Parallel Job Scheduling Under Dynamic Workloads*

Eitan Frachtenberg[1,2], Dror G. Feitelson[2], Juan Fernandez[1], and Fabrizio Petrini[1]

[1]CCS-3 Modeling, Algorithms, and Informatics Group
Computer and Computational Sciences (CCS) Division
Los Alamos National Laboratory (LANL)
{eitanf,juanf,fabrizio}@lanl.gov

[2]School of Computer Science and Engineering
The Hebrew University, Jerusalem, Israel
feit@cs.huji.ac.il

Abstract

Jobs that run on parallel systems that use gang scheduling for multiprogramming may interact with each other in various ways. These interactions are affected by system parameters such as the level of multiprogramming and the scheduling time quantum. A careful evaluation is therefore required in order to find parameter values that lead to optimal performance. We perform a detailed performance evaluation of three factors affecting scheduling systems running dynamic workloads: multiprogramming level, time quantum, and the use of backfilling for queue management — and how they depend on offered load. Our evaluation is based on synthetic MPI applications running on a real cluster that actually implements the various scheduling schemes. Our results demonstrate the importance of both components of the gang-scheduling plus backfilling combination: gang scheduling reduces response time and slowdown, and backfilling allows doing so with a limited multiprogramming level. This is further improved by using flexible coscheduling rather than strict gang scheduling, as this reduces the constraints and allows for a denser packing.

Keywords: Cluster computing, dynamic workloads, job scheduling, gang scheduling, parallel architectures, heterogeneous clusters, STORM, flexible coscheduling

*This work was supported by the U.S. Department of Energy through Los Alamos National Laboratory contract W-7405-ENG-36, and by the Israel Science Foundation (grant no. 219/99).

1 Introduction

Multiprogramming on parallel machines may be done using two orthogonal mechanisms: time slicing and space slicing [4]. With time slicing, each processor runs processes belonging to many different jobs concurrently, and switches between them. With space slicing, the processors are partitioned into groups that serve different jobs. Gang scheduling (GS) is a technique that combines the two approaches: all processors are time-sliced in a coordinated manner, and in each time slot they are partitioned among multiple jobs.

Early evaluations of gang scheduling showed it to be a very promising approach [6, 11]. Gang scheduling provided performance that was similar to that of dynamic partitioning, but without requiring jobs to be programmed in special ways that tolerate reallocations of resources at runtime. The good performance was attributed to the use of preemption, which allows the system to recover from unfortunate scheduling decisions, including those that are a natural consequence of not knowing the future.

More recent research has revisited the comparison of gang scheduling with other schemes, and has led to new observations. One is that gang scheduling may be limited due to memory constraints, and therefore its performance is actually lower than what was predicted by evaluations that assumed that memory was not a limiting factor. The reason that memory is a problem is the desire to avoid paging, as it may cause some processes within parallel jobs to become much slower than other processes, consequently slowing down the whole application. The typical solution is to allow only a limited number of

jobs into the system, effectively reducing the level of multiprogramming [2, 22]. This hurts performance metrics such as the average response time because jobs may have to wait a long time to run.

Another observation is that alternative scheduling schemes, such as backfilling [12], may provide similar performance. Backfilling is an optimization that improves the performance of pure space slicing by using small jobs from the end of the queue to fill in holes in the schedule. To do so, it requires users to provide estimates of job runtimes. Thus it operates in a more favorable setting than gang scheduling, that assumes no such information. Moreover, moving short jobs forward achieves an effect similar to the theoretical "shortest job first" algorithm, which is known to be optimal in terms of average response time.

Our goal is to improve our understanding of these issues, by performing an up-to-date evaluation of gang scheduling and a comparison with other schemes. To do so, we must also find good values for its main parameters, namely the multiprogramming level (MPL) and the length of the time slicing quantum. As these parameters are intimately related to the dynamics of the workload, the evaluation is done using a dynamic workload model.

To achieve this goal, we have implemented several scheduling algorithms on top of STORM, a scalable, high-performance, and flexible resource management system [10]. STORM allows the incorporation of many job scheduling algorithms with relative ease, and monitors their performance while running actual MPI application on a real cluster.

The focus of this paper is dynamic workloads, with arrivals and execution times of jobs that are representative of real-world scenarios. Evaluations using dynamic workloads are extremely important, as they expose the ability of the scheduler to stream jobs through the system, and the limits on the achievable utilization. This is essentially a feedback effect, where a good scheduler packs the input load better than other schedulers, causing jobs to flow through the system faster, which in turn decreases the momentary system load and makes it easier to handle additional incoming jobs. This paper is thus an extension of work reported in [9], where we focused on simple, static workloads, and evaluated the properties of scheduling schemes under various controlled job mixes.

The rest of this paper is organized as follows. The next section details the methodology and environment we use for our experiments. Section 3 studies the effect of different MPLs, and how this depends on the use of backfilling. In Section 4, we establish the range of usable time quanta for dynamic workloads in STORM, in two different cluster architectures. Once these parameters are understood, Section 5 proceeds to compare the different scheduling schemes under varying system load. Finally, we offer concluding remarks and directions for future use in Section 6.

2 Background and Methodology

This section describes important issues and choices for our evaluation methodology. Many of the choices we made are similar to those of Zhang et al. in [22], which also conducted similar evaluations. The main differences in methodology between their work and ours, is the workload model being used and the evaluation environment (simulation vs. emulation, respectively).

2.1 Scheduling Algorithms

For this study, we have chosen to compare the following scheduling schemes:

- First-Come-First-Serve scheduling (FCFS)
- Vanilla gang scheduling (GS)
- Flexible variants of coscheduling: spin-block (SB), and flexible coscheduling (FCS)
- Backfilling as in EASY
- Combinations of EASY with other schemes

Gang scheduling (also known as explicit coscheduling) was first proposed by Ousterhout in [16] as a time-sharing scheduling method that uses global synchronization to switch the running jobs at regular intervals (time slices). Like with batch scheduling, GS gives each job the illusion of a dedicated machine, while maintaining relatively short response times for short jobs. On the other

hand, GS requires global coordination of processes, which is not often implemented scalably.

