Job-Length Estimation and Performance in Backfilling Schedulers

Dmitry Zotkin and Peter J. Keleher
University of Maryland, College Park
College Park, MD 20742
keleher@cs.umd.edu

June 15, 1999

Abstract

Backfilling is a simple and effective way of improving the utilization of space-sharing schedulers. Simple first-come-first-served approaches are ineffective because large jobs can fragment the available resources. Backfilling schedulers address this problem by allowing jobs to move ahead in the queue, provided that they will not delay subsequent jobs.

Previous research has shown that inaccurate estimates of execution times can lead to better backfilling schedules. We characterize this effect on several workloads, and show that average slowdowns can be effectively reduced by systematically lengthening estimated execution times. Further, we show that the average job slowdown metric can be addressed directly by sorting jobs by increasing execution time. Finally, we modify our sorting scheduler to ensure that incoming jobs can be given hard guarantees. The resulting scheduler guarantees to avoid starvation, and performs significantly better than previous backfilling schedulers.

1 Introduction

First-come, first-serve (FCFS) scheduling is widely used. However, it is far from an ideal scheduling policy. The primary problem is that fragmentation causes many processors to remain idle and utilization suffers [3]. While there exist solutions that provide theoretically optimal performance (gang scheduling [2] and dynamic partitioning [6]), they are usually impractical for implementation in production systems. The simplest approach that provides efficient scheduling is to use *backfilling* [1].

Backfilling refers to an optimization of FCFS policies where jobs that are not currently at the head of the job queue are allowed to bypass jobs that arrived earlier. The intuitive criteria guiding this process is that a job can only be moved forward if it will not interfere with other jobs in queue. For example, assume a situation where there are idle processors in the system, but not enough to run the first job in the queue. Overall throughput might be improved by immediately running a short job that requires a small number of processors but resides in the middle of the queue. Two ways of backfilling are examined in [1]. Conservative backfilling allows jobs to bypass earlier jobs only when it will not delay any prior job. EASY backfilling [5, 7] allows jobs to move forward whenever it can do so without slowing the first job in the queue. Both backfilling schemes rely on users estimating the length of the jobs they are submitting for the execution. This information is used in determining whether a job is "sufficiently small" to run without creating interference. Any job not completed in the estimated time period is killed. Observed slowdown was comparable for both backfilling algorithms, while the conservative approach has an additional guarantee that jobs can not suffer unbounded delays.

A more surprising result was that overestimation of job lengths provides better performance that accurate estimation. The tentative explanation given by the authors is that inaccurate estimates give the algorithm flexibility to find better schedules.

This paper makes two contributions. First, we thoroughly characterize this effect, showing under which conditions and with what job mixes it occurs. Our results suggest that current schedulers can be improved by mutating estimated execution times in a controlled manner.

Second, we show that the typical metric used to evaluate the performance of job schedulers, average slowdown, is heavily influenced by the short jobs that dominate job mixes in existing installations. We show that average slowdowns can be addressed directly by sorting incoming jobs as they arrive. Finally, we characterize the effect that guaranteeing to avoid starvation has on average slowdown.

2 Background

Backfilling is the algorithm used by the "EASY" scheduler on the IBM SP2. The input for the scheduler is the queue of jobs that are to be run on the system. Each job is described by the number of processors it requires, an estimate of the execution time, and the submission time. Jobs enter the queue when submitted by the user. The algorithm has access only to the information about jobs that are currently in the queue. When a job is allocated, it is started on a partition of the requested size and allowed to run for a duration equal to the execution time estimate. If this limit is reached, the job is killed in order to avoid delaying other jobs.

