Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling

Ahuva W. Mu'alem   Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University, 91904 Jerusalem, Israel
feit@cs.huji.ac.il

Abstract

Scheduling jobs on the IBM SP2 system and many other distributed-memory MPPs is usually done by giving each job a partition of the machine for its exclusive use. Allocating such partitions in the order that the jobs arrive (FCFS scheduling) is fair and predictable, but suffers from severe fragmentation, leading to low utilization. This situation led to the development of the EASY scheduler which uses aggressive backfilling: small jobs are moved ahead to fill in holes in the schedule, provided they do not delay the *first* job in the queue. We compare this approach with a more conservative approach, in which small jobs move ahead only if they do not delay *any* job in the queue, and show that the relative performance of the two schemes depends on the workload: for workloads typical on SP2 systems, the aggressive approach is indeed better, but for other workloads both algorithms are similar. In addition we study the sensitivity of backfilling to the accuracy of the runtime estimates provided by the users, and find a very surprising result: backfilling actually works better when users over-estimate the runtime by a substantial factor.

Keywords: parallel job scheduling, backfilling, runtime estimates, workload modeling, performance metrics.

1 Introduction

The scheduling scheme used on most distributed-memory parallel supercomputers is variable partitioning, meaning that each job receives a partition of the machine with its desired number of processors [5]. Such partitions are allocated in a first-come first-serve (FCFS)

This paper supercedes the preliminary version published in IPPS/SPDP'98 [7].

©2001 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

manner to submitted jobs. But this approach suffers from fragmentation, where free processors cannot meet the requirements of the next job, and therefore remain idle until additional ones become available. As a result system utilization is typically in the range of 50–80% [21, 16, 8, 11, 15].

It is well known that the best solutions for this problem are to use dynamic partitioning [20] or gang scheduling [6]. However, these schemes have practical limitations. The only efficient and widely used implementation of gang scheduling was the one on the CM-5 Connection Machine; other commercial implementations are too coarse-grained for real interactive support, and do not enjoy much use. To the best of our knowledge, dynamic partitioning has not been implemented on production machines at all.

A simpler approach is to re-order the jobs in the queue, that is, to use non-FCFS policies [9]. Consider a scenario where a number of jobs are running side by side, and the next queued job requires all the processors in the system. An FCFS scheduler would then reserve all the processors that are freed for this queued job, and leave them idle. A non-FCFS scheduler would schedule some other smaller jobs, that are behind the big job in the queue, rather than letting the processors idle [12, 1]. Of course, this runs the danger of starving the large job, as small jobs continue to pass it by. The typical solution to this problem is to allow only a limited number of jobs to leapfrog a job that cannot be serviced, and then start to reserve (and idle) the processors anyway. The point at which the policies are switched can be chosen so as to amortize the idleness over more useful computation, by causing jobs that create significant idleness to wait more before making a reservation.

A somewhat more sophisticated policy is to require users to estimate the runtime of their jobs. Using this information, only short jobs — that are expected to terminate in time — are allowed to leapfrog a waiting large job. This approach, which is called backfilling, was developed for the IBM SP1 parallel supercomputer installed at Argonne National Laboratory as part of EASY (the Extensible Argonne Scheduling sYstem) [17], which has since been integrated with the LoadLeveler scheduler from IBM for the SP2 [23]. Users are expected to provide accurate runtime estimates, as a low estimation may lead to killing the job before it terminates, while a high estimation may lead to a long wait time and possibly to excessive CPU quota loss.

The EASY backfilling algorithm only checks that jobs that move ahead in the queue do not delay the first queued job. We show that this aggressive approach can lead to unbounded queueing delays for other queued jobs, and therefore prevents the system from making definite predictions as to when each job will run. We therefore compare it with an alternative conservative approach, in which short jobs are moved ahead only if they do not delay any job in the queue. It turns out that for the workloads measured on SP2 systems, the original EASY algorithm provides better performance, so the added predictability of the conservative approach would come at a cost. However, using workloads from other systems, we find that both algorithms have about the same performance. In this case the conservative algorithm is preferable to the EASY algorithm, due to its improved predictability.

The main problem with backfilling is that it requires estimates of job runtimes to be available. In order to check the sensitivity to the accuracy of estimates, we investigate the accuracy of real estimates and their effect on performance. The surprising results are one, that user estimates are extremely unreliable, and two, that exaggerated estimates actually

lead to better performance than tight estimates! We conclude the paper by considering ways in which these new insights can be put to use in order to improve the scheduling of parallel supercomputers.

2 Backfilling Algorithms

Backfilling is an optimization in the framework of variable partitioning. In variable partitioning, users define the number of processors required for each job, and this number does not change during the execution; thus jobs can be described as requiring a rectangle in processor/time space (we will always draw time on the horizontal axis, and processors on the vertical axis). The jobs then run on dedicated partitions of the requested size. The name "variable partitioning" reflects the fact that the partitions are created in different sizes as needed.

With backfilling, users also provide an estimate of the runtime. This enables the scheduler to predict when jobs will terminate, and thus when the next queued jobs will be able to run. In particular, it is possible to identify "holes" in the schedule, and small jobs that can fit into these holes. This is the essence of backfilling.

It is desirable that a scheduler with backfilling will support two conflicting goals: to move as many short jobs forward as possible, in order to improve utilization and responsiveness, and to avoid starvation for large jobs, and in particular, to be able to predict when each job will run. Different versions of backfilling balance these goals in different ways.

2.1 Conservative Backfilling

Conservative backfilling is the vanilla version usually assumed in the literature (e.g. [10, 6]), although it seems not to be used. In this version, backfilling is done subject to checking that it does not delay *any* previous job in the queue. We call this version "conservative" backfilling to distinguish it from the more aggressive version used by EASY, as described below. Its advantage is that it allows scheduling decisions to be made upon job submittal, and thus has the capability of predicting when each job will run and giving users execution guarantees. Users can then plan ahead based on these guaranteed response times. Obviously there is no danger of starvation, as a reservation is made for each job when it is submitted.

In order to perform allocations, conservative backfilling maintains two data structures. One is the list of queued jobs and the times at which they are expected to start execution. The other is a profile of the expected processor usage at future times. When a new job arrives, the following allocation procedure is executed:

Algorithm conservative backfill:

1. Find anchor point:
   (a) Scan the profile and find the first point where enough processors are available to run this job. This is called the anchor point
   (b) Starting from this point, continue scanning the profile to ascertain that the processors remain available until the job's expected termination

