Backfilling with Fairness and Slack for Parallel Job Scheduling

Angela C. Sodan and Wei Jin
University of Windsor, Computer Science, Windsor, Ontario, Canada

Corresponding author’s email address: acsodan@uwindsor.ca

Abstract. Parallel job scheduling typically combines a basic policy like FCFS with backfilling, i.e. moving jobs to an earlier than their regular scheduling position if they do not delay the jobs ahead in the queue according to the rules of the backfilling approach applied. Commonly used are conservative and easy backfilling which either have worse response times but better predictability or better response times and poor predictability. The paper proposes a relaxation of conservative backfilling by permitting to shift jobs within certain constraints to backfill more jobs and reduce fragmentation and subsequently obtain better response times. At the same time, deviation from fairness is kept low and predictability remains high. The results of the experimentation evaluation show that the goals are met, with response-time performance lying as expected between conservative and easy backfilling.

1. Introduction
Parallel job scheduling requires packing of jobs with different runtimes and different requirements in regards to the numbers of CPU resources. Jobs are typically queued, and different scheduling policies are possible. First-come-first-served (FCFS) is often considered to be a fair scheduling approach but leads to high fragmentation and subsequently high response times if used on its own. Other approaches like priority scheduling (with priorities typically based on job runtimes) or fair-share priority scheduling are often practically applied. Very often the basic scheduling policy is combined with backfilling: 60% of the Top50 machines apply some form of backfilling. Backfilling means that jobs can move to an earlier scheduling position if they do not delay jobs ahead in the queue according to certain criteria. This improves utilization and subsequently average response times.

Two forms of backfilling [10] are commonly used:

- Conservative backfilling: permits moving jobs ahead if they do not delay any waiting job ahead of them. If combined with FCFS, it therefore maintains the fairness of FCFS (except that backfilled jobs receive preferential treatment) and supports prediction of response times. However, conservative backfilling can lead to wasted space and non-optimal utilization and subsequently increased average response time (see Figure 1 a).

- Easy backfilling\(^1\): permits moving jobs ahead if they do not delay the first waiting job (which prevents starvation). Easy backfilling tends to provide better utilization in high-load

---

\(^1\) Note that the original scheduler proposing easy backfilling was called EASY [8] and was a combination of FCFS and easy backfilling. In this paper, we consider the backfilling approach as being separate from the basic scheduling policy, i.e., a priority scheduler could also be combined with either conservative or easy backfilling.
phases but also means giving preference to shorter jobs which may push back other jobs (see Figure 1 b). Thus, easy backfilling provides less fairness. Since jobs are often pushed back, predictability is low.

As the contribution of this paper, we present a new slack-based backfilling approach which addresses the above problems and is computationally cheap and easy-to-integrate into the overall scheduling policy. The approach is based on conservative backfilling but relaxes it by permitting constrained delays, called slack. The goal is to increase utilization in high-load phases and subsequently response times by better packing, while keeping the schedule close to FCFS. The latter means that both fairness and predictability are maintained with acceptable deviation. Improved utilization is demonstrated in Figure 1 c. Backfilling Job 6 achieves higher utilization improvement than Job 4 or Job 5. Under the objective of maximizing utilization improvement, therefore Job 6 would be chosen.

![Diagram](image_url)

Figure 1. Differences between backfilling approaches: a) FCFS with conservative backfilling: no waiting job may be delayed, b) FCFS with easy backfilling (EASY): only the first waiting job may not be delayed, and c) FCFS with slack backfilling: all waiting jobs may be delayed but only to some extent. With slack backfilling, Job 6 gives better utilization than Job 4 or Job 5 and is therefore chosen (slack constraint is met).
To accomplish these goals, combinations of all possible backfill jobs are considered. This involves trying different combinations with the goal of finding the best combination. To make the search feasible, several heuristics are applied to prune the search tree. In addition, the number of waiting jobs considered for backfilling is limited to a lookahead of a certain number of waiting jobs. This is a feasible restriction, considering that anyway closeness to FCFS matters.

The approach is evaluated via simulation with several workloads. The results demonstrate that slack backfilling maintains high fairness and predictability, while performing better than conservative backfilling.

