Scheduling Best-Effort and Real-Time Pipelined Applications on Time-Shared Clusters *

Yanyong Zhang Anand Sivasubramaniam

Department of Computer Science & Engineering
The Pennsylvania State University
University Park, PA 16802.
{yyzhang, anand}@cse.psu.edu

ABSTRACT

Two important emerging trends are influencing the design, implementation and deployment of high performance parallel systems. The first is on the architectural end, where both economic and technological factors are compelling the use of off-the-shelf computing elements (workstations/PCs and networks) to put together high performance systems called clusters. The second is from the user community that is finding an increasing number of applications to benefit from such high performance systems. Apart from the scientific applications that have traditionally needed supercomputing power, a large number of graphics, visualization, database, web service and e-commerce applications have started using clusters because of their high processing and storage requirements. These applications have diverse characteristics and can place different Quality-of-Service (QoS) requirements on the underlying system (low response time, high throughput, high I/O demands, guaranteed response/throughput etc.). Further, clusters running such applications need to cater to potentially a large number of users (or other applications) in a time-shared manner. The underlying system needs to accommodate the requirements of each application, while ensuring that they do not interfere with each other.

This paper focuses on the CPU resources of a cluster and investigates scheduling mechanisms to meet the responsiveness, throughput and guaranteed service requirements of different applications. Specifically, we propose and evaluate three different scheduling mechanisms. These mechanisms have been drawn from traditional solutions on parallel systems (gang scheduling and dynamic coscheduling), and have been extended to accommodate the new criteria under consideration. These mechanisms have been investigated using detailed simulation and workload models to show their pros and cons for different performance metrics.

Keywords: Parallel Scheduling, Coscheduling, Simulation, Clusters.

*This research has been supported in part by NSF grants CCR-9988164, CCR-9900701, DMI-0075572, Career Award MIP-9701475, and equipment grant EIA-9818327.

1. INTRODUCTION

Two important emerging trends are influencing the design, implementation and deployment of high performance parallel systems. At the architectural end, both economic and technological factors are compelling the use of off-the-shelf computing elements (workstations/PCs and networks) to put together high performance clusters. With workstation and network prices dropping daily, and very little infrastructure needed to build clusters, these platforms are becoming commonplace in numerous computing environments.

On the application end, the widespread deployment and availability of clusters for high performance computing has encouraged the user community to try out different demanding problems on these platforms. It is no longer just the traditional scientific or engineering community (for computational physics, chemistry, computer-aided design, etc.) that are using such systems. An increasing number of commercial applications (web servers [16], database engines, e-commerce, image servers, media bases, etc.) are being hosted on clusters (that are usually back-ends for a web-based interface). For instance, IBM supplies a DB-2 distribution [3] that can exploit a cluster's capabilities. The Microsoft Terraserver [12] hosts the earth's multi-resolution satellite images on a cluster. Further, with computing becoming more visual and perceptual, there are a large number of graphics, visualization and image/video processing applications that can also benefit from the capabilities of a cluster. Clusters are being deployed in enterprises and academic departments to cater to the computing needs of numerous users.

These emerging applications often have diverse characteristics in terms of their computation, communication and I/O requirements from the underlying system. They also impose diverse Quality-of-Service (QoS) demands (low response time for transactions and interactivity, high throughput, and even bounded response times or guaranteed throughput) that can be significantly different from those required for traditional scientific applications. In addition, the clusters are not just running one application at a time. Either there could be several entirely different applications (with different demands and QoS parameters) or a single application performs several duties for a large number of users/clients (in clusters deployed as servers for commercial applications, with perhaps different QoS parameters for each client - depending on how important the client is). We observe the former situation even in our university (both in the departmental as well as in the university-level clusters), where different users want to try out their applications (either in the developmental stage or actually running them) on the cluster at the same time. In all these environments it is important for the cluster to cater to the potentially large number of users/applications in a time-shared manner (batching is not really a good idea for these environments, though this may have been an acceptable solution in

certain supercomputing centers). The underlying system needs to accommodate the requirements of each application, while ensuring that they do not interfere with each other. While cluster deployment has become easy, its management and insulation of one job from the execution of another is still a difficult problem [1].

Using simple space sharing (partitioning the cluster nodes and running only one process on each node) is not a good idea in these environments because of the possibly poor utilization and response times that such strategies have shown [8, 28]. Time sharing parallel jobs on parallel systems is a well-researched problem [8, 28, 13, 30, 2, 7, 19]. An important issue that they all try to address is to keep communication between the processes of a job in mind. Specifically, it is important to ensure that the sender process of a job is scheduled when the intended receiver of the same job is scheduled and is waiting for the message. In many parallel systems [21], including clusters [22, 17], message sends and receives are implemented entirely at the user-level (without involving the kernel), with busy-waiting (spinning) used for receives to lower the communication costs. Consequently, with the operating system unaware of user-level messaging actions, situations where the sender and receiver are not concurrently scheduled could happen unless there is a coordinated effort across all the nodes (synchronized). One common technique that is used on several platforms to address this problem is gang scheduling [9, 15]. Gang scheduling, however, employs strict synchronization between the schedulers running at each node of the cluster to decide on the next set of processes to run at their respective nodes, and when to perform the context switch. This requires that time quanta be fairly coarse-grain to offset the synchronization overheads, making the system possibly less responsive to some of the applications we are considering. Further, job allocation requires finding a time slot across the entire cluster where the required number of CPUs are free, potentially becoming inefficient. A new class of scheduling strategies, broadly referred to as dynamic coscheduling, has been proposed [13, 30, 2, 7, 19] to approximate coscheduled execution without explicitly synchronizing the nodes. They also offer the potential of better utilizing the cluster.

While both gang scheduling and dynamic coscheduling strategies can be used for jobs that are only asking for low response times or high throughput (referred to as *Best-Effort (BE)* jobs in this paper), they are not tuned to handle applications that require bounded response times or guaranteed throughput (referred to as *Real-Time (RT)* jobs in this paper) since they do not take into consideration any deadlines/guarantees that jobs may have. These strategies base their decisions on who to schedule, and when, on either initial static assignments (as in the case of gang scheduling) or on messaging actions during execution (as in the case of dynamic coscheduling). They have not been studied or extended for RT jobs, or a mixed workload containing both BE and RT jobs which we have argued above to be important for a cluster. It is important to ensure that the presence of BE jobs does not affect the guarantees for the RT jobs, and the presence of RT jobs does not starve out the BE jobs. While these issues have been studied in the context of CPU scheduling and resource management on uniprocessor systems, there has been no prior exploration of these issues for communicating parallel jobs on clusters.

