The resource Usage Aware Backfilling

F. Guim, I. Rodero and J. Corbalan [⋆]

Computer Architecture Department, Technical University of Catalonia (UPC), Spain

Abstract. Job scheduling policies for HPC centers have been extensively studied in the last few years, especially backfilling based policies. Almost all of these studies have been done using simulation tools. All the existent simulators use the runtime (either estimated or real) provided in the workload as a basis of their simulations. In our previous work we analyzed the impact on system performance of considering the resource sharing (memory bandwidth) of running jobs including a new resource model in the Alvio simulator. Based on this studies we proposed the *LessConsume* and *LessConsume Threshold* resource selection policies. Both are oriented to reduce the saturation of the shared resources thus increasing the performance of the system. The results showed how both resource allocation policies shown how the performance of the system can be improved by considering where the jobs are finally allocated.

Using the *LessConsume Threshold* Resource Selection Policy, we propose a new backfilling strategy : the *Resource Usage Aware Backfilling* job scheduling policy. This is a backfilling based scheduling policy where the algorithms which decide which job has to be executed and how jobs have to be backfilled are based on a different *Threshold* configurations. This backfilling variant that considers how the shared resources are used by the scheduled jobs. Rather than backfilling the first job that can moved to the run queue based on the job arrival time or job size, it looks ahead to the next queued jobs, and tries to allocate jobs that would experience lower penalized runtime caused by the resource sharing saturation.

In the paper we demostrate how the exchange of scheduling information between the local resource manager and the scheduler can improve substantially the performance of the system when the resource sharing is considered. We show how it can achieve a close response time performance that the shortest job first Backfilling with First Fit (oriented to improve the start time for the allocated jobs) providing a qualitative improvement in the number of killed jobs and in the percentage of penalized runtime.

1 Introduction

Several works focused on analyzing job scheduling policies have been presented in the last decades. The goal was to evaluate the performance of these policies with specific workloads in HPC centers. A special effort has been devoted to evaluating backfilling-based ([4][22]) policies because they have demonstrated an ability to reach the best performance results (i.e: [12] or [21]). Almost all of these studies have been done using

[⋆] This paper has been supported by the Spanish Ministry of Science and Education under contract TIN200760625C0201 and by the IBM/BSC MareIncognito project under the grant BES-2005-7919.

simulation tools. To the best of our knowledge, all the existent simulators use the runtime (either estimated or real) provided in the workload as a basis of their simulations. However, the runtime of a job depends on runtime issues such as the specific resource selection policy used or the resource jobs requirements.

In [15] we evaluated the impact of considering the penalty introduced in the job runtime due to resource sharing (such as the memory bandwidth) in system performance metrics, such as the average bounded slowdown or the average wait time, in the backfilling policies in cluster architectures. To achieve this, we developed a job scheduler simulator (Alvio simulator) that, in addition to traditional features, implements a job runtime model and resource model that try to estimate the penalty introduced in the job runtime when sharing resources. In our previous work and we only considered in the model the penalty introduced when sharing the memory bandwidth of a computational node. Results showed a clear impact of system performance metrics such as the average bounded slowdown or the average wait time. Furthermore, other interesting collateral effects such as a significant increment in the number of killed jobs appeared. Moreover the impact on these performance metrics was not only quantitative.

Using the conclusions reached in our preliminary work, in [16] we described two new resource selection policies that are designed to minimize the saturation of shared resources. The first one, the *LessConsume* attempts to minimize the job runtime penalty that an allocated job will experience. It is based on the utilization status of shared resources in the current scheduling outcome and the job resource requirements. The second once, the *LessConsume Threshold*, finds an allocation for the job that satisfies the condition that the estimated job runtime penalty factor is lower than a given value *Threshold*. This resource selection policy was designed to provide a more sophisticated interface between the local resource manager and the local scheduler in order to find the most appropriate allocation for a given job. Both resource allocation policies showed how the performance of the system can be improved by considering where the jobs are finally allocated. They showed a very important improvement in the percentage of penalized runtimes of jobs due to resource sharing saturation, and more importantly, in the number of killed jobs. Both have reduced by four or even six times the number of killed jobs versus the traditional resource selection policies.

In this paper we propose a new backfilling strategy: the *Resource Usage Aware Backfilling* job scheduling policy. This is a backfilling based scheduling policy where the algorithms which decide which job has to be executed and how jobs have to be backfilled are based on a different *Threshold* configurations. In brief, this backfilling variant is based on the Shortest-Backfilled First backfilling variant. Rather than backfilling the first job that can be moved to the run queue, it looks ahead to the next queued jobs, and tries to allocate jobs that would experience lower penalty factors. However, it also takes into consideration the expected response time of the jobs that it evaluates during the backfilling process. The presented paper uses the model described in our previous work where the memory usage is considered.

The rest of the paper is organized as follows: section 2 presents the related work; section 3 briefly introduces the resource and runtime models that we proposed; next, the *LessConsume* Threshold resource selection policies are described; the RUA-Backfilling is presented in section 5; in section 6 we present an evaluation of different scheduling configurations; and finally, in section 9 we present the conclusions of this work.

2 Related work

Authors like Feitelson, Schwiegelshohn, Calzarossa, Downey or Tsafrir have modeled logs collected from large scale parallel production systems. They have provided inputs for the evaluation of different system behavior. Such studies have been fundamental since they have allowed an understanding how the HPC centers users behave and how the resources of such centers are being used. Feitelson has presented several works concerning this topic, among others, he has published papers on log analysis for specific centers [10], general job and workload modeling [8][11][9], and, together with Tsafrir, papers on detecting workload anomalies and flurries [25]. Calzarossa has also contributed with several workload modellization surveys [1][2]. Workload models for moldable jobs have been described in works of authors like Cirne et al. in [5][6], by Sevcik in [18] or by Downey in [7]. These studies have been considered in the design of new scheduling strategies.

