

Scheduling restartable jobs with short test runs

Ojaswirajanya Thebe¹, David P. Bunde¹, and Vitus J. Leung²

¹ Knox College, {othebe,dbunde}@knox.edu

² Sandia National Laboratories, vjleung@sandia.gov

Abstract. In this paper, we examine the concept of giving every job a trial run before committing it to run until completion. Trial runs allow immediate job failures to be detected shortly after job submission and benefit short jobs by letting them run and finish early. This occurs without incurring a significant penalty on longer jobs, whose average and maximum waiting time are actually improved in some cases. The strategy does not require preemption and instead uses the ability to kill and restart a job from the beginning, which it does at most once for each job. While others have proposed similar strategies, our algorithm is distinguished by its determination to give each job a fixed-length trial run as soon as possible. Our study is also more focused, including a detailed description of the algorithm and an examination of the effect of varying the length of a trial run.

1 Introduction

It is widely known that user estimates of job runtimes are highly inaccurate (eg. [8],[10]). Typically the worst overestimates are explained by pointing to programs that fail early in their execution. For example, Perković and Keleher [12] say “The presence of large runtime overestimations indicates the presence of applications still in development, and therefore, have high probability to die prematurely either because of bugs or because they run in a new environment”. At the same time, job queues on large machines can be long, potentially preventing these failures from being discovered for quite some time. Waiting for an hour only to discover that your program died from an immediate segmentation fault increases the frustration already inherent in debugging tasks.

Furthermore, job failures turn out to be surprisingly common. Figure 1 reports the number and percentage of jobs that fail in traces from the Parallel Workloads Archive [3]. Many of the traces contain significant numbers of jobs that fail.

Based on the frequency of job failures and the frustration of waiting to discover them, we believe it is important to design schedulers so that they attempt to detect jobs that quickly fail as soon after submission as possible. Some of the failures are likely to be hardware problems, the detection of which cannot be improved by changes to the scheduler. When a job fails because of a programming error or something wrong in the runtime environment, however, this failure can be detected by starting the job soon after its submission. Since failing jobs are

Trace	Num. Jobs	Num. failed	% failed
CTC-SP2-1995-1.swf	70,918	6,972	9.8
CTC-SP2-1996-2.1-cln.swf	77,222	16,669	21.6
DAS2-fs0-2003-1.swf	219,618	2,643	1.2
DAS2-fs1-2003-1.swf	39,356	1,554	4.0
DAS2-fs2-2003-1.swf	65,382	1,994	3.1
DAS2-fs3-2003-1.swf	66,112	1,143	1.7
DAS2-fs4-2003-1.swf	32,953	602	1.8
KTH-SP2-1996-2.swf	28,489	7,948	27.9
LANL-CM5-1994-3.1-cln.swf	122,060	20,368	16.7
LANL-O2K-1999-1.swf	116,996	23,670	20.2
LLNL-Atlas-2006-1.1-cln.swf	38,194	10,250	26.8
LLNL-Thunder-2007-1.1-cln.swf	118,791	7,933	6.7
LLNL-uBGL-2006-1.swf	19,405	6,835	35.2
LPC-EGEE-2004-1.2-cln.swf	220,695	10,490	4.8
SDSC-Par-1995-2.1-cln.swf	53,970	906	1.7
SDSC-Par-1996-2.1-cln.swf	32,135	814	2.5
SHARCNET-2005-1.swf	1,194,184	1,003,277	84.0

Fig. 1. Failing jobs by trace. Only traces with at least one failing job are presented. There were 2 other traces that reported all jobs succeeding, 2 that reported all jobs having “unknown” exit status, and 4 that reported various mixtures of succeeding, canceled, or unknown exit status. Also note that the number of jobs varies from the value reported in the Parallel Workloads Archive, sometimes greatly. We exclude jobs with unknown exit status and those that were canceled without running.

only identified after they fail, this requires that all jobs be started soon after submission. If the system supports preemption, it is possible to do exactly this; as soon as a job arrives, preempt other jobs to give it sufficient processors to run for a brief period of time, after which the new job is itself preempted and the previous jobs resumed. In this way, any job failure occurring at the beginning of the job would be detected nearly immediately. If the period is brief enough, the previously-running jobs are not greatly inconvenienced. Thus, we are left with an engineering tradeoff to choose the length of a job’s initial run, with longer runs finding more failures and shorter runs minimizing disruption to already-running jobs.

Unfortunately, preemption is difficult to implement in a large multiprocessor system because preempting a job requires saving its state on each of its processors and also catching all “in flight” messages traveling between them. Because of these difficulties, many multiprocessor systems do not support preemption. Instead, our algorithms use *restarts*, in which a job can be stopped and restarted, but does so from the beginning of its execution, effectively losing its progress from the first run. Restarts are less powerful than preemption and should be simpler to implement; it is not necessary to save any state, but merely to kill the job and ignore any of its messages. It is still technically challenging to restart jobs that perform side effects (eg. file I/O), but we believe it is easier for systems to implement restarts than preemption. In exchange for being easier to implement,

restarts impose greater cost on jobs on which they are used; all work previously done on that job is lost.

Now we can give the outline of our scheduling idea. As above, we attempt to start every job soon after its submission. We call the first time a job is started its *trial run*, which we only allow to continue for a bounded period of time. Jobs that do not fail (or complete) within this time are killed to be restarted later. When a job is restarted is controlled by a *base scheduler* such as First-Come First-Served (FCFS) or EASY [9]. We call the combination of trial runs and the base scheduler a *timed-run scheduler*, which can be viewed as the base scheduler operating within a framework that manages trial runs. Our intent is for the timed-run scheduler to behave similarly to its base scheduler except for identifying failing jobs more quickly. In particular, once the base scheduler decides to start a job, that job is never restarted; our algorithm only kills jobs at the end of their trial run when relatively little work is lost by doing so. We say a job is *committed* when it has been started by the base scheduler. Exactly when a job should be committed proved to be a more subtle decision than we originally thought; we discuss this decision later in the paper.

As a side effect of giving jobs trial runs, the timed-run scheduler also benefits jobs that successfully finish within their trial run. We use the term *short jobs* to denote jobs that complete or fail during their trial run and *long jobs* to denote the others. Allowing short jobs to cut in front of longer jobs generally improves the system’s average response time, though at some cost in fairness. For a short trial run length, we believe that the affect on long jobs is minimal in exchange for the benefits provided to short jobs, especially jobs that fail immediately after they start.

We show that this strategy can greatly reduce the time to detect problems in short failing jobs, the jobs on which users will be most frustrated to wait. The benefits of our strategy extend to all short jobs, which form a significant fraction of many workloads. The improvement is achieved with a non-preemptive strategy that restarts each job at most once. It is generally realized without significantly penalizing long jobs and even improves their average and maximum response time in some cases.

