Time-sharing Parallel Applications With Performance Isolation and Control

Bin Lin Ananth I. Sundararaj [*] Peter A. Dinda

Northwestern University, EECS, Evanston, IL
{b-lin, pdinda, ais}@northwestern.edu

Abstract

Most parallel machines, such as clusters, are space-shared in order to isolate batch parallel applications from each other and optimize their performance. However, this leads to low utilization or potentially long waiting times. We propose a self-adaptive approach to time-sharing *such machines that provides isolation and allows the execution rate of an application to be tightly controlled by the administrator. Our approach combines a periodic real-time scheduler on each node with a global feedback-based control system that governs the local schedulers. We have developed an online system that implements our approach. The system takes as input a target execution rate for each application, and automatically and continuously adjusts the applications' real-time schedules to achieve those rates with proportional CPU utilization. Target rates can be dynamically adjusted. Applications are performance-isolated from each other and from other work that is not using our system. We present an extensive evaluation that shows that the system remains stable with low response times, and that our focus on CPU isolation and control does not come at the significant expense of network I/O, disk I/O, or memory isolation.*

1 Introduction

Tightly-coupled computing resources such as clusters are typically used to run batch parallel workloads. An application in such a workload is typically communication intensive, executing synchronizing collective communication. The Bulk Synchronous Parallel (BSP) model [25] is commonly used to understand many of these applications. In the BSP model, application execution alternates between phases of local computation and phases of global collective communication. Because the communication is global, the threads of execution on different nodes must be carefully scheduled if the machine is time-shared. If a thread on one node is slow or blocked due to some other thread unrelated to the application, all of the application's threads stall.

[*]This work was partially done while this author was at Intel Corporation, Hillsboro, OR. This author is now at Microsoft Corporation, Redmond, WA.

This work is in part supported by NSF Awards ANI-0093221, ANI-0301108, and EIA-0224449, and by gifts from VMware, Dell, and Symantec.

To avoid stalls and provide predictable performance for users, almost all tightly-coupled computing resources today are space-shared. In space-sharing [24], each application is given a partition of the available nodes, and on its partition, it is the *only* application running, thus avoiding the problem altogether by providing complete performance isolation between running applications. Space-sharing introduces several problems, however. Most obviously, it limits the utilization of the machine because the CPUs of the nodes are idle when communication or I/O is occurring. Space-sharing also makes it likely that applications that require many nodes will be stuck in the queue for a long time and, when running, block many applications that require small numbers of nodes. Finally, space-sharing permits a provider to control the response time or execution rate of a parallel job at only a very course granularity. Though it can be argued theoretically that applications can be always built such that computation and I/O overlap all the time, thus preventing stalls, practically speaking, this is rarely the case. We propose a new self-adaptive approach to time-sharing parallel applications on tightly-coupled computing resources like clusters, *performance-targetted feedback-controlled real-time scheduling*. The goals of our technique are to provide

- performance isolation within a time-sharing framework that permits multiple applications to share a node, and
- performance control that allows the administrator to finely control the execution rate of each application while keeping its resource utilization automatically proportional to execution rate.

Conversely, the administrator can set a target resource utilization for each application and have commensurate application execution rates follow.

In performance-targetted feedback-controlled real-time scheduling, each node has a periodic real-time scheduler. The local application thread is scheduled with a $(period, slice)$ constraint, meaning that it executes $slice$ seconds every $period$. Notice that $slice/period$ is the utilization of the application on the node. Our implementation uses our previously described [11] and publicly available VSched tool. VSched is a user-level periodic real-time scheduler for Linux that we originally developed to explore scheduling interactive and batch workloads together. Section 3 provides an overview.

Once an administrator has set a target execution rate for an application, a global controller determines the appropriate constraint for each of the application's threads of execution and then contacts each corresponding local scheduler to set it. The controller's input is the desired application execution rate, given as a percentage of its maximum rate on the system (i.e., as if it were on a space-shared system). The application or its agent periodically feeds back to the controller its current execution rate. The controller automatically adjusts the local schedulers' constraints based on the error between the desired and actual execution rate, with the added constraint that utilization must be proportional to the target execution rate. In the common case, the only communication in the system is the feedback of the current execution rate of the application to the global controller, and synchronization of the local schedulers through the controller is very infrequent. Section 4 describes the global controller in detail.

It is important to point out that our system schedules the CPU of a node, not its physical memory, communication hardware, or local disk I/O. Nonetheless, in practice, we can achieve quite good performance isolation and control even for applications making significant use of these other resources, as we show in our detailed evaluation (Section 5). Mechanisms for physical memory isolation in current OSes and VMMs are well understood and can be applied in concert with our techniques. As long as the combined working set size of the applications executing on the node does not exceed the physical memory of the machine, the existing mechanisms suffice. Communication has significant computational costs, thus, by throttling the CPU, we also throttle it. The interaction of our system and local disk I/O is more complex. Even so, we can control applications with considerable disk I/O.

The primary contributions of our work to the state of the art are the following:

- We have described, implemented, and evaluated a new approach to time-sharing parallel applications with performance isolation. The approach is based on periodic real-time scheduling of the nodes combined with global control of the real-time constraints.
- We have demonstrated that this approach also provides a simple way to control the execution rate of applications while maintaining efficiency.

2 Related work

Our work ties to gang scheduling, implicit co-scheduling, real-time schedulers, and feedback control real-time scheduling. As far as we aware, we are the first to develop real-time techniques for scheduling parallel applications that provide performance isolation and control. We also differ from these areas in that we show how external control of resource use (by a cluster administrator, for example) can be achieved while maintaining commensurate application execution rates. That is, we can reconcile administrator and user concerns.

