A job scheduling approach for multi-core clusters based on virtual malleability

Gladys Utrera[1], Siham Tabik[2], Julita Corbalan[1], and Jesús Labarta[3]

[1] Technical University of Catalonia (UPC) 08034 Barcelona, Spain
{gutrera, juli}@ac.upc.edu
[2] University of Malaga, 29071 Malaga, Spain
stabik@uma.es
[3] Barcelona Supercomputing Center (BSC) 08034 Barcelona, Spain
jesus.labarta@bsc.es

Abstract. Many commercial job scheduling strategies in multi processing systems tend to minimize waiting times of short jobs. However, long jobs cannot be left aside as their impact on the performance of the system is also determinant. In this work we propose a job scheduling strategy that maximizes resources utilization and improves the overall performance by allowing jobs to adapt to variations in the load. The experimental evaluations include both simulations and executions of real workloads. The results show that our strategy provides significant improvements over the traditional EASY backfilling policy, especially in medium to high machine loads.

Keywords: job scheduling, MPI, malleability

1 Introduction

Modern computational clusters tend to have thousands of execution units [5]. In order to make these investments profitable, such clusters must have many users (clients). This leads to a large amount of job submissions that often exceeds the cluster capacity. Figure 1 shows a typical weekly load of the Marenostrum machine [1]. Many of these clusters are composed by nodes of multi-core processors. Multi-core processors have two or more complete computational cores integrated in the same chip. As a processing core can act as an independent processor or CPU, in this work terms core and CPU are synonyms.

A *job scheduling strategy* (JSS) is an algorithm that allocates resources to submitted jobs while applying system's administrative policies and priorities. A JSS has to deal with a wide variety of applications, from sequential to highly parallel codes, with execution times that varies from minutes to days. This scenario converts the comparison of two JSS into a difficult task. The high cost of the clusters usually makes user satisfaction the main objective for improving performance of the JSSs. For this reason, waiting times of short jobs that exceed by far their execution times are inadmissible. However, long jobs also play an important role in the performance which finally affect short jobs as well.

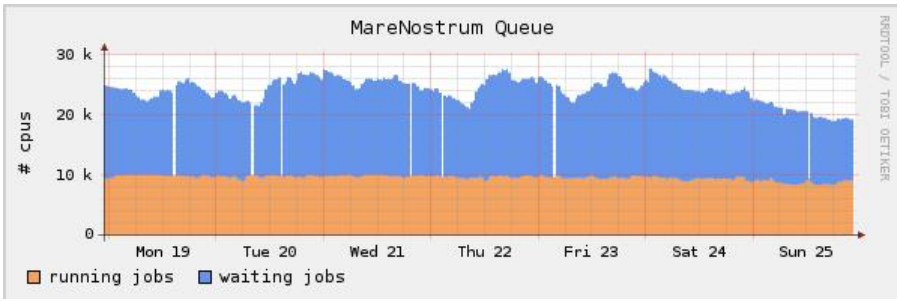
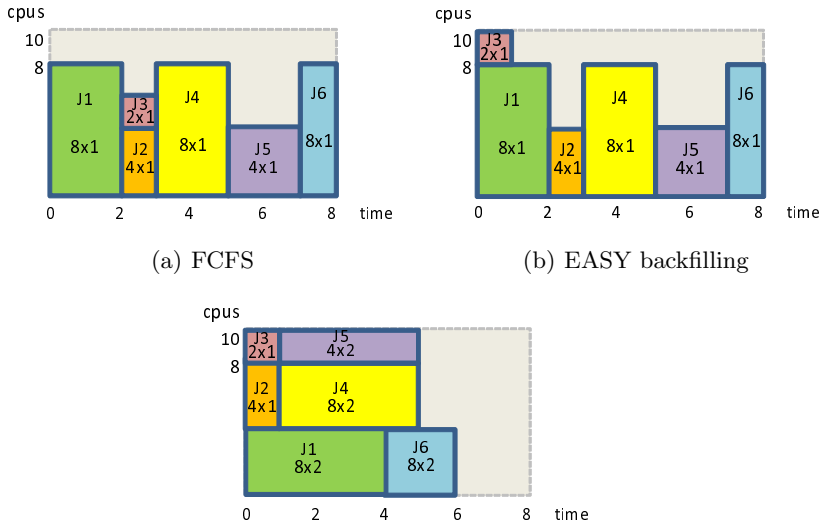


