Another approach to backfilled jobs: applying Virtual Malleability to expired windows

Gladys Utrera, Julita Corbalán, Jesús Labarta
Dept. of Computer Architecture, Jordi Girona, 1-3. Mòdul D6 Campus Nord
Technical University of Catalonia
08034 Barcelona (SPAIN)
Phone: +34-934017001
{gutrera, juli, jesus} @ac.upc.edu

ABSTRACT

An efficient job scheduling must ensure high throughput and good performance. Moreover in highly parallel systems where processors are a critical resource, high machine utilization becomes an essential aspect.

Backfilling consists on moving jobs ahead in the queue, given that they do not delay certain previously submitted jobs. When the execution time of a backfilled job was underestimated, some action has to be taken with it: abort, suspend/resume, checkpoint/restart, remain executing.

In this paper we propose an alternative choice for that situation which consists on apply Virtual Malleability to the backfilled job. This means that its processors partition will be reduced, and as MPI jobs aren't really malleable, we make the job contend with itself for the use of processors by applying Co-scheduling. In this way resources are freed and the job at the head of the queue have a chance to start executing. In addition to this, as MPI parallel jobs can be Moldable, we add this possibility to the scheme.

We obtained better performance than traditional backfilling in about 25 %, especially in high machine utilization. We claim also for the portability of our technique which does not requires special support from the operating system as checkpointing does.

1. INTRODUCTION

An efficient job scheduling must ensure high throughput and good performance. Moreover in highly parallel systems where processors are a critical resource, high machine utilization becomes an essential aspect. A parallel job is an application composed by processes, which run concurrently and cooperate to do a certain computation. When the number of processes is fixed during the whole execution, the job is said to be rigid. If this number can be chosen at the beginning of the execution, the job is said to be moldable and if it can also be modified during execution the job is said to be malleable [8]. This work is focused on MPI jobs [17], and we will assume that they can be moldable.

In order to get better machine utilization and synchronization among processes from a parallel job, they are usually allocated into processors partitions (Space Sharing policies) [13], usually static. This approach suffers from fragmentation [10] which can be alleviated by applying backfilling techniques [15] to the queuing system and/or using Moldable jobs. However this last option is not habitual in production systems.

Backfilling consists on moving jobs ahead in the queue, given that they do not delay certain previously submitted jobs. It has demonstrated to improve system utilization and reduce job wait time versus the same policy without backfill [10]. This approach depends on user time estimate for its effectiveness. In [10],[30] they state that overestimation has little impact on the performance of such policy. However if the execution time of a backfilled job is underestimated, some action has to be taken with it: abort [18], suspend/resume, checkpoint/restart [18], remain executing [29], [26].

In this paper we propose an alternative choice to do when the window time of a backfilled job has expired, which is to fold the backfilled job by applying Self-Coscheduling [28] to it. This means that the processors partition from the expired job is reduced, and as MPI jobs aren't malleable, it will have to compete with itself for the use of resources. At this point we apply co-scheduling techniques to the processes from the parallel job in order to synchronize them and progress execution. In this way resources are freed and the job at the head of the queue have a chance to start executing. In addition to this, as MPI parallel jobs can be Moldable [8], we will add this possibility to the scheme to increase system utilization.

We implement our proposal on a shared memory multiprocessor system and evaluated it using several dynamic workloads composed by NAS Benchmarks [2], varying the percentage of long and short jobs.

ICS '05, June 20–22, 2005, Boston, MA, USA.

We obtained better performance than traditional backfilling and a recently proposed scheme, which combines Aggressive Backfilling with Moldability [24], especially for high machine utilization. We claim also for the portability of our technique, which does not require special support from the operating system as checkpointing do.

The rest of the paper is organized as follows: In Section 2 we present the related work. Then in Section 3 and 4 is the motivation and a description of our proposed technique. Next in Section 5 we describe the execution framework. After that in Section 6 we present the evaluation and finally in Section 7 the conclusions remarks and future work.

2. RELATED WORK

The simplest way to schedule jobs is to use the First-Come-First-Served (FCFS) policy. This approach suffers from low system utilization. The backfilling technique first developed by Liftka [15] was proposed to improve system utilization and has been implemented in several production schedulers [9]. This technique is greatly known to increase user satisfaction since small jobs tend to get through faster, while bypassing larger ones.

To avoid starvation of larger jobs, the Conservative Backfilling requires that the execution of jobs selected out of order does not delay any job arrived earlier [10]. The EASY backfilling takes a more aggressive approach and allows short jobs to skip ahead provided they do not delay the job at the head of the queue [15].

There are several variations proposed in the literature to Backfilling techniques. In [23] Srinivasan et al. present the Selective Backfilling wherein jobs do not get a reservation until their expected slowdown exceeds some threshold. In [26] Talby and Feitelson show an enhanced backfill scheduler that support priorities. Ward et al. in [29] suggest the use of a Relaxed Backfill strategy, which is similar, except that the slack is a constant factor and does not depend on priority.

There is an interesting work about which jobs to choose to backfill in [21], where Shmueli and Feitelson use dynamic programming to look deeper into the queue and select a set of jobs, which together would maximize the machine utilization.

In [11] Frachtenberg et al evaluate the impact of multiprogramming level, time quantum and the use of backfilling on scheduling systems running dynamic workloads. They also show the advantage of packing backfilling with scheduling strategies like flexible coscheduling [6] or gang scheduling [12] as demonstrated in [30].

