Probabilistic Backfilling

Avi Nissimov and Dror G. Feitelson

Department of Computer Science
The Hebrew University of Jerusalem

Abstract. Backfilling is a scheduling optimization that requires information about job runtimes to be known. Such information can come from either of two sources: estimates provided by users when the jobs are submitted, or predictions made by the system based on historical data regarding previous executions of jobs. In both cases, each job is assigned a precise prediction of how long it will run. We suggest that instead the whole distribution of the historical data be used. As a result, the whole backfilling framework shifts from a concrete plan for the future schedule to a probabilistic plan where jobs are backfilled based on the probability that they will terminate in time.

1 Introduction

Scheduling parallel jobs for execution is similar to bin packing: each job needs a certain number of processors for a certain time, and the scheduler has to pack these jobs together so that most of the processors will be utilized most of the time. To perform such packing effectively, the scheduler needs to know how many nodes each job needs, and for how long. The number of processors needed is typically specified by the user when the job is submitted. The main question is how to estimate how long each job will run.

The simplest solution to this question is to require the user to provide a runtime estimate [1]. However, logs of jobs that have run on large scale parallel supercomputers reveal that user runtime estimates are very inaccurate [2]. The reason for this is that systems typically kill jobs that exceed their estimate, giving users a strong incentive to over-estimate the runtimes of their jobs.

The alternative to user-provided estimates is system-generated predictions. Practically all systems collect information about jobs that have run in the past. This information can then be mined to generate predictions about the runtimes of newly submitted jobs. Algorithms for generating such predictions are described in Section 2.

Prediction algorithms typically work in two steps. Given a newly submitted job, they first scan the available historical data and look for "similar" jobs that have executed in the past. For example, similar jobs may be defined as all the jobs that were executed on behalf of the same user on the same number of processors. They then apply some function to the runtimes of this set of jobs. For example, the function can be to compute the distribution of runtimes, and

E. Frachtenberg and U. Schwiegelshohn (Eds.): JSSPP 2007, LNCS 4942, pp. 102–115, 2008.
© Springer-Verlag Berlin Heidelberg 2008

extract the 90th percentile. This value is then used as the runtime prediction for the new job.

Our starting point is to observe that this prediction-generation process loses information: we have information about the runtimes of many previous similar jobs, but we reduce this into the single number — the prediction. Why not use all the available information instead? This means that scheduling decisions will be made based on assumed distributions of runtimes, rather than based on predictions of specific runtimes.

The advantage of making a specific prediction is that the scheduling becomes deterministic: when we want to know whether a job can run or not, we assume it will run for the predicted time, and then check whether we have enough processors that are free for this duration. But if we use a distribution, we are reduced to probabilistic arguments. For example, we may find that there is an 87% chance that the processors will be free for the required time. But this is actually a more accurate representation of the situation at hand, so it has the potential to lead to better decisions.

We apply the above ideas in the context of backfilling schedulers. Backfilling is an optimization usually applied to FCFS scheduling that allows small and short jobs to run ahead of their time provided they fit into holes that were left in the schedule. In our new approach, this fit becomes a probabilistic prediction; jobs will be backfilled provided there is a high probability that they will fit. In other words, we define a threshold τ and perform the backfilling provided that the probability that the job will not terminate in time is less than τ.

In keeping with the spirit of backfilling, the meaning of "will not terminate in time" is that the backfilled job will delay the first queued job. The algorithm for calculating this is described in detail in Section 4. The results of simulations that assess how well this performs are then shown in Section 5.

2 Algorithms That Use Predictions

There are many different algorithms that require predictions or user estimates of job runtimes, including EASY backfilling and shortest-job-first. In EASY backfilling, jobs are backfilled provided they do not delay the first queued job [1]. One of the conditions used to verify this is that the backfilled job will terminate before the time when enough processors for the first queued job will become available. This requires knowing the runtimes of currently executing jobs (in order to find out when they will free their processors), and the runtime of the potential backfill job (to find out if it will terminate in time). Shortest job first requires runtime knowledge in order to sort the jobs.

