Backfilling with Guarantees Granted upon Job Submission

Alexander M. Lindsay[1,⋆], Maxwell Galloway-Carson[2], Christopher R. Johnson[2], David P. Bunde[2], and Vitus J. Leung[3]

[1] iBASEt
aml.lindsay@gmail.com
[2] Knox College
{mgallowa,crjohnso,dbunde}@knox.edu
[3] Sandia National Laboratories
vjleung@sandia.gov

Abstract. In this paper, we present scheduling algorithms that simultaneously support guaranteed starting times and favor jobs with system-desired traits. To achieve the first of these goals, our algorithms keep a profile with potential starting times for every unfinished job and never move these starting times later, just as in Conservative Backfilling. To achieve the second, they exploit previously unrecognized flexibility in the handling of holes opened in this profile when jobs finish early. We find that, with one choice of job selection function, our algorithms can consistently yield a lower average waiting time than Conservative Backfilling while still providing a guaranteed start time to each job as it arrives. In fact, in most cases, the algorithms give a lower average waiting time than the more aggressive EASY backfilling algorithm, which does not provide guaranteed start times. Alternately, with a different choice of job selection function, our algorithms can focus the benefit on the widest submitted jobs, the reason for the existence of parallel systems. In this case, these jobs experience significantly lower waiting time than Conservative Backfilling with minimal impact on other jobs.

1 Introduction

Backfilling has been a standard feature of multiprocessor scheduling algorithms since it was introduced by Lifka [7] in the Extensible Argonne Scheduling sYstem (EASY). In a survey of parallel job scheduling, Feitelson et al [4] characterize backfilling with three parameters, the number of reservations or jobs with guaranteed start times, the order of queue jobs, and the amount of lookahead into the queue. In this paper, we describe variations of backfilling where all jobs are given a guarantee upon their arrival, Conservative Backfilling [8]. However, unlike Conservative Backfilling, we are interested in supporting job priorities other than First-Come-First-Serve (FCFS) [10]. Also, while we do not use any lookahead into the queue, one of our algorithms does delay making decisions

⋆ Work done while Alex was a student at Knox College.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 142–153, 2011.

until more data is available. Thus, our algorithms add a fourth parameter, when decisions are made, to the three parameters mentioned above.

A key benefit of Conservative Backfilling is that each job is granted a guaranteed starting time when it is submitted. (It may start earlier, but will not be delayed later than this time.) These guarantees lead Conservative Backfilling to benefit wide jobs, jobs requiring many processors, relative to other backfilling strategies (e.g. [13]). From a fairness standpoint, this guarantee ensures that wide or long jobs, which are less likely to benefit from backfilling, are not harmed by jobs that backfill more easily. These guarantees also make the scheduler more predictable since each user has a bound on when their jobs will run.

Conservative backfilling maintains a profile containing a tentative schedule for all jobs. When a job arrives, it is placed in the earliest possible spot within the profile, i.e. it is scheduled to start at the earliest time that does not disturb any previously-placed job. The only other profile changes occur when a job finishes early, creating a "hole" that potentially allows other jobs to move earlier. In this case, Conservative initiates *compression*, the reexamination of each job in the order of its current starting time in the profile. Each job is removed from the schedule and then reinserted at the earliest possible time. Compression never delays a job since the job can always fit back into the profile at the same spot, but some jobs move earlier, into a hole or spaces vacated by jobs that have themselves moved. Since no job's planned start time is ever delayed, each job's initial reservation is an upper bound on its actual starting time.

Because Conservative compression reschedules jobs based on the profile's order, intuition suggests that it tends to preserve job order, closing holes by sliding the end of the profile earlier. (Of course, job order does change when a job fits into a hole that earlier jobs could not use.) Since the profile is built as jobs arrive, this gives Conservative a FCFS tendency. This is desirable from a fairness perspective, but may not support a specific system's goals. For example, some systems may wish to favor short jobs to improve average response time and systems oriented toward capability computing may wish to favor wide jobs.

