Randomization, Speculation, and Adaptation in Batch Schedulers

Dejan Perković Peter J. Keleher

Department of Computer Science
University of Maryland
College Park, Maryland 20742
{*dejanp|keleher*}*@cs.umd.edu*

This paper proposes extensions to the backfilling *job-scheduling algorithm that significantly improve its performance. We introduce variations that sort the "backfilling order" in priority-based and randomized fashions. We examine the effectiveness of guarantees present in conservative backfilling and find that initial guarantees have limited practical value, while the performance of a "no-guarantee" algorithm can be significantly better when combined with extensions that we introduce. Our study differs from many similar studies in using traces that contain user estimates. We find that actual overestimates are large and significantly different from simple models. We propose the use of speculative backfilling and speculative test runs to counteract these large overestimations. Finally, we explore the impact of dynamic, system-directed adaptation of application parallelism. The cumulative improvements of these techniques decrease the bounded slowdown, our primary metric, to less then 15% of conservative backfilling.*

1. Introduction

We present a study of batch schedulers based on the conservative backfilling [2] algorithm. A *batch scheduler* is a scheduler for non-interactive jobs. Such a scheduler manages resources on large cluster(s) of dedicated computer nodes interconnected with high-speed networks that must be shared among many end users. Jobs have a large variety of shapes. Some are highly parallel, while others are sequential. Some last a few seconds, while others can last a day. Scheduling those jobs optimally is known to be hard.

Conservative backfilling [2], a commonly used scheduling algorithm, performs much better than a basic first-come-first-serve (FCFS) policy. FCFS suffers from fragmentation, and therefore low utilization, when used alone. Backfilling allows smaller jobs to move forward in the schedule as long as such movement does not cause any other scheduled jobs to be further delayed. This caveat allows backfilling schedulers to guarantee a start time for each job at the time the job is submitted. Although this algorithm performs significantly better then FCFS, a recent study by Zotkin [11] of the performance of the EASY backfilling scheduler [3] indicated that schedules could be significantly improved by sorting the queue of waiting jobs by job length before backfilling.

0-7803-9802-5/2000/$10.00 (c) 2000 IEEE.

We study a variety of backfilling enhancements that can improve scheduling performance even beyond the improvement reported in Zotkin [11]. *Static* approaches require schedulers to assume that jobs have fixed parallelism and running time estimations. *Speculative* approaches allow the scheduler to use modified job parallelism and estimated running times. We use three static approaches: sorting queue by length, *randomization* (with and without sorting the queue by length), and *guarantee-elimination*, which abandons the potentially expensive guarantees maintained in conservative backfilling. We use two speculative approaches: *speculative backfilling*, where the scheduler aggressively speculates on actual job running time, *speculative test runs*, where we run long jobs for a short time to see if they will abort, and *job adaptation*, where the scheduler attempts in several ways to use different job configurations in order to get better response times.

All of these approaches can be combined to form a single scheduler having much smaller average slowdowns and waiting times.

2. Background and experimental setup

We briefly summarize the functioning of a backfilling scheduler. We assume a single physical resource, i.e. a single machine of width (number of processors) n. The scheduler maintains a current schedule of all jobs that have been submitted, but not yet finished running. An easy way to visualize the schedule is as a two-dimensional chart with width n, and a y axis that starts at time *0* and goes up to infinity (Figure 1). The representation of a job is the set of blocks showing the nodes that the job will occupy in each time unit in the future. The schedule shows four already-submitted jobs, together with their scheduled start times. The new job will be given a start time of 7, because it does not fit earlier. However, if job *A* finishes in one time unit instead of two, the new job could execute on the last three nodes starting at time 2. Scheduling in advance can only be done because users provide estimates of the running time of their jobs. Estimates are notoriously imprecise, so users are encouraged to provide generous estimates. Any jobs that exceed estimated running times are killed.

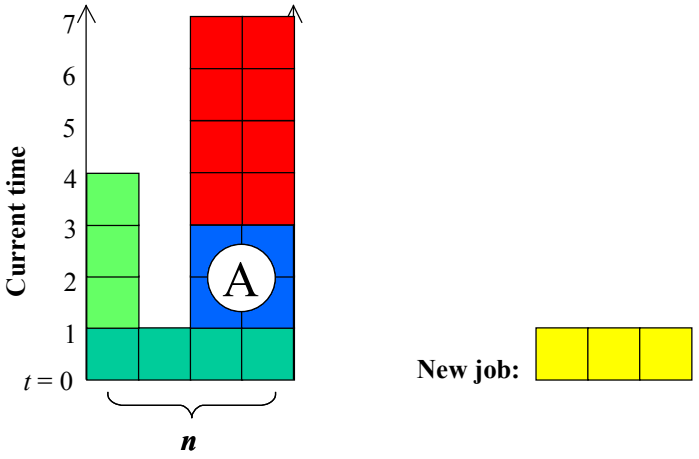

Figure 1: Backfilling scheduler: The new job will be given a guaranteed start time of 7, although it may start sooner if other jobs finish early.