There has been some debate in the literature on whether accurate runtime predictions are actually important. Somewhat surprisingly, the first papers on this issue indicated that *inaccurate* predictions lead to improved performance [3,4]. However, more recent research has shown that accurate estimates are indeed beneficial [5,6,7], thus providing added motivation for the quest for more accurate predictions.

Several algorithms have been suggested to enable runtime predictions based on historical data. Gibbons first partitions the historical data into classes based on the user and the executable. Importantly, executions on different numbers of processors are included in the same class. He then finds a quadratic least-squares fit of the runtime as a function of the number of processors used. This is used to compute a prediction for the requested number of processors [8]. Smith et al. also divide jobs into classes, but use various job attributes in addition to the user and executable. They then use the mean runtime of all previous executions in the class as the prediction [9]. Mu'alem and Feitelson suggest using the mean runtime in the class plus 1.5 standard deviations, to reduce the danger of under-prediction [2]. Tsafrir et al. use the simplest scheme of all: they just use the average runtime of the last two terminated job that have been submitted by the same user [7].

The problem with using runtime predictions other than user estimates with backfilling is the fact that the jobs may be under-predicted. Killing the jobs in this situation is highly undesirable, since the users have neither tools to avoid it nor indication that this is going to happen. Therefore, the only reasonable way to solve under-prediction is to violate the reservation for the first job in the queue and delay it until the processors become available [7]. But there is no promise such delays will ever stop, unless we forbid future backfilling, because the backfilled jobs may in their turn also be under-predicted.

The same question arises when the predictions are initially set too large, like when using doubling (or tripling, quadrupling and so on) [7]. If the system were a single-user system, then this strategy would probably be good — it pushes forward the jobs with less requirements (on average), so the average waiting time is expected to decrease. However, since we are dealing with multi-user systems, such an approach is insufficient, and may appear extremely unfair.

In this work, however, we use EASY-backfilling as the base algorithm. According to [7], when the predictions are correct, the overall performance of EASY-backfilling usually improves.

3 Predicting Job Runtime Distributions

Both backfilling-based schedulers and SJF use single-value predictions due to their simplicity. But in fact predicting a job's runtime cannot usually be done deterministically. A job's runtime depends on many factors, that include not only system internal conditions such as the network load, but also terminations due to errors and user cancellations. These last factors are external to the system, and they greatly complicate runtime prediction. Errors usually show incorrect behavior pretty soon after a job starts, and many faults may be discovered long before a job would have terminated without the error. Users also know this, and they tend to test partial output soundness soon after their jobs start. Therefore, in cases of errors the job is usually terminated or canceled almost immediately. For instance, 2613 out of 5275 (~50%) canceled jobs in the SDSC-SP2 trace whose user estimates were set for at least 200 minutes were canceled

within 20 minutes after their start times (this and other traces we use come from the Parallel Workloads Archive [10]; see Table 1). Modeling these scenarios is impossible with single-value predictions: a single value can give either a mean or a quantile of the job's runtime distribution, but cannot model a multi-modal distribution.

Another problem with single-value predictions is the fact that they should contain all the information upon which scheduling decisions are made. Different deviations from the real runtime cause different and possibly incomparable damage. This leads to prediction policies that are scheduler-dependent. An extreme example is backfilling, which kills jobs whose runtimes are longer than the user estimates. Thus, over-predictions are much less damaging than under-predictions. Therefore the user estimates tend to be biased upwards, as users tend to give high estimates fearing their jobs will be killed.

Predicting the distribution of a job's runtime is based on the concept of locality of sampling [11]. This makes a distinction between the global distribution of runtimes, when looking at a long time span, and the local distribution of runtimes that is observed at a certain instant. The idea is that runtimes — like other workload attributes — exhibit locality, so the local distribution is much more predictable than the global one.

To utilize this observation, we divide time into short slices, and characterize the runtime distribution in each one using binning. In particular, the model groups jobs arriving within each 15-minute slice of time together. The runtime distribution was modeled for each slice individually. The modeling is done by defining a set of discrete bins, and counting how many job runtimes fall in each bin. The bin sizes used were logarithmic, with ranges that grow by a factor of 1.8; this gives a better resolution for short jobs, which are more numerous. The values 15 minutes and 1.8 were selected empirically so as to maximize the observed locality in several different workload traces [12].

