Variations of Conservative to improve fairness

Avinab Rajbhandary[1], David P. Bunde[1], and Vitus J. Leung[2]

[1] Knox College, Galesburg IL, USA,
arajbhan@knox.edu, dbunde@knox.edu
[2] Sandia National Laboratories, Albuquerque NM, USA,
vjleung@sandia.gov

Abstract. We apply recent variations of Conservative backfilling in an effort to improve scheduler fairness. These variations modify the compression operation while preserving the key property that jobs never move later in the profile. We assess the variations using two known measures of scheduler fairness. Each of the variations turns out to be better than Conservative according to one of the metrics.

1 Introduction

This paper looks at scheduling to achieve fairness. From the very beginning of job scheduling research, some notions of fairness have been sought, such as measures to prevent job starvation. Many times, however, fairness has been a secondary consideration behind various performance-oriented metrics such as utilization or response time. Concern for these metrics has led to a variety of different backfilling strategies. We turn this around and look at the use of backfilling to improve measures of fairness.

Our algorithms are variations of the well-known Conservative scheduling algorithm [10]. The specific variations, PC and DC, were developed to exploit some apparent flexibility in the compression operations that Conservative performs when a job finishes before its estimated completion time [8]. These variations perform compression by rescheduling jobs according to a user-specified priority function. By supplying first-come first-served (FCFS) as the priority function, we create two scheduling algorithms that attempt to use backfilling to favor early-arriving jobs, matching an intuitive notion of FCFS as a fair scheduling strategy.

To evaluate these algorithms, we use two previously-formulated notions of job-level fairness [14]. The first of these is that it is unfair for jobs to run out of arrival order, directly incorporating the idea of FCFS. The other notion is that each job deserves an equal share of the system resources. Each of these notions has been formalized, the first in metrics that compare job starting times with their "fair starting time" and the second as metrics that compare the resources jobs receive relative to their "fair share".

We evaluate the algorithms using trace-based simulations run using traces from the Parallel Workloads Archive [2]. For each job, we take its arrival time, number of processors used, actual processing time, and estimated processing time. The simulated runs are evaluated using the fairness metrics. We find that the two fairness metrics are significantly different and that each of them is favored by one of the scheduling algorithms.

The rest of this paper is organized as follows. We describe the algorithms and fairness metrics in Sections 2 and 3. Then we describe our simulation results in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2 Algorithms

In this paper, we examine two new scheduling algorithms, both of which are based on Conservative Backfilling [10]. Conservative maintains a profile giving a tentative schedule for all queued jobs. Each job's starting time in this profile serves as a reservation, a time by which the job is guaranteed to start. Newly arriving jobs are placed into this profile at the earliest possible time that does not interfere with any other job. When a job finishes early (i.e. in less than its estimated processing time), this profile must be adjusted. If the jobs are simply rescheduled from scratch in the order they arrived, the resulting profile may cause a job to violate its reservation; the reservation may have required backfilling which is no longer possible in the new profile. Instead, Conservative initiates *compression*, in which each queued job is removed from the profile and rescheduled to the earliest possible time that does not interfere with any other job (including those that arrived after it). Since each job can fit back into its current spot, no job is ever moved to a later time, meaning that Conservative can always give users an upper bound on the starting time of their job.

The order in which jobs are rescheduled during compression is not entirely specified. One effective choice is to use the order that jobs appear in the old profile. This order is attractive because it allows jobs to be rescheduled as they are encountered in a traversal of the profile. Using the as-currently-scheduled order also allows the new profile to be built from scratch since later jobs cannot interfere with a job's ability to reschedule to an earlier time. We use this compression order for the implementation of Conservative that we use as a baseline. The simulator whose results are reported in the original paper to use the name "Conservative" [10] also used the as-currently-scheduled order for compression [3]. Intuition suggests that this compression order tends to preserve the order of jobs in the profile. Since the profile is built as jobs arrive, the initial order has a first-come first-served (FCFS) tendency, making Conservative a logical baseline schedule with respect to fairness.

The idea of performing compression by rescheduling jobs in the order that they originally arrived to enhance fairness was actually suggested in the original paper [10]. Using this order would mean that the rescheduling operations must place jobs into the full profile rather than building a new one from scratch. It turns out that rescheduling in this order can also leave unnecessary gaps in the schedule. Figure 1 shows an instance where this occurs. Part (a) shows the profile of 4 jobs which arrived in ascending numeric order; job 4 backfilled to reach the position shown. The displayed width of each job is its estimated running time.

When job 2 finishes early, job 3 moves slightly earlier (shown in (b)) and then job 4 moves earlier (shown in (c)), leaving a large gap. Job 3 could move earlier, but it has already been rescheduled.