Fig. 1. Marenostrum load and wait queue during a week [1]



(a) FCFS

(b) EASY backfilling

(c) FCFS-malleable

Fig. 2. Job scheduling under : (a) FCFS, (b) EASY backfilling and (c) FCFS-malleable

To quantify the performance of JSSs, this work uses three metrics, namely the *response time*, *slowdown* and *fragmentation*. The first two metrics depend on the waiting and execution times of jobs while the last one measures the utilization of the computing system. The following classification of job flexibility is commonly accepted in the literature [14]: *rigid* jobs, which are compiled to be run with a specific and fixed number of processes; *moldable* jobs can be executed on multiple CPU partition sizes, but once the execution starts, these sizes cannot be modified; *malleable*, if the size of the assigned partition can also be modified during the execution. Moldable and malleable jobs would increase system utilization. However, they are not the common case in production systems. This paper presents a JSS, called *First Come First Served-malleable (FCFS-malleable)*, that

minimizes waiting times by maximizing system utilization no matter the jobs' execution time, nor their flexibility type. The proposed JSS is based on the idea of *Virtual Malleability* (VM)[24]. VM allows jobs to adapt to changes in the number of CPUs at runtime preserving the original number of processes. Figure 2 exemplifies two JSSs from literature, FCFS and EASY backfilling, and FCFS-malleable on a hypothetical machine of 10 CPUs where the x- and y- axis represent time and number of CPUs respectively. Each rectangle is labeled with the name of the job it represents (e.g. J1) and the number of processes per CPU (e.g., 8×1 means that each of the 8 processes is assigned to a different CPU). The jobs arrived to the wait queue in the following order: $J1, J2, J3, J4, J5, J6$ with execution times equal to $2, 1, 1, 2, 2, 1$ time units respectively.

Figure 2(a) shows the execution of the workload under FCFS. The execution of each job is delayed till there are enough CPUs for it. Meanwhile a group of available CPUs is not used even though there are jobs in the wait queue. For example, 2 CPUs are idle when J1 is running and 6 CPUs are idle when J5 is running which generates fragmentation.

Figure 2(b) shows how *EASY backfilling* [16] works. To alleviate the delayed execution and fragmentation problem it tries to forward jobs ahead in the queue when there are enough resources for them to be executed and provided they don't delay the first job in the queue. For example, J3 can start together with J1. However, EASY backfilling does not always find a suitable hole for a waiting job so fragmentation is not always eliminated (e.g. after J1 and J3 finish).

Figure 2(c) demonstrates how the JSS proposed in this paper behaves. As there are jobs in the wait queue, VM is applied to J1 (the oldest) so it starts its execution shrunk in 4 CPUs together with J2 and J3. Even shrinking J2 and J3, there are not enough CPUs for J4, so it has to wait and J2 and J3 can run expanded. This restriction is due to the fact that in this work VM can reduce the number of CPUs only up to half of the total assigned number. After J2 and J3 finish, J4 and J5 can start their executions shrunk. Finally, J6 starts expanded as there are no jobs in the wait queue [4].

The experimental results showed that FCFS-malleable overcomes EASY backfilling by 28% in average slowdown and 31% in average response time.

The main contributions of this work are as follows:

- A new JSS based on the concept of VM. The algorithm is easy to implement and does not require neither application recompilation nor any previous knowledge about the application.
- A study of the impact on the performance when applying VM to individual applications, taking into account intranode contention.
- Evaluations and comparison of the proposed JSS using both simulator and a runtime system.

[4] We assume that two processes are executed twice slower on a single CPU than on two separated CPUs. Different studies indicate that this time could be by far less then twice [22][26], since computation and communication can be overlapped.

