Utilization and Predictability in Scheduling the IBM SP2 with Backfilling

Dror G. Feitelson Ahuva Mu'alem Weil
Institute of Computer Science
The Hebrew University of Jerusalem
91904 Jerusalem, Israel
{feit,ahumu}@cs.huji.ac.il

Abstract

Scheduling jobs on the IBM SP2 system 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 motivated Argonne National Lab, where the first large SP1 was installed, to develop the EASY scheduler. This scheduler, which has since been adopted by many other SP2 sites, 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 show that a more conservative approach, in which small jobs move ahead only if they do not delay *any* job in the queue, produces essentially the same benefits in terms of utilization. Our conservative scheme has the added advantage that queueing times can be predicted in advance, whereas in EASY the queueing time is unbounded.

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 [2]. Such partitions are allocated in a first-come first-serve (FCFS) manner to interactive jobs that are submitted directly, and to batch jobs that are submitted via a queueing system such as NQS. But this approach suffers from fragmentation, where processors cannot meet the requirements of the next queued job and therefore remain idle. As a result system utilization is typically in the range of 50–80% [12, 9, 4, 7].

It is well known that the best solutions for this problem are to use dynamic partitioning [11] or gang scheduling [3]. 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 implementations are too coarse-grained for real interactive support, and do not enjoy much use. Dynamic partitioning has not been implemented on production machines at all.

A simpler approach is to just re-order the jobs in the queue, that is, to use non-FCFS policies [5]. Consider the following scenario, where a number of jobs are running side by side, and the next queued job requires all the processors in the system. A 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 [8, 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. 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 Lab as part of EASY (the Extensible Argonne Scheduling sYstem) [10], which has since been integrated with the LoadLeveler scheduler from IBM for the SP2 [13].

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 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 then go on to show that an alternative approach, in which short jobs are moved ahead only if they do not delay any job in the queue, has essentially the same benefits as the more aggressive EASY algorithm. As this approach has the additional benefit of making an exact reservation for each job immediately when it is submitted, it is preferable to the EASY algorithm. The comparison of the algorithms is done both with a general workload model and with specific workload traces from SP2 installations.

2 Backfilling

Backfilling is an optimization in the framework of variable partitioning. In this framework, users define the number of processors required for each job and also provide an estimate of the runtime; thus jobs can be described as requiring a rectangle in processor/time space (Fig. 1). The jobs then run on dedicated partitions of the requested size. Note that users are motivated to provide an accurate estimate of the runtime, because lower estimates mean that the job may be able to run sooner, but if the estimate is too low the job will be killed when it overruns its allocation.

Once runtime estimates are available, it is possible to predict when jobs will terminate,

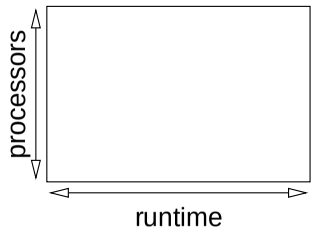

Figure 1: *Graphical representation of a job in processor/time space.*

and thus when the next queued jobs will be able to run. With FCFS scheduling, queueing time is estimated based on previous jobs in the queue. However, FCFS suffers from fragmentation and delays to short jobs that are stuck behind long ones. Backfilling improves upon this by moving short jobs ahead in the queue to utilize "holes" in the schedule. The name "backfilling" was coined by Lifka to describe the EASY scheduler for the Argonne SP1 [10], although the concept was also present in earlier systems (e.g. [8]).

It is desirable that a scheduler with backfilling will support two conflicting goals: on one hand, it is desirable to move as many short jobs forward, in order to improve utilization and responsiveness. On the other hand, it is also desirable 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. [6, 3]), 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.

It is easier to describe the algorithm to decide if a certain job can be used for backfilling as if it starts from scratch at each scheduling operation, with no information about prior commitments (Fig. 2). This algorithm creates a profile of free processors in future times as a linked list. Initially, this is a monotonically decreasing profile based on the currently running jobs (top of Fig. 3). Then the queued jobs are checked in order of arrival, to see if they can backfill and start execution immediately. However, jobs that cannot start immediately cannot be ignored. Rather, the profile is scanned to find when enough processors will be available for each queued job to start (this point in time is called the *anchor point* for that job). Then scanning is continued to see that the required processors will stay available till it terminates. If so, the job is assigned to this anchor point, and the profile is updated to reflect the processors allocated to it.

input:

- list of queued jobs with nodes and time requirements
- list of running jobs with node usage and expected termination times
- number of free nodes

algorithm conservative backfill from scratch:

1. generate processor usage profile of running jobs
   (a) sort the list of running jobs according to their expected termination time
   (b) loop over the list dividing the future into periods according to job terminations, and list the number of processors used in each period; this is the usage profile
2. try to backfill with queued jobs
   (a) loop on the list of queued jobs in order of arrival
   (b) for each one, scan the profile and find the first point where enough processors are available to run this job. this is called the anchor point
      i. starting from this point, continue scanning the profile to ascertain that the processors remain available until the job's expected termination
      ii. if so, update the profile to reflect the allocation of processors to this job
      iii. if not, continue the scan to find the next possible anchor point, and repeat the check
   (c) the first job found that can start immediately is used for backfilling

