Performance Evaluation of Gang Scheduling for Parallel and Distributed Multiprogramming

Fang Wang[1] Marios Papaefthymiou[2] Mark Squillante[3]

[1] Yale University, New Haven CT 06520, USA
[2] University of Michigan, Ann Arbor MI 48109, USA
[3] IBM Research Division, Yorktown Heights NY 10598, USA

Abstract. In this paper we explore the performance of various aspects of gang scheduling designs. We developed an event-driven simulator of a vanilla gang scheduler that relies on the Distributed Hierarchical Control (DHC) structure. We also developed three variations of the vanilla gang scheduler that rely on a push-down heuristic and on two job-migration schemes to decrease response times by reducing processor idle time. We evaluated the gang schedulers on a compiled, one-month long history of jobs from the Cornell Theory Center that was scheduled by EASY-LL, a particular version of LoadLeveler with backfilling. Our results demonstrate the significant performance improvements that can be achieved with gang scheduling. They also show the performance impact of various aspects in the design of gang schedulers. We identify and discuss the potential benefits of several approaches for addressing a number of gang scheduling issues that, under certain workload conditions, become important in practice. Our techniques include heuristics for mapping jobs to processors and for choosing time quanta, block paging for reducing memory overheads, and the allocation of multiple time-slices to smaller jobs per timeplexing cycle.

1 Introduction

Resource management schemes have become essential for the effective utilization of high-performance parallel and distributed systems that are shared among multiple users. The main objective of resource schedulers is to achieve high overall system throughput, while at the same time providing some guarantee for the performance of individual jobs in the system. This is a particularly challenging task, given that typical workloads of multiprogrammed multicomputers include a large number of jobs with diverse resource and performance requirements.

The two basic mechanisms used in multicomputer schedulers are time-sharing and space-sharing. Time-sharing ensures that no job monopolizes the system's resources and can be suitable for scheduling jobs with relatively small processing requirements. A job may not require the attention of the entire system, however. Moreover, abundant empirical evidence indicates that program dependencies and communication costs may limit the degree of achievable parallelism (e.g., [2, 16]). In these situations, space-sharing can increase throughput by partitioning resources and reducing the underutilization of system partitions.

Gang scheduling is a flexible scheduling scheme that combines time-sharing and space-sharing with the goal of providing the advantages of both approaches, including high system throughput and low response times for short-running jobs. The roots of gang scheduling can be traced back to the coscheduling concept described in [18]. This two-dimensional division (in time and space) of resources among jobs can be easily viewed as having the resource allocations governed by a scheduling matrix, where each column represents a specific processor and each row represents a particular time-slice, or quantum. Each non-empty matrix entry (i,j) contains a job (or set of jobs), which represents the allocation of the j^{th} processor to this job during the i^{th} quantum. The set of entries containing the same job (or set of jobs) on a given row is called a partition. The number of partitions and the size of each partition can vary both within and across rows. When a job is submitted, it is assigned to a partition on a particular row. Each partition is allocated a specific time quantum associated with its row, with the possibility of having the partitions on a given row use different quantum lengths. When the time quantum for a partition expires, the resources are reallocated and the job(s) of the partition(s) on the next row are scheduled to execute on the system. Within each partition, resources may be dedicated or time-shared. Thus, gang scheduling supports time-sharing at the partition level and at the individual job level.

Gang scheduling encompasses a very broad range of schedulers depending on the particular schemes used for partitioning resources and for sharing resources within each partition. One particular approach is based on the distributed hierarchical control structure [4, 5, 6]. Within the context of the above description, this scheme can be logically viewed as having a scheduling matrix with $\log P + 1$ rows, where the i^{th} row contains 2^i partitions each of size $P/2^i$, $0 \leq i \leq \log P$, and P denotes the number of system processors. A somewhat different approach, which can be conceptually viewed as a generalization of Ousterhout's original global scheduling matrix, has also been considered [13, 14].

Due to its promising characteristics, gang scheduling has attracted considerable attention in recent years. Gang schedulers based on the distributed hierarchical control structure [4, 5, 6] have been implemented for the IBM RS/6000 SP2 [8, 30] and for clusters of workstations [9, 29]. Similarly, another form of gang scheduling has been implemented on both the IBM SP2 and a cluster of workstations [13, 14]. The performance of gang scheduling schemes that use distributed hierarchical control has been analyzed from a queueing-theoretic perspective [21, 22]. Moreover, the performance of several gang scheduling algorithms has been studied by simulation on synthetically generated workloads [3, 7].

In this paper we present an empirical evaluation of various gang scheduling policies and design alternatives based on an actual parallel workload. Our focus is on the distributed hierarchical control approach to gang scheduling, although many of the principles and trends observed in this study are relevant to other forms of gang scheduling. Our study includes an examination of a vanilla gang scheduling scheme [4, 21] and two variations of this scheme that use *push-down* and *job-migration* heuristics to increase system throughput and decrease response times by minimizing idle partitions. These scheduling strategies are simulated under a workload that we obtained by post-processing a trace of the workload characteristics for one month at the Cornell Theory Center [10, 11, 12]. The original workload was scheduled on 320 processors of the IBM SP2 at Cornell's Theory Center using EASY-LL, an enhanced version of the basic LoadLeveler scheduler that uses backfilling to reduce the response times of jobs with small resource requirements [20].

The objectives of our evaluation study were to assess the effectiveness of different aspects of gang scheduling designs under a variety of heuristics for assigning jobs to processors and across a range of memory overheads. We investigated a greedy scheme for the vanilla and the push-down scheduler and two priority-based policies for migrating and redistributing jobs. In our experiments, both job-migration policies perform better than the vanilla and the push-down schemes. Our first job-migration scheme favors jobs with small resource requirements and achieves significantly shorter response times than EASY-LL for most job classes in the system. Our other job-migration policy favors large jobs and performs better than either EASY-LL or any of our gang scheduling schemes in most job classes with large resource requirements. For jobs with small resource requirements, however, EASY-LL outperforms this particular gang scheduler. As context-switch costs increase due to factors such as memory and communications overheads, the performance of the three gang scheduling policies degrades significantly, especially for short quanta. We propose an approach to effectively reduce the performance impact of such memory overheads as part of our study.

The remainder of this paper is organized as follows. We begin in Section 2 with a more detailed description of EASY-LL and the gang scheduling policies considered. In Section 3 we present the mechanisms examined for mapping jobs to processors. A brief overview of our simulator engine is given in Section 4. We describe the workload used in our study in Section 5, and continue with the presentation of our experimental results in Section 6. We then discuss in Section 7 some of the practical implications of our results, as well as approaches to improve gang scheduling performance and current ongoing aspects of our study. Our concluding remarks are presented in Section 8.

2 Scheduling policies

In this section we describe the scheduling policies examined in our study. We first present the notation and terminology used throughout this paper. We then give a brief overview of the EASY-LL scheduling scheme, followed by a description of the gang scheduling policies we considered.

2.1 Preliminaries

The basic parameters associated with serving any given job j is the *arrival time* α_j, the *dispatch time* β_j, and the *completion time* ϵ_j of the job. When this job is submitted to the system at time α_j, it is placed into a particular queue based on the scheduling policy. At time β_j, the job is moved from this queue to a specific partition and receives service for the first time. At time ϵ_j the job finishes its execution and exits the system. The (cumulative) amount of time for which job j actually receives service is its *service time* S_j.