2 Related work

The problem of scheduling and allocating resources to jobs in parallel systems has been addressed in previous research from two perspectives. While some works focus on providing support, libraries and runtime systems, to make individual applications malleable [20, 7, 8, 11, 24, 15] others attempt to integrate heuristics or new techniques to backfilling or FCFS policies [23, 10, 9, 21].

In [20] the moldability is applied in conjunction with *folding* [18] to reduce wait times. They create as many threads as the available number of CPUs and execute the job in the assigned partition. As the load increments, the partition of the latest arrived job is reduced to a half, freeing CPUs. The folding must be done at explicit synchronizing points in the source code. This generates additional wait time and requires extra effort from programmers and system support.

In [7] the authors modify the number of used CPUs at predetermined points of execution. The evaluation was carried out using workloads of only up to 10 jobs of NAS Benchmarks, with small classes like A and B. The inclusion of dynamic malleability support using a resource manager was also studied in [8, 11] They all require applications to be malleable or code modifications.

Certain levels of oversubscription improve resources utilization. The scheduling of jobs on the assigned CPUs can be done explicitly as in Gang Scheduling [13]. A static schedule of parallel communicating processes must be computed a priori, and a global context switch is used to coschedule these communicating processes. In this way, these processes have the illusion of running on a dedicated (but slower) system. These schemes usually require long time quanta to amortize the high context switch and synchronization costs, making the system less responsive for interactive and I/O-intensive applications. Furthermore, they keep the CPU idle while a process is performing I/O or is waiting for a message within its allotted time quantum. In [27] they propose an alternative to overcome this problem by matching pairs of processes: compute bound with I/O bound to share a time slice. The idea looks interesting, but they still have synchronization costs and extra effort to find, if possible, the correct matching. On the other hand, the experiments use applications with no more than 16 processes.

Implicit coscheduling [6, 28] tries to overcome the drawbacks of explicit coscheduling by relying on local schedulers, so that interactive and I/O-bound jobs are properly handled. They use the communication behavior of parallel processes to make scheduling decisions. Compared to explicit coscheduling, these strategies are easier to implement on clusters, and have better scalability and reliability characteristics basing exclusively on local knowledge.

In [25] it is shown that when a job shares CPUs with itself response time and stability are improved. The fact of oversubscribing CPUs was recently readdressed in multi-core systems [15]. Using several applications from different programming models they demonstrated that oversubscribing up to 8 tasks to each single CPU improves throughput over pure space sharing.

The most commonly used backfilling strategies are EASY backfilling and Conservative backfilling [19]. Conservative backfilling prioritizes predictability in response times and fairness, while EASY backfilling provides better response

times, especially for short jobs. Many previous related works focus on making backfilling strategies more flexible by integrating moldability with them [23]. The partition size for a job is selected based on its scalability and turnaround time by applying the Downey model [10]. Results demonstrate a gain in performance over pure backfilling and pure moldability [9] on individual applications. In [21] they propose a relaxation of conservative backfilling by allowing all waiting jobs to be delayed but only to some extent which improved response time predictability and resource utilization. Bounded slowdown obtained worse results than backfilling.

The JSS presented in this paper has the following advantages over state-of-art: no predefined synchronization points are required to vary the number of CPUs; jobs are unaware of the changes in the number of assigned CPUs and consequently there is no overhead derived from data redistribution, process creation or elimination at runtime; no need for recompilation; no job classification is required; no prior knowledge of the job is needed; all the jobs are candidate to be shrunk and moldability is not required. Finally, evaluations were done using both real executions with workloads made from benchmarks with large variations in data size, number of processes and communication degree. Simulations use well-known workload traces rather than synthetic workloads with few jobs and CPUs.

3 The FCFS-malleable job scheduling strategy

First, this section describes our JSS and defines the metrics that are used for the comparisons of JSSs. After that, a brief description of the implementation of the runtime system is provided. A deeper description can be found in [26].