Two-phase spin-blocking is a scheduling mechanism in which processes busy-wait when they reach a synchronization point, but if synchronization is not achieved, they block. The scheduling scheme is to use this in conjunction with the normal, priority-based local scheduling of Unix systems. This results in a coscheduling algorithm similar to implicit coscheduling [1]. Unlike GS, it uses only implicit information to synchronize communicating processes (gangs), without an explicit synchronization mechanism. SB was found to be very efficient and simple to implement, excelling mostly for loosely-coupled jobs.

Flexible coscheduling (FCS) is a hybrid scheme that uses a global synchronization mechanism combined with local information collection [9]. In a nutshell, FCS monitors communication behavior of applications to classify them into one of three classes: (1) fine-grained communicating processes that require coscheduling, (2) loosely-coupled or non-communicating processes that do not require coscheduled running, and (3), frustrated processes, that require coscheduling, but are not able to communicate effectively due to load-imbalance problems. FCS then uses this information to schedule each process based on its class.

All these algorithms make use of a queue for jobs that arrive and cannot be immediately allocated to processors. The allocation of jobs from the queue can be handled with many heuristics, such as first-come-first-serve, shortest-job-first, backfilling, and several others. We chose to implement and use EASY backfilling [12], where jobs are allowed to move forward in the queue if they do not delay the first queued job.

Zhang et al. studied these issues in [22] and found that combining backfilling with gang-scheduling can reduce the average job slowdown when the MPL is bounded (our experiments described in Section 3 agree with these results). They also point out that for coscheduling algorithms such as GS, there is no exact way of predicting when jobs will terminate (or start), because the effective multiprogramming level varies with load. To estimate these times, we use the method they recommend, which is to multiply the original run time estimate by the maximum MPL.

2.2 Evaluation using Emulation

In this paper we study the properties of several scheduling algorithms in various complex scenarios, involving relatively long dynamic workloads. Since the number of parameters that can affect the performance of such systems is quite large, We have decided to use an actual implementation of a scheduling system on a real cluster. By using and measuring a real system, we are able to take into account not only those factors we believe we understand, but also unknown factors, such as complex interactions between the applications, the scheduler, and the operating system.

We do make however two simplifying assumptions, where we believe we need not or cannot encapsulate all the nuances of the real world:

- We use a synthetic workload model instead of a real workload trace. The synthetic workload model is believed to be representative of several real workload traces [13], and allows us more flexibility than real traces, since we can use it for any given machine size, workload length, or offered load.
- We do not run real applications, which would have forced us to study their scalability and communication properties (a sizable and worthy research subject in its own merit). Instead, we use a simple parallel synthetic application, where we can control its most important properties with a high degree of precision.
- We reduce all times from the model by a constant factor in order to complete the runs faster.

The first two issues are detailed in Sections 2.3 and 2.4 below.

2.3 Workload

The results of performance evaluation studies may depend not only on the system design but also on the workload that it is processing [5]. It is therefore very important to use representative workloads that have

the same characteristics as workloads that the system may encounter in production use.

One important aspect of real workloads is that they are dynamic: jobs arrive at unpredictable times, and run for (largely) unpredictable times. The set of active jobs therefore changes with time. Moreover, the number of active jobs also changes with time: at times the system may be empty, while at others many jobs are waiting to receive service. The overall performance results are an average of the results obtained by the different jobs, which were actually obtained under different load conditions.

These results also reflect various interactions among the jobs. Such interactions include explicit ones, as jobs compete for resources, and implicit ones, as jobs cause fragmentation that affects subsequent jobs. The degree to which such interactions occur depends on the dynamics of the workload: which jobs come after each other, how long they overlap, etc. It is therefore practically impossible to evaluate the effect of such interactions with static workloads in which a given set of jobs are executed at the same time.

The two common ways to evaluate a system under a dynamic workload are to either use a trace from a real system, or to use a dynamic workload model. While traces have the benefit of reflecting the real workload on a specific production system, they also risk not being representative of the workload on other systems. We therefore use a workload model proposed by Lublin [13], which is based on invariants found in traces from three different sites, and which has been shown to be representative of other sites as well [20]. This also has the advantage that it can be tailored for different environments, e.g. systems with different numbers of processors.

While the workload model generates jobs with different arrival times, sizes, and runtimes, it does not generate user estimates of runtimes. Such estimates are needed for EASY backfilling. Instead of using the real runtime as an estimate, we used loose estimates that are up to 5 times longer. This is based on results from [15], which show that overestimation commonly occurs in practice and is beneficial for overall system performance.

Using a workload model, one can generate a workload of any desired size. However, large workloads