With this ambitious goal in mind, this paper sets out to extend parallel scheduling mechanisms for multi-class (BE and RT) workloads on clusters. We specifically focus on two job classes in this study: BE parallel jobs, and RT pipelined (and parallel) jobs. The BE jobs are well-understood (such as traditional scientific applications), and have been the target of prior scheduling strategies as well [28]. The RT jobs have been selected based on our experiences with real-time vision applications on clusters [25]. These applications are computationally intense, and the processing capabilities of a uniprocessor system are often insufficient to handle the real-time requirements of such applications. For instance, an obstacle detection algorithm on flying aircrafts (based on a steady input video stream) can demand computation rates higher than 10G operations per second [25]. These requirements are only likely to get more stringent as the applications increase in complexity, and the image resolutions (and data rates) become more demanding. These applications have a sequence of pipelined tasks that need to be performed in such a way that a frame of processed data can be output every d seconds (termed the deadline). We have shown that by partitioning the tasks amongst the workstations of a medium size (16-32 nodes) cluster and streaming the data through these tasks can meet the real-time requirements of this application [26]. Incidentally, the disturbances due to the multiprogrammed nature (shared with other projects) of our departmental cluster during this exercise was in fact the motivation for the work in this paper.

This paper presents three new mechanisms (1GS, 2DCS-TDM, and 2DCS-PS) for handling a mixed workload containing a steady incoming stream of BE and RT jobs on a cluster. 1GS is an extension of traditional gang scheduling, which uses partitioning of the time quanta between the two workload classes to enforce guarantees and prevent starvation. The other two are two-level hierarchical schedulers that allocate a proportion of the time to each class, asking the low level scheduler within each class to manage its processes. The differences between 2DCS-TDM and 2DCS-PS lie in the granularity at which the time division is managed. 2DCS-TDM does rigid time-division multiplexing at a larger granularity, with a global synchronization across the cluster demarcating the division. 2DCS-PS does proportional share scheduling at a much finer granularity without any explicit synchronization. In addition to the mechanisms, admission control strategies for each are also proposed.

The next section gives a quick overview of previous research. The workload and system model are presented in Section 3 with the details of the scheduling strategies in Section 4. Results from simulation experiments are discussed in Section 5, and a summary of the contributions in Section 6.

2. PREVIOUS RESEARCH

We summarize the previous research related to this paper into three categories. The first is on scheduling parallel jobs (BE) on parallel machines. The second and third categories are on scheduling real-time and multi-class (BE and RT) workloads on single node systems.

2.1 Parallel Job Scheduling

While there is a considerable amount of previous research on space-sharing (CPUs are exclusively allocated to a scheduled job) parallel machines, our focus here is primarily on time-shared (or a combination of space and time-shared systems). In general, it is not a good idea to allow the scheduler at each node to independently manage its processes since there could be situations when a process waiting for a message is scheduled on one node even though the sender process has not been scheduled elsewhere. The effect of this problem is more severe in systems with user-level messaging where busy-wait for receiving a message is common (and the kernel is unaware of these receives). Two general mechanisms to address this problem are gang scheduling (GS) and dynamic coscheduling (DCS).

GS schedules the processes of a job at their respective nodes at the same time (for a given time quantum). One way of implement-

ing this is using an Ousterhout matrix [15] where the rows (M) denote the time quanta (as many as the multiprogramming level) and the columns (p) denote the CPUs. The matrix defines what is scheduled at each node during every quantum. At the end of each time quantum, all the nodes synchronize before scheduling the next row.

Strict synchronization and less flexibility in filling idle slots have led to proposals of approximating coscheduled execution without the associated overheads. These mechanisms are broadly referred to as dynamic coscheduling (DCS). Two main messaging actions are used to develop DCS solutions. The first is how to wait for a message - busy-wait (spinning); blocking after spinning for a while (Spin-Block (SB)); or yielding the CPU (lowering priority) after spinning for a while (Spin-Yield (SY)). The other action is on what happens when a message arrives - performing no action (allow the waiting process to pick it up whenever it is eventually scheduled); immediately scheduling the receiver (Demand-based Scheduling); or periodically checking messages and scheduling process that can perform useful work (Periodic Boost (PB)). Using extensive implementations on actual clusters [13], detailed simulations [30] and analytical models [20], we have demonstrated how these mechanisms can be a better alternative for clusters compared to traditional GS. Specifically, PB and PB-SB have been shown to be the best alternatives amongst the choices over a wide spectrum of workloads and cluster environments. Since these schemes are used in this paper, we briefly describe how they work (further details are in [13, 30]).

In PB, processes wait for messages by spinning (consuming CPU cycles) in the hope that the message would be coming shortly. There is a background kernel activity checking message queues in round-robin order (starting with the current process), and picking one with useful work to do (either (a) it is performing CPU computation, or (b) the message that it is waiting for has arrived and the process can proceed past the receive operation). If there is such a process, then the current one is pre-empted, to make room for the selected process. PB tries to reduce the number of system calls, interrupts and the overheads of blocking/unblocking. In SB, on a receive, processes spin for a certain amount of time, following which they make a system call to block themselves. Upon message arrival, the interrupt service routine unblocks the receiver (if it has blocked) which is soon scheduled due to a priority boost. PB-SB is essentially a combination of these two, wherein receive is implemented as SB and there is also a background kernel activity checking message queues (PB).

2.2 Real-Time Scheduling

There is a plethora of prior research on scheduling resources (CPU in particular) for real-time systems. With processes specifying their deadlines, techniques such as Earliest Deadline First (EDF), Rate Monotonic (particularly for periodic jobs), Least Laxity First etc., prioritize real-time processes taking this factor and other issues (when they arrive, how much work they have to do, etc.) into account for scheduling. Proportional allocation of resources to a mixture of jobs with periodic deadlines has also been studied [27]. There has also been prior work on scheduling parallel real-time workloads [4, 11, 18].

2.3 Single Node Multi-class Scheduling

While schemes such as EDF or Rate Monotonic can work for systems in which there are only real-time jobs, there is a potential starvation problem that can materialize with mixed workloads (that contain best-effort applications). Hierarchical schedulers [10] and proportional share schedulers [14, 24, 23, 6] to accommodate multi-class workloads have been proposed. Essentially, they try to perform some kind of time-division multiplexing between the different classes, with different resolutions and criteria used for deciding the time divisions.

To our knowledge, there has been no prior research examining scheduling mechanisms for parallel systems running both best-effort and real-time applications that have processes communicating with each other.

3. WORKLOAD AND SYSTEM MODEL

3.1 Workload Model

As was mentioned earlier, this study focuses specifically on scheduling two workload classes: (a) Real-Time Pipelined Applications, denoted as RT, with a requirement of completing a specified amount of computation within a given period; and (b) Best-Effort Applications, denoted as BE, whose only requirement is to complete the execution as early as possible (defined as response time).

The RT applications are modeled based on our experiences with real-time vision applications on a cluster [25]. An implementation of such an RT application on a cluster involves taking its task graph (each video frame flows through the nodes of the task graph one after another), partitioning it (exploiting both parallelism and pipelining) and then streaming the input video frames one after another through the graph (potentially crossing machine boundaries). Our experiences with these applications suggest that the task graphs (usually acyclic) are reasonably long (though several of the contiguous stages of the pipeline could be mapped on to the same process on a workstation), but the fan-out of the graph is often quite small (between 1 and 2) [25].

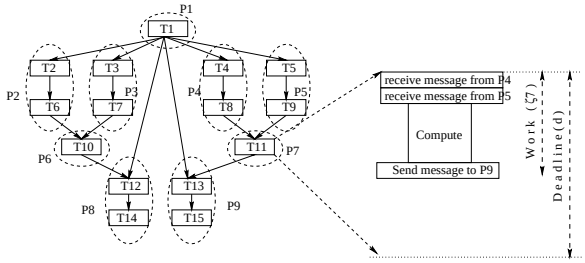


Figure 1: Example task graph of an RT job with 15 tasks (T1-T15) mapped on to 9 processes (P1-P9). P7 has been enlarged to illustrate the typical behavior of a process.

