Resource Sharing Usage Aware Resource Selection Policies for Backfilling Strategies

F. Guim [1], J. Corbalan [2] and J. Labarta [3]

Barcelona Supercompuning Center
Jordi Girona 13, Barcelona, Spain
[1]`francesc.guim@bsc.es`
[2]julita.corbalan@bsc.es [3]jesus.labarta@bsc.es

KEYWORDS

Resource Selection Policies, Resource Sharing Consideration, Backfilling, Job Scheduling

1 ABSTRACT

Job scheduling policies for HPC centers have been extensively studied in the last few years, specially 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 of running jobs including a new resource model in the Alvio simulator.

In this paper we present two new Resource Selection Policies that have been designed using the conclusions reached in our preliminary work. First, the *Find Less Consume Distribution* 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 have also described the *Find Less Consume Threshold Distribution* 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*.

2 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 (Chiang et al. (2002)Tsafrir et al. (2005)) policies because they have demonstrated an ability to reach the best performance results (i.e: Feitelson et al. (2004) or Talby and Feitelson (1999)). Almost all of these studies have been done using 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 Guim et al. (2007) 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 this work we have 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.

In this paper we describe two new resource selection policies that are designed to minimize the saturation of shared resources. The first one, the *Find Less Consume Allocation* (henceforth referred to as *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 *Find Less Consume Threshold Distribution* (henceforth referred to as *LessConsumeThreshold*) , 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 has been 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 rest of the paper is organized as follows: section 3 briefly introduce the resource and runtime models that we proposed; next, the two resource selection policies we propose are described; in section 6 we present their evaluation; and finally in section 7 we present the conclusions of this work.

Proceedings of the 2008 High Performance
Computing & Simulation Conference ©ECMS
Waleed W. Smari (Ed.)
ISBN: 978-0-9553018-7-2 / ISBN: 978-0-9553018-6-5 (CD)

3 MODELING RUNTIME PENALTY IN ALVIO

In this section we provide a brief characterization for the runtime model that we designed for evaluate the resource sharing in the Alvio simulator. In Guim et al. (2007) 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 witting 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.

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). 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: $allocation\{\alpha\} = \{[t_0,t_1], P = \{p_{\{g,n_h\}},..p_{\{s,n_t\}}\}\}$ 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.

3.3 Modeling the conflicts

The model that we have presented in the previous section has some properties that allows 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.

[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

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 overloaded during any part of the interval. In the affirmative case a runtime penalty will be added to the jobs that belong to the overloaded subintervals. To model this properties we defined the Shared Shadows 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 overload. 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]:

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

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

$$PenlizedTime = (t_y - t_x) * Penalty \quad (3)$$

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 overload 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 the node penalized time to all the jobs that belong to the Window. This model has been designed for the

[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

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. Future work will be focused on modelizing the penalty model for the rest of shared resources of the HPC local scenarios that can impact in the performance of the system.

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.

4 THE LESSCONSUME RESOURCE SELECTION POLICY

The core algorithm of this selection policy is similar to the 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]

The LessConsume policy, given a required start time t_{req} and given the job α with a requested runtime of $\alpha_{\{RunTime,rt\}}$ and number of requested processors $\alpha_{\{CPUS,p\}}$, finds in the reservation table the $\alpha_{\{CPUS,p\}}$ processor allocation that tries to minimize the job runtime penalties due to resource sharing saturation closest to the t_{req}. To do this, the selection policy follows these steps:

1. For each processor p_i in the reservation table, the resource selection policy selects the interval of time $[t_{x_i}, t_{y_i}]$ that is closest to the t_{req} that satisfies it: no process is allocated to the processor during the interval and its length is equal or greater to $\alpha_{\{RunTime,rt\}}$. All the buckets associated to the selected intervals of time are added to the set *Buckets* where they are strictly ordered by the interval start time.

2. Given all the different buckets $\left\{b_{(1,t_{1_0},t_{1_1})},..,b_{(N,t_{N_0},t_{N_1})}\right\}$ associated with the reservation table that are included in the set *Buckets*, the resource selection policy will select the first $\alpha_{\{CPUS,p\}}$ buckets that satisfy the condition that their interval of time shares at least the runtime required by the job.