The goal of gang scheduling [19, 9] is to "fix" the blocking problems produced by blindly using time-sharing local node schedulers. The core idea is to make fine-grain scheduling decisions collectively over the whole cluster. For example, one might have all of an application's threads be scheduled at identical times on the different nodes, thus giving many of the benefits of space-sharing, while still permitting multiple applications to execute together to drive up utilization, and thus allowing jobs into the system faster. In essence, this provides the performance isolation we seek, while performance control depends on scheduler model. However, gang scheduling has significant costs in terms of the communication necessary to keep the node schedulers synchronized, a problem that is exacerbated by finer grain parallelism and higher latency communication [10]. In addition, the code to simultaneously schedule all tasks of each gang can be quite complex, requiring elaborate bookkeeping and global system knowledge [23].

Implicit co-scheduling [1] attempts to achieve many of the benefits of gang scheduling without scheduler-specific communication. The basic idea is to use communication irregularities, such as blocked sends or receives, to infer the likely state of the remote, uncoupled scheduler, and then adjust the local scheduler's policies to compensate. This is quite a powerful idea, but it does have weaknesses. In addition to the complexity inherent in inference and adapting the local communication schedule, the approach also doesn't really provide a straightforward way to control effective application execution rate, response time, or resource usage.

The feedback control real-time scheduling project at the University of Virginia [16, 21, 15, 17] had a direct influence on our thinking. In that work, concepts from feedback control theory were used to develop resource scheduling algorithms to give quality of service guarantees in unpredictable environments to applications such as online trading, agile manufacturing, and web servers. In contrast, we are using concepts from feedback control theory to manage a tightly controlled environment, targeting parallel applications with collective communication.

Feedback-based control was also used to provide CPU reservations to application threads running on a single machine based on measurements of their progress [22], for controlling coarse-grained CPU utilization in a simulated virtual server [27], for dynamic database provisioning for web servers [2], and to enforce web server CPU entitlements to control response time [14].

There are a wide range of implementations of periodic real-time schedulers, for example [3, 18], including numerous kernel extensions for Linux, for example [8, 20].

3 Local scheduler

In the periodic real-time model, a task is run for $slice$ seconds every $period$ seconds. Using earliest deadline first (EDF) schedulability analysis [12], the scheduler can determine whether some set of $(period, slice)$ constraints can be met. The scheduler then uses dynamic priority preemptive scheduling with the deadlines of admitted tasks as priorities.

VSched is a user-level implementation of this approach for Linux that offers soft real-time guarantees. It runs as a Linux process that schedules other Linux processes. Because the Linux kernel does not have priority inheritance

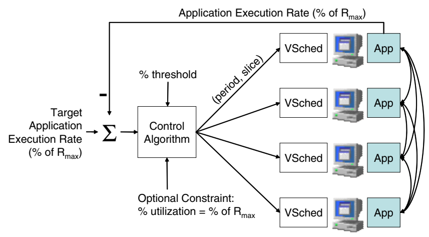

Application Execution Rate (% of R_max)
-
% threshold
(period, slice)
Target Application Execution Rate (% of R_max)
Σ
Control Algorithm
VSched
App
Optional Constraint: % utilization = % of R_max

Figure 1. Structure of global control.

mechanisms, nor known bounded interrupt service times, it is impossible for a tool like VSched to provide hard real-time guarantees to ordinary processes. Nonetheless, as we show in an earlier paper [11], for a wide range of periods and slices, and under even fairly high utilization, VSched almost always meets the deadlines of its tasks, and when it misses, the miss time is typically very small. VSched supports $(period, slice)$ constraints ranging from the low hundreds of microseconds (if certain kernel features are available) to days. Using this range, the needs of various classes of applications can be described and accommodated. VSched allows us to change a task's constraints within about a millisecond.

VSched is a client/server system. The VSched server is a daemon running on Linux that spawns the scheduling core, which executes the scheduling scheme described above. The VSched client communicates with the server over an encrypted TCP connection. In this work, the client is driven by the global controller and we schedule individual Linux processes.

The performance of VSched has been evaluated on several different platforms. It can achieve very low deadline miss rates up to quite high utilizations and quite fine resolutions. VSched can use over 90% of the CPU even on relatively slow hardware and older kernels (Intel® Pentium® III, 2.4 kernel) and can use over 98% of the CPU on more modern configurations (Intel® Pentium® 4, 2.6 kernel). The mechanisms of VSched its evaluation and related work are described in much more detail in an earlier paper [11] and the software itself is publicly available.

4 Global controller

The control system consists of a centralized feedback controller and multiple host nodes, each running a local copy of VSched, as shown in Figure 1. A VSched daemon is responsible for scheduling the local thread(s) of the application(s) under the yoke of the controller. The controller sets $(period, slice)$ constraints using the mechanisms described in Section 3. Currently, the same constraint is used for each VSched. One thread of the application, or some other agent, periodically communicates with the controller using non-blocking communication.

4.1 Inputs

The maximum application execution rate on the system in application-defined units is R_{max}. The set point of the controller is supplied by the user or the system administrator through a command-line interface that sends a message to the controller. The set point is r_{target} and is a percentage of R_{max}. The system is also defined by its threshold for error, ϵ, which is given as percentage points. The inputs Δ_{slice} and Δ_{period} specify the smallest amounts by which the slice and period can be changed. The inputs min_{slice} and min_{period} define the smallest slice and period that VSched can achieve on the hardware.

The current utilization of the application is defined in terms of its scheduled period and slice, $U = slice/period$. The user requires that the utilization be proportional to the target rate, that is, that $r_{target} - \epsilon \leq U \leq r_{target} + \epsilon$.