An example application task graph (that closely matches the computation and dataflow in a morphological filter application [25]), and its mapping on to processes is shown in Figure 1.

The processes can in-turn be mapped on to the nodes (workstations) of the cluster (mapping of tasks to processes, and processes to workstations has been studied elsewhere [8]). Our RT workload generates a sequence of such applications (and the mapping of the tasks on to processes) with varying number of stages and fan-outs. The application requirement is to be able to process a video frame every d seconds (termed the deadline), i.e. a frame should be processed/output by the last stage of the task graph every d seconds (apparently, this is also the requirement for each stage of the pipelined task graph). If the system is not able to achieve this, then a deadline is said to have been "missed". Figure 1 shows an example RT application. Each RT process waits for messages from other processes (in this example, Process 7 waits for messages from Processes 4 and 5), performs a specified amount of computation (ζ) and sends the output (say a frame) to one or more processes (Process 9 in this case). In practice, video streaming/processing appli-

cations operate with a certain number of buffers (b, in terms of the number of frames), and could tolerate a certain amount of slackness, i.e. it suffices that b frames are produced in $\Gamma = b \times d$ seconds (called the *period*). If on the other hand, the system produces only n ($n < b$) frames, then we calculate the *miss rate* as $\frac{b-n}{b}$ during this period. Every incoming RT application has a deterministic task graph (this is largely true for many vision applications) and the deadline parameters (b and d).

The applications for the BE workload are generated based on our prior experiences with the use of such workloads for scheduling studies [30]. Typically, each of the parallel processes of a BE application goes through a sequence of computation, I/O, sending and receiving a message (to one or more processes of the same application running at other nodes) iteratively. Several scientific parallel applications have been shown to fall into this paradigm [5]. The relative computation, I/O and communication intensities (and patterns) can be varied to model different application behaviors. Further details on the workload characteristics for these applications are omitted here and the reader is referred to [30]. No information about the application is made available to the system before-hand, except for the number of CPUs (s) that it requires.

Our workload composer generates a stream of RT and BE jobs with arrival rates λ_{RT} and λ_{BE} respectively. The job parameters (task graph and deadlines for RT jobs, and job size for BE jobs) are generated from a specified distribution for each application.

3.2 System Model

Our cluster model assumes a fixed number of workstations (p), connected by a high speed system area network (with hardware and software message transfer costs). Each node of the cluster has a local scheduler which uses a multilevel feedback priority scheduler. Whenever Gang Scheduling is needed, all the local schedulers synchronize (with an associated cost δ), exchange any information that may be needed, recompute who needs to be scheduled for the next time quantum (Ousterhout matrix [15]), and then switch to that process. The other scheduling mechanisms can be implemented by manipulating the priorities of the processes in the feedback queues. Costs are associated with context switching (ω) and priority queue manipulations.

Each job class has an arrival/waiting queue associated with it. If an RT job cannot be allocated CPUs (the details of allocation are scheduling scheme specific, and will be discussed in the next section), then it is put in the RT-arrival queue to be serviced in FCFS order. However, we also allow a parameter w for each RT job, which specifies the maximum time it can wait in the queue (and if it does not find the CPUs within this time, then it is evicted from the system). A BE job also waits in its queue (BE-arrival) if it does not find the CPUs, but it does not have any w parameter associated with it.

An important consideration from the system administrator's perspective is the fairness to different jobs, so that jobs in one class do not interfere with jobs in another. Hence, our model incorporates a *fairness ratio* ($x : y$, which denotes the relative percentage of cluster time to be spent executing RT and BE jobs respectively), that the system administrator can supply. It should be noted that this ratio becomes important only when one class starts interfering with the other, and the actual proportions may be different when the system is under-utilized in any one of the classes.

4. SCHEDULING MECHANISMS

In the following discussion, we first present three schemes that can be used to handle the mixed workload. All these schemes are discussed without any admission control capabilities. In section 4.4, we illustrate the algorithms that can be used to enforce admission control for each of the three schemes. In essence, we attempt to partition (in terms of *time*) the resources across the cluster between the RT and BE classes based on the fairness ratio. The schemes differ in how this is achieved. One way is to perform the partitioning at a much coarser granularity, and synchronize globally between these partitions. 1GS and 2DCS-TDM use this approach, with the differences between them being in the scheduling that is done within the partitions themselves (the former synchronizes periodically, while the latter leaves it entirely to local schedulers at each node). The other way is to perform the partitioning at a much finer granularity which is done in 2DCS-PS, where there is no explicit (global) synchronization in this scheme.

For each scheme, we explain the *initial allocation* (how is a job allocated the s CPUs when it arrives), the *optimization* (after allocation, are there any enhancements to boost the system utilization), and the *execution* (what happens at each node to execute the processes at every instant).

4.1 One-level Gang Scheduler (1GS)

Our first scheme extends traditional gang scheduling (GS) for the two class workload. We use the Ousterhout matrix ($M \times p$, that is used in normal GS) for process assignment and scheduling [15]. The rows are split into two sets, corresponding to the two application classes (RT and BE), with the relative proportion of rows between the sets conforming to the fairness ratio ($x : y$).

- **Initial Allocation:** When an RT or BE job requesting s CPUs arrives, we look for a row with at least s empty slots in its corresponding set. If we are successful, the job is assigned to those slots. We do not consider allocating an RT job in the BE set of rows, or vice-versa, for the initial allocation (this is because we want to try to adhere to the fairness criteria in the allocation). On the other hand, if we cannot find such a row, then the job is put into the arrival (waiting) queue of that class. An RT job will be evicted from the system after w time specified by the application. When a job departs, we try to allocate CPUs to the waiting jobs in the corresponding arrival queue using the same allocation algorithm in FCFS order.

- **Optimization:** It may happen during the course of execution, that one of the sets (of rows) is under/over utilized than the other. To spread out the load more evenly between the two sets, and still try to adhere to the fairness proportion, we perform the following four-step optimization:

  1. We start with the RT set, and try to find slots for jobs that can be replicated within that set of rows itself (to facilitate gang scheduling, the entire job needs to be replicated, and these need to be replicated down the same columns of the matrix since we do not consider migration). It should be noted that we only consider jobs that are not yet achieving the desired throughput (number of produced frames $n < b$) for replication, and leave the jobs which are already producing the desired throughput (number of produced frames $n \geq b$) in place.
  2. We next perform a similar operation for the jobs in the BE set: trying to find rows in that set to accommodate existing BE jobs. Again, migration is not considered. However, in this step, all BE jobs (since a BE job can potentially benefit from as much of the CPU that it can get) are considered as candidates.

3. At the end of these two operations, there could still be idle slots in the two sets. As a result, we try to find rows in BE set to accommodate existing RT jobs whose desired throughput has not yet been achieved.
4. Similarly, BE jobs can potentially be replicated in the idle slots of the RT set.