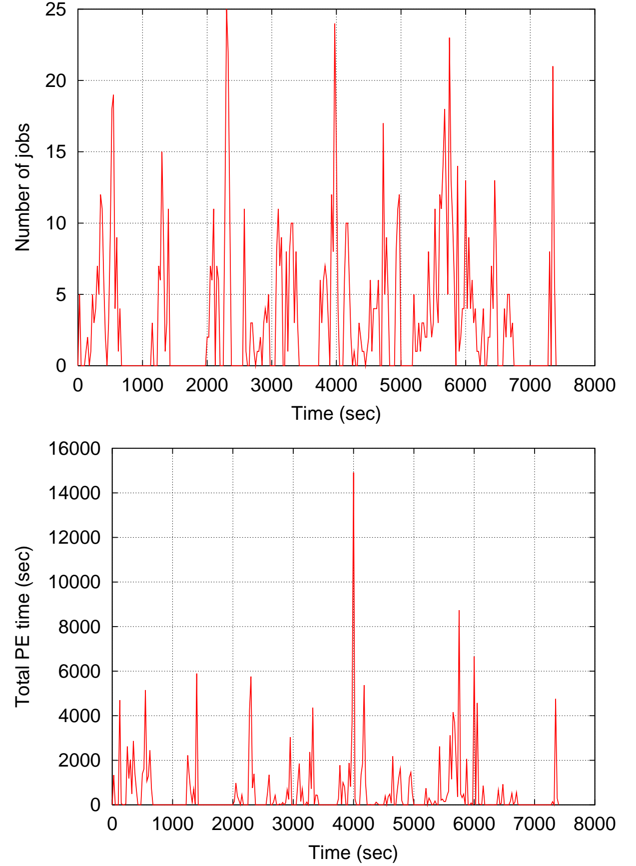


Figure 1: Model of 1000 arrivals in terms of jobs and requested CPU time. The X axis is emulated time, which is the model time divided by 100. Data is grouped into bins of 25 sec.

will take a long time to run. We therefore make do with a medium-sized workload of 1000 jobs, that arrive over a period of about 8 days. The job arrivals are bursty, as shown in Fig. 1, matching observations from real workloads. Fig. 2 shows the distribution of job sizes. As is often seen in real workload traces, most jobs sizes tend to be small (with the median here at just under 4 PEs), and biased toward powers of two.

To enable multiple measurements under different conditions, we shrink time by a factor of 100. This means that both runtimes and inter-arrival times are divided by a factor of 100, and the 8 days can be emulated in about 2 hours. Using the raw workload data, this leads to a very high load of about 98% of the system capacity. To check other load conditions

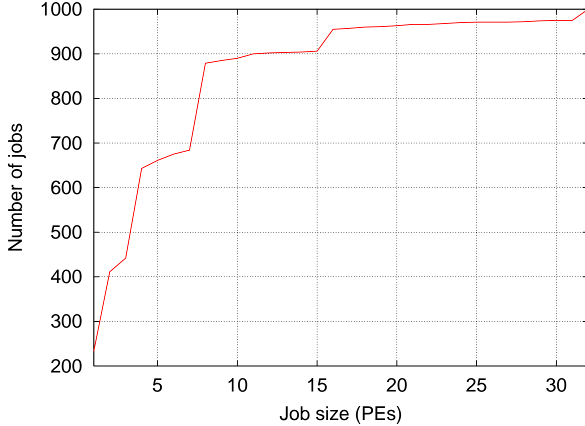


Figure 2: Cumulative distribution of job sizes.

we divide the execution times by another factor, to reduce the jobs' run time and thus reduce the load.

2.4 Test Application

A large part of High Performance Computing (HPC) software can be modeled using the bulk-synchronous parallel (BSP) model. In this model a computation involves a number of *supersteps*, each having several parallel computational threads that synchronize at the end of the superstep [21, 7]. We chose to use a synthetic test application based on this model, to enable easy control over important parameters, such as execution time, computation granularity and pattern, and so forth. Our synthetic application consists of a loop that computes for some time, and then exchanges information with its nearest neighbors in a ring pattern. The amount of time it spends computing in each loop (the computation granularity) is chosen randomly with equal probability from one of three values: fine-grained (5 ms), medium-grained (50 ms), and coarse-grained (500 ms).

2.5 Experimental Environment

The hardware used for the experimental evaluation was the 'Crescendo' cluster at LANL/CCS-3. This cluster consists of 32 compute nodes (Dell 1550), one management node (Dell 2550), and a 128-port Quadrics switch [17] (using only 32 of the 128 ports). Each compute node has two 1 GHz Pentium-III processors, 1 GB of ECC RAM, two independent 66 MHz/64-bit PCI buses, a Quadrics QM-400 Elan3 NIC

[17, 18, 19] for the data network, and a 100 Mbit Ethernet network adapter for the management network. All the nodes run Red Hat Linux 7.3 with Quadrics kernel modifications and user-level libraries. We further modified the kernel by changing the default HZ value from 100 to 1024. This has a negligible effect on operating system overhead, but makes the Linux scheduler re-evaluate process scheduling every ≈ 1 ms. As a result scheduling algorithms (in particular SB) become more responsive [3].

We perform our evaluations by implementing the desired scheduling algorithms in the framework of the STORM resource manager [10]. The key innovation behind STORM is a software architecture that enables resource management to exploit low-level network features. As a consequence of this design, STORM can enact scheduling decisions, such as a global context switch or a heartbeat, in a few hundreds of microseconds across thousands of nodes. In this environment, it is relatively easy to implement working versions of various job scheduling and queuing schemes, using either global coordination, local information, or both.

STORM produces log files of each run, containing detailed information on each job (e.g. its arrival, start, and completion times, as well as algorithm-specific information). We then use a set of scripts to analyze these log files and calculate various metrics. In this paper, we mostly use average response time (defined as the difference between the completion and arrival times) and average bounded slowdown. The bounded slowdown of a job is defined in [8], and we modified it to make it suitable for time-sharing multiprogramming environments by using:

$$Bounded\,Slowdown = \max\left\{\frac{T_w + T_r}{\max\{T_d, \tau\}}, 1\right\}$$

Where:

- T_w is the time the job spends in the queue.
- T_r is the time the job spends running.
- T_d is the time the job spends running in dedicated (batch) mode.

- τ is the "short-job" bound parameter. We use a value of 10 seconds of real time (0.1 sec emulated).

In some cases, we also divide the jobs into two halves: "short" jobs, defined as the 500 jobs with the shortest execution time, and "long" jobs — the complementing group. For this classification, we always use the execution time as measured with FCFS (batch) scheduling, so that the job groups remain the same even when job execution times change with different schedulers.