To reduce complexity it is desirable to track only a limited number of distinct distributions. The tradeoff here is that using more distributions increases accuracy, but also increases the complexity of the modeling. Therefore we want to find the number that provides good accuracy at an acceptable cost. In most cases it turned out that 16 distinct distributions provide reasonable results.

Coming up with representative distributions involves a learning process. Once enough data is available (we use one week's worth of activity) a set of 16 representative distributions is learned using an iterative process. The learnt distributions are then used in the HMM model described below as predictions for the different jobs. Typically only 2–3 iterations are needed to converge to an acceptable set; using more typically results in overfitting. The criterion for convergence is that the distance from the previous model, multiplied by the square root of the number of samples (all the jobs observed so far) is smaller than a given threshold. Later, as more data is accumulated, this will grow again beyond the threshold, and the learning process is repeated using all the additional data accumulated so far. Thus the quality of the model is expected to improve the longer the system is in use.

The final stage of the modeling is to create a Hidden Markov Model (HMM) to describe transitions and see how things change. The model has 16 states, corresponding to the different runtime distributions. States may have self-loops to account for situations where the local distribution stays the same for more than 15 minutes. Checking the observed distributions of how long each state is in effect indeed revealed that in the vast majority of cases this is geometrically distributed.

The distributions and model are learned on-line as more jobs are submitted and terminate. Thus when running the algorithm on a job trace, initially it is impossible to provide good predictions. when enough information accumulates, the model tracks the state that the system is in, and uses the distribution that characterizes this state as the prediction for newly submitted jobs.

4 Using Distributions in the Scheduling Algorithm

Given historical data regarding previous job executions, one can fit a model of the distribution of runtimes or just use the empirical distribution. This section discusses the ways how this information can be practically used by a scheduler. In particular, we base our work on the EASY backfilling scheme.

Given that runtimes are continuous, keeping historical data about multiple jobs can burden the system and increase the complexity of the scheduling algorithm. For that reason we will assume the distribution is discretized by dividing the runtimes into N bins. The sizes of the bins will be logarithmic: there will be many bins for short runtimes, and the top bins each represent a large range of runtimes.

EASY backfilling maintains a queue of waiting jobs (ones that have been submitted but have not yet started) ordered by their submission times. The steps of the EASY backfilling scheduling procedure, which is executed each time a job arrives or terminates, are as follows:

1. As long as there are enough idle processors to start the first job in the wait queue, remove this job from the queue and start it.
2. Given the first job in the queue that cannot start because of insufficient idle processors, find when the required number will become free and make the reservation for this job.
3. Continue scanning the queue, and start (backfill) jobs if they don't violate this reservation.

However, the idea of the algorithm can be expressed more concisely. In fact, Step 2 is more of an implementation issue than part of the core of the algorithm. Thus steps 2 and 3 can be united as follows: "Continue scanning the queue, and start jobs if this doesn't delay the start time of the first job in the queue". This is independent of how the condition of not delaying the first job is verified. And we can also relax the condition, and replace it with a condition that it will not be delayed with a high probability.

In EASY backfilling each job is assigned a single value for its predicted runtime, and this prediction is used as the exact runtime in a very deterministic way. But if we don't have a single-value prediction, but rather a distribution, it is not possible to make such a decision in a deterministic way. Instead, there are many cases with different probabilities that may contradict each other. Therefore we need to summarize all these possibilities. To do so, we define a single parameter that is the confidence probability τ. Our new condition for backfilling will be that *the probability that the backfilling postpones the start of the first job in the queue is less than* τ.

Let us now formalize this idea. For simplicity, it is assumed that the job runtimes are independent; thus, for each two jobs with runtimes R_1, R_2, we have that $\Pr(R_1, R_2) = \Pr(R_1)\Pr(R_2)$ (as usual, here and everywhere, $\Pr(R_1)$ denotes the probability of random variable R_1 to have its value). In particular, this means that the event of the availability of processors at different times due to terminations of the currently running jobs and the distribution of the backfilled job's runtime are independent.