When a job ends, the algorithm checks if the job at the head of the queue can be started. If the number of available processors is insufficient, then all jobs that are currently running on the system are sorted in order of their expected completion, and the algorithm determines the *shadow time* – the time when there will be a sufficient number of processors available for the first job in the queue. Free processors in excess of this number at the shadow time are termed "extra nodes." The algorithm attempts to exploit extra nodes, together with free processors before the shadow time, by selecting and running the first job that either:

1. will use only currently free nodes and be finished by the shadow time, or

2. will use only the extra nodes

The algorithm repeats until jobs fitting these criteria can no longer be found. No such job will delay the first job on the queue. Backfilling therefore guarantees that once a job comes to the head of the queue and is assigned a starting time, the job will never be delayed past this time. Note that this does not mean that no other job will be delayed due to the backfilling, and jobs may be delayed many times before arriving at the head of the queue, but the algorithm favors overall utilization over the fairness of the queuing time.

Both conservative and aggressive backfilling [1] perform nearly identically with respect to system utilization. Thus, we use the related metric of the bounded average slowdown to quantify the quality of the schedule. The slowdown sd_i for the i^{th} job in the queue is defined as:

$$sd_i = 1 + \frac{T_q}{T_{br}}, \quad T_{br} = max(T_r, 10)$$

where T_r is the execution time of the job and T_q is the time spent in the queue (job start time minus job submission time). The threshold of 10 seconds prevents very short jobs from overly influencing the average slowdown [1].

Trace	KTH	LNL	SUD
Number of processors	100	256	256
Total number of jobs	28456	22076	18415
Average load (in jobs)	70.8	43.3	12.9

Table 1: Workload trace description

3 Workload description

We use three sets of data in our experiments. Two of them are actual workload traces from parallel machines, and the third is an artificial trace constructed in order to verify whether the above effect appears when the workload is not dominated by the short jobs. For each job in the workload, the trace contains job arrival (submission) time, actual (not estimated) job execution time, and the number of processors required by job. Information about the traces is summarized in Table 1.

The two traces from real systems are KTH, from the 100-node machine installed at the Royal Institute of Technology in Stockholm, Sweden, and LNL from the 256-node machine at the Lawrence Livermore National Lab. In addition, we generated an artificial trace called Synthetic Uniform Distribution (SUD), with a uniform distribution of job lengths in the range from 1 to 240 minutes.

Histograms of the number of jobs with different job lengths are given in Figures 1 and 2 (recall that SUD has a uniform distribution). Note that the y-axis is logarithmic, so the number of short jobs in the traces is overwhelming. In fact, more than half of jobs in LNL trace and more than one third in KTH trace are shorter than one minute in duration. This seems to affect algorithm behavior; we introduced the third (SUD) trace with a uniform distribution of job lengths in order to verify whether this factor is indeed important.

Note the peaks on both histograms around 120, 180, and 240 minutes. We theorize that this is the result of users choosing round numbers for estimates, and jobs being killed off as the estimates are exceeded. Unfortunately, we have no information on which jobs exceed their limits.

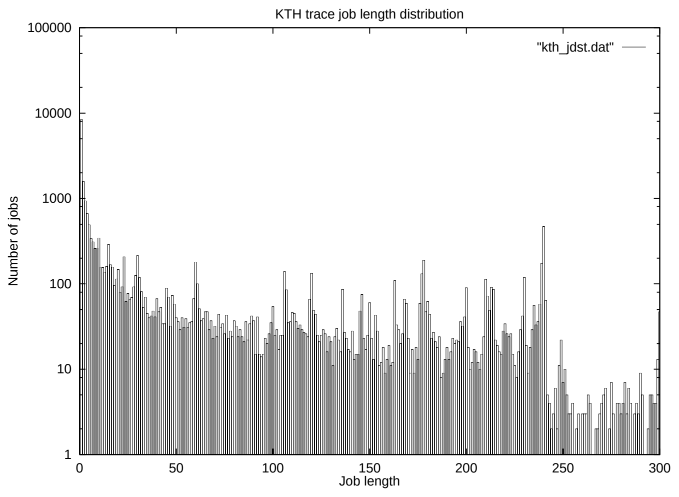

Figure 1: KTH job length distribution (seconds)