3.1 FCFS-malleable algorithm

Algorithm 1 Code executed at job arrival

```
1: GetJobFromWaitQueue(J)
2: if FreeCpus ≥ J.cpusRequested then
3:    J.cpusAllocated ← J.cpusRequested
4:    Execute(J) //This function updates FreeCpus
5: else
6:    listOfOrderedJobs = SortByArrivalTime(listOfRunningJobs)
7:    //Selects as many jobs as required to execute J
8:    listOfCandidates = SelectCandidateJobs(listOfOrderedJobs, J)
9:    if listNotEmpty(listOfCandidates) then
10:      J.cpusAllocated = J.cpusAllocated/2;
11:      for V = JobsInList(listOfCandidates) do
12:         V.cpusAllocated = V.cpusAllocated/2
13:         Shrunk(V) //This function updates FreeCpus and NumJobsShrunk
14:      end for
15:      Execute(J)
16:   end if
17: end if
```

FCFS-malleable combines FCFS with VM. Applying VM to a job consists on running it on a pool of CPUs with a size less than or equal to the number of processes. In the case the number of CPUS is smaller than the number of processes, each CPU will have binded a queue of processes belonging to the same job. The size of this queue is called *multiprogramming level* (MPL). The maximum MPL was set to 2. This maximum level was chosen based on: memory bandwidth, number of CPUs per node, number of entry points to the interconnection network.

FCFS-Malleable is an event driven algorithm executed at job arrival and at job ending. Algorithm 1 shows the code executed when a new job arrives. At that event, the algorithm evaluates whether it is possible to start a new job depending on the available CPUs. If so, it starts the job with as many CPUs as requested (lines 3-4). Otherwise, the JSS tries to free CPUs and execute the job by applying VM to some jobs that are already running, including J if necessary. When a job finishes execution, if the wait queue is not empty, Algorithm 1 is applied, otherwise, running shrunk jobs are expanded to the newly freed CPUs. Several criteria to decide which job to shrink or expand first were evaluated: the oldest first, the one with less CPU utilization, the longest first and, the shortest first. Our experiments showed that the oldest one first is the best option. Line 8 of Algorithm 1 implements this option.

Metrics used for evaluations. In order to quantify the performance of our technique and make comparisons with others JSSs from bibliography three metrics were used: average response time, average slowdown and fragmentation.

Response time, is the time elapsed between the job submission and termination. This metrics evidences long jobs performance and is calculated by averaging the response times of all the jobs across a workload. For example, the average response times of the example shown in Figures 2(a), 2(b), and 2(c) are equal to $4.66, 4.33$ and 3.66 time units respectively.

Slowdown relates execution and wait time as it is shown in formula (1). This metrics indicates short jobs performance and is calculated by averaging the slowdown of all the jobs across a workload. It is important to note that to calculate the value of the average slowdown for the FCFS-malleable policy, the execution time in numerator of formula 1 is obtained using VM (i.e. with the overheads of running shrunk included). The average slowdown in Figures 2(a), 2(b), and 2(c) are equal to $3.5, 3.16$ and 2.5 time units respectively. The utilization of the system is usually addressed as the percentage of CPUs that are busy running jobs. As we are concerned only when there are jobs in the wait queue, we will use the fragmentation concept instead (see formula (2)). In the example provided in introduction, the fragmentation values are equal to 30% for Figures 2(a) and 2(b) and 0% for 2(c).

$$Slowdown = \frac{WaitTime + ExecutionTime}{ExecutionTimeExpanded} \tag{1}$$

$$Fragmentation = \frac{\sum_{whenWaitQueueNotEmpty}^{t=start\ to\ termination} freeCPUs}{WorkloadTotalTime * TotalCPUs} \tag{2}$$

3.2 Runtime system implementation

Let us now describe the runtime system and relevant details of implementation of the experimental framework. The runtime system is composed by a job scheduler (JS) and a runtime library. The JS receives as input a trace file and the JSS to apply. The trace file has identifications of the jobs, their arrival times and the number of requested CPUs [4]. The JS tracks information about node allocation, jobs in the wait queue, and already finished jobs.