These results are based on event-based simulations using traces from the Parallel Workloads Archive. We assume that the system being evaluated uses pure space-sharing to run rigid jobs.

We note that others have proposed similar strategies in the past. What distinguishes our algorithm is its focus on giving each job a fixed-length trial run as soon as possible. We also give a more thorough evaluation of trial runs in isolation, giving a detailed description of the algorithm and an examination of the effect of varying the length of a trial run.

The rest of the paper is organized as follows. In Section 2 we fully specify the timed-run scheduling strategy. Then, in Section 3 we evaluate this strategy. We discuss related work in Section 4. We conclude with a discussion of future work in Section 5.

2 Timed-Run Scheduling

Now, we are ready to formally define the timed-run algorithm. It maintains a list of jobs awaiting a trial-run in addition to whatever data structures are required for the base scheduling algorithm. Newly-arrived jobs are added to the end of this list as well as to the base scheduler’s data structures. Whenever a job arrives or processors are freed due to a job completion or termination, the timed-run scheduler traverses this list looking for jobs to start. Any jobs encountered during this traversal that can start are removed from the list and started on their trial run. Only if no jobs can start trial runs is the base scheduler allowed to start jobs.

Our goals when designing this algorithm were to give jobs their trial runs as early as possible while impacting the base scheduler as little as possible. The prioritization of trial runs is reflected in our choice to look for jobs in the trial run list before consulting the base scheduler. Because the jobs are considered for trial runs in order of their arrival, we slightly favor earlier-arriving jobs and provide some measure of fairness. The jobs are not forced to receive trial runs in the order they arrive, however, to facilitate giving as many jobs as possible their trial runs soon after they arrive. We also allow the base scheduler to run jobs even when there are still jobs waiting for trial runs (provided none of them can start) to minimize the impact on the base scheduler. This decision and allowing trial runs to occur out of order both penalize large jobs, but we felt this discrimination was justified to avoid draining the machine just for a trial run of a large job. We consider it the base scheduler’s responsibility to make such weighty decisions. In addition, we felt that failures of small jobs were more “justified” since users should test large programs on a smaller scale before running them on many processors.

The other obvious decision to make when implementing the timed run scheduler is the duration of trial runs. We initially chose 90 seconds as the trial run length because this was the value given by Mu’alem and Feitelson [10] in their discussion of failing jobs. Another value mentioned in the literature is 1 minute, which Chiang and Vernon [2] observed was sufficient to complete 12–33% of jobs requesting over an hour and 11–42% of jobs requesting over 10 hours in a trace from NCSA’s Origin 2000. They did not discuss the cause of these dramatic overestimates, but it seems likely that job failures played a role. Lawson and Smirni [7] suggest 180 seconds, which they observed to exclude most jobs that crashed. We discuss the effect of varying the trial run length in Section 3.2.

2.1 Optimizations and complications

We decided on the aspects of timed-run scheduling described above without much difficulty. While implementing it and examining the schedules produced by our initial prototype, however, we discovered a number of complications. We now describe these and the policy decisions we made to resolve them.

Jobs wait until finishing their trial runs before committing. The first complication we discovered applies even to very small input instances. What should the scheduler do when the machine is idle and a single job arrives? As described above, the algorithm will select this job for a trial run and then schedule it again by following the base scheduling algorithm. Obviously, the job should not be started twice, but it seems premature to commit it to run to completion simply because it was the first job to arrive after an idle period. Nor is this necessarily a rare case since the same situation occurs if a job starts a trial run and then is selected by the base scheduler. We resolved this by not allowing a job to commit during its trial run. During this time, the base scheduler acts as if the job cannot fit on the machine.

To improve performance when a job’s trial run and its selection by the base scheduler occur together, we implemented a fairly obvious optimization: when a job completes its trial run, if the timed run scheduler will decide to start the same job for its committed run, we simply continue that job rather than stopping and restarting it. This optimization complicates the scheduler’s logic somewhat, but clearly improves the schedule since it avoids wasting the time already spent on the trial run.

Jobs continue trial runs until replaced. After implementing the above, we noticed a related optimization. Consider the following job instance, scheduled on a 100-processor machine with 90-second trial runs and a First-Come First-Served (FCFS) base scheduler:

Job	Arrival time	Number processors	Runtime
<i>A</i>	0	80	300
<i>B</i>	100	100	40
<i>C</i>	110	20	> 90

This instance is scheduled as shown in Fig. 2. Notice that job *A* continues after its trial run because nothing else has arrived when it finishes the trial run. Job *C* does not get to continue, however, because FCFS wants to run job *B* first. Terminating job *C* at time 200 is not strictly necessary, however, since job *B* cannot start until time 300. Instead, we allow jobs that complete their trial runs to continue running until the scheduler has another use for their processors, either for a different job’s trial run or for a committed run. For the example above, this means that job *C* is allowed to continue until time 300. If it has length between 90 and 190, this allows it to complete. Even if job *C* requires more time than this, nothing is lost since the processors it uses would have been idle otherwise. Note that extensions are granted even when a job’s estimated running time indicates that it will not complete because the estimate may be inaccurate.

Avoiding restarts and extending trial runs are both achieved using lazy job termination. When a job finishes its trial run, it is added to a collection of jobs that can be terminated if needed. The scheduler makes its decisions as if all jobs in this collection had been terminated. If the scheduler decides to start a job

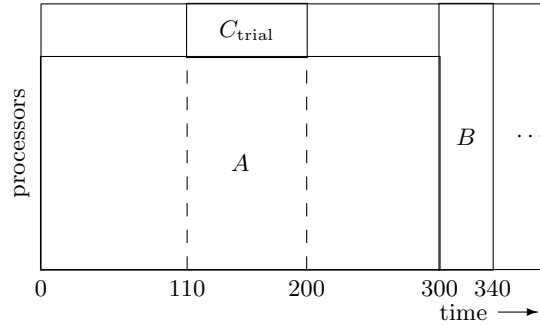


Fig. 2. Short jobs continue running until replaced

that requires some of their processors, jobs from the collection are terminated as needed, beginning with the one whose trial run ended longest ago.

Long jobs must wait for their turn in the base scheduler. The next complication we encountered required a more difficult policy decision. Consider the following instance, again scheduled on a 100-processor machine with 90-second trial runs and FCFS scheduling:

Job	Arrival time	Number processors	Runtime
<i>A</i>	0	70	90
<i>B</i>	5	70	60
<i>C</i>	10	50	200
<i>D</i>	20	20	140
<i>E</i>	25	30	40

Two possible schedules are shown in Fig. 3. The difference is in when job *D* is committed. At time 150, job *D* has completed its trial run. The other job in the system is job *C*, which has not had a trial run, but should run first according to the base scheduler (FCFS).