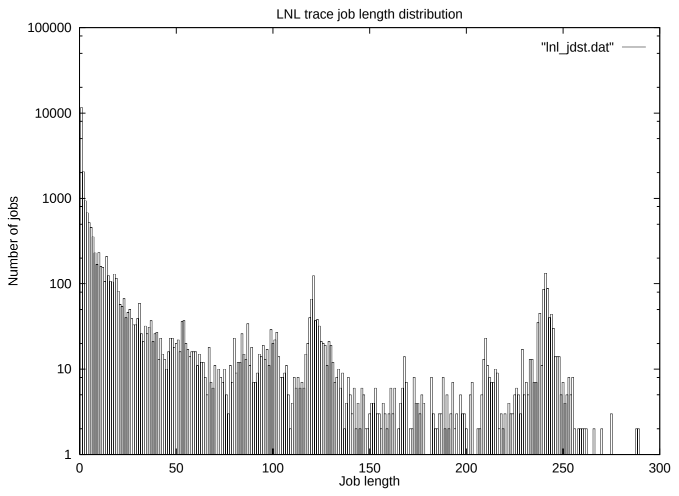

Figure 2: LNL job length distribution (seconds)

4 Experimental setup and results

We built a simple simulator of the EASY scheduler and conducted a number of experiments using the traces described above. Since the traces do not keep the estimates of execution time given by the users, we derive the estimate T_e from the actual execution time T_r with one free parameter R. We use two methods of computing T_e. The *deterministic* method sets T_e to RT_r. The *randomized* approach sets $T_e = rnd(T_r, 2RT_r)$, where $rnd(a, b)$ denotes a random number between a and b with uniform distribution. We introduced the coefficient of 2 in the second formula so that two distributions with the same R have the same mean and can be plotted against each other. $R = 1$ means completely accurate estimations; large values of R give less accurate estimates of job execution time.

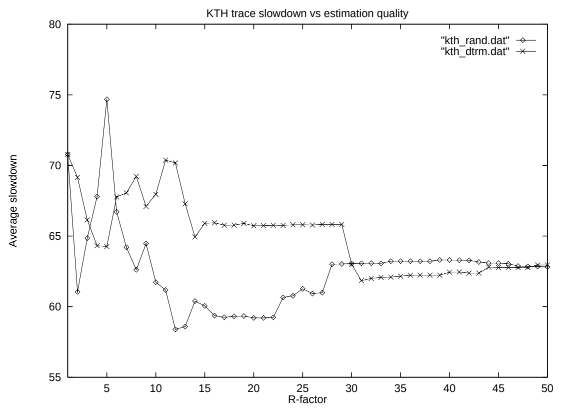

Figure 3: KTH average bounded slowdown vs. R

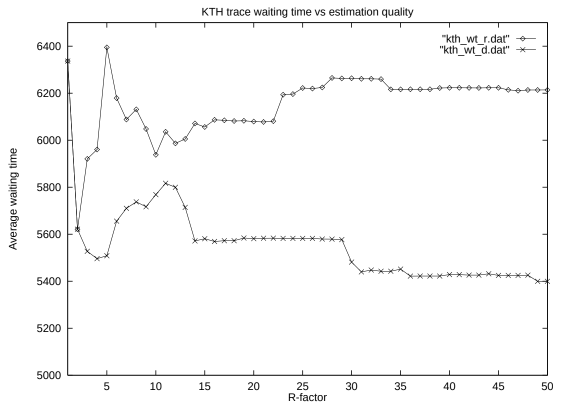

Figure 4: KTH average waiting time vs. R

Plots of bounded average slowdown and absolute slowdown (average waiting time) versus R are given in Figures 3-8 for the three traces used in the study. Tables 2 and 3 provide comparisons of backfilling with strict FCFS.

General tendencies are the same for both real traces.