From the early nineties, local scheduling architectures and policies have been one of the main goals of research in the area of high performance computing. Backfilling policies have been deployed in the major HPC centers. A backfilling scheduling policy is an optimization of the simplest scheduling algorithm: First-Come-First-Serve (FCFS). It starts jobs that have arrived later than the job at the head of the wait queue if the estimated start time of this job is not delayed. Typically, this is called a reservation for the first job. This backfilling is the most basic backfilling policy proposed by Lifka et al. in [20] and it is called EASY-Backfilling. Many variants of this first proposal have been described in several papers. The differences between each of them can be identified as follows:

- The order in which the jobs are backfilled from the wait queue: in the EASY variant the jobs are backfilled in arrival order, other variants have proposed backfilling the jobs in shortest job first order (Shortest-Job-Backfilled-First [23][22]). More sophisticated approaches propose dynamic backfilling priorities based on the current wait time of the job and the job size (LXWF-Backfilling [4]).
- The order in which the jobs are moved to the head of the wait queue, i.e.: which job is moved to the reservation. Similar to backfilling priorities, in the literature many papers have proposed pushing the job to the reservation in FCFS priority order or using the LXWF-Backfilling order.
- The number of reservations that the scheduler has to respect when backfilling jobs. The EASY variant is the most aggressive backfilling since the number of reservations is 1. As a result, in some situations the start time for the jobs that are queued behind the head job may experience delays due to the backfilled jobs. More conservative approaches propose that none of the queued jobs are delayed for a backfilling job. However, in practice, this last kind of variant is not usually used in real systems.

General descriptions of the most frequently used backfilling variants and parallel scheduling policies can be found in the report that Feitelson et al. provide in [12]. Moreover, a deeper description of the conservative backfilling algorithm can be found in [21], where the authors present a characterization and explain how the priorities can be used to select the appropriate job to be scheduled.

Backfilling [20] policies have been the main goal of study in recent years. As with research in workload modeling, authors like Frachtenberg have provided the community with many works regarding this topic. In [12] general descriptions of the most commonly used backfilling variants and parallel scheduling policies are presented. Moreover, a deeper description of the conservative backfilling algorithm can be found in [21], where the authors present policy characterizations and how the priorities can be used when choosing the appropriate job to be scheduled. Other works are [13] and [4].

More complex approaches have been also been proposed by other researchers. For instance, in [17] the authors propose maintaining multiple job queues which separate jobs according to their estimated run time, and using a backfilling aggressive based policy. The objective is to reduce the slowdown by reducing the probability that short job is queued behind a long job. Another example is the optimization presented by Shmueli et al. in [19] which attempts to to maximize the utilization using dynamic programming to find the best packing possible given the system status.

3 The Runtime Model

In this section we provide a brief characterization for the runtime model that we designed for evaluate the resource sharing in the Alvio simulator. The main goal of this simulator is to model the different scheduling entities and computing resrouces that are included in the current HPC architectures. Despite the simulator allows to simulate distributed systems, in the work presented in this paper only one HPC center is considered. The accesses to the HPC resources are controlled by two different software components: the Job Scheduler and the Local Resources Manager. The figure 1 provides a general overview of the different elements that are involved in a HPC system and their relations. As can be observed, these computational resources are composed by a set of physical resources (the processors, the memory, the I/O system etc.) that are managed by the local scheduler and the local resource manager (LRM). The local scheduler has the responsibility of scheduling the jobs that the users submit and the local resource manager has the responsibility to control the access to the physical resources.

In [15] we present a detailed description of the model and its evaluation.

3.1 The Job Scheduling Policy

The job scheduling policy uses as input a set of job queues and the information provided by the local resource manager (LRM) that implements a Resource Selection Policy (RSP). It is responsible to decide which of the jobs that are actually waiting to be executed have to be allocated to the free resources. To do this, considering the amount of free resources it selects the jobs that can run and it requires to the LRM to allocate the job processes.

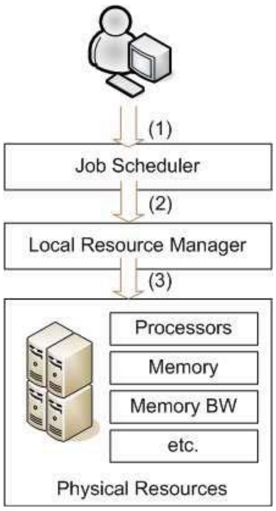


Fig. 1: The local scheduler internals

3.2 The Resource Selection Policy

The Resource Selection Policy, given a set of free processors and a job α with a set of requirements, decides to which processors the job will be allocated. To carry out this selection, the RSP uses the Reservation Table (RT, see figure 2). The RT represents the status of the system at a given moment and is linked to the architecture. The reservation table is a bi dimensional table where the X axes represent the time and the Y axes represent the different processors and nodes of the architecture. It has the running jobs allocated to the different processors during the time. One allocation is composed of a set of buckets[1] that indicate that a given job α is using the processors $\{p_0,..,p_k\}$ from *start time* until *end time*.

An allocation is defined by: $allocation\{\alpha\} = \left\{[t_0,t_1], P = \left\{p_{\{g,n_h\}},..p_{\{s,n_t\}}\right\}\right\}$ and indicates that the job α is allocated to the processors P from the time t_0 until t_1. The allocations of the same processors must satisfy that they are not overlapped during the time.

Figure 3 provides an example of a possible snapshot of the reservation table at the point of time t_1. Currently, there are three jobs running in three different job allocations:
$a_1 = \left\{[t_0,t_2], \left\{p_{\{1,node_1\}}, p_{\{2,node_1\}}, p_{\{3,node_1\}}\right\}\right\}$
$a_2 = \left\{[t_1,t_3], \left\{p_{\{4,node_1\}}, p_{\{5,node_1\}}, p_{\{1,node_2\}}, p_{\{2,node_2\}}\right\}\right\}$
$a_3 = \left\{[t_1,t_4], \left\{p_{\{5,node_2\}}\right\}\right\}$

[1] The $b_{(i,t_{i_0},t_{i_1})}$ bucket is defined as the interval of time $[t_x,t_y]$ associated to the processor p_i

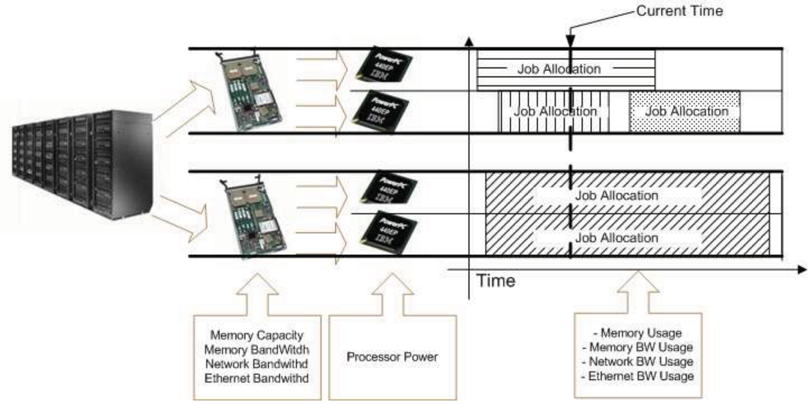


Fig. 2: The reservation table

3.3 Modeling the conflicts

The model that we have presented in the previous subsection has some properties that allow us to simulate the behavior of a computational center with more details. Different resource selection policies can be modeled. Thanks to the Reservation Table, it knows at each moment which processors are used and which are free.