A number of important performance measures are used to compare the different scheduling policies considered in our study. In particular, the job parameters $\alpha_j, \beta_j, \epsilon_j$ and S_j can be used to define the following performance metrics for the execution of job j:

– *response time* $\mathcal{R}_j$, where $\mathcal{R}_j = \epsilon_j - \alpha_j$
– *queueing time* $\mathcal{Q}_j$, where $\mathcal{Q}_j = \beta_j - \alpha_j$
– *execution time* $\mathcal{E}_j$, where $\mathcal{E}_j = \epsilon_j - \beta_j$
– *sharing time* $\mathcal{H}_j$, where $\mathcal{H}_j = \mathcal{E}_j - S_j$
– *waiting time* $\mathcal{W}_j$, where $\mathcal{W}_j = \mathcal{R}_j - S_j$

In these definitions, we have split "waiting time" (respectively, "service time") into two separate components $\mathcal{W}_j$ and $\mathcal{Q}_j$ (respectively, $\mathcal{E}_j$ and $\mathcal{H}_j$) to take the time sharing into account. Thus, the total waiting time $\mathcal{W}_j$ is the sum $\mathcal{Q}_j + \mathcal{H}_j$ of the time spent waiting on the queue ($\mathcal{Q}_j$) and the time that job j is swapped out ($\mathcal{H}_j$). Also, the execution time $\mathcal{E}_j$ is the sum $S_j + \mathcal{H}_j$ of the service time (S_j) and the sharing time during which job j is swapped out ($\mathcal{H}_j$).

2.2 EASY-LL

Our consideration here of the LoadLeveler and EASY-LL schedulers is based upon the use of the versions of these schedulers at the Cornell Theory Center when the workload traces used in our experiments were collected. This version of LoadLeveler schedules jobs in the order of their arrival times. The job at the head of the queue is dispatched and begins its execution as soon as sufficient resources become available in the system. LoadLeveler does not support preemption. Once a job begins its execution, it continues until it terminates. Thus, in LoadLeveler we have $\mathcal{W}_j = \mathcal{Q}_j$ and $\mathcal{E}_j = S_j$. This LoadLeveler scheme may not be suitable for interactive execution, as system access is blocked for every job that arrives immediately after any single job with large resource requirements.

The EASY-LL scheduler is a variation of this version of the LoadLeveler scheme that uses a backfilling heuristic to improve response time for short-running tasks. When submitting their jobs to the system queue, users request a specific number of processors and provide an estimate of the execution times of their jobs on the requested resources. Whenever new resources become available or a new job is submitted, EASY-LL schedules the first job in the queue that fits within the available resources and whose execution does not delay the dispatching of any job ahead of it in the queue. Thus, small jobs can bypass larger jobs, provided they do not delay the execution of the larger jobs.

2.3 Vanilla gang scheduling

Generic gang scheduling under distributed hierarchical control views a parallel computing system as a collection of P identical processors and a hierarchy of

$L = \log P + 1$ different *classes* of jobs. At any time, the system is serving jobs from a specific class. When serving jobs of class i, the system is divided into $P/2^i$ *partitions* where each partition consists of 2^i processors. For example, a 256-processor system has 9 classes, where class 0 has 256 partitions each with one processor, class 1 has 128 partitions each with two processors, and so on. A first-come first-served queue is associated with each class from which the corresponding partitions select jobs for execution. An example of this hierarchical, binary-tree view of the gang scheduling system is illustrated in Fig. 1. Jobs are allocated to partitions in their corresponding classes according to a specific job assignment policy (described in Section 3).

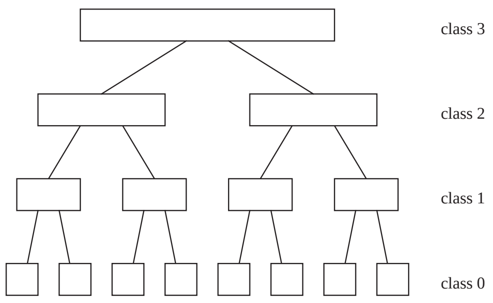

Fig. 1. Binary-tree view of an 8-processor multicomputer system under the distributed hierarchical control structure.

During the operation of the system, each class i is allocated a time-slice of certain length. Processors are dedicated to each of the L classes in a time-shared manner by rotating the time allocated to the job classes. The time interval between successive time-slices of the same class is called the *timeplexing cycle* of the system, which we denote by T. A system-wide switch from the current class i to the next class $i - 1$ (modulo L) occurs when at least one of the following two events becomes true:

- The time-slice of class i has expired.
- There are no jobs of class i in the system.

In the vanilla gang scheduling scheme, when the number of partitions in a class exceeds the number of jobs assigned to it, the excess partitions remain idle during the time-slice. Therefore, the system may be significantly underutilized in light load situations.

2.4 Gang scheduling with push-down

A simple variation of the vanilla gang scheduling scheme uses a push-down heuristic to reduce the number of idle partitions at any time. In push-down gang scheduling, every partition that is idle during its designated time-slice is reconfigured into two partitions for the class below it, each of which is half the size of the original partition. The partitions are reconfigured recursively until they find a job(s) from a class below that is allocated to the same set of processors. Thus, a fraction of the original partition's time-slice is allocated to jobs belonging to the class(es) below (i.e., it is "pushed down" to the lower partitions), and at any time the system may be serving jobs from more than one class. With push-down, the actual length of the timeplexing cycle is workload dependent. Assuming that no job finishes before the expiration of its quantum, the actual timeplexing cycle is equal to the maximum number of busy nodes in any path from the root to the leaves of the distributed hierarchical control tree.

2.5 Gang scheduling with job migration

The third gang scheduling policy we consider borrows characteristics from both the vanilla and the push-down schedulers. Under this policy, all jobs waiting in the queue of a class are scheduled to execute in the beginning of the corresponding time-slice. As is the case with push-down gang scheduling, idle partitions are assigned jobs from other classes according to some job assignment priority scheme. These jobs are free to execute on *any* idle partition, however. Therefore, gang scheduling with migration is more flexible and may result in fewer idle partitions than gang scheduling with push-down. In a manner similar to the vanilla gang scheduler, switches are system-wide and occur when the time-slice of the class expires. Thus, even though the system may not be serving any job from the class that corresponds to the active time-slice, jobs are reallocated in logical synchrony, and the timeplexing cycle remains fixed.

The overhead of such a migration scheme can be quite large. In our current study, we ignore this overhead and thus use the job-migration policy in our experiments to explore the upper bound on gang scheduling performance under distributed hierarchical control. A few methods for attempting to achieve the performance of this migration scheme in practice, such as *tree-packing*, are discussed in Section 7.

2.6 Processor counts that are not powers of 2

It is straightforward to embed the scheduling schemes described in the previous subsections in systems with 2^i processors. When processor counts are not powers of 2, however, there are several ways to implement the distributed hierarchical control structure. Fig. 2 illustrates the general approach that we adopted in our scheduling policies. For one time-slice during each timeplexing cycle, system resources are configured as a single 320-processor system. Subsequently, the 320 processors are partitioned into a 64-processor and a 256-processor system, each of which is viewed as a binary tree. With the push-down scheduling policy, the two trees proceed independently. With the job-migration policies, however, the switches in the two trees are synchronized. Due to their different heights, the two trees may be serving different classes at a time. In our implementations, when the 256-processor subtree is serving class 8 or 7, the 64-processor subtree is serving class 1 or 0, respectively. From class 6 and below, both trees serve the same class during each time-slice.

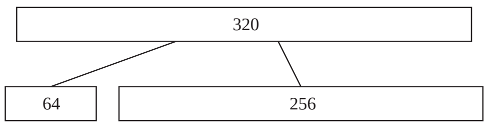

Fig. 2. Distributed hierarchical control structure for a 320-processor system.