The resulting matrix assignments hold only until the next job arrival or departure event. At this event, we completely remove duplicates in our matrix (leaving the existing jobs in their initial allocation slots), and then perform the allocation followed by the optimization/replication described above. This cleansing of the matrix makes it more fair to newer arrivals or for waiting jobs.

- **Execution:** Once allocation and optimization are performed, scheduling at each node is no different from normal gang scheduling. At every time quantum (duration Q), each node schedules the corresponding process of a job in its slot, after which it synchronizes with the other nodes before proceeding to the next row/quantum. Receives for both BE and RT jobs are implemented using busy-wait (spinning).

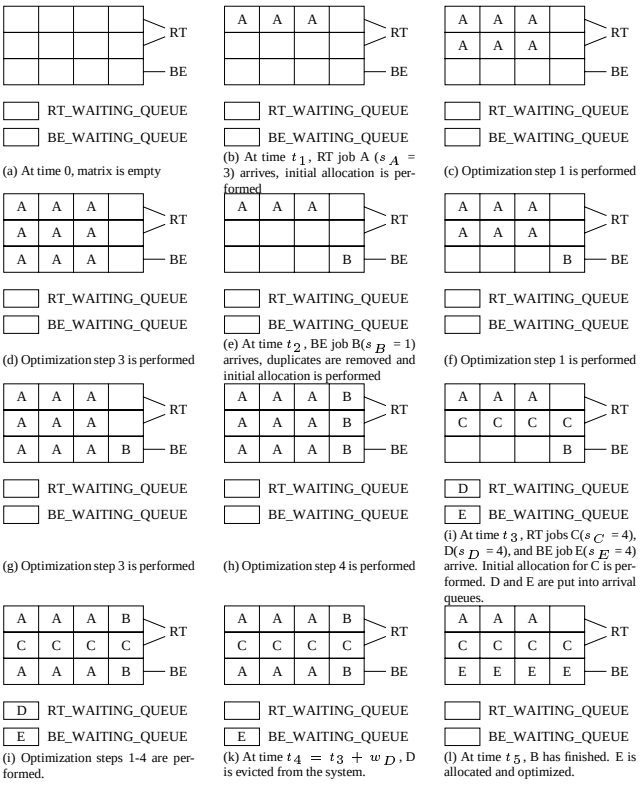


Figure 2: An example illustrating 1GS with p = 4, M = 3, $(x : y)$ = 2:1

Figure 2 shows 1GS at work. With this approach, the division of rows between the two classes and the initial allocation of a job ensures that the CPU is allocated between the two classes as specified by the fairness ratio $(x : y)$. The duplication within and across the two classes can allow for better CPU utilization and a more even balance of the load across the classes.

4.2 Two-level Dynamic Coscheduler with Rigid TDM (2DCS-TDM)

While 1GS is simple to implement, and has some advantages in terms of being able to provide deterministic admission control (discussed later), there are a few shortcomings, which are primarily a consequence of gang scheduling. First, there is the cost of paying a synchronization overhead (δ) at the end of each time quantum (even within a class), which can become quite significant for loosely coupled systems such as clusters (can run into milliseconds for medium to large scale clusters). Using larger time quanta to offset this cost, can make the system less responsive (approaches batching or space sharing [30]). Second, there is a higher chance of fragmentation (idle slots), since gang scheduling requires that processes of a job be allocated across the same row of the matrix. Finally, gang scheduling can also result in lower system utilization if processes of a job are more skewed or I/O intensive (leading to idle times either due to blocking for a message or waiting for I/O completion within its time quantum). These reasons lead us to the next mechanism where we use a two-level (hierarchical) scheduler. The top level global (across the nodes) scheduler simply does a round-robin time allocation (Time-division Multiplexing (TDM)) of the entire cluster between the RT and BE classes with the proportion of the time corresponding to the fairness ratio $(x : y)$. A low level local (at each node) scheduler for each class manages the processes for its time-division.

- **Allocation:** An incoming RT job (with size s) is assigned CPUs as follows. Let us say that Σ_i represents the sum of the work (ζ) of all the RT processes already assigned to node i, and Ω_i represents the minimum $b \times d$ of all such RT processes at node i. The CPUs are sorted in descending order of their laxity (defined as $\frac{x}{x+y} \times \Omega_i - \Sigma_i$). The s processes of the RT jobs are sorted in descending order of their work (ζ). One by one, the sorted list of processes are assigned to the CPU with the most laxity (and updating the laxity after each such assignment). Note that more than one process of a job can be assigned to the same CPU if it is lightly loaded. When a BE job arrives in the system, it is assigned to those CPUs that have the least number of BE processes assigned to them (BE job information, other than s is unavailable to the system). Without admission control, this scheme allows any number of RT or BE jobs to be allocated CPUs in the two time-divisions (and can thus result in better utilization than 1GS).
- **Execution:** As mentioned earlier, the top level scheduler allocates the time (say at a granularity of Υ units) to the two classes in the $x : y$ ratio (i.e. the RT and BE classes alternatively get $\frac{x}{x+y} \times \Upsilon$ and $\frac{y}{x+y} \times \Upsilon$ units respectively), with a global synchronization (with cost δ) used to denote the time-division change.

  The low level scheduler for each class is as follows.

  – For the RT class, the scheduler we employ is Spin-Block with Earliest Deadline First (EDF), i.e. at every instant within this time-division, the process with the earliest deadline is the one that is executing as long as it has not blocked (waiting for a message or I/O). The waiting for a message is implemented as Spin-block (explained in Section 2). Pre-emption of the current RT process is possible when some other RT process with an earlier deadline on that node becomes ready (arrives newly or the event that it is waiting for has unblocked it), or the time-division for this class expires (in which case a BE process needs to be scheduled).

– For the BE class, the scheduler we employ uses Periodic Boost with Spin Block (PB-SB) explained in Section 2 which has been shown to give good performance for best-effort applications.

- **Optimization:** When a node is in the RT division, and there are no RT processes ready to execute (either blocked or not present at all) or the RT processes that are ready have already completed their required computations (i.e. they have produced $n \geq b$ frames, and hence it may not help scheduling them any further in its current period), we allow the BE scheduler on that node to take over at this time to schedule its class of processes using PB-SB. Similarly, in the BE time division, if there are no processes ready to proceed (blocked due to I/O or message wait or there are no BE processes on that node at all), then the RT scheduler is allowed to schedule its processes. In these cases, a BE job that is being scheduled in the RT time division is preempted if an RT job becomes ready, and vice-versa.