Our first inclination was to start job *D* immediately in this situation since it seems wasteful to idle processors while waiting for a job that has already started (albeit only for a trial run). Our eventual conclusion, however, was to delay job *D* until its predecessor gets committed. The reason for this decision is to allow for the possibility that another job arrives during the trial run of job *C*. If we committed job *D* and a newly-arrived job prevents job *C* from committing, then the timed run scheduler would have committed jobs out of the order given by the base scheduler, violating our intention to make the timed run scheduler an augmentation of the base scheduler rather than its replacement. Note that delaying when jobs are committed in this way could harm our performance. An alternate solution would be to start job *D*, but kill it if job *C* ends up not committing. This would be similar to the speculative backfilling of Perković and Keleher [12].

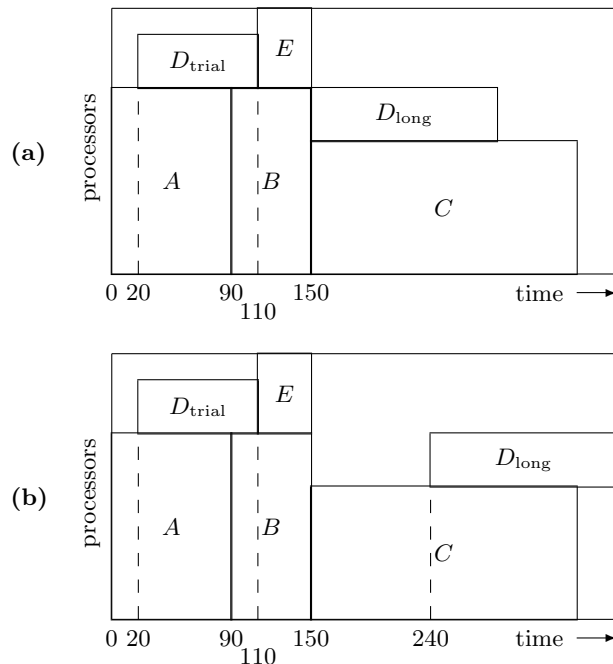


Fig. 3. Two possible ways to schedule the long run of job waiting for a job starting its trial run. In (a), job D starts as soon as job C begins its trial run. In (b), it waits for job C to be committed.

Dealing with job reservations. The final complication we encountered while implementing the timed run strategy is how to combine it with base schedulers where jobs are given reservations. In keeping with our goal to give each job a trial run shortly after it arrives, our algorithm favors trial runs over committing jobs in the order given by the base scheduler. This means reservations may be violated since newly-arrived jobs can (briefly) grab processors at any time. However, we do recognize that reservations are desirable from a user perspective since they promote fairness and make the system more predictable for users. Thus, we wished to achieve a compromise by preserving the spirit of reservation-wielding base schedulers while violating the specific reservations.

For the EASY scheduler, there is a relatively straightforward way to achieve this compromise. We simply disabled the error checking that reports when a guarantee is violated. This works because our implementation of EASY (following [10]) does not build an entire schedule. Rather, it stores the jobs in arrival order, the currently-running jobs with their estimated completion times, and the first job's guaranteed time. To make a scheduling decision, it traverses the list of waiting jobs and starts any job that can be run without violating the guarantee.

It is much less clear how to use timed-run scheduling with algorithms that provide guarantees to more than one job such as Conservative backfilling. One solution is to rebuild the estimated schedule whenever trial runs cause it to break, but this could greatly slow down the scheduler. Another idea is to give

initial guarantees with some slack to allow for trial runs by later jobs, but this seems to violate the spirit of Conservative backfilling. We believe more research is warranted on this question.

3 Experimental Results

To evaluate the timed-run strategy, we used an event-driven simulator. Events were generated for job arrivals, job completions, and at the end of trial runs. The data for our simulations were obtained from the online Parallel Workloads Archive [3]. All traces were in the standard workload format, from which we read the job arrival time, processors requested, actual running time, and user-estimated runtime (when available). For actual runtime, we used field 4 (“run time”) if it was available and field 6 (“Average CPU time used”) if it was not. We also used the status field (number 11) to identify failing jobs, but only as a post-processing step. Cleaned versions of the traces were used when available; the full filenames for the used traces are given in Fig. 1. We excluded the SHARCNET trace from our simulations because of its extraordinarily-high failure rate.

3.1 Ninety second trial runs

We compared FCFS and EASY schedulers to their timed-run counterparts using average and maximum waiting time. We used waiting time since it is in line with our goal to minimize the absolute time before detecting a failure. It also lessens the emphasis on small jobs relative to slowdown or bounded slowdown. Note that the waiting times we record for a job under the timed-run scheduler is until that job starts the run that finishes, NOT the wait until the job gets a trial run. Put another way, the waiting time of a job is its completion time minus its arrival time minus its actual running time.

Our initial simulations used a trial-run length of 90 seconds. Fig. 4 gives the results with the average or maximum taken over all jobs. Fig. 5 shows the percent improvement in average response time achieved by switching from a normal scheduler to a timed-run scheduler. (The traces are numbered in the order they appear in Fig. 1.) From the results, the timed-run scheduler generally performs as expected, decreasing average waiting time in nearly all cases. The exceptions are all in the DAS2 family of traces. These traces, from a group of clusters used for distributed computing research, have quite low utilization (all less than 20%) so they are not representative of typical production workloads. Quite a few of the improvements in the other traces are significant, particularly with the FCFS base scheduler.

Also importantly, the improvement in average waiting time does not occur at the expense of increased maximum waiting time. Instead, maximum waiting time is largely unchanged, with the worst result an increase of less than 4%. On the KTH-SP2 and LANL-CM5 traces, using timed-run scheduling with FCFS actually improves it by a significant margin.

The results were better for FCFS than EASY. This is unsurprising since trial runs can act as an ad hoc version of backfilling. When working with FCFS, there are many opportunities for jobs to move up and the result is a significantly better schedule. The EASY base scheduler does a better job keeping the processors busy and so offers less room for improvement.