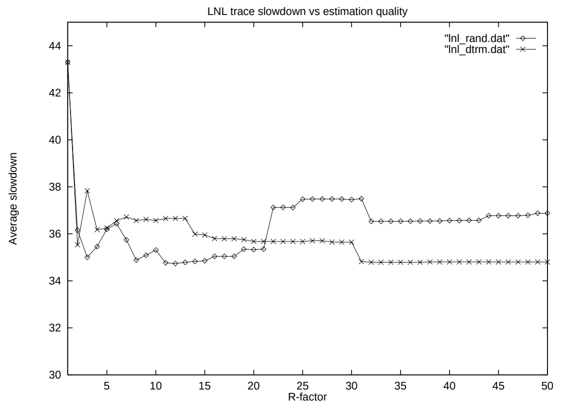

Figure 5: LNL average bounded slowdown vs. R

R-factor	KTH_{dtm}	LNL_{dtm}	SUD_{dtm}	KTH_{rnd}	LNL_{rnd}	SUD_{rnd}
FCFS	8800.01	1600.34	640.22	8800.52	1600.97	640.03
1	70.78	43.29	12.89	70.78	43.29	12.89
2	69.16	35.53	12.14	61.05	36.16	12.61
3	66.14	37.83	12.07	64.86	35.00	13.06
5	64.26	36.24	12.38	66.72	36.20	13.40
20	65.73	35.67	12.50	59.21	35.33	13.88
50	62.95	34.80	12.51	63.91	36.87	13.90

Table 2: Average slowdown for on-line version

R-factor	KTH_{dtm}	LNL_{dtm}	SUD_{dtm}	KTH_{rnd}	LNL_{rnd}	SUD_{rnd}
FCFS	459.03	37.63	1740.23	459.06	37.61	1740.28
1	6.34	1.87	38.25	6.34	1.87	38.25
2	5.62	1.66	35.98	5.62	1.78	38.04
3	5.53	1.72	36.26	5.92	1.66	38.81
5	5.51	1.68	36.17	6.39	1.79	39.79
20	5.58	1.65	35.93	6.08	1.78	41.03
50	5.40	1.62	35.81	6.21	1.89	40.98

Table 3: Average waiting time for on-line version, times 10^3 seconds

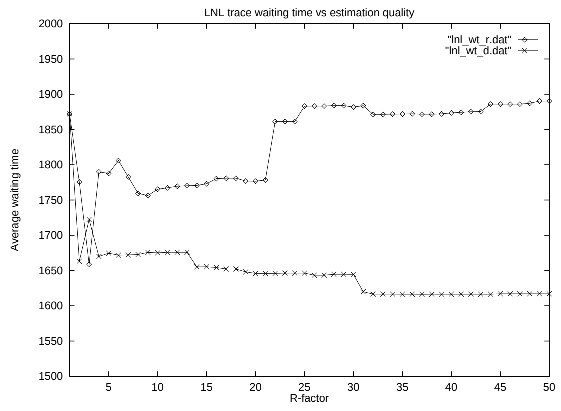

Figure 6: LNL average waiting time vs. R

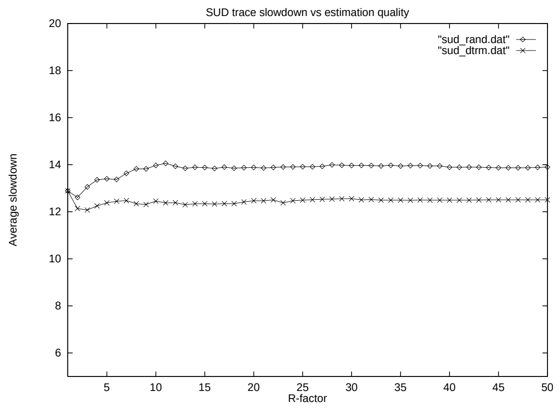

Figure 7: SUD average bounded slowdown vs. R

A sharp drop is observed when R is changed from 1 to 2. The randomized method performs better than the deterministic method until an R value of about 25-30. Finally, the randomized method achieves its best slowdown at an R value of about 10-12, where the improvement over the case of $R = 1$ is almost 25%.