3. In the case that in step *2* the number of buckets that satisfied the required conditions was lower than the required processors, this implies that there were not enough buckets which shared the required amount of time. In these situations, the first bucket $b_{(i,t_{i_0},t_{i_1})}$ with start time greater than to t_{req} is selected, the t_{req} is updated as $t_{req} = t_{i_0}$ and the steps *1*, *2* and *3* are iterated again.
At this point, from the buckets obtained in the first step we will have three different subsets:
   - The buckets $\Pi_{disc} = \left\{b_{(k,t_{k_0},t_{k_1})},..,b_{(l,t_{l_0},t_{l_1})}\right\}$ that have already been selected since they cannot form part of a valid allocation for the specified requirements.
   - The buckets $\Pi_{sel} = \left\{b_{(m,t_{m_0},t_{m_1})},..,b_{(n,t_{n_0},t_{n_1})}\right\}$ that have been selected for the base allocation and that conform to a valid allocation which satisfy the requirements for the job. In the case that the penalties of this allocation cannot be improved by a valid allocation, this set of buckets will be used.
   - The buckets $\Pi_{toCh} = \left\{b_{(o,t_{o_0},t_{o_1})},..,b_{(p,t_{p_0},t_{p_1})}\right\}$ that have not already been evaluated. These buckets will be used to try to improve the base allocation. Thus, the LessConsume policy will try to find out if any of these buckets could replace any of the already selected buckets reducing the estimated penalty factor of the job.

4. For each of the buckets in the set of the selected buckets Π_{sel}, the algorithm checks the estimated penalty that a process of the given job would achieve if it were allocated to the given bucket. Each of the selected buckets has an associated penalty factor. If all the buckets have an associated penalty of *1* the allocation definition based on these buckets is defined and returned. Otherwise, the *last valid allocation* is initialized based on this set of buckets and the selection process goes ahead.

5. For each bucket $b_{(r,t_{r_0},t_{r_1})}$ in the set of buckets Π_{toCh}:
   (a) If the number of buckets in the set Π_{sel} is lower than the number of requested processors, the bucket is added to the set and the next iteration continues.
   (b) Similar to the previous step, the algorithm evaluates the penalty $penalty_r$ that a process of the job would have if this bucket were used to allocate it.
   (c) The bucket $bMax_{(p,t_{p_0},t_{p_1})}$ of the set Π_{sel} with the highest penalty $penalty_p$ is selected.

[3]The penalty factor is computed:
$PenaltyFactor_{\alpha} = \frac{\alpha_{\{RunTime,rt\}}+\alpha_{\{PenalizedRunTime,prt\}}}{\alpha_{\{RunTime,rt\}}}$

(d) In case that the penalty that $penalty_r$ is lower than the penalty $penalty_p$, on one hand the bucket $b_{(r,t_{r_0},t_{r_1})}$ is added to the set Π_{sel} and removed from the set Π_{toCh}. On the other hand, the bucket $bMax_{(p,t_{p_0},t_{p_1})}$ is removed from the Π_{sel} and added to the set Π_{disc}. Note that at this point the inserted bucket may not share the interval required to allocate the job to the rest of the buckets of the set Π_{sel}.

   i. The buckets of the set Π_{sel} that do not share the required time are removed from this set and added to the Π_{disc}.

   ii. If the number of buckets of the Π_{sel} it is the number of requested processors *the last valid allocation* is built based on this set. If the current penalty of all the buckets is 1, the current set of buckets is returned as the allocation. Otherwise the algorithm goes ahead with the next iteration to evaluate the next bucket of the set Π_{toCh}.

6. The *last valid allocation* is returned.

As an example, suppose that the current scheduling outcome is the one presented in the figure 1 and that an allocation has to be found for a job with requested processors $\alpha_{\{CPUS,3\}}$ and runtime $\alpha_{\{RunTime,20secs\}}$.

1. Firtsly, in the step *1* the LessConsume algorithm would define the set $\Pi_{toCh} = \{b1-3, b7, b17, b4-6, b11-12, b14-16, b13, b18\}$.

2. In the step *2* the three buckets *1*, *2* and *3* would be selected. Note that all of them satisfy the condition that they all share an interval of time greater than the required runtime.

3. In the step *3* a valid allocation is created with the buckets selected in the previous step. Thus, the set of buckets Π_{sel} is composed of $\Pi_{sel} = \{b1, b2, b3\}$. Note that these buckets are removed from the set Π_{toCh}.

4. In the step *4* the penalty associated with each of the buckets is computed. In the example, due to the resource saturation, the three buckets have an associated penalty of $penalty_1$,$penalty_2$ and $penalty_3$ greater than one (marked with a gray area in the figure 2). With these buckets the basic allocation is created because they share the required amount of time.

5. In step *5* the algorithm has to evaluate whether the current allocation can be improved by replacing any of the buckets of $\{b1, b2, b3\}$ by any of the buckets that have not been evaluated Π_{toCh}.