Backfilling algorithms have been designed to support these goals (e.g. [14,5,1,11]), but they do so by reordering the profile, which sacrifices the key benefit of Conservative scheduling: its ability to give jobs guaranteed starting times when they are submitted. In this paper, we present scheduling algorithms that simultaneously support guaranteed starting times and favor jobs with system-desired traits. To achieve the first of these goals, our algorithms keep a profile with potential starting times for every unfinished job and never delay these starting times, just as in Conservative. To achieve the second, they exploit previously unrecognized flexibility in the handling of holes that appear in the profile. Specifically, we present two algorithms using the following kinds of flexibility:

– *job selection*: Although Conservative always tries to move the next job in the profile into a hole, any job that fits can be moved into a hole. (This idea is also used in [9].)

– *timing*: Although Conservative closes holes as soon as they form, the scheduler is only required to identify jobs that it wants to start immediately. Thus, some decisions can be deferred until more information (e.g. more job arrivals or early completions) is available.

We analyze our algorithms using an event-based simulator run with traces from the Parallel Workloads Archive [3]. From the traces, our simulator takes an arrival time, a required number of processors, a running time, and an estimated running time for each job. The estimated running time gives the scheduler an upper bound on the job's running time, but most jobs "end early", with actual running time less than their estimate. Throughout, we assume that jobs need exactly the requested number of processors (rigid jobs), that each processor can run at most one job at a time (pure space-sharing) and that each job finishes in exactly its given running time (no interference between jobs).

We find that, with one choice of job selection function, our algorithms consistently yield a lower average waiting time than Conservative while still providing each job a guaranteed start time when it arrives. In fact, in most cases, our algorithms give better waiting times than the more aggressive EASY algorithm [7], which does not provide guaranteed start times. Alternately, with another job selection function, our algorithms significantly lower waiting times for the widest jobs with minimal impact on other jobs.

The rest of the paper is organized as follows. We describe our algorithms in Section 2 and relevant related work in Section 3. Then we give our experimental results in Section 4 and conclude in Section 5.

2 Algorithms

Now we present our algorithms to exploit the flexibility discussed above.

2.1 Prioritized Compression

Our first algorithm is *conservative with Prioritized Compression* (PC). This algorithm maintains two data structures, a profile with the tentative schedule and a *compression queue* of jobs ordered by a system-specific priority function.

When a job arrives into the system, it is placed into the profile exactly as in Conservative and also added to the compression queue. When a job finishes early and creates a hole, PC compresses the schedule by trying to reschedule each job in the order given by the compression queue; it tries to reschedule the first job in the compression queue, then the second, and so on. This differs from Conservative, which considers jobs in the order they occur in the profile, but PC preserves the key feature that no job moves later in the profile; a job accepts rescheduling only when it benefits and a job is only permitted to make moves that do not interfere with any other job.

By using a customized order for compression, PC allows high-priority jobs to benefit from the hole even if they begin much later in the profile. Doing so adds another wrinkle to the compression operation, however. Consider the profile

shown in Figure 1(a); time is on the x-axis, with the current time at the far left. Suppose job A finishes early and is removed. If the resulting profile is compressed with the order E, C, D (Longest Job First), only jobs C and D are rescheduled. This yields the profile shown in Figure 1(b), with job E delayed even though it could also be started. To avoid unnecessary idle time like this, the compression algorithm for PC returns to the front of the compression queue each time a job is rescheduled. (Conservative does not need to do so since rescheduling one job cannot benefit a previously-considered job when the profile order is used.)

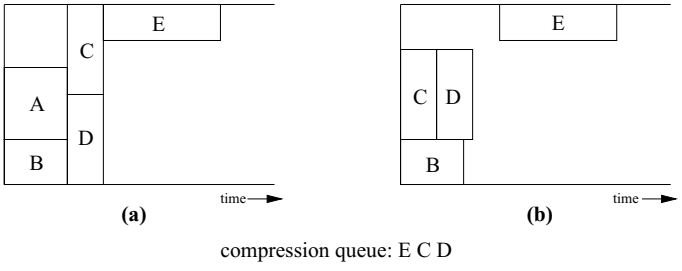

Fig. 1. Profile showing need to return to beginning of the compression queue after each successful rescheduling. (a) Initial profile before job A terminates early. (b) Profile after rescheduling jobs E, C, and D once each in that order.