Lawson and Smirni presented a multiple-queue backfilling approach, where each job is assigned a queue and a partition depending on its estimated execution time [14]. In their simulation results they obtain a gain in performance with respect to a single queue-backfilling.

Backfilling techniques are based on having information about runtimes estimates. There is a work about runtime estimation through repeated executions [3], and another through compile-time analysis [19]. In practice users provide it as a runtime upper-bound [10].

About integrating Moldability to Backfilling techniques in [24] Srinivasan et al. propose a technique that selects the partition size for a job based on its scalability and turnaround time by applying the Downey model [1]. They add Aggressive Backfilling and demonstrate a gain in performance over pure Backfilling and pure Moldability [4].

In [27] Utrera et al. they propose an algorithm based on Moldability and Folding to implement Malleability on MPI jobs and to select the appropriate partition size and folding times. They demonstrate to outperform other typical Moldability techniques and work well especially with low machine utilization and bursts arrivals.

3. MOTIVATION

The main goal of this work is to reduce fragmentation in order to improve machine utilization resulting in a better system performance. Moldability achieves that purpose by adjusting the partition sizes according to the available resources at the beginning of the execution of each job. However there are jobs that are not “fully moldable” as their size range is cuadratic or power of two, this is a common case when working with matrixes. In addition there must be a compromise between reducing wait queue time and incrementing execution time. For example for long jobs it would not be beneficial for them to be assigned a small partition since their execution time will be so much degraded.

For these reasons in spite of applying Moldability, fragmentation is not eliminated at all. So here come the Backfilling techniques, which will be in charge of filling holes generated in the situations described above.

As Backfilling techniques rely on user runtime estimates, there may be inaccuracy. In this work, jobs are classified in long and short ones. If a backfilled job doesn’t finish execution within the window time assigned, it will prevent the job at the head of the queue from executing. To treat this problem, the backfilled job can be: 1) aborted [18], 2) suspended/resumed, 3) checkpointed/restarted, 4) remain executing during a period of time [29], [26]. Except for the option 4), the scheduler will have to reinsert the job in the wait queue. In the option 2) the job must be resumed in the same processors partition, unless it is running on a shared memory multiprocessor, in which case it would be also advisable to minimize the memory impact. This may add a considerable delay for resuming the job. In addition not all the operating systems have support for option 3).

In this paper we propose another alternative to solve this problem. We don’t abort execution, nor do we delay the execution of the job at the head of the queue. We just reduce the partition size of the backfilled job by applying Self-Coscheduling [28] to it. That is the job is folded four times at most, reducing by four its partition size and apply Co-Scheduling to the processes local queues generated. In a way we made the backfilled job “malleable” [27], thus freeing resources for the highest priority job. Notice that if the backfilled job had aborted there would be even more free resources. In our scheme, as we are working with moldable jobs, we can adjust such difference to the newly available partition.

As a result we would not have to reinsert the job in the queue and the backfilled job will be able to continue execution, minimizing delays because of inaccuracy of runtime estimates.

4. BACKFILLING WITH MALLEABLE JOBS (BFM)

In this section we describe the technique. For implementation details see next section.

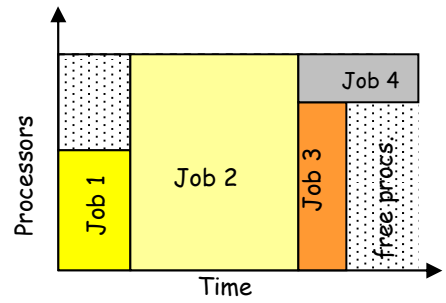
As already mentioned we work with moldable jobs. We select the partition size by applying the FJT algorithm [27]. This decision is taken just before the job starts executing and is based on its type, that means if it is long or short, and the number and type of the jobs currently running and in the wait queue. This algorithm demonstrated to work better than other moldability algorithms from the literature and forms part of a policy to implement virtual malleability on MPI jobs. The Virtual Malleability scheme consists on start executing Long jobs folded if there is a possibility in the near future for it to expand to possibly newly available processors. This situation is favoured when the jobs arrivals are by bursts.

We are focusing on high loaded machines where there are usually jobs waiting to execute with no partition assigned yet. In preliminary experiments we have seen that in this context, the virtual malleability as presented in [27] has little or any gain, as jobs won't have the opportunity to enlarge their processors partition. For this reason we will apply Virtual Malleability only to the backfilled jobs with underestimated times as we will describe below. For the rest of jobs, no matter if they are long short, we will apply Moldability.

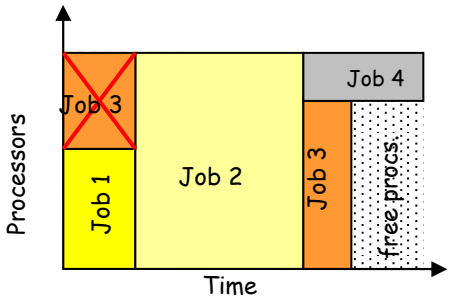
Each time a job at the head of the queue could not start execution immediately, we proceed to find a suitable job in the wait queue to be backfilled. That is a small job that can be adjusted to the available partition size at that moment. We continue backfilling jobs until the window expires or the waiting job can start executing. We state that the window time has expired if all the currently running jobs are "out of order", that means all were backfilled.

