These results will not be presented here due to lack of space. All simulation results presented here use the 20-minutes value.

In addition, the SBH Predictor requires an ordered list of similarity criteria, by which jobs in a session are matched to the job for which a prediction is required. Each criterion defines whether the number of processors (P), the user estimate (E), or the executable

```
OnJobArrival (Job job):
    for each criterion in the similarity criteria list
        for each of the last three sessions of job's user,
        ordered by descending start time
            if there exists at least one terminated job in session,
            which matches the current similarity criterion, then
                return <job, median of all matching jobs in session>
    // (following line is only reached if no match was found)
    return <job, job.estimate>
```

Listing 4. *Session Based History (SBH) Predictor*

(X) should match; the algorithm only uses jobs that match the given criterion to generate a prediction. For example, if the criteria list is [PEX,PX,EX], then the algorithm will first look only for jobs that match the new job in parallelism, estimate, and executable; if there is no such job in the current session, it will look for jobs that match in parallelism and executable; and if there is no such job as well, it will look for jobs that match in estimate and executable. If no matching job is found at all, then the algorithm repeats the search in the previous sessions, in descending order. If no matching job is found in any session (for example, in the first job of a new user, or when a user starts working with a new executable), the algorithm resorts to using the user estimate as the prediction (Listing 4).

The algorithm can be configured in several ways, which are underlined in the above pseudo-code. We found two parameters to be the most influential. The first is the similarity criteria list, since it defines the balance between predicting by exact matches, and not being able to predict at all (when the criteria is too strict). The second is the order of the algorithm's two loops. The algorithm as explained above and defined in Listing 4 uses "Depth-First Search" (DFS): Its first priority is to find an exact match to the current criterion, at the expense of relying on the farther past. For example, if the setting is [PEX,PX,EX], then this algorithm will prefer to predict based on a job that matches in parallelism, estimate, and executable three sessions ago, rather than predict based on a job that matches only in parallelism and executable from the current session. An alternative strategy would be "Breadth-First Search", in which the two loops are exchanged, and the algorithm first looks at the current session for any match to the similarity criteria list, and only searches past sessions if no match is found.

5.3. Empirical Results

The SBH Predictor can be configured in a vast number of ways, but most parameters and combinations have a marginal effect. Table 4 and Figure 5 compare the performance and accuracy of RUH with the two best SBH configurations we have found. Note that the numbers indicate improvement over the best RUH