The feedback input $r_{current}$ comes from the parallel application we are scheduling and represents its current execution rate as a percentage of R_{max}. To minimize the modification of the application and the communication overhead, our approach only requires high-level knowledge about the application's control flow and only a few extra lines of code.

4.2 Control algorithm

The control algorithm (or simply the algorithm) is responsible for choosing a $(period, slice)$ constraint to achieve the following goals

1. The error is within threshold: $r_{current} = r_{target} \pm \epsilon$, and

2. That the schedule is efficient: $U = r_{target} \pm \epsilon$.

The algorithm is based on the intuition and observation that application performance will vary depending on which of the many possible $(period, slice)$ schedules corresponding to a given utilization U we choose, and the best choice will be application dependent and vary with time. For example, a finer grain schedule (e.g. (20ms, 10ms)) may result in better application performance than coarser grain schedules (e.g. (200ms, 100ms)). At any point in time, there may be multiple "best" schedules.

The control algorithm attempts to automatically and dynamically achieve goals 1 and 2 in the above, maintaining a particular execution rate r_{target} specified by the user while keeping utilization proportional to the target rate.

We define the error as

$$e = r_{current} - r_{target}.$$

At startup, the algorithm is given an initial rate r_{target}. It chooses a $(period, slice)$ constraint such that $U = r_{target}$ and $period$ is set to a relatively large value such as 200 ms. The algorithm is a simple linear search for the largest $period$ that satisfies our requirements.

When the application reports a new current rate measurement $r_{current}$ and/or the user specifies a change in the target rate r_{target}, e is recomputed, followed by:

- If $|e| > \epsilon$ decrease $period$ by Δ_{period} and decrease $slice$ by Δ_{slice} such that $slice/period = U = r_{target}$. If $period \leq min_{period}$ then we reset $period$ to the same value as used at the beginning and again set $slice$ such that $U = r_{target}$.
- If $|e| \leq \epsilon$ do nothing.

It should be noticed that the algorithm always maintains the target utilization and searches the $(period, slice)$ space from larger to smaller granularity, subject to the utilization constraint. The linear search is, in part, done because multiple appropriate schedules may exist. We do not preclude the use of algorithms that walk the space faster, but we have found our current algorithm to be effective.

5 Evaluation

In presenting our evaluation, we begin by explaining the experimental framework. Then we show the range of control that the scheduling system has made available. This is followed by an examination of using the algorithm described above to prevent the inevitable drift associated with simply using a local real-time scheduler. Next, we examine the performance of the algorithm in a dynamic environment, showing their reaction to changing requirements. We then illustrate how the system remains impervious to external load despite the feedback. Next, we show how the system scales as it controls increasing numbers of parallel applications. Finally, we examine the effects of local disk I/O and memory contention.

5.1 Experimental framework

As mentioned previously, Bulk Synchronous Parallel (BSP [6]) model is used to characterize many of the batch parallel workloads that run in tightly coupled computing resources such as clusters. In most of our evaluations we used a synthetic BSP benchmark, called Patterns, written for PVM [5]. Patterns is described in more detail in a previous paper [7], but the salient points are that it can execute any BSP communication pattern and run with different compute/communicate (comp/comm) ratios and granularities. In general, we configure Patterns to run with an all-to-all communication pattern on four nodes of our IBM e1350 cluster (Intel® Xeon® 2.0 GHz, 1.5 GB RAM, Gigabit Ethernet interconnect, Linux 2.4.20). Each node runs VSched, and a separate node is used to run the controller. Note that all of our results involve CPU and network I/O.

We also evaluated the system using an NAS (NASA Advanced Supercomputing) benchmark. In particular, we use the PVM implementation of the IS (Integer Sort) benchmark developed by White et al. [26]. It performs a large integer sort, sorting keys in parallel as seen in large scale computational fluid dynamic (CFD) applications. IS combines integer computation speed and communication with, unlike Patterns, different nodes doing different amounts of computation and communication.

5.2 Range of control

To illustrate the range of control possible using periodic real-time scheduling on the individual nodes, we ran Patterns with a compute/communicate ratio of 1:2, making it quite communication intensive. Note that this configuration is conservative: it is far easier to control a more loosely coupled

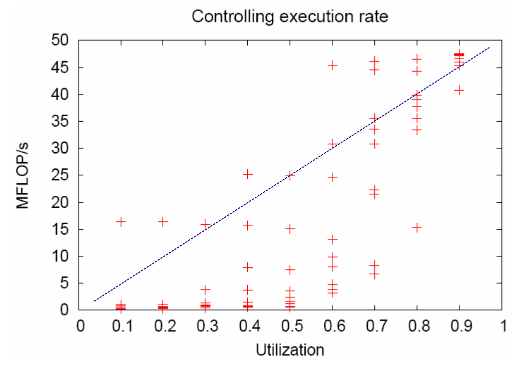


Figure 2. Compute rate as a function of utilization for different $(period, slice)$ choices.