	Trace	Average waiting time			Max waiting time		
		Regular	Timed	% Imp.	Regular	Timed	% Imp.
FCFS	CTC-SP2-1995	1,510,879	375,397	75.2	4,591,814	4,523,823	1.5
	CTC-SP2-1996	17,404	9,075	47.9	151,596	149,117	1.6
	DAS2-fs0	542	593	-9.4	61,016	61,884	-1.4
	DAS2-fs1	194	221	-13.9	148,765	148,765	0
	DAS2-fs2	1,259	2,121	-68.5	943,142	943,322	-0.0
	DAS2-fs3	1,724	14	99.2	139,955	139,955	0
	DAS2-fs4	1,790	715	60.1	213,849	214,185	-0.2
	KTH-SP2	389,892	42,893	89.0	1,018,341	535,982	47.4
	LANL-CM5	79,474	35,659	55.1	470,233	422,697	10.1
	LANL-O2K	6,626	1,716	74.1	195,670	195,676	-0.0
	LLNL-Atlas	76	24	68.4	23,256	20,792	10.6
	LLNL-Thunder	14,815	2,610	82.4	119,059	117,149	1.6
	LLNL-uBGL	343	332	3.2	9,478	9,478	0
	LPC-EGEE	212	130	38.7	57,115	57,829	-1.3
	SDSC-Par95	31,748	7,181	77.4	384,382	356,854	7.2
SDSC-Par96	30,217	12,669	58.1	211,381	207,343	1.9	
EASY	CTC-SP2-1995	30,832	25,279	18.0	3,433,169	3,255,087	5.2
	CTC-SP2-1996	3,325	2,932	11.8	148,392	152,919	-3.1
	DAS2-fs0	326	584	-79.1	61,796	62,170	-0.6
	DAS2-fs1	185	217	-17.3	148,765	148,765	0
	DAS2-fs2	543	1,526	-181.0	943,142	943,322	-0.0
	DAS2-fs3	9	9	0	139,955	139,955	0
	DAS2-fs4	1,213	712	41.3	213,849	214,185	-0.2
	KTH-SP2	6,856	5,607	18.2	262,194	264,395	-0.8
	LANL-CM5	8,242	7,489	9.1	180,064	180,884	-0.5
	LANL-O2K	588	573	2.6	195,026	195,676	-0.3
	LLNL-Atlas	29	22	24.1	25,669	25,669	0
	LLNL-Thunder	338	227	32.8	117,757	117,900	-0.1
	LLNL-uBGL	337	332	1.5	9,478	9,478	0
	LPC-EGEE	212	130	38.7	57,115	57,829	-1.3
	SDSC-Par95	4,768	3,991	16.3	331,475	330,651	0.2
SDSC-Par96	8,853	7,320	17.3	192,838	190,935	1.0	

Fig. 4. Waiting time for all jobs with 90 second trial run length. All times are in seconds.

Since our main motivation was to promptly identify failing jobs so their users could be notified soon after the jobs have been submitted, we also compare the failing job waiting times between regular scheduling and timed-run scheduling in Fig. 6. Surprisingly, the results were not as good for failing jobs as they

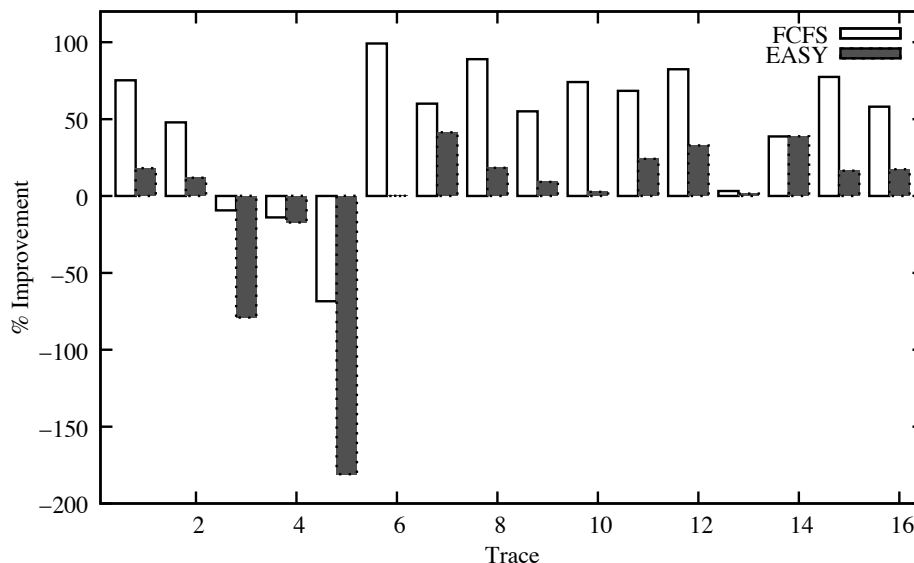


Fig. 5. Percent improvement in average waiting time for all jobs from normal to timed-run scheduler

were for all jobs. The percentage improvements for average waiting time are generally smaller for FCFS and they essentially disappear for EASY. The other patterns are still there, though; FCFS is improved much more than EASY, the DAS2 traces contributed negative outliers to the percent improvement in average waiting time, and the affect on maximum waiting time of adding trial runs ranges is minimal with some improvements and a couple of good values.

We explain this with the observation that failing jobs are not necessarily short jobs. Although failing jobs ending prematurely is consistently one of the explanations given for the poor quality of user estimates, it turns out that job failures do not cause the short jobs in these traces. Figure 7 gives the percent of all jobs and the percent of failed jobs that are short in each trace. For all but 3 of the 16 traces, short jobs make up a smaller percentage of failing jobs than they represent of the trace as a whole. Only in LLNL-uBGL of these three is the difference large. However, there seems to be no relationship between the results in Fig. 4 and Fig. 6 and the percentage of failed jobs that are short. This could be due to the fact that the total number of failed jobs that are short is small compared to the total number of jobs that are short.

Figure 8 shows the average and maximum waiting times for short jobs. Providing jobs with trial runs does result in short jobs waiting for considerably less time before running. Figure 9 shows the average and maximum waiting times for failed short jobs. The results were similar for average waiting time, but considerably improved for maximum waiting time. Here again, there is no relationship between the results in Fig. 8 and Fig. 9 and the percentage of all and failed jobs that are short because the numbers of short failed jobs are much smaller

	Trace	Average waiting time			Max waiting time		
		Regular	Timed	% Imp.	Regular	Timed	% Imp.
FCFS	CTC-SP2-1995	1,380,025	425,579	69.2	4,591,772	4,523,823	1.5
	CTC-SP2-1996	16,934	10,115	40.3	147,977	149,117	-0.8
	DAS2-fs0	179	336	-87.7	51,403	30,776	40.1
	DAS2-fs1	374	373	0.3	71,090	71,090	0
	DAS2-fs2	24,187	19,516	19.3	583,024	583,204	-0.0
	DAS2-fs3	554	231	58.3	91,801	85,929	6.4
	DAS2-fs4	3,718	272	92.7	153,213	105,992	30.8
	KTH-SP2	372,511	53,781	85.6	1,011,003	521,937	48.4
	LANL-CM5	78,565	52,569	33.1	470,188	419,325	10.8
	LANL-O2K	8,293	2,804	66.2	195,004	154,073	21.0
	LLNL-Atlas	94	43	54.3	20,792	20,792	0
	LLNL-Thunder	16,451	2,369	85.6	118,490	117,149	1.1
	LLNL-uBGL	960	939	2.2	9,478	9,478	0
	LPC-EGEE	80	76	5	39,890	19,183	51.9
	SDSC-Par95	36,868	29,690	19.5	303,115	243,298	19.7
SDSC-Par96	27,604	26,353	4.5	192,790	191,162	0.8	
EASY	CTC-SP2-1995	40,435	37,759	6.6	3,377,733	3,199,651	5.3
	CTC-SP2-1996	3,805	3,426	10.0	139,269	123,526	11.3
	DAS2-fs0	59	333	-464.4	5,134	30,735	-498.7
	DAS2-fs1	341	365	-7.0	71,090	71,090	0
	DAS2-fs2	1,573	1,617	-2.8	583,024	583,204	-0.0
	DAS2-fs3	170	85	50	3,728	4,130	-10.8
	DAS2-fs4	250	264	-5.6	105,838	105,992	-0.1
	KTH-SP2	6,746	6,066	10.1	248,239	250,350	-0.9
	LANL-CM5	9,997	10,959	-9.6	168,939	177,924	-5.3
	LANL-O2K	747	740	0.9	44,053	44,703	-1.5
	LLNL-Atlas	60	48	20	25,669	25,669	0
	LLNL-Thunder	390	306	21.5	115,287	117,446	-1.9
	LLNL-uBGL	953	939	1.5	9,478	9,478	0
	LPC-EGEE	80	76	5	39,890	19,183	51.9
	SDSC-Par95	13,108	13,667	-4.3	153,504	152,767	0.5
SDSC-Par96	16,138	16,688	-3.4	181,275	190,935	-5.3	