The downside of returning to the beginning of the compression queue after each successful rescheduling operation is that jobs can be moved more than once. For example, consider the profile depicted in Figure 2(a) and suppose again that job A finishes early. If the profile is compressed with the order D, C (Widest Job First), the first rescheduling operation improves the planned start time of job D, producing the profile shown in Figure 2(b). Once job C is rescheduled, however, job D can be moved again, resulting in the profile shown in Figure 2(c).

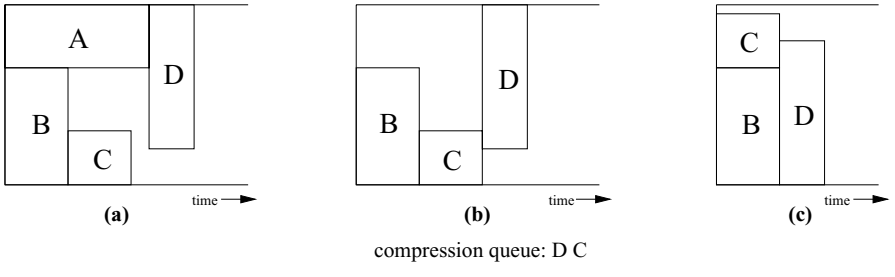

Fig. 2. Example where PC compression moves the same job twice. (a) Initial profile before job A terminates early. (b) Profile after first compression of job D. (c) Profile after compressing job C and then job D again.

Since jobs can move more than once, a natural question is how long compression will take. We return to this question in Section 4.3.

2.2 Delayed Compression

Our second algorithm is *conservative with Delayed prioritized Compression* (DC). It keeps a prioritized compression queue just like PC, but also exploits flexibility in the timing of compression by deferring some rescheduling operations. Specifically, when a job finishes early, DC's compression operation only reschedules jobs that can begin immediately, deliberately leaving holes in the profile. For example, consider the profile depicted in Figure 3(a) and suppose job A finishes early. If DC compresses with order D, E, F (Longest Job First), it would leave the profile as depicted in Figure 3(b), with a hole after job C even though the planned starting time of job F could be improved. By deferring this improvement, algorithm DC leaves itself flexibility in case a high-priority job arrives or another job finishes early. Note that once the running system reaches the hole, the scheduler must fill the hole; this requires an additional check when a job finishes and the profile indicates idle time for some of its processors.

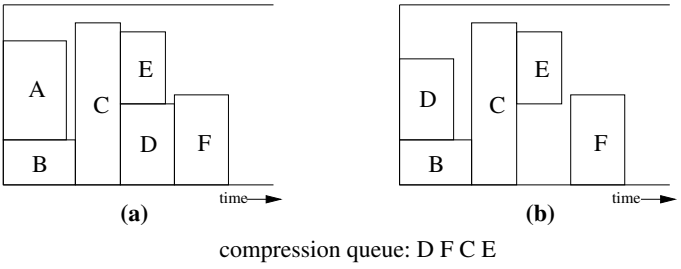

Fig. 3. Example where the DC algorithm deliberately leaves a hole in the profile. (a) Initial profile before job A terminates early. (b) Profile after compression.

One issue with deliberately leaving holes in the profile is that newly-arrived jobs can backfill into them. For example, suppose a short job arrived after the compression operation shown in Figure 3. If this job fits into the hole left when job D was moved, it can backfill there and bypass job F as well as any later jobs. While this backfill operation may be fine if the scheduler wishes to favor short jobs, it can completely undermine the scheduler's priority mechanism if a different priority function is being used. To avoid this, DC also handles job arrivals differently than Conservative. Rather than immediately adding a new job to the profile, DC instead adds it to the compression queue. The algorithm then reschedules any job before the new job in the compression queue whose new start time would be before the estimated completion of the new job, i.e. those higher-priority jobs that could be delayed by the new job. Once the new job is reached in the compression queue, it is scheduled and compression ends. This modified treatment of job arrivals closes holes when necessary to protect rescheduling opportunities for high-priority jobs. In the example shown in Figure 3, DC would reschedule F if a lower-priority job arrives and could be scheduled to finish after the end of C (the earliest possible start time of F). Alternately, the hole could be occupied by a new job with higher priority.