3 Job assignment policies

An important component of every gang scheduling approach is the policy that it uses to assign dispatched jobs to partitions. This policy can have a profound impact on the performance of a gang scheduler, as we have found in our numerous experiments, and these performance issues are highly dependent upon the workload characteristics. So far, we have experimented with three policies for assigning jobs within each class. The first two policies are used together with the vanilla and the push-down scheduler. Under these policies, each job is assigned to a partition when it starts to execute for the first time and remains assigned to the same partition for the duration of its execution time. The third policy is used together with the migration scheduler and assigns jobs to partitions on every switch via different priority schemes.

Our first job assignment policy is a greedy, first-fit strategy that always starts from the leftmost branch of the tree and assigns each new job to the first available partition in that class. In a lightly loaded system, this scheme will load the left branch of the tree, while leaving the partitions on the right branch idle. Under these circumstances, the degree to which push-down can be applied on the right branch is limited, and thus it can become ineffective for the right branch.

The second policy we investigate is a very simple, "weight-oriented" allocation scheme. Every node in the tree is associated with a weight equal to the sum of the jobs allocated to that node and all its descendents in the tree. Node weights are updated whenever a job finishes or is dispatched to a partition in the tree. When assigning a job to a partition in class i, we select the one with the lightest weight node in the level i of the tree. Such a partition can be found by a straightforward tree traversal that recursively looks for the lightest branch at each level until it reaches level i. The lightest partition in the class has the smallest number of jobs allocated in level i and below. This scheme is a local optimization procedure that does not take into account the remaining service times of the currently dispatched jobs or future job arrivals. Moreover, the details of its definition are meant to work in unison with the push-down scheduler. Under a different approach, such as a push-up scheme, the weights and traversals should be modified to match the corresponding properties of the approach employed.

For gang schedulers that support job migration, we experimented with a simple job assignment policy that maps jobs to idle partitions in a greedy manner. During each time-slice, jobs waiting to execute in the queue of the corresponding class are dispatched. If processors remain idle, they are assigned jobs from other classes. Each time a job is dispatched, it may run on a different set of processors. Since it is assumed that there is no overhead for migrating jobs to different partitions (see Section 2.5), there is no loss of generality while obtaining the best gang scheduling performance. A straightforward job assignment strategy would be to impose a priority on the system's classes and to assign idle processors to jobs of other classes by considering these classes in decreasing priority. In our experiments, we investigated two extremes of this simple priority scheme. We looked at a top-down order that starts from the root and proceeds to the leaves, thus favoring jobs with large resource requirements. Conversely, we also studied a bottom-up approach that traverses classes from the leaves to the root, thus favoring jobs with small resource requirements.

4 Simulator

We developed an event-driven simulator engine to experiment with the various gang scheduling policies described above. Our simulator has four different events: job arrival, job completion, time-slice expiration, and context switch. All events are inserted into an event queue, and the earliest event in the queue is triggered first. In this section we outline the operation of our simulation and explain the design choices we made in order to simplify its implementation.

In general, when a class i job arrives, if class i is currently being served and has partitions available, the job is dispatched to an available partition according to one of the schemes described in Section 3. If no partition in class i is available, the job is inserted into the queue for class i.

If class i is currently not being served, the job is handled according to the specifics of the gang scheduling policy under consideration. In the vanilla policy, the job is simply inserted into the waiting queue for class i. With the push-down and job-migration policies, if there are available partitions in another class j that can have their class j time-slice pushed down to host the newly arrived class i job, the new job will be dispatched accordingly. If there is no available partition or no time can be pushed down, the job will be added to the class i queue.

When a job completes, its partition becomes available, and the weight associated with each node is updated. If there are jobs waiting in the queue of the current class, the available partition may be assigned to a new job according to one of the mechanisms described in Section 3. Otherwise, under the push-down and migration policies, the time-slice of the available partition is allocated to a job of another class in a manner similar to that described above for an arrival that finds the system serving another class while processors are idle.

When the time-slice of a class expires, every job that is currently executing is stopped and its remaining execution time is updated. Subsequently, a context switch event occurs. A context switch also occurs when all the partitions in a class become idle, and there are no jobs waiting in the queue. Thus, the time remaining in the quantum is not wasted idling.

The context switch event starts the execution of the jobs in the next class that are ready to run. Preempted jobs resume their execution, and jobs in the queue are dispatched if there are still partitions available. With push-down scheduling, if any partitions remain idle, their time-slices are pushed down for jobs in lower classes. In a similar manner, with the job-migration policy, idle partitions will be assigned jobs from the queues of higher or lower classes in the system.

In order to avoid a complex software implementation, we made a few simplifying assumptions in the simulation of the migration scheduler. First, our simulator dispatches new jobs only in the beginning of each time-slice. Thus, whenever jobs arrive or finish in the middle of a time-slice, our simulator does not take advantage of the possibly idle processors in the system. Second, our simulator does not account for the various overheads (e.g., communication, data transfer, system state, etc.) incurred for migrating jobs among processors. In view of the relatively light loads we experimented with, dispatching jobs only in the beginning of time-slices should not significantly change the trends of our performance results. Moreover, our second assumption yields an upper bound on the performance of gang scheduling under distributed hierarchical control, and we wanted to quantify this potential benefit under an actual workload. This and other related issues, including several mechanisms that can be used to reduce the cost of data transfers, are discussed further in Section 7.

5 Workload characteristics

We experimented with a collection of jobs that were submitted to the Cornell Theory Center SP2 during the month of August 1996. The execution of these jobs on the SP2 was managed by the EASY-LL scheduler described in Section 2.2. Our workload comprised 6,049 jobs that requested up to 320 processors and had nonzero CPU times. We categorized these jobs into ten classes according to their resource requirements. Each job requesting p processors was assigned to class i, where $2^{i-1} < p \leq 2^i$. This classification scheme facilitated the direct performance comparison of the gang scheduling policies with the EASY-LL scheduler on a 320-processor system.

The statistics of this workload are given in the table of Fig. 3. The first two columns of the table give the class number and the job counts in each class. The third column gives the average service time for the jobs in each class. This number is intrinsic to each job and the number of processors it executes on. The fourth and fifth columns give the average waiting and response time that was achieved for each class using the EASY-LL scheduler. The last column gives the normalized average response time for the jobs in each class. An interesting point about these data is that, with the notable exception of the uniprocessor class 0, the average response times increase almost monotonically with the number of processors in the class. Also, with the exception of class 8, the normalized response times increase monotonically with the number of processors.

class	jobs	S_j	$\mathcal{W}_j$	$\mathcal{R}_j$	$\mathcal{R}/S_j$
0	2,408	16,854	9,105	25,959	1.5
1	515	18,100	12,122	30,222	1.7
2	669	9,571	10,315	19,885	2.1
3	582	3,563	13,946	17,510	4.9
4	848	8,822	24,947	33,768	3.8
5	465	9,008	69,157	78,165	8.7
6	420	93,189	93,040	102,359	11.0
7	74	12,168	128,895	141,062	11.6
8	45	300	146,141	146,441	488.4
9	23	5,912	147,240	153,153	26.0

Fig. 3. Job counts, service times, waiting times, response times, and normalized response times for the one-month workload from the Cornell Theory Center under EASY-LL. Service, waiting, and response times are measured in seconds.