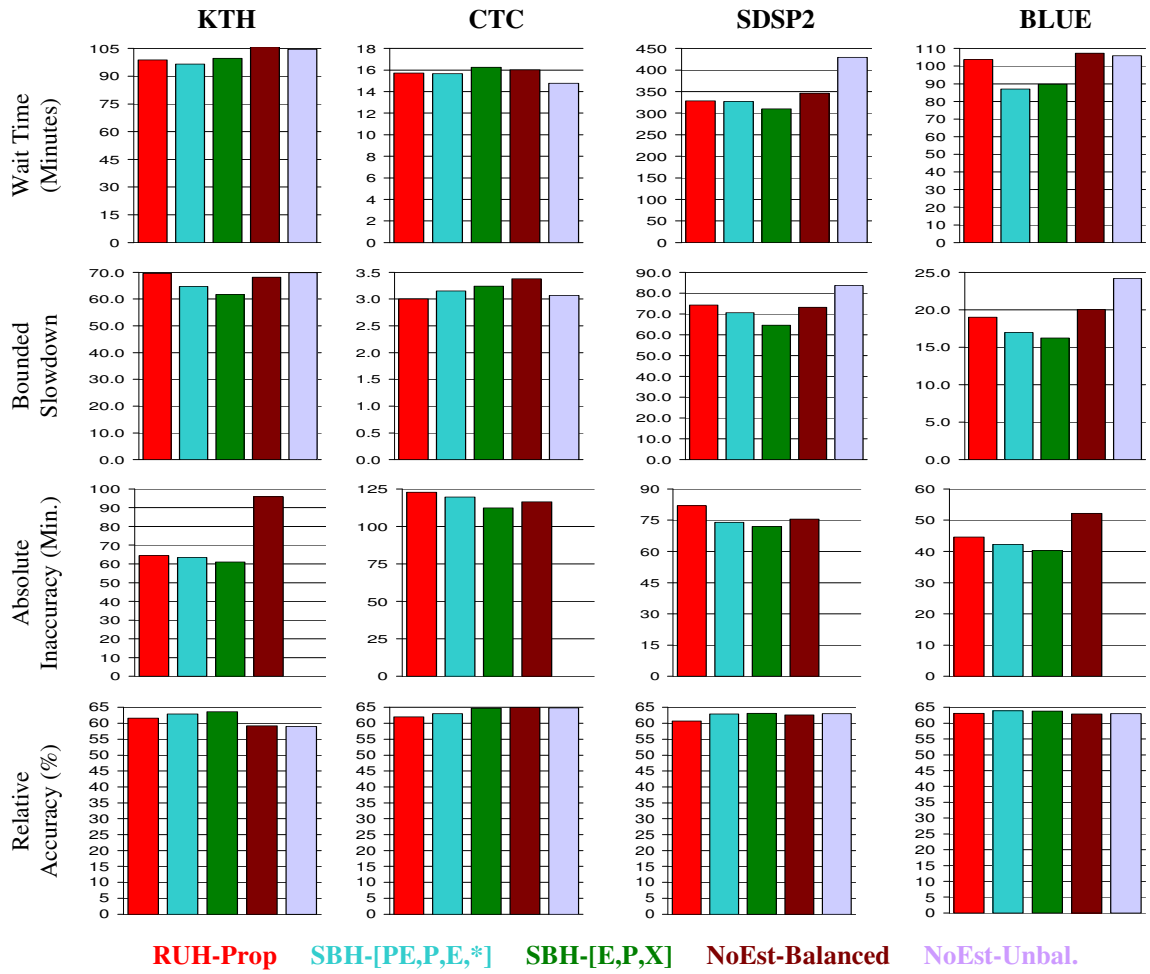

Figure 5: *Performance and Accuracy Comparison of RUH, SBH and No-Estimate Predictors*

configuration. This translates to a 24% gain in wait time and 47%-53% in absolute inaccuracy over the currently deployed EASY-based schedulers.

After analyzing thousands of simulations, we uncovered several other interesting properties of the session-based predictor, summarized in the next paragraphs. These insights may be applicable to the design of other predictors, as well as for session-based analysis of parallel workloads for other reasons.

Configuration:	**Improvement over RUH, Average over all logs:**			
	Wait Time	Bounded Slowdown	Absolute Inaccuracy	Relative Accuracy
Unlimited DFS, [PE,P,E,*], Propagation	5%	4%	5%	2%
Unlimited DFS, [E,P,X], Propagation	4%	8%	9%	3%

Table 4. *Best SBH configurations and their gain over RUH*

Searching in sessions far back in history is useful. In contract to the RUH Predictor, SBH prediction requires a users' entire history to work optimally. Limiting the algorithm to look for less than 10 sessions back deteriorates performance, and surprisingly, even limiting the algorithm to 30 sessions results in performance that is 2%-3% worse, in all logs, than that of the equivalent unlimited configuration. Only a small fraction of a system's users even have that many sessions, so the effect must be caused by better understanding of these few extremely active users.

Depth-First Search is significantly better than Breadth-First Search. Over a large set of simulations, DFS performed better than the equivalent BFS configuration in over 90% of the cases. The average gap in favor of DFS, over all four logs, was 5% in average wait time and 17% in average absolute error. Note that this happens even though we use DFS to *unlimited depth* (not just 3 sessions back). In plain words, basing a prediction on an exact match that is

five months old is better than basing it on an approximate match that is five minutes old. According to the previous paragraph, this happens in practice.

Session-Based Prediction based on proximity in time alone in not enough. Using similarity criteria such as [*] (match any job) or [PEX,*] resulted in performance that is even worse than that of RUH. It seems that separating sessions by means of the think time between jobs alone is not enough to distinguish between different activities of the user, at the level of precision that is required for accurate runtime prediction.

Generalizing the above three paragraphs, we safely conclude that *exact similarity is more important than proximity in time* for runtime prediction. Thanks to this principle, the session-based predictor is the first to successfully use to full history of a parallel workload.

Propagation is generally better than non-propagation. Enabling propagation means taking advantage new information sooner, and indeed using this optimization was preferable in 75% of the cases, resulting in an average improvement (over all four logs) of 10% in average wait time and 6% in average absolute error.