3 Related Work

Backfilling was introduced by Lifka [7] in the Extensible Argonne Scheduling sYstem (EASY). In a survey of parallel job scheduling, Feitelson et al [4] characterize variations in backfilling with three parameters, the number of reservations, the order of queue jobs, and the amount of lookahead into the queue. We add a fourth parameter, when the profile can be reordered.

Reservations have been used since the early days of parallel batch schedulers [2]. EASY [7] uses one reservation. At the other extreme, Conservative Backfilling [8] gives all jobs a reservation. Talby and Feitelson [14] and Srinivasan et al [13] suggest an adaptive number of reservations. The Maui Scheduler [5] has a parameterized number of reservations. Chiang et al [1] suggest that four is a good number of reservations.

EASY and Conservative Backfilling use First-Come-First-Serve (FCFS) order. The FCFS Scheduling Algorithm has been analyzed by Schwiegelshohn and Yahyapur [10]. Perkovic and Keleher [9] study Conservative Backfilling with random queue ordering both with and without sorting by length and random reordering as well. Reordering the backfill queue for EASY is proposed by Tsafrir et al [15].

Talby and Feitelson [14] combine three types of priorities in the order of queue jobs. The Maui Scheduler has even more components in its order of queue jobs. Chiang et al [1] propose generalizations of the Shortest Job First (SJF) scheduling algorithm to order queue jobs. They also use *fixed* and *dynamic* reservations. With dynamic reservations, job reservations and the ordering of job reservations can change with each new job arrival or if the priorities of waiting jobs change. With fixed reservations, job reservations can only move earlier in order, even if a job has no reservation or a job that has a later reservation attains a higher priority. Leung et al [6] study fixed and dynamic variations of Conservative Backfilling in the context of fairness.

All the above algorithms use no lookahead. Shmueli and Feitelson [11] use one reservation, various queue orderings, and lookahead into the queue. All of these algorithms reorder the profile when a job arrives or terminates early. All of our algorithms give every job a reservation, use various queue orderings based on the length or width of the jobs, and use no lookahead into the queue, a combination that is not used by any of the algorithms above. Additionally, some of our algorithms delay to varying degrees when the profile is reordered. Our PC algorithm reorders the profile when a job arrives or terminates early like all of the algorithms above. Our DC algorithm reorders the profile only when a job arrives or can run immediately.

4 Experimental Results

As described in the introduction, we evaluate our algorithms with an event-based simulator running traces from the Parallel Workloads Archive [3]. Figure 4 lists the traces used. These are all traces with estimated running times except

for LLNL-uBGL, which is omitted because its waiting time shows almost no variation for any of the algorithms we examined. Jobs in these traces without user estimates are given accurate estimates. (Simulations by Smith et al. [12] suggest that better estimates reduce average waiting time for Conservative scheduling. The effect of inaccurate estimates on EASY is the subject of many papers; Tsafrir and Feitelson [16] summarize and attempt to settle the issue.)