The implementation of FCFS-malleable uses a library (VM library) that implements the concept of VM. The VM library was constructed using the *Message Passing Interface* [2], MPI, interposition mechanism. MPI was selected for being the most widely used and for its portability across shared and distributed memory architectures. The VM library is linked dynamically with jobs and communicates with the JS via TCP/IP sockets. This avoids the necessity of job recompilation. The library is in charge of CPU allocation and scheduling of processes. Process migrations are only allowed within a node and when VM is applied. Otherwise, processes remain binded to their assigned CPUs. The whole mechanism is transparent to the user. A job is said to run shrunk when is executed on a CPUs partition smaller than its number of processes. Processes belonging to the same job compete with themselves for the use of CPUs. A job is said to run expanded when is executed on a CPUs partition equal to its number of processes.

The scheduling of processes on a CPU is done by applying implicit coscheduling (see Section 2 for more details): only local knowledge (e.g. local communication events) is taken into account to make scheduling decisions. In particular Self co-scheduling [25] is applied. A running process yields the CPU and blocks immediately every time it executes a blocking operation (e.g. wait for a message that has not arrived yet). This type of scheduling promotes the overlapping of communication and computation phases.

The experiments were performed on a multi-core cluster with 10240 IBM Power PC 970MP cores at 2.3 GHz (2560 JS21 blades), 20 TB of main memory, 2510 nodes, and interconnection networks: Myrinet and Gigabit Ethernet. The operating system is Linux: SuSe Distribution. Each node has 4 cores sharing memory and each L2 cache is shared by every 2 cores.

4 Simulator

An event-driven simulator was constructed to extensively evaluate and compare JSSs. The simulator uses trace files in format of [4] as input and output. The following information about jobs is required to do the simulations: execution time, requested CPUs, requested time, CPU utilization[5]. Notice that FCFS-malleable may vary the number of CPUs of jobs at runtime. Thus, for FCFS-malleable we know only the *expanded* execution time of jobs. Next we provide a model to estimate the execution time of jobs when VM is applied to them.

[5] The field "CPU utilization" is used only by FCFS-malleable. In this work we refer to CPU utilization of a job to the average CPU time used by all its processes. That is the time when the CPU is doing useful work (i.e. computation).

Formula (3) arises from empirical observations. It estimates the execution time of a job when it runs isolated on different number of CPUs using the VM library. The value of MPL can vary during the execution time and is greater than 1 every time the job runs shrunk and is equal to 1 every time the job runs expanded. The parameter $CPUUtil$ is the percentage of CPU utilization when the job runs expanded. The parameter $execTime$ corresponds to the expanded execution time of the job. The parameter OV represents the overhead generated by the contention suffered when using the interconnection network. In our simulations, OV was set to random values between 0 and 1 as trace files have neither information about the communication-computation ratio nor the message sizes. We validated the proposed model by comparing results of simulations with real executions of several synthetic workloads.

$$estimatedIsolatedExecTime = \sum_{t=start}^{t=termination} execTime * MPL(t) * CPUUtil \tag{3}$$

Our final model is described by formula (4).

$$estimatedExecTime = estimatedIsolatedExecTime + execTime * OV \tag{4}$$

4.1 Validation of the simulator

A synthetic workload trace was constructed to validate the simulator by applying the model in [17]. The trace was adjusted to have 150 jobs to be launched during 2 hours with average machine loads from 30% to 90%.

	EASY backfilling						FCFS-malleable					
	Avg wait		Avg resp		Avg sld		Avg wait		Avg resp		Avg sld	
%load	S	R	S	R	S	R	S	R	S	R	S	R
30	55.0	60.0	107.0	113.0	2.1	2.1	1.6	1.3	73.5	73.3	1.2	1.4
50	69.6	74.0	121.3	125.0	6.5	6.5	13.2	13.8	93.5	93.0	3.2	3.2
70	93.0	100.0	145.2	152.0	10.7	10.3	36.8	32.0	118.3	113.0	6.0	5.7
90	175.0	162.0	228.0	214.0	19.0	16.0	118.0	111.0	220.0	198.9	10.8	9.6

Table 1. Comparison of average wait times, response times and slowdowns between simulator and runtime system

In order to execute the trace generated with [17] in the runtime system, we substituted applications in the trace for real applications. Applications in the trace were matched according to their execution time and number of processes. In this way interarrival times were kept with the same characteristics as of the original trace. We used the NAS Parallel Benchmarks [3] classes A, B, C and D and number of processes varying from 1 to 128. We chose these benchmarks as they include widely used kernels. We executed the synthetic traces under FCFS-malleable and EASY backfilling JSSs both on simulator and runtime system. Table 1 provides the average waiting time, response time and slowdown obtained with the simulator (S) and with the real execution (R). The average relative error of the simulator compared to the runtime system is equal to 7%. Considering that the average gain of FCFS-malleable over EASY backfilling in the runtime system is around 30% we concluded that this error is acceptable.