Figure 2: *The conservative backfilling algorithm, when run from scratch disregarding previous execution guarantees.*

An example is given in Fig. 3. 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 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

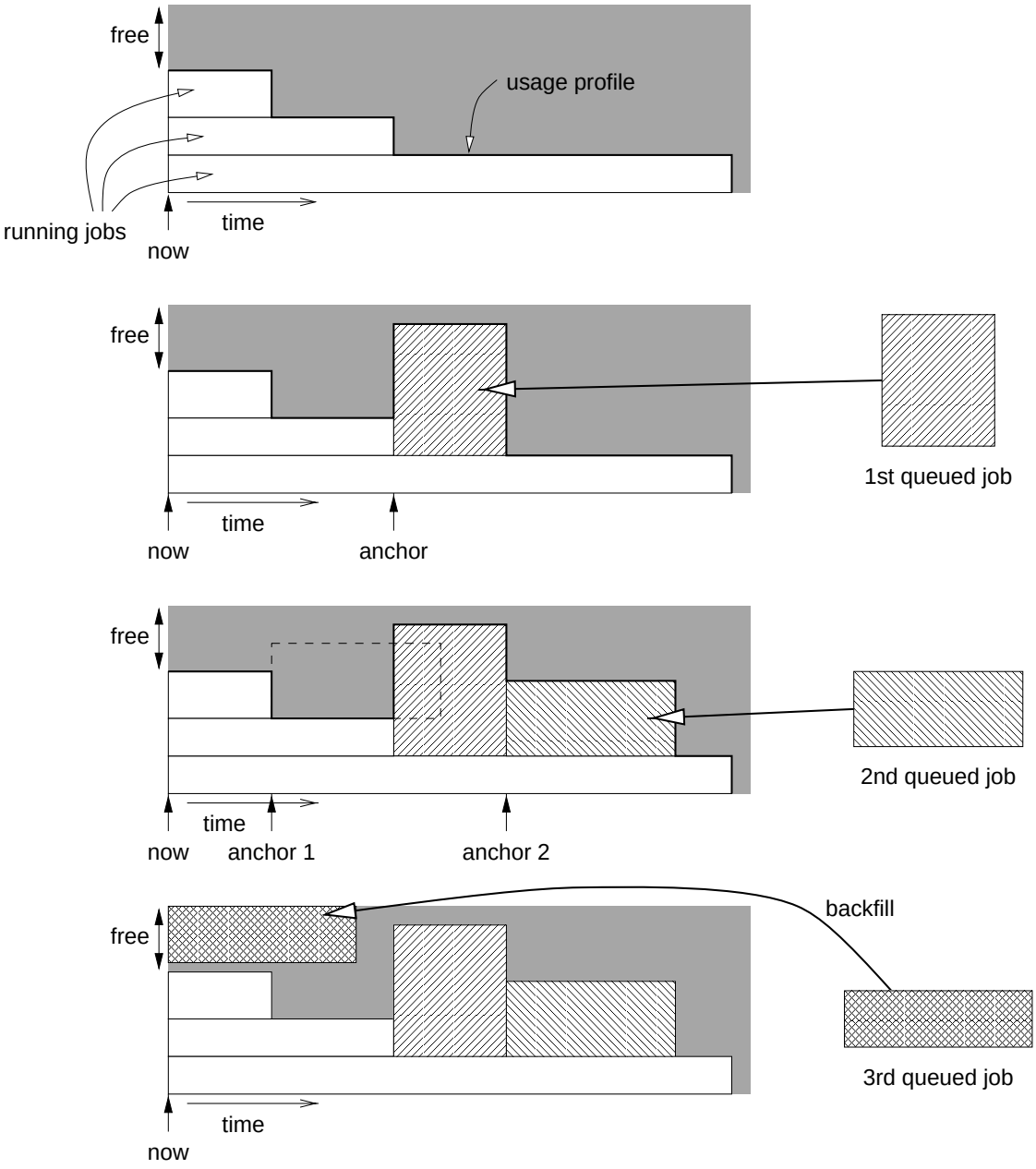

Figure 3: *Example of conservative backfilling.*

job adds at most one item to the profile. As the profile is scanned once for each queued job, the complexity of the algorithm is quadratic 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. But they may actually be able to run sooner because previous jobs terminated earlier than expected. The question is what to do when this happens. Options are

- do nothing, and allow future arrivals to use the idle processors via backfilling, or use this to increase the flexibility of the scheduling, as described below.
- initiate a new round of backfilling when these resources become available. this can move small jobs way ahead of their originally assigned start time.
- retain the original schedule, but compress it. This stays closest to the start times decided when the jobs were submitted so it may be the most convenient for users.