The following notation will be useful. Suppose the current time is t_0. Assuming that the job we are considering is indeed backfilled, we denote its (unknown) true termination time by t_e. For each time t, $t_0 \le t < \infty$, we denote by $c(t)$ the number of processors that are released by the currently running jobs before and including time t. Also, let c_q be the number of processors that must be released to start the first job in the queue, and c the number needed to start both the first job and the backfill job together. Armed with these notations, we can say that the algorithm should backfill iff

$$\Pr(\exists t \in (t_0, t_e) : c_q \le c(t) < c) \ < \ \tau.$$

In words: the probability that there exists some time t before the termination of the backfilled job when the number of released processor's is enough to start the first job in the queue but not enough to run both jobs — so the backfilling postpones the start of the first job in the queue — is smaller than τ.

However, the termination time t_e is not known. Instead, we have a distribution. Integrating over all the possible termination times of the backfilled job we then receive the condition

$$\int_{t_0}^{\infty} \Pr(t_e, \exists t \in (t_0, t_e) : c_q \le c(t) < c)\, \mathrm{d}t_e \ < \ \tau.$$

Since by assumption of job runtimes independence t_e and $c(t)$ are independent, this probability is

$$\int_{t_0}^{\infty} \Pr(t_e) \Pr(\exists t \in (t_0, t_e) : c_q \le c(t) < c)\, \mathrm{d}t_e \ < \ \tau.$$

The first factor in the integrand is modeled by the predictor — it is exactly the predicted distribution of the job's runtime. As noted above these probabilities are typically modeled discretely, by dividing the runtime into bins and predicting the probability to fall into each bin. The second factor is much harder to calculate.

Algorithm 1. Runtime bin probability recalculation

```
 1 double[] recalculate(Job job)
 2    // old model
 3    double old_p[N] = job.model;
 4    // new distribution model
 5    double new_p[N];
 6    double upperBound = job.userEstimate;
 7    double lowerBound = currentTime-job.startTime;
 8    for each runtime bin j do {
 9       double newBinStart =
10           max{bin[j].start, min{lowerBound, bin[j].end}};
11       double newBinEnd =
12           min{bin[j].end, max{upperBound, bin[j].start}};
13       new_p[j] = old_p[j]*
14             (log(newBinEnd )-log(newBinStart )) /
15             (log(bin[j].end)-log(bin[j].start));
16    }
17    normalize(new_p);
18    return new_p;
19 }
```

First of all, in order to calculate the second factor, we must calculate the probability $\Pr(c(t) \geq c)$ for any given time t and any given requirement c. The probability of processor availability given termination probabilities at time t of the currently running jobs is calculated using Dynamic Programming. The matrix cell $M_t[n][c]$ denotes the probability that at time t, the jobs $1..n$ have released at least c processors. This is calculated recursively as

$$M_t[n][c] = M_t[n-1][c] + (M_t[n-1][c-c_n] - M_t[n-1][c]) \cdot P_t[n].$$

The first term denotes the case when enough processors are already idle without termination of job number n. The second term is the probability that only the termination of job n freed the required processors. This is the product of two factors: that jobs $1..n-1$ freed at least $c-c_n$ but not c processors, and that the last job terminated in time (c_n is number of processor used by job number n, and $P_t[n]$ is the probability that job number n terminates not later than t). The initialization of the dynamic programing sets the obvious values: $M_t[*][0] = 1$ (we are sure that the jobs have released at least 0 processors), and $M_t[0][*] = 0$ (zero jobs do not release any number processors). In the algorithm implementation, these values may be calculated on-the-fly; for instance, if $c < c_n$ (the number of required processors is smaller than the number of processors used by job number n), then $M_t[n-1][c-c_n]$ doesn't exist in the real matrix, because the index is negative, but can easily be substituted by 1, so that $M_t[n][c] = M_t[n-1][c] + (1 - M_t[n-1][c]) \cdot P_t[n]$. If n is the number of running jobs, then $\Pr(c(t) \geq c) = M_t[n][c]$.