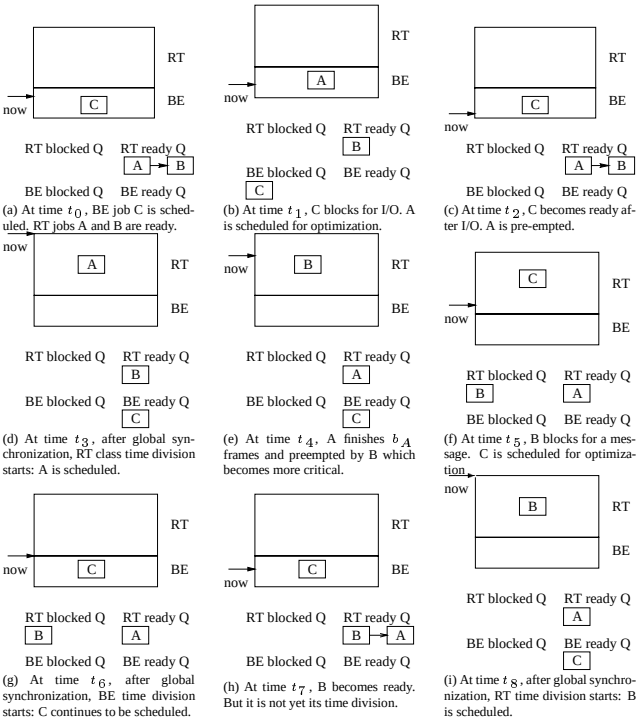


Figure 3: An example illustrating 2DCS-TDM with $(x : y)$ = 2:1 on each node. The boxes show the relative proportion of time allocated to the two classes, and the job inside the box shows what is currently scheduled. The arrow (now) indicates which division is currently being serviced. The ready queue shows what should be scheduled next, and is maintained based on the criticality of the deadlines for RT and based on the priority for BE. Blocked queue holds processes waiting for I/O or a message. *Note that except for the global synchronization between the two time divisions, all other scheduling decisions/actions are entirely local to a node.*

Figure 3 shows how the algorithm works on each node using an example. Unlike 1GS, this scheme does not incur significant synchronization overheads except when the entire cluster shifts from one time division to another (and this cost (δ) would be insignificant compared to the granularity of TDM Υ). Further, there is much less fragmentation since we are not really constrained by row granularity allocations (in fact, we do not use any matrix at all in this approach). The proportional time-division multiplexing in this scheme allows the fairness ratio to be maintained between the two classes irrespective of the arrival pattern of the two classes of jobs.

4.3 Two-level Dynamic Coscheduler with Proportional Share Scheduling (2DCS-PS)

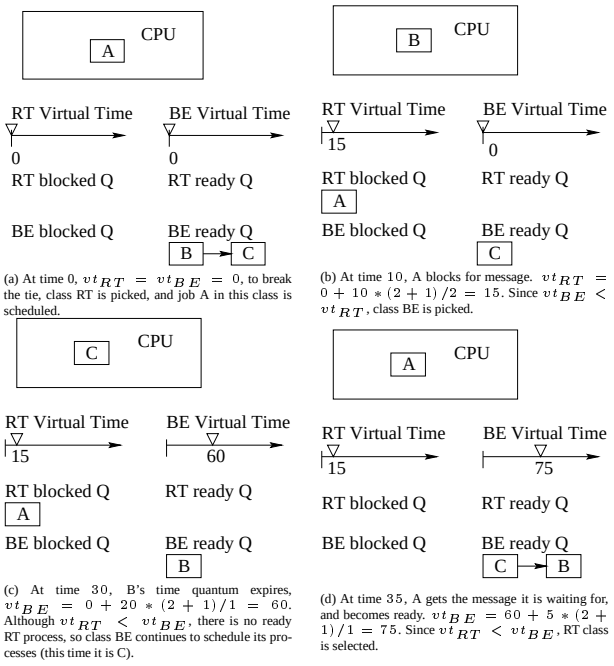
While a coarse-grain TDM as is done in 2DCS-TDM is one way of allocating the CPU resources between the two classes, it may however lead to some inefficiencies within a time division. If there is an under-utilization of one of the divisions by the jobs of that class (causing alternates to be scheduled from the other class), the alternates may not be scheduled exactly as would be desirable. For instance, let us say that there is an under-utilization of the RT class by its jobs, and alternates are selected from the BE class within this division. Such BE processes may not be exactly coscheduled (since each local scheduler is making an independent decision amongst the choices it has). In the third scheme, 2DCS-PS, we attempt to address such situations.

- **Allocation:** Allocation of CPUs upon job arrival is the same as in 2DCS-TDM.

- **Execution:** In this scheme again, we have a 2-level local scheduler at each node, with the top level deciding whether to schedule an RT or BE class next (at a scheduling event). A scheduling event is one of the following: job arrival, job departure, I/O completion, message arrival in spin-block, and time quantum expiration. Fairness ratio $(x : y)$ is enforced by using the notion of a virtual time (vt) for each class. Let us say that a job of class RT is running when a scheduling event occurs, and has been running for time Δ since the last scheduling event. We update the virtual time for this class vt_{RT} as $vt_{RT} = vt_{RT} + \frac{\Delta}{x/x+y}$. On the other hand, if a BE class job is running, its corresponding virtual time (vt_{BE}) is updated similarly. Essentially, the virtual time of the class with the higher proportion is advanced more slowly. After these calculations, the top level scheduler asks the second level scheduler of the class with the lower virtual time to perform its duties. If the selected class does not have any processes to run, then the scheduler of the other class is asked to proceed. It is important to adhere to the fairness ratio only when both classes of jobs coexist (are ready to run) in the system. Hence this algorithm is used only when there is a choice of selecting a job from either class to run. Whenever a job of one class becomes ready (and there was no job in that class ready to run earlier), the virtual time of this class is set equal to that of the other class. The second level schedulers for the two classes are identical to those described for 2DCS-TDM (i.e. Spin-block with EDF for RT class, and PB-SB for BE class). Similar ideas for proportional scheduling have been used in the context of multiple workload classes on single node systems previously [14, 24, 23, 6], but this is the first study to extend these ideas for communicating parallel processes on multiple CPU systems.

- **Optimization:** This scheme already performs CPU multiplexing between the classes at a much finer granularity. When there are no processes ready to execute in one class, the CPU is given to the other class automatically. No further optimizations are needed.

Figure 4 illustrates this scheme showing scheduling decisions made at a node with an example. Since this scheme performs



(a) At time 0, $vt_{RT} = vt_{BE} = 0$, to break the tie, class RT is picked, and job A in this class is scheduled.

(b) At time 10, A blocks for message. $vt_{RT} = 0 + 10 * (2 + 1)/2 = 15$. Since $vt_{BE} < vt_{RT}$, class BE is picked.

(c) At time 30, B's time quantum expires, $vt_{BE} = 0 + 20 * (2 + 1)/1 = 60$. Although $vt_{RT} < vt_{BE}$, there is no ready RT process, so class BE continues to schedule its processes (this time it is C).

(d) At time 35, A gets the message it is waiting for, and becomes ready. $vt_{BE} = 60 + 5 * (2 + 1)/1 = 75$. Since $vt_{RT} < vt_{BE}$, RT class is selected.

Figure 4: An example illustrating 2DCS-PS with $(x : y)$ = 2:1 on each node. The currently scheduled class is shown in the box. *Note that these scheduling decisions/actions are entirely local to a node, and there is no global synchronization at any time.*

proportional share scheduling at a much finer granularity (scheduling events) than 2DCS-TDM, it can provide more opportunities for coscheduling. However, the downside is the possibility (overhead) of switching processes much more often than 2DCS-TDM.