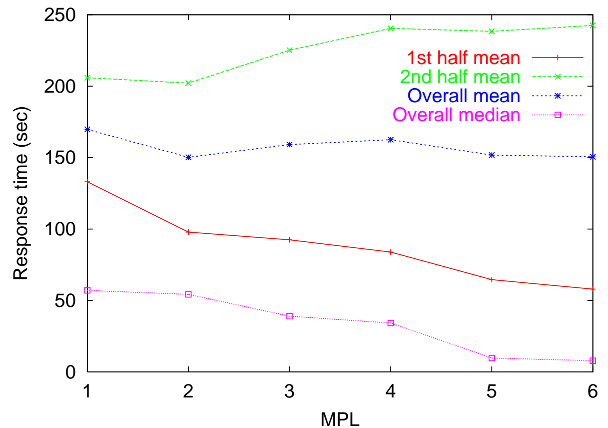


Figure 3: Response time with different MPLs.

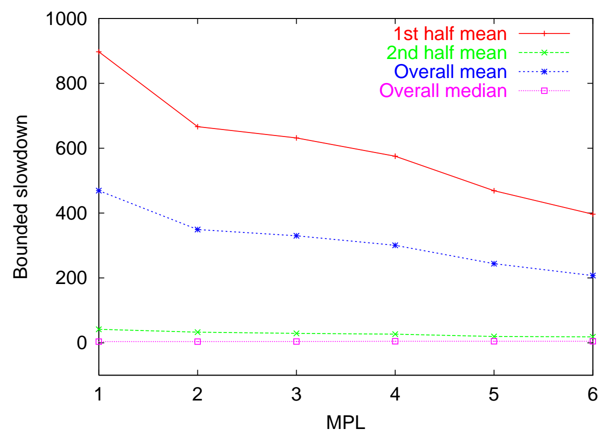


Figure 4: Bounded slowdown with different MPLs.

3 Effect of Multiprogramming level

3.1 Experiment Description

The premise behind placing a limit on the MPL is that scheduling algorithms should not dispatch an unbounded number of jobs concurrently. One obvious reason for this is to avoid exhausting the physical memory of nodes. We define the MPL to be the maximum allowable over-subscription of processors. Naturally, the MPL for the FCFS scheme is always one, whereas for coscheduling algorithms it is higher than 1. We study this property in this section.

We set out to test the effect of the MPL on gang-scheduling, with two goals in mind: (1) obtain a better understanding on how a limited multiprogramming level affects serving of dynamic workloads, and (2) find a good choice of an MPL value for the other sets of experiments. In practice, the question of how the MPL affects scheduling is sometimes moot, since often applications require a sizable amount of physical memory. In this case, multiprogramming several applications on a node will generally lead to paging and/or swapping, having a detrimental effect on performance that is significantly more influential than any possible scheduling advantage. While some applications are not as demanding, or can be "stretched" to use a smaller memory footprint, we do accept the existence of a memory wall. Moreira et al. showed in [14] that an MPL level of 5 provides in practice similar performance to that of an infinite MPL, so we put the bound on the maximum value the MPL can reach at 6.

To study the effect of the MPL, we use GS and a workload file consisting of 1000 jobs with an average offered load of $\approx 74\%$. GS was chosen due its relative popularity (being the most basic coscheduling method), and its simplicity. The 74% load value was chosen so that it would stress the system enough to bring out the difference between different MPL values, without saturating it[1]. We ran the test with all MPL values from 1 to 6, and analyzed the resulting log files.

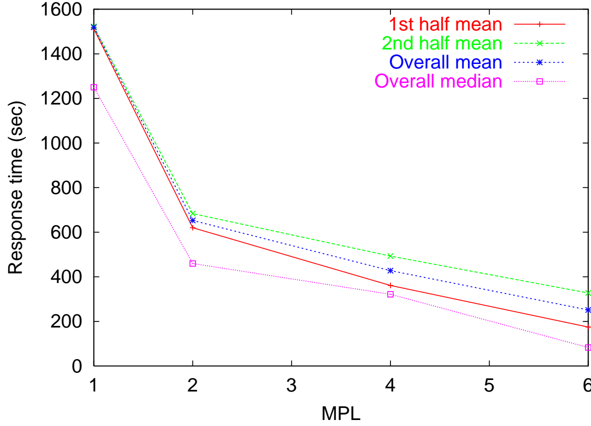


Figure 5: Response time with different MPLs (no backfilling).

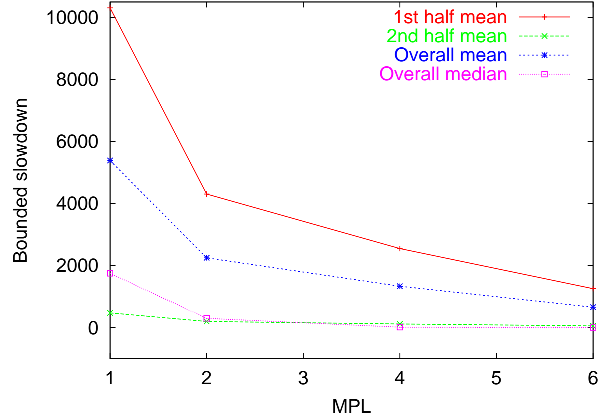


Figure 6: Bounded slowdown with different MPLs (no backfilling).

3.2 Results and Discussion

Figure 3 shows the effect of the MPL on response time. The average response time decreases somewhat when changing from batch scheduling (MPL 1) to coscheduling (MPL 2 or more), and then stays at about the same level. This improvement corresponds to an enhanced ability of the scheduler to keep less jobs waiting in the queue. Having more available slots, the scheduler can dispatch more jobs from the queue, which is particularly significant for short jobs: These can start running soon after their arrival time, complete relatively quickly, and clear the system. To confirm this claim, let us observe that the average response time for the shorter 500 jobs indeed decreases for higher MPLs, while that of the longer 500 jobs increases at a similar rate. Furthermore, the median response time (which is dominated by the shorter jobs), decreases monotonically.