The above requires calculating the probabilities of running job terminations before or at the time t (denoted above as $P_t[\,]$). Each time the scheduler is called, a larger part of the distributions becomes irrelevant, because the jobs have already run longer than the times represented by the lower bins. Therefore the distributions needs to be recalculated. Because our data is discretized, the job runtime probabilities are estimated only at the ends of the runtime bins. The upper and lower bounds of job runtimes are the user estimate (since the job is killed after it; this is used even before the job starts) and the current runtime of the job (`currentTime-job.startTime`). Log-Uniform intra-bin interpolation is used. Algorithm 1 presents the recalculation procedure for the runtime bin probabilities. **Line 3** receives the job distribution model as proposed by the predictor (reminder: N is the number of the runtime bins, and $j = 1..N$ is the index of a runtime bin). **Lines 9-12** ensure that the new runtime bin boundaries satisfy the old bin boundaries and global boundaries. If the runtime bin doesn't intersect the global boundaries then `newBinStart==newBinEnd` and therefore `new_p[j]=0`. **Lines 13-15** recalculate the probability measure remainders after log-uniform interpolation.

After all the events representing the possible termination of a job are inserted into a list and sorted by the time, one can easily calculate the vector of termination probabilities at time t.

Let $A(t)$ be the event that t is the real start time of Q[0] (the first job in the queue) without backfilling, and that backfilling of the job delays the first job beyond this time. This means that

$$A(t) = (t \in (t_0, t_e)) \wedge (\forall s < t, c(s) < c_q) \wedge (c_q \leq c(t) < c)$$

In words, t is before the end time of backfilled job and t is the first time when the first job in the queue can start but only if this job isn't backfilled. Therefore, according to our probabilistic backfilling condition, the backfilling should happen iff

$$\Pr(\exists t \in (t_0, t_e) : A(t)) \ < \ \tau$$

But the events are disjoint, therefore the total probability is the integral of probabilities:

$$\int_{t_0}^{t_e} \Pr(A(t))\mathrm{d}t \ < \ \tau \tag{1}$$

The problem is to calculate $\Pr(A(t))$.

Suppose $t \in (t_0, t_e)$. Let us change the definition of t to be discrete time (in any units). Due to the monotonicity of $c(t)$, $\Pr(A(t)) = \Pr(c(t-1) < c_q \wedge c_q \leq c(t) < c)$. If $c(t) \geq c$ or $c(t-1) \geq c$, then $c(t) \geq c_q$, since $c > c_q$ and $c(t)$ is monotonous (see Venn diagram in Figure 1). Therefore,

$$\begin{aligned}\Pr(A(t)) &= \Pr(c(t) \geq c_q) - \Pr(c(t-1) \geq c_q \vee c(t) \geq c) \\ &= M_t[n][c_q] - \Pr(c(t-1) \geq c_q \vee c(t) \geq c).\end{aligned}$$

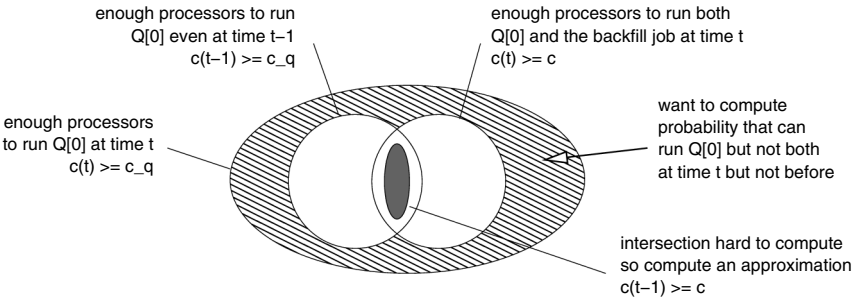

Fig. 1. Explanation of the $\Pr(A(t))$ formula

But in the second term, the two events in the disjunction don't imply each other, so

$$\begin{aligned}\Pr(\, c(t-1) \geq c_q \;\vee\; c(t) \geq c\,) = \\ = \Pr(c(t-1) \geq c_q) \;+\; \Pr(c(t) \geq c) \;-\; \Pr(c(t-1) \geq c_q \wedge c(t) \geq c) \\ = M_{t-1}[n][c_q] \;+\; M_t[n][c] \;-\; \Pr(c(t-1) \geq c_q \wedge c(t) \geq c)\end{aligned}$$