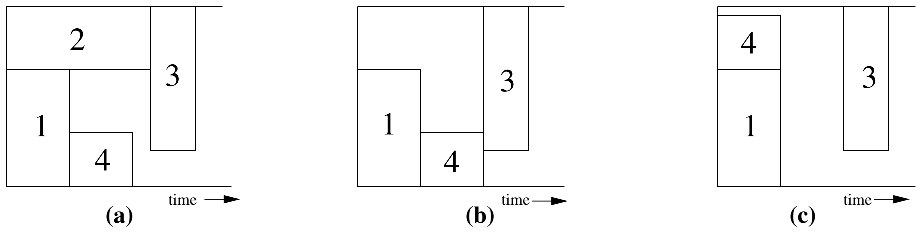

Fig. 1. Instance where compressing in order of job arrival leads to a gap in the profile. Jobs arrive in numerical order. (a) Original profile. (b) Profile after 2 finishes early and 3 is rescheduled. (c) Ending profile after 4 is rescheduled, leaving a gap.

Lindsay et al. [8] addressed the problem of unnecessary gaps based on an example similar to Figure 1, but with a different compression order. They proposed two variations of Conservative that are parameterized by a priority function specifying in which compression occurs. Prioritized Compression (PC) attempts to reschedule the jobs in the order specified by this priority function rather than the order they occur in the profile. To resolve the issue of unnecessary gaps, it returns to the highest-priority job whenever a job is successfully rescheduled, potentially rescheduling a job multiple times during a single compression operation. The second variation, conservative with Delayed prioritized Compression (DC), further modifies compression by only rescheduling jobs that can start immediately or if new lower-priority jobs arrive. The goal in delaying rescheduling operations is to allow holes opened by early job completions to grow as much as possible before backfilling. There are some priority functions that are aided by this growth. Both PC and DC preserve the property of Conservative that jobs never start later than the guaranteed start time given when they arrive.

In an effort to improve fairness, we evaluate PC and DC using the FCFS priority function, which orders jobs based on their arrival time. As noted above, Conservative already has a FCFS tendency since it prioritizes jobs in the order of the profile, which is constructed as jobs arrive. Reordering the compression operations by explicitly using the FCFS priority function potentially allows the scheduler to move the profile toward FCFS order even when jobs are initially placed out of this order.

In addition to these algorithms, we also compare our results against EASY [7], a more aggressive backfilling algorithm which will backfill a job unless doing so delays the current first job in the queue. This algorithm is used in practice to promote high system utilization and the restriction that the first job in the queue cannot be harmed by backfilling is sufficient to guarantee that no job starves forever [10], but it has been shown to discriminate against jobs requiring many processors since these jobs have difficulty backfilling (e.g. [17]). Thus, EASY represents a choice that could be used on systems not overly concerned with fairness.

3 Definitions of fairness

To quantify fairness, we look at two types of metrics, following a classification by Sabin and Sadayappan [14]. The first of these is based on the notion that it is unfair for jobs to "cut" in line and run ahead of jobs that arrived earlier. The second is based on the notion that each job in the queue deserves an equal share of the system resources.

3.1 Fair Start Time

When people are waiting, cutting in line (aka "queue jumping") is viewed as a violation of social justice, with the seriousness dependent on how long one has waited for the resource [9]. If the goal is to avoid cutting, then the gold standard for fairness would be FCFS without backfilling, since it never starts a job before all earlier-arriving jobs. There are two issues with this characterization. The first is performance-related; scheduling without backfilling will reduce system utilization and make all users unhappy. The second is that it assumes that the jobs suffer envy rather than just wanting to minimize their own start time; one job receiving prompt service because it can jump ahead in the queue is not unfair unless other jobs are disadvantaged. Sabin and Sadayappan [14] use the analogy of service in a restaurant to explain this: Restaurant customers typically expect to be served in FCFS order, but do not normally object if someone who just ordered a drink receives it immediately because such an order is quick and does not cause a delay in anyone else's service. In parallel job scheduling, the analogous phenomenon is *benign backfilling*, where jobs arriving later can backfill without delaying the start time of other jobs. Thus, Conservative would be fair under this definition if job lengths were accurately estimated. (The issue of accurate estimates is important because an apparently benign backfill can delay other jobs if their position in the profile is based on an inaccurate estimate.)

Based on the idea that the key to unfairness is delaying a job past its "rightful" start time, Sabin and Sadayappan [14] defined a job's *strict Fair Start Time* (strict FST) as the starting time a job would get if no jobs arrived after it.

One issue with strict FST is that inaccurate estimates can create sets of strict FSTs that are not all together feasible. For example, consider the profile illustrated in Figure 2. Figure 2(a) shows the Conservative backfilling schedule with two jobs. The shaded portion of job J_1 shows the actual duration of this job, whose length is significantly overestimated. Since job J_2 can start as soon as job J_1 completes, its fair start time is at the end of the shaded region. Now job J_3 arrives and the profile becomes as shown in Figure 2(b). Job J_3 has backfilled since the reservation for job J_2 is based on the estimated processing time of job J_1 rather than its actual processing time. Starting each job at the given fair start times would require running jobs J_2 and J_3 simultaneously, however, and is therefore infeasible.