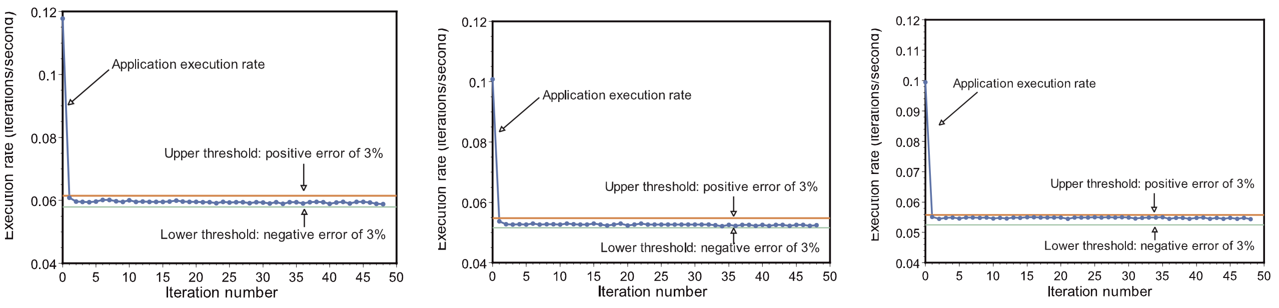
parallel application with VSched. We ran Patterns repeatedly, with different $(period, slice)$ combinations. Figure 2 shows these test cases. Each point is an execution of Patterns with a different $(period, slice)$, plotting the execution rate of Patterns as a function of Patterns utilization on the individual nodes. Notice the line on the graph, which is the ideal control curve that the control algorithm is attempting to achieve, control over the execution rate of the application with proportional utilization ($r_{current} = r_{target} = U$). Clearly, there *are* choices of $(period, slice)$ that allow us to meet all of the requirements.

5.3 Schedule selection and drift

Although there clearly exist $(period, slice)$ schedules that can achieve an execution rate with (or without) proportional utilization, we cannot simply use only the local schedulers for several reasons:

- The appropriate $(period, slice)$ is application dependent because of differing compute/communicate ratios, granularities, and communication patterns. Making the right choice should be automatic.
- The user or system administrator may want to dynamically change the application execution rate r_{target}. The system should react automatically.
- Our implementation is based on a *soft* local real-time scheduler. This means that deadline misses will inevitably occur and this can cause timing offsets between different application threads to accumulate. We must monitor and correct for these slow errors. Notice that this is likely to be the case for a hard local real-time scheduler as well if the admitted tasks vary across the nodes.

Figure 3 illustrates what we desire to occur. The target application execution rate is given in iterations per second, here being 0.006 iterations/second. The current execution rate $r_{current}$ is calculated after each iteration and reported to the controller. This is Patterns running with a 1:1 compute/communicate ratio on two nodes. The lower curve is that of simply using VSched locally to schedule the application. Although we can see that the rate is correct for the first few iterations, it then drifts downward, upward, and once again downward over the course of the experiment. The roughly straight curve is using VSched, the global controller,



(a) high (5:1) comp/comm ratio (b) medium (1:1) comp/comm ratio (c) low (1:5) comp/comm ratio

Figure 4. System in stable configuration for varying comp/comm ratio.

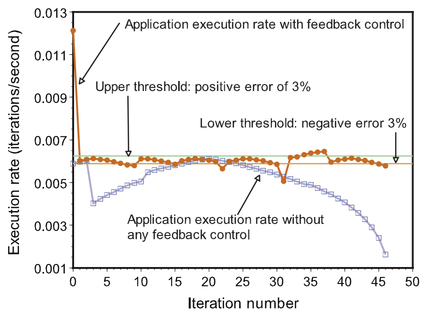


Figure 3. Elimination of drift using global feedback control; 1:1 comp/comm ratio.

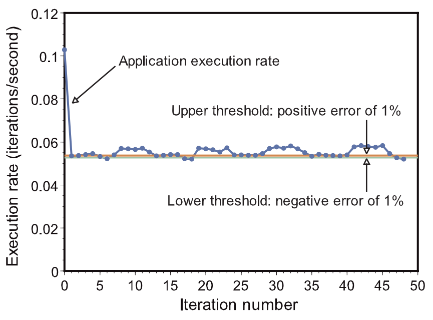


Figure 5. System in oscillation when error threshold is made too small; 1:1 comp/comm ratio.

and the control algorithm. We can see that the tendency to drift has been eliminated using global feedback control.

5.4 Evaluating the control algorithm

We studied the performance of the control algorithm using three different compute/communicate ratios (high (5:1) ratio, medium (1:1) ratio, and low (1:5) ratio), different target execution rates r_{target}, and different thresholds ϵ. In all cases $\Delta_{period} = 2$ ms, where Δ_{period} is the change in period effected by VSched when the application execution rate goes outside of the threshold range, the *slice* is then adjusted such that $U = r_{target}$.

Figure 4 shows the results for high, medium, and low test cases with a 3% threshold. We can see that the target rate is easily and quickly achieved, and remains stable for all three test cases. Note that the execution rate of these test cases running at full speed without any scheduling are slightly different. $r_{current}$ is calculated in the end of every iteration.

Next, we focus on two performance metrics:

- Minimum threshold: What is the smallest ϵ below which control becomes unstable?
- Response time: for stable configurations, what is the typical time between when the target execution rate r_{target} changes and when the $r_{current} = r_{target} \pm \epsilon$?

Being true for all feedback control systems, the error threshold will affect the performance of the system. When the threshold ϵ is too small, the controller becomes unstable and fails because the change applied by the control system to correct the error is even greater than the error. For our control algorithm, when the error threshold is $< 1\%$, the controller will become unstable. Figure 5 illustrates this behavior. Note that while the system is now oscillating, it appears to degrade gracefully.

Figure 6 illustrates our experiment for measuring the response time. The target rate is changed by the user in the middle of the experiment. Our control system quickly adjusts the execution rate and stabilizes it. It shows that the response time is about 32 seconds, or two iterations, for the case of 1:1 compute/communicate ratio. The average response time over four test cases (1 high, 2 medium, and 1 low compute/communicate ratios) is 30.68 seconds. In all cases, the control algorithm maintains $U = r_{target}$ as an invariant by construction.