6 Simulation results

Initially, we experimented with the vanilla gang scheduling policy. It soon became evident that this scheme was performing consistently and significantly worse than EASY-LL under the Cornell Theory Center workload. Since the vanilla scheduler switches all partitions in the same class synchronously, this relatively light parallel workload resulted in several idle partitions. We thus turned our attention to the other gang schedulers, due to their promising handling of idle partitions. In this section we discuss the results we obtained by simulating the push-down scheduler and our two job-migration policies on the one-month workload from the Cornell Theory Center. For the push-down policy, we only present the results we obtained with the weight-oriented job assignment policy, since it consistently outperformed the first-fit policy, as expected.

Figs. 4, 5, 6, and 7 give the performance of the gang scheduling schemes for context-switch costs of 1 second and 16 seconds. In each of these graphs, the y-axis gives the mean response time for the jobs in the corresponding class normalized by the mean service time for that class. The timeplexing cycle, which is given on the x-axis, is divided uniformly among the system's ten classes. We arrived at the worst-case context-switch cost of 16 seconds by assuming that the jobs have a 64MB working set on each processor [11] which must be loaded in its entirety at the rate of 1 page/millisecond for a page size of 4KB, given the characteristics of many parallel scientific applications [19] and the (potentially) large degree of multiprogramming with ten classes. Note that EASY-LL is not affected by the context-switch overheads of gang scheduling, and the corresponding curves in Figs. 4 – 7 represent the last column in the table of Fig. 3.

Our results show that the scheduling policy with migration from the leaves achieves shorter response times than EASY-LL for eight out of the ten classes. Recall that this migration policy favors jobs with smaller resource requirements. This results in better performance for the small job classes than that provided

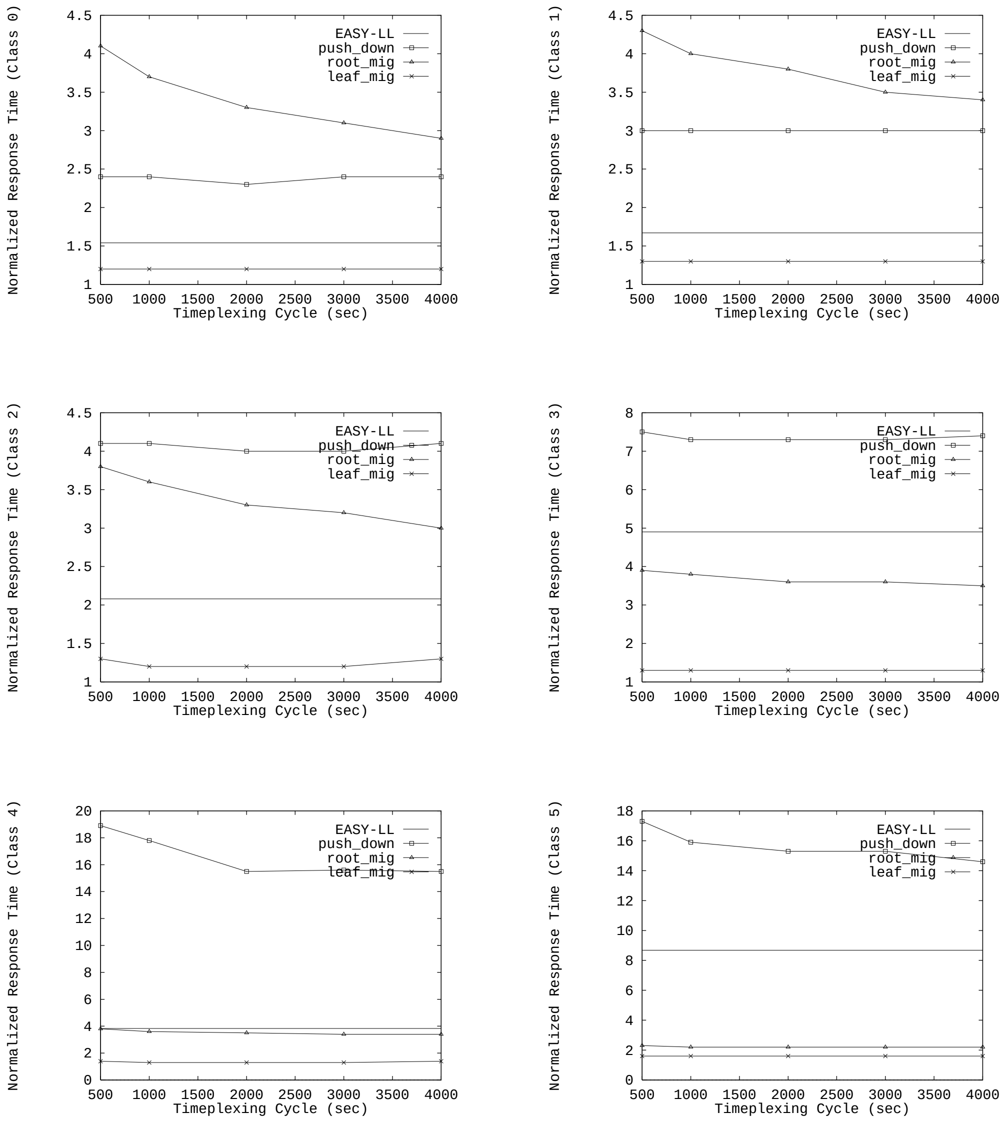

Fig. 4. Normalized response times of classes 0–5 for the push-down gang scheduler, the migration scheduler with jobs assigned from the root, and the migration scheduler with jobs assigned from the leaves. Quanta are allocated uniformly, and the context-switch cost for each class is 1 sec.

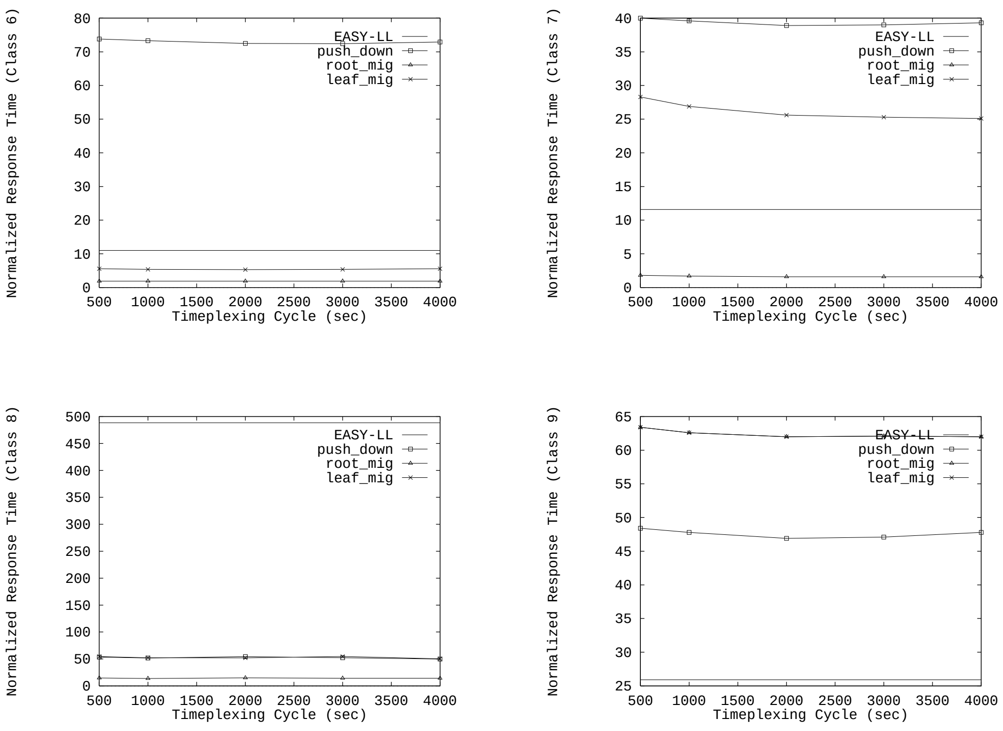

Fig. 5. Normalized response times of classes 6–9 for the push-down gang scheduler, the migration scheduler with jobs assigned from the root, and the migration scheduler with jobs assigned from the leaves. Quanta are allocated uniformly, and the context-switch cost for each class is 1 sec.

under EASY-LL, which also attempts to improve performance for jobs with smaller resource requirements via backfilling. In some cases, these performance improvements are quite significant, with a reduction in the normalized response times by factors that typically range between 2 and 4. Some of the larger job classes also receive improved performance under migration from the leaves in comparison to EASY-LL; e.g., the normalized response time of class 8 decreases by almost a factor of 10. However, the favoring of smaller job classes under migration from the leaves degrades the performance of classes 7 and 9 relative to EASY-LL, where our results show a decrease in normalized response time by about a factor of 3.