This effect becomes more pronounced when looking at the bounded slowdown (Fig. 4). We can clearly see that the average slowdown shows a consistent and significant decrease as the MPL increases. This is especially pronounced for the first 500 jobs, that show a marked improvement in slowdown, especially when moving from MPL 1 to MPL 2.

Still, the improvement of these metrics as the MPL increases might seem relatively insignificant compared to our expectations and previous results. To better understand why this might be the case, we repeated these measurements, but with backfilling disabled. Figures 5 and 6 show the results of these experiments. Here we can observe a sharp improvement in both metrics when moving from batch to gang scheduling, and a steady improvement after that. The magnitude of the improvement is markedly larger than that of the previous set. This might be explained by the fact that when backfilling is used, we implicitly include some knowledge of the future, since we have estimates for job run times. In contrast, gang-scheduling assumes no such knowledge and packs jobs solely by size. Our results indicate that some knowledge of the future (job estimates) and consequentially, their use in scheduling decisions as employed by backfilling, renders the advantages of a higher MPL less pronounced. These results also agree with the simulated evaluations in [22].

Having the best overall performance, we use an MPL of 6 in the other sets of experiments, combined with backfilling.

[1] repeating this experiment with an average offered load of $\approx 78\%$, gave similar results. Above that, GS becomes saturated for this workload

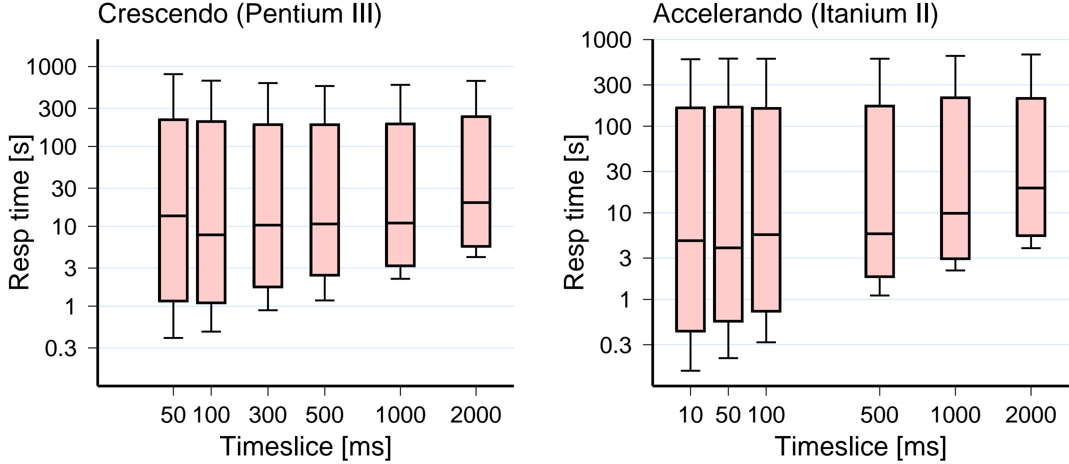


Figure 7: Distributions of response times for different time quanta (logscale). Each bar depicts the 5%, 25%, 50%, 75%, and 95% percentiles.

4 Effect of Time Quantum

4.1 Experiment Description

Another important factor that can have an effect on a scheduling system's performance and responsiveness is the time quantum. Scheduling schemes that employ time sharing in the form of distinct time slots, have to make a choice of the duration of each time slot, the time quantum. A short time slice generally increases the system's responsiveness, since jobs do not have to wait for long periods before being inserted into the system, which benefits mostly short and interactive jobs. On the other hand, a very small time quantum can significantly tax the system's resources due to the overhead incurred by frequent context-switches. In [10] it was shown that STORM can effectively handle very small time quanta, in the order of magnitude of a few milliseconds, for simple static workloads. This is not necessarily the case in our experimental setup, that includes a complex dynamic workload, and a relatively high MPL value, increasing the load on the system (more pending communication buffers, cache pressure, etc.).

For this set of experiments, we use again the workload file with 1000 jobs and an average offered load of $\approx 74\%$. We ran the scheduler with different time quantum values, ranging from 2 seconds down to 50 ms. Since the overhead caused by short time slices is largely affected by the specific architecture, we were also interested in repeating these experiments on a different architecture. To that end, we also ran the same experiment on the 'Accelerando' cluster at LANL/CCS-3, where each node contains two 1 GHz Itanium-II CPUs, 1 GB of RAM, a PCI-X bus, and a QsNET network similar to the one we use on the 'Crescendo' cluster.

4.2 Experimental Results

Fig. 7 shows the distribution of response times with different time quanta, for the two cluster architectures. Each bar shows the median response time (central horizontal divider), the 25% and 75% percentiles (top and bottom edges of box), and the 5% and 95% percentiles (whiskers extending up and down). The 5% rank is defined by short jobs, and monotonically decreases with shorter time quanta, which confirms our expectations. The 95% rank represents all but the longest jobs, and does not change much over the quanta range, except for the 50 ms quantum on Crescendo, where response times for most jobs increase slightly, probably due to the overhead associated with frequent context switching. The different effect of the quantum on the 5% and 95% ranks suggests that short jobs are much more sensi-