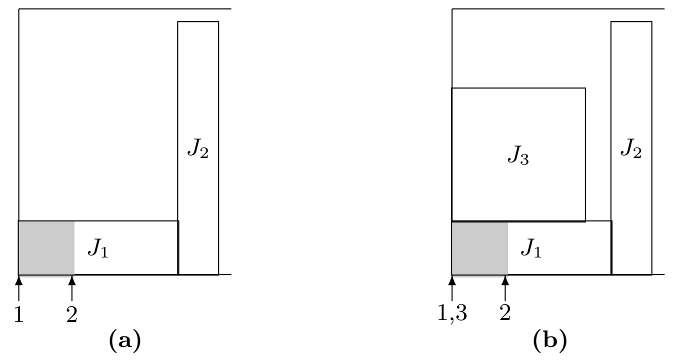

Fig. 2. Instance where strict fair start times are infeasible. Anticipated schedule (a) before and (b) after arrival of job J_3. The shaded region and the block of job J_1 shows its actual length and estimated time respectively. Labels below the figures indicate the strict fair start time of each job.

The recognition that backfilling decisions can make the strict FSTs infeasible justifies a variation. Sabin and Sadayappan [14] define the *relaxed Fair Start Time* (relaxed FST) of a job as its starting time if no jobs arrive after it, but it is not allowed to backfill. In particular, it must start no earlier than the last of the other jobs in the queue when it arrives. This yields generally larger FST values and avoids sets of infeasible fair start times.

To calculate the amount by which a specific job was unfairly treated, we consider the difference between one of the FST values and its actual starting time. To prevent algorithms from benefiting by preferentially treating jobs, we take the maximum of this difference and zero. Averaging this over all jobs gives either the *average strict unfairness* or the *average relaxed unfairness*, depending on which FST value is used. These metrics, proposed by Sabin and Sadayappan [14], are fairness analogs of average waiting time. Other metrics based on FST are discussed in Section 5.

As an aside, we note that the FST values require considerable effort to compute. To do so, our simulator copies the current state whenever a job arrives and runs that copy until the job starts, either normally for strict FST or after the last other queued job starts for relaxed FST. This is inconvenient for our experiments, but we note that the calculation is only required for reporting the fairness metrics. As none of the algorithms use the metric values in their operation, this step would not be required for a production scheduler.

3.2 Resource Equality

The other measure of fairness that we consider is based on the idea of *resource equality*, a notion developed for serial jobs by Raz et al. [12] and extended to parallel jobs by Sabin and Sadayappan [14]. The basic idea is that each *active* job, i.e. one that has arrived but not yet been completed, deserves an equal share of system resources. A job's perception of unfairness is then the amount less than this that it receives.

There are two subtleties in dividing system resources equally. First of all, no job's fair share of the processors is allowed to exceed the number that it wants to use. For example, in a 30 processor system, if a 10-processor job and a 20-processor job are active, the smaller job is not considered to be unfairly treated for only getting 10 processors rather than the $30/2 = 15$ that would be an equal share. Secondly, fair shares are based on the number of processors in use rather than the total system size. For example, if there are 3 active jobs on a 100-processor system but only use 90 processors are being used, each job's fair share is $90/3 = 30$ rather than $100/3 = 33.\overline{3}$. This prevents fragmentation from being the cause of unfairness and helps make the scheduler goals of fairness and utilization orthogonal.

We use two ways to calculate the fair share of job J_i. Both are defined in terms of its arrival time a_i, completion time c_i, and number of processors p_i. The first one is its *unweighted fair share*:

$$\int_{a_i}^{c_i} \min\left\{\frac{\text{util}(t)}{\text{active}(t)}, p_i\right\} dt \tag{1}$$

where util(t) and active(t) are the numbers of processors in use and the number of active jobs respectively. For the *weighted fair share*, we replace active(t) by the proportion of all requested processors that are from job J_i:

$$\int_{a_i}^{c_i} \min\left\{\frac{p_i}{\sum_{J_j \text{ is active}} p_j} \cdot \text{util}(t), p_i\right\} dt \tag{2}$$

This modification increases the fair share allocated to larger jobs, with the idea that they should get a larger portion of the system.

From either of the measures of a job's fair share given in equations 1 and 2, we can calculate the corresponding measure of fairness by subtracting the amount of resources it actually received, which is the product of its processing time and the number of processors used. For a job's *unweighted unfairness*, we subtract the resources received from equation 1. Similarly, for its *weighted unfairness*, we subtract from equation 2. We report the average of these values over all jobs.