The last term is pretty hard to calculate. However, it has a lower bound of $M_{t-1}[n][c]$ — the probability that before the last event there were enough processors to run both jobs (which implies $c(t-1) \geq c_q$ and $c(t) \geq c$). Using all the above considerations leads to the bound

$$\Pr(A(t)) \;\geq\; (M_t[n][c_q] - M_t[n][c]) \;-\; (M_{t-1}[n][c_q] - M_{t-1}[n][c]) \qquad (2)$$

The integral over t of $\Pr(A(t))$ in Equation (1) turns into a sum when time is discretized. Replacing $\Pr(A(t))$ with the lower bounds from Equation (2) leads to a telescoping series. Since the first item equals 0 (because initially the number of processors is less than c_q), the total sum is $M_{t_e-1}[n][c_q] - M_{t_e-1}[n][c]$. But although each of $M_t[n][c_q], M_t[n][c]$ is monotonically growing as a function of t, their difference is not monotonous; while all $\Pr(A(t)) \geq 0$, and their sum is monotonous. This means we have a tighter bound of

$$\sum_{t\in(t_0,t_e)} \Pr(A(t)) \geq \max_{t\in(t_0,t_e)} \{M_t[n][c_q] - M_t[n][c]\}$$

To summarize, the version of EASY backfilling that uses runtime distributions rather than point predictions will backfill a job if the following condition holds

$$\sum_{t_e} \Pr(t_e) \max_{t\in(t_0,t_e)} \{M_t[n][c_q] - M_t[n][c]\} \;<\; \tau$$

That is, if the probability that such a time exists is less than the threshold. Algorithm 2 presents the simplified pseudo-code of this backfilling scheduler. Some

Algorithm 2. The distribution-based condition for backfilling.

```
bool shouldBackfill(Job job) {
   List events = <list of job terminations sorted by time>
   int n = <# of running jobs>
   double P[n];
   // max(t){M[n][c]-M[n][c0]}
   double pMax = 0;
   double result = 0;
   int c0 = <# of processors needed to run Q[0]>
   int c =  <# of processors needed to run both jobs>
   for each j=runtime bin do {
      for each e in events before bin[j].end do {
         P[e.job] += e.probability;
         <calculate M using Dynamic Programming, given P>
         pMax = max{pMax, M[n][c0]-M[n][c]};
      }
      result += job.model[j]*pMax;
   }
   return result < THRESHOLD;
}
```

notes on the implementation: The `result` variable is monotonously growing, so once it is bigger than the threshold the total result is false for sure, so no further calculations are run. The `pMax` variable is also monotonously growing. This means that if the remaining runtime bin probability multiplied with the current `pMax` together with the current `result` are bigger than the threshold, it is also enough to stop calculating and return false. These improvements are very important, since the scheduler runs on-line. Our simulations (reported in the next section) indicate that indeed the overhead of the scheduler is very low: simulating a full year of activity, with order of 100,000 calls to the scheduler, takes about half an hour.

If the predictor returns no runtime prediction (which might happen if no historical data is available), then the single probability event is inserted, which is the user estimate with probability of 1. If this is the case for all the running jobs, then the algorithm works exactly like the original EASY algorithm: all the termination events come from the running jobs' terminations by user estimates, and therefore the algorithm works in a very deterministic way.

5 Results

The probabilistic backfilling scheme described above was evaluated by comparing it with EASY backfilling, using simulations of several workloads available from the Parallel Workloads archive [10] (Table 1). In these workload logs, jobs that are canceled before they start have 0 runtime and also 0 processors. These jobs were removed from the simulation. If a job requires more processors than the machine has, the requirement is aligned to the machine size.

Table 1. Workloads used in the analysis and simulations. Average wait and run times are in minutes.

log	duration	jobs	avg wait	avg run
CTC SP2	6/96–5/97	77,222	425.7	188.0
KTH SP2	9/96–8/97	28,489	334.6	161.8
SDSC SP2	4/98–4/00	59,725	429.6	123.6
SDSC Blue	4/00–1/03	243,314	720.2	95.5

The runtime distributions were modeled using a Hidden-Markov Model with 16 states, where each state corresponds to a runtime distribution. The model grouped jobs arriving within a 15-minute slice of time together. The runtime distribution was modeled using logarithmic bins, with ranges that grow by a factor of 1.8. The details of the modeling are presented in detail in [12]. The threshold used for the probabilistic backfilling was $\tau = 0.05$.