4.4 Incorporating Admission Control

All the above schemes have been described without any admission control for RT jobs, which would be fine if the system is lightly loaded. In fact, only in 1GS is there a waiting queue (the other two admit all arrivals) due to the limitation of number of rows in the Ousterhout matrix. However, the waiting queue in 1GS is only to hold jobs that cannot find the required slots in the matrix (finding the slots does not mean that they or the jobs that are already admitted will meet the deadline since no such check is made). We next explore schemes for regulating the admission of incoming jobs. We perform admission control only for RT jobs and allow BE jobs to come into the system as long as they can find CPUs (they always will in 2DCS-TDM and 2DCS-PS).

An RT job A with a task graph (that is already partitioned into s_A processes) and real-time parameters $\Gamma_A = b_A \times d_A$, and ζ_{A_j} defining the work to be performed by each process j of A. (see Section 3) may need to be regulated. The maximum work ζ_{max} performed by any of its processes (i.e. maximum of all ζ_{A_j}) is used to implement admission control as follows:

- **1GS:** In this scheme, we know that an RT job will get at least one time slice in any time window of size $\Upsilon = M \times Q$ (the time to cycle through all the rows of the Ousterhout matrix). The admission control scheme simply checks if one or more rows of the matrix can be allocated for the incoming job so that it can produce at least $\frac{\Upsilon}{d_A}$ frames in Υ time units. The number of rows r that it requires to meet its deadline is given by $r = \frac{\frac{\Upsilon}{d_A} \times \zeta_{max}}{Q} = \frac{M \times \zeta_{max}}{d_A}$. Note that it needs to find r rows with s_A empty slots down the same columns (since we do not consider migration) that are in the RT set. If we can find r such rows, then A is admitted and the matrix is updated. Else, A is put in the waiting queue.
- **2DCS-TDM** and **2DCS-PS**: We use the following algorithm for admission control in both these schemes. The algorithm tries to see if there is a possible way of accommodating the RT jobs such that they will meet their deadlines if they were allocated their proportional share even within the period of the most stringent RT job (this still does not guarantee that all admitted jobs will meet their deadlines, since actual job execution times cannot be predetermined). Remember that Σ_i represents the sum of the work (ζ) of all the RT processes at node i, and Ω_i represents the minimum $b \times d$ of all RT processes at node i. Then, process j of the incoming job A can be assigned to node i if $\Sigma_i + \zeta_{A_j} \leq min(\Omega_i, b_A \times d_A) \times \frac{x}{x+y}$. If this inequality does not hold, then j cannot be assigned to any other node as well (since we are examining the node with the most slackness first). Else, the process is assigned and Σ_i and Ω_i are updated ($\Omega_i = min(\Omega_i, b_A \times d_A), \Sigma_i = \Sigma_i + \zeta_{A_j}$), and we try the next process of this job. If all processes of A can be thus assigned, job A is admitted.

While the above description gives an overall illustration of how the admission control algorithm works, there are some other subtleties that we have also accounted for. In 1GS, Υ also includes the context switch time δ (from one row to the next). In 2DCS-TDM, we also include the cost of switching the time division δ (requiring global synchronization) in the calculations. Finally, in both 2DCS-TDM and 2DCS-PS, we also include the overhead of unblocking the process (since an RT activity Spin-blocks to wait for a message) conservatively (since it may not always block) to estimate the job execution time, apart from the context switch times.

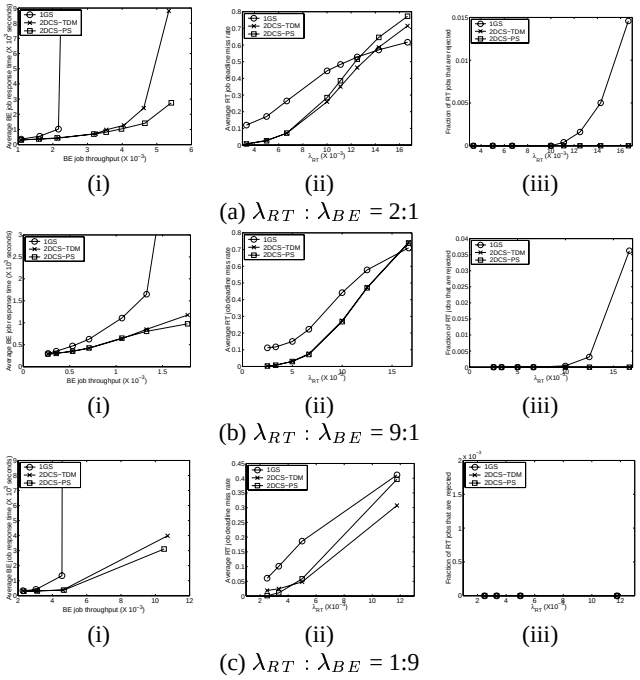
Another important observation to make is the difference in the nature of the admission control algorithms between that for 1GS and the ones for the other two. The former is a deterministic admission control scheme where we are guaranteed that no deadlines will be violated (once admitted, an RT job is guaranteed to get at least r time quanta in each Υ time units). On the other hand, the algorithms for the latter two schemes are mainly heuristics and do not enforce any such guarantees. Since there are no synchronized scheduling efforts across the nodes in these schemes, it could happen that a process could execute longer than that specified by the static parameter ζ, making the above calculations inaccurate. While this may be a drawback, their advantage is that they can allow more RT jobs into the system than the algorithm for 1GS which has a much stricter control. The experiments described later explore these trade-offs.

5. PERFORMANCE RESULTS

We have developed a detailed simulator (extended a previous one described in [30]) to incorporate the described mechanisms. All scheduling activities at a node, including the details of a 60-level feedback queue that is the basis of the local scheduler at each node (modeled after the Solaris scheduler), have been simulated. The appendix gives some of the simulation parameters that are used in the experiments below.

5.1 Comparing the Schemes

An important issue to examine is the effect of the interference of one workload class on another for the three considered schemes. Consequently, we have varied the proportion of RT to BE jobs arriving in the system as 2:1 (load conforming to fairness ratio), 9:1



(i) (ii) (iii)

(a) $\lambda_{RT} : \lambda_{BE}$ = 2:1

(i) (ii) (iii)

(b) $\lambda_{RT} : \lambda_{BE}$ = 9:1

(i) (ii) (iii)

(c) $\lambda_{RT} : \lambda_{BE}$ = 1:9

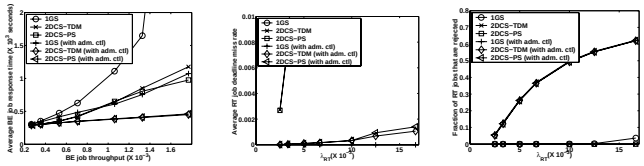
Figure 5: Comparing the schemes for different workload mixes. (a) is a workload conforming to the fairness ration, (b) has a much higher RT load and (c) has a much higher BE load. For each load, (i) shows the average response time for BE jobs as a function of the delivered throughput to BE jobs, (ii) shows the miss rate for the RT jobs as a function of the arrival rate ($\lambda_{RT} + \lambda_{BE}$), and (iii) shows the % of RT jobs that were rejected because they could not be scheduled within w.