In contrast, we do not have a pronounced minima in case of the artificial trace, where only a 6% decrease in slowdown is observed in the deterministic case, and randomized R is worse than accurate estimates. This poor performance is easily explained by the proportionately smaller number of small jobs in the trace. Recall that the vast majority of jobs in the real traces were small, and that small jobs benefit more from increased R values.

We also performed an offline simulation in order to get a sense of the best achievable backfilling schedule. Jobs are all given identical submission times, although they are still placed in the queue in the same order. Placing all jobs in the queue at the same time allows the algorithm to perform unlimited lookahead. Delay for all jobs is calculated from the same moment at the

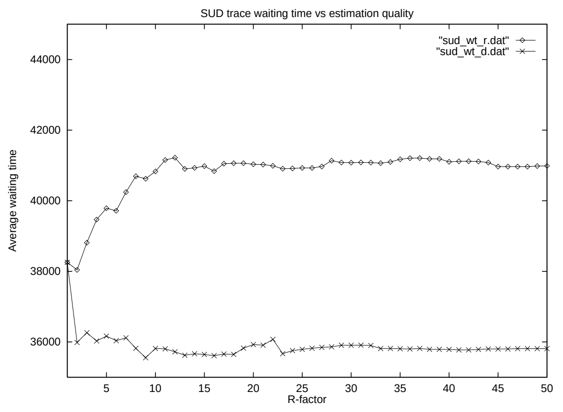

Figure 8: SUD average waiting time vs. R

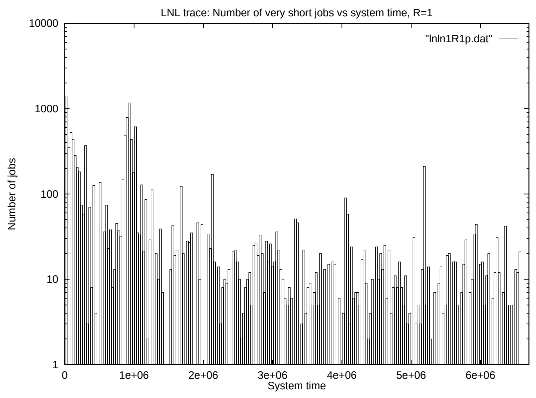

Figure 9: LNL: Number of short jobs at time T versus T, $R = 1$

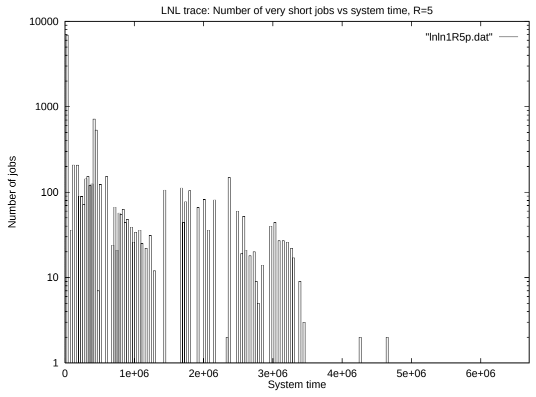

Figure 10: LNL: Number of short jobs at time T versus T, $R = 5$

beginning of the run. In the real system, the only information available to the algorithm is information about jobs that are currently in the queue.

Tables 4 and 5 show average slowdown and waiting times for the off-line case. The results show that the total execution time is almost independent of R. Although larger R values give consistently worse utilization, and thus longer execution times, the difference is small. In the LNL trace, for example, $R = 1$ gives $6.87 \cdot 10^6$ seconds, whereas $R = 5$ gives $6.91 \cdot 10^6$. We speculate that the increased lookahead allows gaps to be filled effectively at any R, removing the pressure to increase gap size by increasing R.