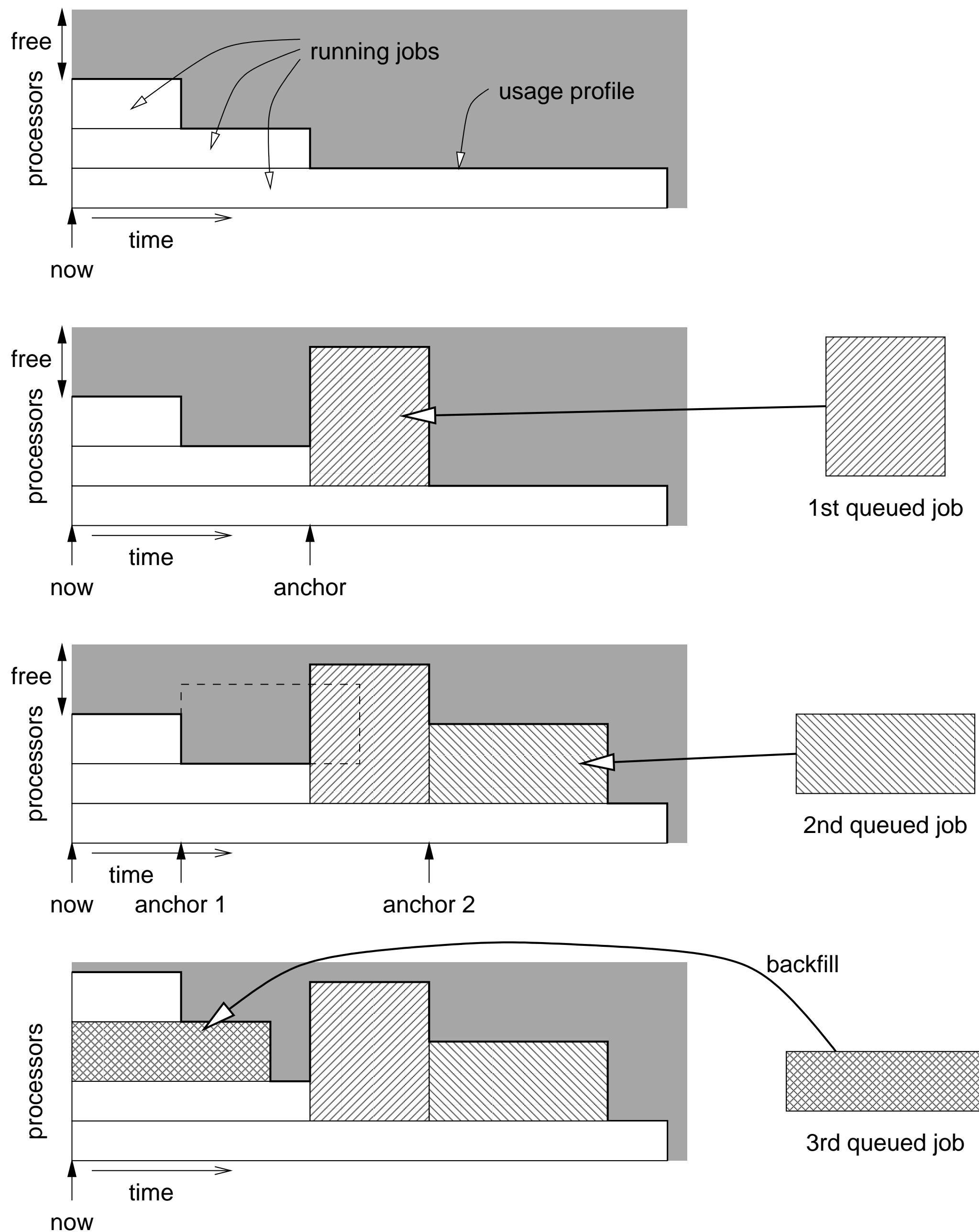

Figure 1: *Example of conservative backfilling.*

(c) If not, return to (a) and continue the scan to find the next possible anchor point

2. Update the profile to reflect the allocation of processors to this job, starting from its anchor point
3. If the job's anchor is the current time, start it immediately

An example is given in Fig. 1. The first job in the queue does not have enough processors to run, so a reservation for it is made after the first two running jobs terminate. The second

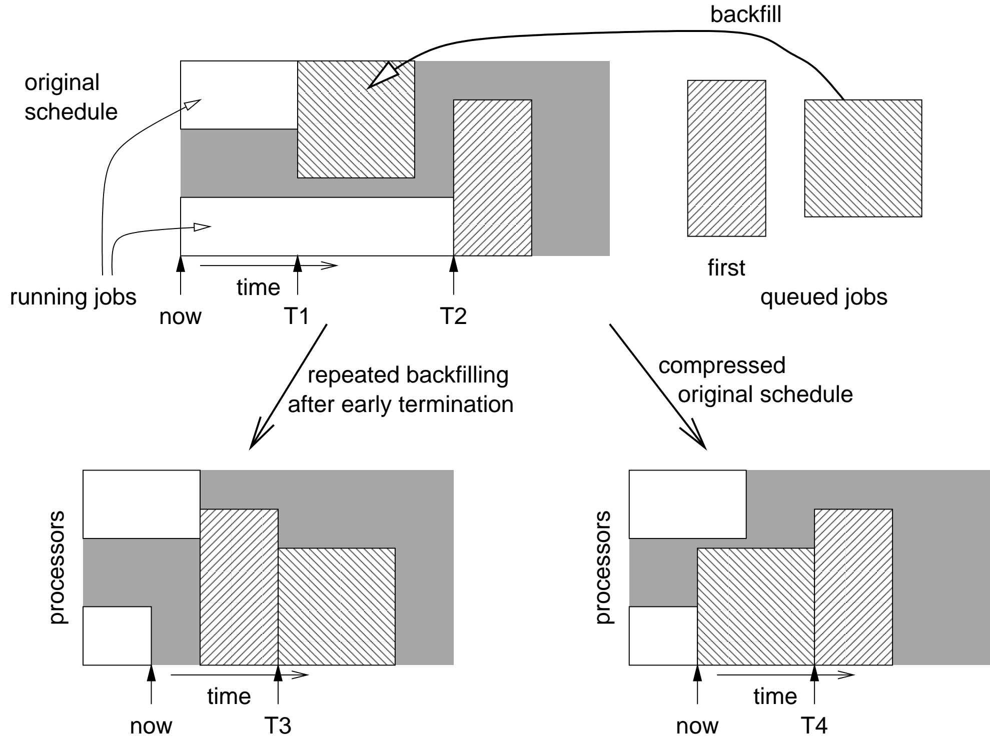

Figure 2: *Repeated backfilling after a running job terminates earlier than expected may cause a job that was expected to backfill to actually run later than the original prediction. It is therefore better to just compress the original schedule.*

queued job has a potential anchor point after only one job terminates, but that would delay the first job; therefore the second anchor point is preferred. Thus adding job reservations to the profile is the mechanism that guarantees that future arrivals do not delay previously queued jobs. The third job can be scheduled immediately, so it is used for backfilling.

It is most convenient to maintain the profile in a linked list, as it may be necessary to split items into two when a newly scheduled job is expected to terminate in the middle of a given period. In addition, an item may have to be added at the end of the profile whenever a job extends beyond the current end of the profile. The length of the profile is therefore proportional to the number of jobs in the system (both queued and running), because each job adds at most one item to the profile. As the profile is scanned once for each new job, the complexity of the algorithm is linear in the number of jobs.

The above algorithm leaves one question unanswered. Jobs are assigned a start time when they are submitted, based on the current usage profile, and the system guarantees that they will start by this time at the latest. But they may actually be able to run sooner because previous jobs terminated earlier than expected, leaving a gap in the planned schedule.

Given such a gap, one may decide to re-schedule all the jobs. However, this may violate the system's execution guarantees. In some cases, this guaranteed time will be the result of backfilling with this job. If a new round of backfilling is done later, *with different data about job runtimes due to an early termination*, the same job may not be backfilled and will therefore run much later than the guaranteed time. An example is given in Fig. 2: according

to the original schedule, the second queued job can backfill and start at time $T1$, but after the bottom running job terminates much earlier than expected, the first queued job can start earlier too, leaving no space for backfilling. The second queued job therefore has to start at the later time $T3$.

The preferred choice is therefore to compress the existing schedule. To do so, each job is removed from the profile, and then re-inserted at the earliest possible time. Jobs provably do not get delayed, because at worse each job will be re-inserted in the same position it held previously. The jobs can be considered in the order of arrival, so jobs that are waiting longer get a better chance to move forward. The complexity of compression is quadratic, because the profile is scanned again for each job.

2.2 EASY Backfilling

Conservative backfilling moves jobs forward only if they do not delay any previously queued job. EASY backfilling takes a more aggressive approach, and allows short jobs to skip ahead provided they do not delay *the job at the head of the queue* [17]. Interaction with other jobs is not checked, and they may be delayed, as shown below. The objective is to improve the current utilization as much as possible, subject to some consideration of queue order. The price is that execution guarantees cannot be made, because it is impossible to predict how much each job will be delayed in the queue. Thus the algorithm is actually not as deterministic as stated in its documentation.

The algorithm is as follows:

Algorithm EASY backfill:

1. Find the shadow time and extra nodes
   (a) Sort the list of running jobs according to their expected termination time
   (b) Loop over the list and collect nodes until the number of available nodes is sufficient for the first job in the queue
   (c) The time at which this happens is the shadow time
   (d) If at this time more nodes are available than needed by the first queued job, the ones left over are the extra nodes
2. Find a backfill job
   (a) Loop on the list of queued jobs in order of arrival
   (b) For each one, check whether either of the following conditions hold:
      i. It requires no more than the currently free nodes, and will terminate by the shadow time, or
      ii. It requires no more than the minimum of the currently free nodes and the extra nodes
   (c) The first such job can be used for backfilling

This is executed repeatedly whenever a new job arrives or a running job terminates, if the first job in the queue cannot start. In each iteration, the algorithm identifies a job that can backfill if one exists.

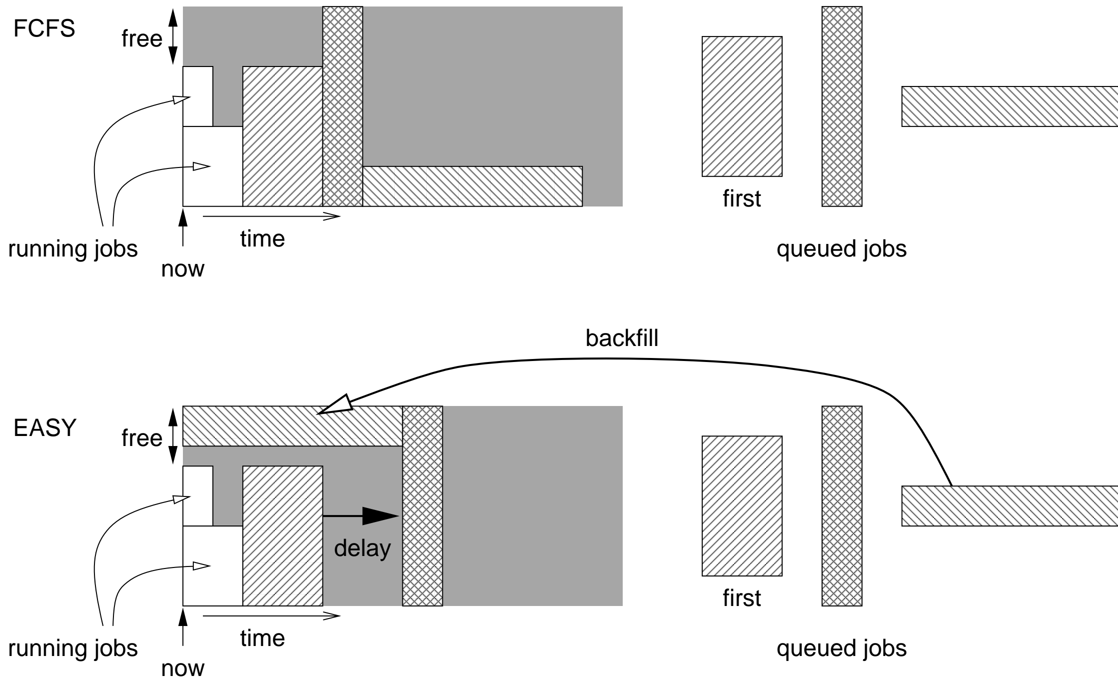

Figure 3: *In EASY, backfilling may delay queued jobs.*

This algorithm has two properties that together create an interesting combination.