We note that the fairness metrics based on fair share are easier to compute than those based on FST. We compute them as a post-processing step, though it would be possible to keep a running total of each job's fair share as it ran. The only tricky part is that its rate of increase changes each time the system's utilization or set of active jobs changes. Thus, at each job arrival or completion, we increase the fair share values to reflect the contribution since the last arrival or completion event.

4 Results

We evaluated the algorithms with these fairness metrics using an event-based simulator run with traces from the Parallel Workloads Archive [2]. Figure 3 lists the traces used. We largely follow the lead of [8] in selecting traces except that we add the ANL-Intrepid trace. We also removed DAS2-fs0 and HPC2N because the fair start time calculations were taking inordinately long; this deserves closer examination, but the culprit seems to be the queue length, which causes the simulations from each job arrival to complete very slowly.

Even with these omissions, our study uses most of the traces with estimated running times. The exceptions other than the above are LLNL-uBGL (which showed almost no variation between the Conservative, PC, and DC algorithms [8]), Sandia Ross (whose entry in the archive warns about its use because the machine size was changed during the period recorded in the trace), and RICC (excluded for time reasons and which we plan to study subsequently). Jobs in the traces without user estimates are given accurate estimates. (Simulations by Smith et al. [16] 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 [19] summarize and attempt to settle the issue.)

Name	Full file name	# jobs
ANL-Intrepid	ANL-Intrepid-2009-1.swf	68,936
CTC-SP2	CTC-SP2-1996-2.1-cln.swf	77,222
DAS2-fs1	DAS2-fs1-2003-1.swf	39,348
DAS2-fs2	DAS2-fs2-2003-1.swf	65,380
DAS2-fs3	DAS2-fs3-2003-1.swf	66,099
DAS2-fs4	DAS2-fs4-2003-1.swf	32,952
KTH-SP2	KTH-SP2-1996-2.swf	28,489
LANL-CM5	LANL-CM5-1994-3.1-cln.swf	122,057
LLNL-Atlas	LLNL-Atlas-2006-1.1-cln.swf	38,143
LLNL-Thunder	LLNL-Thunder-2007-1.1-cln.swf	118,754
LPC-EGEE	LPC-EGEE-2004-1.2-cln.swf	220,679
SDSC-BLUE	SDSC-BLUE-2000-3.1-cln.swf	223,669
SDSC-DS	SDSC-DS-2004-1.swf	85,006
SDSC-SP2	SDSC-SP2-1998-3.1-cln.swf	54,041

Fig. 3. Traces used in simulations

The trace job counts given in Figure 3 differ from the values given in the Parallel Workloads Archive [2] 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 Fair start time: DC

The first thing that jumped out of our results was that DC does very badly for FST-based fairness. Figure 4 shows the percent improvement of DC over Conservative for average strict and relaxed unfairness. (Calculating percent improvement as (Conservative - Other)/Conservative.) The values are nearly always negative, meaning that DC performed substantially worse than Conservative.

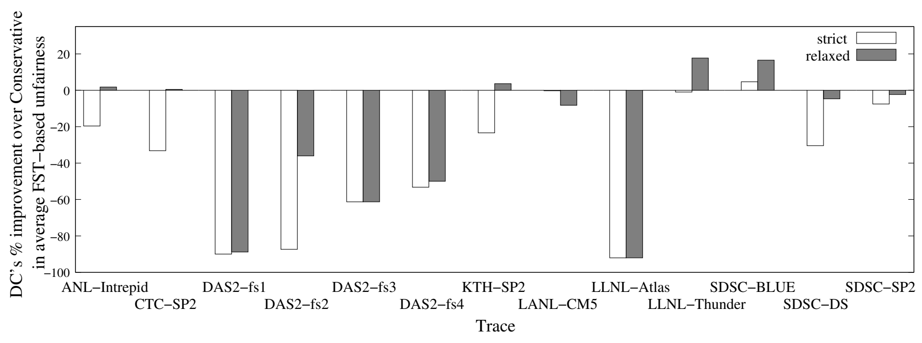

Fig. 4. Improvement in average strict and relaxed unfairness of DC over Conservative. Not shown is LPC-EGEE for which all algorithms except DC produce average unfairness of 0; DC gives unfairness ~ 0.102 for both ($-\infty$ improvement).

The delays before compression operations seem to make DC particularly prone to assigning jobs very low strict FSTs, as in Figure 2. When this happens, the algorithm is made to seem particularly unfair since the jobs cannot meet the unrealistic fair start times. Consider the following set of jobs:

Job	Arrival time	# processors	Processing time	User estimate
J_1	0	90	100	200
J_2	1	45	100	200
J_3	2	40	95	200
J_4	3	90	100	200
J_5	4	45	100	200

Shortly after all these jobs have arrived, the profile of both Conservative and DC is as shown in Figure 5. They also generate identical strict FSTs for the first four jobs, as shown in the figure. (We use Conservative here for concreteness, but Conservative, EASY, and PC all generate the same schedule and fair start times on this instance.) The FST of job J_4 comes from starting jobs J_2 and J_3 immediately after the (early) completion of job J_1 and then starting J_4 immediately after J_2 completes. With Conservative, the FST for job J_5 is then determined by when it can run after job J_4. For DC, however, job J_4 doesn't compress when job J_1 finishes early and jobs J_2 and J_3 start. (Recall that DC only reschedules jobs if they can start immediately or to prevent newly-arrived lower-priority jobs from backfilling.) Thus, job J_5 is able to backfill as soon as job J_3 finishes; job J_4 cannot start at this time because job J_2 is still running. The result is a delay for J_4, making DC significantly unfair.