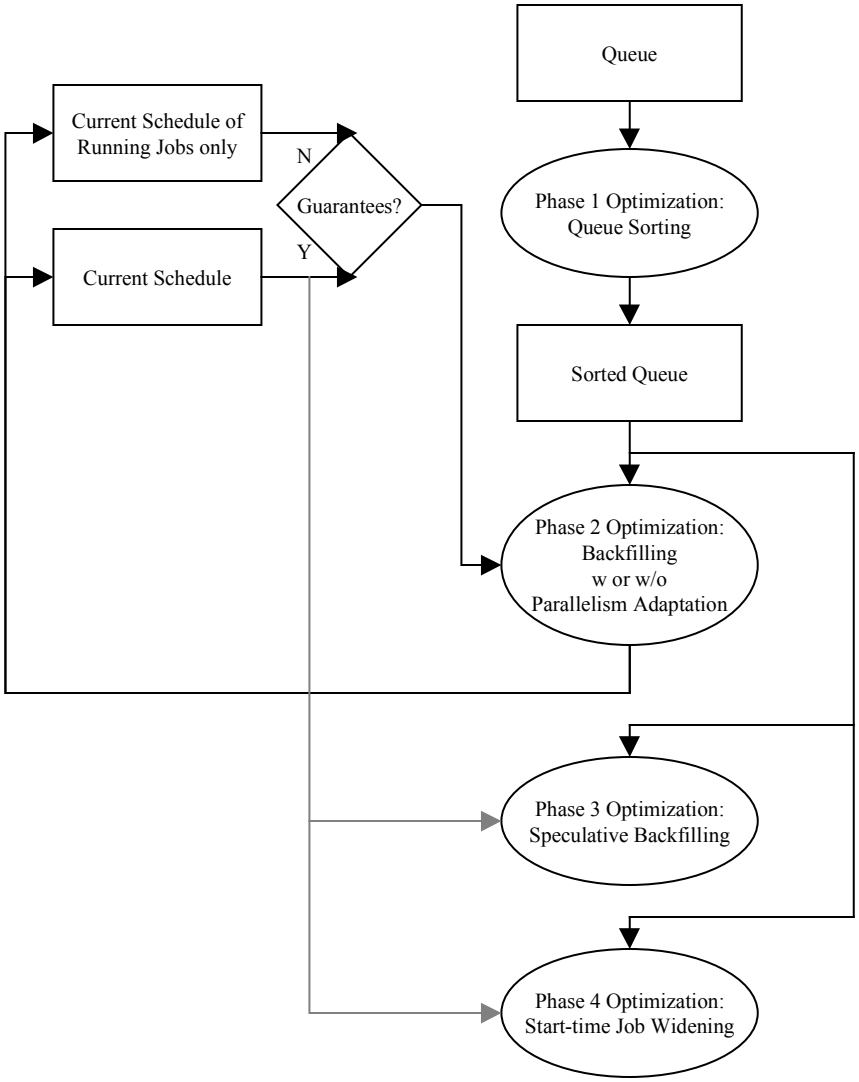

Figure 2: Flow Chart of simulated optimizations – Executed whenever a new job comes or a running job finishes earlier than it estimated.

Number of nodes	430
Number of jobs	79302
Average job parallelism	11
Median job parallelism	2
Average running time	10984
Average estimated running time	24324
Median estimated running time	10800
Average overestimation factor	176
Median overestimation factor	3.90

Table 1: Trace Characterization – All durations are expressed in seconds

Conservative backfilling [2], which we use in this paper, employs start-time guarantees. Each job in the schedule is guaranteed to start no later than at its currently scheduled start-time. Whenever a new job arrives, it is scheduled at the earliest possible time so as not to interfere with guaranteed start-times of all other jobs in the schedule. Whenever a job finishes earlier than estimated, conservative backfilling attempts to reschedule each waiting job in the arrival order to an earlier than its current guaranteed start-time.

We show the flow chart of the simulated optimizations in Figure 2. These optimizations are independent, and can be combined as shown in the Figure 2. The scheduler executes all but one of these optimizations as well as the original conservative backfilling algorithm sequentially every time a new job arrives or a running job finishes earlier than it expected. For example, Queue sorting can be applied with any other optimization technique such as speculative backfilling. Conservative backfilling would be represented only by the "Phase 2", which would use the unsorted queue of waiting jobs, and the "Current Schedule" implied by the use of guarantees. All of the optimizations will be explained in detail in the following sections.

Our results are produced via simulation, using as input a one year long trace from the Cornell Theory Center's 430-node cluster. The CTC trace is quite similar in all but one respect to the Swedish Royal Institute of Technology in Sweden and Lawrence Livermore traces used in prior work [3, 11]. The sole difference is that the CTC trace contains actual running time estimates supplied by users. Traces in previous backfilling studies did not contain this information. Although we use only a single trace, this trace covers an entire year of a large center that accommodates a wide range of users. In combination with the similarity of this trace to the other two mentioned above, we are confident that the CTC trace is reasonably representative of large batch installations.

We summarize the trace characteristics in Table 1. Note that the average overestimation factor, the ratio of estimated to actual running times, is 176, while the median overestimation factor is only 3.9. This implies a large number of very short jobs that had large estimates. We find that the distribution of overestimation significantly differs from simple models used to synthetically generate estimated running times such as constant or uniformly distributed overestimation. In general, jobs that are estimated to be long have smaller median overestimation, but higher average and maximum overestimation than jobs that are estimated to run short. We show some trace characteristics grouped by job shapes in Table 2b.

Our primary metric is bounded slowdown although we use delay (elapsed time between the submission and the start of job execution) and response time (elapsed time between the submission and the end of job execution) as well. The bounded slowdown of a job is defined as:

$$BoundedSlowdown = \frac{FinishTime - SubmitTime}{Max(ActualRunningTime, BoundTime)}$$

We use a bound time of 10 sec. Bound time is used to limit the influence of short jobs on average of the metric.

3. Queue sorting and randomization

In addition to sorting the backfilling queue by length [11], we introduce several new sorting criteria. The main factor of these criteria is randomization. We use two types of randomized orders: priority-based and fully-randomized (referred to hereafter as `random`). In both cases, priorities are randomly assigned to jobs and sorting reflects

	Number of Jobs	Average Overestimation Factor	Number of jobs with zero runtime
Short-Narrow (SN)	14486	22	23
Short-Wide (SW)	25702	14	2
Long-Narrow (LN)	25526	460	108
Long-Wide (LW)	13588	110	1

Table 2: Trace Characterization by Job Shapes – Jobs are grouped by median job parallelism and median job overestimation factor

Average	Median	15% of jobs	85% of jobs
16.33	3.89	<1.67	<14.83

Table 3: Ratio of guaranteed and actual delays for job's that did not start immediately

priorities. With `priority`, the same randomly-assigned order assigned to a job for the entire the job is in the queue. With `random`, on the other hand, priorities change every time backfilling is done. Both priority and randomized sorting criteria refer to these as priorities. Each of these two sorting criteria can be further combined with sorting by length. One way to do this is to construct a sorting criterion that divides job priority by estimated job running time (descending order is assumed).