2. Related work

Different definitions of fairness with different corresponding metrics are possible, depending on the criteria: typically seniority, service requirements, or cost. Most common in queuing approaches is to use seniority; with the fairest approach then being first-come-first-served (FCFS). Service-oriented fairness would permit jobs with lower service requirements (shorter runtime) to move ahead of jobs that arrived earlier but run longer. This corresponds to the often applied priority scheduling with higher priority for shorter jobs. However, the metric neglects the difference in arrival times between jobs of different classes. Thus, a combined metric is proposed in [11], which considers both the difference between the service times of two jobs and the difference in their arrival times. The difference between deserved (an equal fraction of the processor time among all jobs) vs. the received service is calculated as positive or negative discrimination. Overall unfairness then is defined as the variance of the individual jobs’ discrimination. However, this approach is presented for an M/M/1 queuing system which does not reflect problems in parallel job scheduling like packing varying numbers of jobs into the machine at the same time (space sharing) or applying backfilling.

Some approaches for parallel job scheduling use relative response time (often called slowdown) which is the ratio between response time and runtime. This metric combines wait time (and therefore indirectly differences in arrival times) with runtimes (service times) into a single value. In the context of an M/M/1 system, the argument is raised that mere slowdowns are counter-intuitive since FCFS would be rated as always unfair [1]. In [7], relative expected slowdowns are compared between running and waiting jobs and running jobs preempted if waiting jobs receive significantly worse service defined via the slowdowns. However, slowdown is typically combined with a delay factor. Thus, in [13], λ-fairness is used which means an approach is fair if no job is delayed by more than a factor of λ. (response time is not increased by more than a factor λ).

λ-fairness can be considered a special case of slack-based scheduling. In [16], slacks are introduced as delay factors and used to modify the order of the jobs for better utilization. This means that slacks serve as constrained flexibility and reordering is permitted as long as no job is delayed beyond its slack. Slacks are calculated dynamically based upon dynamically adjusted priorities, a static slack factor, and a static average wait time. This idea has been used in various approaches like [4][15][19]. However, in the approach presented in [19], slack factors, applied to wait time, need to be as large as 2 to obtain a benefit.

Using slack as a constraint is supported by the definition of scheduler goodness in [16], which is based on overall happiness defined as the happiness of the unhappiest user.

Another fairness idea is fair share which approximates giving certain users or user groups certain shares of the machine (considering both time and space) over certain time intervals, which then are mapped to priorities [5]. The approach has become widespread but is tuned by many parameters and lacks a theoretical basis.

In [12], two unfairness metrics are proposed and evaluated (without resulting into a preference for one or the other). One metric transfers the seniority/service-time metric of [11] to parallel processing by weighting the resource share according to the size of the job. The second metric determines unfairness as jobs being delayed vs. their relaxed fair start time, which is seniority-oriented. The relaxed fair start time expresses the start time of a job without this job being backfilled (while other
jobs ahead in the queue may backfill). In both cases, fairness is calculated after the execution of a workload to rate and compare the fairness of different scheduling policies, as does the original M/M/1-oriented metric of [11]. Our approach is closely related to the second metric as this metric can be modified to apply in dynamic evaluation and works well for non-preemptive scheduling.

In regards to backfilling, typical approaches make backfilling decisions one-job-at-a-time. An exception is the approach in [14] to which our approach is closely related. However, the approach in [14] is applied to both filling and backfilling together (i.e. whenever new jobs are packed into the machine) and is oriented toward easy backfilling; only the first job is guaranteed to be scheduled at the next possible time (resources available). Otherwise, the approach considers the whole combination of jobs (with a certain lookahead) that can potentially be filled/backfilled and selects the combination which provides the highest utilization (defined as the currently used resources vs. the available resources). An optimal solution is found via dynamic programming. However, this approach does not consider fairness. In [15], we have also optimized the next combination of jobs to be scheduled, looking further ahead into the placement of a whole group of jobs and into the resulting schedule and utilization over the corresponding time frame. Jobs are reordered within certain slacks, defined by general slack factors. However, though successfully improving the performance, the approach is expensive.