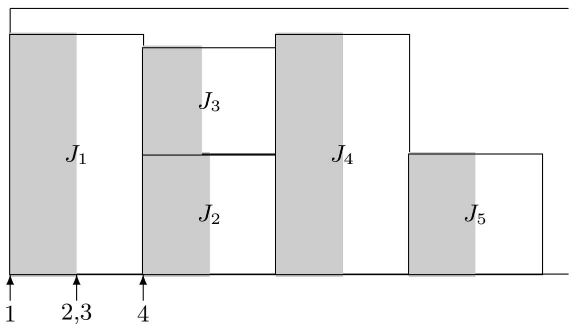

Fig. 5. Profile after all jobs arrive in instance showing DC's potential for unfairness. The shaded region and the block of each job show its actual length and estimated time respectively . Labels below the figures indicate the strict fair start time of each job.

Note that if job J_5's running time were accurately estimated (and the instance is otherwise unchanged), both Conservative and DC would backfill it. In this case, both would assign identical strict FSTs and they would register as equally unfair. As previously noted, however, job lengths are typically overestimated. This is where DC's hesitation to compress comes in; it doesn't move job J_4 earlier when job J_1 finishes early, allowing it to backfill job J_5 even when its length is overestimated. This tendency to backfill was a design goal of DC, but it seems to be a liability according to the FST metrics even when the FCFS priority function is used.

Note that the example described above only directly explains why DC is so unfair when using the strict FST measure; the instance shown relies on backfilling job J_5. We have a larger example showing that DC can also assign low values to relaxed FST.

4.2 Fair start time: PC

PC does much better according to the FST-based fairness measures. Figure 6 shows the percent improvement of PC over Conservative for the FST-based measures. (EASY is also included for comparison.) On the strict measure, PC does as well as Conservative on the DAS2-fs3 and LPC-EGEE traces, but beats it on all the others (admittedly by only 0.16% on LLNL-Atlas). On the relaxed measure, the performance is mostly the same: matching Conservative on DAS2-fs3 and LPC-EGEE, beating it by a small amount on LLNL-Atlas, and winning handily on most of the others. The exception is LLNL-Thunder, where it loses to Conservative by nearly 39%. We are not sure of the cause of this poor performance, but note that this trace gave PC and DC difficulty in previous work [8] as well. LLNL-Thunder is also the trace in which the smallest fraction of the jobs have user estimates supplied in the trace (32.47%). Since our simulator assigns accurate estimates to jobs without them, this means that only about a third of the jobs in this trace finish early, greatly reducing the opportunities PC has to use its special backfilling operation.

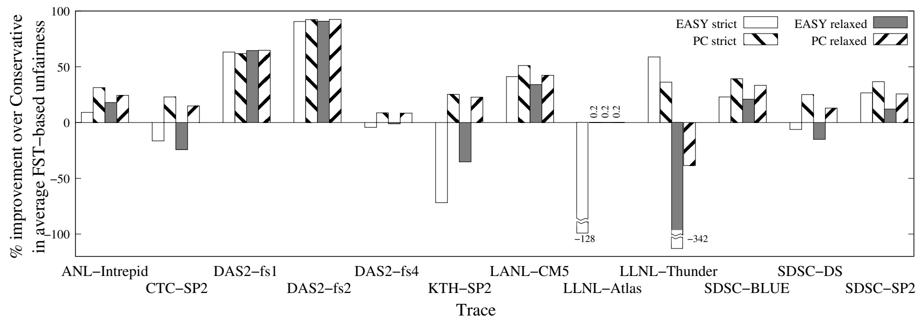

Fig. 6. Improvement in average strict and relaxed unfairness of EASY and PC over Conservative. Not shown are LPC-EGEE (all algorithms except DC produce unfairness of 0) and DAS2-fs3 for which PC gives no improvement and EASY produces "improvements" of $-3,010\%$ and $-2,784\%$ for strict and relaxed unfairness respectively.

Although it is not targeted at fairness, Figure 6 also reveals that EASY improves upon Conservative for many of the traces. It is less consistent than PC, however, and performs substantially worse on some of the traces. For strict fairness, PC does at least as well on all but two of the traces, DAS2-fs1 and LLNL-Thunder, and its performance on DAS2-fs1 is comparable (a 61.9% improvement vs 63.2% for EASY). For relaxed, PC does at least as well as EASY on all but one of the traces; the exception this time is LLNL-Atlas, on which it gives a 0.16% improvement vs 0.22% for EASY.