The second option — re-scheduling all the jobs — sounds very promising, but turns out to violate the execution guarantees made by conservative backfilling. The guarantee is embodied in the system's prediction of when each job will run. As each job is submitted, the system scans the usage profile, finds the earliest time that the new job can run without delaying any previous job, and *guarantees* that the job will start at this time or earlier. 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. 4: 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 compression, meaning that the original schedule is retained, but each job is moved forward as much as possible. This can be done in either of two ways. In the first, the profile is re-generated from scratch, but the jobs are considered in the order they appear in the original schedule, rather than in the order of arrival. Returning to the example in Fig. 4, the second queued job stays in front and moves up from $T1$ to the time of the early termination, while the first queued job moves up from $T2$ to $T4$. The second option is to retain the profile and update it one job at a time. For each job, we remove it from the profile, and then re-insert it at the earliest possible time. This approach has two advantages: first, the jobs can be considered in the order of arrival, so jobs that are waiting longer get a better chance to move forward. Second, jobs provably do not get delayed, because at worse each job will be re-inserted in the same position is held previously.

The use of compression also has another implication: as the schedule is maintained and isn't changed by future events, it also makes sense to maintain the usage profile continuously. As jobs arrive and terminate, the profile is updated rather than being re-generated from scratch each time. Thus the algorithm in Fig. 2 is replaced by two separate procedures:

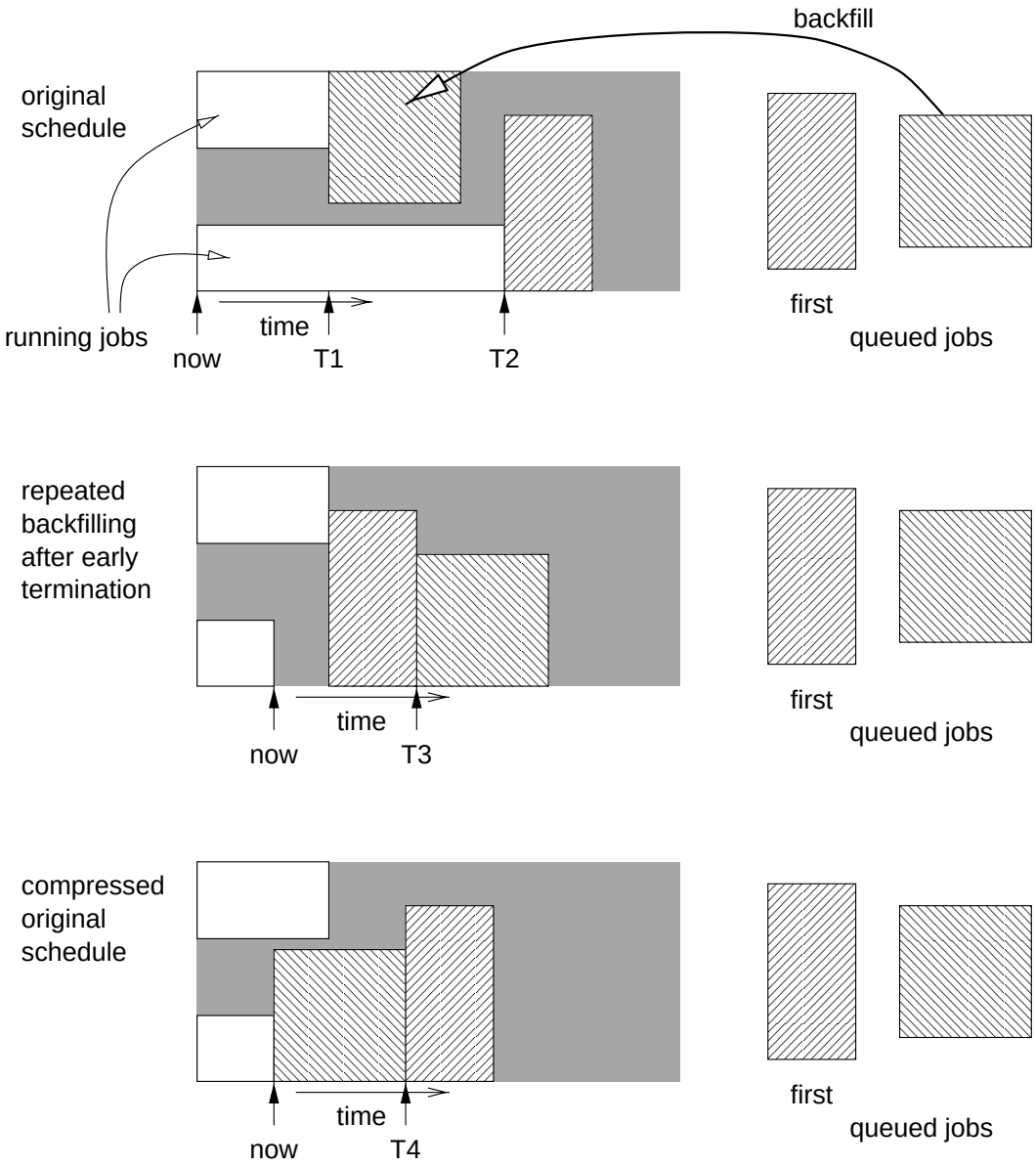

Figure 4: *Repeated backfilling after a running job terminates earlier than expected may cause a job that was expected to backfill to actually run later than the original prediction. It is therefore better to just compress the original schedule.*

input:

- list of queued jobs with nodes and time requirements
- list of running jobs with node usage and expected termination times
- number of free nodes

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

Figure 5: *The EASY backfilling algorithm.*

- Upon arrival, the first possible starting time for the new job is found based on the current profile, and the profile is updated. This is just the inner loop of the original algorithm.
- Upon termination, the profile is scanned and the schedule is compressed.