Fig. 6. Waiting times for failing jobs with 90 second trial run length. All times are in seconds.

short jobs as...	% of jobs	% of failed jobs
CTC-SP2-1995	27.4 (19,404 jobs)	12.6 (880 jobs)
CTC-SP2-1996	21.6 (16,699 jobs)	15.0 (2,507 jobs)
DAS2-fs0	61.5 (134,991 jobs)	50.4 (1,331 jobs)
DAS2-fs1	65.6 (25,803 jobs)	33.1 (514 jobs)
DAS2-fs2	64.5 (42,191 jobs)	10.4 (207 jobs)
DAS2-fs3	76.1 (50,321 jobs)	74.6 (853 jobs)
DAS2-fs4	42.9 (14,129 jobs)	48.0 (289 jobs)
KTH-SP2	32.9 (9,375 jobs)	28.5 (2,267 jobs)
LANL-CM5	30.2 (36,910 jobs)	8.1 (1,650 jobs)
LANL-O2K	30.9 (36,132 jobs)	20.6 (4,877 jobs)
LLNL-Atlas	51.9 (19,809 jobs)	47.5 (4,872 jobs)
LLNL-Thunder	59.1 (70,246 jobs)	65.2 (5,176 jobs)
LLNL-uBGL	56.7 (11,008 jobs)	92.3 (6,306 jobs)
LPC-EGEE	69.9 (154,221 jobs)	9.7 (1,013 jobs)
SDSC-Par95	60.9 (32,845 jobs)	0.1 (1 job)
SDSC-Par96	44.4 (14,268 jobs)	0.5 (4 jobs)

Fig. 7. Short jobs by trace

than the numbers of short jobs and a meaningful comparison cannot be made between the two.

3.2 Varying trial-run length

We also investigated the effects of varying the length of the trial-run. An ideal length would balance catching failing jobs and increasing responsiveness by letting short jobs finish during their trial-run against making jobs wait too long while trial-runs occur. For this experiment, we tried trial-run lengths increasing from 0 (no trial-runs) to 400 seconds.

For our experiment, performance was based on the average and maximum for job waiting times. We generated separate statistics for long jobs (those that do not finish during their trial run), short jobs (jobs that finish during their trial run) and all jobs (long and short). We did this to see how long potential failed jobs that could be identified (short jobs) would have to wait. It was also useful to see how long and short jobs affect the performance of the scheduling strategy.

The results from our experiment are presented in Fig. 10–15. Figures 10 and 11 show the average waiting time for short jobs. Note that a “short” job is one shorter than the trial run length so the jobs considered varies with the trial run length. This figure provides an idea of how much time jobs failing within their trial run need to wait. Quick trial runs go through all available jobs faster, and so short job waiting times are lower since they get to finish quickly. However, the data have a distinct spike for extremely short trial-run lengths. This is because there are only a few jobs having extremely low runtimes, and when they do appear in the system, they need to wait for long jobs to finish and free processors before they get a chance to run. The average waiting time decreases after the spike since there are now more short jobs and they do not all need to wait for

	Trace	Average waiting time			Max waiting time		
		Regular	Timed	% Imp.	Regular	Timed	% Imp.
FCFS	CTC-SP2-1995	1,318,476	228	100.0	4,590,152	44,471	99.0
	CTC-SP2-1996	16,699	168	99.0	136,578	44,812	67.2
	DAS2-fs0	368	38	89.7	61,016	30,829	49.5
	DAS2-fs1	99	81	18.2	148,765	148,765	0
	DAS2-fs2	193	17	91.2	493,691	5,184	99.0
	DAS2-fs3	2,233	8	99.6	139,955	139,955	0
	DAS2-fs4	2,971	363	87.8	213,849	167,308	21.8
	KTH-SP2	385,907	1,447	99.6	1,011,593	332,674	67.1
	LANL-CM5	88,619	774	99.1	469,988	131,094	72.1
	LANL-O2K	4,333	48	98.9	195,670	194,164	0.8
	LLNL-Atlas	80	2	97.5	18,650	13,159	29.4
	LLNL-Thunder	13,601	21	99.8	119,015	27,750	76.7
	LLNL-uBGL	600	583	2.8	9,478	9,478	0
	LPC-EGEE	198	63	68.2	52,944	31,915	39.7
	SDSC-Par95	31,669	488	98.5	377,558	93,456	75.2
SDSC-Par96	34,299	902	97.4	209,900	69,317	67.0	
EASY	CTC-SP2-1995	16,522	990	94.0	1,796,427	893,478	50.3
	CTC-SP2-1996	1,281	319	75.1	76,878	46,473	39.5
	DAS2-fs0	169	39	76.9	61,796	30,919	50.0
	DAS2-fs1	96	81	15.6	148,765	148,765	0
	DAS2-fs2	43	16	62.8	14,867	3,902	73.8
	DAS2-fs3	10	8	20	139,955	139,955	0
	DAS2-fs4	2,476	363	85.3	213,849	167,308	21.8
	KTH-SP2	4,810	1,877	61.0	196,289	196,212	0.0
	LANL-CM5	5,334	925	82.7	149,382	122,215	18.2
	LANL-O2K	289	66	77.2	194,962	194,164	0.4
	LLNL-Atlas	20	3	85	18,036	18,036	0
	LLNL-Thunder	119	14	88.2	103,251	18,501	82.1
	LLNL-uBGL	593	583	1.7	9,478	9,478	0
	LPC-EGEE	198	63	68.2	52,944	31,915	39.7
	SDSC-Par95	2,386	858	64.0	150,065	97,707	34.9
SDSC-Par96	4,524	1,082	76.1	124,194	69,317	44.2	

Fig. 8. Waiting times for short jobs with 90 second trial run length. All times are in seconds.