4.3 Fair share

While PC clearly outperforms the other algorithms for the FST-based fairness metrics, the situation with fair share metrics is much less clear. Figure 7 shows the percent improvement over Conservative for the other algorithms on the unweighted and weighted measures respectively.

For unweighted fairness, DC seems to be the best algorithm, beating Conservative on all but three of the traces (DAS2-fs1, DAS2-fs3, and LLNL-Thunder) and outperforming all the other algorithms on 11 of the 14 traces. For the weighted measure, both PC and DC do fairly well, each defeating the other algorithms on 6 of the traces. With both measures, the improvements are generally by less than 5%, however. The fairness metrics based on fair share seem to be much harder to improve.

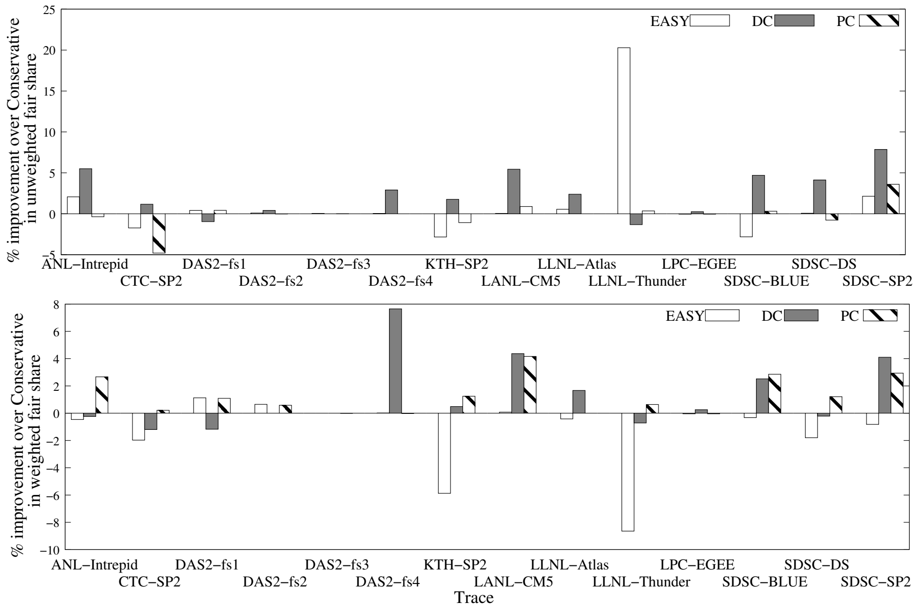

Fig. 7. Improvement in unweighted (top) and weighted (bottom) unfairness (fair share approach) over Conservative.

4.4 Response time

We conclude our presentation of the results by showing that our algorithms are not achieving fairness at great cost in terms of traditional performance-oriented measures. Figure 8 shows percentage improvements over Conservative on average waiting time. DC beats Conservative on all but one of the traces (DAS2-fs3), achieving double digit improvements on five of them. PC is worse than Conservative on 9 of the traces, but always by less than 3.5% and by less than 2% on all but two of them.

5 Related work

There are several types of previous work related to our study.

PC and DC. Lindsay et al. [8] originally proposed PC and DC to improve either overall system responsiveness or the treatment of wide jobs (i.e. those using large numbers of processors). With the shortest job first priority function, PC and DC reduced average waiting time and average bounded slowdown relative to Conservative and EASY on most traces. Notably, this is achieved without greatly penalizing particular jobs since PC and DC still achieved lower average waiting time than Conservative and EASY when the average was taken over just the top 5% or top 1% of the waiting times. With the widest job first priority

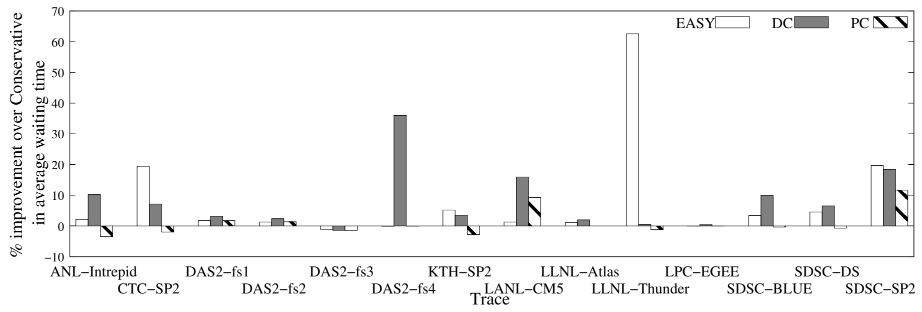

Fig. 8. Improvement in average waiting time over Conservative.

function, PC and DC reduced the average waiting time of wide jobs by 10–35% while still also improving the overall average waiting time.