The motivation for randomization in queue ordering lies in the distribution of job shapes in the queue. Short jobs with small parallelism (those requiring few processors) in the front of the queue are quickly backfilled and run. The result is a queue that is frontloaded with long, awkwardly-shaped jobs followed by short jobs with large parallelism, and finally short jobs with small parallelism. Any induced randomization makes the shape distribution more uniform, with the net effect of short jobs being moved forward in the queue. Recall that these jobs have a disproportionate effect on average slowdowns.

4. Backfilling with no guaranteed times

Conservative backfilling, even as modified in Section 3, employs delay guarantees. Guarantees prevent starvation *and* provide an indication to users of when their jobs will run. Since guarantees limit the ability of algorithms to reorder jobs, we briefly investigate their value by looking at their usefulness as a measure of actual delays. Table 3 shows the statistics on the ratio of initial guaranteed delay to actual delay for jobs that did not start immediately in conservative backfilling (`BF`). It can be observed that both average and median ratios are large. Even more important is the wide range of these ratios: 70% of the jobs that did not start immediately have ratios between 1.67 and 14.83 for conservative backfilling.

An average of 61% of jobs start immediately. In other words, the majority of jobs have zero delay even without having guarantees. We draw two conclusions. First, guarantees might be necessary for only a minority of the jobs, and second, guarantees are quite inaccurate as a predictor of when jobs will run.

An algorithm that dispenses with guarantees could do a complete rescheduling at every execution of the backfilling algorithm. This was described in [2], but not evaluated. The performance of this algorithm would heavily depend on the sorting criterion and actual job characteristics. Using arrival order, this algorithm could significantly degrade performance by attempting backfilling of long-wide and long-narrow jobs first while removing guarantees of short jobs in the queue.

With a randomized order, the effects depend on the characteristics of long-narrow jobs. If long-narrow jobs are long enough to have longer delays than short-wide jobs, and therefore better queue position, the `no-guarantee` algorithm would reinforce the benefit for short-wide jobs given by randomized and priory orders. By removing guarantees for long jobs, it would allow more short-wide jobs to be backfilled. If long-narrow job delays are not long enough, short-wide jobs could be hurt by removal of their guarantees and scheduling long-narrow jobs instead. Nonetheless, both short-wide and long-narrow jobs, which might not be as long as estimated (see Table 2), would benefit by having better chance to be at the front of the queue than in conservative backfilling, where they would be behind the long-wide jobs. In the case of shortest-first order, `no-guarantee` algorithm could help by removing guarantees for long jobs, and allowing more short jobs to be backfilled.

One problem that we need to address is the possibility of starvation. For example, long-wide jobs could starve if the high load situation persists forever. The users with large jobs would certainly not appreciate possibility of job starvation, while the users with small jobs would care less as long as the sorting criterion is not explicitly against short jobs. The solution to the starvation problem is in modifying the sorting criterion to use the sum of the target sorting criterion and a weighted delay. For exam-

Algorithm	Sorting Criterion	Abbrev for guarantee alg.
Backfilling	D	BF
Priority	P	P
Random	R	R
Shortest-First	1/L	1/L
Priority + Shortest-First	P/L	P/L
Random + Shortest-First	R/L	R/L

Table 4: Queue sorting algorithms – D, P, R, and L are delay, priority, random number, and estimated job running time, respectively

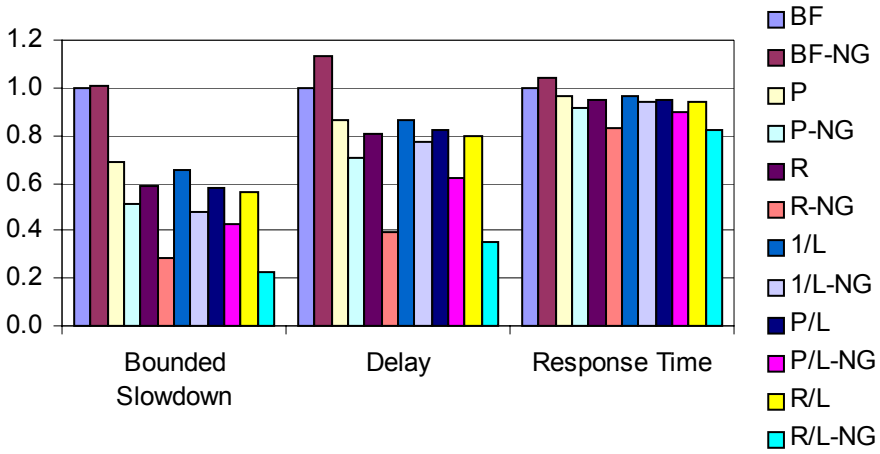

Figure 3: Comparison of algorithm mixes

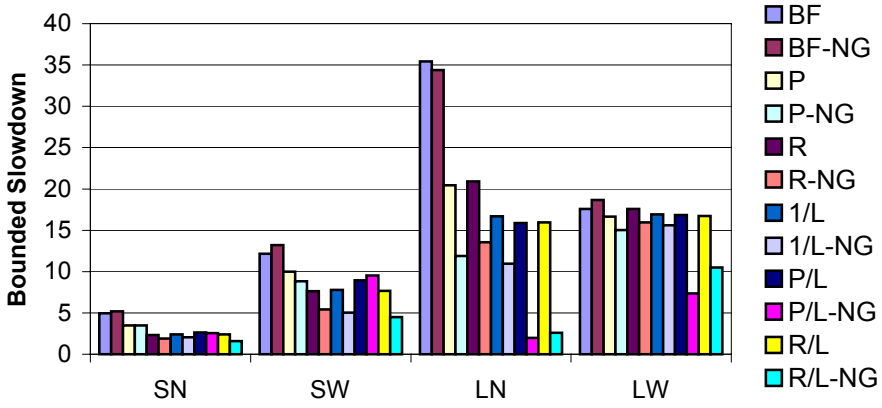

Figure 4: Bounded Slowdown for four shape groups: Short-Narrow, Short-Wide, Long-Narrow, and Long-Wide

ple, `random` algorithm would use $RandomNumber + Weight * Delay$ as sorting criterion, where Weight is a small constant set by the system, and RandomNumber and Delay are the current random priority assigned to a job and its current delay. By using a low weight, the weighted delay would not affect jobs that are started quickly. Only in persistently highly loaded systems would a long wide job's delay be significant factor in the sorting criterion. If all jobs have large enough delays so that target criterion has low weight, the algorithm would converge to the algorithm with guarantees.