(higher RT fraction) and 1:9 (higher BE fraction), and Figures 5 (a), (b) and (c) illustrate the results for these three experiments respectively. For each experiment, the results are provided in 3 graphs: (i) the average response time for BE jobs, (ii) the average deadline miss rates for RT jobs, and (iii) the fraction of RT jobs that are rejected (not admitted). These experiments do not use any admission control.

To explain the results, we would like to point out the criteria of importance to the two workload classes. For the BE jobs, it is important to reduce the time in the Ready queue, as well as the time spent in the Blocked queue (when waiting for messages). The first observation we make is that 1GS is significantly worse than the other two schemes in terms of the response time for BE jobs (for all three experiments). While 1GS reduces the waiting time in the blocked queue for BE jobs (than the other schemes), the time spent in the Ready queue is higher than the other two schemes. However, the main factor in its poor performance is the time spent by a job in the arrival queue before it can be allocated the CPUs (the others do not incur any waiting time in the arrival queue). This observation has also been substantiated by earlier exercises showing the benefits of dynamic coscheduling over Gang Scheduling for BE jobs [30]. Between the other two schemes, 2DCS-PS has a finer granularity of multiplexing the system between the two classes, thus providing better scope for co-scheduling than 2DCS-TDM (see Section 4). As a result, jobs in 2DCS-PS spend less time in the Blocked queue compared to 2DCS-TDM, resulting in a lower response time for BE jobs. We find that this effect is less important when the fraction of jobs in one class becomes significantly higher than the fraction in the other class (see Figure 5 (b)(i) and (c)(i)). 2DCS-PS will approach the behavior of 2DCS-TDM when the granularity (frequency) of multiplexing between the two classes becomes very coarse. This frequency of multiplexing becomes very high when the arrival rates of the two classes are conforming to the fairness ratio, and becomes lower when they become less conforming.

Moving on to the RT class, it is more important that jobs in this class get their required share of the CPU during the course of execution . If we look at any given window of time (of at least Υ) during the system execution, 2DCS-TDM guarantees (deterministic guarantee) that RT jobs will get a certain fraction of the CPUs within this window. On the other hand, for the same time window, 2DCS-PS can only provide a statistical guarantee (over the long run). Such statistical aberrations can result in deadline misses, making 2DCS-PS a little less attractive than 2DCS-TDM for RT class. The aberrations are worse (as a percentage of the number of deadlines within the window) if the RT fraction is lower, making the differences more noticeable in Figures 5 (c)(ii). As with BE class, 1GS fares worse than the other schemes for RT class as well for lower arrival rates. Looking at the miss rate graphs, the behavior of 1GS may appear somewhat counter-intuitive (i.e. at higher arrival rates, its miss rate becomes lower than that for the other schemes). This is because the actual load on the system is much lower since there is a significant rejection ratio (Figure 5(iii) in the experiments). Without admission control, the two dynamic coscheduling schemes do not reject any jobs.

5.2 Impact of Admission Control



(a) Average response time for BE jobs (b) Average miss rate for RT jobs (c) Fraction of RT jobs rejected

Figure 6: Impact of admissin control with $\lambda_{RT} : \lambda_{BE} = 9 : 1$

The previous experiments have used the strategies without any admission control mechanisms. We next augment them with the admission control algorithms described earlier, and the results are presented in Figure 6 for a workload with a 9:1 proportion of RT to BE jobs.

With a lower number of RT jobs admitted into the system, BE jobs have less competition, and have the opportunity of borrowing more time from RT time divisions. As a result, admission control tends to improve the response time of BE jobs as well.

We notice that admission control significantly lowers the miss rate for RT jobs. Specifically, we find that the miss rate drops from 70% to 0.1% for the dynamic coscheduling mechanisms. For the 1GS mechanism, which employs deterministic admission control (we are guaranteed to not violate any deadline), the miss rate drops to zero. This is one factor is favor of 1GS, which can turn out to be a good alternative on systems where strict deadline adherence is a critical issue.

While the admission control algorithms are very effective at dropping the miss rates, this reduction comes at a heavy cost of rejecting several RT jobs (in fact, the reject rates are higher than 50% which can be unacceptable in several situations). However, this suggests directions for future research where one may be able to tolerate a slightly higher miss rate (than what we have achieved) to admit more jobs. This may include relaxing the algorithms, and perhaps opting for statistical admission control schemes. Still, the rejection

rates for the dynamic coscheduling mechanisms are comparable to those for 1GS, making them more suitable for this avenue of research.

In addition to these experiments, the reader is referred to [29] where issues related to stringency of deadlines and span of RT jobs are examined.

6. CONCLUDING REMARKS

With numerous applications currently running on (and anticipated for) clusters, and often operating in a multiprogrammed setting, there is a critical need to meet the diverse demands of the different applications while ensuring that they do not interfere with each other. Apart from the low response times and high throughput required of many applications (such as the scientific codes that high performance systems have traditionally targeted), bounded response times and/or guaranteed throughput are also becoming important considerations (for emerging applications). Already, parallel job scheduling is a hard problem with a goal of scheduling communicating processes at the same time on their respective CPUs while reducing idle times and scheduling overheads. The new Quality-of-Service (QoS) requirements add a new dimension to this problem that has not been considered in depth previously.

Focusing specifically on a two class workload - best-effort (BE) applications that have traditionally been used on clusters, and real-time pipelined (and parallel) applications (that are representative of an important class of vision applications) - we have presented three new scheduling mechanisms to handle the different desirables of these classes. These mechanisms have been built upon traditional parallel job scheduling mechanisms - Gang Scheduling and Dynamic Coscheduling.

The primary contributions of this paper are in the three scheduling algorithms (1GS, 2DCS-TDM, 2DCS-PS) for handling the two class workload on clusters. These algorithms use different levels of granularity to manage the time divisions between the two classes. 1GS uses gang scheduling, with a certain number of time quanta (rows of the Ousterhout matrix) reserved for each class. 2DCS-TDM and 2DCS-PS use hierarchical schedulers, with the top level scheduler deciding the class to be scheduled, and the low level performing the scheduling within each class. The top-level scheduler in the former uses coarse-grain time divisions, while the latter uses fine-grain proportional sharing mechanisms. Spin-block with Earliest Deadline First is used for the RT class, while Periodic Boost with Spin-block is used for the BE class. Optimizations for these algorithms have also been proposed to enhance system utilization during the idle periods of any time division.

This paper has also evaluated these mechanisms in detail with simulation-based experiments. It has been shown that 1GS is quite inflexible, in terms of initial job allocation as well as in terms of optimization to improve system utilization. As a result, it has a higher deadline miss rate than the others for RT jobs, and also higher response times for BE jobs. The inflexibility is also responsible for turning away (rejecting) many RT jobs from the system. The main advantage of 1GS is that it helps us implement a deterministic admission control algorithm with zero miss rates. The two dynamic coscheduling algorithms are much better in terms of BE response times (which has already been observed in several previous studies [13, 30]) and for low RT miss rates. They are able to effectively utilize idle times in one class with jobs from the other class, without significant interferences (or degradation of service quality). Further, they depend much less on the execution on other nodes (less global synchronization). In fact, 2DCS-PS has an entirely local scheduler at each node, and the avoidance of any global synchronization makes it an attractive option from the reliability and fault-tolerance perspectives as well (which is an important consideration for large scale clusters).