Name	Full file name	# jobs	% w/ estimates
CTC-SP2	CTC-SP2-1996-2.1-cln.swf	77,222	99.99
DAS2-fs0	DAS2-fs0-2003-1.swf	219,571	100
DAS2-fs1	DAS2-fs1-2003-1.swf	39,348	100
DAS2-fs2	DAS2-fs2-2003-1.swf	65,380	100
DAS2-fs3	DAS2-fs3-2003-1.swf	66,099	100
DAS2-fs4	DAS2-fs4-2003-1.swf	32,952	100
HPC2N	HPC2N-2002-1.1-cln.swf	202,876	100
KTH-SP2	KTH-SP2-1996-2.swf	28,489	100
LANL-CM5	LANL-CM5-1994-3.1-cln.swf	122,057	90.75
LLNL-Atlas	LLNL-Atlas-2006-1.1-cln.swf	38,143	84.85
LLNL-Thunder	LLNL-Thunder-2007-1.1-cln.swf	118,754	32.47
LPC-EGEE	LPC-EGEE-2004-1.2-cln.swf	220,679	100
SDSC-BLUE	SDSC-BLUE-2000-3.1-cln.swf	223,669	100
SDSC-DS	SDSC-DS-2004-1.swf	85,006	100
SDSC-SP2	SDSC-SP2-1998-3.1-cln.swf	54,041	99.94

Fig. 4. Traces used in simulations

The trace job counts given in Figure 4 differ from the values given in the Parallel Workloads Archive [3] because we ignored jobs that were partial executions (they were checkpointed and swapped out; status 2, 3, or 4) and jobs that were cancelled before starting (status 5 and running time ≤ 0). We also ignored 8 jobs in the SDSC-DS trace with running time -1 (unknown).

4.1 Increasing Responsiveness

Since user-perceived performance is the typical goal of scheduling, we first consider how our algorithms can improve average waiting time. For this metric, it is beneficial to run short jobs before long ones so we use Shortest Job First as our priority function. Figure 5 presents the results as a percent improvement over the average waiting time achieved by Conservative. We also include EASY for comparison since it backfills aggressively, benefiting short jobs since they backfill more easily. The exact results vary by traces, but our algorithms outperform Conservative on all traces except DAS2-fs3. In fact, they outperform EASY in the majority of cases. The most notable exception is the LLNL-Thunder trace, which has the lowest percent of jobs with user estimates (only 32%; see Figure 4). This may explain the relatively poor performance of our algorithms on that trace since jobs without estimates do not finish early, reducing the number of holes our algorithms can exploit. Of our algorithms, DC generally beats PC.

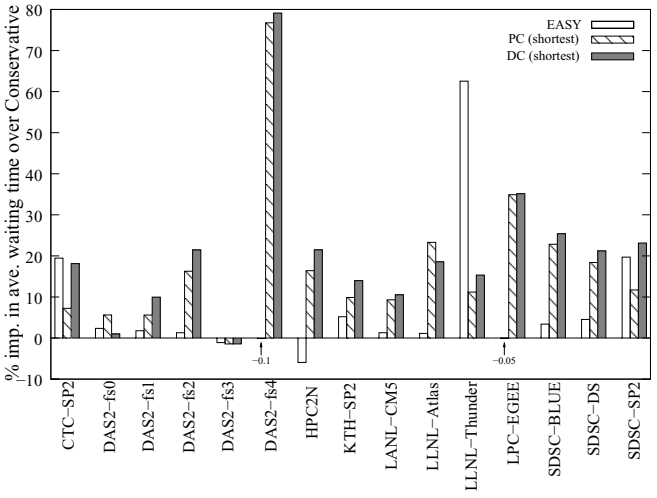

Fig. 5. Average waiting time relative to Conservative

Furthermore, our algorithms achieve these benefits without greatly delaying other jobs. To see this we looked at the average waiting time for the 5% of jobs with the greatest waiting time. See Figure 6 for the results, again presented as a percent improvement over Conservative's performance on the same measure. As in the overall average waiting time, our algorithms generally outperform Conservative, though there are more exceptions (DAS2-fs3, LANL-CM5, and SDSC-SP2). Comparing to EASY yields a similar picture as well, again with LLNL-Thunder as the outlier. The pattern remains when looking at the 1% of jobs with the greatest waiting time (see Figure 7); our algorithms give significantly better performance for the DAS2-fs2, DAS2-fs4, LLNL-Atlas, and LPC-EGEE traces, significantly worse performance for the LANL-CM5 and SDSC-SP2 traces, and comparable (within 10%) or mixed performance for the others.