5. Performance of priority, randomized, shortest-first, and `no-guarantee` algorithms

We use a naming convention for the algorithms to reflect the formula used as sorting criterion (assuming descending order) of the algorithms. Table 4 shows the formulae for sorting criteria and short names for each of the algorithms. `No-guarantee` algorithms will have –NG suffix.

Figure 3 shows the performance of `guarantee` and `no-guarantee` algorithms with different sorting criteria. Randomized backfilling performs better than priority backfilling with or without guarantees. As expected, `no-guarantees` algorithms performed better with all sorting criteria except arrival order.

As in [11], it can be noted that shortest-first order significantly decreases slowdown. Priority backfilling (with 3 priorities only) performs slightly worse than shortest-first backfilling, while randomized performs better. The combination of these sorting criteria and a `no-guarantee` strategy gives even better results. The bounded slowdown is decreased for 57% and 77% from conservative backfilling, and 35% and 65% from shortest-first backfilling for `P/L-NG` and `R/L-NG`, respectively. The delay is decreased for 38% and 65% from conservative backfilling, and 28% and 59% from shortest-first backfilling for `P/L-NG` and `R/L-NG`, respectively.

The reason why the backfilling algorithm with shortest-first sorting criterion benefits from added randomization lies in large slowdowns of large-narrow jobs as compared to slowdowns of short jobs. These slow-

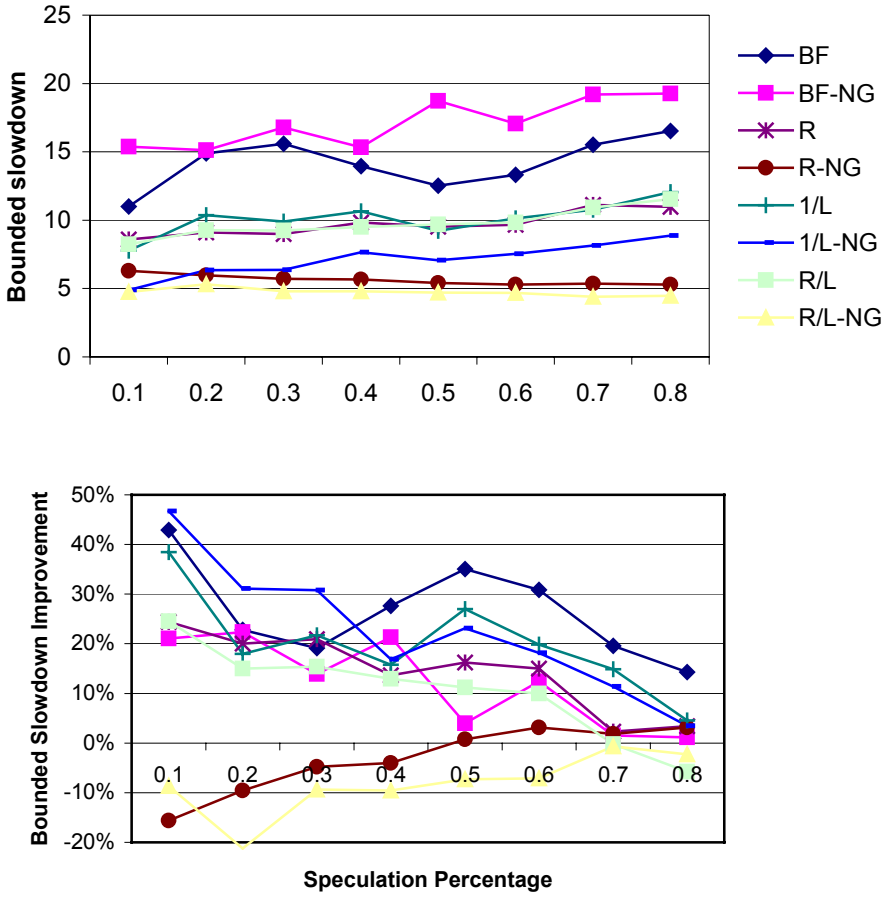

Figure 5: Speculative backfilling for different speculation percentages. Improvements are relative to corresponding algorithms without speculation.

downs are due to large overestimation for many long-narrow jobs. Randomization could hurt, if most of long-narrow jobs had smaller slowdowns than short jobs since these have later position then short jobs in the shortest-first ordered queue.

Figure 4 shows bounded slowdowns for each of the shape groups. As compared to conservative backfilling, there are improvements in all groups. Improvements for long wide jobs are smallest since it is hard to directly help them without sacrificing performance of most other jobs.

Overheads of these algorithms will be discussed in section 8.

6. Speculative backfilling

Although the average number of nodes available when there are jobs in the waiting queue is not large, even small improvements in utilization during load bursts can lead to large improvements in average performance. Since user estimates of job running times tend to be much larger than actual running times, an appealing extension of the basic backfilling approach is to speculate that actual job running times might be shorter than estimated. By using a shorter than estimated running time for a job, a *speculated running time*, the job could be started, and thus have shorter delay, when otherwise its estimated running time would prevent it. User estimated running times, however, may be more precise than ones the algorithm speculates, and in such a case the job that is executed with speculative backfilling may have to be killed and rescheduled again. The speculative backfilling is implemented in a separate phase after the regular backfilling; it iterates through all jobs left in the queue after the regular backfilling phase in the same order, and therefore, does not significantly increase algorithm complexity. Speculative backfilling is applicable only to batch jobs that do not have unrecoverable side effects like synchronization with other jobs.