Using the resource requirements for all the allocated jobs, the resource usage for the different resources available on the system is computed. Thus, using the Reservation Table, we are able to compute, at any point of time, the amount of resources that are being requested in each node.

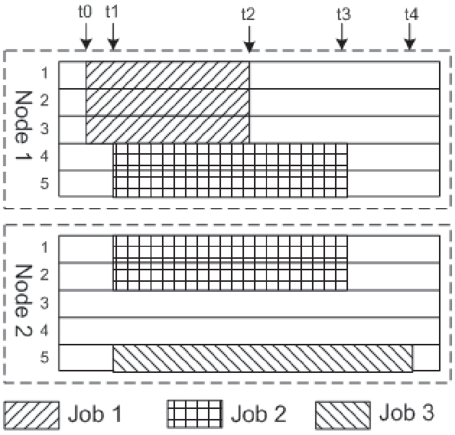


Fig. 3: Reservation Table Snapshot

In this extended model, when a job α is allocated during the interval of time $[t_x, t_y]$ to the reservation table to the processors $p_1, .., p_k$ that belong to the nodes $n_1, .., n_j$, we check if any of the resources that belong to each node is saturated during any part of the interval. In the affirmative case a runtime penalty will be added to the jobs that belong to the saturated subintervals. To model these properties we defined the Shared Windows and the penalty function associated to it.

The Shared Windows A *Shared Window* is an interval of time $[t_x, t_y]$ associated to the node n where all the processors of the node satisfy the condition that: either no process is allocated to the processor, or the given interval is fully included in a process that is running in the processor.

The penalty function This function is used to compute the penalty that is associated with all the jobs included to a given Shared Window due to resources saturation. The input parameters for the function are:

- The interval associated to the Shared Window $[t_x, t_y]$.
- The jobs associated to the Shared Window$\{\alpha_0, .., \alpha_n\}$
- The node n associated to the Shared Window with its physical resources capacity.

The function used in this model is defined as [2]:

$$\forall res \in resources(n) \rightarrow demand_{res} = \sum_{\alpha}^{\{\alpha_0,...,\alpha_n\}} r_{\alpha,res} \tag{1}$$

$$Penalty = \sum_{resources(n)}^{res} \left(\frac{\max(demand_{res}, capacity_{res})}{capacity_{res}} - 1\right) \tag{2}$$

$$PenlizedTime = (t_y - t_x) * Penalty \tag{3}$$

First for each resource in the node the resource usage for all the jobs is computed. Second, the penalty for each resource consumption is computed. This is a linear function that depends on the saturation of the used resource. Thus if the amount of required resource is lower than the capacity the penalty will be zero, otherwise the penalty added is proportional to the fraction of demand and availability. Finally, the penalized time is computed by multiplying the length of the Shared Window and the penalty. This penalized time is the amount of time that will be added to all the jobs that belong to the Window corresponding to this interval of time. This model has been designed for the memory bandwidth shared resource and can be applicable to shared resources that behave similar. However, for other typology of shared resources, such as the network bandwidth, this model is not applicable. Our current work is focused on modeling the penalty model for the rest of shared resources of the HPC local scenarios that can impact in the performance of the system.

[2] Note that all the penalty, resources, resource demands and capacities shown in the formula refer to the node n and the interval of time $[t_x, t_y]$. Thereby, they are not specified in the formula

For compute the penalized time that is finally associated to all the jobs that are running: first, the shared windows for all the nodes and the penalized times associated with each of them are computed; second the penalties of each job associated with each node are computed adding the penalties associated with all the windows where the job runtime is included; and finally, the final penalty associated to the job is the maximum penalty that the job has in the different nodes where it is allocated.

4 The LessConsume Resource Selection Policies

Using the model that we have presented in the previous section we designed two new Resource Selection Policies. First, the *LessConsume* that attempts to minimize the job runtime penalty that an allocated job will experience. Based on the utilization status of the shared resources in current scheduling outcome and job resource requirements, the *LessConsume* policy allocates each job process to the free allocations in which the job is expected to experience the lowest penalties. Second, we designed the *LessConsume Threshold* selection policy which finds an allocation for the job that satisfies the condition that the estimated job runtime penalty factor is lower than a given value *Threshold*. It is a variant of the *LessConsume* policy and was designed to provide a more sophisticated interface between the local resource manager and the local scheduler in order to find the most appropriate allocation for a given job.

The core algorithm of the *LessConsume* selection policy is similar to the *First Fit* resource selection policy. This last one selects the first $\alpha_{\{CPUS,p\}}$ where the job can be allocated. However, in contrast to this previous algorithm, the *LessConsume* policy, once the base allocation is found, the algorithm computes the penalties associated with the different processes that would be allocated in the reservation. Thereafter it attempts to improve the allocation by replacing the selected buckets (used for create this initial allocation) that would have higher penalties with buckets that can be also selected, but that have not been evaluated. The *LessConsume* algorithm will iterate until the rest of the buckets have been evaluated or the penalty factor associated to the job is 1 (no penalty). [3]

In some situations this policy not only minimizes the penalized factor of the allocated jobs, but it also provides the same start times as the first fit allocation policy, which in practice provides the earliest possible allocation start time. However, in many situations the allocation policy of the lower penalty factor provides a start time that is substantially later than that achieved by a *First Fit* allocation. To avoid circumstances where the minimization of the penalty factor results in delays in the start time of scheduled jobs, we have designed the *LessConsume Threshold*. This is a parametrized selection policy which determines the maximum allowed penalty factor allocated to any given job.