4.1 Discussion

The most important result is the dramatic improvement in slowdown with increasing R for the real traces. By contrast, there is only limited improvement in the artificial trace. This, along with the fact that the completion time of the last job is almost constant for different R, leads us to the conclusion that jobs are *rearranged* by the algorithm so that shorter jobs come first. To verify this, we plotted histograms of the number of very short jobs (less than one minute in length) versus wall clock time in Figures 9 and 10. Note that the number of short jobs decreases with time for $R = 1$ (Figure 9), but the decrease is much more pronounced for $R = 5$ (Figure 10).

The observed dependence of average slowdown on the number of small jobs can be explained by noting three factors. First, small jobs affect slowdown the most simply because of their number. Second, a given delay will increase the slowdown for a small job more than for a larger job. Finally, small jobs fit better into gaps in schedules. The backfilling algorithm was designed with system utilization in mind, and does not explicitly try to optimize average slowdown. The algorithm therefore does not directly attempt to minimize the waiting time of small jobs, even though they largely determine overall slowdown.

Intuitively, increasing R increases execution time estimates of every job in the system. In terms of the algorithm's behavior, increasing R with a fixed shadow time decreases the chance that a job will fit into an existing gap, and therefore also decreases the chance that a job is chosen to be backfilled. On the other hand, jobs will finish proportionately earlier than expected, creating larger gaps.

This effect does not disappear in the off-line version of the algorithm. Instead, it becomes more dramatic. The total execution time of the system is not changed, but the jobs are clearly rearranged. Since the actual gaps in the schedule depend only on the actual execution times of jobs, large jobs are less and less likely to fit into gaps with increasing R. Thus, the system essentially *sorts* (or, "filters") jobs by job size. The same is also true for the online version because it differs from

R-factor	KTH_{dtm}	LNL_{dtm}	SUD_{dtm}	KTH_{rnd}	LNL_{rnd}	SUD_{rnd}
FCFS	265.1	163.6	17.02	265.1	163.6	17.02
1	76.82	48.58	7.59	76.82	48.58	7.52
2	19.16	23.97	6.70	26.56	21.85	6.26
3	15.19	16.37	6.82	19.82	21.38	6.43
5	13.26	13.54	6.98	19.09	14.47	6.61
20	14.24	13.15	7.38	14.23	11.78	7.55
50	15.41	12.25	7.51	14.05	7.52	7.74

Table 4: Average slowdown for off-line version, times 10^3

R-factor	KTH_{dtm}	LNL_{dtm}	SUD_{dtm}	KTH_{rnd}	LNL_{rnd}	SUD_{rnd}
FCFS	14.86	3.99	46.12	14.86	3.99	46.12
1	7.36	1.82	28.07	7.36	1.82	28.07
2	3.97	1.30	26.00	4.28	1.28	26.56
3	3.35	1.05	25.70	3.68	1.26	26.40
5	2.78	0.99	25.43	3.29	1.05	26.25
20	2.45	0.93	25.39	2.56	0.88	26.01
50	2.42	0.86	25.35	2.40	0.71	25.88

Table 5: Average waiting time for off-line version, times 10^6

the off-line algorithm only in having a shorter dynamic queue. This sorting is not absolute (as we see from Figures 9 and 10), but the effect on slowdown is dramatic nonetheless because of the extremely large number of small jobs.

5 Extensions to schedulers

Our results suggest two approaches to improving the performance of backfilling schedulers. First, Tables 2 and 3 show that both average slowdowns and average waiting times for the real traces can be improved by using non-unit R values. This suggests the straightforward approach of multiplying user estimates by a constant factor when they enter the system. While there are clear advantages to doing this, the disadvantage is that we can not make as strong a claim about slowdown or waiting time as can be made for conservative backfilling. By multiplying estimated execution times by a constant factor, the initial scheduled running time for a job is likely to be higher than if unaltered estimated execution times were used. However, even conservative backfilling does not allow claims to be made on worst-case performance. Instead, the use of backfilling merely allows one to make the guarantee that once a job's running time is assigned, it is never pushed back. Qualitatively, our guarantee is no different.