In our algorithms, the speculated time increases to fit the length of the hole. If a speculatively run job runs longer than its speculated time, it is killed and returned to the queue for future scheduling. That job would be eligible for future speculative execution with speculated time equal to the average of its last speculated time and its estimated time. The backfilling of a job copy that keeps its guaranteed time (in `guarantee` algorithms) is suspended.

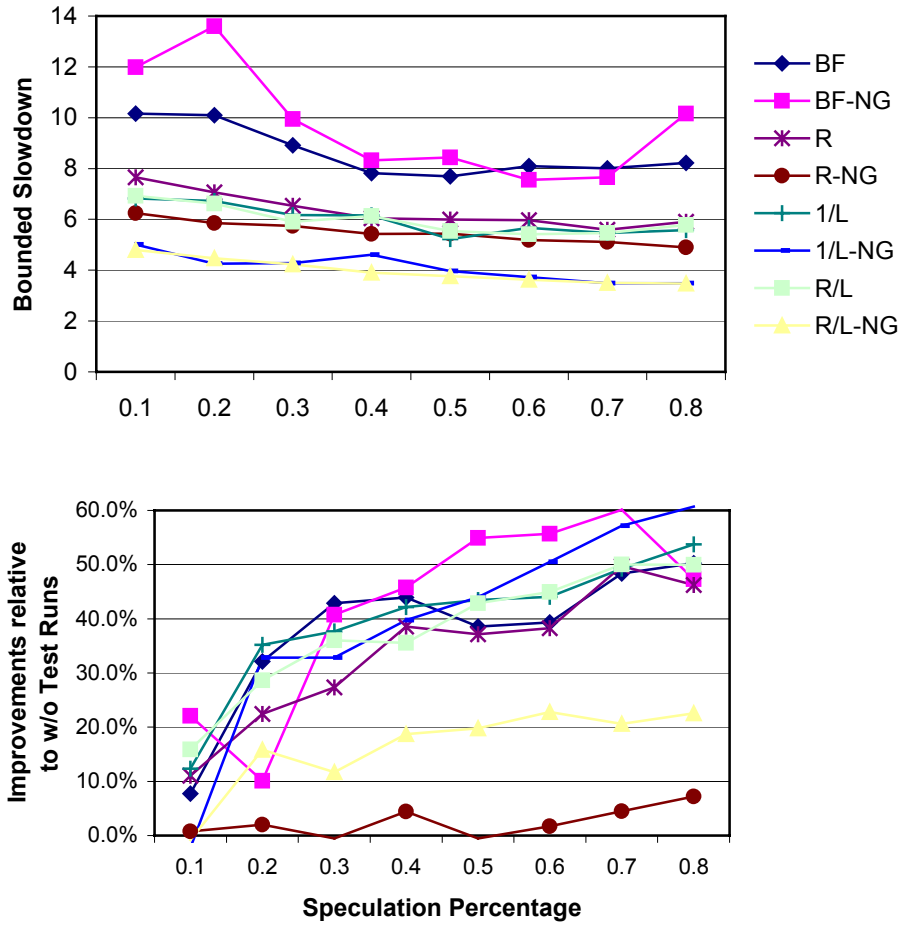

Figure 6: Speculative Backfilling with Test Runs. Improvements are relative to Speculative Backfilling w/o test runs.

6.1 Speculative test runs

The presence of large running time overestimations in the trace indicates the possible presence of applications that are still in development, and therefore, have high probability to die prematurely either because of bugs or because they run in new environment. Those applications create huge slowdowns for themselves as well as make the backfilling schedule busy enough to create large slowdowns for other applications.

Recognizing that the biggest problem in this class of applications comes from jobs that are estimated to run for a long time, we use *speculative test runs* for long jobs. Speculative test runs are part of speculative execution. If a long running job (e.g., longer then 3 hours) cannot be speculatively started, the scheduler attempts to run it for a short time, for example, 5 minutes, and not more then 15 even if there is space for more. If the job finishes in that short time, speculative test runs will have significantly improved response time and slowdown. If the job does not finish in that time, speculative test runs will have wasted some capacity, and the job will have to be restarted again. The wasted capacity, however, is about 2-10% of the minimal estimated job time required for test runs (3 hours in the example) on the appropriate number of nodes. Nonetheless, this is done in speculative scheduling phase, so few of these wasted cycles would have been useful anyway.

6.2 Performance

Figure 5 shows bounded slowdowns for speculative backfilling and corresponding improvements relative to without speculative backfilling. Both figures show the performance for different speculation percentages. The speculation percentage is the ratio between the speculated and user-estimated running times.

The bounded slowdowns, as shown in the Figure 5, tend to be smaller with smaller speculation percentage. The reason for this is that with smaller speculation percentage there are more speculative backfilling attempts, and although the success rate of the speculative backfilling falls, the overall number of successful speculative backfillings is increased. Small speculation percentages tend to act like long test runs: they are easy to specula-