In contrast to this first selection policy, the *LessConsume Threshold* policy allows the scheduler or to the administrator to specify the maximum desired penalty factor that the scheduler accepts for a given job. Thus, it is able to carry out the scheduling

[3] The penalty factor is computed:

$$PenaltyFactor_{\alpha} = \frac{\alpha_{\{RunTime,rt\}}+\alpha_{\{PenalizedRunTime,prt\}}}{\alpha_{\{RunTime,rt\}}}$$

decisions taking into account the resource sharing saturation and it is able to verify how the job response time is affected by different allocations of the job.

The main differences between the two policies is that the second one will stop the process of evaluating all selected buckets when the penalty of the job is lower than the provided *Threshold*. Thus, in some situations this resource selection policy will return an allocation that has a higher penalty that the once that would have returned the *LessConsume* policy, however with a earlier start time. This policy provides the trade off to the scheduler to balance the benefits of delaying the job start time an obtaining a lower threshold, or advancing it and having a higher penalty.

5 The RUA-Backfilling

The *LessConsume Threshold* resource selection policy has been mainly designed to be deployed in two different scenarios. In the first case, the administrator of the local scenario specifies in the configuration files the penalty factor of a given allocated job. This factor could be empirically determined by an analytical study of the performance of the system. In the second, more plausible, scenario, the local scheduling policy is aware of how this parameterized RSP behaves and how it can be used by different factors. In this second case the scheduling policy can take advantage of this parameter to decide whether a job should start in the current time or whether it could achieve performance benefits by delaying its start time. In this last scenario the response time of a job can be improved in two different ways:

- Reducing the final runtime of the job by minimizing the penalty factor associated to the job.
- Reducing the wait time of the job by minimizing the start time of the job.

The Resource Usage Aware Backfilling Scheduling (RUA-Backfilling) policy takes into account both considerations when inspecting the wait queue for backfilling the jobs or finding the allocations for the reservations. In brief, this backfilling variant is based on the Shortest-Backfilled First backfilling variant. Rather than backfilling the first job that can moved to the run queue, it looks ahead to the next queued jobs, and tries to allocate jobs that would experience lower penalty factors. However, it also takes into consideration the expected response time of the jobs that it evaluates during the backfilling process.

The different parameters of the RUA-Backfilling are:

1. The number of ***reservation*** (number of the jobs in the queue whose estimated start time can not be delayed) is *1*.
2. The different ***Thresholds*** that will be used to calculate the appropriate allocation for the job that is moved from the reservation. In the evaluation the thresholds used by the policy are $RUA_{thresh} = \{1.15; 1.20; 1.25; 1.5\}$.
3. The jobs are moved from the wait queue to the reservation using the First Come First Serve priority. This priority assures that the jobs submitted to the system will not suffer starvation.

4. The backfilling queue is ordered using a dynamic criteria that is computed each time that the backfilling processes is required. It is described below.

When a job α has to be moved to the reservation the following algorithm is applied:

1. For each *Threshold* in the RUA_{thresh} specified in the configuration of the policy (in the presented evaluation $\{1.15; 1.20; 1.25; 1.5\}$):
   (a) The allocation based on the *LessConsume Threshold* resource selection policy with a parameter of $PenaltyFactor = Threshold$ is requested from the local resource manager.
   (b) The slowdown for the job is computed based on the start/wait times, the penalized runtime of the job in the returned allocation, the current wait time of the job and its requested runtime.
2. The allocation with less slowdown is selected to allocate the job. The local scheduler contacts the local resource manager to allocate the job in the given allocation.

Given the jobs that are queued in the backfilling queue, the backfilling algorithm behaves as follows:

1. In the first step, for each job α in the backfilling queue its allocation is computed based on the algorithm introduced in the previous paragraph. If the start time for the returned allocation is the current time the job is added to the backfilling queue where the allocations are ordered by the penalized factor associated to the allocation and secondly by its length. Note that each job has only one assigned allocation.
2. In the second step, the backfilling queue has all the jobs that could be backfilled in the current time stamp ordered in terms of the associated penalty. The queue is evaluated and the first job that can be backfilled is allocated to the reservation table using allocation computed in the previous step. Note that the allocation will be exactly the same as the one computed in the first step.
   (a) If no job can be backfilled the process of backfilling is terminated.
   (b) Otherwise, steps 1 and 2 will be iterated again.

The key concept of this backfilling variant is to find out the allocation that provides the best slowdown for the job that is moved to the reservation, and to backfill the jobs in the manner that the saturation of the shared resources is minimized. The second goal will reduce the number of killed jobs due to resource sharing saturation.

Note that in this algorithm the allocations are computed using the estimated runtime that is provided by the user. In the version of the policy evaluated in this scenario we have supposed that when a job is allocated with a penalty factor of $\alpha_{penalty}$, the estimated runtime is updated according this penalty. Based on our studies in prediction systems in backfilling policies [14], in our future versions of the RUA we plan to use the predicted runtime in the *LessConsume Threshold* and keep the original user requested runtime.

6 Experiments

In this section we characterize the different experiments that we defined in order to validate the performance of the different scheduling strategies that we propose.

6.1 Workloads

For the experiments we used the cleaned [24] versions of the workloads SDSC Blue Horizon (SDSC-BLUE) and Cornell Theory Center (CTC) SP2. For the evaluation experiments explained in the following section, we used the 10000 jobs of each workload plus 10000 jobs that were used in order to warm-up the system and achieve a steady state. Based on these workload trace files, we generated three variations for each one with different memory bandwidth pressure:

- HIGH: 80% of jobs have high memory bandwidth demand, 10% with medium demand and 10% of low demand.
- MED: 50% of jobs have high memory bandwidth demand, 10% with medium demand and 40% of low demand.
- LOW: 10% of jobs have high memory bandwidth demand, 10% with medium demand and 80% of low demand.

6.2 Architecture

For each of the workloads used in the experiments we defined architecture with nodes of four processors, 6000 MB/Second of memory bandwidth, 256 MB/Second of Network bandwidth and 16 GB of memory. In addition to the SWF [3] traces with the job definitions we extended the standard workload format to specify the resource requirements for each of the jobs. Currently, for each job we can specify the average memory bandwidth required (other attributes can be specified but are not considered in this work). Based on our experience and the architecture configuration described above, as a first approach we defined that a *low memory bandwidth demand* consumes 500 MB/Second per process; a *medium* memory bandwidth demand consumes 1000 MB/Second per process; and that a *high memory bandwidth demand* consumes 2000 MB/Second per process. These memory requirements were selected based on the typology of jobs that were running in our centers.