3. Backfilling with Fairness and Slack

3.1. Slack-based Backfilling Algorithm

Our proposed slack-based backfilling algorithm consists of two parts which apply at job-submission time and at job-scheduling time. In difference to some other approaches, the job-scheduling time for considering relaxed ordering is exclusively the backfilling time, i.e. at each scheduler invocation, jobs are first filled regularly. However, backfilling then not only relaxes the rules in regards to delays for other waiting jobs but looks at the overall situation and attempts to find the best group of jobs to be backfilled rather than making decisions one-by-one as most other schedulers do. The basic backfilling approach used for estimations for fair start time and as a default (if no combination meeting the criteria can be found) is conservative backfilling. The backfilling approach support two possible objectives:

- Objective 1: closeness to FCFS while reasonably improving utilization and
- Objective 2: high utilization improvement while staying reasonably close to FCFS.

3.1.1. Actions at job-submission time

First, the fair start time of the job is calculated (the term and the idea originate from [12], the exact corresponding term being 'relaxed fair start time'). For the jobs ahead, FCFS with conservative backfilling is applied but the job itself is not backfilled but placed after all jobs submitted before it are packed into the machine. The fair start time is obtained by simulating the schedule ahead. With FCFS and conservative backfilling, this is a feasible approach. Note that when the job is actually scheduled, it may be backfilled. This means that backfilling is not considered a guarantee or potential source of later unfairness if backfilling is no longer possible. Rather backfilling is considered as a potential bonus.

After knowing the job’s fair start time, a slack factor \( F_{\text{slack}} \) is applied to the resulting estimated wait time \( T_{\text{wait}} \) to define the maximum permitted delay. \( F_{\text{slack}} \) is calculated in the range \([F_{\text{slack,min}}, F_{\text{slack,max}}]\) with an exponential function of wait time:

\[
F_{\text{slack}}(T_{\text{wait}}) = \min \left( F_{\text{slack,max}}, k_2 e^{-k_1 T_{\text{wait}}} + F_{\text{slack,min}} \right)
\]

with \( k_2 \) to be set such that \( k_2 \geq F_{\text{slack,max}} - F_{\text{slack,min}} \).

\[(1)\]
This means that $F_{\text{slack}}$ decreases non-linearly with larger wait times to avoid too much additional delay for jobs that already have to wait a long time. $F_{\text{slack}}$ represents the maximum delay possible per job until it is scheduled. However, to avoid using up this slack in scheduling decisions early during the waiting time of the job and to keep some flexibility for later backfilling-decision points, a partial slack $F_{\text{slack,partial}}$ is calculated which is the slack that may be used in any particular scheduling decision and is proportional to how long the job already waited ($T_{\text{wait,curr}}$). The slack constraint used during backfilling is based on $F_{\text{slack,partial}}$ (not on $F_{\text{slack}}$). Note that $F_{\text{slack,partial}} \leq F_{\text{slack}}$. The first-time partial slack that may be used in a scheduling decision at any point of time is defined as

$$F_{\text{slack,partial}} = F_{\text{slack,min}} + (F_{\text{slack}} - F_{\text{slack,min}})*T_{\text{wait,curr}}/T_{\text{wait,est}}$$  \hspace{1cm} (2)$$

At later decision points, the slack $F_{\text{slack,curr}}$ that was already used should be considered which gives

$$F_{\text{slack,partial}} = F_{\text{slack,min}} + (F_{\text{slack}}/F_{\text{slack,curr}} - F_{\text{slack,min}})*1/F_{\text{slack,curr}}*T_{\text{wait,curr}}/T_{\text{wait,est}}$$  \hspace{1cm} (3)$$

3.1.2. Actions at job-scheduling time

At backfilling time, the algorithm performs the following steps:

1) Select a limited number $n_{\text{lookahead}}$ of backfilling (lookahead) candidates ($S_{\text{cand}}$) from the head of the waiting queue.
2) To find a good solution according to the chosen objective:
   - Try backfilling different combinations of jobs from $S_{\text{cand}}$.
   - Evaluate each solution according to either Criterion 1 (Objective 1) or Criterion 2 (Objective 2).
   - Evaluate the delays of all non-selected jobs. Only solutions passing the corresponding criterion and not violating the slack constraint of any non-selected job will be considered and become part of the possible solutions $\text{solPoss}$.
3) Select best solution from $\text{solPoss}$ according to Objective 1 or Objective 2.