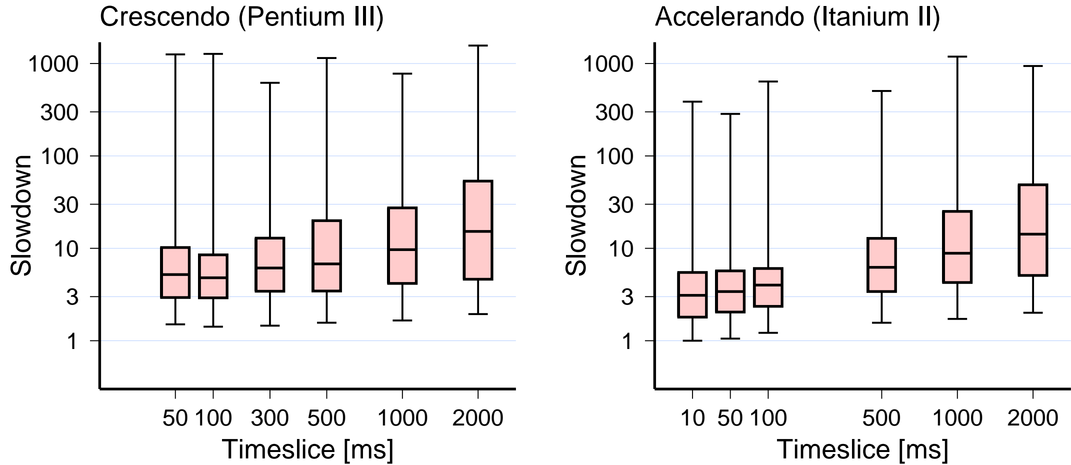


Figure 8: Distribution of slowdown for different time quanta (logscale). Each bar depicts the 5%, 25%, 50%, 75%, and 95% percentiles.

tive to changes in the time quantum than the rest of the jobs.

The median reaches a minimum value at a time quantum of $\approx$ 100 ms and $\approx$ 50 ms on Crescendo and Accelerando respectively. Running with shorter time quantum on Crescendo yields unreliable results, with degraded performance.

Fig. 8 shows the distribution of slowdown for the same time quanta values. The interpretation of this figure is *reversed*, since the 5% mark represents mostly very long jobs (that have a low wait time to run time ratio, and thus a low slowdown value). On the other end, the 95% mark shows the high sensitivity of the slowdown metric to changes in the wait and run times of short jobs. Slowdown also seems to have a minimal median value at $\approx$ 100 ms on Crescendo, and 50 ms (or even 10 ms) on Accelerando. Based on these results, and since we run all our other experiments on Crescendo, we decided to use a time quantum of 100 ms for the other measurements.

5 Effect of Load

5.1 Experiment Description

We now reach the last part of this study, where we investigate the effect of load on different scheduling algorithms. In this section, we try to answer the following questions:

- How well do different algorithms handle increasing load?
- How do different scheduling algorithms handle short or long jobs?
- How does the dynamic workload affect the scheduler's performance?

When using finite workloads, one must be careful to identify when the offered load is actually high enough to saturate the system. Using an infinite workload, the jobs queues would keep on growing on a saturated system, and so will the average response time and slowdown. But when running a finite workload, the queues would only grow until the workload is exhausted, and then the queues would slowly clear since there are no more job arrivals. The metrics we measure for such a workload are therefore meaningless, and we should ignore them for loads that exceed each scheduler's saturation point.

To identify the saturation points, we used graphs like the one shown in Fig. 9. This figure shows jobs in the system over time, i.e. those jobs that arrived and are not yet finished. It is easy to see that the system handles loads of 78% and 83% quite well. However, the burst of activity in the second half of

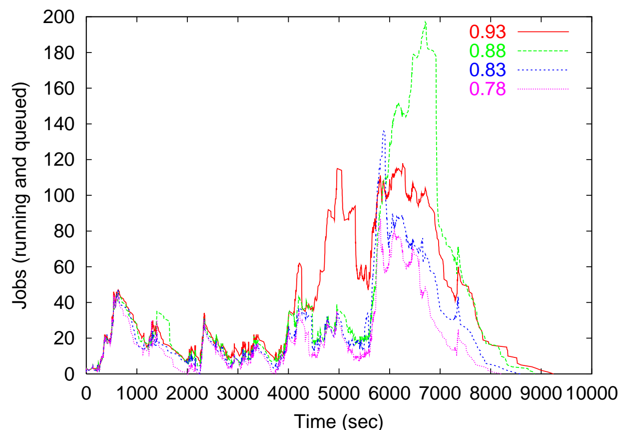


Figure 9: Number of running and queued jobs as a function of time for offered loads of 78%–93%, when using gang scheduling.

the workload seems to cause problems when the load is increased to 88% or 93% of capacity. In particular, it seems that the system does not manage to clear enough jobs before the last arrival burst at about 7300 seconds. This indicates that the load is beyond the saturation point. Using this method, we identified and discarded those loads that saturate each scheduling scheme.

The results for this workload indicate that FCFS seems to saturate at about 78% load, GS and SB at about 83%, and FCS at 88%.

5.2 Results and Discussion

Figures 10(a) and 10(b) show the average response time and slowdown respectively, for different offered loads and scheduling algorithms. The near-linear growth in response times with load is due to our method of varying load, by multiplying run times of jobs by a load factor. Both metrics suggest that FCS seems to perform consistently better than the other algorithms, and FCFS (batch) seems to perform consistently worse than the others. Also, FCFS saturates at a lower load than the other algorithms, while FCS supports a load of up to 88% in our tests.

To understand the source of this differences, let us look at the median response time and slowdown (Figures 10(c) and 10(d) respectively). A low median response time suggests good handling of short jobs, since most of the jobs can be considered relatively short. On the other hand, a low median slowdown indicates preferential handling of long jobs, since the lowest-slowdown jobs are mostly long jobs, that are affected less by wait time than short jobs[2]. FCFS shows a high average slowdown and a low median slowdown. This indicates that while long jobs enjoy lower waiting times (driving the median slowdown lower), short jobs suffer enough to significantly raise the average response time and slowdown.