The complexity of the insertion procedure is only linear in the number of jobs, rather than quadratic. 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* [10]. Interaction with other

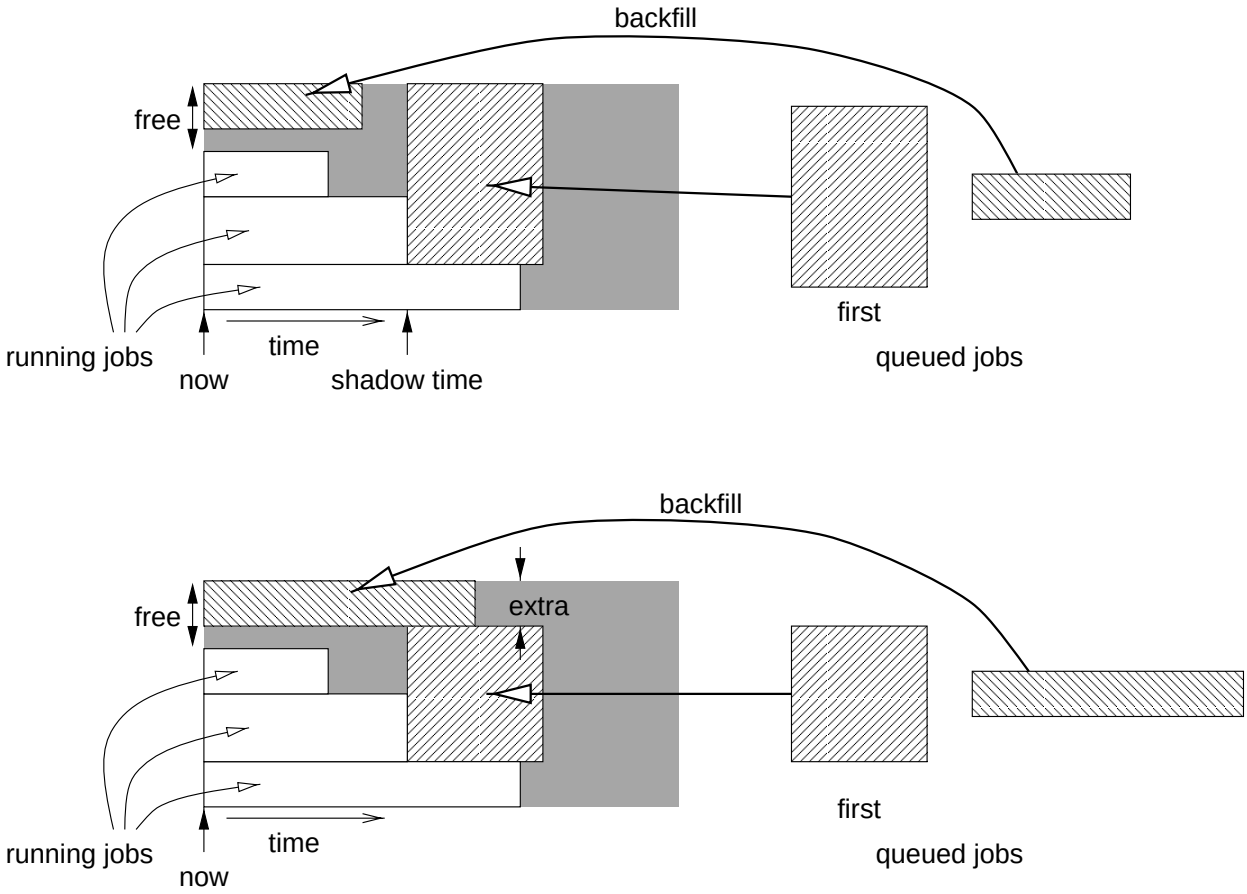

Figure 6: *The two conditions for backfilling in the EASY algorithm.*

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 shown schematically in Fig. 5. This algorithm is executed if the first job in the queue cannot start, and identifies a job that can backfill if one exists. Such a job must require no more than the currently available processors, and in addition it must satisfy either of two conditions that guarantee that it will not delay the first job in the queue (Fig. 6): either it will terminate before the time when the first job is expected to commence (the "shadow" time), or else it will only use nodes that are left over after the first job has been allocated its nodes (the "extra" nodes).