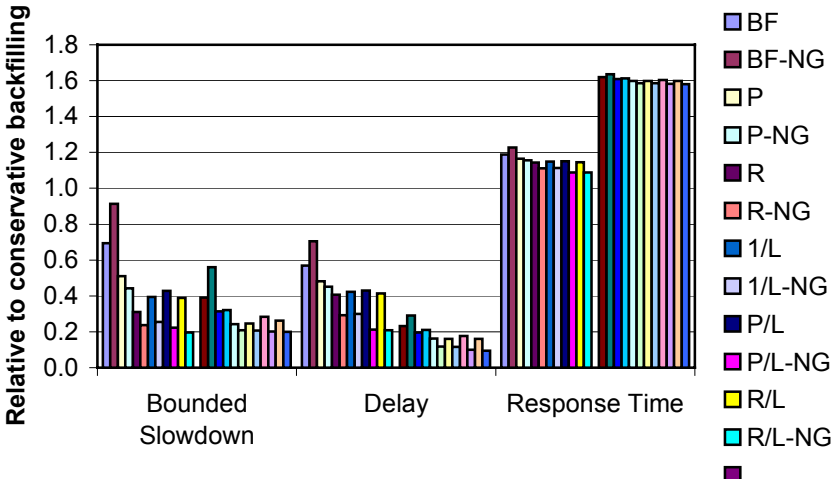

Figure 7: Half-sized (left sets) and Quarter-sized (right sets) jobs

tively backfill, and if successfully completed, the bounded slowdown improvements are much larger then the relatively small cost per speculative backfilling when they do not succeed.

Figure 6 shows bounded slowdowns for speculative backfilling with test runs, and the corresponding improvements relative to the speculative backfilling without test runs. Test runs significantly improved performance of speculative backfilling, and especially so for large speculation percentages.

Figure 6 also indicates that overall performance is little sensitive to speculation percentages for most of the algorithms. In all algorithms, the bounded slowdown slightly decreases with the higher speculation percentages. This is the opposite from the behavior without test runs, as shown in Figure 5. The explanation is that when the short test runs are used, there is little to gain with using short speculated running times to counter the time wasted when actual running times are longer. At the same time, with higher speculation percentage the probability of successful speculation is higher.

7. Application adaptation

A lucrative approach to improve backfilling performance is to shape applications according to the current load conditions. During high load bursts, parallel jobs could be narrowed to use less resources, to improve backfilling chances, and when started, to allow more jobs to start at the same time. In low load situations, jobs could be widened to run shorter, and therefore, occupy fewer nodes in future.

This approach is applicable only if applications expose the possibilities for application molding. They would have to specify both the resources required (number of nodes) and the performance (estimated running time) for each alternative. Active Harmony [Keleher, June 1999] provides such an interface that could be used even for running time molding.

Under this general approach, there is a multitude of actual techniques that can be used. In this paper, we explore only several techniques intended to improve performance of the backfilling scheduler. In all experiments, it is assumed that applications have constant efficiency for all job alternatives that we explore.

7.1 Techniques and Performance

The first technique we consider for exploiting application alternatives uses only one alternative all the time: either half-size or quarter-size alternative. There is no adaptation here after a job has been submitted and its parallelism reduced. Figure 7 shows performance relative to conservative backfilling. The left set of bars in each metric is for the case when half-size alternative is used always, except for sequential jobs. The right sets are for the case when quarter-size alternative is used for all jobs wider than 4 nodes; otherwise, half-size alternative is used for non-sequential jobs. Improvements on bounded slowdown and delay are large, while the average response time is significantly increased.

Reducing parallelism increases chances for jobs to get backfilled, and therefore, decreases their delay. This has a consequence on both short-wide jobs and jobs that are estimated to run for long time, but fail. In that sense, this technique acts towards the same application as speculative backfilling and test runs. On the other hand, reducing job parallelism by factor 2 increases job running time, and incurs slowdown of at least 2. As consequence, many otherwise high slowdown jobs will have their slowdown decreased by large margins, while long jobs will be even longer and incur a large increase on average response times. More parallelism reduction tends to affirm more slowdown benefits for otherwise slower algorithms.

Using half-size or quarter-size alternatives can be very inefficient in low load situations. In fact, many jobs could be widened in low load situations, and therefore, decrease its response time. The greedy adaptation examines obtainable response times for each of job's alterna-

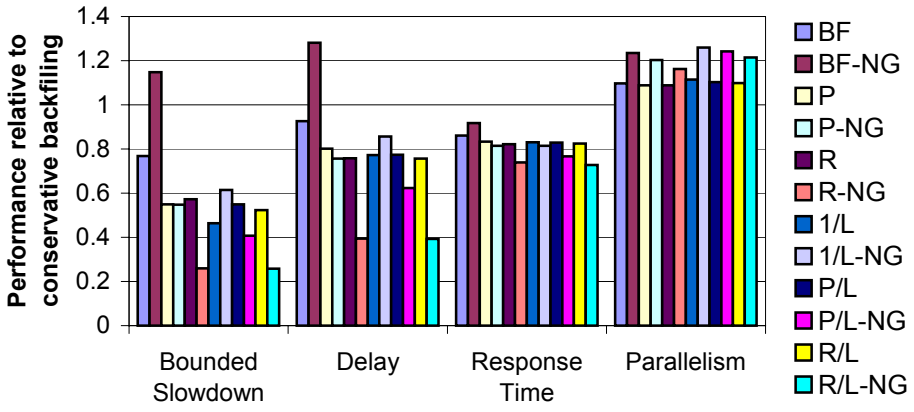

Figure 8: Greedy adaptation

tives at every backfilling attempt for that job, and chooses one with the least response time (if backfilling is possible at all). Figure 8 shows performance of greedy adaptation when using three application alternatives for all jobs: regular, half-size (jobs are narrowed, if possible), and double-size (jobs are widened). Half-size alternatives allow more jobs to run at the same time, though twice as long as in regular alternatives. Double-size alternatives decrease job running times by half as they use twice as many nodes as required in regular alternatives.

Overall best greedy algorithm (one with quarter-size instead of double-size alternative and with restrictions for long jobs) only matches the bounded slowdowns of half-size alternative applied always, and is still slightly worse than when quarter-size job alternative is applied always. We therefore examine a non-greedy adaptation algorithm, named *start time widening,* on top of default half-size alternative that can selectively widen starting jobs. After completing regular backfilling phase, some nodes may still be available. Start time widening uses these nodes by increasing parallelism and decreasing estimated running time of some of the jobs that are scheduled to start. The scheduling algorithm uses an additional phase after regular backfilling phase to perform start time widening. By using different phase, the scheduler avoids the problems shown in greedy strategies for `no-guarantee` algorithms, where long jobs may widen and prevent new short jobs being scheduled.