	Trace	Average waiting time			Max waiting time		
		Regular	Timed	% Imp.	Regular	Timed	% Imp.
FCFS	CTC-SP2-1995	1,269,261	130	100.0	4,577,842	9,826	99.8
	CTC-SP2-1996	13,863	223	98.4	136,139	44,812	67.1
	DAS2-fs0	231	7	97.0	51,403	2,869	94.4
	DAS2-fs1	22	1	95.5	5,063	402	92.1
	DAS2-fs2	27,366	124	99.5	493,691	1,605	99.7
	DAS2-fs3	168	35	79.2	498	434	12.9
	DAS2-fs4	5,376	44	99.2	153,213	3,882	97.5
	KTH-SP2	375,074	1,902	99.5	1,009,877	123,851	87.7
	LANL-CM5	88,591	1,085	98.8	468,623	91,456	80.5
	LANL-O2K	9,043	122	98.7	195,004	37,286	80.9
	LLNL-Atlas	91	5	94.5	15,384	13,159	14.5
	LLNL-Thunder	18,610	100	99.5	117,910	20,144	82.9
	LLNL-uBGL	1,040	1,017	2.2	9,478	9,478	0
	LPC-EGEE	199	24	87.9	39,890	10,052	74.8
	SDSC-Par95	214	62	71.0	214	62	71.0
SDSC-Par96	12,777	623	95.1	43,400	2,355	94.6	
EASY	CTC-SP2-1995	22,819	2,356	89.7	1,429,601	679,629	52.5
	CTC-SP2-1996	1,980	440	77.8	76,878	46,473	39.5
	DAS2-fs0	31	7	77.4	3,524	2,869	18.6
	DAS2-fs1	3	1	66.7	1,149	402	65.0
	DAS2-fs2	437	124	71.6	6,284	1,606	74.4
	DAS2-fs3	168	35	79.2	498	434	12.9
	DAS2-fs4	49	44	10.2	3,882	3,882	0
	KTH-SP2	3,791	1,696	55.3	196,289	196,212	0.0
	LANL-CM5	6,893	1,217	82.3	118,805	91,075	23.3
	LANL-O2K	461	183	60.3	37,194	41,770	-12.3
	LLNL-Atlas	36	6	83.3	18,036	18,036	0
	LLNL-Thunder	146	14	90.4	49,357	3,085	93.7
	LLNL-uBGL	1,033	1,017	1.5	9,478	9,478	0
	LPC-EGEE	199	24	87.9	39,890	10,052	74.8
	SDSC-Par95	62	62	0	62	62	0
SDSC-Par96	2,356	875	62.9	7,269	2,355	67.6	

Fig. 9. Waiting times for failed short jobs with 90 second trial run length. All times are in seconds.

long jobs to finish. The waiting time increases after that because we are adding more overhead time for each job to have a trial run, and that also allows more jobs to enter the system.

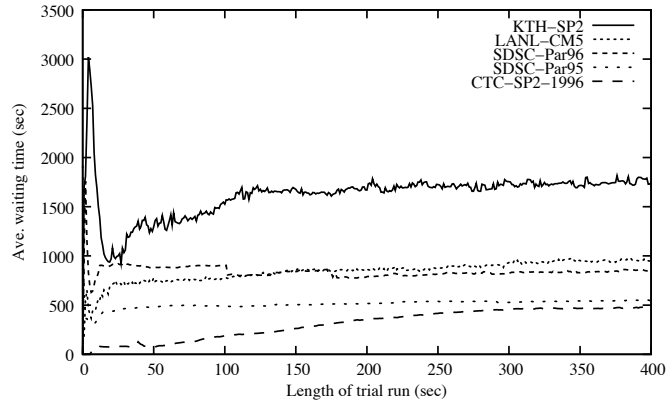


Fig. 10. Average waiting time for short jobs with varying trial run lengths (FCFS)



Fig. 11. Average waiting time for short jobs with varying trial run lengths (EASY)

Figures 12 and 13 show the overall average waiting time for all jobs over increasing trial-run lengths. The two behaviors we see are gradually decreasing and gradually increasing average waiting time. For the most part we see a gradual decrease in average waiting time as the trial time is increased. This is probably due to an increasing number of jobs becoming short and having their waiting times dramatically reduced. In the one trace (LANL-CM5) where this does not

happen, the waiting time increases probably due to the overhead time for each job to have a trial run.

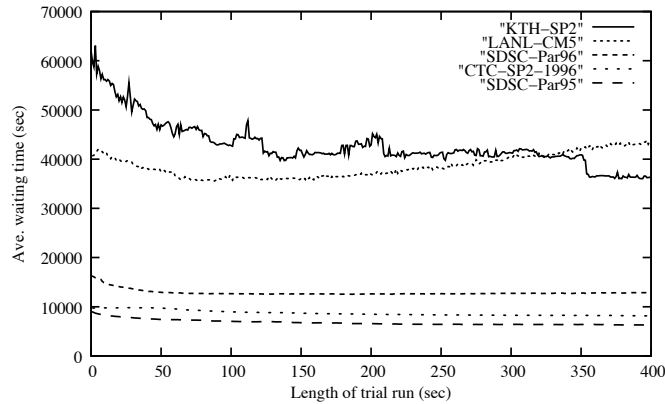


Fig. 12. Average waiting time for all jobs with varying trial run lengths (FCFS)

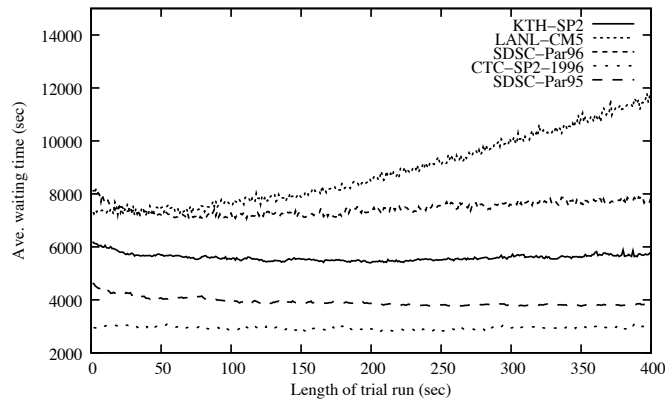


Fig. 13. Average waiting time for all jobs with varying trial run lengths (EASY)

Long jobs have a higher maximum waiting times than short jobs. Figures 14 and 15, which show the maximum waiting time for all jobs, are also the same graphs as the maximum waiting time for long jobs. The two behaviors we see are the steps downward and the gradually increasing maximum waiting time. The reasons for both are simple. For the flat-line in the steps, that maximum waiting time is due to the same long job waiting in the base scheduler. However, when the trial runs are long enough, that job gets to run as a short job and does not need to wait in the base scheduler. So the maximum waiting time drops.