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

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

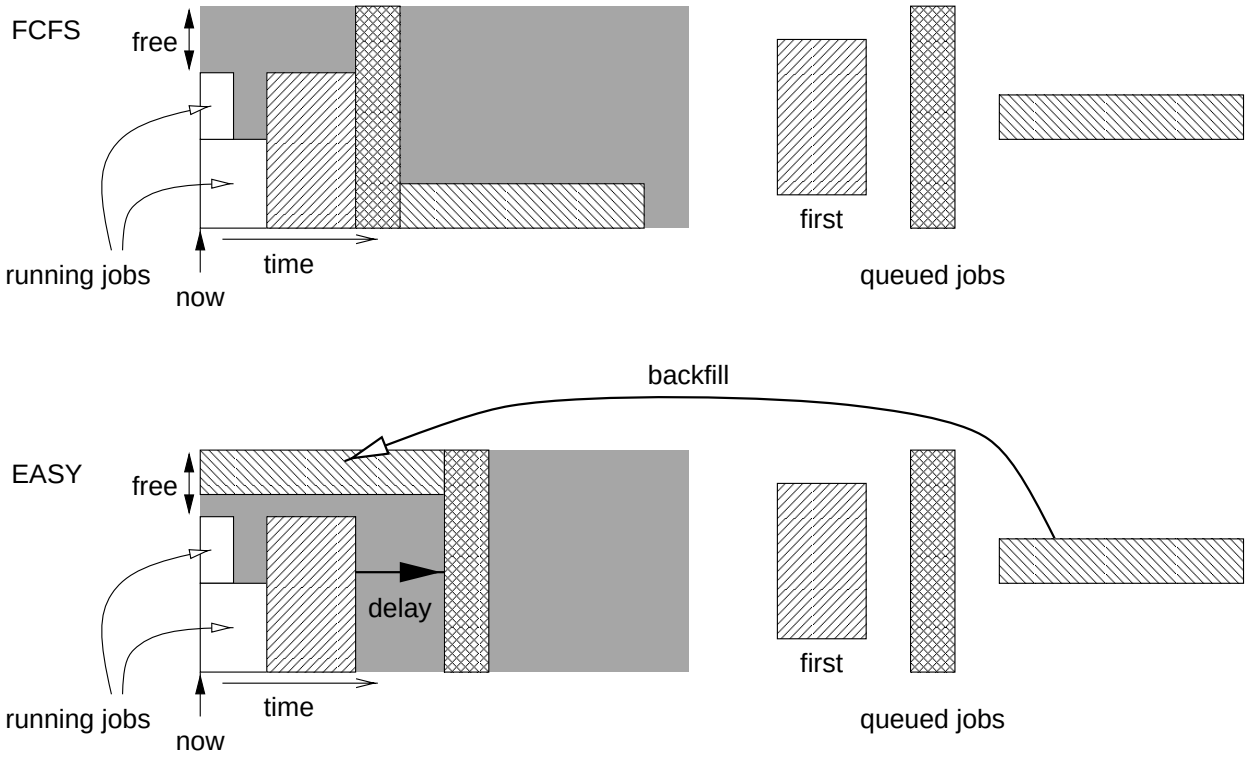

Figure 7: *In EASY, backfilling may delay queued jobs.*

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. 7: 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.

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. A counter example is shown in Fig. 8. Therefore detailed simulations are required to evaluate the real contribution of this approach. The results of such simulations are presented below.

3 Experimental Results

A number of experiments were conducted to compare the different versions of backfilling described above. The first was based on a general parallel workload model, and assumed perfect knowledge about job runtimes. The second made direct use of workload traces collected from SP2 sites using EASY.

3.1 Evaluation with Workload Model

The first simulation used a workload model derived from traces taken on several production systems, and used previously in [3]. Such a model allows the load on the simulated system to be modified in a controlled manner, to see how performance depends on system load. As the model does not contain user estimates of the runtime, we use the actual times as the estimate. This means that both algorithms benefit from accurate estimates.

The performance metric used is the functional relationship of bounded slowdown on load. This should be understood as a queueing system, where load causes jobs to be delayed. Slowdown is used rather than response time to normalize all jobs to the same range. Bounded slowdown eliminates the emphasis on very short jobs [5]; a threshold of 10 seconds was used. For the record, the equation is

$$b\_sld = \begin{cases} \dfrac{T_\ell}{T_d} & \text{if } T_d > 10 \\ \\ \dfrac{T_\ell}{10} & \text{otherwise} \end{cases}$$

where b_sld is the bounded slowdown, T_d is the job's runtime on a dedicated system, and T_ℓ is the job's runtime on the loaded system (i.e. the actual runtime plus the time waiting in the queue).

Results are that the performance of conservative backfilling and EASY backfilling is practically identical, indicating that the aggressiveness of EASY backfilling is unwarranted (Fig. 9). The conservative algorithm only has a slight advantage at very high, practically unrealistic, loads. Interestingly, both algorithms also perform about the same amount of backfilling, with the conservative one doing a bit more than the more aggressive EASY!