In order to avoid the influence of the runtime differences between the traces we used waiting time for the performance metric. The system is a multi-user system, therefore fairness is also an issue. Therefore, the L_1-type metrics that take the average or sum of all the jobs' metric values are not enough — a job that suffers from bad service is not compensated by the fact that in average the jobs wait little in the queue. In order to present the complete picture of what is going on for all the jobs the full CDFs of the waiting times are presented.

Figure 2 compares the conventional EASY scheduler with the probability-based scheduler. The X-axis is the waiting times of the jobs in a logarithmic scale, and the Y-axis its CDF. The CDF doesn't start from 0, since there a large fraction of the jobs don't wait in the queue at all: around 50% of the jobs for SDSC Blue and KTH, slightly less for SDSC SP2, and more than 75% of the jobs for CTC.

As the results are shown in the form of a CDF, a curve that is lower and more to the right implies higher wait times and thus worse performance. Conversely, a curve that is higher and to the left indicates lower wait times and better performance. The arrows represent the fraction of jobs for which waiting time improved due to the probabilistic approach — this is the interval of the CDFs where the results for the probabilistic scheme (dashed line) are to the left of and above the EASY results (solid line).

The conclusions of this chart is that usually most of the jobs that had to wait at all are better off using the probabilistic approach. Note that the X-axis is logarithmic, and actually covers a very large range — it changes by a factor of 2.5×10^6. Therefore, when the line moves left even for a little, this may represent an improvement factor of 2. Also, it looks that if the job started waiting, it usually waits for at least a minute. Another finding is that there is a place in the chart where the line is almost straight. This means that the waiting time distribution at some intervals is close to a log-uniform distribution.

Tables 2 and 3 summarize the improvements in the wait time metric in the form of the arithmetic and geometric means. The formula for calculating the

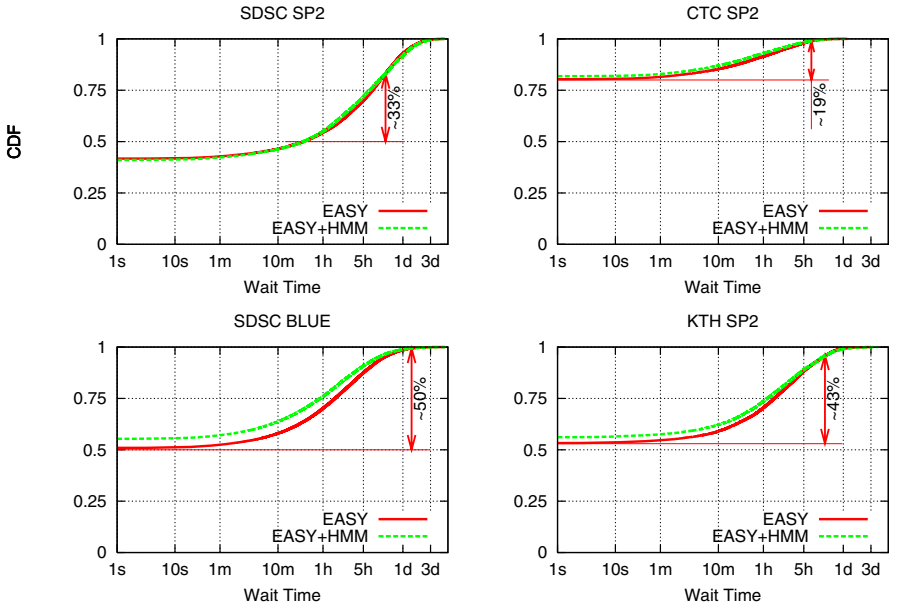

Fig. 2. CDF of waiting time for Probabilistic EASY vs. base EASY. The arrows show the jobs that benefit from the probabilistic approach.