Prioritized backfilling. Many other scheduling schemes have been proposed that use a priority function in connection with backfilling. A typical approach is to order the jobs by priority and then backfill to improve utilization. Jackson et al. [4] describe a version of this used in the Maui scheduler that provides a reservation to the highest-priority queued job (essentially a prioritized version of EASY). Perković and Keleher [11] add elements of randomization and speculation to this approach. These approaches differ from ours because, like Conservative, PC and DC provide guaranteed starting times to all jobs from the time they are submitted.

Fair start time. Srinivasan et al. [17] give a precursor to FST based specifically on Conservative. They define FST as the earliest possible start time a job would have received under FCFS conservative if the scheduling strategy were suddenly changed to strict FCFS without backfill at the instant the job arrived. This version of FST has the advantage of being independent of the scheduler being considered since the FST is always computed using Conservative. The disadvantage is that it partially combines the effects of scheduler throughput and fairness; a scheduler that achieves shorter waiting times will tend to appear more fair since those waiting times also impact where each job finishes relative to its FST. This phenomenon may conflict with user perceptions since increased backfilling could result in both more "cutting" and greater fairness. Sabin and Sadayappan [14] resolved this paradox by generalizing the FST calculation to the definitions we use.

Leung et al. [6] introduce a "hybrid" FST that considers the allocation of specific processors. This is calculated using per-node estimated completion times. Under this scheme, a job's FST is the earliest time that enough nodes will be free; the estimated completion time of these nodes is then updated. The implied schedule is more restrictive than Conservative as holes cannot be used, but it is less restrictive than strict FCFS.

As mentioned above, our average unfairness metric is an analog of average waiting time since it is the difference from when the job "should" start when it does start. Other metrics can be derived from FSTs as well. Sabin et al. [13] use *fair turnaround time*, which adds job running times and is thus an analog to flow time. Sabin and Sadayappan [14] introduce *fair slowdown*, which is the ratio of this to job running time, making it an analog of slowdown or stretch.

Fair share. The idea of the fair share metric comes out of an effort to quantify fairness in queueing systems; see Avi-Itzhak et al. [1] for a survey. Raz et al. [12] extended this to multi-server and multi-queue systems (but with serial jobs). They used the Resource Allocation Queueing Fairness Measure (RAQFM), which uses the philosophy that all the active users in system deserve an equal share of system resources. This includes the refinement that only the actively used resources should be shared, which becomes our use of only the active processors rather than the total number. Sabin and Sadayappan [14] extended this to the fair share fairness metrics we use, though they did not actually compute the unweighted measure.

Other approaches. A variety of other scheduling mechanisms have been proposed to achieve various measures of fairness. Schwiegelshohn and Yahyapour [15] introduce a preemptive FCFS (PFCFS) algorithm where a job in the schedule may be preempted by a later arriving job. To prevent starvation, they assign each job a weight equal to its resource consumption and limit the amount of time a job can be delayed by later arriving jobs. Fairness is then measured using a new metric λ *fairness*; a scheduling strategy is λ-fair if no job can have its flow time increased more than a factor of λ by later arriving jobs.

Sabin et al. [13] advocate "dynamic reservations", in which the entire schedule is recomputed from scratch. This lessens the damage caused when later jobs backfill ahead of earlier ones (since these decisions can be revisited until the jobs actually start), but it eliminates the scheduler's ability to give jobs guaranteed starting times when they arrive. Srinivasan et al. [17] propose a scheduler that adds reservations to ameliorate unfairness without rebuilding the schedule. Their strategy does not give reservations to jobs initially, but does once their estimated slowdown (waiting time plus estimated running time over estimated running time) reaches a threshold value. Leung et al. [6] compare the effect of these strategies with several other measures designed to encourage fairness and/or prevent starvation: job runtime limits, job priorities based on the submitter's recent usage, and differential treatment for jobs of heavy users.

Rather than consider fairness on a per-job basis, Klusáček and Rudová [5] consider fairness to each user by considering a measure of the average waiting times for each user's jobs. This is combined with traditional performance-oriented metrics into a multi-objective optimization problem, to which tabu search is applied.

Stoica et al. [18] introduced a scheduling algorithm that uses a market paradigm to achieve user-level fairness and and also provides users with some control over the relative performance of their jobs. In their system, each user has a savings account in which they receive virtual money at a constant rate. To run a job, users create an expense account for it and transfer money to the job. Each job uses its funds to buy the system resources it requires at market rates. The allocation of system resources to each user depends upon the rate at which they receive money and users can control their jobs' relative performance by adjusting the rates at which they are funded.