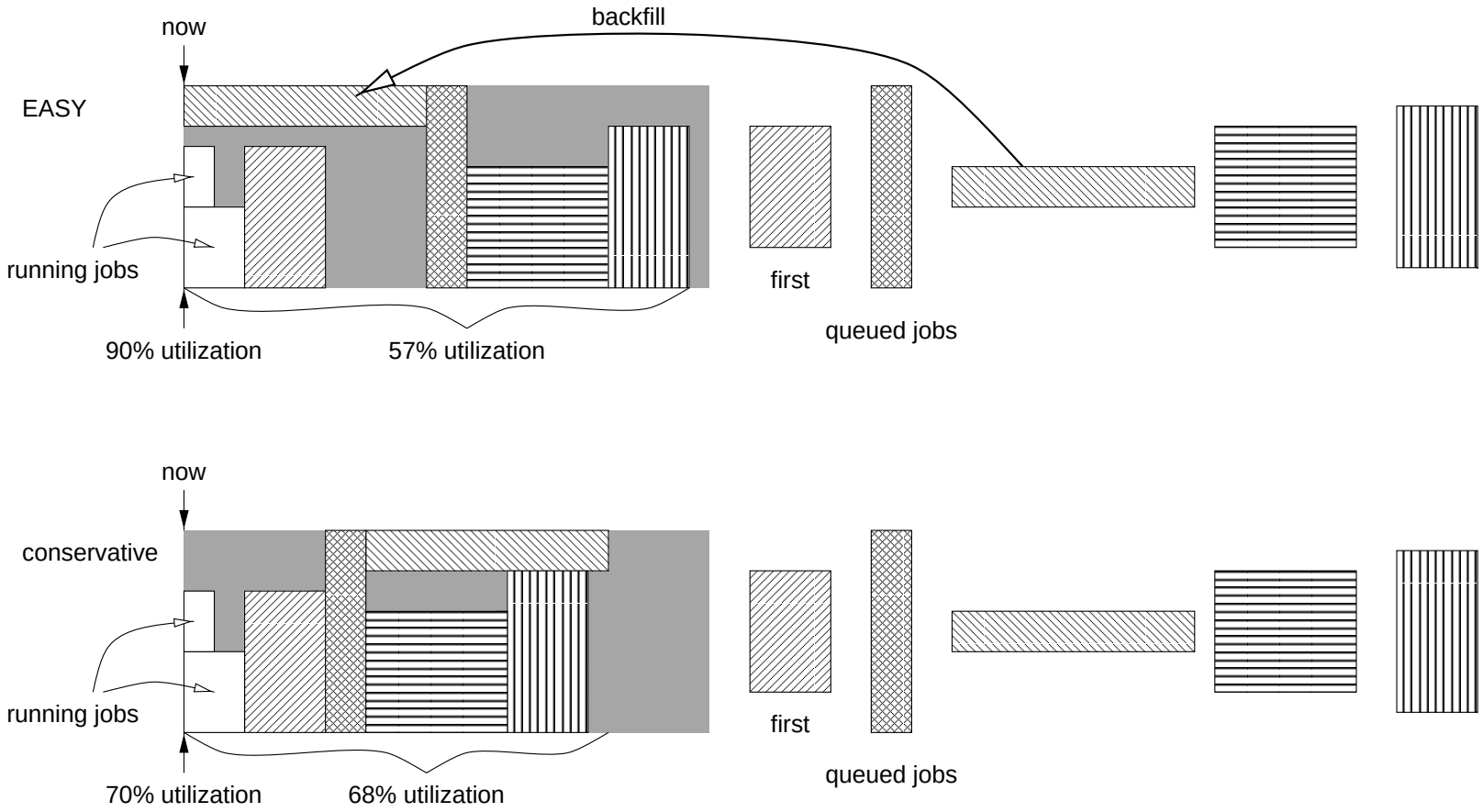

Figure 8: *EASY backfilling is more aggressive than conservative backfilling, so it improves the utilization on the short run. However, it may degrade utilization on the long run.*

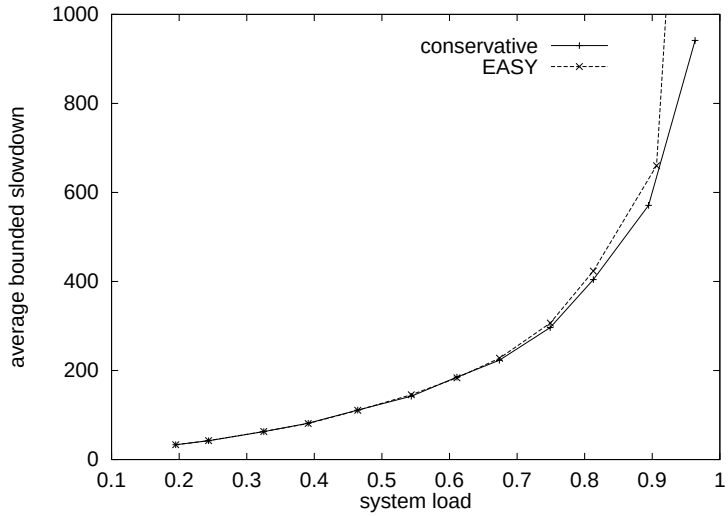

Figure 9: *Experimental results comparing conservative backfilling and EASY backfilling.*