7 Scenarios

In the experiments we evaluate the impact of the RUA-Backfilling in the system. To do this, we compare its performance against the Shortest Job Backfilled First policy under the *LessConsume* resource selection policies. For the analysis of the RUA-Backfilling job scheduling policy the following configurations were evaluated:

1. The Shortest Job Backfilled First scheduling policy using:
   - The **LessConsume** resource selection policy.
   - The **LessConsume Threshold** resource selection policy with four different factors (*1*, *1,15*, *1,25* and *1,5*).
   - The First-Fit resource selection policy.
2. The RUA-Backfilling policy.

All the simulations have used the the job runtime model with resource sharing model introduced in the first part of this paper. In the rest of the section we analyze the different configurations that we have introduced: first, we provide a discussion concerning the differences between using the *LessConsume* and *LessConsume Thresolds* policies in the SJBF Backfilling variant. Next, we compare the performance of the SJBF Backfilling with the First-Fit and LessConsume resources selection policies against the results obtained using the RUA-Backfilling.

8 Evaluation

In this section we present the evaluation of the RUA-Backfilling job scheduling policy. However, in order to provide a characteriztation of the *LessConsume* resource selection policies, first we present their performance analysis. This analysis is used later on in the discussion of the RUA-Backfilling.

8.1 The LessConsume and LessConsume Threshold

Tables 1 and 2 present the 95_{th} percentile and average of the bounded slowdown for the CTC and SDSC centers for each of the three workloads for the *First Fit* (FF), *LessConsume* and *LessConsume Threshold* resource selection policy. The last one was evaluated with three different factors: *1*, *1,15*, *1,25* and *1,5*. In both centers the *LessConsume*policy performed better than the *LessConsume Threshold* with a factor of *1*. One could expected that the *LessConsume* should be equivalent to use the *LessConsume Threshold* with a threshold of *1*. However, note that this affirmation would be incorrect. This is caused due to the *LessConsume* policy evaluates all the buckets in a subset of all the possible allocations. The goal of this policy is to optimize the *First Fit* allocation but without carry out a deeper search of other possibilities. However, the *LessConsume Threshold* may look further in the future in the case that the penalty is higher than the provided threshold. Thereby, this last one is expected to provide higher wait time values. On the other hand, as we had expected, the bounded slowdown decreases while increasing the factor of the *LessConsume Threshold* policy. In general, the ratio of increment of using a factor of *1* and a factor of *1,5* is around a 20% in all the centers and workloads.

Center	MEM	FF	LC	LCT=1	LCT=1,15	LCT=1,25	LCT=1,5
CTC	High	4,2	5,94	7,92	6,12	5,32	5,23
	Med	2,8	3,55	4,22	3,82	3,65	3,52
	Low	2,2	3,12	3,62	3,82	3,45	3,52
SDSC	High	99,3	110,21	128,08	115,28	109,51	106,23
	Med	55,4	68,06	74,32	72,83	71,37	68,52
	Low	37,8	45,37	57,27	52,86	42,28	42,28

Table 1: Bounded-Slowdown - 95_{th} Percentile

Center	MEM	FF	LC	LCT=1	LCT=1,15	LCT=1,25	LCT=1,5
CTC	High	8,2	10,44	18,02	12,38	11,32	13,54
	Med	5,3	6,65	7,52	8,05	6,85	7,75
	Low	3,2	5,62	5,92	5,82	8,84	8,56
SDSC	High	22,56	24,41	27,37	25,54	24,26	23,53
	Med	11,32	12,51	14,76	14,46	14,17	13,6
	Low	7,54	7,8	9,08	9,5	8,47	8,27

Table 2: Bounded-Slowdown - Average

Center	MEM	FF	LC	LCT=1	LCT=1,15	LCT=1,25	LCT=1,5
CTC	High	10286	12588	17945	15612	14555	10188
	Med	8962	9565	13391	13123	9186	9094
	Low	4898	5034	6544	7198	5235	5759
SDSC	High	55667	63293	70964	69632	59978	41779
	Med	44346	45164	58713	59300	47440	45616
	Low	32730	35092	38265	37499	33374	33785

Table 3: Wait Time - 95_{th} Percentile

Center	MEM	FF	LC	LCT=1	LCT=1,15	LCT=1,25	LCT=1,5
CTC	High	20082	24576	35035	30480	28416	19890
	Med	13443	14347	20086	19684	13779	13641
	Low	7124	7322	9518	10469	7614	8376
SDSC	High	12647	14379	16122	15819	13626	9491
	Med	9061	9228	11996	12116	9693	9320
	Low	3931	4214	4595	4503	4008	4057

Table 4: Wait Time - average

The performance of these two resource policies, compared to the performance of the *First Fit* policy, shows that *LessConsume* policies give an small increment in the bounded slowdown. For instance, in the CTC high memory pressure workload the 95_{th} percentile of the bounded slowdown has increased from 4,2 in the *First Fit* to 5,94 in the *LessConsume* policy, or to 7,92 and 5,23 in the *LessConsume Threshold* with thresholds of *1* and *1,5* respectively.

Center	**MEM**	**FF**	**LC**	**LCT=1**	**LCT=1,15**	**LCT=1,25**	**LCT=1,5**
CTC	High	8,8	8,01	7,69	7,87	7,91	8,1
	Med	4,8	3,81	3,01	3,52	4,06	3,90
	Low	0,92	0,78	0,51	0,72	0,62	0,80
SDSC	High	11,8	11,33	8,31	10,37	11,58	11,64
	Med	6,7	6,01	4,70	4,85	5,64	5,96
	Low	1,4	1,03	0,75	0,81	0,94	1,19

Table 5: Percentage of Penalized Runtime - 95_{th} Percentile

Center	**MEM**	**FF**	**LC**	**LCT=1**	**LCT=1,15**	**LCT=1,25**	**LCT=1,5**
CTC	High	7,8	6,81	6,98	7,17	7,39	7,34
	Med	3,8	2,54	2,93	3,02	3,1	3,20
	Low	0,72	0,7	0,21	0,42	0,33	0,64
SDSC	High	15,1	10,32	7,41	11,73	10,85	12,32
	Med	10,2	7,2	4,70	4,85	5,64	5,96
	Low	5,2	4,53	2,56	3,11	4,58	4,23

Table 6: Percentage of Penalized Runtime - average

Tables 3 and 4 show the 95_{th} and average of the wait time for the CTC and SDSC centers for each of the three workloads for the *First Fit*, *LessConsume* and *LessConsume Threshold* resource selection policy. This performance variable shows similar pattern to the bounded slowdown. The *LessConsume* policy shows a better performance result that using the *LessConsume Threshold* with a factor of *1*.