   (a) As the current number of buckets in Π_{sel} is equal to the requested processors the algorithm continues the iteration.

   (b) The bucket with the maximum penalty is selected *b1* (> 1).

   (c) In the first iteration the bucket *b7* is evaluated. The penalty factor that processing the job α would experience if it were allocated using this bucket would be $penalty_7 = 1$.

   (d) As the penalty $penalty_7$ is lower than the $penalty_1$, on one hand the bucket *b1* is removed form the Π_{sel} and inserted to Π_{desc}. On the other hand the bucket *b7* is removed from the Π_{toCh} and inserted in Π_{sel}. The reservation table at this point is shown in figure 3. Since not all the penalties of the selected buckets are 1 the algorithm goes ahead with the next bucket.

   (e) As in the first iteration, since the current number of buckets in Π_{sel} is equal to the requested processors the algorithm continues the iteration.

   (f) The bucket with the maximum penalty is selected *b2*.

   (g) In the second iteration, bucket *b17* is evaluated. The penalty factor that a process of the job α would experience if it were allocated using this bucket would be $penalty_{17} = 1$.

   (h) As the penalty $penalty_{17}$ is lower than the $penalty_2$, on one hand the bucket *b2* is removed form the Π_{sel} and inserted to Π_{desc}. On the other hand the bucket *b17* is removed from the Π_{toCh} and inserted in Π_{sel}. The reservation table at this point is shown in figure 4. Since all the penalties of the selected buckets is 1, the algorithm returns the allocation based on the currently selected buckets, because they provide the minimum penalty.

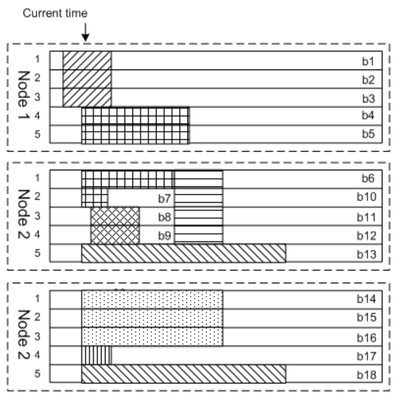

Figure 1: Less Consume Example

In the previous example, the start time for the allocation computed using the LessConsume resource allocation policy is the same as would have been obtained by using a First Fit resource selection policy. Thus, in the

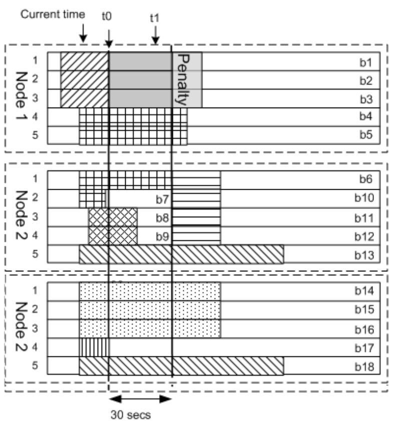

Figure 2: Less Consume Example - General Step 1

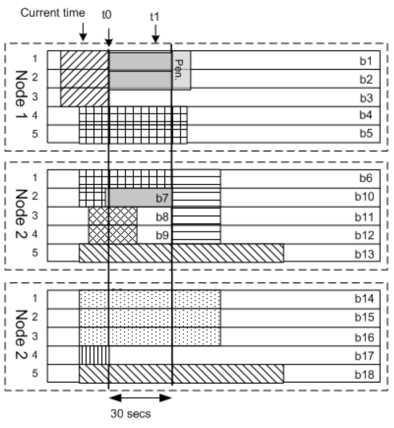

Figure 3: Less Consume Example - General Step 2

previous example, the start time remained equal to the First Fit RSP, and the penalty associated with the job has been reduced. However, in some situations, the LessConsume policy may provide allocations with later start times than those using First Fit. For instance, if in this example the process allocated to the bucket b_{17} had experimented a penalty of *1.5* the LessConsume algorithm would have iterated again looking for an allocation with less penalty factor but later start time.

5 THE LESSCONSUME THRESHOLD RESOURCE SELECTION POLICY

As we have shown in the example, 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 FirstFit 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 LessConsumeThreshold RSP. This is a parametrized selection policy which determines the maximum allowed penalty factor allocated to any given job.

This 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 parametrized 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 the following subsection we describe a backfilling based resource selection policy that uses the LessConsumeThreshold RSP to decide which jobs should be backfilled and how to allocate the jobs.