Once the window expires, we reduce the partition size of the backfilled jobs starting with the oldest one, until there are enough resources for the first job in the wait queue. We achieve that reduction by applying the Self Co-Scheduling [28] technique. It consists on folding the job by four, which make the job contend with itself for the use of processors and apply Co-scheduling [28] to the processes. For implementation details see the Cpu Manager section.

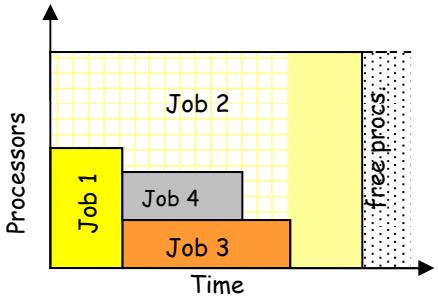
This dynamic adjustment to a smaller set of processors, allows that later in time, if new resources become available and the wait queue is empty, the shrinked jobs will have the possibility to expand and take advantage of them, thus resulting in a dynamic partition for the backfilled jobs and gives the idea of virtual malleability.



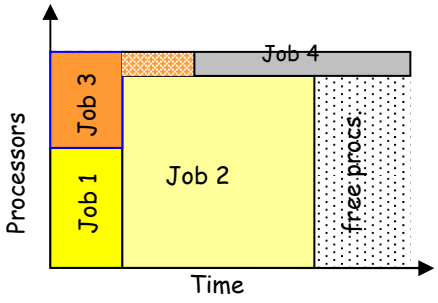
(a) FIFO



(b) Aggressive Backfilling



(c) Virtual Malleability (FJT)



(d) Backfilling with Malleability

Figure 1. Scheduling schemes for moldable jobs.

Above in Figure 1, we can observe and example of several scheduling schemes for four moldable jobs that arrive at the same time, where *job 1, job 3* and *job 4* are Short and *job 2* is Long. The Fig.1(a) is the FIFO example, where we can observe the fragmentation generated when the Long job is not able to take advantage of the free processors because the partition is not big enough for it. The Fig 1 (b) is the same example but applying Aggressive Backfilling [15] where *job 3* is backfilled and adjusted to the available partition. However the job is aborted because the window time has expired. The Fig 1(c) shows the same workload under the FJT algorithm with Virtual Malleability included. We can see that *job 2* starts execution in the available partition but with a number of processes greater than the number of free processors. Only after *job 3* finishes execution, *job 2* is able to expand totally. Meanwhile it runs folded adapting to the currently available processors partition. Although execution time for *job 2* is incremented due to the overhead generated because of the folding, it has reduced its queue wait time to zero. In addition queue wait time for *job 3* and *job 4* is reduced also compared to FIFO in Fig.1 (a) and Aggressive Backfilling in Fig.1 (b). Finally in Fig 1. (c) we show our proposal, the Backfilling with Malleability where *job 3* is backfilled but when the window time expires, instead of aborting it we apply the Virtual malleability by folding it and enabling *job 2* to start execution immediately in the newly available partition. Notice that in this case *job 2* will have a smaller partition than in the rest of the schemes. However *job 3* succeed to continue execution and as soon as it finishes, *job 4* will be able to start executing. As we can see in Fig.1 when all the jobs finishes execution, option (c) has sooner than the rest the biggest free processors partition.

5. EXECUTION FRAMEWORK

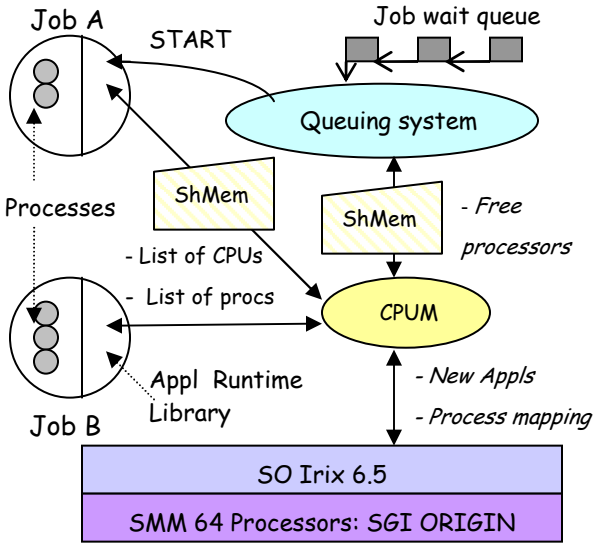


Figure 2 Execution framework: Launcher, CPUM, Jobs and shared information between them.

In this section we will describe the implementation and characteristics of the queuing system used for the evaluation, the Launcher which implements the processor allocation strategies, the Cpu Manager (CPUM) [16], which manages the process to processor mapping, and the Application runtime library which implements the processor scheduling.

In Figure 2 we show the execution environment. The FJT Moldability policy and BFM Backfilling is implemented in a distributed way along the Launcher and the CPUM and the Application runtime library. In addition, is external to the operating system, the applications and the MPI library.

5.1 Queuing system: Launcher

The Launcher is the user-level queuing system used in our execution environment. It applies the queuing system policy (FIFO or backfilling) to a list of jobs belonging to a predefined workload. The Launcher in coordination with the CPUM controls the maximum multiprogramming level (MPL[1]).