We have shown that our algorithms significantly improve the average waiting time when using the shortest job first priority function. It is worth noting that they mostly outperform Conservative under this measure with other natural priority functions as well. Specifically, we considered the priority functions FIFO, Widest (most requested processors) Job First, Longest (in estimated time) Job First, Shortest Job First, and Narrowest (fewest requested processors) Job First for both of our algorithms. Out of 150 combinations of trace, algorithm, and priority function, only 49 (33%) of them were worse than Conservative. Most of the differences were small (generally $< 10\%$, many $< 2\%$), with a majority of the big improvements appearing in Figure 5 and the significantly negative values generally associated with the Longest Job First or Widest Job First priority functions. (The worst single value is -38% for DC with Longest Job First.)

Overall, DC with Shortest Job First seems to be a very good choice for increasing responsiveness. It gave better average waiting time than Conservative and EASY in eleven out of fifteen traces. It only had worst average waiting time than both Conservative and EASY in one trace and just EASY in three others.

4.2 Favoring Wide Jobs

To demonstrate the flexibility of our algorithms, we also look at a different scheduling goal: improving the performance of wide jobs. These are jobs that, because of a large computational or memory requirement, must run on many processors. From a capability perspective, wide jobs are the reason to build large systems since they cannot run otherwise.

To benefit these jobs, we run our algorithms with the Widest Job First priority function. We measure schedule quality with the average waiting time of the