Figure 9 shows bounded slowdowns and response times for combinations of half-sizing, start time widening and speculative backfilling. As compared with half-sized applications, start time widening gives significant improvements. Start time widening and speculative backfilling, when done together, are done independently and in different scheduling phases, but still using the same backfilling ordering as initial backfilling phase.

We can see that start-time widening improves performance for most of the algorithms when applied together with initial half-sized applications as compared to using half-sized applications only. It also reduces average response time for no-guarantee algorithms where the application of start-time widening has more available nodes. Together with speculative backfilling and initial half-sized application, however, start-time widening improves little or even makes slightly worse bounded slowdown. The explanation is that speculative backfilling takes most of the available nodes and leaves little to start-time widening.

The bounded slowdown for several of the algorithms decreases to around 15% of the bounded slowdown of conservative backfilling. The average delay for R/L-NG algorithm decreases to 21% of the delay for conservative backfilling.

8. Overhead Analysis

All of the backfilling improvement techniques are independent from each other, and thus the overhead of the combined techniques is the sum of overheads of all applied techniques. Conservative backfilling itself has a $O(n^2)$ complexity.

Each of queue sorting algorithms relies on sorting, which is known to have O(n*logn) complexity. All of the queue sorting techniques except R/L, however, can be sorted more efficiently. For example, `Priority` can only do a single insertion into already sorted queue when a new job comes, and a single deletion when a job is started. This has O(logn) complexity. `Random` could use bucket sort to sort random numbers with known distribution to achieve O(n) complexity, too. The maximum queue length observed in the experiments conducted for this paper was approximately 1,000 jobs. Less then 20,000 jobs were delayed in the queue during 1 year long period. Therefore, this kind of overhead should not be considered large.

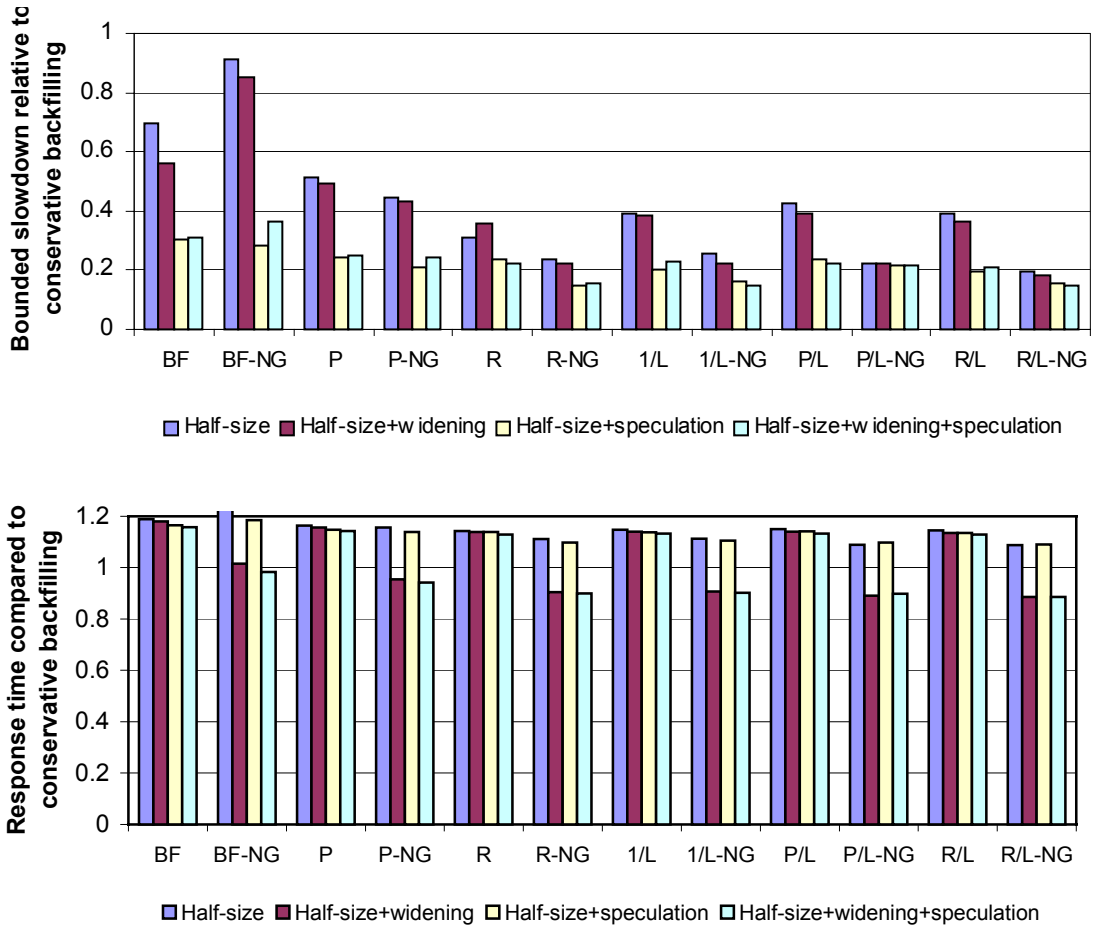

Figure 9: Combination of Half-sizing, Start time widening, and Speculative backfilling

Discarding guarantees from conservative backfilling does not increase the theoretical complexity of the backfilling scheduler assuming that there is no starvation (see the last paragraph of Section 4 for more on how to avoid starvation).

Speculative backfilling has the same complexity as backfilling itself, though we expect it to run much faster due to the condition that job can be speculatively backfilled only if it is to start immediately. Similarly, start-time widening has even smaller complexity due to use of only jobs that are scheduled to start immediately.

Greedy adaptation, as described, has about three times larger complexity due to requirement to find the best obtainable response time by backfilling one of the three available alternatives.