Whenever there is a free processors partition, the Launcher tries to execute the job at the head of the wait queue. It decides, by applying the FJT algorithm, the number of processes and the initial number of Folding times according to the size of the processors partition and the job type. If there are enough processors that satisfy the minimal requirements of the job, then it is launched.

If backfilling is allowed, then the Launcher looks for expired windows of previously backfilled jobs starting with the oldest one, until there are no more backfilled jobs or the just freed partition is suitable for the job at the head of the wait queue. In order to free processors, the Launcher, depending on the Backfilling policy established, it aborts each backfilled job or applies Virtual Malleability to it. On the contrary, if there wasn't any expired window, then the Launcher proceeds to examine the wait queue to find a job to be backfilled.

5.2 Cpu Manager (CPUM)

The CPUM is a user-level scheduler. Once the Queuing system has launched the job, it enters under the CPUM control. The CPUM implements the processor allocation policies, deciding where the job will be allocated. The communication between the CPUM, the Launcher and the jobs is done through shared memory by control structures.

The CPUM wakes up periodically and at each quantum expiration examines if new jobs have arrived to the system or have finished execution and updates the control structures and if necessary depending on the scheduling policy, redistributes processors.

5.3 Application Runtime Library

In order to get control of MPI jobs we use the DiTools Library [20]. They implement a dynamic interposition mechanism that allows us to intercept functions like the MPI calls or a system call routine such that the Sginap, which is invoked by the MPI library when it is performing a blocking function. These functions provide information to the CPUM or gets information from it. In addition using this mechanism we can inhibit the

[1] We call MPL to the number of processes attached to a processor. When applying the Folding technique, each job will have its own maximum MPL depending on the Folding times.

execution of the Sginap routine. This is useful when having several processes allocated to a processor as when applying Virtual Malleability. The Sginap wrapper, is in charge of doing context switching each time the spin time has expired and decides which process goes next following. We set the spin time to zero, that is Blocking immediately and followed a Round Robin scheme for the next process to execute at processor local queue. We use also this interposition mechanism to initialise some control structures of the Application runtime library and to find out each process MPI rank. This is useful for the process to processor mapping.

All the techniques were implemented without modifying the source code of the native Silicon Graphics [22] MPI library and without recompilation of the applications.

6. EVALUATION

In this section we describe the performance evaluation. First there is a brief enumeration of all the techniques implemented and evaluated. Then we describe the hardware platform. After that we comment the applications and workloads used for the evaluation and finally we present the performance results.

6.1 Evaluated techniques

In this section we describe the evaluated techniques in order to compare with the BFM.

The FIFO technique was chosen as a reference. We choose the FJT [27] for being the moldability technique with the possibility of Virtual malleability, which also demonstrated to outperforms others from the literature. We add to this Backfilling with the option of abortions (Aggressive Backfilling) and malleability (Backfilling with Malleability).

These are the selected techniques for the evaluation:

1. ***FIFO***: First come, first served. The jobs are dispatched in the same order as they arrive to the system. We assigned the maximum number of processes specified for each application.
2. ***Virtual Malleability*** **(FJT4)**: FJT for choosing job sizes and at most MPL=4 for Long jobs, making them virtually malleable. Backfilling is not allowed. For a more detailed description see Section 4.
3. ***Virtual Malleability with Backfilling*** **(FJT4+BFA)**: FJT for choosing job sizes with at most MPL=4 for Long jobs, making them virtually malleable, and Aggressive Backfilling [10]. The same as 2) but with backfilling allowed.
4. ***Pure Moldability with Backfilling*** **(FJT1+BFA)**: FJT for choosing job sizes, with MPL=1 for Long and Short jobs, and Aggressive Backfilling. Here jobs are just moldable, and Backfilling is allowed.
5. ***Backfilling with Malleability*** **(FJT1+BFM)**: FJT for choosing job sizes with MPL=1 for Long and Short jobs; here Long and Short jobs are just moldable except for the backfilled jobs, that become malleable if their window time expires.

6.2 Architecture

Our implementation was done on a CC-NUMA shared memory multiprocessor, the SGI Origin 2000 [2]. It has 64 processors, organized in 32 nodes with two 250MHZ MIPS R10000 processors each. The machine has 16 Gb of main memory of nodes (512 Mb per node) with a page size of 16 Kbytes. Each pair of nodes is connected to a network router. The operating system where we have worked is IRIX version is 6.5 and on its native MPI library with no modifications.

6.3 Applications and Workloads

To drive the performance evaluations we consider applications from the MPI NAS Benchmarks suite [13] and the Sweep3D [25]. Applications such as the BT admits only quadratic job sizes, the CG and LU admit power of 2 job sizes and the Sweep3D admits any job. We restricted the job sizes to the maximum number of available processors that was 60.

As stated in [8] performance results especially for backfilling techniques depend mostly on the workloads used for the evaluations. In [11] they support that while traces from a real system have the benefit that reflects the real workload on a specific production system, these traces may not be representative of the workloads of other systems. On the other hand these traces couldn't be used for real executions as in our case. In addition it is impractical for us to reproduce a year of execution of a production system to perform the evaluations.

So we run our evaluations over dynamic workloads. The workloads were sized to run during 10 minutes of linear CPU time. Each workload has about 120 jobs. But depending on the policy being evaluated the workload execution time could take more than this. In order to repeat the same experiment as many times as needed we generated the workload as a list of applications with their arrival time, with an exponential distribution, is passed as a parameter to the Launcher. These traces were constructed following the format described in [7].