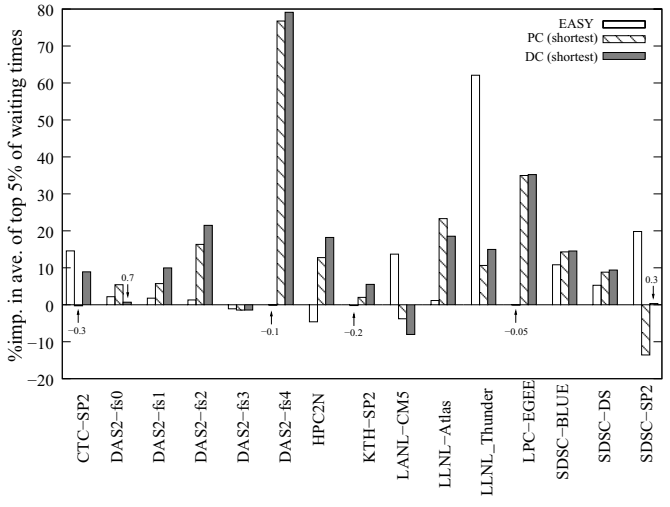

Fig. 6. Average of top 5% of waiting times relative to Conservative

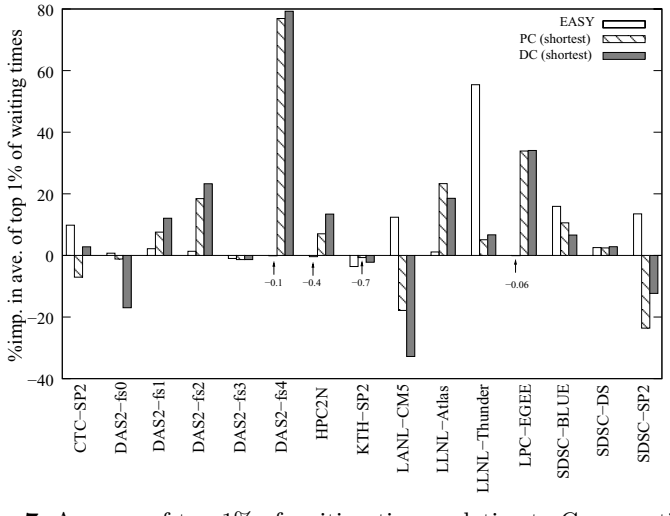

Fig. 7. Average of top 1% of waiting times relative to Conservative

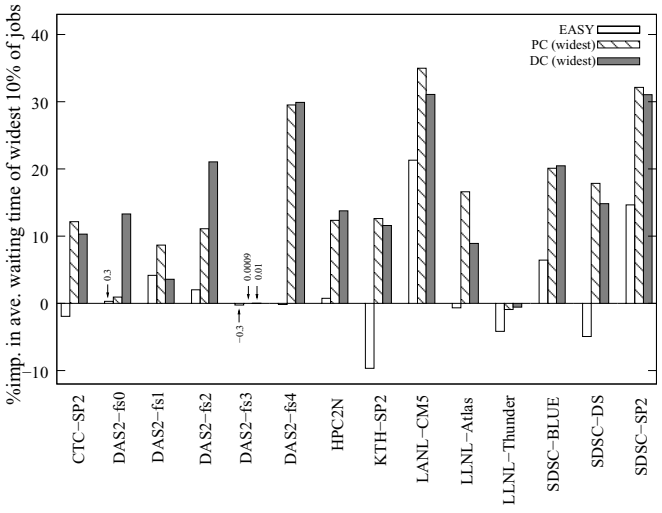

Fig. 8. Average waiting time of widest 10% of the jobs relative to Conservative

widest 10% of the jobs in each trace. Figure 8 shows the results as a percent improvement over Conservative. The LGC-EGEE trace is not included since each of its jobs requests a single processor. On the other traces, our algorithms outperform Conservative on all traces except LLNL-Thunder, the trace with relatively few user estimates. (The improvement on the DAS2-fs3 trace is admittedly negligible.) It is unclear which of them is preferable. Our algorithms also outperform EASY, which is not surprising since wide jobs have difficulty backfilling and thus benefit from the guaranteed start times given by our algorithms.

As when we tried to improve overall system responsiveness, we investigate the performance of non-favored jobs. Figure 9 plots average waiting time of all jobs, again relative to Conservative. The results are mixed, but not consistently bad and the negative values are of fairly small magnitude. Thus, it seems that our algorithms benefit wide jobs without greatly impairing overall performance.

4.3 Scheduler Running Time

As mentioned in Section 2.1, there is a question as to how long compression will take with our algorithms, particularly PC. We instrumented our simulations of Conservative and PC to measure the work required for compression. Specifically, we counted how many times the algorithms looked at an event (a job's planned start or end time) in the profile. In the worst case (Longest Job First priority on DAS2-fs0), PC examined nearly 580 times as many events as Conservative. This case is an extreme outlier; in only two other traces (HPC2N and DAS2-fs3) did PC examine more than 43 times as many events as Conservative (127 and 72 times, respectively). Even in the outlier case, however, the scheduler running

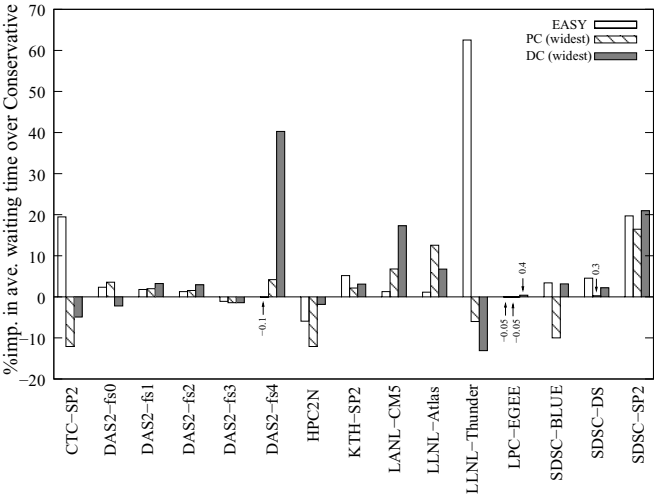

Fig. 9. Average waiting time relative to Conservative

time was not excessive; the total simulation time for that trace was less than 24 hours on a laptop, meaning the scheduler spent less than 0.4 seconds scheduling and rescheduling each job on average.