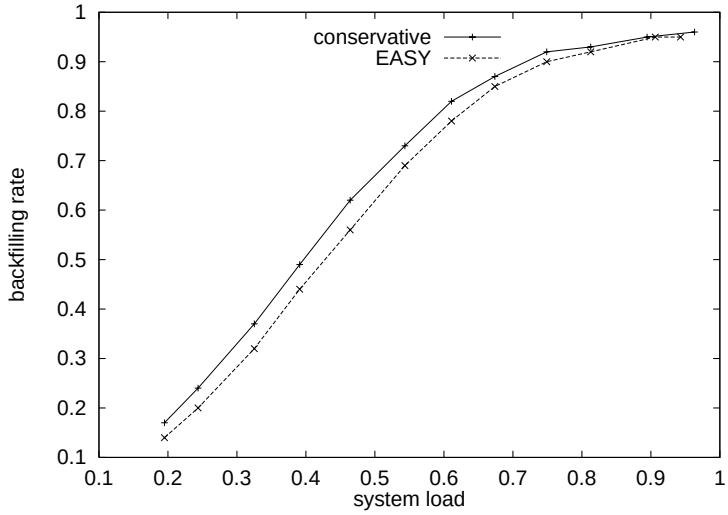

Figure 10: *The amount of backfilling done by the two schemes.*

(Fig. 10) In any case, at high loads nearly all jobs are backfilled, which explains the big improvement that is observed relative to FCFS.

3.2 Evaluation with Real Workload

Using a real workload for evaluating a scheduler is important for two reasons. First, workloads change from installation to installation, so this sort of evaluation is the most accurate for a specific site. Second, the observed workload actually depends on the scheduler being used, because users adapt their requests to what the system supports. Thus it is best to

		33		49		58	
month	*jobs*	EASY	cons	EASY	cons	EASY	cons
Jan	635	8176	8306	1769	1586	492	413
Feb	562	1886	2580	328	318	201	201
Mar	1406	10077	10366	1088	1137	540	519
Apr	493	510	496	112	119	39	38
May	529	2954	2850	410	397	145	145
Jun	566	2551	2508	438	421	213	213
Jul	550	4075	3341	623	607	315	312
Aug	462	3238	3540	807	860	251	251

Table 1: *Average bounded slowdown results for the IUCC workload.*

compare backfilling algorithms for the SP2 using a direct trace from an SP2 system using EASY. An additional advantage is that actual user estimates are then available. We used two traces: one from the 64-node machine installed at the Inter-University Computation Center (IUCC) in Tel-Aviv, Israel, and the other from the 100-node machine installed at the Royal Institute of Technology (KTH) in Stockholm, Sweden.

The workload trace recorded at IUCC covers the period of January through August of 1997, and includes 5203 useful jobs. During this time, the machine was scheduled by IBM's LoadLeveler, and EASY was not used. Therefore this trace does not include data about user estimates, and we again used the real times as the estimates. The workload trace from KTH spans the period from October 1996 to August 1997, and includes 28240 jobs. This system was scheduled using EASY, and the user estimates were recorded in the trace and used by us.

Given that all the details — including each job's time of arrival — are part of the workload data, the simulations just provide a single data point. To better characterize the relationship between the algorithms we therefore do two things: first, we report the results for each month separately, leading to multiple data points for somewhat different workloads. Second, in the case of the IUCC machine, we simulate the system at three different sizes, thus (artificially) creating different load conditions. The sizes used are 33, 49, and 58 nodes. This reflects the actual division of nodes into pools: 6 nodes are used for interactive work and as a file server, leaving 58 for all the batch jobs. Of these, 9 nodes are used as the general batch pool, leaving 49 nodes for a special pool used exclusively by large jobs with 16/17 or 32/33 nodes (17 and 33 nodes are used by master-slave type jobs). Previously, this pool only had 33 nodes.

The results for IUCC are shown in Table 1. In most cases, the results indicate that both algorithms lead to similar or even identical average bounded slowdown values. In some cases the conservative algorithm has a lower average slowdown, and in some the EASY algorithm leads to a lower average slowdown. The only trend is that the EASY algorithm tends to perform better under high loads (that is, when less nodes are assumed). In the 33-node simulation, EASY beat conservative half of the time, whereas in the 58-node simulation

month	*jobs*	EASY	cons
Sep	86	2	2
Oct	2377	93	76
Nov	1988	128	135
Dec	2294	86	124
Jan	2899	97	81
Feb	2908	122	134
Mar	2078	104	118
Apr	2820	83	92
May	4061	67	60
Jun	2694	37	31
Jul	2160	32	34
Aug	1925	50	57

Table 2: *Average bounded slowdown results for the KTH workload.*

conservative won half of the time the the rest were a tie. In any rate, it seems that the conservative algorithm does not degrade performance relative to the more aggressive EASY algorithm.

The results for KTH are shown in Table 2. Again, the performance of both algorithms is similar, with a slight and non-decisive advantage for EASY (it is better in 7 cases, while conservative is only better in 4).

To summarize, the results of the simulations using real workload traces agree with the results of the simulations using the workload model: both show that the performance obtained from the two algorithms is similar. In other words, the performance of the conservative algorithm is about as good as that of the EASY algorithm, and the added predictability comes at no cost in performance.