Property 1 *Queued jobs may suffer an unbounded delay.*

Proof sketch: The reason for this is that if a job is not the first in the queue, new jobs that arrive later may skip it in the queue. While such jobs are guaranteed not to delay the first job in the queue, they may indeed delay all other jobs. This is the reason that the system cannot predict when a queued job will eventually run. An example is shown in Fig. 3: the backfill job does not delay the *first* job in the queue, but it does delay the *second* job. The length of the delay depends on the length of the backfill job, which in principle is unbounded. ∎

In practice, though, the job at the head of the queue only waits for currently running jobs, so if there is a limit on job runtimes then the bound on the queueing time is the product of this limit and the rank in the queue. But even without such a bound, we still have:

Property 2 *There is no starvation.*

Proof sketch: The queueing delay for the job at the head of the queue depends only on jobs that are already running, because backfilled jobs will not delay it. Thus it is guaranteed to eventually run (because the running jobs will either terminate or be killed when they exceed their declared runtime). Then the next job becomes first. This next job may have suffered various delays due to jobs backfilled earlier, but such delays stop accumulating once it becomes first. Thus it too is guaranteed to eventually run. The same arguments show that every job in the queue will eventually run. ∎

As noted, EASY sacrifices predictability for potentially improved utilization, by using more aggressive backfilling. However, it is not clear that increasing the *momentary* utilization at a given instant also contributes to the *overall* utilization over a long time, and counter examples can be constructed. Therefore detailed simulations are required to evaluate the real contribution of this approach. The results of such simulations are presented next.

3 Experimental Results

3.1 Methodology

The experiments are based on an event-based simulation, where events are job arrival and termination. Upon arrival, the scheduler is informed of the number of processors the job needs, and its estimated runtime. It can then either start the job's simulated execution, or place it in a queue. Upon a job termination, the scheduler is notified and can schedule other queued jobs on the freed processors. The runtime of jobs is part of the input to the simulation, but is not given to the scheduler. It is assumed that the runtime does not depend in any way on scheduling decisions.

The workloads used to drive the simulations were the following:

- Traces of the jobs submitted to the following supercomputers:

  CTC : The Cornell theory Center 512-node IBM SP2 (79296 jobs from July 1996 to May 1997)

  KTH : The Swedish Royal Institute of Technology 100-node IBM SP2 (28490 jobs from October 1996 to August 1997)

  SDSC : The San-Diego Supercomputer Center 128-node IBM SP2 (67665 jobs from April 1998 to April 2000)

  Par : The San-Diego Supercomputer Center 416-node Intel Paragon (115595 jobs from January 1995 to December 1996)

  CM5 : The Los Alamos National Lab 1024-node Connection Machine CM-5 (201387 jobs from October 1994 to September 1996)

- Workload models developed based on these and other traces:

  Feitelson : a general model based on data from 6 different traces, including CTC and Par above [4] (350000 jobs)

  Jann : a model developed specifically for the CTC trace [14] (100000 jobs)

All these workloads are available on-line from the Parallel Workloads Archive [22]. Only the first three logs contain actual user estimates of runtime. In other cases, accurate estimates are assumed (that is, the actual runtime is used for the estimate).

Traces are simulated using the exact data provided, with possible modifications as noted (e.g. to check the impact of different estimates of runtime). For models, the load on the simulated system is modified by multiplying the interarrival times by a certain factor. For

example, if by default the model produces a load of 0.688, we can create a higher load of 0.8 by multiplying all interarrival times by a factor of $\frac{0.688}{0.8} = 0.86$. Using different factors enables the functional relationship of performance on load to be measured.

The performance metrics used are the average response time and the average bounded slowdown. Slowdown is response time normalized by running time. Bounded slowdown eliminates the emphasis on very short jobs due to having the running time in the denominator [9]; a threshold of 10 seconds was used. For the record, the equation is

$$b_sld = \begin{cases} \dfrac{T_w + T_r}{T_r} & \text{if } T_r > 10 \\[2ex] \dfrac{T_w + T_r}{10} & \text{otherwise} \end{cases}$$

where *b_sld* is the bounded slowdown, T_r is the job's runtime on a dedicated system, and T_w is the job's waiting time. We also collected data on the waiting time; the results were similar.

When using models, 90% confidence intervals for the response time were calculated using the batch means method [13]. Each batch size was 3333 job terminations, with the first batch discarded to account for warmup effects (for the Jann model, batches were just under 1000 jobs). The simulation continued until any of the following three conditions was met: 100 batches were completed, or the confidence interval was smaller than 5% of the mean, or the mean response time exceeded a certain high threshold (30000 seconds, determined experimentally to be where it starts to shoot up). In practice, it turned out that most of the simulations took 100 batches and achieved an accuracy of about 6–9%.

3.2 The Results

The results of simulations using the two models are presented in Fig. 4. They indicate that the relative performance of EASY and conservative backfilling depends on the workload used and on the performance metric! Specifically, according to the Feitelson model (F), both schemes are practically identical. According to the Jann model (J), EASY has better (lower) average response times under high loads, but slightly worse (higher) bounded slowdown.

The results for the actual workload traces are reported for each month individually, so as to create multiple data points for somewhat different load conditions. They are shown in Table 1. Again, there is a difference between the different workloads and metrics. In general, the SP2 workloads favor the EASY backfilling over conservative backfilling. The only case in which conservative is a possible contender is when using the bounded slowdown metric and the KTH trace.

The non-SP2 traces seem to also favor EASY backfilling when measured by the response-time metric, but not for the bounded slowdown metric. Using the Par trace leads to inconclusive results for this metric. With the CM-5 trace, there seems to be a clear preference for conservative backfilling.

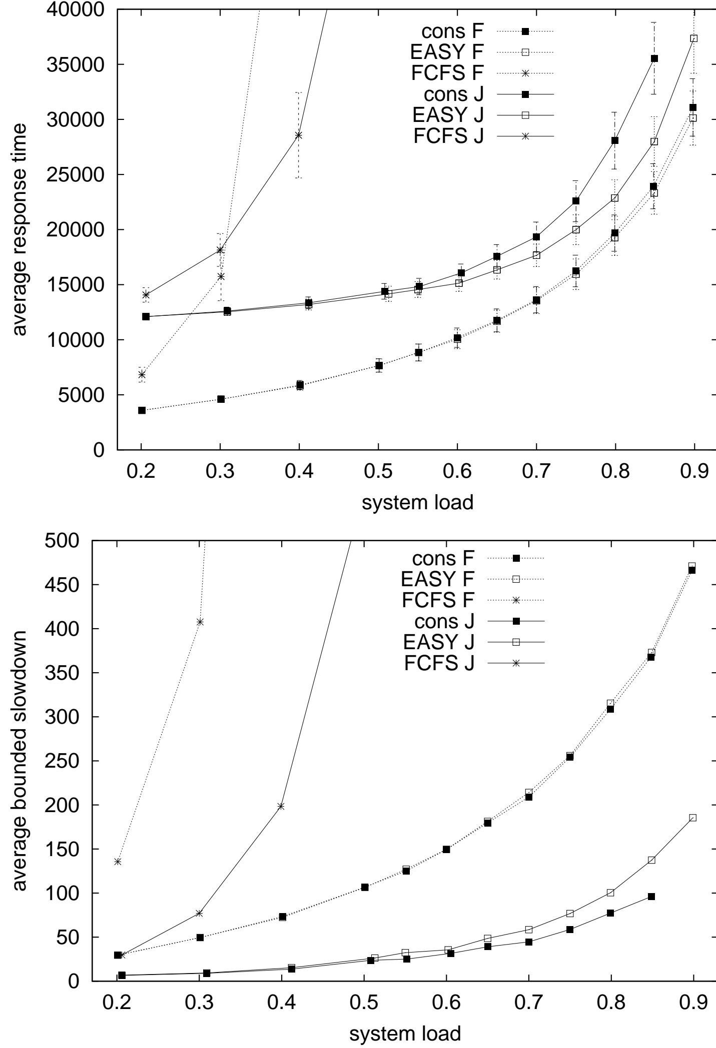

Figure 4: *Comparison of conservative backfilling and EASY backfilling using two workload models.*

3.3 Discussion

To summarize, the simulation results are somewhat inconclusive, and depend on the workload and metric being used. For most of the combinations checked, the performance of the EASY backfilling algorithm was better than that of conservative backfilling. However, in some cases the two algorithms seemed to provide similar performance, and in one case conservative was better than EASY[1].