The main differences between the two policies is that in the steps *4)* and *5.d.2)* if the number of buckets of the set Π_{sel} is the required processors for the job, and they have an associated penalty factor lower or equal to the specified *Threshold*, then the allocation will be defined and returned based on the current set. Note, that in the LessConsume policy the algorithm would iterate evaluating the rest of the free buckets. On the other hand, note that policy enforces that the job allocation penalty must be lower than the provided threshold. Thus if in the step *6)* of the algorithm an allocation is found but has a higher penalty the *treq* will be updated as in the step *4)* and the process would be iterated again. Note that the LessConsume would be stopped at this point due to it has a valid allocation with the lowest penalty that could achieve by optimizing the First First allocation.

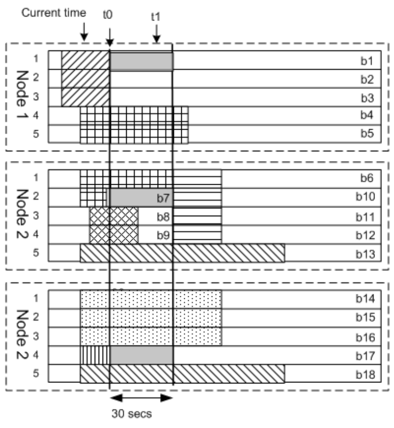

Figure 4: Less Consume Example - General Step 3

EXPERIMENTS

In this paper we evaluate the effect of considering the memory bandwidth usage when simulating the Shortest Job Backfilled Firts policy under several workloads and both LessConsume resource selection policies (the **Less-**

Center	M.	FF	LC	1	1,15	1,25	1,5
CTC	H	8,8	8	7,6	7,8	7,9	8,1
	M	4,8	3,8	3	3,5	4,0	3,9
	L	0,9	0,7	0,5	0,7	0,6	0,8
SDSC	H	12	11	8,3	10	11	12
	M	6,7	6,1	4,7	4,8	5,6	5,9
	L	1,4	1,1	0,7	0,8	0,9	1,1

Table 1: Percentage of Penalized Runtime - 95_{th} Percentile

Center	M.	FF	LC	1	1,15	1,25	1,5
CTC	H	4,2	5,9	7,92	6,1	5,3	5,2
	M	2,8	3,5	4,22	3,8	3,6	3,5
	L	2,2	3,12	3,62	3,8	3,4	3,5
SDSC	H	99	110	128	115	109	106
	M	55	68	74	72	71	68
	L	37	45	57	52	42	42

Table 2: Bounded-Slowdown - 95_{th} Percentile

ConsumeThreshold with the thresholds *1*, *1,15*, *1,25* and *1,5*). Two different workloads from the Feitelson workload archive have been used. 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.

5.1 Workloads

For the experiments we used the cleaned Tsafrir and Feitelson (2003) 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 first 10000 jobs of each workload. 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.

5.2 Architecture

For each of the workloads used in the experiments we defined an 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 Chapin et al. (1999) 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, 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.

6 EVALUATION

Table 2 present the 95_{th} percentile of the bounded slowdown for the CTC and SDSC centers for each of the three workloads for the FirstFit, LessConsume and LessConsumeThreshold resource selection policy. The last one was evaluated with four different factors: *1*, *1,15*, *1,25* and *1,5*. In both centers the LessConsume policy performed better than the LessConsumeThreshold with a factor of *1*. One could expected that the LessConsume should be equivalent to use the LessConsumeThreshold with a threshold of *1*. However, note that this affirmation would be incorrect. This is caused due to the LessConsume policy at the step *6)* of the presented algorithm will always stop. The goal of this policy is to optimize the First Fit allocation but without carry out a deeper search of other possibilities. However, the LessConsumeThreshold 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 LessConsumeThreshold 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.

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 LessConsumeThreshold with thresholds of *1* and *1,5* respectively.

The 95^{th} percentage of penalized runtime is presented in the table 1. 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 LessConsumeThreshold with a factor of *1,5*. This percentage of penalized runtime was reduced with respect to the First Fit when using all the

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

Table 3: Number of Killed Jobs 95th Percentile

different factors in both centers.

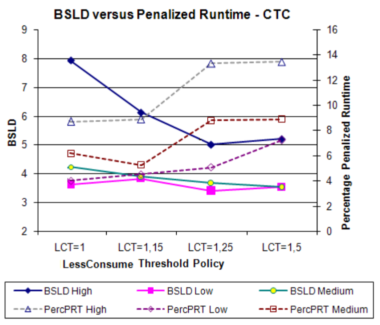

Figure 5: BSLD versus Percentage of Penalized Runtime - CTC Center

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 LessConsumeThreshold 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.