We classified the jobs in Long and Short ones. Below is a table with the criterion to classify jobs in Long and Short depending on their linear execution times.

Table 1. Job type depending on their linear execution time

Long	Linear Exec. Time	Type
bt.A	2441 seconds	Long
cg.B	4385 seconds	Long
lu.W	177 seconds	Short
Sweep3D	50 seconds	Short

The workloads were constructed varying the percentage of each other, and the machine utilization. These are described in Table 2.

Table 2. Workloads used for the evaluations

Machine utilization	100	80	60	40
% Long jobs	80	60	40	20
%Short jobs	20	20	20	20

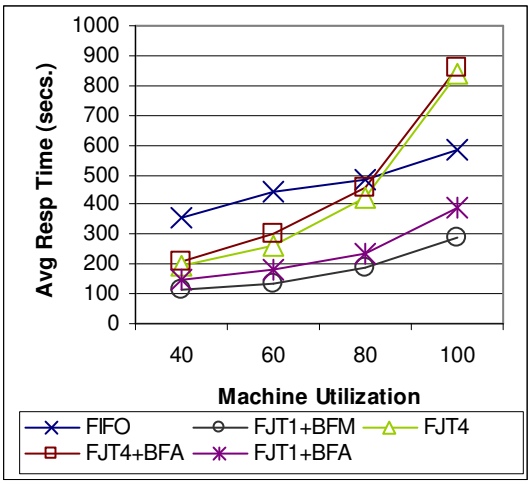
As shown in Table 1, machine utilization goes from 40 to 100%, distributed in 20% for Short jobs and the rest for Long ones. However, Long jobs represent only at most 30% of the total number of jobs. This distribution, was used in the evaluations in [30], which were extracted from a real production system. In addition in the collection of workloads logs available from Feitelson's archive in [5] the distribution percentages between long and short jobs are mostly around 20 to 30% for short jobs and 70 to 80% for long ones.

6.4 Performance results

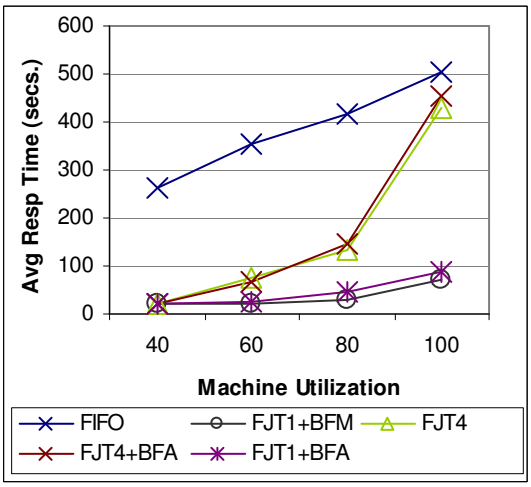
In this section we present the performance results for the different policies enumerated in section 6.1 evaluated under the workloads detailed in section 6.3 for Long and Short jobs. We present results for long and short jobs separately. First we analyze the average response times, which include queue wait time and execution times. After that we show the average execution times and the average queue wait times. As the applications repeat within each workload, we consider the average response, queue and execution time respectively for each one. Then we study the standard deviation of the queue wait times as a measure of the degree of confidence of the executions from a user point of view. Finally we present the average number of processors used by each workload under the policies evaluated.

6.4.1. Response Times

In Figure 3 we can see the average response times for Long (a) and Short (b) jobs. We can notice that the proposed technique, the Backfilling with Malleable Jobs (BFM), combined with Moldability has the best performance, it achieves a gain from 20 to 30% for high machine utilization over the traditional backfilling combined with Moldability for Long jobs and from 5 to 10% for Short ones.



(a) Long jobs

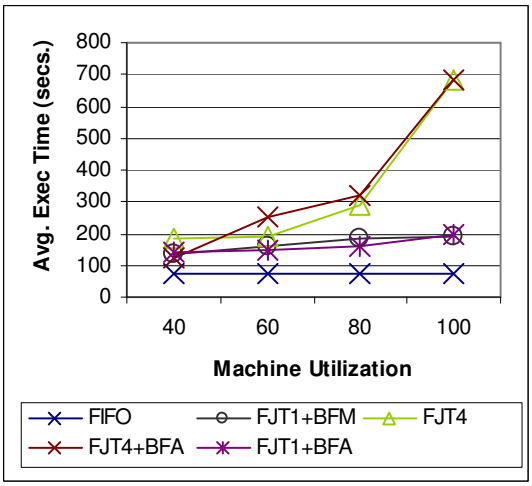


(b) Short jobs.

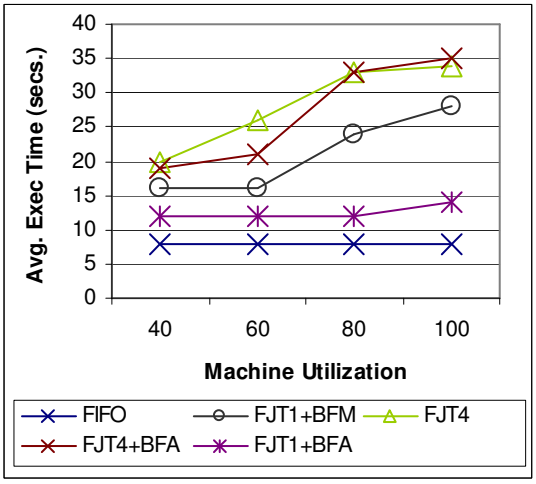
Figure 3. Comparing the Average Response Times of the backfilling policies evaluated.