5 Discussion

We have presented a couple of algorithms that exploit flexibility in Conservative backfilling to improve various measures of performance while still retaining its ability to give jobs a guaranteed starting time as they arrive. We are impressed by the potential of these algorithms, but there is ample room for future research. More work is needed to understand why the algorithms perform better on some traces than others and to distinguish between the algorithms. It would also be interesting to consider other priority functions, including user-assigned job priorities, to further explore the flexibility in job selection. For the flexibility in timing, one of our algorithms closes holes as soon as possible and the other closes holes only when more jobs arrive or a job can run. We can further explore the flexibility in timing by closing holes only when a job can run.

Acknowledgments. A.M. Lindsay, M. Galloway-Carson, C.R. Johnson, and D.P. Bunde were partially supported by contracts 763836 and 899808 from Sandia National Laboratories. Sandia is a multipurpose laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under Contract No. DE-AC04-94AL85000. We also thank all those who contributed traces to the Parallel Workloads Archive.

References

1. 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)
2. Das Sharma, D., Pradhan, D.K.: Job scheduing in mesh multicomputers. In: Proc. Intern. Conf. on Parallel Processing Workshops, pp. 251–258 (1994)
3. Feitelson, D.: The parallel workloads archive, `http://www.cs.huji.ac.il/labs/parallel/workload/index.html`
4. Feitelson, D.G., Rudolph, L., Schwiegelshohn, U.: Parallel job scheduling — A status report. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2004. LNCS, vol. 3277, pp. 1–16. Springer, Heidelberg (2005)
5. Jackson, D., Snell, Q., Clement, M.: Core algorithms of the maui scheduler. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 2001. LNCS, vol. 2221, pp. 87–102. Springer, Heidelberg (2001)
6. Leung, V., Sabin, G., Sadayappan, P.: Parallel job scheduling policies to improve fairness - a case study. In: Proc. 6th Intern. Workshop on Scheduling and Resource Management for Parallel and Distributed Syst. (2010)
7. Lifka, D.: The ANL/IBM SP scheduling system. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1995 and JSSPP 1995. LNCS, vol. 949, pp. 295–303. Springer, Heidelberg (1995)
8. Mu'alem, A.W., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. IEEE Trans. on Parallel and Distributed Syst. 12(6), 529–543 (2001)
9. Perković, D., Keleher, P.J.: Randomization, speculation, and adaptation in batch schedulers. In: Proc. 2000 ACM/IEEE Conf. on Supercomputing (2000)
10. Schwiegelshohn, U., Yahyapour, R.: Analysis of first-come-first-serve parallel job scheduling. In: Proc. 9th ACM/SIAM Symp. on Discrete Algorithms, pp. 629–638 (1998)
11. Shmueli, E., Feitelson, D.G.: Backfilling with lookahead to optimize the performance of parallel job scheduling. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2003. LNCS, vol. 2862, pp. 228–251. Springer, Heidelberg (2003)
12. Smith, W., Taylor, V., Foster, I.: Using run-time predictions to estimate queue wait times and improve scheduler performance. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 1999, IPPS-WS 1999, and SPDP-WS 1999. LNCS, vol. 1659, pp. 202–219. Springer, Heidelberg (1999)
13. Srinivasan, S., Kettimuthu, R., Subramani, V., Sadayappan, P.: Selective reservation strategies for backfill job scheduling. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2002. LNCS, vol. 2537, pp. 55–71. Springer, Heidelberg (2002)
14. Talby, D., Feitelson, D.G.: Supporting priorities and improving utilization of the IBM SP scheduler using slack-based backfilling. In: Proc. 13th Intern. Parallel Processing Symp., pp. 513–517 (1999)
15. Tsafrir, D., Etsion, Y., Feitelson, D.G.: Backfilling using system-generated predictions rather than user runtime estimates. IEEE Trans. on Parallel and Distributed Systems 18(6), 789–803 (2007)
16. Tsafrir, D., Feitelson, D.G.: The dynamics of backfilling: Solving the mystery of why increased inaccuracy may help. In: Proc. IEEE Intern. Symp. on Workload Characterization, pp. 131–141 (2006)