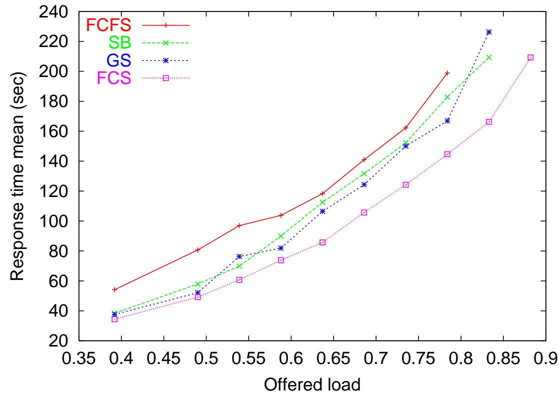
To verify these biases, we look at the CDF of response times for the shorter 500 jobs and longer 500 jobs separately, as defined in Section 3.2 (Figs. 11 and 12). The higher distribution of short jobs with FCS attests to the scheduler's ability to "push" more jobs toward the shorter response times. Similarly, FCFS's preferential treatment of long jobs is reflected in Fig. 12.

We believe the reason for FCS's good performance is its ability to adapt to various scenarios that occur during the execution of the dynamic workload [9]. In particular, FCS always co-schedules a job in its first few seconds of running (unlike SB), and then classifies it according to its communication requirements (unlike GS). If a job is long, and does not synchronize frequently or effectively, FCS will allow other jobs to compete with it for machine resources. Thus, FCS shows a bias toward short jobs, allowing them to clear the system early. Since short jobs dominate the workload, this bias actually reduces the overall system load and allows long jobs to complete earlier than with GS or SB. The opposite can be said of the FCFS scheme, which shows a bias toward long jobs, since they do not have to compete with other jobs.

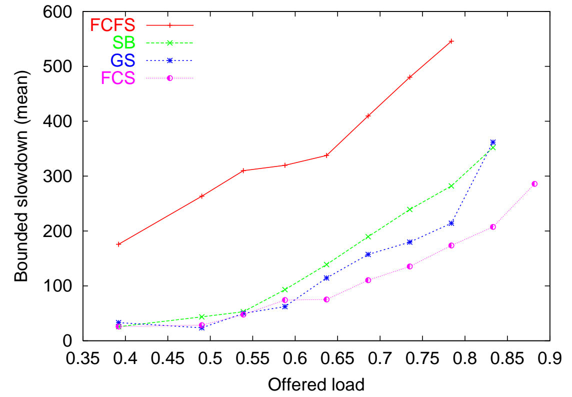
6 Conclusions and Future Work

In this paper we studied the effect of dynamic workloads on several job scheduling algorithms, using a detailed experimental evaluation. Our results confirm some of the previous results obtained with different workloads and in simulated environments. In particular, we identified several scheduling parameters that affect metrics such as response time and slowdown, and quantified their contribution:

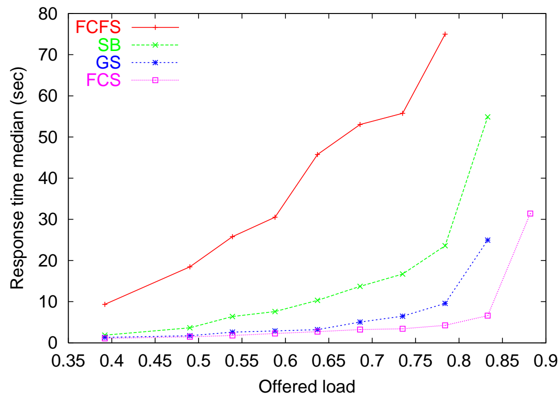
[2] This reversal of meaning is also seen in Figures 7 and 8.



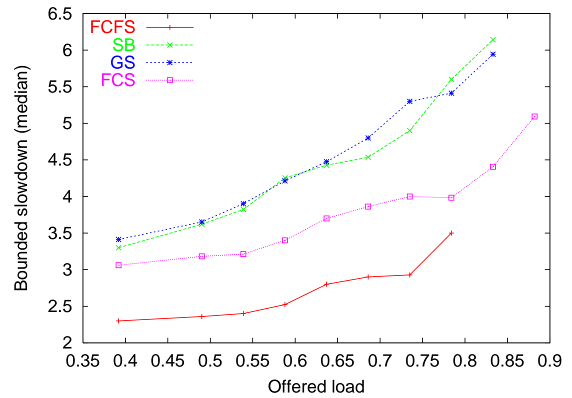
(a) Response time - mean



(b) Bounded slowdown - mean



(c) Response time - median



(d) Bounded slowdown - median

Figure 10: Response time and bounded slowdown as a function of offered load

- Multiprogramming (coscheduling): coscheduling allows shorter waiting times, in particular for short jobs.
- The multiprogramming level: increasing it enables better packing and handling of queued jobs.
- Backfilling: Using an EASY backfilling strategy to handle queued jobs improves jobs packing over time, and shortens their wait time. This it true both for batch scheduling, and for coscheduling schemes.
- The scheduling algorithm itself. We found that FCS is consistently better than the other algorithms for the measured metrics. On the other hand, batch scheduling is consistently worse than the other algorithms.

The bottom line is that using preemption (e.g. gang scheduling) in conjunction with backfilling leads to significant performance improvements, and at the same time the use of backfilling allows the use of a very limited multiprogramming level. To further improve this combination, FCS should be used instead of strict GS. The increased flexibility of FCS allows better utilization and faster flow of jobs through the system, leading to lower response time and slowdown results. To further improve these metrics, we intend to experiment with additional mechanisms such as explicit prioritization of small and/or short jobs as part of the queue management.

