Utilization and Predictability in Scheduling the IBM SP2 with Backfilling

Dror G. Feitelson Ahuva Mu'alem Weil
Institute of Computer Science
The Hebrew University, 91904 Jerusalem, Israel

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. An alternative is to use 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 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 incoming jobs. But this approach suffers from fragmentation, where available processors cannot meet the requirements of the next queued job and therefore remain idle.

It is well known that the best solutions for this problem are to use dynamic partitioning [8] 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 [4]. 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 [6, 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) [7], which has since been integrated with the LoadLeveler scheduler from IBM for the SP2 [9].

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. The jobs then run on dedicated partitions of the requested size. Note that

An extended version of this paper is available as Technical Report CS-98-2, accessible from URL http://www.cs.huji.ac.il/~feit.

1063-7133/98 $10.00 © 1998 IEEE

Input:

- Queued jobs with nodes and time requirements
- Running jobs with nodes and expected termination
- Number of free nodes

Algorithm conservative backfill from scratch:

1. Generate processor usage profile of running jobs
   (a) Sort running jobs according to expected termination time
   (b) Divide the future into periods according to job terminations, recording the nodes used in each; this is the usage profile
2. Try to backfill with queued jobs
   (a) Loop on queued jobs in order of arrival
   (b) For each, find the first point where enough nodes are available. This is called the anchor point
      i. Verify that the nodes remain available until the job's expected termination
      ii. If so, update the profile to reflect allocation of nodes to this job
      iii. If not, find next possible anchor point, and repeat the check
   (c) The first job found that can start immediately is used for backfilling

Figure 1. *The conservative backfilling algorithm.*

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.

2.1. Conservative Backfilling

Conservative backfilling is the vanilla version usually assumed in the literature (e.g. [5, 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. 1). 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. 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.

It is most convenient to maintain the profile in a linked list, as it may be necessary to split items in 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 queued job, the complexity of the algorithm is quadratic in the number of jobs.

Note that jobs may be able to run sooner than their assigned start time, because previous jobs may terminate earlier than expected. When this happens it is necessary to compress the schedule, meaning that each job is moved forward as much as possible. To do so, each job is removed from the profile, and then re-inserted 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. This reduces the complexity of scheduling new jobs from quadratic to linear.

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* [7]. 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.

Input:

- Queued jobs with nodes and time requirements
- Running jobs with nodes and expected termination
- Number of free nodes

Algorithm EASY backfill:

1. Find the shadow time and extra nodes
   (a) Sort running jobs according to expected termination time
   (b) Find when enough nodes will be available for the first queued job; this is the shadow time
   (c) If this job does not need all the available nodes, 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 check whether either of these conditions holds:
      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 2. *The EASY backfilling algorithm.*

The algorithm is shown schematically in Fig. 2. 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: 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.*

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. 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. ∎

3. Experimental Results

A number of experiments were conducted to compare the two 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 [4]; a threshold of 10 seconds was used. For the record, the equation is

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

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 are practically identical, indicating that the aggressiveness of EASY backfilling is unwarranted (Fig. 3). In fact, the conservative algorithm even has a slight advantage at very high, practically unrealistic, loads. Interestingly, both algorithms also perform about the

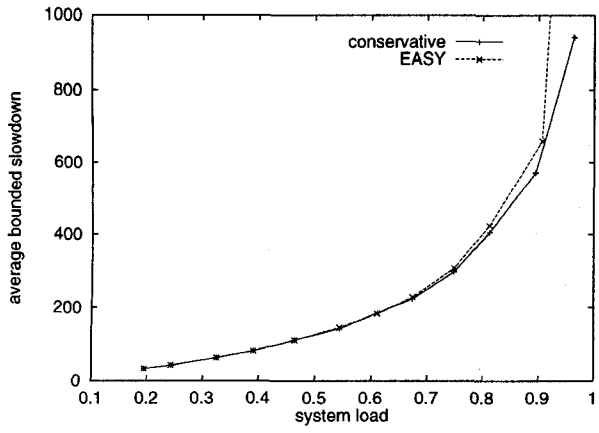

Figure 3. *Experimental results comparing conservative backfilling and EASY backfilling.*

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

month	EASY	cons
Sep	2	2
Oct	93	76
Nov	128	135
Dec	86	124
Jan	97	81
Feb	122	134
Mar	104	118
Apr	83	92
May	67	60
Jun	37	31
Jul	32	34
Aug	50	57

Table 1. *Average bounded slowdown results for the IUCC workload (above) and the KTH workload (right).*

same amount of backfilling, with the conservative one doing a bit more than the more aggressive EASY! 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 Workloads

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 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, which was running LoadLeveler (without user estimates), and the other from the 100-node machine installed at the Royal Institute of Technology (KTH) in Stockholm, Sweden, which actually used EASY.

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, which reflect the actual division of nodes into pools.

The results are shown in Table 1. In most cases, the results indicate that both algorithms lead to similar or even identical average bounded slowdown values. There is no clear advantage to one over the other. In any rate, it seems that the conservative algorithm does not degrade performance relative to the more aggressive EASY algorithm. Thus the results for the real workload agree with those of the model.

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

*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 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.

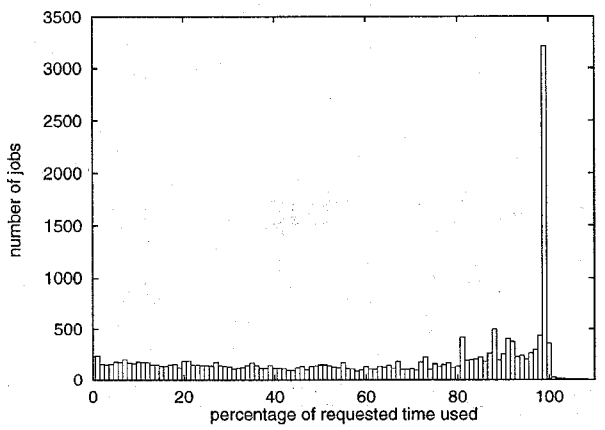

Figure 4. *Actual runtimes as a percentage of user estimates, from the KTH machine.*

Table 2. *Average bounded slowdown for the complete KTH workload with varying runtime estimates.*

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

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 2, where $f = 1$ indicates completely accurate estimates, and the bottom line gives the results of the actual user estimates from the trace. The main conclusions are that 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. In addition, 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.

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.

Acknowledgements

This research was supported by the Ministry of Science and Technology. Many 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*, pp. 238–261, Springer Verlag, 1997 (LNCS 1291).

[4] 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*, pp. 1–34, Springer Verlag, 1997 (LNCS 1291).

[5] R. Gibbons, "*A historical application profiler for use by parallel schedulers*". In *Job Scheduling Strategies for Parallel Processing*, pp. 58–77, Springer Verlag, 1997 (LNCS 1291).

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

[7] D. Lifka, "*The ANL/IBM SP scheduling system*". In *Job Scheduling Strategies for Parallel Processing*, pp. 295–303, Springer-Verlag, 1995 (LNCS 949).

[8] 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.

[9] J. Skovira, W. Chan, H. Zhou, and D. Lifka, "*The EASY - LoadLeveler API project*". In *Job Scheduling Strategies for Parallel Processing*, pp. 41–47, Springer-Verlag, 1996 (LNCS 1162).