Table 2. Arithmetic mean of waiting times

Trace name	EASY	Probabilistic	
CTC SP2	21.3 min	18.1 min	-15.2%
SDSC SP2	364 min	373 min	+2.6%
SDSC Blue	131 min	105 min	-19.5%
KTH SP2	114 min	113 min	-0.6%
Total			-8.7%

Table 3. Geometric mean of waiting times

Trace name	EASY	Probabilistic	
CTC SP2	28.2 sec	25.3 sec	-10.1%
SDSC SP2	639 sec	635 sec	-0.7%
SDSC Blue	203 sec	135 sec	-33.6%
KTH SP2	181 sec	147 sec	-18.9%
Total			-16.7%

geometric mean is $\exp(\int f(w) \ln \max\{w, w_{\min}\} \mathrm{d}w)$, where w is the job's waiting time, $f(w)$ is its PDF and $w_{\min}$ is the commonly used threshold of 10 seconds,

see for instance [13]. Therefore, the improvement in the geometric mean metric value is exactly the area between the lines of the chart that are to the right of $w = w_{\min}$.

6 Conclusions

Scheduling algorithms such as backfilling and SJF require job runtimes to be known, or at least predicted. Previous work has always assumed that such predictions have to be point estimates. In contradistinction, we investigate the option of predicting the *distribution* from which the actual runtime will be drawn. This is then integrated into the EASY backfilling algorithm, and shown to reduce the expected waiting time and improve the wait-time distribution.

Once a distribution-based probabilistic backfilling algorithm is in place, several courses of additional research suggest themselves. One is a comparison with the performance obtained by other (single value) prediction schemes. Another is a deeper investigation of alternative ways to predict distributions. In this work we used a rather complex HMM-based prediction scheme. A possible alternative is to just use the empirical distribution of jobs by the same user. This holds promise because it provides more focus on the local process, as opposed to the HMM which takes a global view at the possible expense of predictions for a single job. But a thorough experimental study is needed to verify and quantify the relative performance of the two approaches.

References

1. Lifka, D.: The ANL/IBM SP scheduling system. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 1995. LNCS, vol. 949, pp. 295–303. Springer, Heidelberg (1995)
2. Mu'alem, A.W., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. IEEE Trans. Parallel and Distributed systems 12(6), 529–543 (2001)
3. Feitelson, D.G., Mu'alem Weil, A.: Utilization and predictability in scheduling the IBM SP2 with backfilling. In: International Parallel Processing Symposium, Number 12, pp. 542–546 (1998)
4. Zotkin, D., Keleher, P.J.: Job-length estimation and performance in backfilling schedulers. In: International Symposium on High Performance Distributed Computing, Number 8 (1999)
5. Chiang, S.H., Arpaci-Dusseau, A., Vernon, M.K.: The impact of more accurate requested runtimes on production job scheduling performance. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2002. LNCS, vol. 2537, pp. 103–127. Springer, Heidelberg (2002)
6. Tsafrir, D., Feitelson, D.G.: The dynamics of backfilling: solving the mystery of why increased inaccuracy may help. In: IEEE International Symposium on Workload Characterization, pp. 131–141 (2006)
7. Tsafrir, D., Etsion, Y., Feitelson, D.G.: Backfilling using system-generated predictions rather than user runtime estimates. IEEE Trans. Parallel and Distributed systems 18(6), 789–803 (2007)

8. Gibbons, R.: A historical application profiler for use by parallel schedulers. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 1997. LNCS, vol. 1291, pp. 58–77. Springer, Heidelberg (1997)
9. Smith, W., Foster, I., Taylor, V.: Predicting application run times using historical information. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 1998. LNCS, vol. 1459, pp. 122–142. Springer, Heidelberg (1998)
10. Parallel workloads archive, `http://www.cs.huji.ac.il/labs/parallel/workload/`
11. Feitelson, D.G.: Locality of sampling and diversity in parallel system workloads. In: 21st International Conference on Supercomputing, pp. 53–63 (2007)
12. Nissimov, A.: Locality and its usage in parallel job runtime distribution modeling using HMM. Master's thesis, The Hebrew University (2006)
13. Feitelson, D.G., Rudolph, L., Schwiegelshohn, U., Sevcik, K.C., Wong, P.: Theory and practice in parallel job scheduling. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 1997. LNCS, vol. 1291, pp. 1–34. Springer, Heidelberg (1997)