6. Estimation-Less Prediction

The Session-Based Predictor uses all the available information about a job to predict its runtime. In this section we present predictors, session-based as well, for the case in which user estimates are not available. This can be used to relieve users from the need to supply estimates, as well as in other situations where estimates are simply not available, or not available for all jobs.

Session-Based Predictors enable us to build the highest performing predictors to date that don't use user estimates, since SBH relies on estimates less than previous algorithms. By measuring the performance and accuracy loss caused by discarding estimates, we enable making an informed decision. Building a No-Estimate SBH Predictor requires replacing the estimates in the three places where they are potentially used:

1. As a default prediction in case no matching past job is found (the last line in Listing 4 is reached). This happens in slightly less than 1% of the jobs in the predictors given in this section. The estimate can be replaced here by always predicting 1 second (i.e. applying the Constant predictor).
2. As a strategy for dealing with missed deadlines, as long as the job's current runtime is smaller than its estimate. The way to replace the estimate here (as we found empirically) is by multiplying the current prediction by a factor of 10 on every miss.

Configuration:	**Improvement over RUH, Average over all logs:**			
	Wait Time	Bounded Slowdown	Absolute Inaccuracy	Relative Accuracy
Balanced, DFS, [PX,P,X,*]	-5%	-4%	-13%	1%
Unbalanced, DFS, [PX,P,X,*]	-8%	-11%	N/A	1%

Table 5. *Best No-Estimate SBH Configurations and Their Performance Relative to RUH*

3. As a similarity criterion for matching past jobs to the predicted job. Obviously, Estimates cannot be used in similarity criteria lists if they are not available.

Table 5 and Figure 5 present two high-performing SBH variants that do not use estimates. Both variants don't use propagation, and using it has a negligible effect. The "Balanced" variant stops the exponential growth of predictions after it hits a certain (high) threshold, while the "Unbalanced" variant does not. Not stopping the exponential growth results in better performance in three of the four logs. However, this comes at the expense of an out-of-bounds absolute error, and also at some cost to fairness, since the high over-estimations that are inevitably created enable more backfilling of short jobs (that is, predicted-to-be-short jobs), at the expense of long ones.

The empirical results show that the best configuration of SBH without user estimate is still not as good as RUH. Compared to the best SBH predictor with user estimates, the average wait time is about 10% worse, and the average absolute inaccuracy is 22% higher. So the bottom line is, as expected, that requiring users to indicate estimates for submitted jobs is still useful when predictors are used. On the other hand, if user estimates are not available, the above predictors are the best alternative to date.

7. Information-Less Prediction

The problem of runtime prediction with no information is highly practical in grids and other distributed co-scheduled machines, where a single user session may be executed on several separate computers, and so the history of an incoming job may not be available to a local scheduler. The remote scheduling protocol may not even transfer original user identifiers or estimates, let alone complete histories.

Information-less predictors are also interesting since they help to assess the value of information that the more informed predictors enjoy, in terms of the difference in performance and accuracy between them. This can aid in

Predictor:	Improvement over EASY, Average over all logs:			
	Wait Time	Bounded Slowdown	Absolute Inaccuracy	Relative Accuracy
Constant 1 Second	16%	13%	41%	37%
RUH with Propagation	17%	32%	41%	71%

Table 6. *Performance and accuracy gain of Constant*

deciding whether collecting the extra information, which requires effort in grid-like systems, is worthwhile.

The Constant predictor (defined in section 4) using a constant of one second, provides a very surprising result. Although it uses *no information* to make predictions – not the job's user estimate, not past jobs, not other attributes of the job – it succeeds to match the performance of RUH in the wait time and absolute error metrics, and significantly outperform EASY in the two other metrics. Note that the accuracy is achieved here by first guessing one second, and gradually increasing the prediction as the job runs longer. The initial low guess enables to backfill any job, regardless of its attributes – and the algorithm indeed backfills on average 22% more jobs than EASY. However, this is not done at the expense of long jobs (since all jobs have an equal chance to backfill). Also, the gradual ascent in prediction during runtime maintains a small gap between a job's runtime and its reserved shadow time, which helps in maintaining fairness as well.

Constant values higher than 1 second produce worse results, which become far worse as the constant increases. This happens because the scheduler can't differentiate between short and long waiting jobs (since all jobs have the same initial prediction), and it's harder to backfill waiting jobs, since they are longer and thus require longer shadow times to be able to backfill.