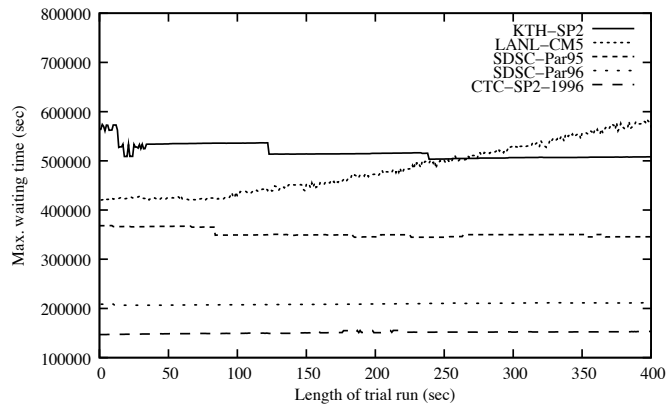


Fig. 14. Maximum waiting time for all jobs (FCFS)

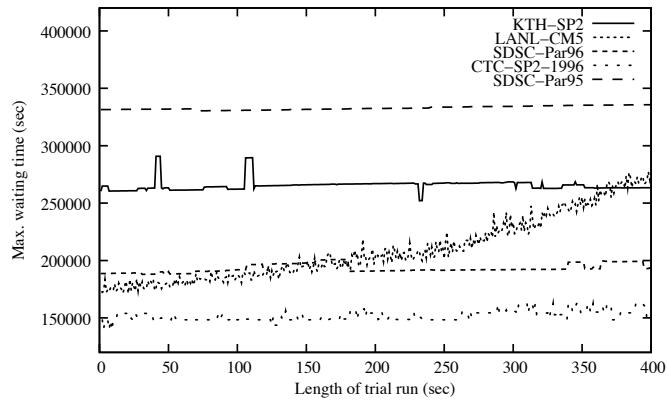


Fig. 15. Maximum waiting time for all jobs (EASY)

The gradual increase in the maximum waiting time is due to the extra overhead time from running each trial for a longer duration. This also allows more jobs to enter the system which increases the chances of jobs getting pushed further back.

From Fig. 10–15, we see that both the sixty seconds of Chiang and Vernon [2] and the ninety seconds of Mu’alem and Feitelson [10] give a reasonable balance between finding failures and minimizing wasted time during trial runs. The value of 180 seconds suggested by Lawson and Smirni [7] is also reasonable, but perhaps a bit too high, particularly for the LANL-CM5 trace.

4 Background and Related Work

Several other researchers have devised schedulers around the observation that many short jobs are submitted with greatly inflated estimates. The most similar idea appears in a system described by Perković and Keleher [12]. Their scheduler uses a number of different techniques, but among them are “speculative test runs” and “speculative backfilling”. Speculative test runs are a version of our trial runs; jobs with long estimated running time (over 3 hours) are allowed a brief run on the machine (5–15 minutes) in the hopes of finishing early. This differs from what we do in that we give a trial run to all jobs whereas Perković and Keleher [12] give speculative test runs only to some fraction of the jobs and do so primarily as part of a larger speculative scheduling phase.

The other part of Perković and Keleher’s speculative scheduling phase is speculative backfilling: starting a job in a “hole” that occurs in the schedule even when that job will not be able to complete unless its running time is over-estimated. At the end of the hole, the speculatively backfilled job is killed if it has not already finished. In this way, only processors that would have been idle anyway are used in the speculation. This is similar to what our scheduler does when it continues to run jobs whose trial runs have expired, but we do not purposely start jobs speculatively after their single trial run (though we could). Again, this differs from our scheduler because we give trial runs to all jobs. The other main difference between our work and that of Perković and Keleher [12] is in our tighter focus; due to the number of ideas presented in their paper, they describe the idea only briefly and do not analyze the effect of this optimization alone or the effect of varying the length of a speculative execution.

Snell et al. [14] explored an idea similar to speculative backfilling. They allowed jobs to backfill even when there was not enough time in the schedule for them to complete, killing running jobs as needed to honor reservations. They considered a number of criteria for selecting the jobs to kill, finding that it was best to either kill the job with the most (estimated) time remaining or the job that was most recently started. These strategies improved system performance, but by relatively small amounts, apparently because of the work lost when jobs were killed. (Unlike in our strategy, they might kill a job that had already been processed for a considerable period of time.)

Lawson and Smirni [7] also use speculative execution to identify jobs that are much shorter than estimated. Their algorithm is based on work by Lawson et al. [6] that places jobs into separate queues based on their running time. Each queue is serviced by part of the system so that short jobs do not wait for long jobs but no job can starve. The base algorithm assumes that job durations were estimated accurately, but Lawson and Smirni [7] found that similar ideas work when using estimated running times as long as jobs whose estimated running time exceeds 1,000 seconds were given a 180-second speculative run as soon as possible to detect over-estimates. Their algorithm was shown to improve the slowdown of nearly all types of jobs, but sometimes hurt the very longest jobs. They also considered the effect of jobs having different priorities or some of the jobs having predetermined reservations, but did not consider changing the length of the speculative run.

A manual version of timed-run scheduling was also proposed by Chiang et al. [1]. They were concerned with the accuracy of user estimates and felt that users would be able to more accurately predict the runtime of many jobs by first making a “test run” on a smaller version of the problem or with slightly different input parameters. They tested the effect of test runs equaling 10% of the estimated run time but not more than 1 hour and then users submitting the real job with reasonably accurate estimates. They showed that such a scheme would lead to performance improvements despite the overhead of the test runs. As with our algorithm, their test runs have the effect of identifying and finishing short jobs quickly. Our system differs in that it makes trial runs automatically rather than assuming users would make them manually. The runs themselves are less time-consuming in our scheduler, but also do not provide improved estimates.

Others who have considered similar ideas to timed-run scheduling have done so in the context of systems that support preemption, which is much more flexible than the job restarts that we allow. Most related is work by Chiang and Vernon [2]; they consider backfilling with “immediate service”, which attempts to give each newly-arriving job a one-minute run before putting it in the queue. It does this by preempting the currently-running jobs with lowest current slowdown among jobs that have not been preempted in 10 minutes. They showed that this strategy significantly improves average slowdown while having minimal effect on 95th percentile waiting time. This is similar to our results with the FCFS base scheduler, but preemption allows them to improve even on a scheduler with backfilling. For other strategies utilizing preemption, see Kettimuthu et al. [5] and its references.