To understand the differences in performance, it is instructive to study the amount of backfilling performed (Fig. 5). According to the Feitelson model, both do the same amount of backfilling, which matches the prediction of equal performance according to this model. Using the Jann model, we find that EASY backfills a slightly larger percentage of the jobs

[1]By chance, the preliminary version of this paper used the combinations that predict equal performance [7].

trace	mon	load	jobs	response time			bounded slowdown		
				EASY	cons	difference	EASY	cons	difference
CTC	jul	0.539	7950	11394	11605	+1.9%	4.9	5.5	+12.2%
	aug	0.584	7273	11558	11728	+1.5%	3.3	4.6	+39.4%
	sep	0.566	6167	14950	15360	+2.7%	5.8	5.9	+1.7%
	oct	0.547	7257	9963	10298	+3.4%	3.0	3.6	+20.0%
	nov	0.531	7917	10621	10684	+0.6%	2.0	2.2	+10.0%
	dec	0.514	7896	9173	9445	+3.0%	2.5	4.5	+80.0%
	jan	0.588	7519	10921	11214	+2.7%	3.3	4.1	+24.2%
	feb	0.580	8189	12669	12911	+1.9%	3.5	4.4	+25.7%
	mar	0.591	6915	15646	16766	+7.2%	4.5	6.6	+46.7%
	apr	0.577	6124	12642	13632	+7.8%	5.3	6.1	+15.1%
	may	0.555	6082	14512	15506	+6.8%	5.0	8.8	+76.0%
KTH	oct	0.669	2377	13375	12243	-8.5%	103.4	77.6	-25.0%
	nov	0.689	2006	18854	18978	+0.7%	152.9	151.6	-0.9%
	dec	0.689	2313	16694	19209	+15.1%	87.1	125.9	+44.5%
	jan	0.758	2917	15924	17436	+9.5%	95.4	95.7	+0.3%
	feb	0.798	2942	16959	18534	+9.3%	119.9	115.5	-3.7%
	mar	0.724	2074	18333	17934	-2.2%	110.7	131.2	+18.5%
	apr	0.720	2853	14825	17260	+16.4%	60.7	105.4	+73.6%
	may	0.678	4066	11055	11179	+1.1%	77.3	69.0	-10.7%
	jun	0.743	2715	14789	14782	-0.0%	33.6	31.4	-6.5%
	jul	0.620	2180	17996	18226	+1.3%	35.9	36.1	+0.6%
SDSC	may	0.621	2755	12711	13189	+3.8%	23.1	21.8	-5.6%
	jun	0.733	2478	12713	13243	+4.2%	13.9	14.2	+2.2%
	jul	0.749	2813	13851	14683	+6.0%	23.0	43.5	+89.1%
	aug	0.858	3540	23243	26087	+12.2%	34.3	36.4	+6.1%
	sep	0.712	12646	7197	7738	+7.5%	20.4	21.1	+3.4%
	oct	0.870	4534	23146	22077	-4.6%	82.3	79.1	-3.9%
	nov	0.678	3103	10927	12309	+12.6%	33.9	48.0	+41.6%
	dec	0.765	2896	17884	19080	+6.7%	45.2	45.5	+0.7%
	jan	0.829	2791	22374	23553	+5.3%	75.1	78.8	+4.9%
	feb	0.878	2703	26671	34586	+29.7%	119.9	181.6	+51.5%
	mar	0.830	2946	27144	32519	+19.8%	117.5	115.0	-2.1%
	apr	0.861	3684	20486	22027	+7.5%	94.9	78.7	-17.1%
	may	0.875	2535	33708	42438	+25.9%	121.5	151.3	+24.5%
	jun	0.854	2469	45360	57052	+25.8%	137.4	152.8	+11.2%
	jul	0.912	1265	55977	86264	+54.1%	206.0	259.4	+25.9%
	aug	0.909	1902	46507	64178	+38.0%	238.5	295.5	+23.9%
	sep	0.890	2162	38132	52325	+37.2%	98.6	111.1	+12.7%
	oct	0.872	1950	36544	42882	+17.3%	137.2	158.3	+15.4%
	nov	0.926	1988	48851	64493	+32.0%	271.2	339.8	+25.3%
	dec	0.855	1733	35331	46306	+31.1%	215.7	237.4	+10.1%
	jan	0.907	1499	38679	48489	+25.4%	103.8	134.0	+29.1%
	feb	0.920	1128	59197	75950	+28.3%	81.2	116.1	+43.0%
	mar	0.854	1199	44866	47946	+6.9%	139.7	98.8	-29.3%
	apr	0.858	946	48279	54548	+13.0%	131.6	156.5	+18.9%

Table 1: **(a)***Simulation results for the three IBM SP2 trace files. Differences denote the change when switching from EASY to conservative.*

than conservative backfilling. However, the simulations based on the traces suggest that the amount of backfilling performed is similar, and in one case (SDSC), conservative even performs *more* backfilling but achieves *worse* results. Thus it is not a question of how much backfilling is done, but more of which jobs are backfilled.

We are therefore left with a unique situation in which the workloads dictate the results (the only previous study to systematically check the influence of the workload concluded that

trace	mon	load	jobs	response time EASY	cons	difference	bounded slowdown EASY	cons	difference
Par	jan	0.547	5289	6844	7115	+4.0%	78.8	88.9	+12.8%
	feb	0.563	4809	8113	8353	+3.0%	60.4	63.9	+5.8%
	mar	0.686	5084	8361	9075	+8.5%	92.9	103.4	+11.3%
	apr	0.604	10685	4120	3910	-5.1%	64.7	43.1	-33.4%
	may	0.736	7251	9637	9322	-3.3%	146.3	105.5	-27.9%
	jun	0.573	6043	6752	6720	-0.5%	72.4	75.3	+3.9%
	jul	0.626	4875	6338	6785	+7.1%	38.5	52.9	+37.6%
	aug	0.602	3072	7349	7578	+3.1%	20.6	24.8	+20.7%
	sep	0.676	3300	7169	7210	+0.6%	28.0	28.0	+0.1%
	oct	0.590	6038	3054	3167	+3.7%	26.5	32.6	+22.9%
	nov	0.720	12116	3328	3359	+0.9%	95.7	101.1	+5.6%
	dec	0.595	7495	3244	3207	-1.1%	56.5	53.6	-5.0%
	jan	0.679	2856	8848	8940	+1.0%	43.2	39.5	-8.6%
	feb	0.678	5312	7382	7586	+2.8%	154.9	170.4	+10.0%
	mar	0.634	3781	10255	10964	+6.9%	35.0	44.8	+28.1%
	apr	0.764	4115	13059	10890	-16.6%	326.7	145.2	-55.6%
	may	0.742	3255	9328	9489	+1.7%	7.9	8.7	+10.3%
	jun	0.701	3824	14580	14045	-3.7%	126.2	56.5	-55.2%
	jul	0.658	2562	12558	12478	-0.6%	23.2	20.7	-10.7%
	aug	0.578	2542	7698	7908	+2.7%	8.7	17.5	+102.7%
	sep	0.570	2050	10568	10693	+1.2%	41.0	36.2	-11.7%
	oct	0.537	2670	10051	10030	-0.2%	19.7	15.2	-22.7%
	nov	0.429	2831	7978	7964	-0.2%	1.9	1.5	-19.9%
	dec	0.430	2123	13078	13108	+0.2%	1.5	1.8	+21.1%
CM5	oct	0.686	5746	5564	5543	-0.4%	30.7	27.2	-11.2%
	nov	0.884	6069	20270	20519	+1.2%	102.6	84.3	-17.8%
	dec	0.700	4702	7856	8011	+2.0%	32.9	31.5	-4.3%
	jan	0.647	3323	7433	7811	+5.1%	24.2	25.9	+7.2%
	feb	0.711	4413	8211	8295	+1.0%	28.8	27.3	-5.1%
	mar	0.673	4754	5643	5577	-1.2%	24.7	20.3	-17.9%
	apr	0.782	4747	7894	8023	+1.6%	34.6	26.8	-22.7%
	may	0.779	4717	23954	22673	-5.3%	148.2	125.5	-15.3%
	jun	0.886	5608	18093	18735	+3.5%	124.5	107.3	-13.8%
	jul	0.902	6457	17679	21747	+23.0%	135.8	112.4	-17.2%
	aug	0.730	6181	6284	6160	-2.0%	101.6	44.3	-56.4%
	sep	0.802	5678	17678	17467	-1.2%	110.0	89.9	-18.2%
	oct	0.788	5087	6109	6136	+0.4%	39.2	31.1	-20.6%
	nov	0.818	3706	8411	8919	+6.0%	28.2	26.9	-4.9%
	dec	0.703	4003	6274	6457	+2.9%	22.5	25.0	+11.4%
	jan	0.601	4110	4169	4217	+1.2%	16.1	14.2	-12.0%
	feb	0.461	3842	3358	3406	+1.4%	9.6	8.5	-10.9%
	mar	0.715	4111	5763	5758	-0.1%	20.1	18.9	-6.3%
	apr	0.788	4300	5616	6184	+10.1%	28.9	36.0	+24.7%
	may	0.807	4832	9030	10038	+11.2%	54.7	67.3	+22.9%