To support the two objectives, the following metrics are calculated which consider those jobs that were delayed vs. their estimated fair start time $FST$ and to what extent on average ($\text{OverallUnfairness}$, abbrev. as OverallUnfair below) and those jobs that were treated preferentially ($\text{OverallPreferentialTreatment}$, abbrev. as OverallPrefer below), i.e. started earlier than their $FST$. Note that this fair start time is calculated at job submission time as an estimate, whereas the original posterior evaluation in [12] uses the final scheduling order.

$$\text{OverallUnfairness} = \sum_{\text{in jobs}} \max(\text{AST}_i - FST_i, 0)/n$$  \hspace{1cm} (4)$$

$$\text{OverallPreferentialTreatment} = \sum_{\text{in jobs}} \max(FST_i - \text{AST}_i, 0)/n$$  \hspace{1cm} (5)$$

with $\text{AST}$ being the actual start time and $n$ being number of jobs.

For evaluation of possible solutions (kept in $\text{solPoss}$), either Criterion 1 (Formula 6) or Criterion 2 (Formula 7) are used. Utilization $\text{Util}$ is measured as the fraction of resources being used.

$$\text{OverallUnfair} / \min\{\text{OverallUnfair}\mid i \text{ in } \text{solPoss}\} \leq \text{OverallUnfairRatio}_{\text{max},1} \wedge$$
$$\text{OverallPrefer} / \{\text{OverallPrefer}\mid \min\{\text{OverallUnfair}\mid j \text{ in } \text{solPoss}\}=i\} \leq \text{OverallPreferRatio}_{\text{max},1} \wedge$$
$$\text{Util} / \{\text{Util}\mid \min\{\text{OverallUnfair}\mid j \text{ in } \text{solPoss}\}=i\} \geq \text{UtilRatio}_{\text{min},1}$$  \hspace{1cm} (6)$$

This means that the deviation from FCFS should stay below certain thresholds, while utilization is improved beyond a certain threshold.
\[
\text{Max}[\text{Util}_i \mid i \in \text{solPoss}] / \text{Util} \leq \text{UtilRatio}_{\text{min}, 2} \land \\
\text{OverallUnfair}_k \leq \text{OverallUnfairRatio}_{\text{max}, 2} \land \\
\text{OverallPrefer}_k \geq \text{OverallPreferRatio}_{\text{max}, 2} 
\] (7)

This means that the deviation of the solution from the maximum possible utilization improvement should be below a certain threshold, while the deviation from FCFS stays below certain thresholds.

Note that the thresholds used in Formula (6) and (7) (indexed 1 and 2) may have different values.

Among the possible solutions in solPoss, finally the solution with lowest overall unfairness (Objective 1) or highest utilization improvement (Objective 2) is selected. If none of the job combinations meets the corresponding criteria, the job combination with lowest overall unfairness (Objective 1) or highest utilization improvement (Objective 2) is used.

The approach employs branch-and-bound for approaching an optimal solution. Branches in the tree correspond to combinations of jobs, i.e. deeper levels mean adding more jobs to the combination. Several heuristics are used to prune the search tree and make the search feasible. They were mentioned as part of the algorithm description above: branches are pruned if the sum of the job sizes in the combination exceeds the number of available resources, if the slack of any non-selected job is violated, or if the criterion of the corresponding goal is not met. Moreover, the criteria, cf. Formula (6) and (7) express a “good-enough” rather than an optimal solution.

3.2. Implementation

The slack backfilling is currently implemented in our preemptive Scojo-PECT scheduler [2] which uses coarse-grain time slices. However, the backfilling approach is usable with any scheduling policy (potentially with even higher improvements due to more backfilling options). Scojo-PECT separates jobs classes based on runtime into short-running, medium-running, and long-running jobs. These classes are scheduled in different time slices, with FCFS per class/slice. The scheduler supports preemption to disk as a default which means that it works well with memory-intensive jobs. Jobs are resumed on the same resources in the following scheduling round in their next slice. Estimation of response times is easily possible if using conservative backfilling since the policy is FCFS.