6.4.2 Execution and Queue Wait Times.

Analyzing execution times of the applications under BFM backfilling, Short jobs perform worse than Aggressive Backfilling. This is due the folding effect when applying BFM when the window expires, as the processes from a job must compete with themselves for the use of processors. This can be seen clearly in Figure 4(b). Moreover as load increments this effect is more noticeable. We can see the same effect on Long jobs under Virtual malleability with and without Backfilling.



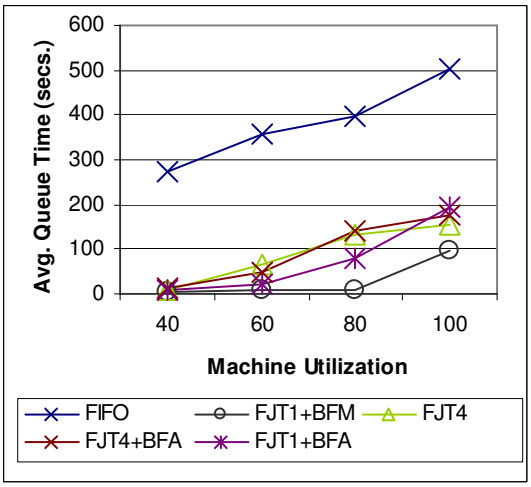
(a) Long jobs



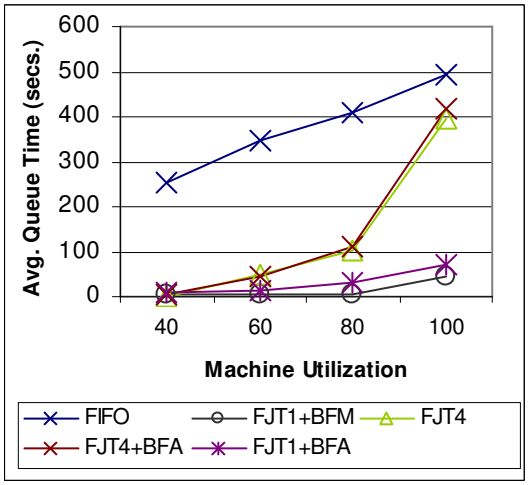
(b) Short jobs

Figure 4. Comparing the Average execution time of the backfilling policies evaluated.

The response times as already mentioned are composed by execution and queue wait times. From the graphic above it is easy to deduce that as the execution time are equal or worse under BFM than under Aggressive Backfilling, the gain obtained by our proposal in response time comes from the fact that we have minimized queue wait times dramatically compared to the rest of policies evaluated, as can be seen in Figure 5.



(a) Long jobs



(b) Short Jobs

Figure 5. Comparing the Average Queue Times of the backfilling policies evaluated.

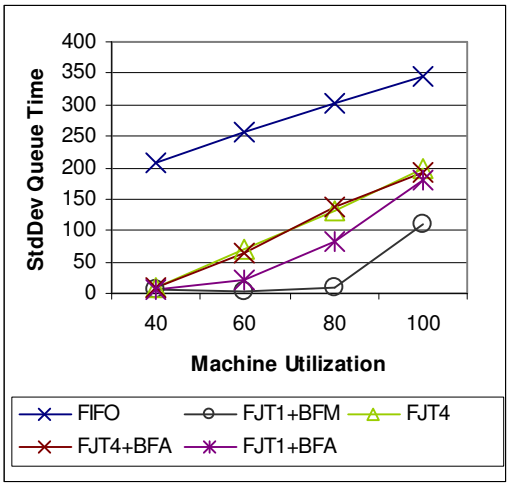
Under BFM backfilling, no matter if the window expires, the work done by short jobs is not wasted, once they are backfilled, they will run until completion. In addition long jobs can start executing immediately concurrently with the expired backfilled jobs, thus reducing queuing times for long and short jobs as showed in Figure 5. If aborting, jobs will have to be reinserted in the queue, becoming eligible to be backfilled again. Thus incrementing their queue wait time and wasting resources.

6.4.3 Standard Deviation of Queue Wait Times

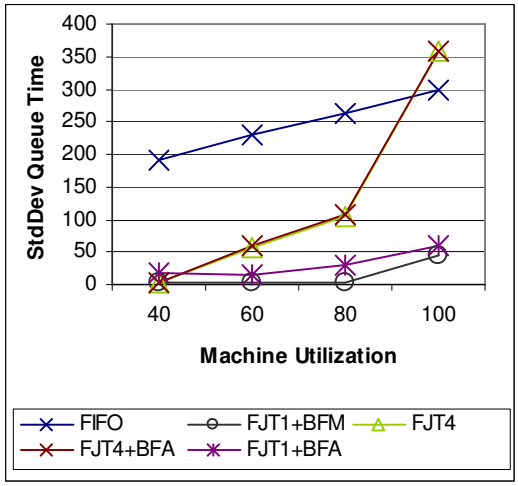
This represents a measure of fairness about queue times and is shown in Figure 6. Moreover they represent a degree of confidence about turn around time form a user point of view.