5.5 Summary of limits of control algorithm

Figure 7 summarizes the response time, communication cost to support the feedback control, and threshold limits of our control system. Overall we can control with a quite small threshold ϵ. The system responds quickly, on the order of a couple of iterations of our benchmark. The communication cost is minuscule, on the order of just a few bytes per iteration. Finally, these results are largely independent of the compute/communicate ratio.

High (5:1) compute/communicate ratio			Medium (1:1) compute/communicate ratio			Low (1:5) compute/communicate ratio		
Response time	Threshold limit	Feedback comm. cost	Response time	Threshold limit	Feedback comm. cost	Response time	Threshold limit	Feedback comm. cost
29.16 s	2 %	32 bytes/iter	31.33 s	2 %	32 bytes/iter	32.01 s	2 %	32 bytes/iter

Figure 7. Response time and threshold limits for the control algorithm.

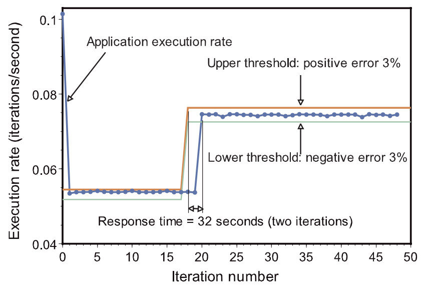


Figure 6. Response time of control algorithm; 1:1 comp/comm ratio.

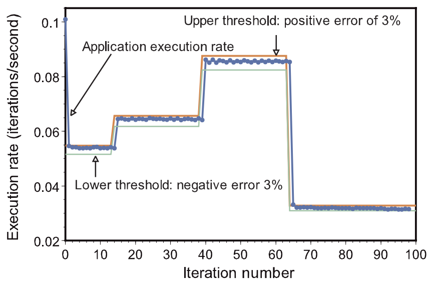


Figure 8. Dynamically varying execution rates; 1:1comp/comm ratio.

The exceptionally low communication involved in performance-targetted feedback-controlled real-time scheduling is a natural consequence of the deterministic and predictable periodic real-time scheduler being used on each node.

5.6 Dynamic target execution rates

As we mentioned earlier, using the feedback control mechanism, we can dynamically change the target execution rates and our control system will continuously adjust the real-time schedule to adapt to the changes. To see how our system reacts to user inputs over time, we conducted an experiment in which the user adjusted his desired target rate four times during the execution of the Patterns application. As shown in Figure 8, the control algorithm works well. After the user changes the target rate, the algorithm quickly adjusts the schedule to reach the target.

5.7 Ignoring external load

Any coupled parallel program can suffer drastically from external load on any node; the program runs at the speed of the slowest node. We have previously shown that the periodic real-time model of VSched can shield the program from such external load, preventing the slowdown [11]. Here we want to see whether our control system as a whole can still protect a BSP application from external load.

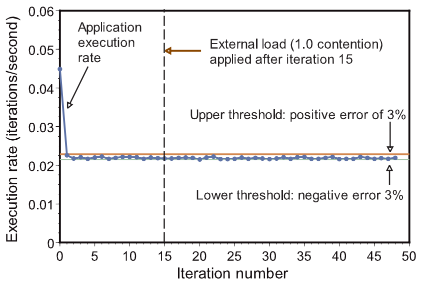


Figure 9. Performance of control system under external load; 3:1 comp/comm ratio; 3% threshold.

We executed Patterns on four nodes with the target execution rate set to half of its maximum rate. On one of the nodes, we applied external load, a program that contends for the CPU using load trace playback techniques [4]. Contention is defined as the average number of contention processes that are runnable. Figure 9 illustrates the results. At roughly the 15th iteration, an external load is placed on one of the nodes in which Patterns is running, producing a contention of 1.0. We note that the combination of VSched and the feedback controller are able to keep the performance of Patterns independent of this load. We conclude that our control system can help a BSP application maintain a fixed stable performance under a specified execution rate constraint despite external load.

5.8 NAS IS Benchmark

When we ran the NAS IS (Integer Sort) benchmark without leveraging our control system, we observed that different nodes have different CPU utilizations. This is very different from the Patterns benchmark, which does roughly the same amount of computation and communication on each node. In our experiment, for a specific configuration of NAS IS executing on four nodes, we observed an average utilization of $\sim$28% for two nodes and $\sim$14% average utilization for the other two nodes.

This variation has the potential to challenge our control system, since in our model we assume the same target utilization U on each node, and we apply the same schedule on each node. We ran an experiment where we set the target utilization to be half of the maximum utilization among all

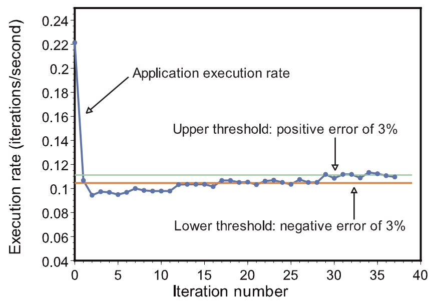


Figure 10. Running NAS benchmark under control system; 3% threshold.

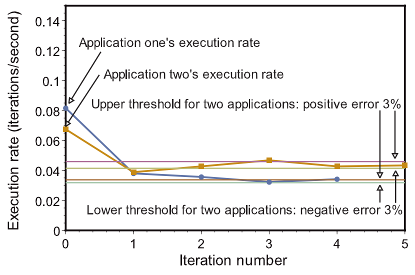


Figure 11. Running of two Patterns benchmarks under the control system, 1:1 comp/comm ratio.

nodes, i.e. 14%. Figure 10 illustrates the performance in this case. We can see that the actual execution rate is successfully brought to within ϵ of the target rate.

We are currently designing a system in which the global controller is given the freedom to set a different schedule on each node thus making our control system more flexible.