The job-migration policy that gives higher priority to the jobs closer to the root of the control tree outperforms EASY-LL for classes 5 through 8. Moreover, for sufficiently large timplexing cycles, this policy performs as well as EASY-LL for classes 3 and 4. In comparison with the migration policy that favors jobs at the leaves, this policy breaks even for class 5 and performs better for classes 6 and above. As expected, both job-migration policies achieve the same performance

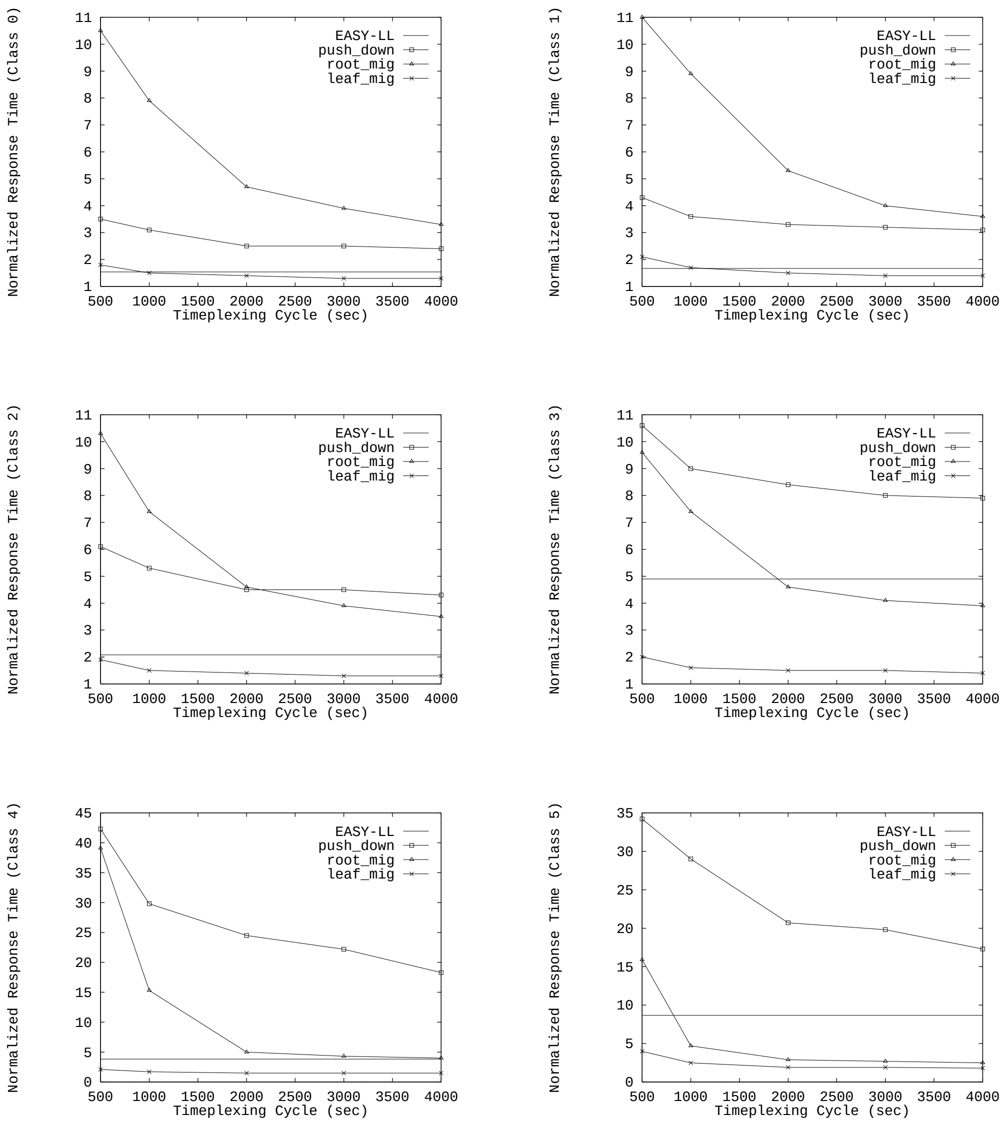

Fig. 6. Normalized response times of classes 0–5 for the push-down policy, the job-migration policy from the root, and the job-migration policy from the leaves. Quanta are allocated uniformly, and a worst-case context-switch cost of 16 sec is assumed for each class.

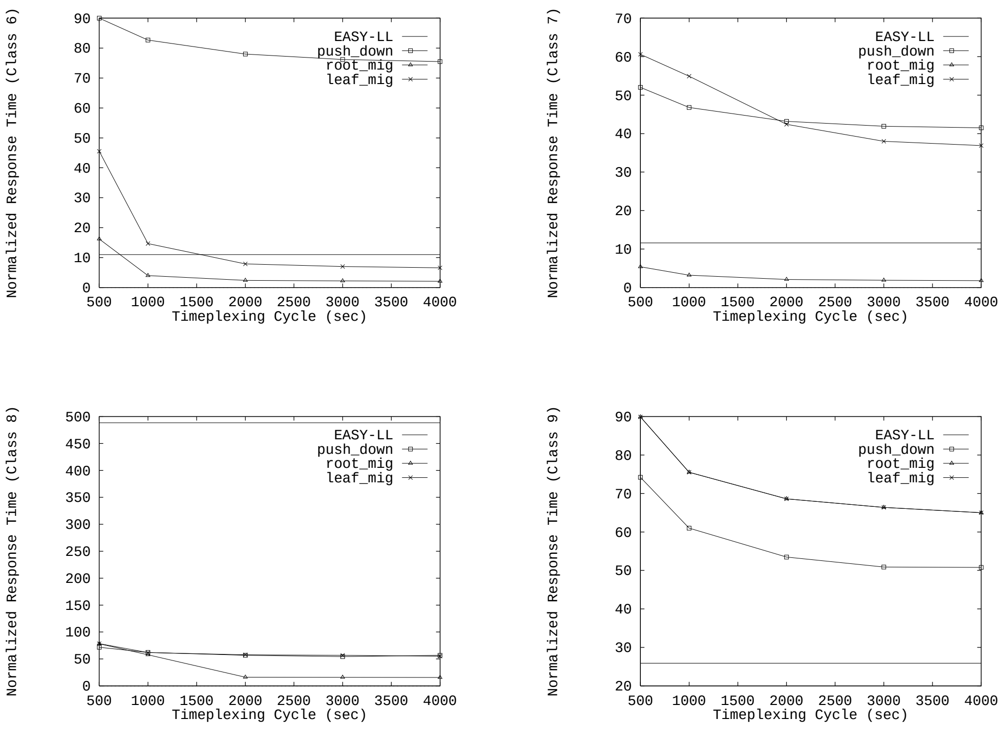

Fig. 7. Normalized response times of classes 6–9 for the push-down policy, the job-migration policy from the root, and the job-migration policy from the leaves. Quanta are allocated uniformly, and a worst-case context-switch cost of 16 sec is assumed for each class.

on class 9, since the timplexing cycle is fixed and the jobs in that class cannot fit in any smaller partition of the system. The gains achieved for the large classes by migrating jobs from the root come at a sometimes significant performance hit for the smaller classes, however. For classes 0 through 2, migration from the root performs worse than all the scheduling policies we considered.