As we can observe in the figure 6, the standard deviation of the queue times for the BFM technique are near zero, which means that in general the queue wait times are very close to the average showed in figure 5. However, this effect does not occur

under the rest of the techniques. Moreover as the load increments, response times could become unpredictable; a user don't have a clear interval of confidence within his job will be about to finish execution.



(a) Long jobs

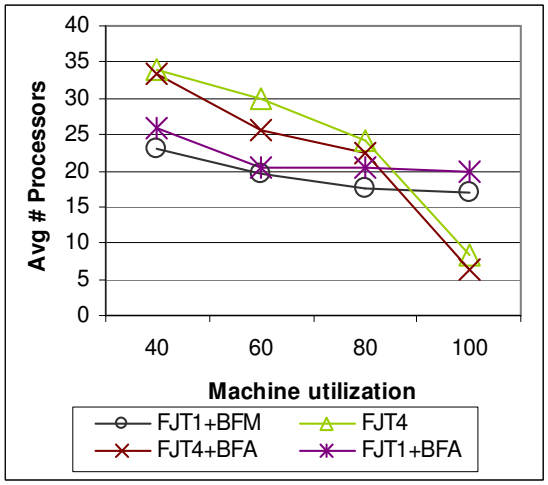


(b) Short jobs

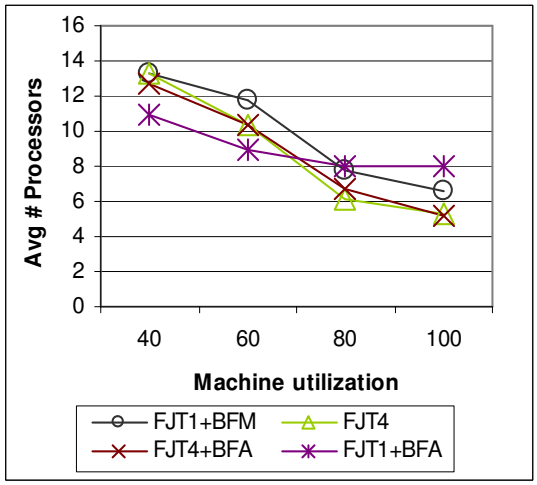
Figure 6. Comparing the Std. Deviation of Queue times of the backfilling policies evaluated

6.4.4 Processors partition size

We can see in Figure 6, that under Malleability techniques (FJT4) with or without backfilling, with low machine utilization, they obtain in average the biggest partition size. This is because this technique can take advantage when load goes down, of new freed processors, as Long jobs can expand to them. However as load increments, Long jobs will execute shrinked all the time. Let's observe also that under BFM backfilling, Long jobs have the smallest partition. This comes from the fact that when the window of a backfilled job expires, even its processors partition is reduced, the partition reserved for the job at the head of the queue would not as big as the obtained if the backfilled job had aborted.



(a) Long jobs



(b) Short jobs

Figure 6. Comparing average processor partitions sizes.

7. CONCLUSION REMARKS AND FUTURE WORK

We presented a choice to improve traditional Backfilling techniques when the window has expired. Instead of aborting or suspend and reinsert the job in the queue, we just reduce its partition size by applying Self-Coscheduling. In addition we allow Moldability augmenting the possibility to find a suitable partition to backfill a job or to enter executing.

We implement our proposal, the Backfilling with Malleability, and compared with other Moldability and Backfilling techniques, under several dynamic workloads and demonstrated to overcome them in performance in about 20 to 30% especially for high machine utilization.

We think is portable and can be supported by any operating system. It reduces memory swapping generated by aborts/suspensions, prevents the queuing system from reinserting in the queue and re-executing the job in the future. Notice that if the job is reinserted in the queue it will be eligible again to be backfilled.

In the future we are planning to extend our experiment to more varied workloads to evaluate the impact in the performance of including gradually rigid jobs.

We would like also to optimize the Self-Coscheduling technique, when folding jobs, in order to apply a complex mapping from processes to processors taking into account communication groups and so on.

8. ACKNOWLEDGMENTS

This work was supported by the Ministry of Science and Technology of Spain under contract TIN2004-07739-C02-01. And has been developed using the resources of the DAC at the UPC and the European Centre for Parallelism of Barcelona (CEPBA).

9. BIBLIOGRAPHY

[1] W.Cirne. Using Moldability to Improve the Performance of Supercomputer Jobs. Ph.D Thesis. Computer Science and Eng. University of California San Diego, 2001.

[2] D. Bailey, T. Harris, W. Saphir, R. Wijngaart, A. Woo and M. Yarrow, "The NAS Parallel Benchmarks 2.0", Technical Report NAS-95-020, NASA, December 1995.

[3] M. V. Devarakonda, R. Iyer. Predictability of Process Resource Usage: A Measurement Based Study on UNIX. IEEE Trans. Soft. Eng. 15(12), pp.1579-1586, Dec. 1989.

[4] A. Downey. A Model for Speedup of Parallel Programs. Technical Report CSD-97-933. University of California at Berkerley,1997.

[5] D. G. Feitelson. Logs of real parallel workloads from production systems. http://www.cs.hujiac.il/labs/parallel/workload/logs.html.

[6] D.G.Feitelson and M.A.Jette. Improved Utilization and Responsiveness with Gang Scheduling. Job Scheduling Strategies for Parallel Processing, volume 1291 of Lecture Notes in Computer Science. Springer-Verlag 1997.