The LessConsume policy shows how the percentage of penalized runtime and number of killed jobs can be reduced in comparison to the First Fit and First Continuous Fit, by using this policy with EASY backfilling. Furthermore, the LessConsume threshold shows how, with different thresholds, performance results can also be improved. Relaxing the penalty factor results in better performance of the system, and in an increase in the number of killed jobs and the percentage of penalized runtime. The LessConsume policy shows similar performance results as the LessConsumeThreshold with factors of *1,25* and *1,5*.

Figures 6 and 7 present the BSLD of the LessConsume policies 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 LessConsumeThreshold 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 5 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 LessConsumeThreshold with a factor of *1* 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*.

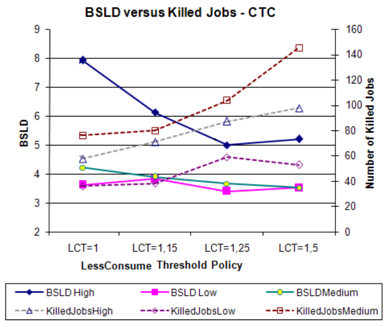

Figure 6: BSLD versus Killed Jobs - CTC Center

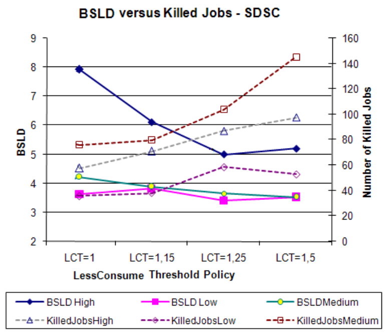

Figure 7: BSLD versus Killed Jobs - SDSC Center

7 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 by using the two LessConsume resource selection policies that consider the resource sharing when the jobs are allocated.

We have described the *Find Less Consume Distribution* 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. We have also described the *Find Less Consume Threshold Distribution* 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*. This resource selection policy has been 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. Thus, this RSP can be used by the scheduler to find an allocation for a given job in an iterative process until the most appropriate allocation is found.

We have evaluated the impact of using the LessConsume and LessConsumeThreshold (Thresholds *1*, *1.15*, *1.25* and *1.5*) with the Shortest Job Backfilled first. In this evaluation, we have used the workloads described in the previous chapter where we evaluated the impact of memory bandwidth sharing on the performance of the system. Both resource allocation policies show how the performance of the system can be improved by considering where the jobs are finally allocated. The bounded slowdown of both policies show slightly higher values than those achieved by a First Fit resource allocation policy. However, they show a very important improvement in the percentage of penalized runtimes of jobs, and more importantly, in the number of killed jobs, showing a very good balance in the increment of the BSLD. Both have reduced by four or even six times the number of killed jobs in all the evaluated workloads. Note that each of the indicated thresholds depends on the center. In the SDSC the a threshold of *1* or *1.15* shows a good balance of performance (BSDL) and number of killed jobs and percentage of penalized runtime, and in the CTC center the appropriate threshold is *1.5*.

ACKNOWLEDGEMENTS

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

REFERENCES

Chapin, S. J., Cirne, W., Feitelson, D. G., Jones, J. P., Leutenegger, S. T., Schwiegelshohn, U., Smith, W., and Talby, D. (1999). Benchmarks and standards for the evaluation of parallel job schedulers. *Job Scheduling Strategies for Parallel Processing*, vol 1659:pp. 66–89.

Chiang, S.-H., Arpaci-Dusseau, A. C., and Vernon, M. K. (2002). 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.

Feitelson, D. G., Rudolph, L., and Schwiegelshohn, U. (2004). Parallel job scheduling - a status report. *Job Scheduling Strategies for Parallel Processing: 10th International Workshop, JSSPP 2004*, 3277 / 2005:9.

Guim, F., Corbalan, J., and Labarta, J. (2007). Modeling the impact of resource sharing in backfilling policies using the alvio simulator. *MASCOTS*.

Talby, D. and Feitelson, D. (1999). Supporting priorities and improving utilization of the ibm sp scheduler using slack-based backfilling. *Parallel Processing Symposium*, pages pp. 513–517.

Tsafrir, D., Etsion, Y., and Feitelson, D. G. (2005). Backfilling using runtime predictions rather than user estimates. *Technical Report 2005-5, School of Computer Science and Engineering, The Hebrew University of Jerusalem.*

Tsafrir, D. and Feitelson, D. G. (2003). Workload flurries. Technical report, School of Computer Science and Engineering and The Hebrew University of Jerusalem.