Table 1: **(b)***Simulation results for the non-SP2 trace files.*

workloads affect the quantitative results, but not the qualitative results [18]). The problem is that these workloads are rather complex, and it is not clear exactly what features are the decisive ones. We therefore turn to Talby et al. [26], who made a detailed statistical comparison of workloads and models. That work indicates that the CTC and KTH traces and the Jann model are indeed similar to each other, and distinct from other workloads such as the CM5 and Par traces and the Feitelson model (the SDSC trace was not included in the Talby paper). Specifically, the SP2 workloads seem to have higher than average runtimes and lower than average degrees of parallelism. This also matches the contradictory findings

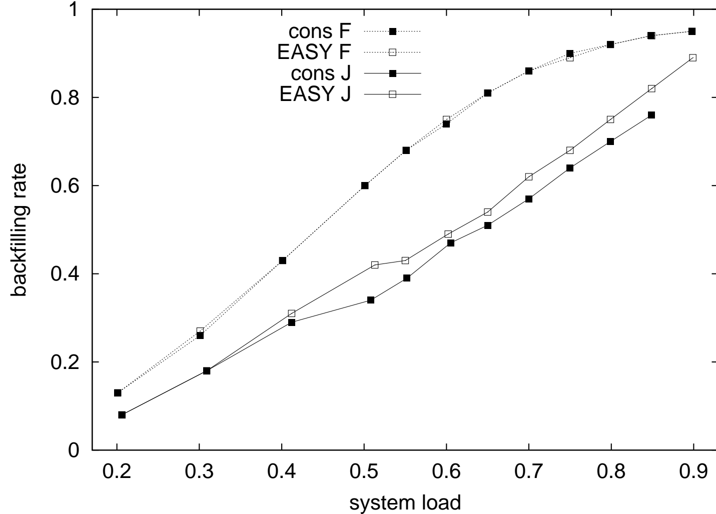

Figure 5: *The amount of backfilling done by the two schemes.*

according to the Jann model, which indicate that EASY is selective with respect to job size.

We have verified the observations regarding the differences among the workloads by plotting the cumulative distributions of runtimes for different job sizes for all the different traces and models. Fig. 6 shows a subset, including the comparison of the Feitelson and Jann models with the CTC trace. We next tried to verify whether these characteristics of the workloads are indeed responsible for the distinct behavior of the backfilling algorithms. To do so, we modified the Feitelson model so that the distributions of runtimes will mimic those of the CTC trace. This included two distinct modifications: changing the distribution of job sizes to emphasize small jobs (denoted by Fs), and changing the distribution of runtimes to emphasize longer jobs (Fl). The combination of these modifications (Fsl) leads to distributions that are very close to both the CTC trace and the Jann model (Fig. 6).

The simulation results were that indeed both the modifications are needed (see the Fs, Fl, and Fsl graphs in Fig. 7). The modifications to the runtime distribution alone made a small difference to the response time measurements. Adding the modifications to the size distribution enlarged the difference considerably. The modifications to the size distribution alone were enough to make a difference to the bounded slowdown measurements. However, the differences between the EASY and conservative schedulers on the modified Feitelson model were still smaller than on the Jann model. It therefore seems that there are some other workload differences at play as well. We checked and refuted two additional candidates: the distribution of interarrival times, which turned out to be very similar for the two models, and the feature of repetitive execution of jobs that is present only in the Feitelson model.

4 User Estimates of Runtime

The concept of backfilling is based on estimates of job runtimes. It has been assumed that users would be motivated to provide accurate estimates, because jobs would run faster if

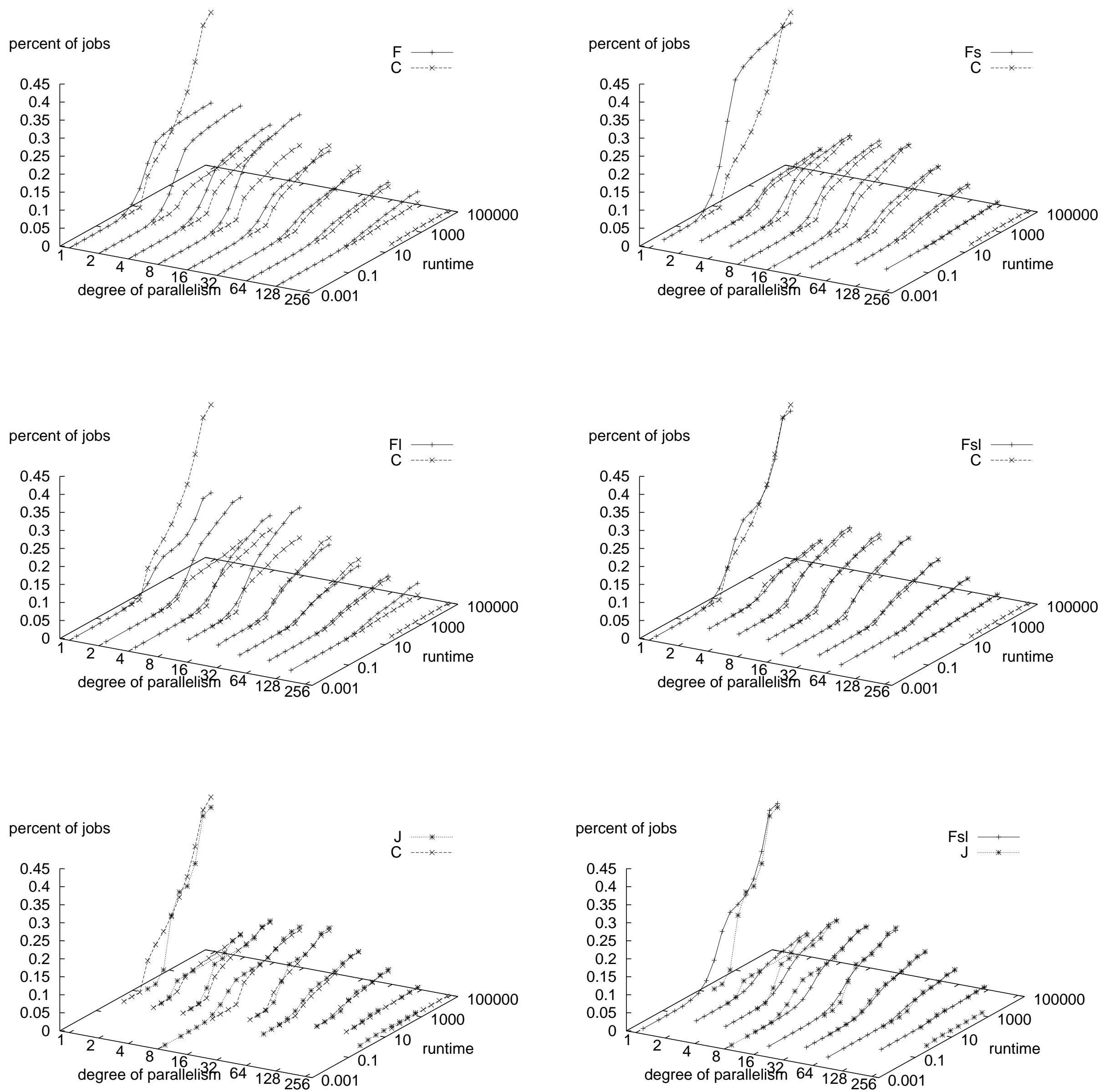

Figure 6: *Comparisons of cumulative runtime distributions of jobs with different sizes in the Feitelson, Jann, and CTC workloads.*

the estimates are tight, but would be killed if the estimates are too low. However, this assumption needs to be checked.

In order to study user runtime estimates we used workload data from the three IBM SP2 installations mentioned above. The workload data comes in the form of a log of all jobs executed on the machine during a certain period. The information on each job includes the estimated runtime provided by the user upon submittal, and the time the job actually ran.