5 Results and analysis

Cleaned traces from Parallel Workload Archive [4] were used in our experiments. A cleaned trace does not contain flurries of activity by individual users which may not be representative of normal usage. Table 2 summarizes the workloads characteristics.

The columns show the names of the used workloads, total number of CPUs in the machine, number of jobs in the workload, average CPU utlization, average CPU utilization by long jobs and the ratio between the average number of requested CPUs by the machine capacity. For example, the workload in figure 1 has this ratio equal to 2. We have classified long jobs as the ones with number of processes greater than 64 and execution times greater than 8 hours and short jobs as the ones with execution times less than 10 minutes.

Workload	Cpus	Jobs	Avg CPU Util	Avg long jobs CPU Util	Req.Cpus/Cpus
CTC	430	20K-25K	57 %	70%	5.8
SDSC Blue	1152	20K-25K	23 %	70%	3.8
SDSC	128	40K-45K	66 %	90%	8.8

Table 2. Description of the workload log traces used for simulation

The CTC trace contains records from IBM SP2 located at the Cornell Theory Center. SDSC and SDSC Blue traces are from the San Diego Supercomputing Center. We now present the experimental results obtained from simulations using the workloads traces from Table 2.

5.1 Experimental results

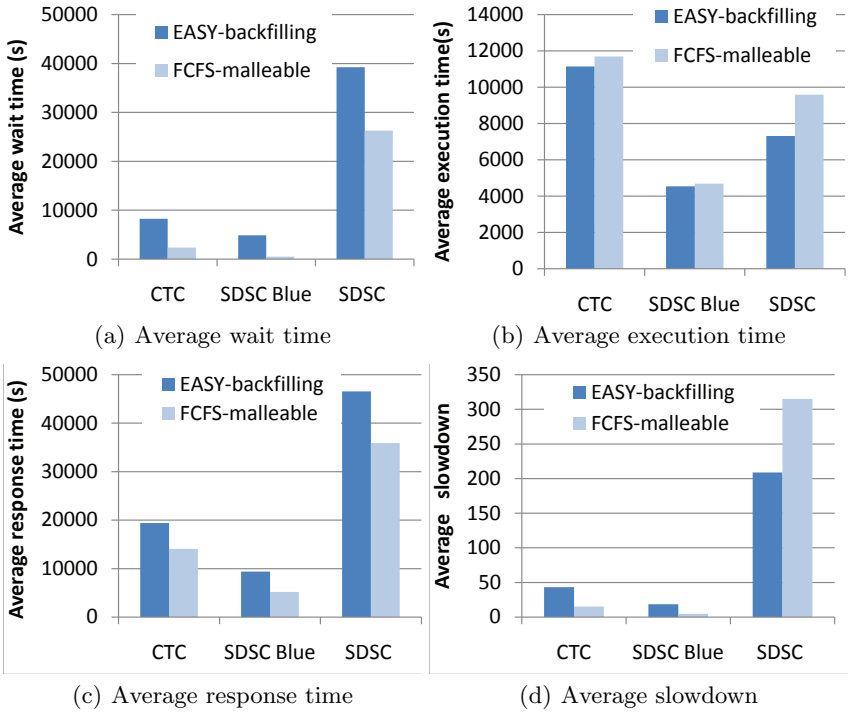
Figures 3(a), 3(b), 3(c) and 3(d) show the average wait time, execution time, response time and slowdown respectively for CTC, SDSC and SDSC Blue workloads under FCFS-malleable and EASY backfilling JSSs[6].