To minimize fragmentation, unused resources in any slice can be temporarily (until the end of the slice) used by other job classes. This non-type slice backfilling always observes the backfilling rules of the host and of the guest class and only starts new jobs if they can continue running on the same resources in their own slice. Non-type slice backfilling is crucial for good performance and means that actual response times are often shorter than estimated, though preemption overhead can also mean that jobs finish later than expected.

4. Experimental Evaluation

4.1. Experimental Set-up

Slack backfilling was tested within Scojo-PECT, using simulation and different workloads. Two synthetic workloads (generated by the Lublin-Feitelson Workload Model [9]) and two real traces (from the Feitelson Workload Archive [3]) were used, each with 10,000 jobs. Slack backfilling was compared to both conservative and easy backfilling for each workload. For the characteristics of the workloads, see Table 1.

| TABLE 1. WORKLOADS USED WITH MACHINE SIZE $N_{\text{machine}}$ AND UTILIZATION $Util$. |
|---------------------------------------------------------------|
| Machine | Lublin Synthetic 1 | Lublin Synthetic 2 | CM5 | CM5 BLUE |
| $N_{\text{machine}}$ | 128 | 128 | 1024 | 1152 |
| $Util$ in % | 81.5 | 85 | 78 | 74 |
| $RS_M$ | 0.32 | 0.385 | 0.43 | 0.375 |
Scojo-PECT currently makes time-slice decisions in time intervals of 1h: time not needed for an optional short-job slice is split into a slice for medium-running and a slice for long-running jobs. Time shares can be specified as the relative fraction among the slice-lengths between medium-running and long-running jobs ($RS_{M/L}$). The experiments used a fraction that corresponds to the results of a common priority scheduler (giving higher priority to short vs. medium and medium vs. long jobs). Jobs were classified as medium jobs ($M$) if running $\geq 10$ minutes and $< 3$ h, and as long jobs if running $\geq 3$ h ($L$).

Since Scojo-PECT is a preemptive scheduler, swap cost needed to be modeled. The assumption was that only the amount per specific machine node needed to accommodate the new job is swapped out at job-switch time. Though most clusters are based on Linux and Linux does not support such swapping, small modifications of Linux can make this possible [18]. Each node was assumed to have 8 Gbyte of memory (which is a typical size) with 7 Gbyte being available to applications. Swap cost per Gbyte was assumed to be 14 sec (as per personal discussion with system administrators). A job-memory-requirement distribution was derived from the CM5 trace and from information of system administrators. Basically, most jobs have relative low memory requirements, which means that the jobs running on the same resources in different coarse-grain time slices may all be kept in memory. However, some jobs require more memory and make swap-in/swap-out necessary. $L$ jobs tend to have more jobs with high memory requirements than $M$ jobs, which was reflected in the job-memory-requirement distribution definition. For backfilling parameters, see Table 2.

| TABLE 2. BACKFILLING PARAMETERS. |
|----------------------------------|
| $n_{lookahead}$                  | 20 |
| $[F_{slack,min}, F_{slack,max}]$ | [1.05,1.2] or [1.2,1.5] |
| $k_1/k_2$                        | 0.000005 / 0.4 ($T_{wait}$ measured in sec) |
| OverallUnfairRatio$_{max,1}/max,2$ | 1.3 |
| OverallPreferRatio$_{max,1}/max,2$ | 1.3 |
| UtilRatio$_{min,1}/UtilRatio_{min,2}$ | 1.3 |
| $N_{waitjobs}$ defining high-load phase | 16 |

For the evaluation, average response times $R$ and average bounded slowdowns are used. Bounded slowdown is defined as $max(R / max(T_{runtime,bound}), 1)$. The bound prevents too much influence on the overall average from very short-running jobs.