The other contributions of this paper are in the admission control schemes for the three scheduling mechanisms. These schemes can help us bring down the miss rates to very low values. There is a cost of turning away (rejecting) a large percentage of jobs in these schemes, and that is an issue of continuing and future work. In particular, we would like to develop admission control strategies that can work with specifiable rejection and miss rate parameters (that can be set by the system administrator). We are also looking to develop additional scheduling strategies to improve system utilization even further without adding significant overheads. Finally, we would like to consider scheduling strategies for more classes of workloads, and more system and workload conditions. We believe that this paper has opened a new and important topic for future research in the area of scheduling and resource management for clusters.

7. REFERENCES

[1] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster Reserves: a mechanism for resource management in cluster-based network servers. In *Proceedings of ACM Sigmetrics*, 2000.

[2] A. C. Arpaci-Dusseau, D. E. Culler, and A. M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proceedings of the ACM SIGMETRICS 1998 Conference on Measurement and Modeling of Computer Systems*, 1998.

[3] `http://www.ibm.com/software/data/db2/`.

[4] D. Babbar and P.Krueger. On-line hard real-time scheduling of parallel tasks on partitionable multiprocessors. In *Proceedings of the 1994 International Conference on Parallel Processing*, pages II: 29–38, August 1994.

[5] D. Bailey et al. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[6] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time(bvt) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, December 1999.

[7] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proceedings of the ACM SIGMETRICS 1996 Conference on Measurement and Modeling of Computer Systems*, pages 25–36, 1996.

[8] D. G. Feitelson. A Survey of Scheduling in Multiprogrammed Parallel Systems. Technical Report Research Report RC 19790(87657), IBM T. J. Watson Research Center, October 1994.

[9] D. G. Feitelson and L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grained Synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, December 1992.

[10] P. Goyal, X. Guo, and H. M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of 2nd Symposium on Operating System Design and Implementation*, pages 107–122, October 1996.

[11] D. D. Kandlur, D. L. Kiskis, and K. G. Shin. HARTOS: a distributed real-time operating system. *Operating Systems Rev.*, 23(3):72–89, July 1989.

[12] Microsoft TerraServer. `http://www.terraserver.microsoft.com`.

[13] S. Nagar, A. Banerjee, A. Sivasubramaniam, and C. R. Das. A Closer Look at Coscheduling Approaches for a Network

of Workstations. In *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 96–105, June 1999.

[14] J. Nieh and M. S. Lam. The Design, Implementation and Evaluation of SMART: A scheduler for Multimedia Applications. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.

[15] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22–30, May 1982.

[16] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-Based Network Servers. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, 1998.

[17] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*, December 1995.

[18] K. Ramamritham, J. A. Stankovic, and P-F. Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184–194, April 1990.

[19] P. G. Sobalvarro. *Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, January 1997.

[20] M. S. Squillante, Y. Zhang, A. Sivasubramaniam, N. Gautam, H. Franke, and J. Moreira. Analytic Modeling and Analysis of Dynamic Coscheduling for a Wide Spectrum of Parallel and Distributed Environments . Technical Report CSE-01-004, Penn State University, CSE department, February 2001.

[21] Thinking Machines Corporation, Cambridge, Massachusetts. *The Connection Machine CM-5 Technical Summary*, October 1991.

[22] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995.

[23] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of 1st Symposium on Operating System Design and Implementation*, November 1994.

[24] C. A. Waldspurger and W. E. Weihl. Stride scheduling:deterministic proportional-share resource management. Technical Report Technical Memo MIT/LCS/TM-528, MIT laboratory for Computer Science, Jun 1995.

[25] M-T. Yang. *An Automatic Pipelined Scheduler for Real-Time Vision Applications*. PhD thesis, Dept. of Computer Science & Eng., The Pennsylvania State University, September 2000.

[26] M-T. Yang, R. Kasturi, and A. Sivasubramaniam. An Automatic Pipeline Scheduler for Real-Time Vision Applications. In *To appear in Proceedings of the International Parallel and Distributed Processing Symposium*, April 2001.

[27] H. Zhang and S. Keshav. Comparison of rate-based service disciplines. In *Proceedings of the conference on Communications architecture & protocols*, pages 113 – 121, 1991.

[28] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam. Improving Parallel Job Scheduling by Combining Gang Scheduling and Backfilling Techniques. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 133–142, May 2000.

[29] Y. Zhang and A. Sivasubramaniam. Scheduling Best-Effort and Real-Time Pipelined Applications on Time-Shared Clusters . Technical Report CSE-01-003, Penn State University, CSE department, February 2001.

[30] Y. Zhang, A. Sivasubramaniam, J. Moreira, and H. Franke. A Simulation-based Study of Scheduling Mechanisms for a Dynamic Cluster Environment. In *Proceedings of the ACM 2000 International Conference on Supercomputing*, pages 100–109, May 2000.

APPENDIX

A. SIMULATION PARAMETERS

Unless explicitly stated otherwise in the paper, the following simulation parameters were used. It should be noted that we have not attempted to perform a comprehensive evaluation of the entire design space extensively varying all the parameters (this is overly ambitious and well beyond the scope of this paper). Rather, our point here is to merely compare the approaches for at least a few reasonable operating values. Many of the values for the hardware and system parameters have been drawn from our earlier experimental exercises on actual systems [13, 30].

- p = 16
  We have also considered other cluster sizes, and similar trends in the results were observed.
- $x : y$ (fairness ratio) = 2:1
- M (for 1GS) = 6
- Q (for 1GS) = 500 millisecs. Time Quantum for BE jobs in 2DCS-TDM and 2DCS-PS varies from 20 ms to 200 ms depending on priority level (similar to Solaris scheduler) [13]. RT jobs in these two classes run to completion/block or until pre-empted by other events.
- Υ (for 2DCS-TDM) = 3 secs
- δ (Synchronization cost in 1GS and 2DCS-TDM) = 1 millisecs
  This is a overly aggressive value that has been chosen intentionally. Even with these optimistic view, we are trying to see how well they compare with 2DCS-PS which does not require any synchronization.
- ω (Context switch overhead) = 200 microsecs
- I (Interrupt cost) = 50 microsecs
  Required whenever Spin-block is employed for receives.
- Overhead for moving process from one scheduling queue to another = 3 microsecs.
  Incurred at pre-emption points, and when certain events occur (PB, message arrival for SB, etc.).
- Checking if a process has a pending message (for PB) = 2 microsecs
- Interval between periodic activity in PB = 1 millisec
  Has been argued to give close to the best performance in [30].
- RT job parameters: duration = 3 minutes, d = 1/30 secs, b = 100 frames, $s = p$, w = 15 secs
  Other varying durations have also been experimented with, and we find there are no significant differences in the trends. d of 1/30 secs is typical of vision applications requiring 30 frames to be processed in a second. b is the buffering capability of the underlying system. s has been chosen to span the entire cluster, since we are dealing with a small/medium sized system, and our experiences suggest that a vision application may require all of its capabilities [25].
- BE job duration follows Erlang distribution with mean of 2 minutes. The size (s) follows a uniform distribution between 2 and p. Computation, communication (frequency, size and pattern), and I/O within each process of a BE job have been discussed earlier in [30].