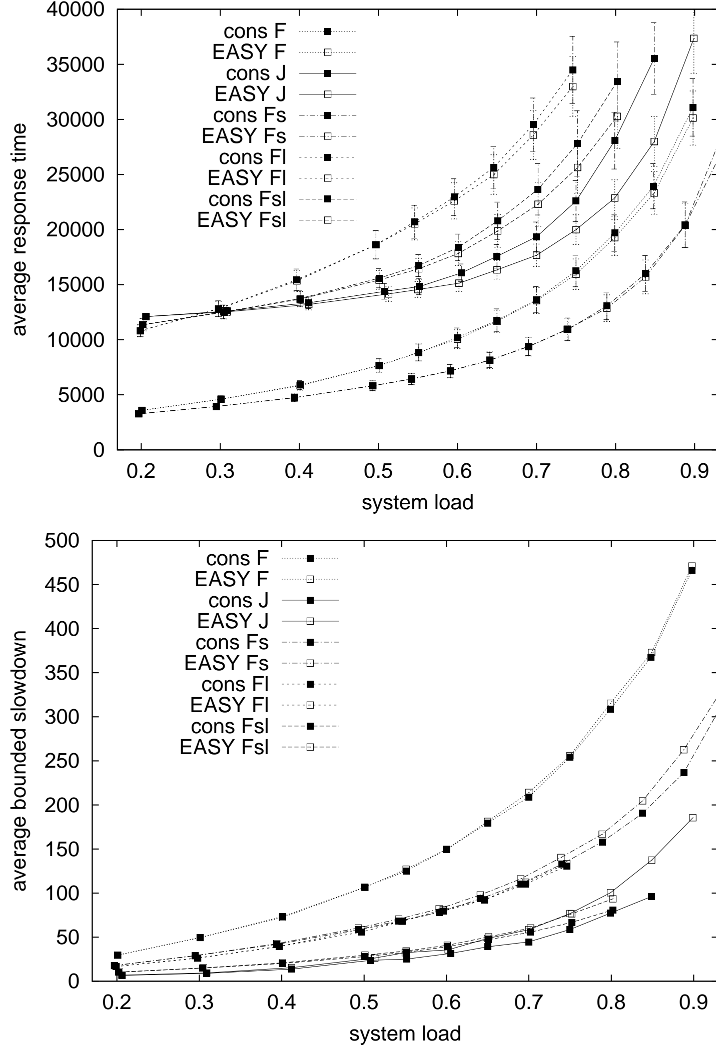

Figure 7: *Comparison of conservative backfilling and EASY backfilling for modified versions of the Feitelson model.*

The following is based on a job-by-job comparison of these two times.

4.1 The Quality of User Estimates

The results of the analysis are shown in Fig. 8, with CTC on the top, then KTH, and SDSC below. On the left is a histogram showing what percentage of the requested time was actually used. At first glance this seems promising, as it has a very pronounced component at exactly 100% (see Table 2 for exact numbers). However, closer inspection shows that practically all of the jobs in this peak actually reached their allocated time and were then killed by the system[2].

[2]Note that this is not necessarily bad: applications may checkpoint their state periodically, and then be restarted from the last checkpoint after being killed. However, there is no direct data about how often this

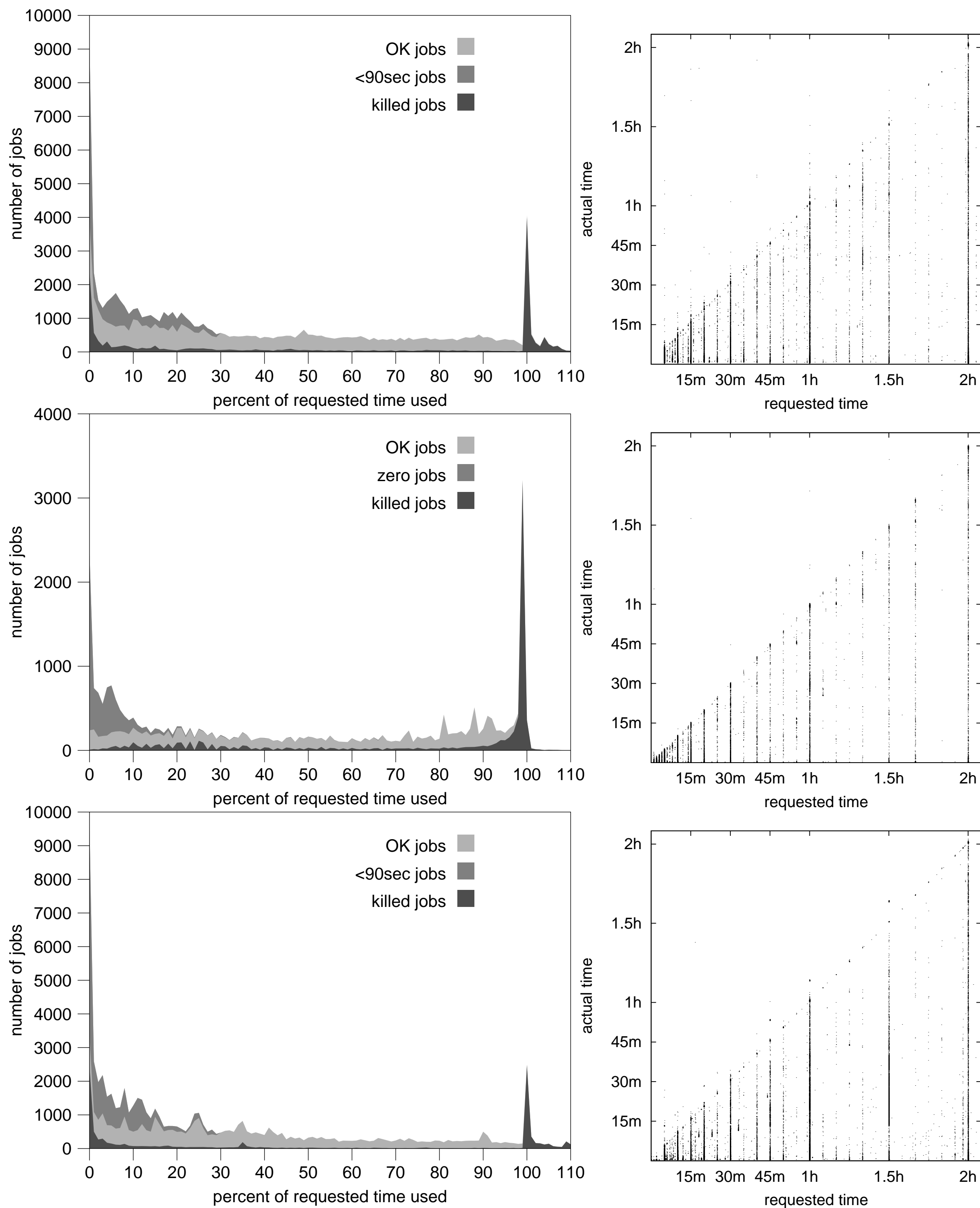

Figure 8: *User runtime estimates and actual runtimes, from CTC (top), KTH (middle), and SDSC (bottom). See explanations in the text.*

parameter	CTC number	CTC %	KTH number	KTH %	SDSC number	SDSC %	comment
total jobs	79296		28490		67665		
killed jobs	16671	21.0	7948	27.9	11175	16.5	% of total
100% peak	4806	6.1	3601	12.6	3006	4.4	% of total
killed	4547	94.6	3590	99.7	2855	95.0	% of 100% peak
0% peak	9027	11.4	2333	8.2	9553	14.1	% of total
zero jobs			6374	22.4			% of total
<90sec jobs	18589	23.4	9361	32.9	19719	29.1	% of total
<2hr jobs	35541	44.8	16128	56.6	43515	64.3	% of total

Table 2: *Parameters of the workloads and numerical values for components of the histograms in Fig. 8.*

The rest of the distribution is quite flat, but with somewhat higher values at low percentages, and another peak at zero, which is obviously bad. The CTC and SDSC data indicates that many of the jobs in the zero peak were killed, and the rest of the excess jobs at low percentages were very short (less than 90 seconds). The KTH data contains additional information: it shows that all the extra jobs at low percentages, including the zero peak, are what we call zero-length jobs. These are jobs in which the first node was deallocated before the last node was allocated, so there was no time at which all the nodes were being used simultaneously. This situation most probably indicates that the job failed immediately upon loading. We conjecture that the situation on the other two systems is similar. Thus the extra jobs at low percentages and the zero peak provide testimony about the difficulty of getting jobs to run, but do not say much about user estimates. On a related vein, about 8% of the jobs in the SDSC data were removed before they even started to run; these were not included in the analysis reported here.

Concentrating on the jobs that ran for 90 seconds or more and terminated normally, we find that the histogram is quite flat. The conclusion is that user estimates are actually rather poor. However, it should be noted that they do provide a good upper-bound on the running time (only a relatively small fraction of the jobs were killed because they exceeded their estimated time). The conclusion is that users find the motivation to overestimate so that jobs will not be killed much stronger than the motivation to provide accurate estimates to enable the scheduler to perform better packing.