5.9 Time-sharing multiple applications

To see how well we can provide time-sharing for multiple parallel applications, we simultaneously executed multiple Patterns benchmarks on the same four nodes of our cluster.

Figure 11 shows the results of running two Patterns applications, each configured with a 1:1 compute/communicate ratio. One was configured with a target rate of 30%, with the other set to 40%. We can clearly see that the actual execution rates are quickly brought to within ϵ of the target rates and remain there for the duration of the experiment. Next, we consider what happens as we increase the number of Patterns benchmarks running simultaneously. In the following, each Patterns benchmark is set to execute with identical 10% utilization. We ran Patterns with a 3:1 compute/communicate ratio. Figure 12 shows our results. Each graph shows the execution rate (iterations/second) as a function of the iteration, as well as the two 3% threshold lines. Figure 12(a) contains two such graphs, corresponding to two simultaneously executing Patterns benchmarks, (b) has three, and so on.

Overall, we maintain reasonable control as we scale the number of simultaneously executing benchmarks. Further, over the thirty iterations shown, in all cases, the average execution rate meets the target, within threshold.

We do notice a certain degree of oscillation when we run many benchmarks simultaneously. Our explanation is as follows. When VSched receives and admits a new schedule sent by the global controller, it will interrupt the current task and re-select a new task (perhaps the previous one) to run based on its deadline queue. As the number of parallel applications increases, each process of an application on an individual node will have a smaller chance of running uninterrupted throughout its slice. In addition, there will be a smaller chance of each process starting its slice at the same time.

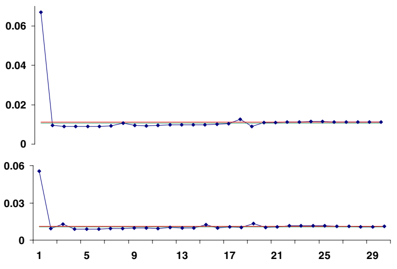
The upshot is that even though the process will continue to meet its deadlines locally, it will be less synchronized with processes running on other nodes. This results in the application's overall performance changing, causing the global controller to be invoked more often. Because the control loop frequency is less than the frequency of these small performance changes, the system begins to oscillate. However, the degradation is graceful, and, again, the long term averages are well behaved.

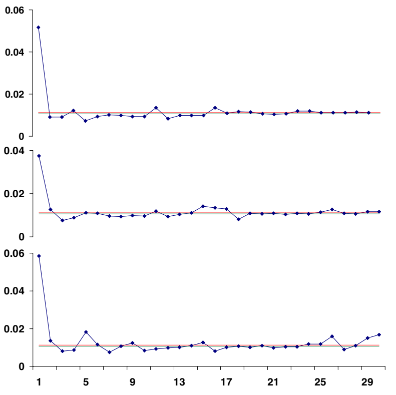
5.10 Effects of local disk I/O

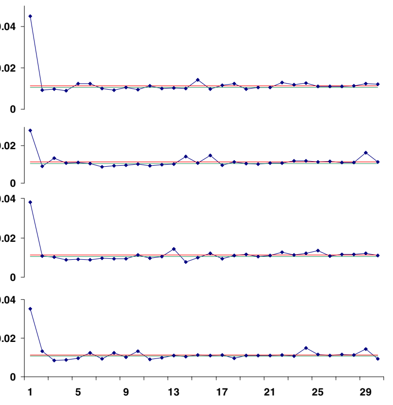
Although we are only scheduling the CPU resource, it is clear from the above that this is sufficient to isolate and control a BSP application with complex collective communications of significant volume. Is it sufficient to control such an application when it also extensively performs local disk I/O?

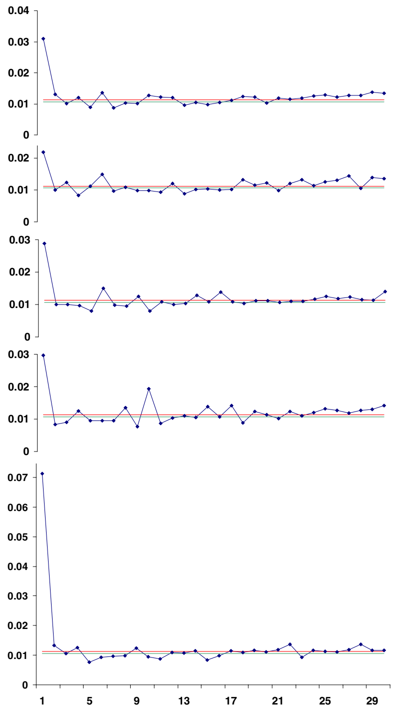
To study the effects of local disk I/O on our scheduling system, we modified the Patterns benchmark to perform varying amounts of local disk I/O. In the modified Patterns, each node writes some number of bytes sequentially to the local IDE hard disk during each iteration. It is ensured that the data is written to the physical disk by using `fsync()` call.

In our first set of experiments, we configured Patterns with a very high (145:1) compute/communicate ratio, and 0, 1, 5, 10, 20, 40, and 50 MB per node per iteration of local disk I/O. Our target execution rate was 50% with a threshold of 3%. Figure 13 shows the results for 10, 20, and 40 MB/node/iter. 0, 1, 5 are similar to 10, while 50 is similar to 40. For up to 10 MB/node/iter, our system effectively maintains control of the application's execution rate. As we exceed this limit, we develop a slight positive bias; the application runs faster than desired despite the restricted CPU utilization. The dominant part of the time spent on local disk I/O is spent waiting for the disk. As more I/O is done, a larger proportion of application execution time is outside of the control of our system. Since the control algorithm requires that the CPU utilization be equal to the target execution rate, the actual execution rate grows. In the second set of experiments, we fixed the local disk I/O to 10 MB/node/iter (the maximum controllable situation in the previous experiment) and varied the compute/communicate ratio, introducing different amounts of network I/O. We used a target rate of 50%. We used seven compute/communicate ratios ranging from 4900:1 to 1:3.5.