The 95_{th} percenage of penalized runtime is presented in the table 5 and the average is shown in the table table 6. The penalized runtime clearly increases by incrementing the threshold. For instance, the 95_{th} Percentile of the percentage increases from 8,31 in the SDSC and the high memory pressure workload with a factor of *1* until 11,64 with a factor of *1,5*. The *LessConsume*, different from to the two previously described variables, shows similar values to the *LessConsume Threshold* with a factor of *1,5*. This percentage of penalized runtime was reduced with respect to the *First Fit* when using all the different factors in both centers.

Center	MEM	FF	LC	LCT=1	LCT=1,15	LCT=1,25	LCT=1,5
CTC	High	428	120	57	70	87	97
	Med	247	101	76	77	102	99
	Low	64	45	36	38	58	52
SDSC	High	475	105	87	130	127	130
	Med	255	89	76	79	103	145
	Low	51	34	22	27	33	41

Table 7: Number of Killed Jobs 95_{th} Percentile

The number of killed jobs is the performance variable that showed most improvement in all the memory pressure workloads. The number of killed jobs is qualitatively reduced with the *LessConsume Threshold* with a factor of *1*: for example with the high memory pressure workload and the CTC center, the number of killed jobs was reduced from 428 with the *First Fit* to 70. The other threshold factors also showed clear improvements; the number was halved. As to the *LessConsume* policy, the number of killed jobs was reduced by a factor of 4 compared to the *First Fit* and the high and medium memory pressure workloads of both centers.

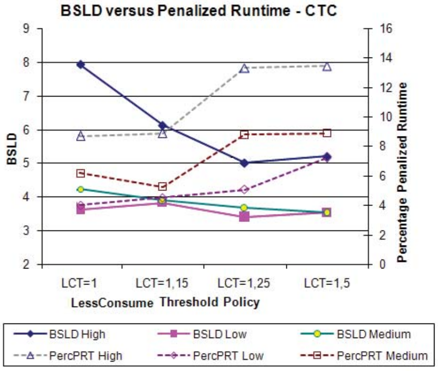


Fig. 4: BSLD versus Percentage of Penalized Runtime - CTC Center

The *LessConsume* policy shows how the percentage of penalized runtime and number of killed jobs can be reduced in comparison to the *First Fit*, by using this policy with EASY backfilling. In traditional scheduling architectures this RSP can be used rather than traditional policies, without any modifications in local scheduling policies. Furthermore, the *LessConsume* threshold shows how, with different thresholds, performance results can also be improved. Higher penalty factors result in better performance of the system. However, in this situation the number of killed jobs and the percentage of penalized runtime is increased. The *LessConsume* policy shows similar performance results as the *LessConsume Threshold* with factors of *1,25* and *1,5*.

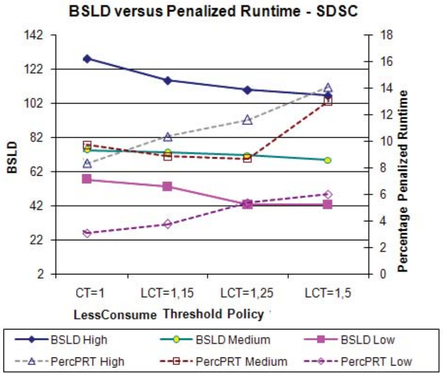


Fig. 5: BSLD versus Percentage of Penalized Runtime - SDSC Center

8.2 The thresholds trade offs

Figures 4, 5, 6 and 7 present the performance of the *LessConsume* policies (using bounded slowdown) against the percentage of penalized runtime of the jobs and the number of killed jobs. The goal of these figures is to show the chance that the *LessConsume Threshold* and *LessConsume* policies have to improve the performance of the system while achieving an acceptable level of performance. As can be observed in figures 6 and 4 a good balance is achieved in the CTC center using the threshold of *1,15* where both the number of killed Jobs and the percentage of penalized runtime converge are in acceptable values. In the case of the SDSC center, this point of convergence is not as evident as the CTC center. Considering the tendency of the bounded slowdown, it seems that the *LessConsume Threshold* with a factor of *1.15* is an appropriate configuration for this center, due to the fact that the penalized runtime and the number of killed jobs presents the lowest values, and the bounded slowdown shows values that are very close to the factors of *1,15* and *1,25*. However, the configuration of the *LessConsume Threshold* with a factor of *1,15* also shows acceptable values.

8.3 The RUA-Backfilling

In the previous subsections we have present the performance that the LessConsume resource selection policies achieve when they are used together with the SJBF-Backfilling variant. We have observed that the *LessConsume Threshold* resource selection policy can provide good results when the used threshold is between *1.15* and *1.25* depending on the workload. Using this results in the first RUA-Backfilling version shown in this

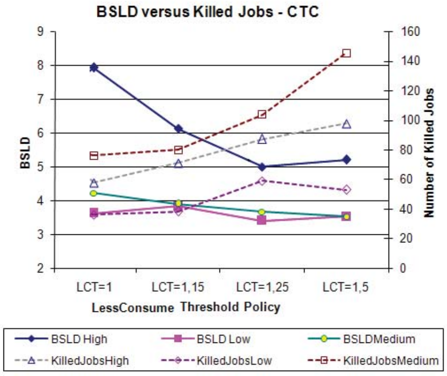


Fig. 6: BSLD versus Killed Jobs - CTC Center

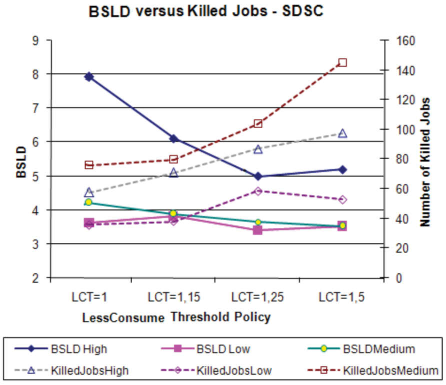


Fig. 7: BSLD versus Killed Jobs - SDSC Center

paper we decided to use threshold values presented in the section 5. In this section we present the benefits of the usage of a backfilling variant that interacts with the local resource manager against the traditional approaches.