FCFS-malleable JSS obtained better average response time in all the traces, especially in trace SDSC Blue. This workload contains jobs with low CPU utilization, which leads to higher degree of overlap of communication and communication. In addition, this workload has no sequential jobs, thus all the jobs are eligible for applying VM.

As it was expected, average execution times are larger under FCFS-malleable due to the reduction on the number of CPUs. However, these execution times are not twice larger than the execution times in EASY backfilling.

FCFS-malleable obtains substantially better average slowdowns in CTC and SDSC Blue but not in SDSC. This means that the performance of short jobs is degraded in that workload. Analyzing this penalization we found that it was due to the strong presence of sequential jobs and the high CPU utilization of long jobs. EASY backfilling outperformed FCFS-malleable only on jobs with execution time less than 3 minutes and number of processes less than 16. EASY backfilling failed to find a suitable hole to forward long sequential jobs or with high degree of parallelism. This study can be found in [26].

[6] Variations of backfilling policies are used in most of the Top50 machines[12]. EASY backfilling is used as a reference for performance comparison in almost every job scheduling research. That is why we chose EASY backfilling for our comparisons.



(a) Average wait time

(b) Average execution time

(c) Average response time

(d) Average slowdown

Fig. 3. Average wait, execution, response time and slowdown for CTC, SDSC and SDSC Blue under FCFS-malleable and Easy backfilling

The CPU utilization for long jobs is the highest for the SDSC workload (see Table 2). We re-simulated the SDSC workload trace varying the average value of CPU utilization of long jobs between 60% and 100%. We observed that for long jobs with CPU utilization under 90% the average slowdown for FCFS-malleable is smaller than for EASY backfilling. Due to lack of space we omitted that study here, but it can be found in [26].

Workload	EASY backfilling	FCFS-malleable
CTC	0.75	0.91
SDSC Blue	0.76	0.98
SDSC	0.89	1.47

Table 3. Average MPL

FCFS-malleable managed to eliminate fragmentation in all the workloads while EASY backfilling had fragmentation percentages from 6 for CTC to 14 for SDSC. Table 3 shows the average MPL of the three workloads for EASY backfilling and FCFS-malleable. MPL was calculated by averaging the total number of processes in the system per CPU. FCFS-malleable has average MPL below 2 (the maximum). This means that the workloads have variations so that jobs could expand from time to time decreasing in this way the average value of MPL. The value of the average MPL for the SDSC trace means that half of the CPUs run shrunk jobs all the time.

6 Conclusions and Future work

In this work we proposed a new job scheduling strategy (JSS) for multi-core clusters: FCFS-malleable. Evaluations on the target architecture were carried out using a job scheduler and a runtime system implemented for that purpose. In addition, to extend evaluations to workloads from production systems, a simulator was constructed. Experimental results showed that FCFS-malleable outperforms EASY backfilling by 28% in average slowdown and by 31% in average response time. In addition, our JSS reduces fragmentation thanks to its capability to adapt jobs to available resources by shrinking and expanding them.

Although in this work we compete with backfilling, our JSS can be combined with it to take the most of both strategies. We are currently evaluating this approach. Memory bandwidth was not taken into account in the current study. We are working on an accurate estimation of the overhead caused by limited memory bandwidth.

Acknowledgements. This work was supported by the Ministry of Science and Technology of Spain under contracts TIN2007-60625, TIN2006-01078, TIN2010-16144 and Juan de la Cierva and the postdoctoral contract funded by the University of Malaga.