[7] D. G. Feitelson, B. Nitzberg. Jobs Characteristis of a Production Parallel Scientific Workload on the NASA Ames Ipsc/860, in JSSPP Springer-Verlag, Lectures Notres in Computer Science, vol. 949, pp. 337-360, 1995.

[8] D.G. Feitelson, L. Rudolph, U. Schiwiegelshohn, K.Sevcik and P. Wong. Theory and Practice in Parallel Job Scheduling. Lecture Notes in Computer Science, 1291:1--34, 1997.

[9] D. Jackson, Q. Snell and M. Clement. Core Algorithms of the Maui Scheduler. In Worshop on Job Sched Strategies for Parallel Processing, pp. 87-102, 2001.

[10] A.M Weil and D. Feitelson. Utilization, Predictabiligy, Workloads and User Runtimes Estimates in Scheduling the IBM SP2 with Backfilling. In IEEE Trans. on Parallel and Distributed Syst. 12(6), pp.529-543, Jun. 2001.

[11] E. Frachtenberg, D. Feitelson, J. Fernández, F. Petrini. Parallel Job Scheduling Under Dynamic Workloads. JSSPP 2003.

[12] E. Frachtenberg, D. G. Feitelson, F. Petrini, and J. Fernandez, "Flexible coscheduling: mitigating load imbalance and improving utilization of heterogeneous resources ". In 17th Intl. Parallel & Distributed Processing Symp., Apr 2003.

[13] A.Gupta, A.Tucker, and S. Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Jobs. In Proceedings of the 1991 ACM SIGMETRICS Conference, pp 120-132, May 1991.

[14] B. Lawson and E. Smirni. Multiple-queue Backfilling Scheduling with Priorities and Reservations for Parallel Systems. In Job Sched. Strategies for Parallel Processing, D.G. Feitelson and L. Rudolph (eds.), Springer Verlag, Lect. Notes Comp. Sc. Vol. 2537, pp. 72-87, 2002.

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

[16] X. Martorell, J. Corbalán, Dimitrios S. Nikolopoulos, Nacho Navarro, Eleftherios D. Polychronopoulos, Theodore S. Papatheodorou, Jesús Labarta: A Tool to Schedule Parallel Applications on Multiprocessors: The NANOS CPU MANAGER. JSSPP 2000: 87-112.

[17] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. Int. Journal of SuperComputer Jobs, 8(3/4):165-414, 1994.

[18] Q. Snell, Mark J. Clement, David B. Jackson: Preemption Based Backfill. JSSPP 2002: 24-37.

[19] V.Sarkar. Determining Average Program Execution Times and Their Variance. In Proc. SIGPLAN Conf. Prog. Lang. Dessign and Implementation, pp. 298-312, Jun 1989.

[20] Albert Serra, Nacho Navarro, and Toni Cortes. DITools: Application-level support for dynamic extension and flexible composition. In Proc. USENIX Annual Technical Conf., pp 225--238, 2000.

[21] E. Shmueli, D.Feitelson. Backfilling with Lookahead to Optimize the Performance of Parallel Job Scheduling. Springer Verlag 2003. Lectures Notes Comp. Science.

[22] Silicon Graphics, Inc. IRIX Admin: Resource Administration, Document number 007-3700-005, http://techpubs.sgi.com, 2000.

[23] S. Srinivasan, R. Kettimuthu,V. Subramani, P. Sadayappan. Characterization of Backfilling strategies for Parallel Job Scheduling. In Proc. of 2002 Intl. Workshops on Parallel Proc, Aug, 2002.

[24] S. Srinivasan, V. Subramani, R. Kettimuthu, P. Holenarsipur, and P. Sadayappan. Effective Selection of Partition Sizes for Moldable Scheduling of Parallel Jobs. In

Proceedings of the 9th Intl. Conference on High Performance Computing, Dec. 2002.

[25] Sweep3D Bench http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html

[26] D. Talb, D. Feitelson. Supporting Priorities and Improving Utilization of the IBM SP Scheduler Using Slack-Based Backfilling. In 13th Intl. Parallel Proc. Symp. (IPPS), pp.513-517, Apr. 1999.

[27] G. Utrera, J. Corbalán, J. Labarta. Implementing Malleability on MPI Jobs. In Proceedings of the Parallel Architecture and Compilation Techniques, 13th International Conference on (PACT'04), pp. 215-224, Antibes Juan-les-Pins, France, Sep 29 - Oct 03, 2004.

[28] G. Utrera, J. Corbalán, J. Labarta. Scheduling of MPI applications: Self Co-Scheduling.Euro-Par 2004, Lecture Notes in Computer Science 3149, pp 238-245.

[29] W. Ward Jr., C. L. Mahood, J. E. West. Scheduling Jobs on Parallel Systems Using a Relaxed Backfill Strategy. JSSPP 2002.

[30] Y. Zhang, H. Franke, J.Moreira, A. Sivasubramaniam. Improving Parallel Job Scheduling by Combining Gang Scheduling and Backfilling Techniques. IPDPS 2000.

[31] C. McCann and J. Zahorjan, "Processor allocation policies for message passing parallel computers". In SIGMETRICS Conf. Measurement & Modeling of Comput. Syst., pp. 19--32, May 1994.