The figures 9 and 8 present the performance that the RUA-Backfilling scheduling policy has achieved with respect the Shortest-Job-Backfilled First Backfilling (SJBF-Backfilling) with the *First Fit* and *LessConsume* Resource Selection Policies. The results shows also how each of the policies behaved with the three different memory pressure workloads for the SDSC and CTC workloads. The figure shows the 95_{th} Percentile of the BSLD, the Wait time and the Percentage of Penalized Runtime that the jobs have experimented and the number of killed jobs that three scheduling strategies have achieved in the simulations.

The bounded slowdown shows how in both workloads the RUA-Backfilling achieves slightly worst performance that the SJBF-Backfilling with the first fit selection policy.

In both cases the difference between the BSLD is less than a 5%. For instance, the 95_{th} Percentile of the BSLD with the SJBF-Backfilling in the SDSC workload with high memory pressure is 100 and with the RUA-Backfilling is around 110. Note that we could expect that this last one should achieve smaller BSLD that the once obtained by the SJBF-Backfilling with FF due to it takes into account the resource usage. However, in the RUA-Backfilling the number of jobs that are used for compute the BSLD (number of finished jobs) is substantially bigger that the once used in the other (400 less in the SJBF). Respect the SJBF-Backfilling with the *LessConsume* resource selection policy, the RUA-Backfilling shows in both workload better bounded slowdowns.

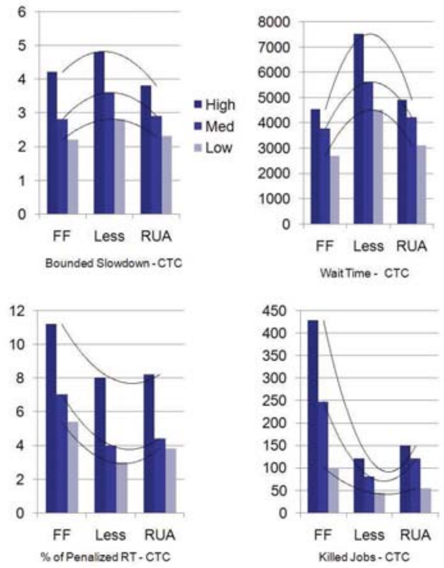
The wait time shows similar patterns that the Bounded Slowdown. However, the CTC workload shows higher differences between the SJBF-Backfilling and the other two strategies. For example, while the 95_{th} Percentile of wait time for the RUA-Backfilling and the SJBF-Backfilling with FF remains around 4000 and 5000 seconds in the high pressure scenario, the SJBF-Backfilling with *LessConsume* presents 95_{th} Percentile of the wait time around 7000. This, may indicate that the RUA Backfilling is more stable than using the *LessConsume* with a non resource usage aware scheduling strategy.

Finally, the number of killed jobs and the 95_{th} Percentile of percentage of penalized runtime show a qualitative improvement respect the SJBF-Backfilling with *First Fit*. For example, the RUA-Backfilling shows a reduction of a 500% in the number of killed jobs in the high memory pressure scenario of the SDSC workload and a reduction of 300% in the CTC scenario also with the high memory pressure scenario. Although the percentage of penalized run time shows an improvement in both center using the RUA-Backfilling, a higher improvement is shown in the SDSC center. For example, in this last case the percentage of penalized runtime is reduced a 50% in the workload with a medium memory pressure.

The RUA-Backfilling has demonstrated how the exchange of scheduling information between the local resource manager and the scheduler can improve substantially the performance of the system when the resource sharing is considered. It has shown how it can achieve a close response time performance that the SJBF-Backfilling with FF, that is oriented to improve the start time for the allocated jobs, providing a qualitative improvement in the number of killed jobs and in the percentage of penalized runtime. On the other hand, it has demonstrated how it can also obtain substantial improvement in these last two variables regarding the SJBF-Backfilling with *LessConsume* scheduling strategy, that is oriented to minimize the job runtime penalty due to resource saturation of the sharing resources.

9 Conclusions

In this paper we have shown how the performance of the system can be improved by considering resource sharing usage and job resource requirements in the new RUA Backfilling variant. In this proposal the local scheduler cooperates with the local resource manager in order to find out the allocation that minimizes the job runtime penalty due to the saturation of the resource sharing. This is a backfilling variant scheduling policy where the algorithms which decide which job has to be executed and how jobs have to be backfilled are based on a different configurations of the *LessConsume Threshold*



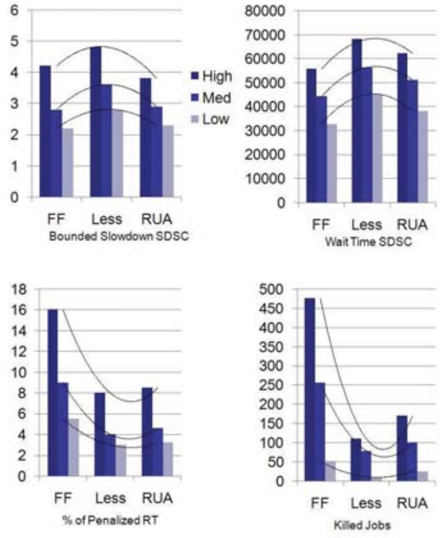
Bounded Slowdown - CTC

Wait Time - CTC

% of Penalized RT - CTC

Killed Jobs - CTC

Fig. 8: RUA Performance Variables for the CTC Workload



Bounded Slowdown SDSC

Wait Time SDSC

% of Penalized RT

Killed Jobs

Fig. 9: RUA Performance Variables for SDSC Workload

resource selection policy that we proposed in our previous work. In the first part of the paper we have introduced the key concepts of our previous works that are used in the RUA-Backfilling algorithm. First the runtime model used in our simulator, and second, the *Find LessConsume* and LessConsume Threshold resource selection policies.

In this paper we evaluate the effect of considering the memory bandwidth usage in the different scheduling strategies under several workloads. Two different workloads from the Standard Workload Archive have been used in the experiments (the SDSC Blue Horizon (SDSC-BLUE) and the Cornell Theory Center). For each of them we have generated three different scenarios: with high (HIGH), medium (MED), and low (LOW) percentage of jobs with high memory demand. We have evaluated the impact of using the *LessConsume* and *LessConsume Threshold* with the Shortest Job Backfilled first and the RUA-Backfilling presented in this paper. These synthetic workloads have been used as a first approach to evaluate the potential of the proposed techniques.