References

1. Marenostrum. http://www.bsc.es/marenostrum-support-services.
2. MPI library. http://www.mcs.anl.gov/research/projects/mpi/.
3. NAS Parallel Benchmarks. http://www.nas.nasa.gov/Resources/Software/npb.html.
4. Parallel workload archive. Available at http://www.cs.huji.ac.il/labs/parallel/workload/.
5. Top500 supercomputers sites. http://www.top500.org/.
6. A. C. Arpaci-Dusseau. Implicit coscheduling: coordinated scheduling with implicit information in distributed systems. *ACM Trans. Comput. Syst.*, 19:283–331, August 2001.
7. J. Buisson, O. Sonmez, H. Mohamed, W. Lammers, and D. Epema. Scheduling malleable applications in multicluster systems. In *In Proc. of the IEEE International Conference on Cluster Computing 2007*, pages 372–381, 2007.
8. M. C. Cera, Y. Georgiou, O. Richard, N. Maillard, and P. O. A. Navaux. Supporting malleability in parallel architectures with dynamic cpusets mapping and dynamic MPI. In *Proc. of the 11th International conference on Distrib. computing and networking*, ICDCN'10, pages 242–257, Berlin, Heidelberg, 2010. Springer-Verlag.
9. W. Cirne and F. Berman. Using moldability to improve the performance of supercomputer jobs. *J. Parallel Distrib. Comput.*, 62:1571–1601, October 2002.
10. A. B. Downey. A model for speedup of parallel programs. Technical report, University of California at Berkerley, 1997.
11. K. El Maghraoui, T. J. Desell, B. K. Szymanski, and C. A. Varela. Dynamic malleability in iterative MPI applications. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '07, pages 591–598, Washington, DC, USA, 2007. IEEE Computer Society.
12. C. Ernemann, M. Krogmann, J. Lepping, and R. Yahyapour. Scheduling on the top 50 machines. In *JSSPP'04*, pages 17–46, 2004.
13. D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306 – 318, 1992.

14. D. G. Feitelson and L. Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *In Job Scheduling Strategies for Parallel Processing*, pages 1–26. Springer-Verlag, 1996.
15. C. Iancu, S. Hofmeyr, Y. Zheng, and F. Blagojevic. Oversubscription on multicore processors. In *In 24th International Parallel and Distributed Processing Symposium (IPDPS*, pages 1–11, 2010.
16. D. A. Lifka. The ANL/IBM SP scheduling system. In *In Job Scheduling Strategies for Parallel Processing*, pages 295–303. Springer-Verlag, 1995.
17. U. Lublin and D. G. Feitelson. The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. *Journal of Parallel and Distributed Computing*, 63:2003, 2001.
18. C. McCann and J. Zahorjan. Processor allocation policies for message-passing parallel computers. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '94, pages 19–32, New York, NY, USA, 1994. ACM.
19. A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, 2001.
20. J. Padhye and L. W. Dowdy. Dynamic versus adaptive processor allocation policies for message passing parallel computers: An empirical comparison. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 224–243, London, UK, 1996. Springer-Verlag.
21. A. C. Sodan and W. Jin. Backfilling with fairness and slack for parallel job scheduling. *Journal of Physics: Conference Series*, 256(1):012–023, 2010.
22. V. Subotic, J. Labarta, and M. Valero. Simulation environment for studying overlap of communication and computation. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 115–116, White Plains, NY, March 2010.
23. R. Sudarsan and C. J. Ribbens. Scheduling resizable parallel applications. *Parallel and Distributed Processing Symposium, International*, 0:1–10, 2009.
24. G. Utrera, J. Corbalán, and J. Labarta. Implementing malleability on MPI jobs. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 215–224, Washington, DC, USA, 2004. IEEE Computer Society.
25. G. Utrera, J. Corbalán, and J. Labarta. Scheduling of MPI applications: Self-coscheduling. In *Euro-Par 2004 Conference, 31th Aug - 3rd Sept Italy Proceedings*, volume 3149 of *Lecture Notes in Computer Science*, pages 238–245. Springer, 2004.
26. G. Utrera, S. Tabik, J. Corbalán, and J. Labarta. A job scheduling approach to reduce waiting times. Technical report, Technical University of Catalonia, UPC-DAC-RR-2012-1, Oct. 2011.
27. Y. Wiseman and D. G. Feitelson. Paired gang scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 14(6):581–592, 2003.
28. Y. Zhang, A. Sivasubramaniam, J. Moreira, and H. Franke. A simulation-based study of scheduling mechanisms for a dynamic cluster environment. In *Proceedings of the 14th international conference on Supercomputing*, ICS '00, pages 100–109, New York, NY, USA, 2000. ACM.