The same data is shown again in the scatter plot on the right of the figure, which shows pairs of estimated runtime and the corresponding actual runtime (only jobs requesting up to 2 hours are shown, which is about half of the jobs — see "<2hr" line in Table 2). This shows that users often, but not always, round their estimates to a "nice" number (typically multiples of 5 minutes, or, for longer jobs, multiples of 10 or 30 minutes). However, despite the relatively wide repertoire of estimates that are used, all of them are equally inaccurate: for every popular estimate, there is a nearly continuous line of dots representing jobs with runtimes ranging uniformly from zero up to the estimate. The system typically kills jobs

is actually done. Indirect data from KTH is that 793 of the jobs killed by the system had requested 4 hours, which is the limit imposed during the daytime. As the peak at 100% contains 3215 jobs, this leads to a maximal estimate of about one job in four.

metric	trace	runtime estimates: original	uniform in $[r, f \cdot r]$: $f=1$	$f=2$	$f=4$	$f=11$	$f=31$	$f=101$	$f=301$
using EASY backfilling									
bounded slowdown	CTC	3.82	4.10	3.12	3.04	3.02	3.06	3.03	**3.01**
	KTH	84.0	67.6	67.0	**62.7**	63.7	64.7	64.9	65.8
	SDSC	84.2	**70.1**	72.7	76.3	76.3	78.9	82.0	83.9
response time	CTC	12053	12234	11976	11923	11896	11895	**11889**	11890
	KTH	15568	15001	14717	**14645**	14880	15028	15110	15127
	SDSC	24519	21976	**21801**	22451	23148	23977	24739	24978
using conservative backfilling									
bounded slowdown	CTC	5.00	3.71	2.62	2.39	2.38	**2.37**	2.40	2.37
	KTH	89.7	68.7	50.0	49.3	47.5	**47.4**	49.4	49.8
	SDSC	96.0	68.3	**56.0**	58.9	63.9	63.3	67.8	67.3
response time	CTC	12495	12639	12201	12062	11983	11965	**11964**	11964
	KTH	16288	16098	14940	**14878**	15095	15391	15538	15651
	SDSC	29422	23239	**21550**	22800	25220	29284	32999	32862

Table 3: *The effect of user estimate quality on performance.*

that do not terminate by the estimated time, leading to the triangular shape of the scatter plot.

4.2 Are Good Estimates Really Needed?

In order to check the sensitivity of the backfilling algorithms to such poor estimates, we tested them with estimates of various qualities. Using the three workload files, we generated new user estimates that (for each job) are chosen at random from a uniform distribution in the range $[r, f \cdot r]$, where r is the job's actual runtime, and f is a "badness" factor: the larger f, the less accurate the estimates. $f = 1$ indicates completely accurate estimates. For each value of f, 10 measurements were made with different random number generator seeds. The same set of 10 seeds was used for the different traces and different fs.

The results are shown in Table 3, together with the results of using the original user estimates from the traces. Two conclusions can be reached:

- Accurate estimates are not necessarily the best. It seems that if the estimates are somewhat inaccurate, this gives the algorithms some flexibility that leads to better schedules. This result has since been corroborated by Zotkin and Keleher [27].
- Our model of inaccuracy does not capture the full badness of real user estimates. The results for the original estimates are typically worse than those with our randomized estimates.

4.3 Modeling User Estimates of Runtime

The second conclusion motivated a search for a better model of the relationship between the actual runtime of jobs and the estimates produced by users. Such a model is needed for two reasons. First, it is useful as part of a general workload model that can be used to study different job scheduling schemes. For example, this would allow the simulations

reported in Section 3 to be repeated with realistic user estimates, rather than having to assume completely accurate estimates (which we now know probably lead to overly pessimistic performance results). Second, an accurate model is required in order to study whether and how the inaccuracy of user estimates can be exploited by the scheduler.

The proposed model is quite simple. The flat histogram of Fig. 8 implies that

$$T_r/T_e = u$$

i.e. that the ratio of the actual runtime to the estimate can be modeled as a uniformly distributed random variable. By changing sides we get

$$T_e = T_r/u$$

so given a runtime T_r we can generate an estimate T_e that, while unrelated to the actual user estimate for this particular job, is expected to lead to the same general statistics of all the estimates taken together. To complete the model we just need to note that in about 10% of the jobs the estimate is actually too small, and for short jobs the estimates are too large by a factor of about 10. The final model is therefore

1. With probability of 10% return $0.99 \times T_r$
2. Otherwise create an estimate of T_r/u, where u is uniform in the range $[0, 1]$.
3. If $T_r < 90$, multiply the estimate by 10.
4. If the estimate is outrageous, truncate it to some upper bound (e.g. 24 hours).

4.4 The Alternative: Estimates Based on Historical Information

It is well known that the workload on parallel supercomputers is highly repetitive. This means that the same users tend to run the same programs over and over again, sometimes up to hundreds of executions in a row [8, 3]. It stands to reason that such repeated executions of the same application would have highly correlated runtimes, and indeed several studies have shown that it is possible to derive crude estimates of runtimes using such information [10, 2, 24]. However, these studies were done in a context that does not penalize underestimation, as is the case with backfilling (where jobs that overrun their estimated time are killed). In this context, an estimation method that tends to overestimate is preferred, even if it is less accurate in absolute terms.

To estimate runtimes based on historical information one must first be able to identify repeated executions. For this purpose, we use the combination of application (that is, executable filename), user, and number of nodes used as an identifier [8, 10]. The estimate is then calculated as the average of previous runs, plus $1\frac{1}{2}$ times their standard deviation. Note that this can be done based on storing only three numbers: the number of previous executions, the sum of their runtimes, and the sum of their runtimes squared. If no specific previous information is available, data for the whole workload is used as a conservative upper bound. Finally, in order to avoid stale data, we discard historical information if it is more than a week old and start from scratch.

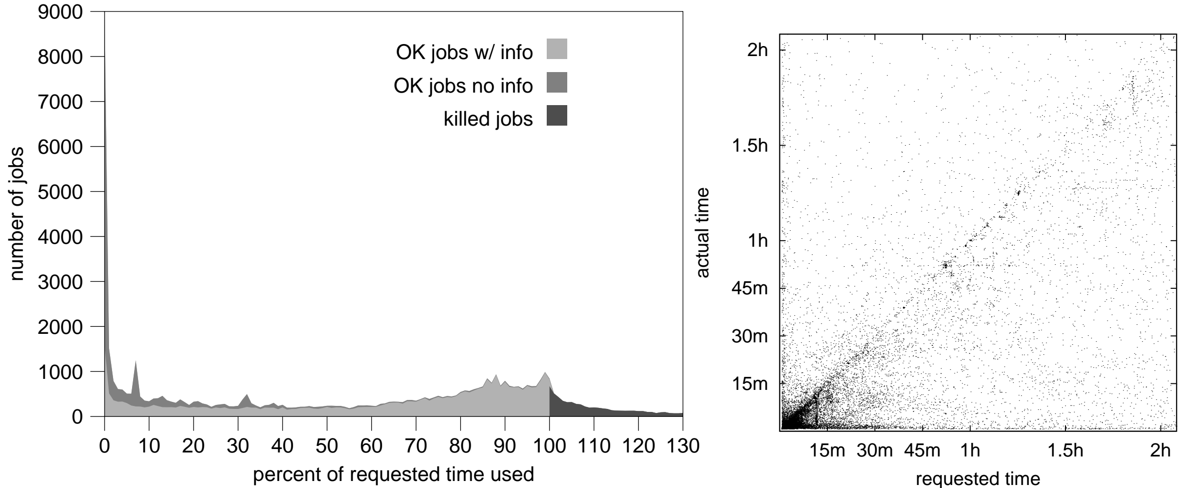

Figure 9: *Quality of system-generated estimates of runtime.*

To evaluate the effectiveness of this approach we used it to estimate the runtimes of all the jobs that were not killed in the CTC workload, and compared the estimates to the actual runtimes as we did for the actual user estimates in Fig. 8. Of the 62630 jobs, there were 45159 (72.1%) for which data was available. The resulting histogram and scatter plot are shown in Fig. 9, and indicate that the estimates have a better profile than those generated by users. However, 12001 jobs (19.2%) suffered from an underestimate, and would have been killed by the scheduler. About half of these (6240, or 10.0%) were jobs for which previous information was available.

It is easy to reduce the number of jobs that receive underestimates by using a more conservative approach, e.g. the average plus 3 standard deviations. However, this reduces the quality of the estimates and leads to a relatively flat histogram. Thus it seems that there is a tradeoff between accuracy and the danger of having jobs killed. In any case, given the large fraction of jobs that are underestimated, it seems that using system-generated estimates for backfilling is not a feasible approach.

4.5 Does it Help to Know that Estimates are Inaccurate?

Based on a repeated execution of experiments such as those described in Section 4.2, Zotkin and Keleher have proposed that the performance of backfilling schedulers can be improved by simply multiplying user estimates by a factor of 2 or more, thus creating looser estimates that give the scheduler more flexibility. However, as we noted above, real user estimates produce worse results than the results produced by accurate runtimes multiplied by a factor. Therefore it is not obvious that this scheme will work with real user estimates.