The second approach is to attack the primary metric, average slowdown, directly. If average slowdown is to be the metric of choice, it makes sense to prioritize jobs based on their overall influence on average slowdown. As short jobs clearly affect this metric more than long jobs for given slowdowns, we can improve average slowdown by prioritizing short jobs ahead of long jobs.

We implement a simple prioritization by maintaining waiting jobs in a queue sorting strictly by ascending estimated execution times. Tables 6 and 7 show average slowdowns and waiting times for sorted FCFS and backfilling (BF) approaches (although "no backfilling" might be a better name for the FCFS runs). For all traces, slowdown drops considerably. Most interesting is the fact that variations of R now have either no or a negative influence on results. The disadvantage of this approach is that worst-case guarantees to the user become correspondingly weaker. Essentially, we are guaranteeing only best-effort scheduling.

Table 8 shows the results of one last variation, which adds a guarantee proportional to the *shadow time* discussed in Section 2 to the sorting approach. The first and last rows repeat backfilling sorting results in order to provide context. The second row shows the result of the sorting backfilling approach, subject to the con-

Algorithm	KTH	LNL	SUD
FCFS	8800.23	1600.43	640.25
BF R=1	70.78	43.30	12.89
BF R=2	69.16	35.53	12.14
BF R=3	66.14	37.83	12.07
BF R=5	64.26	36.24	12.38
SortFCFS	36.55	33.38	21.83
SortBF R=1	22.97	20.28	7.77
SortBF R=2	23.54	19.57	7.53
SortBF R=3	25.27	19.46	7.45
SortBF R=5	24.24	20.97	6.96

Table 6: Comparison of slowdown for different algorithms

Algorithm	KTH	LNL	SUD
FCFS	459	37.6	1740
BF R=1	6.33	1.87	38.25
BF R=2	5.62	1.66	35.98
BF R=3	5.52	1.72	36.26
BF R=5	5.50	1.67	36.17
SortFCFS	14.26	3.13	279.3
SortBF R=1	3.92	1.13	77.46
SortBF R=2	3.98	1.11	74.40
SortBF R=3	3.92	1.11	72.27
SortBF R=5	3.88	1.15	65.10

Table 7: Comparison of average waiting time (times 10^3)

straint that no job can be delayed past the time that would have been guaranteed by the default backfilling scheduler. Call the duration between a job's admittance and the time that would have been guaranteed by the default backfilling scheduler a *shadow interval*. The third row of Table 8 shows the results when the constraint is that no job can be delayed more than two shadow intervals past the job's initial time of admittance, and the fourth row shows the analogous data for three shadow intervals. For the shadow cases, R is equal to 1. Different values of R appear to have little effect on the constrained sorting results.

The results show that sorting by job duration is much more effective than simple backfilling even when subject to constraints. The constrained results are significantly better than the default backfilling approach for both real traces, although the advantage is only 13% for the synthetic traces with uniform distribution. However, the constraint clearly affects performance. The average slowdown for the best constrained sorting for KTH is still more than twice that with unconstrained sorting.

5.1 Metrics

Sorting jobs (within some window) by length is an effective way to decrease average slowdown. However, whether average slowdown is really the proper metric for a workload with jobs of different widths is an open question. The metric assumed in this paper, and others, is *bounded* average slowdown, i.e. average slowdown moderated by a minimum run length:

$$slowdown = \frac{\sum_{i=1}^{J} \frac{resp(i)}{\max(B, exec(i))}}{J} \qquad (1)$$

Algorithm	KTH	LNL	SUD
BF R=1	70.78	43.30	12.89
SortBF, 1 shadow time	49.40	29.99	11.96
SortBF, 2 shadow times	41.72	25.39	10.84
SortBF, 3 shadow times	46.77	23.76	10.57
SortBF R=1	22.97	20.28	7.77

Table 8: Slowdown comparison with shadow

where J is the number of jobs, B is the minimum job length, $resp(i)$ is the response time of the application (delay plus execution time), and $exec(i)$ is the actual execution time. However, this equation treats differently shaped jobs that perform the same computation and finish at the same time very differently.