As these results show, there is an important tradeoff with respect to the priority order used to fill otherwise idle slots in the control tree (via migration or other methods). This priority order should be reflected in the job assignment and other aspects of the gang scheduling policies. Moreover, performance benefits may be realized in various cases by exploiting adaptive (dynamic) schemes that adjust the priority order based upon the state of the system (as well as changes to this state). As a simple example, an adaptive migration scheme based on the extremes considered in this paper could consist of determining whether migration is performed from the root or the leaves on each timeplexing cycle based on the state of the system at the start of the cycle.

The push-down policy almost always performs worse than EASY-LL and the two migration policies. The disappointing performance of this scheme is primarily due to processors left idle during certain time-slices when it is not possible to push these processors down to jobs in classes below. While the simple job assignment policy attempts to balance the allocation of jobs to the control tree upon arrival, it does not take job service times into account and therefore the tree can become unbalanced on departures. This characteristic coupled with the relatively light workload resulted in idling partitions while jobs were waiting in the queues of busy servers.

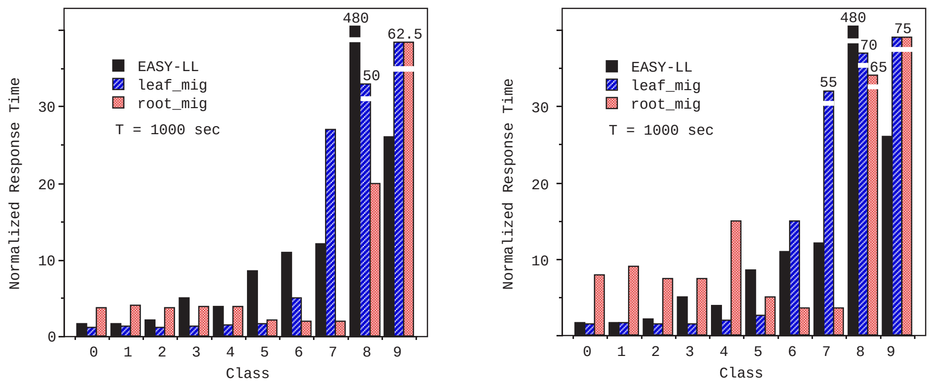

Fig. 8. Normalized response times for EASY-LL and the two migration policies with a timeplexing cycle equal to 1,000 sec. The context-switch costs are 1 sec for the left chart and 16 sec for the right chart.

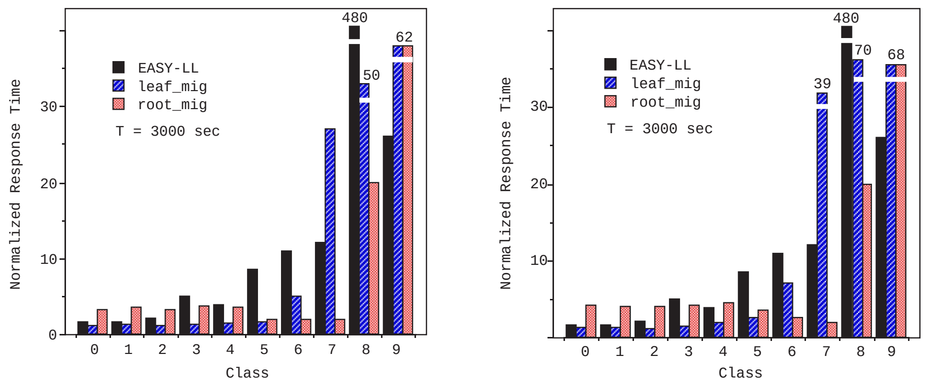

Fig. 9. Normalized response times for EASY-LL and the two migration policies with a timeplexing cycle of 3,000 sec. The context-switch costs are 1 sec for the left chart and 16 sec for the right chart.

Our simulations also show the performance degradation of the gang scheduling policies when the context-switch costs increase. For both values of context-switch overheads that we tried, however, the qualitative trends of our results are similar. The bar charts in Figs. 8 and 9 illustrate the effects of context-switch overheads on the performance of the job-migration schedulers. For low switch costs, the two job migration schemes underperform EASY-LL in several classes. When switch costs increase, however, the performance of both policies degrades, and the number of classes in which they outperform EASY-LL decreases. Performance degradation is more evident for short timeplexing cycles, since in this case context-switch costs become a significant fraction of each time-slice. We address in Section 7.1 ways to effectively reduce some of the dominant causes of this overhead.

7 Discussion

Our simulation results based on the Cornell Theory Center workload data have several important implications on gang scheduling strategies in practice. In this section, we discuss various aspects of these practical implications, as well as current ongoing aspects of our study.

7.1 Memory and paging overheads

Based on the simple analysis in Section 6 to estimate the context-switch memory overhead, as well as the corresponding response time results, memory and paging overheads can have a considerable impact on the performance of large-scale parallel systems that time-share their resources. We now discuss a particular strategy to significantly reduce and effectively eliminate these overheads for a general class of large-scale parallel applications. Our approach is based in part on the concept of *block paging*, which was introduced in the VM/SP HPO operating system [1, 25, 26] and extended in the VM/ESA operating system [23, 24]. A few other systems have since adopted some of these concepts, and related forms of prefetching have recently appeared in the research literature [15, 17]. We first provide a brief overview of the block paging mechanisms used in VM; the interested reader is referred to [1, 23, 24, 25, 26, 27, 28] for additional technical details. We then discuss our approach for addressing memory and paging overheads in large-scale parallel time-sharing systems, which is based on the VM mechanisms and extensions tailored to the parallel computing environments of interest.

The basic idea behind block paging is quite simple: the system identifies sets of pages that tend to be referenced together and then pages each of these sets into memory and out to disk as a unit. This strategy generalizes previous paging and swapping methods in that defining the block size to be 1 page yields demand paging and defining the block size to be the entire working set yields swapping. Block sizes in between these two extremes provide additional degrees of freedom to optimize various performance objectives.

The primary motivation for the block-paging mechanisms in VM was to improve the response times of interactive jobs by amortizing the cost of accessing disk over several pages, thus reducing the waiting time and processor overhead of demand paging techniques and reducing the number of times the paging paths need to be executed. For example, it is shown in [23, 24] that the delay to fetch a single page from a particular IBM 3380 disk configuration is 29ms, whereas the delay to fetch 10 pages from a simpler 3380 configuration is 48ms.