Overall utilization per workload obviously remains the same, even if the packing under backfilling is improved—otherwise, the machine would be overcommitted. However, we can expect that better packing improves the utilization in high-load phases. High-load phases are defined in this work by at least a certain number $N_{waitjobs}$ jobs being in the waiting queue.

4.2. Experimental Results
The results are shown in Figure 2. The results shown are for Objective 1 (closeness to FCFS while reasonably improving utilization). The results for Objective 2 surprisingly looked very similar are are therefore not shown. The results for Objective 1 show that the bounded slowdown was improved by up to 27% (Synthetic 1, L). On average, the improvement was 10.6% if utilization was relatively high (Synthetic 1, Synthetic 2, and CM5). There are less improvements for the lower-utilization workload BLUE. The explanation is obvious (and was confirmed by tests that reduced inter-arrival times in the synthetic workloads to make their load lighter): if the load/utilization is low, waiting queues are shorter and there is therefore less opportunity to make improvements in backfilling. Response times are not shown since their relationship was similar.

Most jobs ($\sim 20\%$) that scheduled differently than estimated by their fair start time, were treated preferentially (due to backfilling and non-type slice backfilling by the scheduler which tends to
decrease response times). Some jobs (~1.5%) were delayed within the defined constraints, i.e. were treated slightly unfair as per our definition.

The utilization difference vs. conservative backfilling in high-load phases was only about 1%. Thus, some benefit may also be due to slightly shorter jobs being better served (more easily backfilled) than slightly longer-running jobs.

![Comparison of Bounded Slowdowns](image)

Figure 2. Bounded slowdown for different workloads and different backfilling approaches.

Testing several different $n_{\text{lookahead}}$ showed that typically 20 jobs, as used in the presented experiments, typically provide the best results—going much higher does not add much benefit, if any. More detailed evaluations can be found in [6].

Comparing slack-based backfilling to conservative and easy backfilling, clearly shows that slack-based backfilling provides results which lie in between conservative and easy backfilling—which was expected by the design of slack backfilling. Moreover, the slack range has an impact on how much benefit is obtained: for the workloads with higher utilization (Synthetic 1, Synthetic 2, and CM5), the larger slack-factor range $[1.2,1.5]$ obviously tends to provide better performance than the range $[1.05,1.2]$. However, even a maximum slack of 1.5 is still relatively low, compared to other approaches reported in the literature such as [19].

The search for a good backfilling job combination was very effective and the pruning of the search tree worked very well: the runtime of the simulation was only increased from about 7min to about 10min per workload and parameter setting, i.e. by about 43%.

The predictability remained high: for Synthetic 1, the 95 percentile was 1.12 ($M$ jobs) and 1.09 ($L$ jobs) if using the slack-factor range $[1.05,1.2]$, and 1.11 ($M$ jobs) and 1.19 ($L$ jobs) if using the slack-factor range $[1.2,1.5]$. The maximum prediction errors were 1.32/1.27 and 1.39/1.37. Thus, obviously the larger slack-factor range creates slightly more deviation from the predicted performance, i.e., there exists a trade-off between predictability and performance.

---

| Workload  | Conservative | Slack-based 1.05 - 1.15 | Slack-based 1.2 - 1.5 | Easy |
|-----------|--------------|-------------------------|----------------------|------|
| Synthetic 1 | 5.5          | 4.8                     | 3.8                  | 6.2  |
| Synthetic 2 | 6.0          | 5.8                     | 4.8                  | 6.4  |
| CM5       | 4.5          | 4.4                     | 4.0                  | 4.6  |
| BLUE      | 5.0          | 4.4                     | 4.1                  | 4.7  |

2 These evaluation do not, however, consider swap cost.
5. Summary and Conclusion

As a compromise between the commonly used conservative and easy backfilling, slack backfilling was proposed. Slack backfilling permits limited push backs of jobs and therefore retains much of the fairness of FCFS with conservative backfilling and of its predictability, as the results confirm. As expected, the performance of slack backfilling in regards to average response times and average bounded slowdowns lies between conservative and easy backfilling.

The presented slack-based approach is computationally relatively cheap to compute and easy to integrate into any scheduling policy, since only affecting the backfilling. Therefore, slack backfilling is a promising approach for general usage.