(a) 2 benchmarks


(b) 3 benchmarks


(c) 4 benchmarks


(d) 5 benchmarks

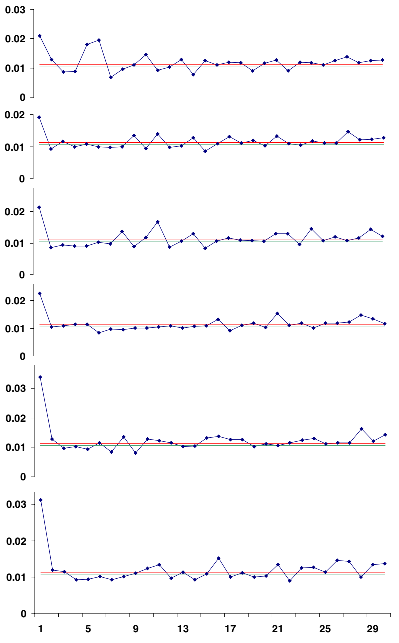

(e) 6 benchmarks

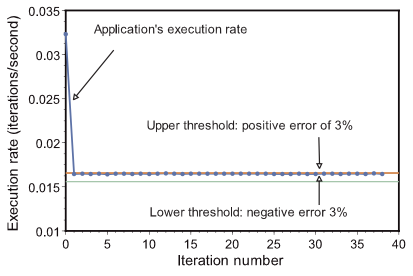
Figure 12. Running multiple Patterns benchmarks; 3:1 comp/comm ratio; 3% threshold.

Figure 14 shows the results for 4900:1, 2:1, and 1:3.5. For high to near 1:1 compute/communicate ratios, our system can effectively control the application's execution rate even with up to 10 MB/node/iteration of local I/O, and degrades gracefully after that.

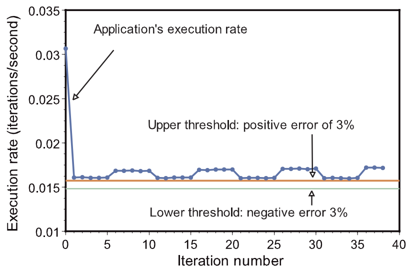
Our system can effectively control the execution rates of applications performing significant amounts of network and local disk I/O. The points at which control effectiveness begins to decline depends on the compute/communicate ratio and the amount of local disk I/O. With higher ratios, more local disk I/O is acceptable. We have demonstrated control of an application with a 1:1 ratio and 10 MB/node/iter of local disk I/O.

5.11 Effects of physical memory use

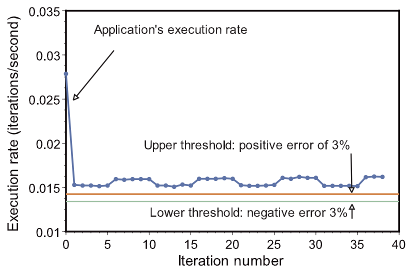
Our technique makes no attempt to isolate memory, but the underlying node OS certainly does so. Is it sufficient?



(a) 10 MB/node/iter I/O

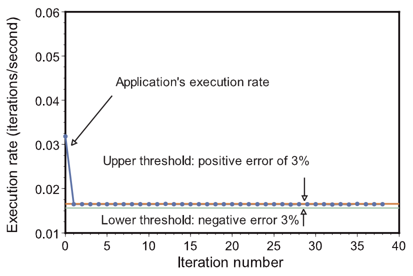


(b) 20 MB/node/iter I/O

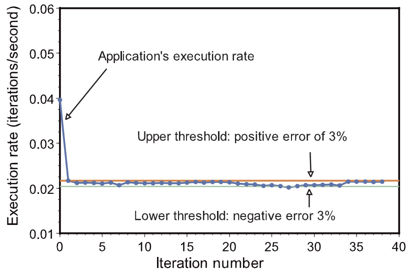


(c) 40 MB/node/iter I/O

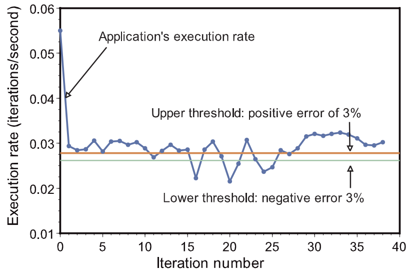
Figure 13. Performance of control system with a high (145:1) comp/comm ratio and varying local disk I/O.



(a) high (4900:1) comp/comm ratio



(b) medium (2:1) comp/comm ratio



(c) low (1:3.5) comp/comm ratio

Figure 14. Performance of control system with 10 MB/node/iter of disk I/O and varying comp/comm ratios.

To evaluate the effects of physical memory contention on our scheduling system, we modified the Patterns benchmark so that we could control its working set size. We then ran two instances of the modified benchmark simultaneously on the four nodes of our cluster. We configured the first instance with a working set of 600 MB and a target execution rate of 30%, while the second was configured with a working set size of 700 MB and a target rate of 40%. Both instances had a compute/communicate ratio of around 130:1. The combined working set of 1.3 GB is slightly less than the 1.5 GB of memory of our cluster nodes.

We used the control algorithm to schedule the two instances, and Figure 15 shows the results of this experiment. We see that despite the significant use of memory by both instances, our system maintains control of both applications' execution rates.