The VM paging system gathers various data and employs a number of algorithms exploiting these data for the creation and dynamic adjustment of page blocks. These algorithms attempt to identify pages that are referenced together. To identify such pages with temporal (and address) affinity, the VM algorithms are applied locally to each address space rather than globally across all address spaces. The page replacement algorithms therefore work on a per address-space basis to select pages of similar age and last-reference for similar treatment and eventual placement on disk. In this manner, the time (and space) affinity of pages is used to create blocks of pages that will be written to and read from disk as a unit, subject to the constraint that the block size is tailored to the characteristics of the particular disk(s) employed in the system. As a specific example, the average block size on VM systems is between 9 and 12 pages with a range of 2 to 20 pages [23, 24]. When a page is fetched from disk as part of a block and is never referenced during the block's residence in memory, then the VM algorithms subsequently eliminate the page from the block. In certain cases, the system also chains together related page blocks for additional optimizations.

When a page fault occurs and the page is part of a block, the VM system issues the I/O request(s) in a manner that attempts to bring this particular page into memory as fast as possible. A program controlled interrupt is associated with the frame assigned to this faulting page, which signals when the required page is in memory. Once this interrupt occurs, the system makes the faulting process ready, therefore allowing it to be immediately scheduled. The remainder of the page block continues to be fetched in parallel with the (possible) execution of the process, thus overlapping the I/O and computation *within* an application. All pages in the block are initially marked as not referenced, with the exception of the one causing the original page fault. This page status information together with other temporal (and address) affinity information are used to minimize failures in accurately predicting future page co-reference and to dynamically maintain page blocks. The analysis in [28] shows that the VM paging algorithms are very effective in maintaining appropriate page blocks (e.g., a page is incorrectly placed in a block – in the sense that it is brought in as part of the block but never referenced – less than 13% of the time in practice) and extremely effective at minimizing the impact of disk performance on interactive response times.

There are two basic approaches to address the performance issues related to the memory management component of large-scale parallel environments in general [19], and especially in systems that time-share their resources. One approach consists of allocating jobs to partitions such that the memory requirements of all jobs on each node of the partition fit within the memory available on that node, thus avoiding the memory overhead problem. This approach can severely limit (or eliminate altogether) the degree of time-slicing, however, and for large-scale parallel computing environments such as the Cornell Theory Center workload considered in our study, it is impossible within the context of the distributed hierarchical control schemes described in Section 2. Another basic approach consists of developing memory management schemes to reduce the performance impact of these memory overheads. In the remainder of this section, we sketch one such approach based in part on block paging.

For each application being executed on a node, the operating system gathers data and employs algorithms much like those in VM for the creation and dynamic adjustment of page blocks. When there is a context-switch to a previously executing job and that job encounters a page fault, the operating system issues an I/O request to bring in the faulting page as quickly as possible and sets a program controlled interrupt[4] on the page frame allocated to the faulting page. As soon as the page is brought into memory, the system returns to the execution of this job and the remaining pages of the block are brought into memory in parallel with the execution of the job. Given the memory reference characteristics of many scientific applications [19], the operating system can continue to bring into memory a number of page blocks that are chained to the faulting page block based on time (and space) affinity. The optimal number of additional page blocks that should be brought into memory depends upon the quantum length allocated to the job (and estimates of the program behavior that could be provided by the compiler). Since the scheduling system controls this parameter, it can work together with the memory management system (possibly together with additional compiler support) to employ a reasonable estimate of the best number of chained blocks to fetch into memory for the current time-slice. These page blocks will replace page blocks already in memory. Once again, since the scheduling system controls the time-slice ordering of the execution of the jobs, it can convey this information to the memory management system so that the page replacement policies displace page blocks that are guaranteed to not be referenced for the longest period of time, thus minimizing the amount of page faults encountered by the system. In this manner, the memory management system can effectively set up a pipeline in which the fetching of the set of pages required by a job during its current time-slice and the corresponding writing of memory-resident page blocks to disk (when necessary) are overlapped with useful computation for the job.

7.2 Quanta allocation

A key parameter of any time-sharing policy is the quantum length assigned to each class of jobs. We have used, and continue to use, an analytic approach [21, 22] to gain insights into this problem with which heuristics can be developed for practical gang scheduling policies. A simple resulting heuristic is based on

[4] The system could also have the job spin on an event associated with the page fault being satisfied, depending upon which is more efficient for the given hardware platform.

the relative utilization of the resources by each class. More formally, we define the relative utilization for class i over a particular interval of interest as $\rho_i \equiv (\lambda_i 2^i)/(\mu_i P)$, where λ_i and μ_i are the mean arrival and service rates for class i over the interval, respectively. We then define the simple heuristic of allocating quanta lengths of $(\rho_i/\rho)T$ to each class i, where T is the timeplexing cycle and $\rho \equiv \sum_i \rho_i$. Note that this approach assumes that time-slices assigned to each class i are primarily consumed by jobs of class i over the time period of interest.

To examine the benefits and limitations of this approach, we ran a number of simulations comparing the above heuristic with the uniform approach (i.e., the quantum length for each class is T divided by the number of classes) where the period of one day was used (a finer granularity of four hours was also examined). In order to estimate the expected per-class relative utilization, we adjusted the quantum of each class every day based on the class' utilization the day before. The simple intuition behind this allocation policy is that system utilization may exhibit some form of "temporal locality" in which partitions that are used more heavily than others over a given time interval are likely to continue to do so over the next interval. In fact, comparisons between this approach and using the actual relative utilization for each day (obtained by an off-line analysis of the trace data) demonstrated only small differences in the mean response times realized for each class. In our preliminary simulations, we therefore set the quantum of class i for day d equal to $(\rho_i(d)/\rho(d))T$, where these parameters are as defined above with the addition of the parameter d.

Our preliminary simulation results suggest that this quanta allocation heuristic can work quite well for heavier load situations in which each class has a non-negligible amount of work, as the system resources are being allocated to equalize the work brought to the system by each class. On the other hand, this approach is not appropriate for (nor is it intended for) migration-based gang schedulers (and to a lesser extent, push-down schemes) under lighter loads, since in this case the classes are grabbing resources assigned to each other and the relative utilization is not very representative of the actual allocation of resources. Hence, one possible solution is to have the system use the ρ-based heuristic during heavier load situations, and then switch to a uniform quanta length policy when the load drops below certain thresholds. We are currently studying such approaches in more detail.

Another important aspect of quanta allocation was also observed based upon our queueing-theoretic gang scheduling analysis [21, 22]. In particular, the setting of these policy parameters in gang scheduling systems must address the complex tradeoff between providing preferential treatment to short-running jobs via small quanta lengths at the expense of larger delays for long-running jobs. By allocating multiple quanta to shorter-running job classes for each quantum allocated to longer-running job classes, the system can generalize the optimization problem at hand and provide additional flexibility to optimize various performance objectives. We are currently working on variations of the basic gang scheduling policy in which certain classes are allocated more than one time slice during each timeplexing cycle and they are executed out of order. For example, instead of visiting classes in the order $<$ class-0, class-1, ..., class-8 $>$, the scheduler could execute a timeplexing cycle order $<$ class-0, class-1, class-2, class-0, class-3, class-4, class-0, ..., class-8 $>$. We believe that such policies can be used to significantly improve the performance of job classes with smaller processing requirements while not degrading considerably the performance of jobs with larger processing requirements.

7.3 Job assignment schemes

As previously noted, the migration scheme considered in our experiments provides an upper bound on gang scheduling performance for the actual workload considered. We have been working on a tree-packing scheme that exploits *parasite allocations* (somewhat similar to the alternative scheduling in [3]) by assigning jobs to partitions in the tree that maximize the number of processors kept busy throughout the timeplexing cycle. Much like the migration scheme, this approach uses a priority-based mechanism for choosing among multiple assignments that are equal with respect to keeping processors busy. We are also developing a push-up gang-scheduling policy that is similar to our push-down policy. When two sibling partitions are idle during their designated time-slices, they can be combined to serve jobs in the higher class. When both push-up and push-down scheduling is used, the idle time-slice may be passed either up or down, depending on which class has the largest number of outstanding jobs. Of course, a broad variety of other criteria can be applied depending on the specific performance objectives that are set forth for the scheduling policy. We are in the process of adding these schemes to our simulator and will be evaluating how well they perform relative to the job-migration method.