References

[1] Avi-Itzhak, B., Levy, H., and Raz, D., “Quantifying Fairness in Queuing Systems—Principles, Approaches, and Applicability”, Probability in the Engineering and Informational Sciences 22, 2008, pp. 495-517.

[2] Esbaugh, B., and Sodan, A.C., “Coarse-Grain Time Slicing with Resource-Share Control in Parallel-job Scheduling”, Proc. High Performance Computing and Communication (HPCC), Houston, LNCS 4782, Springer Verlag, Sept. 2007.

[3] Feitelson Workload Archive, http://www.cs.huji.ac.il/labs/parallel/workload/logs.html, retrieved May 2007.

[4] Islam, M., Balaji, P., Sadayappan, P., and Panda, D.K., “QoPS: A QoS based Scheme for Parallel Job Scheduling., Proc. JSSPP 2003, LNCS 2862, Springer, 2003.

[5] Jackson, D., Snell, Q., and Clement, M.J., “Core Algorithms of the Maui scheduler”, Proc. JSSPP 2001, LNCS 2221, Springer, 2001.

[6] Jin, W., “Backfilling with Fairness and Slack for Parallel Job Scheduling”, Master Thesis, University of Windsor, September 2009.

[7] Kettimuthu, R., Subrasani, V., Srinivasan, S., and Gopalan, T., “Selective Preemption Strategies for Parallel Job Scheduling”, Proc. Int. Conf. on Par. Proc. (ICPP), 2002.

[8] Lifka, D.A., “The ANL/IBM SP Scheduling System”, Proc. JSSPP, LNCS 949, Springer, 1995.

[9] Lublin, U., and Feitelson, D.G., “The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs”, Journal of Parallel and Distributed Computing, 2003, pp. 1105-1122.

[10] Mu’alem, A., and Feitelson, D.G., “Utilization, Predictability, Workloads, and User Runtimes Estimates in Scheduling the IBM SP2 with Backfilling”, IEEE Trans. on Parallel and Distributed Systems 12(6), June 2001.

[11] Raz, D., Levy, H., and Avi-Itzhak, B., “A Resource-Allocation Queueing Fairness Measure”, Proc. SIGMETRICS/Performance, New York, June 2004.

[12] Sabin, G., and Sadayappan, P., “Unfairness Metrics for Space-Sharing Parallel Job Schedulers”, Proc. JSSPP Workshop, Cambridge, Springer, LNCS 3834, June 2005.

[13] Schwiegelshohn, U., and Yahyapour, R., “Fairness in Parallel Job Scheduling”, Journal of Scheduling 3(5), 2000, pp. 297-320.

[14] Shmueli, E., and Feitelson, D.G., “Backfilling with Lookahead to Optimize the Packing of Parallel Jobs”, J. of Parallel and Distributed Computing 65, 2005, pp. 1090-1107.

[15] Sodan, A.C., Kanavallil, A., and Esbaugh, B., “Group-Based Optimization for Parallel Job Scheduling with Scojo-PECT-O”, Proc. HPCS, Quebec City, IEEE, June 2008.

[16] Subramaniam, R., “A Framework for Parallel Job Scheduling”, Ph.D. thesis, University of California, Irvine, 1995.

[17] Talby, D., and Feitelson, D., “Supporting Priorities and Improving Utilization of the IBM SP2 Scheduler Using Slack Based Backfilling”, Proc. IPPS/SPDP, 1999.

[18] B. van Houten, F. Ciesielski, G. Brehmer, G. de Jacquelot, M. Riedmann, and H. Strauss, “Advanced Cluster Software for Meteorology”, ECMWF workshop on Use of HPC in Meteorology.
Reading/UK, Nov. 2006, available at http://www.ecmwf.int/newsevents/meetings/workshops/2006/high_performance_computing-12th/pdf/Henry_Strauss.pdf.

[19] Ward, W., Mahood, C.L., and West, J.E., “Scheduling Jobs on Parallel Systems Using a Relaxed Backfill Strategy”, *Proc. JSSPP*, Springer, LNCS 2537, 2002.