Our results suggest that unless the total working set on the machine is exceeded, physical memory use has little effect on the performance of our scheduling system. It is important to point out that most OS kernels, including Linux, have mechanisms to restrict the physical memory use of a process. These mechanisms can be used to guarantee that the physical memory pressure on the machine does not exceed the supply. A virtual machine monitor such as Xen or VMware provides additional control, enforcing a physical memory limit on a guest OS kernel and all of its processes.

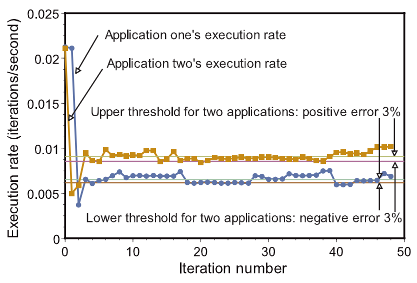


Figure 15. Running two Patterns benchmarks under the control system; high (130:1) comp/comm ratio. The combined working set size is slightly less than the physical memory.

6 Conclusions and future work

We have proposed, implemented, and evaluated a new self-adaptive approach to time-sharing parallel applications on tightly coupled compute resources such as clusters. Our technique, performance-targetted feedback-controlled real-time scheduling, is based on the combination of local scheduling using the periodic real-time model and a global feedback control system that sets the local schedules. The approach performance-isolates parallel applications and allows administrators to dynamically change the desired application execution rate while keeping actual CPU utilization automatically proportional to the application execution rate. Our implementation takes the form of a user-level scheduler for Linux and a centralized controller. Our evaluation shows the system to be stable with low response times. The thresholds needed to prevent control instability are quite reasonable. Despite only isolating and controlling the CPU, we find that memory, communication I/O, and local disk I/O follow.

We are now focusing on how to apply our approach to a wider range of workloads such as web applications that have more complex communication and synchronization behavior, and high-performance parallel scientific applications that have performance requirement which are typically not know a priori and change as the applications proceed [13]. In related work, we are considering how to exploit direct feedback from the end-user in a scheduling system.

References

[1] A. C. Arpaci-Dusseau, D. E. Culler, and A. Mainwaring. Scheduling with implicit information in distributed systems. In *ACM Sigmetrics*, 1998.

[2] J. Chen, G. Soundararajan, and C. Amza. Autonomic provisioning of databases in dynamic content web servers. In *Proceedings of the 3rd IEEE International Conference on Autonomic Computing*, 2006.

[3] H.-H. Chu and K. Narhstedt. CPU service classes for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, June 1999.

[4] P. A. Dinda and D. R. O'Hallaron. Realistic CPU workloads through host load trace playback. In *Proc. of 5th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR)*, May 2000.

[5] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, Massachusetts, 1994.

[6] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251–267, 1994.

[7] A. Gupta and P. A. Dinda. Inferring the topology and traffic load of parallel programs running in a virtual machine environment. In *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, June 2004.

[8] D. Ingram and S. Childs. The linux-srt integrated multimedia operating system: bringing qos to the desktop. In *Proceedings of the IEEE Real-time Technologies and Applications Symposium (RTAS)*, 2001.

[9] M. Jette. Performance characteristics of gang scheduling in multiprogrammed environments. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, pages 1–12, 1997.

[10] Y. K. K. Hyoudou and Y. Nakayama. An implementation of concurrent gang scheduler for pc cluster systems. In *Parallel and Distributed Computing and Networks*, 2004.

[11] B. Lin and P. Dinda. Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In *ACM/IEEE SC 2005 (Supercomputing)*, 2005.

[12] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[13] H. Liu and M. Parashar. Enabling self-management of component based high-performance scientific applications. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing*, 2005.

[14] X. Liu, X. Zhu, S. Singhal, and M. Arlitt. Adaptive entitlement control of resource containers on shared servers. In *Proceedings of the IFIP/IEEE International Symposium on integrated Network Management*, 2005.

[15] C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. H. Son, and M. Marley. Performance specifications and metrics for adaptive real-time systems. In *Proceedings of 21st IEEE Real-Time Systems Symposium*, 2000.

[16] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Special issue of Real-Time Systems Journal on Control-Theoretic Approaches to Real-Time Computing*, 23(12):85–126, September 2002.

[17] C. Lu, X. Wang, and X. Koutsoukos. Feedback utilization control in distributed real-time systems with end-to-end tasks. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):550–561, 2005.

[18] J. Nieh and M. Lam. The design, implementation, and evaluation of SMART: A scheduler for multimedia applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[19] J. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of ICDCS*, 1982.

[20] C. L. Scott A. Brandt, Scott Banachowski and T. Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *Proceedings of IEEE Real-Time Systems Symposium*, 2003.

[21] J. A. Stankovic, T. He, T. F. Abdelzaher, M. Marley, G. Tao, S. H. Son, and C. Lu. Feedback control scheduling in distributed real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium*, 2001.

[22] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, 1999.

[23] P. Strazdins and J. Uhlmann. A comparison of local and gang scheduling on a beowulf cluster. In *Proceedings of the 2004 IEEE International Conference of Cluster Computing*, pages 55–62, 2004.

[24] J. Subhlok, T. Gross, and T. Suzuoka. Impact of job mix on optimizations for space sharing schedulers. In *Proceedings of Supercomputing '96*, November 1996.

[25] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), August 1990.

[26] S. White, A. Alund, and V. S. Sunderam. Performance of the NAS parallel benchmarks on PVM-Based networks. *Journal of Parallel and Distributed Computing*, 26(1):61–71, 1995.

[27] W. Xu, X. Zhu, S. Singhal, and Z. Wang. Predictive control for dynamic resource allocation in enterprise data centers. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium*, 2006.