For example, assume job i has width 1 (needs one processor to run), runs for 100 seconds, and is delayed 1000 seconds before starting. Job j has width 10, runs for 10 seconds, and is delayed 1090 seconds before starting. Slowdowns computed with Equation 1 will be 11 for job i and 110 for job j, even though both use the same total resources, potentially perform the same computation (assuming perfect parallel speedup), and complete at the same time.

A more appropriate metric might be the following:

$$slowdown = \frac{\sum_{i=1}^{J} \frac{resp(i)}{width(i) * \max(B, exec(i))}}{J} \qquad (2)$$

where $width(i)$ is the number of processors that the job requires. With this new formula, both of the above jobs will have an identical slowdown of 11. A sorting scheduler would then sort based on slowdown divided by width, instead of just slowdown.

6 Conclusions

Backfilling is a widespread technique used to improve system utilization and decrease average slowdowns for batched schedulers. These gains are achieved by allowing short jobs to run when the system is otherwise idle, provided that they will not delay jobs that arrived earlier.

Backfilling requires users to provide an estimate of the execution time for each submitted job. This paper has characterized the effect of inexact execution time estimates on average slowdown and waiting times. Our approach consists of systematically looking at slowdown for different random and deterministic modifications of actual execution times. We presented offline experiments that show that the actual effect of increasing estimated times is to frontload short jobs, thereby decreasing average slowdowns.

We verified that large overestimation of the job execution time leads to better average slowdown. We show that this anomaly is caused by the confluence of three factors. First, real jobs traces have an overwhelming number of small jobs. Second, small jobs benefit more than larger jobs from backfilling. Finally, the delay incurred by small jobs affects their slowdown, and hence overall average slowdown, more than large jobs.

We have presented two techniques for improving schedule quality in backfilling schedulers. The first systematically stretches estimated execution times in order to open up larger gaps for backfilling. Recent work by Talby et al. [8] and Hollingsworth et al. [4] takes similar approaches. The former also accomodates priorities, while the latter targets cooperation between loosely-coupled systems.

Our second technique explicitly targets slowdown by sorting waiting jobs by length. The resulting schedules are highly efficient because the small, front-loaded jobs are easy to backfill. The main disadvantage of this approach is that we cannot guarantee to avoid starvation. We investigated a modification of the basic sorting approach that restricts queue order from delaying any job past set bounds. The resulting schedules are less efficient than those resulting from unconstrained sorting, but significantly better than those resulting from current backfilling approaches with similar guarantees.

References

[1] D. Feitelson and A. Weil. Utilization and predictability in scheduling in IBM SP2 with backfilling. In *Proceedings of the 12th International Parallel Processing Symposium*, 1998.

[2] D. G. Feitelson and M. A. Jette. Improved utilization and responsiveness with gang scheduling. In *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[3] 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*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[4] J. K. Hollingsworth and S. Maneewongvatana. Imprecise calendars: an approach to scheduling computational grids. In *The 8th High Performance Distributed Computing Conference*, pages 352–359, 1999.

[5] D. Lifka. The ANL/IBM SP scheduling system. In *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science. Springer-Verlag, 1995.

[6] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2), 1997.

[7] J. Skovira, W.Chan, H. Zhou, and D. Lifka. The EASY - LoadLeveler API project. In *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science. Springer-Verlag, 1996.

[8] David Talby and Dror G. Feitelson. Supporting priorities and improving utilization of the IBM SP scheduler using slack-based backfilling. In *Proceedings of the 13th International Parallel Processing Symposium*, 1999.