The algorithms in the previous two sections also fit a mixed situation, in which some jobs have full historical data while others do not (as in a grid setting). However, this was not simulated, due to the lack of a relevant workload.

8. Summary

This paper makes three contributions. First, it expands the current knowledge on runtime prediction of parallel jobs, by introducing the concept of user sessions and illustrating its effectiveness. The session concept is shown to be valuable in both the full information case and the estimation-less case.

Second, this paper addresses the issues of estimation-less and information-less runtime prediction, presenting and evaluating predictors for both cases. In addition to being useful by themselves, these predictors help in appraising the potential benefit in the extra information. This is an interesting question since this potential benefit must be weighted against the effort that is required to collect the information – either from users, or from a grid's infrastructure.

Third, this paper presents a complete scheduling framework for the use and evaluation of predictors, targeted towards the most popular parallel computer architecture in use today. This framework can be used either by researchers, to evaluate other predictors under the same standardized interface and metrics set, or by practitioners, as a complete and practical scheduling solution that can be used to improve the bottom-line performance and accuracy of current systems.

9. 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, LNCS vol. 2537, pp. 103-127, 2002.

[2] A. B. Downey, "Predicting queue times on space-sharing parallel computers". In 11th Intl. Parallel Processing Symp., pp. 209.218, Apr 1997.

[3] Y. Etsion and D. Tsafrir, "A Short Survey of Commercial Cluster Batch Schedulers". Technical Report 2005-13, School of Computer Science and Engineering, Hebrew University of Jerusalem, May 2005.

[4] R. Gibbons, "A historical application profiler for use by parallel schedulers". In Job Scheduling Strategies for Parallel Processing, LNCS vol. 1291, pp. 58.77, 1997.

[5] D. Jackson, Q. Snell, and M. J. Clement, "Core algorithms of the Maui scheduler". In Job Scheduling Strategies for Parallel Processing '01, pp. 87–102, 2001.

[6] J. P. Jones and B. Nitzberg, "Scheduling for parallel supercomputing: a historical perspective of achievable utilization". Job Scheduling Strategies for Parallel Processing '99, LNCS vol. 1659, pp. 1-16.

[7] N. H. Kapadia, J. A. B. Fortes, and C. E. Brodley. Predictive application-performance modeling in a computational grid environment. In IEEE International Symposium for High Performance Distributed Computing (HPDC), 1999.

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

[9] Hui Li, D. Groep, J. Templon, and L. Wolters, “Predicting Job Start Times on Clusters”. In proceedings of 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2004), April 2004.

[10] 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 & Dist. Syst. 12(6), pp. 529-543, 2001.

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

[12] J. Skovira, W. Chan, H. Zhoi, and D. Lifka, “The EASY – LoadLeveler API project”. In Job Scheduling Strategies for Parallel Processing, LNCS vol. 1162, pp. 41-47, 1996.

[13] W. Smith, I. Foster, and V. Taylor, “Scheduling with advanced reservations”. In 14th Intl. Parallel & Distributed Processing Symp., pp. 127-132, May 2000.

[14] W. Smith, V. Taylor, and I. Foster, “Using run-time predictions to estimate queue wait times and improve scheduler performance”. In JSSPP 1999, LNCS vol. 1659, pp. 202-219, 1999.

[15] D. Tsafrir, Y. Etsion, and D. G. Feitelson, “Backfilling Using Runtime Predictions Rather Than User Estimates”. TR 2005-5, School of CS and Engineering, Hebrew University of Jerusalem, Feb 2005.

[16] D. Tsafrir, Y. Etsion and D. G. Feitelson, "Modeling User Runtime Estimates". In the 11th Workshop on Job Scheduling Strategies for Parallel Processing 2005, LNCS Vol.3834, 2005.

[17] Y. Zhang, H. Franke, J. E. Moreira and A. Sivasubramaniam, "An Integrated Approach to Parallel Scheduling Using Gang-Scheduling, Backfilling, and Migration". In Job Scheduling Strategies for Parallel Processing ’01, pp 133-158, 2001.

[18] J. Zilber, O. Amit and D. Talby, "What is Worth Learning from Parallel Workloads? A User and Session Based Analysis". In International Supercomputing Conference 2005 (ICS '05), June 2005.

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