The RUA-Backfilling has demonstrated how the exchange of scheduling information between the local resource manager and the scheduler can improve substantially the performance of the system when the resource sharing is considered. It has shown how it can achieve a close response time performance that the SJBF-Backfilling with FF, that is oriented to improve the start time for the allocated jobs, providing a qualitative improvement in the number of killed jobs and in the percentage of penalized runtime. On the other hand, it has demonstrated how it can also obtain substantial improvement in these last two variables regarding the SJBF-Backfilling with *LessConsume* scheduling strategy, that is oriented to minimize the job runtime penalty due to resource saturation of the sharing resources.

Concerning the penalty model used in our system, our future work will consider how other shared resources may impact in the performance of the system. Clearly, the penalty function that has been presented in our model, has to be extended for consider penalties that other typologies of resource may show. For instance, the network bandwidth shows patterns in the job execution that are not considered in the penalty function. On the other hand, our future research will evaluate the impact of having inaccurate estimations in the job resource sharing requirements. Related to this, we will work in the usage of prediction techniques in order to estimate the resource requirements of the submitted jobs.

The RUA-Backfilling that we have presented in this paper uses a set of pre-configured *Threshold* for finding out the job allocations. In our research we have stated that the workloads can show very different load patterns during the time. Thus, depending of the epoch, the system may experiment better performance using different *Threshold* values. Considering this phenomena we will extend the RUA in order to dynamically determine which factors should be used in each scheduling moment. The first step will be studying the correlation of the system performance against the load of the system and *Threshold* configuration. Afterward, we will use this information for extend the current RUA-Policy policy.

References

1. M. Calzarossa, G. Haring, G. Kotsis, A. Merlo, and D. Tessera. A hierarchical approach to workload characterization for parallel systems. *Performance Computing and Networking,*

Lect. Notes Comput. Sci. vol., page pp. 102109, 1995.

2. M. Calzarossa, L.Massari, , and D. Tessera. Workload characterization issues and methodologies. *In Performance Evaluation: Origins and Directions, Lect. Notes Comput. Sci.*, page pp. 459482, 2000.
3. S. J. Chapin, W. Cirne, D. G. Feitelson, J. P. Jones, S. T. Leutenegger, U. Schwiegelshohn, W. Smith, and D. Talby. Benchmarks and standards for the evaluation of parallel job schedulers. *Job Scheduling Strategies for Parallel Processing*, pages 66–89, 1999. LNCS 1659.
4. S.-H. Chiang, A. C. Arpaci-Dusseau, and M. K. Vernon. The impact of more accurate requested runtimes on production job scheduling performance. *8th International Workshop on Job Scheduling Strategies for Parallel Processing*, Vol. 2537:103 – 127, 2002.
5. W. Cirne and F. Berman. A comprehensive model of the supercomputer workload. *4th Ann. Workshop Workload Characterization*, 2001.
6. W. Cirne and F. Berman. A model for moldable supercomputer jobs. *15th Intl. Parallel and Distributed Processing Symp.*, 2001.
7. A. B. Downey. A parallel workload model and its implications for processor allocation. *6th Intl. Symp. High Performance Distributed Comput.*, Aug 1997.
8. D. G. Feitelson. Packing schemes for gang scheduling'. *Job Scheduling Strategies for Parallel Processing*, LNCS 1162:89–110, 1996.
9. D. G. Feitelson. Workload modeling for performance evaluation. *In Performance Evaluation of Complex Systems: Techniques and Tools, Lect. Notes Comput. Sci. vol. 2459*, pages pp. 114–141, 2002.
10. 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, Lect. Notes Comput. Sci.*, vol. 949:pp. 337–360, 1995.
11. D. G. Feitelson and L. Rudolph. Metrics and benchmarking for parallel job scheduling. *In Job Scheduling Strategies for Parallel Processing, Lect. Notes Comput. Sci.*, vol. 1459:pp. 1–24, 1998.
12. D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling - a status report. *Job Scheduling Strategies for Parallel Processing: 10th International Workshop, JSSPP 2004*, 3277 / 2005:9, June 2004.
13. D. G. Feitelson and A. Weil. Utilization and predictability in scheduling the ibm sp2 with backfilling. *Proceedings of the 12th. International Parallel Processing Symposium*, pages 542–546, 1998.
14. F. Guim and J. Corbalan. Prediction f based models for evaluating backfilling scheduling policies. *The 8th International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2007.
15. F. Guim, J. Corbalan, and J. Labarta. Modeling the impact of resource sharing in backfilling policies using the alvio simulator. *15th Annual Meeting of the IEEE / ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2007.
16. F. Guim, J. Corbalan, and J. Labarta. Resource sharing usage aware resource selection policies for backfilling strategies. *The 2008 High Performance Computing and Simulation Conference*, 2008.
17. B. G. Lawson and E. Smirni. *Multiple-Queue Backfilling Scheduling with Priorities and Reservations for Parallel Systems*. Springer Verlag, 2002. Lect. Notes Comput. Sci. vol. 2537.
18. K. C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Performance Evaluation*, pages pp. 107–140, 1994.
19. E. Shmueli and D. G. Feitelson. *Backfilling with Lookahead to Optimize the Performance of Parallel Job Scheduling*. Springer Verlag, 2003. Lect. Notes Comput. Sci. vol. 2862.

20. J. Skovira, W. Chan, H. Zhou, and D. A. Lifka. The easy - loadleveler api project. *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, Lecture Notes In Computer Science; Vol. 1162 archive:41 – 47, 1996.
21. D. Talby and D. Feitelson. Supporting priorities and improving utilization of the ibm sp scheduler using slack-based backfilling. *Parallel Processing Symposium*, pages pp. 513–517, 1999.
22. D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling using runtime predictions rather than user estimates. *Technical Report 2005-5, School of Computer Science and Engineering, The Hebrew University of Jerusalem.*, 2005.
23. D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE TPDS*, 2006.
24. D. Tsafrir and D. G. Feitelson. Workload flurries. Technical report, School of Computer Science and Engineering and The Hebrew University of Jerusalem, 2003.
25. D. Tsafrir and D. G. Feitelson. Instability in parallel job scheduling simulation: the role of workload flurries. *In 20th Intl. Parallel and Distributed Processing Symp*, 2006.