To evaluate how well this idea works, we simulated the execution of the three SP2 workloads under EASY backfilling and conservative backfilling, with both the original user estimates and these estimates multiplied by a factor of two. The results for average response time and average bounded slowdown are shown in Table 4. They indicate that in general multiplying the user estimates by two does indeed improve the performance. In the case of

metric	trace	EASY orig	EASY ×2	EASY diff	conservative orig	conservative ×2	conservative diff
bounded slowdown	KTH	84.0	80.0	-4.8%	89.7	69.1	-23.0%
	CTC	3.8	3.5	-7.9%	5.0	4.1	-18.0%
	SDSC	84.2	88.1	+4.6%	96.0	82.4	-14.2%
response time	KTH	15568	15060	-3.3%	16288	15147	-7.0%
	CTC	12053	11944	-0.9%	12495	12291	-1.6%
	SDSC	24519	24134	-1.6%	29422	26222	-10.9%

Table 4: *Effect of multiplying user estimates by two.*

conservative backfilling as measured by the bounded slowdown metric, the improvement is quite significant.

5 Conclusions

Backfilling is advantageous because it provides improved responsiveness for short jobs combined with no starvation for long ones. This is done by making processor reservations for the large jobs, and then allowing short jobs to leapfrog them if they are expected to terminate in time. The expected termination time is based on user input.

SP2 installations using EASY, which introduced backfilling, report much improved support for large jobs relative to early versions of LoadLeveler [19, 15]. However, EASY suffers from some uncertainty regarding the time at which a job will run, because of its aggressive backfilling algorithm. We showed that it is possible to add predictability by using a more conservative form of backfilling, in which short jobs can start running provided they do not delay any previously queued job.

The most interesting aspect of the performance evaluation of this idea is that the results depend on the workload and metric. Specifically, we found that when using workloads characteristic of SP2 sites, the use of conservative backfilling typically comes at the cost of degraded performance; this was not so pervasive for other workloads. This leads to the conjecture that the workload at the SP2 sites may have evolved to match the EASY backfilling algorithm used at these sites. A more detailed study of the workload attributes is now being conducted to try and verify this conjecture.

In addition, we showed that user estimates of runtime are quite bad, but that in fact this has the potential to be beneficial, because backfilling works better if it is allowed some flexibility. Even a simple approach of just multiplying user estimates by a constant leads to improvements. More sophisticated approaches, such as that recently proposed by Talby et al. [25], may be even better.

Acknowledgements

This research was supported by the Ministry of Science and Technology and by the Israel Science Foundation founded by the Israel Academy of Sciences and Humanities. The workload log from the CTC SP2 was graciously provided by the Cornell Theory Center, a high-performance computing center at Cornell University, Ithaca, New York, USA. The

workload log from the KTH SP2 was graciously provided by Lars Malinowsky, who also helped with background information and interpretation. The workload log from the SDSC SP2 was graciously provided by Victor Hazlewood of the HPC Systems group of the San Diego Supercomputer Center (SDSC), which is the leading-edge site of the National Partnership for Advanced Computational Infrastructure (NPACI), and is available from the NPACI JOBLOG repository at http://joblog.npaci.edu. The code for the Jann workload model was graciously provided by Joefon Jann of IBM Research.

References

[1] D. Das Sharma and D. K. Pradhan, "*Job scheduling in mesh multicomputers*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 251–258, Aug 1994.

[2] A. B. Downey, "*Using queue time predictions for processor allocation*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 35–57, Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[3] A. B. Downey and D. G. Feitelson, "*The elusive goal of workload characterization*". *Perf. Eval. Rev.* **26(4)**, pp. 14–29, Mar 1999.

[4] D. G. Feitelson, "*Packing schemes for gang scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 89–110, Springer-Verlag, 1996. Lect. Notes Comput. Sci. vol. 1162.

[5] D. G. Feitelson, *A Survey of Scheduling in Multiprogrammed Parallel Systems.* Research Report RC 19790 (87657), IBM T. J. Watson Research Center, Oct 1994.

[6] D. G. Feitelson and M. A. Jette, "*Improved utilization and responsiveness with gang scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 238–261, Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[7] D. G. Feitelson and A. Mu'alem Weil, "*Utilization and predictability in scheduling the IBM SP2 with backfilling*". In 12th *Intl. Parallel Processing Symp.*, pp. 542–546, Apr 1998.

[8] D. G. Feitelson and B. Nitzberg, "*Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 337–360, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[9] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong, "*Theory and practice in parallel job scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 1–34, Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[10] R. Gibbons, "*A historical application profiler for use by parallel schedulers*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 58–77, Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[11] S. Hotovy, "*Workload evolution on the Cornell Theory Center IBM SP2*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 27–40, Springer-Verlag, 1996. Lect. Notes Comput. Sci. vol. 1162.

[12] Intel Corp., *iPSC/860 Multi-User Accounting, Control, and Scheduling Utilities Manual.* Order number 312261-002, May 1992.

[13] R. Jain, *The Art of Computer Systems Performance Analysis.* John Wiley & Sons, 1991.

[14] J. Jann, P. Pattnaik, H. Franke, F. Wang, J. Skovira, and J. Riodan, "*Modeling of workload in MPPs*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 95–116, Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[15] J. P. Jones and B. Nitzberg, "*Scheduling for parallel supercomputing: a historical perspective of achievable utilization*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 1–16, Springer-Verlag, 1999. Lect. Notes Comput. Sci. vol. 1659.

[16] P. Krueger, T-H. Lai, and V. A. Dixit-Radiya, "*Job scheduling is more important than processor allocation for hypercube computers*". *IEEE Trans. Parallel & Distributed Syst.* **5(5)**, pp. 488–497, May 1994.

[17] D. Lifka, "*The ANL/IBM SP scheduling system*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 295–303, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[18] V. Lo, J. Mache, and K. Windisch, "*A comparative study of real workload traces and synthetic workload models for parallel job scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 25–46, Springer Verlag, 1998. Lect. Notes Comput. Sci. vol. 1459.

[19] L. Malinowsky and P. Öster, "*Scheduling of a parallel workload: implementation and use of the Argonne EASY scheduler at PDC*". In *Applied Parallel Computing*, B. Kågström, J. Dongarra, E. Elmroth, and J. Waśniewski (eds.), pp. 309–314, Springer-Verlag, 1998. Lect. Notes Comput. Sci. vol. 1541.

[20] C. McCann, R. Vaswani, and J. Zahorjan, "*A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors*". *ACM Trans. Comput. Syst.* **11(2)**, pp. 146–178, May 1993.

[21] P. Messina, "*The Concurrent Supercomputing Consortium: year 1*". *IEEE Parallel & Distributed Technology* **1(1)**, pp. 9–16, Feb 1993.

[22] *Parallel workloads archive*. URL http://www.cs.huji.ac.il/labs/parallel/workload/.

[23] J. Skovira, W. Chan, H. Zhou, and D. Lifka, "*The EASY - LoadLeveler API project*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 41–47, Springer-Verlag, 1996. Lect. Notes Comput. Sci. vol. 1162.

[24] W. Smith, I. Foster, and V. Taylor, "*Predicting application run times using historical information*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 122–142, Springer Verlag, 1998. Lect. Notes Comput. Sci. vol. 1459.

[25] D. Talby and D. G. Feitelson, "*Supporting priorities and improving utilization of the IBM SP scheduler using slack-based backfilling*". In 13th *Intl. Parallel Processing Symp.*, pp. 513–517, Apr 1999.

[26] D. Talby, D. G. Feitelson, and A. Raveh, "*Comparing logs and models of parallel workloads using the co-plot method*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 43–66, Springer Verlag, 1999. Lect. Notes Comput. Sci. vol. 1659.

[27] D. Zotkin and P. J. Keleher, "*Job-length estimation and performance in backfilling schedulers*". In 8th *High Performance Distributed Computing Conf.*, 1999.

Ahuva W. Mu'alem received the B.Sc. in Mathematics from the Technion, and the M.Sc. in Computer Science from the Hebrew University in 1999. She is currently a pursuing a Ph.D. in Computer Science at the Hebrew University. She is also a digital artist: see http://www.cs.huji.ac.il/˜ahumu.

Dror G. Feitelson is on the faculty of the School of Computer Science and Engineering at the Hebrew University of Jerusalem, Israel, where he conducts research on parallel operating systems. His recent focus is on the characterization and modeling of workloads on production parallel machines, with the goal of putting the evaluation of parallel job scheduling policies on a more solid basis. He is the co-organizer of an annual series of workshops on the topic of *Job Scheduling Strategies for Parallel Processing*, now in its seventh year.
He received a B.Sc. in Mathematics, Physics, and Computer Science in 1985, an M.Sc. in Computer Science in 1987 (cum laude), and a Ph.D. in 1991, all from the Hebrew University. He then spent 3 years at the IBM T. J. Watson Research Center, where he contributed to the system software of the SP1 and SP2. At Hebrew University, he has lead the design and implementation of the ParPar cluster system, and of the BoW on-line bibliographical repository.