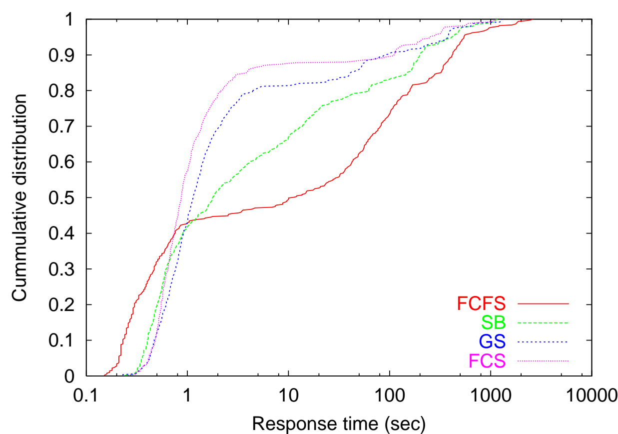


Figure 11: CDF of response times at offered load 74% — 500 shortest jobs.

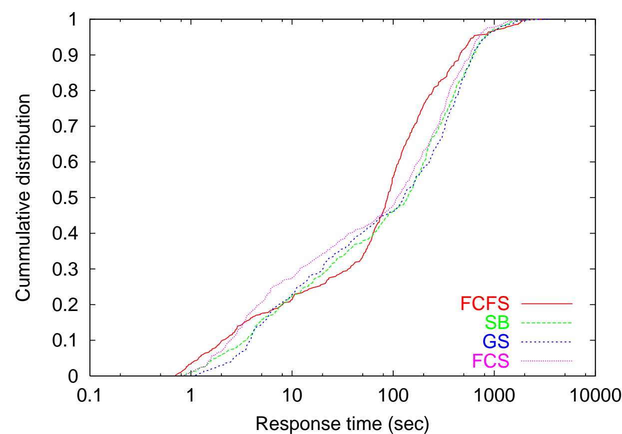


Figure 12: CDF of response times at offered load 74% — 500 longest jobs.

Acknowledgments

We wish to thank Dan Tsafrir for fruitful discussions and useful insights.

References

[1] Andrea Carol Arpaci-Dusseau. Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed Systems. *ACM Transactions on Computer Systems*, 19(3):283–331, Aug 2001.

[2] Anat Batat and Dror G. Feitelson. Gang Scheduling with Memory Considerations. In *International Parallel and Distributed Processing Symposium*, number 14, pages 109–114, May 2000.

[3] Yoav Etsion, Dan Tsafrir, and Dror G. Feitelson. Effects of Clock Resolution on the Scheduling of Interactive and Soft Real-Time Processes. In *SIGMETRICS Conf. Measurement and Modeling of Comput. Syst.*, Jun 2003. (to appear).

[4] Dror G. Feitelson. A Survey of Scheduling in Multiprogrammed Parallel Systems. Research Report RC 19790 (87657), IBM T. J. Watson Research Center, Oct 1994.

[5] Dror G. Feitelson. The Forgotten Factor: Facts; on Performance Evaluation and Its Dependence on Workloads. In Burkhard Monien and Rainer Feldmann, editors, *Euro-Par 2002 Parallel Processing*, pages 49–60. Springer-Verlag, Aug 2002. Lect. Notes Comput. Sci. vol. 2400.

[6] Dror G. Feitelson and Larry Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, Dec 1992.

[7] Dror G. Feitelson and Larry Rudolph. Metrics and Benchmarking for Parallel Job Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–24. Springer-Verlag, 1998. Lect. Notes Comput. Sci. vol. 1459.

[8] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and Practice in Parallel Job Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lect. Notes Comput. Sci.*, pages 1–34. Springer Verlag, 1997.

[9] Eitan Frachtenberg, Dror G. Feitelson, Fabrizio Petrini, and Juan Fernandez. Flexible CoScheduling: Mitigating load imbalance and improving utilization of heterogeneous resources. In *International Parallel and Distributed Processing Symposium*, number 17, April 2003.

[10] Eitan Frachtenberg, Fabrizio Petrini, Juan Fernandez, Scott Pakin, and Salvador Coll. STORM: Lightning-Fast Resource Management. In *Supercomputing 2002*, Baltimore, MD, November 2002.

[11] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. In *SIGMETRICS Conf. Measurement and Modeling of Comput. Syst.*, pages 120–132, May 1991.

[12] David Lifka. The ANL/IBM SP Scheduling System. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 295–303. Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[13] Uri Lublin and Dror G. Feitelson. The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs. *Journal of Parallel and Distributed Computing*, 2003. (to appear).

[14] Jose E. Moreira, Waiman Chan, Liana L. Fong, Hubertus Franke, and Morris A. Jette. An Infrastructure for Efficient Parallel Job Execution in Terascale Computing Environments. In *Supercomputing'98*, Nov 1998.

[15] Ahuva W. Mu'alem and Dror 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, Jun 2001.

[16] John K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *3rd Intl. Conf. Distributed Comput. Syst. (ICDCS)*, pages 22–30, Oct 1982.

[17] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics Network: High Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, January-February 2002.

[18] Quadrics Supercomputers World Ltd. *Elan Reference Manual*, January 1999.

[19] Quadrics Supercomputers World Ltd. *Elan Programming Manual*, May 2002.

[20] David Talby, Dror G. Feitelson, and Adi Raveh. Comparing Logs and Models of Parallel Workloads Using the Co-Plot Method. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 43–66. Springer Verlag, 1999. Lect. Notes Comput. Sci. vol. 1659.

[21] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, Aug 1990.

[22] Y. Zhang, H. Franke, J. E. Moreira, and A. Sivasubramaniam. Improving Parallel Job Scheduling by Combining Gang Scheduling and Backfilling Techniques. In *Intl. Parallel & Distributed Processing Symp.*, number 14, pages 133–142, May 2000.