6 Discussion

The original idea behind PC and DC was to exploit flexibility in the compression operation of Conservative. It was previously shown that this flexibility could be utilized to improve average system response time or to improve the treatment of large jobs. In this study, we have examined whether the same ideas could be used to improve system fairness. We have shown that PC does so for the fairness metrics based on fair start times while DC seems to be better for those based on fair share. Although it would be preferable for a single algorithm to dominate by both metrics, our split result highlights that the different metrics are really measuring different notions of desired behavior. "Fairness" is a somewhat slippery concept, but our results do show that the general approach of modifying Conservative's compression operation has potential to improve it. Notably, both of the algorithms also retain the worst-case predictability of Conservative in that both are able to give arriving jobs a guaranteed start time.

Going forward, we are interested in continuing to explore the fair share metrics to understand how they can be optimized. It is also desirable to develop a modification of DC that avoids its tragic performance on FST-based metrics since it does so well otherwise (in both this study and previous work [8]).

Acknowledgments. We thank the anonymous referees for their helpful comments. A. Rajbhandary and D.P. Bunde were partially supported by contract 899808 from Sandia National Laboratories. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. We also thank all those who contributed traces to the Parallel Workloads Archive.

References

1. B. Avi-Itzhak, H. Levy, and D. Raz. Quantifying fairness in queuing systems: Principles, approaches, and applicability. *Probability in the Engineering and Informational Sciences*, 22(4):495–517, 2008.
2. D. Feitelson. The parallel workloads archive. `http://www.cs.huji.ac.il/labs/parallel/workload/index.html`.
3. D. Feitelson. Personal communication, 2013.

4. D. Jackson, Q. Snell, and M. Clement. Core algorithms of the maui scheduler. In *Proc. 7th Workshop Job Scheduling Strategies for Parallel Processing*, number 2221 in LNCS, pages 87–102, 2001.
5. D. Klusáček and H. Rudová. Performance and fairness for users in parallel job scheduling. In *Proc. 16th Workshop Job Scheduling Strategies for Parallel Processing*, number 7698 in LNCS, pages 235–252, 2012.
6. V.J. Leung, G. Sabin, and P. Sadayappan. Parallel job scheduling policies to improve fairness: A case study. In *Proc. 6th Intern. Workshop Scheduling and Resource Management for Parallel and Distributed Systems*, 2010.
7. D. Lifka. The ANL/IBM SP scheduling system. In *Proc. 1st Workshop Job Scheduling Strategies for Parallel Processing*, number 949 in LNCS, pages 295–303, 1995.
8. A.M. Lindsay, M. Galloway-Carson, C.R. Johnson, D.P. Bunde, and V.J. Leung. Backfilling with guarantees made as jobs arrive. *Concurrency and Computation: Practice and Experience*, 25(4):513–523, 2013.
9. L. Mann. Queue culture: The waiting line as a social system. In *The American Journal of Sociology*, volume 75, pages 340–354, 1969.
10. A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Trans. Parallel and Distributed Syst.*, 12(6):529–543, 2001.
11. D. Perković and P.J. Keleher. Randomization, speculation, and adaptation in batch schedulers. In *Proc. 2000 ACM/IEEE Conf. on Supercomputing*, 2000.
12. D. Raz, B. Avi-Itzhak, and H. Levy. Fairness considerations in multi-server and multi-queue systems. In *Proc. 1st Intern. Conf. Perf. Eval. Methodolgies and Tools*, 2006.
13. G. Sabin, G. Kochhar, and G. Sadayappan. Job fairness in non-preeemptive job scheduling. In *Proc. Intern. Conf. Parallel Processing (ICPP)*, 2004.
14. G. Sabin and P. Sadayappan. Unfairness metrics for space-sharing parallel job schedulers. In *Proc. 11th Workshop Job Scheduling Strategies for Parallel Processing*, number 3834 in LNCS, pages 238–256, 2005.
15. U. Schwiegelshohn and R. Yahyapour. Fairness in parallel job scheduling. *J. Scheduling*, pages 297–320, 2000.
16. W. Smith, V. Taylor, and I. Foster. Using run-time predictions to estimate queue wait times and improve scheduler performance. In *Proc. 5th Workshop Job Scheduling Strategies for Parallel Processing*, number 1659 in LNCS, pages 202–219, 1999.
17. S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. Selective reservation strategies for backfill job scheduling. In *Proc. 8th Workshop Job Scheduling Strategies for Parallel Processing*, number 2537 in LNCS, pages 55–71, 2002.
18. I. Stoica, H. Abdel-Wahab, and A. Pothen. A microeconomic scheduler for parallel computers. In *Proc. 1st Workshop Job Scheduling Strategies for Parallel Processing*, number 949 in LNCS, pages 200–218, 1995.
19. D. Tsafrir and D.G. Feitelson. The dynamics of backfilling: Solving the mystery of why increased inaccuracy may help. In *Proc. IEEE Intern. Symp. on Workload Characterization*, pages 131–141, 2006.