Another area we are currently investigating concerns different mechanisms for assigning jobs to partitions in a balanced manner. Different functions or algorithms may be required for the different policies (vanilla, push-up, push-down). Moreover, during the assignment of a job in class i, the values at level i as well as its parents and immediate children could be used to determine which half of the tree the job should be allocated on. Finally, it would be interesting to prove the optimality of job assignment mechanisms under simple service time and arrival time assumptions for the jobs in the system.

8 Conclusions

In this paper we evaluated the performance of various aspects of several gang scheduling approaches and compared them with EASY-LL. We developed an event-driven simulator of the various policies and evaluated their performance by applying them on an actual parallel workload from the Cornell Theory Center. Our experimental results demonstrate the performance benefits, trade-offs, and limitations of alternative gang scheduling designs under the specific workload conditions we considered. We proposed several approaches for addressing different aspects of gang scheduling in practice and presented evidence for the potential benefits of some of these approaches. We are continuing to explore these and other issues related to different forms of gang scheduling in large-scale and/or distributed parallel computing environments.

Acknowledgments

We gratefully acknowledge and thank the Cornell Theory Center (CTC), and especially Steve Hotovy, Joseph Skovira and Joseph Riordan, for providing us with the workload data from its SP2; we also thank Steve Hotovy for several fruitful discussions about this data. We thank Pratap Pattnaik from the IBM T. J. Watson Research Center (WRC) for several helpful discussions; we also thank Russ Miller and Bill Tetzlaff from WRC for fruitful discussions on the details of VM's block paging mechanisms, and we thank Liana Fong from WRC for information on the LoadLeveler and EASY-LL schedulers.

References

1. T. Beretvas and W. H. Tetzlaff. Paging enhancements in VM/SP HPO 3.4. Technical Report TB GG22-9467, IBM Washington Syst. Center, May 1984.
2. D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Trans. Comp.*, 38:408–423, March 1989.
3. D. G. Feitelson. Packing schemes for gang scheduling. In *Job Sched. Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (eds.)*, pages 89–110. Springer-Verlag, 1996. LNCS Vol. 1162.
4. D. G. Feitelson and L. Rudolph. Distributed hierarchical control for parallel processing. *Computer*, pages 65–77, May 1990.
5. D. G. Feitelson and L. Rudolph. Mapping and scheduling in a shared parallel environment using distributed hierarchical control. In *Proc. International Conf. Parallel Processing*, volume I, pages 1–8, August 1990.
6. D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *J. Parallel and Distr. Comp.*, 16(4):306–318, December 1992.
7. D. G. Feitelson and L. Rudolph. Evaluation of design choices for gang scheduling using distributed hierarchical control. *J. Parallel and Distr. Comp.*, 35:18–34, 1996.
8. H. Franke, P. Pattnaik, and L. Rudolph. Gang scheduling for highly efficient distributed multiprocessor systems. In *Proc. Frontiers'96*, 1996.
9. A. Hori, H. Tezuka, Y. Ishikawa, N. Soda, H. Konaka, and M. Maeda. Implementation of gang-scheduling on workstation cluster. In *Job Sched. Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (eds.)*, pages 126–139. Springer-Verlag, 1996. LNCS Vol. 1162.
10. S. G. Hotovy. Workload evolution on the Cornell Theory Center IBM SP2. In *Job Sched. Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (eds.)*, pages 27–40. Springer-Verlag, 1996. LNCS Vol. 1162.
11. S. G. Hotovy. Personal communication. 1997.
12. S. G. Hotovy, D. J. Schneider, and T. O'Donnell. Analysis of the early workload on the Cornell Theory Center IBM SP2. In *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Comp. Syst.*, pages 272–273, May 1996.

13. N. Islam, A. Prodromidis, M. S. Squillante, L. L. Fong, and A. S. Gopal. Extensible resource mangement for cluster computing. In *Proc. International Conf. Distr. Comp. Syst.*, May 1997.
14. N. Islam, A. Prodromidis, M. S. Squillante, A. S. Gopal, and L. L. Fong. Extensible resource scheduling for parallel scientific applications. In *Proc. Eighth SIAM Conf. Parallel Processing for Scientific Comp.*, March 1997.
15. T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. W. Felten, G. A. Gibson, A. R. Karlin, and K. Li. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proc. USENIX Symp. Operating Syst. Design and Implementation (OSDI)*, pages 19–34, October 1996.
16. V. M. Lo. Heuristic algorithms for task assignment in distributed systems. *IEEE Trans. Comp.*, 37(11):1384–1397, November 1988.
17. T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proc. USENIX Symp. Operating Syst. Design and Implem. (OSDI)*, pages 3–17, October 1996.
18. J. K. Ousterhout. Scheduling techniques for concurrent syst.. In *Proc. Third International Conf. Distr. Comp. Syst.*, pages 22–30, October 1982.
19. V. G. Peris, M. S. Squillante, and V. K. Naik. Analysis of the impact of memory in distributed parallel processing systems. In *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Comp. Syst.*, pages 5–18, May 1994.
20. J. Skovira, W. Chan, H. Zhou, and D. Lifka. The EASY-LoadLeveler API project. In *Job Sched. Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (eds.)*, pages 41–47. Springer-Verlag, 1996. LNCS Vol. 1162.
21. M. S. Squillante, F. Wang, and M. Papaefthymiou. An analysis of gang scheduling for multiprogrammed parallel computing environments. In *Proc. Annual ACM Symp. Parallel Algorithms and Architectures (SPAA)*, pages 89–98, June 1996.
22. M. S. Squillante, F. Wang, and M. Papaefthymiou. Stochastic analysis of gang scheduling in parallel and distributed syst.. *Perf. Eval.*, 27&28:273–296, 1996.
23. W. H. Tetzlaff. Paging in the VM/XA system product. *CMG Trans.*, 66:55–64, 1989.
24. W. H. Tetzlaff. Paging in VM/ESA. In *Proc. CMG'91 Conf.*, pages 723–734, 1991.
25. W. H. Tetzlaff and T. Beretvas. Paging in VM/370 operating systems. *CMG Trans.*, 53:65–76, 1986.
26. W. H. Tetzlaff, T. Beretvas, W. M. Buco, J. Greenberg, D. R. Patterson, and G. A. Spivak. A page-swapping prototype for VM/HPO. *IBM Syst. J.*, 26:215–230, 1987.
27. W. H. Tetzlaff and R. Flynn. A comparison of page replacement algorithms. In *Proc. CMG'92 Conf.*, pages 1136–1143, 1992.
28. W. H. Tetzlaff, M. G. Kienzle, and J. A. Garay. Analysis of block-paging strategies. *IBM J. Res. and Devel.*, 33(1):51–59, January 1989.
29. F. Wang. *Multiprogramming for parallel and distributed systems.* PhD thesis, Computer Science Department, Yale University, 1997.
30. F. Wang, H. Franke, M. Papaefthymiou, P. Pattnaik, L. Rudolph, and M. S. Squillante. A gang scheduling design for multiprogrammed parallel computing environments. In *Job Sched. Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (eds.)*, pages 111–125. Springer-Verlag, 1996. LNCS Vol. 1162.

This article was processed using the LaTeX macro package with LLNCS style