3.3 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 the estimates are tight, but would be killed if the estimates are too low. Using the data contained in the EASY workload traces from KTH, we can check the validity of this assumption.

The data is shown in Fig. 11, and includes records of 20054 jobs. 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% (with 3215 jobs, or 16% of the total). However, this is largely attributed to jobs that reached their allocated time and where then killed by the system — this happened to 3204 of the 3215 jobs, or 99.7%[1].

[1]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 is actually done. Indirect data is that 793 of the jobs killed by the system had requested 4 hours, which

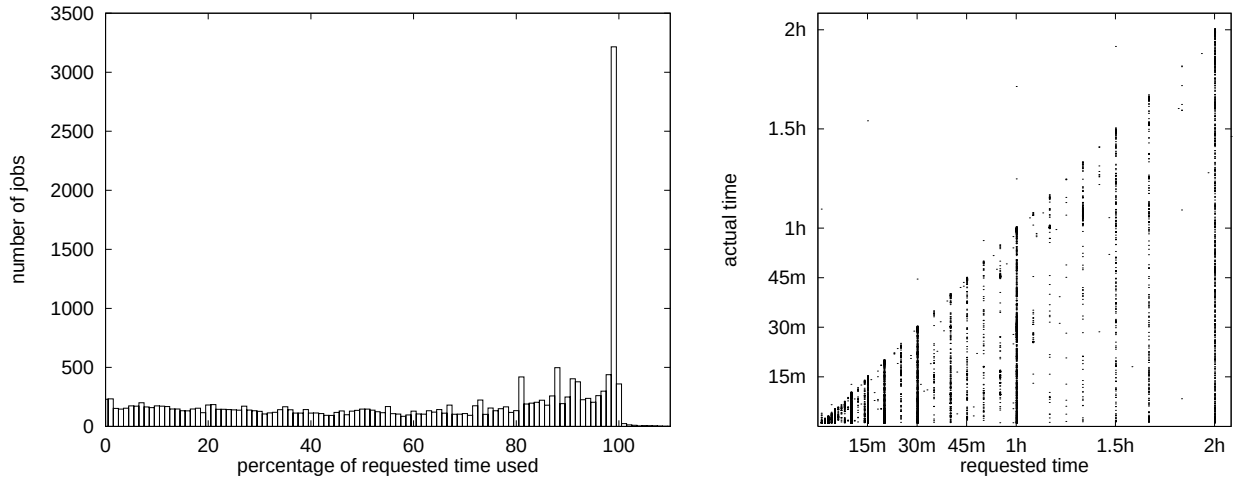

Figure 11: *User runtime estimates and actual runtimes, from the SP2 at the Royal Institute of Technology (KTH), Stockholm, Sweden.*

As the rest of the distribution is quite flat, the conclusion is that user estimates are actually rather poor.

The same data is shown again in the scatter plot on the right of the figure, which shows actual pairs of estimated runtime and the corresponding actual runtime (actually only jobs requesting up to 2 hours are shown, but this is the vast majority of jobs. The highest requests were for 60 hours). This shows that users often, but not always, round their estimates to a "nice" number (typically multiples of 5 minutes up to about an hour and a half). 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 that do not terminate by the estimated time, leading to the triangular shape of the scatter plot.

In order to check the sensitivity of the backfilling algorithms to such poor estimates, we tested them with estimates of various qualities. Using the KTH workload, 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). The results are shown in Table 3, where $f = 1$ indicates completely accurate estimates, and the bottom line gives the results of the actual user estimates from the trace. Three conclusions can be reached:

- Our model of inaccuracy does not capture the full badness of real user estimates. The results for the original estimates are worse than those with our worst estimates.

is the limit imposed during the daytime. It is plausible that many of these were restartable, leading to an estimate of about 1 in 4 jobs.

f	EASY	cons
1	62	61
4	57	53
11	51	44
31	57	45
101	62	57
301	59	52
users	81	84

Table 3: *Average bounded slowdown for the complete KTH workload with varying runtime estimates.*

- 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. We are looking into this phenomenon in a followup study.
- The conservative algorithm seems to operate better than the EASY algorithm when faced with our inaccurate estimates. However, it should be remembered that this is not necessarily true with the inaccurate user estimates.

4 Conclusions

Backfilling is advantageous because it provides improved responsiveness for short jobs combined with no starvation for large 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. However, EASY still does not allow the time at which a job will run to be estimated with any degree of accuracy, because of its aggressive backfilling algorithm. We showed that it is possible to add predictability without loss of utilization by using a more conservative form of backfilling, in which short jobs can start running provided they do not delay any previously queued job.

While backfilling was developed for the SP2, and our evaluations used workload traces from SP2 sites, this work is applicable to any other system using variable partitioning. This includes most distributed memory parallel systems in the market today.

5 Acknowledgements

This research was supported by the Ministry of Science and Technology. Thanks to Jonathan Horen and Gabriel Koren of IUCC and Lars Malinowsky of KTH for their help with the workload traces.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.