We find none of these techniques to have prohibitive overhead that would prevent it from use.

9. Related work

Lifka [3] introduced EASY backfilling approach for batch scheduling. Conservative backfilling, commonly referred to as backfilling, was introduced by Feitelson and Weil [2]. The main difference between EASY and conservative backfilling is that in EASY it is guaranteed that the first job in the waiting queue will not be delayed while in conservative backfilling all jobs are guaranteed not to be delayed. Both algorithms achieve similar performance [2] while preventing the starvation problem.

Several studies [1, 2, 10] showed that performance improvements achieved by backfilling as compared to first-come-first-served policy are significant. The simulation study [1] showed that backfilling performs close, but slightly worse, to the queue searching policies that do not consider job running times and have starvation problem.

There is a single study [11] that uses queue sorting in the context of backfilling. It shows that mean job slowdowns are much better with shortest first queue sorting. Without considering job running times, the study [1] examined different queue sorting and found that both Least-Processor-First-Served and Most-Processor-First-Served improve processor utilization and mean response time slightly although increasing the response time variance. Study [10] found significant improvements with LPFS. Queue searching, which considers all jobs rather then only the first in the queue, was shown [1] to outperform FCFS and queue sorting techniques. Combinations of queue searching and queue sorting did not yield improvements relative to queue searching alone.

An important element in backfilling scheduling is the accuracy of user estimates of their job execution times. The study [11] shows that inaccuracy could be a virtue with which backfilling can produce better schedules. The study used the estimated times increased by either constant factor or a random factor uniformly distributed in a range. Nonetheless, in real traces running time overestimation is spread over a wide range.

An important issue in backfilling algorithms is predictability of wait times. Guarantees that conservative backfilling gives to jobs allow users to perceive them as pointers to how long their jobs will be delayed. We showed that many of the initial guarantees are much larger then the actual delays. Study [9] has developed a method for delay prediction that may be more useful.

There has been a lot of research on how applications can use different number of processors in order to adjust to current load conditions. Two main categories are adaptive partitioning [5, 7, 8] and dynamic partitioning [4, 6]. Adaptive partitioning algorithms make decisions on job partition sizes before their start. Dynamic partitioning can change job partition size during its running time. Most of these studies use response time as metric and assume that all jobs can run on any partition size with the same efficiency. None of these studies considers job running times.

10. 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.

This paper has presented several strategies for improving average slowdowns and delays in backfilling schedulers. First, we characterize the effect of differently "shaped" jobs on overall throughput, and show that randomization of the queue order allows the system to move awkwardly shaped jobs backwards in the queue, eliminating many bottlenecks and increasing overall utilization. Second, we investigate the value of the backfilling guarantee, and show that the same utility can be achieved much more cheaply through a modified sorting criteria. Third, we investigate two classes of speculative approaches: speculating on shorter running times for jobs with very long running-time estimates, and short bounded "test runs". Both approaches significantly decrease average slowdowns while largely using resources that would otherwise have been wasted. Finally, we present results on the impact of "job shaping," where the width of the job is dynamically adjusted by the system to reflect current demand. We show that using jobs with smaller parallelism and correspondingly longer running times achieves better average bounded slowdowns than using original job shapes.

11. References

[1] K. Aida, H. Kasahara, and S. Narita, "Job Scheduling Scheme for Pure Space Sharing Among Rigid Jobs," in *Job Scheduling Strategies for Parallel Processing*, vol. 1291, *Lecture Notes in Computer Science*, D. G. Feitelson and L. Rudolph, Eds.: Springer-Verlag, 1998.

[2] D. G. Feitelson and A. Weil, "Utilization and Predictability in Scheduling the IBM SP2 with Backfilling," in *12th International Parallel Processing Symposium*, 1998.

[3] D. Lifka, "ANL/IBM SP scheduling system," in *Job Scheduling for Parallel Processin*, vol. 949, *Lecture Notes in Computer Science*, D. G. F. a. L. Rudolph, Ed.: Springer-Verlag, 1995, pp. 295-303.

[4] C. McCann and J. Zahorjan, "Processor Allocation Policies for Message-Passing Parallel Computers," in *Proceedings of the 1994 ACM SIGMETRICS Conference*, February 1994, pp. 19-32.

[5] J. Padhye and L. Dowdy, "Dynamic Adaptive Processor Allocation Policies for Message Passing Parallel Computers: An Empirical Comparison," in *Job Scheduling Strategies for Parallel Processing*, vol. 1162, *Lecture Notes in Computer Science*, D. G. Feitelson and L. Rudolph, Eds.: Springer-Verlag, 1996, pp. 224-243.

[6] J. D. Padhye and L. W. Dowdy, "Dynamic versus Adaptive Processor Allocation Policies for Message Passing Parallel Computers: An Empirical Comparison," *Lecture Notes in Computer Science*, vol. 1162, 1996.

[7] E. Rosti, E. Smirni, L. W. Dowdy, G. Serazzi, and B. M. Carlson, "Robust partitioning policies for multiprocessor systems," *Performance Evaluation*, vol. 19(2-3), pp. 141-165, March 1994.

[8] E. Smirni, E. Rosti, G. Serazzi, L. W. Dowdy, and K. C. Sevcik, "Performance Gains from Leaving Idle Processors in Multiprocessor Systems," in *Proceedings of the 24th International Conference on Parallel Processing*. Oconomowoc, WI, August 1995, pp. III:203-210.

[9] W. Smith, V. Taylor, and I. Foster, "Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance," in *IPPS/SPDP '99 Workshop on Job Scheduling Strategies for Parallel Processing*, 1999.

[10] J. Subhlock, T. Gross, and T. Suzuoka, "Impact of Job Mix on Optimizations for Space Sharing Scheduler," in *Supercomputing '96*, 1996.

[11] D. Zotkin and P. J. Keleher, "Job-Length Estimation and Performance in Backfilling Schedulers," in *The 8th High Performance Distributed Computing Conference*, August 1999.