Although technically dissimilar, our overall goal is analogous to that of Shmueli and Feitelson [13], who argue that user productivity is a better metric than waiting time or slowdown. Their scheduler attempts to prioritize jobs whose submitter is likely to still be waiting for the result. Thus, jobs that can be finished shortly after submission are more critical than either long jobs or jobs that have already waited for a significant period of time. This is done so that users are able to continue working rather than needing to switch to another task and incurring a human “context switch” when they refocus on the task

requiring the supercomputer system. Intuitively, the success of our algorithm in quickly identifying failing or unexpectedly short jobs should have a similar benefit. Evaluating this would require modeling the user (as Shmueli and Feitelson [13] do) so that system performance affects job arrival rather than relying on the trace-based simulations we present here.

Our results can also be considered related to the work considering the effect of user estimates on scheduling performance. Many authors (e.g. [10]) have noted that user estimates tend to be dramatically high, partially because of very short jobs (including quick failures) and partially because users tend to reuse estimates once they find something that works and also strongly prefer “round” numbers (e.g. 5 minutes, 1 hour, etc). Accounts initially differed on whether overestimates improved [10, 18] or hurt [1] performance. These observations were eventually reconciled with the finding that overestimates initially help but that extreme overestimates eventually hurt performance [15, 17]. Regardless of this, it seems reasonable that good estimates could be useful since they provide more information to the scheduler. This has led to work trying to get users to improve their estimates [8] as well as work to have a system generate its own estimates [4, 11, 16].

5 Discussion

Our results in Section 3 show that timed-run scheduling can more quickly alert users about jobs that fail and benefit short jobs in general. In some cases, it has even been shown to improve average and maximum waiting times for the entire trace. We feel that this approach is promising and deserves further investigation.

The most obvious open problem is to adapt this strategy to other backfilling schemes. As mentioned in Section 2.1, it is not clear how to preserve the benefits of reservations when the reservations themselves may be violated when trial runs are granted to new jobs. One solution would be to rebuild the estimated schedule whenever trial runs cause reservations to be violated. Another solution would be to give initial guarantees with some slack to allow for trial runs by later arriving jobs. Each of these solutions has its own drawbacks, and we believe more research is necessary to address reservations.

Additionally, experiments with some of our policy decisions in Section 2.1 could further improve the performance of timed-run scheduling. Specifically, the decision that long jobs must wait for their turn in the base scheduler could be replaced by a policy similar to the speculative backfilling of Perković and Keleher [12]. Also, more realism can be introduced into the simulations by modeling the overhead associated with terminating and restarting jobs.

It would also be interesting to evaluate the performance of our algorithm on user models such as those of Shmueli and Feitelson [13] to see if our quick completion of short jobs improves user satisfaction and productivity.

Acknowledgments. O. Thebe and D.P. Bunde were partially supported by contract 763836 from Sandia National Laboratories. We thank the anonymous refer-

ees for many helpful comments. We also thank all those who contributed traces to the Parallel Workloads Archive. Specifically, we thank the following people for the traces we used: Dan Dwyer for the CTC-SP2 traces, Hui Li, David Groep, and Lex Wolters for the DAS2 traces, Lars Malinowsky for the KTH-SP2 trace, Curt Canada for the LANL-CM5 trace, Fabrizio Petrini for the LANL-O2K trace, Moe Jette for the LLNL-Atlas, LLNL-Thunder and LLNL-uBGL traces, Emmanuel Medernach for the LPC-EGEE trace, Reagan Moore and Allen Downey for the SDSC traces, and John Morton and Clayton Chrusch for the SHARCNET trace.

References

1. S.-H. Chiang, A. Arpaci-Dusseau, and M.K. Vernon. The impact of more accurate requested runtimes on production job scheduling performance. In *Proc. 8th Workshop on Job Scheduling Strategies for Parallel Processing*, number 2537 in LNCS, pages 103–127, 2002.
2. S.-H. Chiang and M.K. Vernon. Production job scheduling for parallel shared memory systems. In *Proc. 15th IEEE Intern. Parallel and Distributed Processing Symp.*, 2001.
3. D. Feitelson. The parallel workloads archive. <http://www.cs.huji.ac.il/labs/parallel/workload/index.html>.
4. R. Gibbons. A historical application profiler for use by parallel schedulers. In *Proc. 3rd Workshop on Job Scheduling Strategies for Parallel Processing*, number 1291 in LNCS, 1997.
5. R. Kettimuthu, V. Subramani, S. Srinivasan, T. Gopalsamy, D.K. Panda, and P. Sadayappan. Selective preemption strategies for parallel job scheduling. *Intern. J. of High Performance Computing and Networking*, 3(2/3):122–152, 2005.
6. B. Lawson, E. Smirni, and D. Puiu. Self-adapting backfilling scheduling for parallel systems. In *Proc. 31st Intern. Conf. on Parallel Processing*, pages 583–592, 2002.
7. B.G. Lawson and E. Smirni. Multiple-queue backfilling scheduling with priorities and reservations for parallel systems. In *Proc. 8th Workshop on Job Scheduling Strategies for Parallel Processing*, number 2537 in LNCS, 2002.
8. C.B. Lee, Y. Schwartzman, J. Hardy, and A. Snavely. Are user runtime estimates inherently inaccurate? In *Proc. 10th Workshop on Job Scheduling Strategies for Parallel Processing*, number 3277 in LNCS, 2004.
9. D. Lifka. The ANL/IBM SP scheduling system. In *Proc. 1st Workshop on Job Scheduling Strategies for Parallel Processing*, number 949 in LNCS, pages 295–303, 1995.
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. A. Nissimov and D.G. Feitelson. Probabilistic backfilling. In *Proc. 13th Workshop on Job Scheduling Strategies for Parallel Processing*, number 4942 in LNCS, 2007.
12. D. Perković and P.J. Keleher. Randomization, speculation, and adaptation in batch schedulers. In *Proc. 2000 ACM/IEEE Conf. on Supercomputing*, 2000.
13. E. Shmueli and D.G. Feitelson. On simulation and design of parallel-systems schedulers: Are we doing the right thing? *IEEE Trans. Parallel and Distributed Systems*, to appear.

14. Q.O. Snell, M.J. Clement, and D.B. Jackson. Preemption based backfill. In *Proc. 8th Workshop on Job Scheduling Strategies for Parallel Processing*, number 2537 in LNCS, pages 24–37, 2002.
15. S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. Characterization of backfilling strategies for parallel job scheduling. In *Proc. Intern. Conf. on Parallel Processing Workshops*, pages 514–522, 2002.
16. D. Tsafir, Y. Etsion, and D.G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Trans. on Parallel and Distributed Systems*, 18(6):789–803, 2007.
17. D. Tsafir 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.
18. D. Zotkin and P.J. Keleher. Job-length estimation and performance in backfilling schedulers. In *Proc. 8th IEEE International Symposium on High Performance Distributed Computing*, pages 236–243, 1999.